Compression Planner for Time Series Database with GPU Support

Piotr Przymus[1(✉)] and Krzysztof Kaczmarski[2]

[1] Nicolaus Copernicus University, Toruń, Poland
eror@mat.umk.pl
[2] Warsaw University of Technology, Warsaw, Poland
k.kaczmarski@mini.pw.edu.pl

Abstract. Nowadays, we can observe increasing interest in processing and exploration of time series. Growing volumes of data and needs of efficient processing pushed research in new directions. This paper presents a lossless lightweight compression planner intended to be used in a time series database system. We propose a novel compression method which is ultra fast and tries to find the best possible compression ratio by composing several lightweight algorithms tuned dynamically for incoming data. The preliminary results are promising and open new horizons for data intensive monitoring and analytic systems.

Keywords: Time series database · Lightweight compression · Lossless compression · GPU · CUDA · GPGPU · Compression optimization

1 Introduction

Background – Time Series Databases. Specialized time series databases play important role in industry storing monitoring data for analytical purposes. These systems are expected to process and store millions of data points per minute, 24 h a day, seven days a week, reading terabytes of logs. Due to regression errors checking and early malfunction prediction these data must be kept with fine grained resolution including all details. Solutions like OpenTSDB [17], TempoDB [4] and others deal very well with these kind of tasks. Most of them work on a clone of Big Table approach from Google [8], a distributed hash table with mutual ability to write and read data in the same time.

Querying large volumes of time series may be time consuming and even in case of big clusters leads to system slowdown. On the other hand monitoring of any infrastructure requires real-time or near real-time response. What is more

P. Przymus: The project was partially funded by Marshall of Kuyavian-Pomeranian Voivodeship in Poland with the funds from European Social Fund (EFS) in the form of a PhD scholarships. "Krok w przyszłość – stypendia dla doktorantów V edycja" (Step in the future – PhD scholarships V edition).
K. Kaczmarski: The project was partially funded by National Science Centre, decision DEC-2012/07/D/ST6/02483.

A. Hameurlain et al. (Eds.): TLDKS XV, LNCS 8920, pp. 36–63, 2014.
DOI: 10.1007/978-3-662-45761-0_2

important it is hard to predict a priori what kind of queries may be needed. Various problems may be only investigated by checking all possible correlations. Therefore a system must perform random queries on large data sets. Classical databases even using large computational clusters and map reduce approach can hardly fulfil this requirement since time series processing not only requires large volumes of data but also has computational demands: interpolation, integration and aggregation of millions of time series.

The above problems may be easily handled by a database system equipped with a GPU device used as a coprocessor [7]. An average internet service with about 10 thousands of simultaneously working users may generate around 80 GB of logs every day. If we consider an in-memory database system these data after a compression could fit into two NVIDIA Tesla devices and an average query may be processed within milliseconds compared to seconds or minutes in case of standard systems.

GPU device has its own memory or a separate area in the CPU main memory. CPU and GPU may only cooperate in a *shared nothing architecture*.Thus, the data has to be explicitly transferred from the CPU main memory to the GPU main memory and then back to CPU. Additional data transfer in the pipeline of a query processing often introduces significant overhead which cannot be mitigated. This cost is therefore an important component of the query execution time prediction.

Time Series Compression. Big Table based systems compress data before writing to a long-term storage. It is much more efficient to store data for some time in memory or in a disk buffer and compress it before flushing to disk. This process is known as a *table row rolling*. Systems like HBase [1], Casandra [10] and others offer compression for entire column family. This kind of general purpose compression is not optimized for particular data being stored (i.e. various time series with different compression potential stored in one column family). Similarly in-memory database systems based on GPU processing (like ParStream [3]) tend to pack as many data into GPU devices global memory as possible.

Compression not only improves overall system behaviour by optimizing data transfer but also enables GPU co-processing by minimization of additional data transfer costs. Figure 1 shows the influence of lightweight compression on query processing time including input data reading, processing and creating output. The bar on the left (CPU) presents the basic query processing pipeline with three activities: reading, processing and creating output. (GPU) bar shows processing time on GPU processing which additionally requires some time for data copying. Processing time is much shorter but additional copying makes overall speed-up not so impressive. The next column presents the same configuration but with lightweight compression of the data. Now copying time is much shorter. Thanks to GPU abilities data decompression time is not influencing the overall time noticeably. In the contrary the same approach but run purely on CPU suffers from long decompression time (the last column).

Our previous work on time series compression problems [19–21] and GPU utilization in time series processing showed that GPU may be successfully introduced

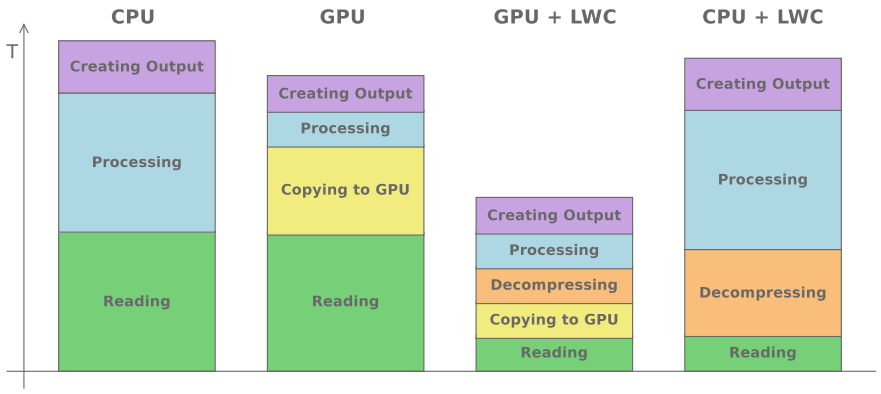

Fig. 1. General query processing time influenced by lightweight compression and GPU processing.

as a database coprocessor if accompanied by a lightweight compression of the stored data. We created a dynamic compression planner which was able to combine several lightweight compression methods in order to achieve the best compression results. However, the optimal compression may not always be acceptable due to possibly long decompression time.

In this work we extend the previous findings by a new multi-objective compression planner which may find a near-optimal[1] multi objective compression plan.

Section 2 presents a general view of the system including time series data model and data flow. Section 3 describes used lightweight compression methods which are suitable for fast GPU processing. Sections 4, 5 and 6 contain the main contribution of our work: the dynamically optimized compression system including plans estimation and bi-objective plan selection. Experimental runtime results are contained in Sect. 7 while Sect. 8 contains final remarks and conclusions.

1.1 Motivation and Related Work

Compression of time series is an interesting and widely analysed computational problem. Lossless methods often use some general purpose compression algorithms with several modifications according to knowledge gathered from data. On the other hand, lossy compression approximate data using, for instance, splines, piecewise linear approximation or extrema extraction [14]. For industrial monitoring systems, lossy compression cannot be used due to possible degradation of anomalies.

An important challenge is to improve compression factor with an acceptable processing time in case of variable sampling periods. Interesting results in the

[1] In this work we understand optimal compression as the best compression within available lightweight algorithms.

filed of lossless compression done on GPU were presented by Fang et al. [13]. Using a compression planner it was possible to achieve significant improvement in overall query processing on GPU by reducing data transfer time from RAM to global device's memory space. The strategy applied in our work is based on statistics calculated from inserted data and used to find an optimal cascaded compression plan for the selected lightweight methods.

The GPU compression topic was raised in several studies. Interesting results on GPU compression where presented by Andrzejewski et al. [5] where Word Aligned Hybrid compression algorithm for GPU was presented. Wu et al. [24] discussed implementation of Lempel-Ziv 77 (LZ77) algorithm on CUDA framework and showed that time complexity of this algorithm was to high on GPU processor when compared to CPU classical implementation. This was caused by too many branches in the algorithm which are not suited well for CUDA model of parallelism.

In a time series database we often observe data grouped into portions of very different characteristics. Optimal compression should be able to apply different compression plans for different time series and different time periods.

In case of lossless compression one can use common algorithms (ZIP, LZO) which tend to consume lot of computation resources [6,26] or lightweight methods which are faster but not so effective. Dynamic composition of several compression methods may improve this significantly by combination of properties of both approaches: it is lossless but much faster than common algorithms, offers acceptable compression ratios and may be computed incrementally. Selection of an optimal strategy (among available lightweight compression algorithms) is done upon data statistical information.

However, the challenge of multiprocessor and multi-GPU computational nodes raise another question: is it possible to improve these methods further including hardware specific information and estimated decompression time? In this work we extend our previous findings by a new multi-objective compression planner which may find an optimal compression plan under compression ratio and decompression speed optimization objectives. A database system will be able to benefit by better estimation of time constraints for query execution.

2 Time Series Database System with Compressed Storage

A typical time series database consists of three layers: data insertion module, data storage and querying engine. This section presents a general view of a prototype heterogeneous time series database system developed as a test-bed for our compression algorithms and optimization methods. It uses GPU as a coprocessor for database operations and data compression. Optimal resources (GPU and CPU) utilization requires a heterogeneous query planner which is addressed in another paper [22]. Here, we shall only focus on assuring optimal compression for this system.

2.1 Time Series Data Model

The data acquisition from ongoing measurements, industrial processes monitoring [15], scientific experiments [23], stock quotes or any other financial and business intelligence sources has got continuous characteristic. These discrete observations T are represented by pairs of a *timestamp* and a *numerical value* (t_i, v_i) with the following assumptions:

- number of data points (timestamps and their values) in one time series should not be limited;
- each time series should be identified by a name which is often called a *metric name*;
- each time series can be additionally marked with a set of *tags* describing measurement details which together with metric name uniquely identifies time series;
- observations may not be done in constant time intervals or some points may be missing, which is probable in case of many real life data (Fig. 2).

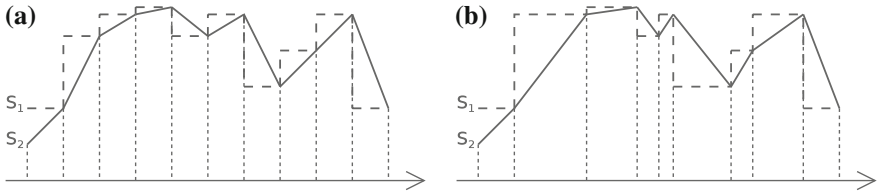

Fig. 2. Time series. (a) fixed time measurements (b) variable time measurements. Characteristics of the plot (s_1 – piecewise constant, s_2 – piecewise linear) depends on the interpretation of the measured data value.

The last assumption is important since industrial applications often cannot guarantee either constant measurement period or correct measurement and data transfer.

Our prototype system does not limit possible data which can be inserted and analysed. The only requirement of our system is that data must have a form of *time series*, which we understand as a collection of observations made sequentially in time [9].

In the presented data model known from for example OpenTSDB [2] one time series is uniquely identified with a metric name and a set of tags. Combination of tags let a user express many different queries in a very simple way. For instance for the input data (timestamp, metric name, value, tags):

```
1386806400 cpu.load 0.20 node=alpha type=system
1386806400 cpu.load 0.10 node=alpha type=user
1386806401 cpu.load 0.30 node=alpha type=system
```

```
1386806401 cpu.load 0.20 node=alpha type=user
1386806400 cpu.load 0.05 node=beta type=system
1386806400 cpu.load 0.10 node=beta type=user
1386806401 cpu.load 0.05 node=beta type=system
1386806401 cpu.load 0.40 node=beta type=user
```

we could issue a query for an overall average *system* type processes processor load for all known nodes for one day by:

```
q?start=2013-12-12:00:00&
  end=2013-12-12:23:59&m=avg:cpu.load{type=system}
```

receiving:

```
1386806400 cpu.load 0.135 node=alpha type=system
1386806401 cpu.load 0.165 node=alpha type=system
```

or for a maximum processor load among all known nodes and processes types:

```
q?start=2013-12-12:00:00&
  end=2013-12-12:23:59&m=max:cpu.load
```

receiving:

```
1386806400 cpu.load 0.20
1386806401 cpu.load 0.40
```

2.2 Data Insertion

In this section we briefly describe data flow in our system, which is composed of three layers: data insertion, long term storage and data retrieval.

The insertion layer is preceded by a set of collectors which gather data from sensors, probes or other sources. These collectors sending data to the data insertion interface are considered external and beyond the scope of this paper.

The general view of our system's basic components and data flow is not different from other databases. The noticeable extension includes GPU and CPU processing. The main data flow indicates: data collection, data buffering, data storing and data querying. Data buffering may use compression on GPU or CPU side. Obviously, CPU compression usually does not require any additional data transfers since Data Buffer and CPU compression may be done by the same device and within the same memory space. However, compression performed by GPU requires extra time for data transferring between RAM and GPU device's global memory. After the data is compressed it may be sent to the database daemon which inserts them into a long term storage.

Similar situation occurs during data retrieval and query evaluation. Fragments of data required for particular query need to be decompressed, filtered and transformed according to the query parameters. All these operations may be done within CPU or with GPU used as an external coprocessor.

Figure 3 indicated the possible transitions and data flow between components in our database system. Each transition may have non zero data transfer time. Each node may transform data changing their size but also consuming time. A sample data insertion procedure could involve the following path: Data Collector, Data Buffer, GPU Compression, Data Buffer, Database Daemon, Storage. Data retrieval contains more nodes, transitions and possible paths. For example, a path of a query evaluated on CPU but with data decompressed on GPU will contain the sequence: Storage → Query Engine → GPU Decompression → Query Engine → CPU Query Processing → Query Engine → Client. Selection of an optimal query plan must involve distributed data processing on heterogeneous devices which we addressed in [22].

Minimisation of data transfer time in a heterogeneous database system is the main driver for the research presented in this paper. Our approach focuses on finding the best possible compression method suited for certain incoming data but from two points of view: compression ratio and decompression time. Both these goals are crucial for current time series database systems.

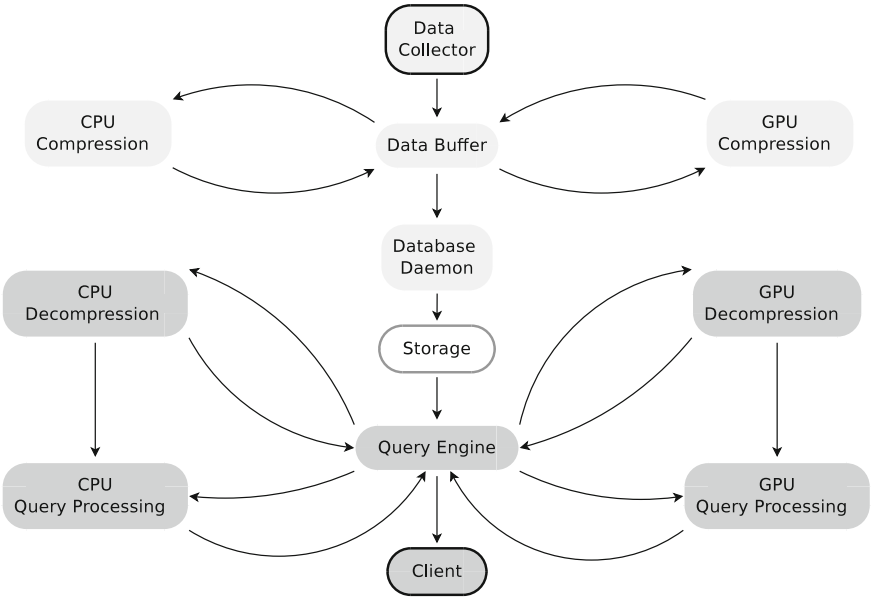

Fig. 3. A simplified data flow diagram including data insertion (light grey), data retrieval (dark grey) and storage (white) layers. Each transition may introduce additional data transfer cost. Each node may transform data influencing their size.

Time Series Storage. One of the most important properties of a time series database system is high performance and scalability. In many industrial solutions these assumptions lead to an architecture based on *Big Table* [8] and Map Reduce [11] applications.In such case time series data are stored at two different levels in the database:

- Data buffer – acts as a buffer for new data which enter the database. Data from the data buffer are periodically compacted, compressed and sent to the data archive. The buffer is composed of two separated data blocks: first one with timestamps and second one with values. This allows us to use different compression plans for both blocks.
- Permanent Storage – works as a long term data archive based on key-value table architecture. Metric name, tags and starting time are encoded in row key and column key.

Flow of Input Data. The data acquisition from ongoing measurements, industrial processes monitoring [15], scientific experiments [23], stock quotes or any other financial and business intelligence sources has got continuous characteristics. We assume that data collectors keep sending data to the system all the time and the system must respond in the real-time. Tight efficiency constraints must be met in order to assure that the data will not wait before being consumed for unacceptably long time.

Due to optimization purposes, data sent to the data storage should be ordered and buffered into portions, minimizing necessary disk operations but also minimizing the distributed storage nodes intercommunication. Buffering also prepares data to be compressed and stored optimally in an archive. Simplicity of data model imposed separated column families for compressed and raw data. Time series are separately compacted into larger records (by a metric name and tags) containing a specified period of time (e.g. 15 min, 2 h, 24 h – depending on the number of observations). This step directly preceded dynamic compression described in the next sections.

Finally, when a single record in a buffer is compressed and ready to be send to the long term permanent storage it is flushed and delivered to a NoSQL database which processes it according to its internal rules.

2.3 Data Retrieval

The last important part of the system is the query engine responsible for user-database interactions.

Execution of database queries is an example of a successful application of GPU co-processors which may accelerate numerous database computations, e.g. relational query processing, query optimization, database compression or supporting time series databases [7,20,21].

Distribution of workload between numerous CPU and GPU devices require careful planning of query execution strategy including not only data transfer costs but also device load, its efficiency or even energy consumption. In our previous publication we elaborated on bi-objective query planner which achieved interesting results in case of a heterogeneous query planning [22].

For the purposes of this work we indicate that the influence of compression methods used in a data storage on query evaluation is twofold. First, it may dramatically reduce data transfer time between system components and second, it may increase query evaluation time by additional decompression.

2.4 Searching for Optimal Compression

In any database system the data transfer time between distributed nodes or components significantly influences the overall performance of the system. This situation may be partially improved by compression but only if its additional cost is justified by gained speed up. Fine tuned lightweight compression methods offer interesting compression ratio with acceptable performance, especially if used on a GPU device [20,21].

In order to select an optimal compression method one must consider the following factors:

- Predicted compressed buffer size
- Predicted compression and decompression time
- Computational resources needed
- Additional method's properties.

Achieving best possible compression ratio and shortest possible working time are two contradicting objectives. Thus, a definition of an optimum solution set should be established. In this paper we use the predominant Pareto optimality [16].Given a set of choices and a way of valuing them, the Pareto set consists of choices that are Pareto efficient. A set of choices is said to be Pareto efficient if we cannot find a reallocation of those choices such that the value of a single choice is improved without worsening values of others choices.As bi-objective optimization is NP-hard, we need an approximate solution [18].

Our compression planner computes the best compression scheme upon all available algorithms, knowing their properties and input data characteristics.

3 Lightweight Compression Algorithms

In this section we present the compression algorithms and their modifications for the parallel execution on a GPU. Detailed description of presented compression algorithms may be found in [12,13,20,26].

Patched Lightweight Compression. The main drawback of many lightweight compression schemes is that they are prone to outliers in the data frame. For example, consider following data frame $\{1, 2, 3, 2, 2, 3, 1, 1, 64, 2, 3, 1, 1\}$, one could use the 2 bits fixed-length compression to encode the frame, but due to the outlier (value 64) we have to use 6-bit fixed-length compression or more computationally intensive 4-bit dictionary compression. Solution to the problem of outliers has been proposed in [26] as a modification to three lightweight compression algorithms. The main idea was to store outliers as exceptions. Compressed block consists of two sections: the first keeps the compressed data and the second exceptions. Unused space for exceptions in the first section is used to hold the offset of the following exceptions in the data in order to create linked list, when there is no space to store the offset of the next exception, a *compulsive exception* is created [26]. For large blocks of data, the linked lists approach may fail because the exceptions may appear sparse thus generate a large number

of compulsory exceptions. To minimise the problem various solutions have been proposed, such as reducing the frame size [26] or algorithms that do not generate compulsive exceptions [12,25]. The algorithms in this paper are based largely on those described by Yan [25]. In this version of the compression block is extended by two additional arrays - exceptions position and exceptions remainders values (i.e. the remaining bits). Decompression involves extracting data using the underlying decompression algorithm and then applying a patch (from exceptions remainders array) in the places specified by the exceptions positions. As exceptions are separated, data patching can be done in parallel. During compression, each thread manages two arrays for storing exception values and positions. After compression, each thread stores exceptions in the shared memory, similarly exceptions from shared memory are copied to the global memory. Patched version of algorithms are only selected if compression ratio improves. Otherwise non patched algorithms are used. Therefore complex exceptions treatment may be omitted speeding up the final compression.

Float to integer scaling (SCALE). Converts float values to integer values by scaling. This solution can be used in case where values are stored with given precision. For example, CPU temperature 56.99 can be written as 5699. The scaling factor is stored in compression header.

Differential representation (DELTA). Stores the differences between successive data points in frame while the first value is stored in the compression header. Works well in case of sorted data, such as measurement times. For example, let us assume that every 5 min the CPU temperature is measured starting from 1367503614 to 1367506614 (Unix epoch timestamp notation), then this time range may be written as $\{300, \ldots, 300\}$.

(Patched) Fixed-length Minimum Bit Encoding (PFL and FL). FL and PFL compression works by encoding each element in the input with the same number of bits thus deleting leading zeros at the most significant bits in the bit representation. The number of bits required for the encoding is stored in the compression header. The main advantage of the FL algorithm (and its variants) is the fact that compression and decompression are highly effective on GPU because these routines contain no branching-conditions, which decrease parallelism of SIMD operations. For the best efficiency dedicated compression and decompression routines are prepared for every bit encoding length with unrolled loops and using only shift and mask operations.Our implementation does not limit minimum encoding length to size of byte (as in [13]). Instead each thread (de)compresses block of eight values, thus allowing encoding with smaller number of bits. For example, consider following data frame $\{1, 2, 3, 2, 2, 3, 1, 2, 3, 1, 1\}$, one could use the 2 bits fixed-length compression to encode the frame.

(Patched) Frame-Of-Reference (PFOR and FOR). Works similarly to FL and PFL, except before compression it transforms each value into an offset from the reference value (for example smallest value) in compression block. Reference value is then stored in compression header. In this situation, we need exactly $\lceil \log_2(\max - \min + 1) \rceil$ bits to encode each value in the frame.

For example, this is useful when storing measurement times, consider time range $\{1367503614, \ldots, 1367506614\}$, then using for we only need $\lceil \log_2(1367506614 - 1367503614 + 1) = 12 \rceil$ bits to store each value in this range (as opposed to 31 bits without this transformation).

(Patched) Dictionary (DICT and PDICT). DICT is suitable for data that have only a small number of distinct values. It uses a dictionary of distinct values. For compression and decompression purposes, dictionary is loaded into the shared memory. Binary search is used during compression to lookup values, then an index of value is used to encode. Decompression simply retrieves values at given index from dictionary. DICT writes indexes using byte-aligned types, for better compression a combination with other compression algorithm should be used. For example, consider data frame {0, 500, 1500, 100, 100, 1500000, 100, 15000} using DICT only 1 byte is needed to store each value (even less if combined with other compression algorithm) in comparison to pure FL where more than 2 bytes would have been used.

Run-Length-Encoding (RLE) and Patched Constant (PCONST). RLE encodes values with a pair: value and run length, thus using two arrays to compress data. Consider following data frame $\{1, 1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3\}$, then RLE would create two arrays: values $\{1, 2, 3\}$ and run length $\{5, 4, 3\}$. PCONST is a specialized version of RLE where almost whole data frame consist of one value with some exceptions. This may be reconstructed using: frame length, constant value and PATCH arrays. For example, let us assume that a measurement is done every five minutes with some exceptions, then delta is almost always constant and equals 300, any other value will be stored as exception.

4 Cascaded Compression Planner

The goal of this part of the system is to find suitable cascaded compression plans for the data gathered in the input buffer. It is composed of three parts:

- selection of suitable cascaded compression plans – mainly based on the specifics of the algorithms and the characteristics of the data set (see rest of this Section).
- evaluation of selected cascaded compression plans – based on the dynamic statistics generator and compressed data size estimation (see Sect. 5).
- bi-objective plan selector – which uses decompression time estimation and compressed data size to choose the final one (see Sect. 6).

Cascaded compression can significantly improve the compression ratio. However, searching for the most efficient compression method even for relatively short plans composed for several compression steps (i.e. using 6 compositions out of 10 algorithms with repetitions) may generate a very large search space (in our example $\sum_{i=1}^{6} 10^i = 1,111,110$). Due to tight time constraints we proposed a reduction of this problem by static planner and hints system.

Note that in fact, the situation is even more complicated, because of possible compression algorithms parametrization, e.g. (P)FL and (P)FOR take as an argument number of j bits used to encode each value, where $1 \leq j \leq 32$. This topic will be discussed with details in Sect. 5.

4.1 Reduction of Compression Plans Search Space: Static Planner

In the first static stage we determined acceptable transitions between compression algorithms which were divided into three categories: $\mathcal{T}$ – initial transformation, $\mathcal{B}$ – base compression, $\mathcal{H}$ – helper compression. The complete compression schema is always composed of algorithms selected from these subsequent categories $\bar{P} \subseteq P = \{(t, b, h) : t \in \mathcal{T}, b \in \mathcal{B}, h \in \mathcal{H}\}$, with the following purposes:

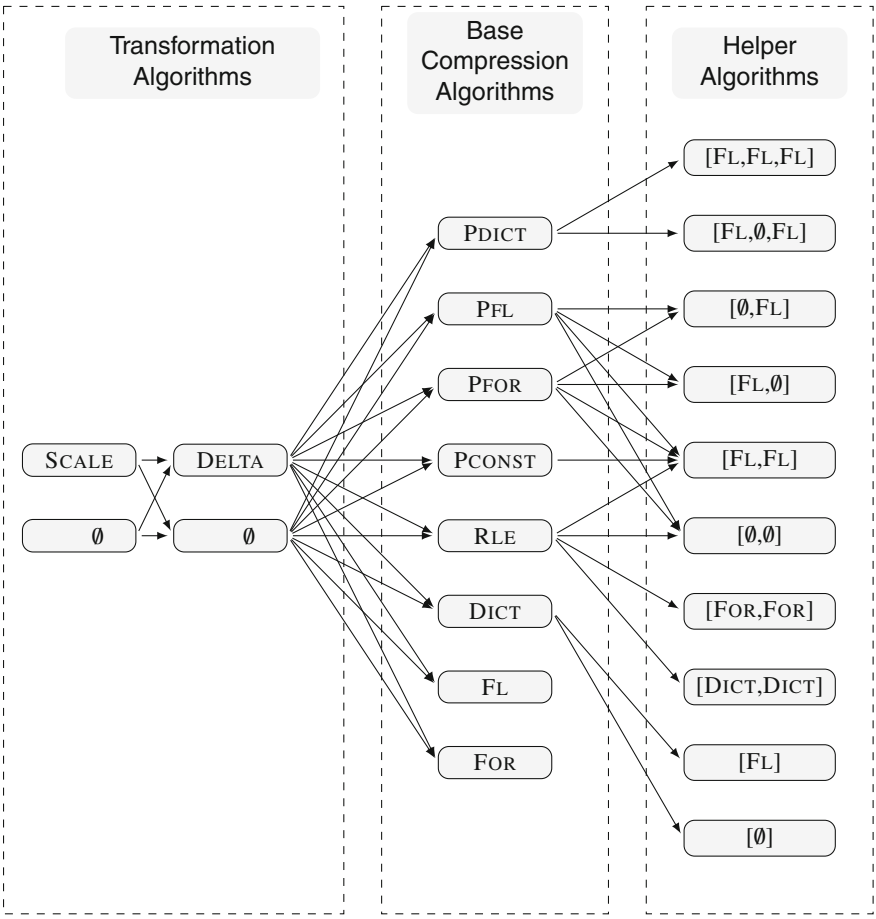

Fig. 4. The composition graph of all available compression plans within the given assumptions. Helper auxiliary algorithms are applied to additional arrays (from one to three) returned by the base compression algorithms.

1. $\mathcal{T}$ – **Transformation algorithms (Scale, Delta).** Improve properties of data storage and prepare for better compression. All algorithms in this section are optional but may be used together (if present must be applied in the given order).
2. $\mathcal{B}$ – **Base compression algorithms (Pdict, Pfl, For, Fl, Dict, Pfor, Rle, Pconst).** Only one algorithm may be selected as the base algorithm. All algorithms in this section generate from one up to three resulting arrays. Some of the resulting arrays, may qualify for further compression using *Helper compression algorithms*.
3. $\mathcal{H}$ – **Helper compression algorithms (For, Fl, Dict).** The algorithms used to compress selected arrays from the previous step. Each of the resulting arrays can be compressed with only one algorithm. In order to minimize the stages of decompression PATCH algorithms, which could create new arrays for compression, are excluded. The base algorithm used may limit algorithms in this section (Fig. 4).

Composition of all sensible paths between algorithms in these three categories leaves only 76 suitable compression plans out of former one million. The longest possible cascaded compression plan may be composed of six steps.

4.2 Manual Tuning of Compression Plans Search Space: Hints System

Another reduction of possible compression plans generated in the first stage can be done manually by a user speeding up further plan choosing. Number and types of hints may vary in different situations. For example, in time series systems timestamps are always sorted and if we consider separated compression methods for timestamps and values we may find different and better plans for them. A hint indicating sorted input may suggest using DELTA before base algorithms. Additionally, for every metric additional features may be specified or even specific compression algorithm may be enforced. Currently supported

Table 1. A sample set of hints for a time series compression planner.

Hints	Meaning
Scale, (P)Fl, Rle, Delta, (P)For, (P)Dict, Pconst	Enforces a specific compression algorithm in the plan
DSORTED	Specify whether the data is distinct and sorted. If true eliminates following algorithms from compression plan: Pconst, Rle, (P)Dict
TIMESTAMP	Automatically added by system to timestamps. Sets DSORTED to True and Scale to False
DATA	Automatically added by the system to time series values. If not specified otherwise it sets DSORTED to False and Scale to False

hints are located in Table 1. After this step we select a subset $P' = P'(D) \subseteq \bar{P}$ of compression plans, where D – is data set to be compressed, and $P'(D)$ is a reduced subset of $\bar{P}$ after using hints elimination.

5 Dynamic Statistics Generator

5.1 Finding the Optimal Parametrization and Compression Size Estimation

In this step, a plan with possibly maximal compression ratio is selected. In order to perform this task the system uses statistics and estimations computed dynamically from incoming data stream, for each metric and rolled time period separately. Pre-computing them and storing aside is not an optimal solution due to necessity of constant update and allocation of additional memory. Please note that if a plan contains an initial transformation algorithm it must be applied before calculating statistics because it influences data (Fig. 5).

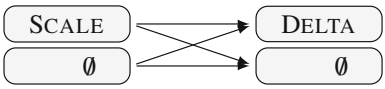

Fig. 5. Possible transformation algorithms and their composition.

Estimation results heavily depend on compression algorithms parameters. In [13] the choice of optimal parameters was straightforward, because the used algorithms supported only compression of value to byte-aligned size (which reduced number of parameters) and did not allow exceptions in data (only one set of parameters was correct). However, in compression algorithms and compression plans which use patching mechanism, optimal parameter selection is more complex. Factors such as the number of generated exceptions and estimated exception compression size should be taken into account. For example, the following data frame $\{1, 2, 3, 2, 32, 3, 3, 1, 64, 2, 1, 1\}$ could be compressed using PFL algorithm using 2 bits, 5 bits or 6 bits fixed-length, generating two exceptions (32, 64), one exception (64) or no exceptions respectively. In this case, for each compression plan (selected in previous stages) a satisfactory set of parameters should be selected in order to correctly estimate compressed data size.

Recall that $P'(D)$ is a reduced subset of P after using static planner and hints elimination. Let $P'(D) \ni p = (t, b, h)$ be a cascaded compression plan, where $t \in \mathcal{T}, b \in \mathcal{B}, h \in \mathcal{H}$ are transformation, base and helper algorithms, respectively. A compression plan may be also written in simplified notation, i.e. $\big($(SCALE, DELTA), PFL,(FL,FL)$\big)$.

Let us denote the data after applying the transformation algorithms by $t(D)$ where $t \in \mathcal{T}$ transforms data D. Now, let $J(p, D)$ estimate compressed size of data D after applying compression plan p, i.e.

$$J(p, D) := J_b(h, t(D)) \tag{1}$$

where $\bar{P} \ni p = (t, b, h)$ is a compression plan from subset of compression plans suitable for data D and J_b is estimation function for base algorithm b (see rest of this Sections for details). A pseudo code of an optimal compression plan selection is presented in listing SelectOptimalCompressionPlan. The rest of this section presents functions J_b for $b \in \{(P)\text{FL}, \text{RLE}, \text{DELTA}, (P)\text{FOR}, (P)\text{DICT}, \text{PCONST}\}$. This description is rather technical and not interested reader may skip to Sect. 6.

Procedure. SelectOptimalCompressionPlan(D)

```
   Input: D
   Result: Plan
 1 P' = P'(D) ;  /* select reduced subset of P after using static planner
   and hints elimination */
 2 min_size = size of data D;
 3 p* = ∅;
 4 for (t, b, h) = p in P'(D) do
 5 |   D' = t(D);
 6 |   if J_b(h, D') > min_size then
 7 |   |   min_size = J_b(h, D');
 8 |   |   p* = p;
 9 |   end
10 end
11 return p*, p* optimal parameters;
```

Table 2. Symbols used in the definition of parametrization optimization

Symbol	Description
D	Dataset
$\#D$	Dataset length
$min(D)$	$\min_{d \in D} d$
$t_{sub}(m, D)$	Subtract m from each $d \in D$
B_{dict}	Number of bits of type used to encode dictionary keys
B_{base}	Number of bits of base type (i.e. 8 bits, 16 bits, 32 bits, 64 bits)
$b_{index}(D)$	The minimum number of bits required to store any number between 0 and $\#D$
$b_{min}D)$	The minimum number of bits required to store any value $d \in D$
$c_{in}(j, D)$	Compressed data size (in bits) of patch index when using j bits FL coding to compress data D
$c_{re}(j, D)$	Compressed data size (in bits) of remainders values when using j bits FL coding to compress data D

5.2 Notation

Most of the notation is gathered in Table 2. For example, B_{dict} represents number of bits of type that is used to encode dictionary keys and B_{base} number of bits of base type (i.e. 8 bits, 16 bits, 32 bits, 64 bits). Please note that in order to simplify the following formulas notation we deliberately omitted the fact that the resulting data should be aligned to a byte.

5.3 Optimal Parametrization and Compression Size Estimation for (P)FL and (P)FOR

For algorithms (P)FL and (P)FOR it is crucial to determine the optimal number of bits needed to compress data D. To estimate size of data compressed with (P)FL and (P)FOR algorithms we use *Bit histogram* statistic. Let us define *Bit histogram* as $s_{bit}(j, D) = \#\{d \in D \colon j \text{ bits are sufficient to write } d\}$ for $1 \leq j \leq 32$ l. It is implemented on GPU using double buffering (registers and shared memory) parallel histogram scheme (Fig. 6).

Now, let b_{min} be the minimum number of bits required to store any value $d \in D$ and let $b_{index}(D)$ be the minimum number of bits required to store any number between 0 and $\#D$, i.e.

$$b_{min}(D) := \max_{1 \leq j \leq 32} \{j \colon s_{bit}(j, D) \neq 0\}, \tag{2}$$

$$b_{index}(D) := \lceil \log_2 \#D \rceil. \tag{3}$$

Consider a compression plan $p = (t, \text{FL}, h) \in P'$, i.e. a plan, where base compression algorithm b is set to FL. Since FL algorithm uses exactly $b_{min}(D)$ bits to compress each value in D, thus estimated compression size equals to

$$J_{FL}(h, D) := b_{min}(D) \cdot \#D, \tag{4}$$

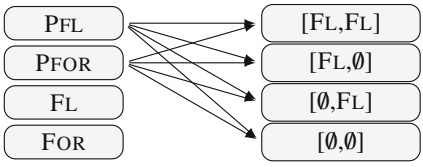

Fig. 6. PFOR, FOR, PFL and FL base algorithms and a possible composition with auxiliary algorithms.

(in case of compression plans with $b = \text{FL}$, helper algorithms are set to $h = \varnothing$). Clearly,

$$j_{FL} := b_{min}(D) \tag{5}$$

is the optimal parameter for FL.

Let now $p = (t, \text{FOR}, h) \in P'$. The estimation differs from the previous case, as now a transformation must be applied to data D. Let us define

$$t_{sub}(m, D) := \{d_i - m \colon d_i \in D\}, \tag{6}$$

which subtracts the value m from all values $d \in D$. To estimate the size of data after applying a compression plan $p = (t, \text{FOR}, h)$, we use

$$J_{FOR}(h, D) := J_{FL}(h, t_{sub}(min(D), D)), \tag{7}$$

i.e. the reference value here is $min(D)$ (in case of compression plans with $b =$ FOR, helper algorithms are again set to $h = \varnothing$). Similarly to the previous case

$$b_{min}(t_{sub}(min(D), D)) \tag{8}$$

is the optimal parameter for FOR.

Before we consider the cases of patching algorithms in compression plans, we need to define helper functions c_{in} and c_{re}:

$$c_{in}(h_{in}, D) := \begin{cases} b_{index}(D) & \text{if } h_{in} = FL, \\ B_{base} & \text{otherwise,} \end{cases} \tag{9}$$

$$c_{re}(j, h_{re}, D) := \begin{cases} (b_{min}(D) - j) & \text{if } h_{re} = FL, \\ B_{base} & \text{otherwise.} \end{cases} \tag{10}$$

The returned values depend on the value of $(h_{in}, h_{re}) = h \in \{\text{FL}, \emptyset\}^2$, which indicates whether a helper compression algorithm is used or not. If $h_{in} = \text{FL}$, then c_{in} returns the number of bits needed to store each element in position array and if $h_{re} = \text{FL}$, then c_{re} returns the number of bits needed to store remainders values (i.e. the original value$-j$ bits). Otherwise, in both cases B_{base} is returned, i.e. number of bits needed to represent base type.

Also, let $c_{out}(j, D)$ be the number of outliers generated when using j bits as base bit encoding, defined as:

$$c_{out}(j, D) = \sum_{l=j+1}^{32} s_{bit}(l, D). \tag{11}$$

Next, let us define a function that returns the estimated compression size for compression plan $p = (t, \text{PFL}, h)$, depending on number of bits j used to encode the values:

$$g_{PFL}(j, h, D) := \#D \cdot j + c_{out}(j, D) \cdot (c_{in}(h_i, D) + c_{re}(j, h_r, D)), \tag{12}$$

where $h = (h_{in}, h_{re})$. The number of used bits j, determines the number of outliers, as each value that needs more then j bits to be written, will be treated as an outlier. Having said that, the returned value depends on the base compression array size (i.e. $\#D \cdot j$), on the number of outliers generated (i.e. $c_{out}(j, D)$) and the way how they will be stored (i.e. $c_{in}(h_i, D) + c_{re}(j, h_r, D)$).

The function J_{PFL} given by

$$J_{PFL}(h, D) := \min_{1 \le j \le 32} g_{PFL}(j, h, D) \tag{13}$$

returns the estimated compression size for compression plan $p = (t, \text{PFL}, h)$. Notice that the smaller j is, the smaller the size of the base compression array becomes and the more outliers we have. This is why we need to minimize g_{PFL} over j. We call

$$j_{PFL} := argmin\ J_{PFL}(h, D) := \min_{1 \le j \le 32} \{j \colon g_{PFL}(j, h, D) = J_{PFL}(h, D)\} \tag{14}$$

the optimal parameter for J_{PFL}, helper algorithms h and data D.

Similarly, for a compression plan $p = (t, \text{PFOR}, h)$ we define

$$J_{PFOR}(h, D) := J_{PFL}(h, t_{sub}(min(D), D)), \tag{15}$$

which returns the estimated compression size. Then

$$j_{PFOR} := argmin\ J_{PFL}(h, t_{sub}(min(D), D)) \tag{16}$$

is the optimal parameter for J_{PFOR}, helper algorithms h and data D.

5.4 Optimal Parametrization and Compression Size Estimation for (P)DICT

PDICT works on dictionary counter array and uses it to build an optimal dictionary with exceptions (minimizing estimated compression size after applying PDICT algorithm and using FL helper algorithms). Therefore, PDICT generates three output arrays: base, indexes and remainders while DICT generates only two: base and indexes from which indexes may be further compressed with FL. For PDICT at least base and remainders arrays must be compressed using FL to improve compression ratio, compared to similar compression plan but with DICT as base algorithm. Indexes array compression is optional.

To estimate size of data compressed with (P)DICT algorithm we use *Dictionary counter* statistic. Let us define $s_{dict}(a, D) = \#\{i \in I \colon a = d_i \in D\}$ as a *Dictionary counter*. As a side effect this generates a dictionary for further usage if needed. Implemented on GPU with sort and reduction operations. Mostly constructed using thrust library.

Now, let d_{keys} be a set of unique values in D, i.e.

$$d_{keys}(D) := \{a : s_{dict}(a)(D) \neq 0, a \in D\} \tag{17}$$

and let $d_{top}(k, D) \subseteq d_{keys}(D)$ be such that $\#d_{top}(k, D) = k$ and for $a \in d_{top}(k, D), b \notin d_{top}(k, topD)$ we have $s_{dict}(a) \geq s_{dict}(b)$ (if there are more sets satisfying this condition, we pick one of them) (Fig. 7).

Let d_{head} returns size of dictionary header array (i.e. array that stores sorted $d_{keys}(D)$)

$$d_{head}(D) := B_{dict} \cdot \#d_{keys}(D). \tag{18}$$

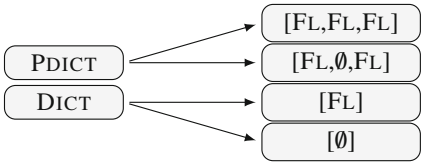

Fig. 7. PDICT and DICT base algorithms and possible composition with auxiliary algorithms.

Let us define helper functions d_{in}, d_{ba} and d_{re}:

$$d_{ba}(j, h_{ba}, D) := \begin{cases} j & \text{if } h_{ba} = FL, \\ B_{dict} & \text{otherwise,} \end{cases} \tag{19}$$

$$d_{in}(h_{in}, D) := c_{in}(h_{in}, D) \tag{20}$$

$$d_{re}(j, h_{re}, D) := \begin{cases} (b_{index}(\#d_{keys}(D)) - j) & \text{if } h_{re} = FL, \\ B_{dict} & \text{otherwise.} \end{cases} \tag{21}$$

The returned values depend on the value of $h \in \{\text{FL}, \emptyset\}^3$, which indicates whether a helper compression algorithm is used or not. If $h_{ba} = \text{FL}$, then d_{ba} returns the number of bits needed to store each element in base compression array position array, if $h_{in} = \text{FL}$ returns the number of bits needed to store each element in position (this is exactly the same as c_{in} in the previous section) and if $h_{re} = \text{FL}$, then d_{re} returns the number of bits needed to store remainders' values (i.e. the original value $- j$ bits).

Consider a compression plan $p = (t, \text{DICT}, h) \in P'$, i.e. a plan, where base compression algorithm b is set to DICT. Then estimated compression size equals to

$$J_{DICT}(h, D) := d_{ba}(h, D) \cdot \#D + d_{head}(D), \tag{22}$$

and includes size needed to store auxiliary dictionary table. Also in case of compression plans with $b = \text{DICT}$, helper algorithms may appear so this is also included.

The following formulas return estimated compression size for compression plan $p = (t, \text{PDICT}, h)$:

$$d_{out}(i, D) := \sum_{a \notin d_{top}(2^i, D)} s_{dict}(a, D) \tag{23}$$

$$\begin{aligned} G_{\text{PDICT}}(i, h, D) := {} & d_{ba}(i, h_{ba}, D) \cdot \#D + d_{head}(D) \\ & + d_{out}(i, D) \cdot \Big(d_{in}(h_{in}, D) + d_{re}(i, h_{re}, D) \Big) \end{aligned} \tag{24}$$

$$J_{\text{PDICT}}(h, D) := \min_{1 \le i \le log_2 B_{dict}} G_{\text{PDICT}}(i, h, D) \tag{25}$$

where $b = \text{PDICT}$, $h = (h_b, h_{in}, h_{re})$ and at least $h_{ba} = \text{FL}$, $h_{re} = \text{FL}$ (i.e. any compression plan in this form $(t, \text{PDICT}, (\text{FL}, h_{in}, \text{FL}))$). Without above

assumption PDICT will not improve compression ratio compared to DICT. We call

$$j_{PDICT} := argmin\ J_{\text{PDICT}}(h, D) := \min_{1 \leq i \leq log_2 B_{dict}} \{i \colon g_{DICT}(i, h, D) = J_{\text{PDICT}}(h, D)\} \tag{26}$$

the optimal parameter for J_{PDICT}, helper algorithms h and data D.

5.5 Optimal Parametrization and Compression Size Estimation for RLE and PCONST

To estimate size of data compressed with RLE algorithm we use *Run length counter*. Let us define $s_{rle}(D) = (a_r, a_v)$ where a_r and a_v are run-length and values arrays generated from data D, respectively. This is implemented on GPU with reduction operation on key-value pairs.

Let us define a function that returns the estimated compression size for compression plan $p = (t, RLE, h)$:

$$J_{\text{RLE}}(h, D) := B_{base} \cdot \#a_r + B_{base} \cdot \#a_v$$

where $(a_r, a_v) = s_{rle}(D)$. Now, additional helper algorithms may be used on a_r and a_v arrays, this however requires additional statistic generation step (Fig. 8).

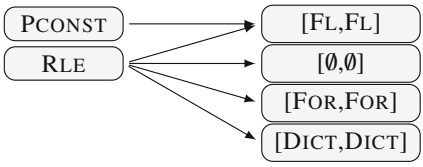

Fig. 8. RLE and PCONST base algorithms and possible composition with auxiliary algorithms.

Lastly, the PCONST algorithm, which in nature reminds RLE however, achieving the objective somewhat differently. In this algorithm a dominant value $d_{top}(1, D)$ is selected from dataset D (and stored in header), all other values are stored as outliers in PATCH arrays (index and remainders arrays). Following formula returns estimated compression size for compression plan $p = (t, \text{PCONST}, h)$:

$$J_{PCONST}(h, D) := B_{base} + \sum_{a \notin d_{top}(1,D)} s_{dict}(a, D) \cdot (c_{in}(h_{in}, D) + c_{re}(0, h_{re}, D)). \tag{27}$$

$$J_{PCONST}(h, D) := B_{base} + d_{out}(0, D) \cdot s_{dict}(a, D) \cdot (c_{in}(h_{in}, D) + c_{re}(0, h_{re}, D)). \tag{28}$$

Note that, we do not divide values when creating remainders array, instead we whole value (without dividing). This is because we only create PATCH arrays

in this algorithm. Decompression involves creation of constant data set using dominant value and applying PATCH array afterwards.[2]

6 Bi-objective Compression Plan Selection

The lightweight compression algorithms, are primarily designed for applications favouring compression/decompression speed over compression ratio. Unfortunately decompression of cascaded compression plan is usually more computationally demanding then decoding a simple compression scheme. We are facing a dilemma, how much of the processing speed may be sacrificed to gain better compression ratio.

Since there is no clear right answer, we propose optimization model which is designed to support compression plan selection in the presence of trade-offs between decompression speed and compression ratio.

Section 5 describes how to estimate compressed data size according to a predicted compression plan. In Sect. 6.2 we will construct a function which estimates decompression speed of cascaded compression plan. Finally, in Sect. 6.3 we will discuss how to combine those two objectives.

6.1 Notation

Table 3 contains a summary of the notation used in this section. Assume a set of units U, a decompression plan set P and a dataset D. The goal is to choose such a compression plan that gives good compression ratio and yet allows fast decompression (i.e. compressed data transfer time + decompression time $\geq$ decompressed data transfer time).

As a result *Cascaded Compression Planner* returns set $P' \subset P$. Let us define:

$$f_t(p'_i, D) = T(u, p'_i, D), \tag{29}$$

$$f_r(p'_i, D) = J(p'_i, D) \tag{30}$$

where $p'_i \in P'$, and u is device on which time measurements where done.

6.2 Decompression Time Estimation

Compression ratio and decompression time are the input parameters to bi-objective compression optimization. Therefore, all candidate compression plans need to have their decompression time estimated. In this, section we will discuss the problem of estimation of decompression time for compression plans.

Estimation of decompression time for the candidate compression plan is calculated based on the estimated decompression time for each algorithm contained in in the plan. Recall that $P'(D) \ni p = (t, b, h)$ is a cascaded compression plan,

[2] Note that this is a certain simplification, i.e. instead $c_{re}(0, h_{re}, \bar{D})$ where $\bar{D}$ is dataset after removing all instances of dominant value.

Table 3. Symbols used in the definition of our optimisation model

Symbol	Description
$U = \{u_1, u_2, \dots u_n\}$	Set of computational units available to process data
D	Dataset
$p_i \in P$	Decompression plan p_i from set of decompression plans P
$T(u, p, D)$	Estimated run time of the decompression plan p using device u on the data D
f_t	Estimated maximal run time for decompression plan
f_r	Estimated compression ratio
f_b	Estimated decompression run time and compression ratio bi-objective scalarization

where $t \in \mathcal{T}, b \in \mathcal{B}, h \in \mathcal{H}$ are transformation, base and helper algorithms, respectively. One can treat t, h as vectors of operations to perform, where operations are lightweight compression algorithms or empty operation $\varnothing$.

Let us denote by $T(p, D)$ the estimated decompression execution time for compression plan p and data D:

$$T(p, D) := \sum_{i=0,1} m_{t_i}(\#D) + T_b(h, D). \tag{31}$$

In the next paragraphs we describe the details of time estimation functions for different lightweight compression algorithms. Not interested readers may skip to the Sect. 6.3 not loosing the main contribution of our work.

Decompression Time Estimation Details. Slightly abusing notation, by $m_\varnothing(l) := 0$ and $m_\varnothing(j, l) := 0$ we denote execution time of empty operation used in transform and helper algorithms sections, respectively.

Next, let us define functions which return estimated decompression time for following algorithms Scale, Delta, Fl, For and Dict. Depending on algorithm type estimation functions require different arguments. We have $m_{\text{Scale}}(l)$, $m_{\text{Delta}}(l)$, $m_{\text{Fl}}(j, l)$, $m_{\text{For}}(j, l)$, $m_{\text{Dict}}(d, l)$, respectively, where l stands for data size, j is the bit length used in Fl and For algorithms, d is the size of used dictionary.

Now, we define decompression estimation functions T_b for compression plan $p = (t, b, h)$ and data D. For $b \in \{\text{Fl}, \text{For}, \text{Dict}\}$ we have:

$$T_{\text{Fl}}(h, D) := m_{\text{Fl}}(j_{\text{Fl}}, \#D) \tag{32}$$

$$T_{\text{For}}(h, D) := m_{\text{For}}(j_{\text{For}}, \#D) \tag{33}$$

$$T_{\text{Dict}}(h, D) := m_{\text{Dict}}(\#d_{keys}(D), \#D) + m_{h_{ba}}(b_{min}(d_{keys}(D)), \#D), \tag{34}$$

where j_{FL}, j_{FOR} are number of bit used for base encoding (see Eqs. 5 and 8), d_{keys} and b_{min} are defined in Eqs. 17 and 2, respectively, h_{ba} is a helper algorithm for DICT.

For $b \in \{\text{PFL}, \text{PFOR}\}$ we need to define auxiliary functions first. Recall that for $b \in \{\text{PFL}, \text{PFOR}\}$, helper algorithms are of the following form $h = (h_{in}, h_{re})$ and functions c_{in}, c_{re} which return the number of necessary bits for helper algorithms are given by Eqs. 9 and 10. Let

$$m_{PFL}(j, o, h, D) := m_{h_{in}}\big(c_{in}(h_{in}, D), o\big) + m_{h_{re}}\big(c_{re}(j, h_{re}, D), o\big), \tag{35}$$

$$m_{PFOR}(j, h, D) := m_{PFL}(j, h, D) \tag{36}$$

where j is the number of bits used for base encoding, o is the number of outliers and D represents data set.

Similarly for $b = \text{PDICT}$, helper algorithms are of the following form $h = (h_{ba}, h_{in}, h_{re})$ and functions d_{ba}, d_{in}, d_{re} which return the number of necessary bits for helper algorithms are given by Eqs. 19, 20 and 21. Let

$$\begin{aligned} m_{PDICT}(j, o, h, D) := {} & m_{h_{ba}}\big(d_{ba}(j, h_{ba}, D), o\big) \\ & + m_{h_{in}}\big(d_{in}(h_{in}, D), o\big) + m_{h_{re}}\big(d_{re}(j, h_{re}, D), o\big) \end{aligned} \tag{37}$$

where j is the number of bits used for base encoding, o is the number of outliers and D represents data set.

Then for any plan where $b \in \{PFL, PFOR, PDICT\}$ we define:

$$T_{\text{PFL}}(h, D) := m_{\text{FL}}(j_{\text{PFL}}, \#D) + m_{PFL}(j_{PFL}, c_{out}(j, D), h, D), \tag{38}$$

$$T_{\text{PFOR}}(h, D) := m_{\text{FOR}}(j_{\text{PFOR}}, \#D) + m_{PFOR}(j_{PFOR}, c_{out}(j, D), h, D), \tag{39}$$

$$T_{\text{PDICT}}(h, D) := m_{\text{DICT}}(2^{j_{\text{PDICT}}}, \#D) + m_{PDICT}(j_{PDICT}, d_{out}(j, D), h, D) \tag{40}$$

where $j_{PFL}, j_{\text{PFOR}}, j_{\text{PDICT}}$ are optimal parameters for PFL, PFOR and PDICT (see Eqs. 14, 16 and 26), c_{out} and d_{out} return the number of a outliers (see Eqs. 11 and 23).

Similarly for PCONST, define

$$T_{\text{PCONST}}(h, D) := m_{CONST}(\#D) + m_{PFL}(0, d_{out}(0, D), h, D). \tag{41}$$

Decompression time of RLE depends only on data length [13], let function $m_{\text{RLE}}(l)$ return estimated decompression time for RLE:

$$T_{\text{RLE}}(h, D) := m_{RLE}(D) + \text{decompression time of helpers algorithms used.} \tag{42}$$

6.3 Bi-objective Compression Planner

We will use *a priori* articulation of preference approach which is often applied to multi-objective optimization problems. It may be realized as the scalarization of objectives, i.e., all objective functions are combined to form a single function. In this work we will use *weighted product* method, where weights express user preference [16]. Let us define:

$$f_b(u, p, D) = f_r(p, D)^{w_r} \cdot f_t(u, p, D)^{w_t}$$

where w_t and w_c are weights which reflect how important cost and time is (the bigger the weight the more important the feature). It is worth to mention that a special case with $w_t = w_r = 1$ (i.e., without any preferences) is equivalent to Nash arbitration method (or objective product method) [16].

We can also extend this to handle a set of devices U. This allows us to consider compression plans taking into account all devices on which the data may be decompressed. Let us define function f_m: $f_m(p, D) = (\max_{u_k \in U} t_t(u_k, p, D))^{w_t} + f_r(p, D)^{w_r}$ this function optimizes for the slowest device for each plan.

7 Preliminary Runtime Results

In this section we discuss effectiveness of the dynamic compression planner in the context of the resulting compression ratio. To fully evaluate the proposed compression framework, we still need to perform more experiments regarding the context of decompression speed for many other data sets and devices. The detailed processing time analysis including threads instruction throughput and effectively achieved memory bandwidth needs a lot of runtime trials and will be addressed in the next paper.

We compared effectiveness of a dynamic compression planner and a single static plan within the same CF (Column Family – portion of data rolled in a database) by running the prototype system on samples from a set of network servers monitoring. The data included memory usage, the number of exceptions reported, services occupancy time or CPU load. Data covered a sample of 20 days of constant monitoring and contained about 91 K data points in a few time series being a sample from a telecommunication monitoring system.

We used the following equipment: *Nvidia® Tesla C2070 (CC 2.0)* with 2687 MB; 2 x Six-Core processor *AMD® Opteron™* with 31 GB RAM, *Intel®* RAID Controller *RS2BL040* set in RAID 5, 4 drives *Seagate® Constellation ES ST2000NM0011* 2000 GB, Linux kernel 2.6.38–11 with the CUDA driver version 5.0.

7.1 Evaluation of the Compression Planner

The evaluation was divided into two parts. The first measured efficiency of the dynamic planner and was intended to prove the basic contribution of this work. The second checked efficiency of GPU based statistics evaluation when compared to CPU and proved contribution concerning time efficiency.

7.2 Dynamic Compression Planner Evaluation

Figure 9 shows compression ratio (original size/compressed size) using several static plans (one compression plan for the whole column family) and a dynamic plan (dynamically chosen compression plan for different metrics, tags and time ranges). In case of timestamps, five static plans were generated using DELTA algorithm combined with five base compression methods (and helper compression algorithms if suitable). Similarly, for data values five plans where selected except

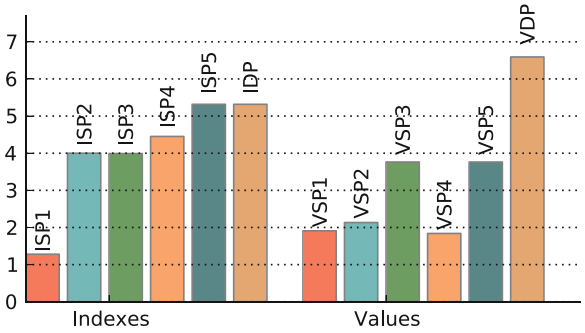

Fig. 9. Efficiency of the prototype dynamic compression system working on GPU for sample time series. Compression ratio (original size/compressed size) for static (*SP) and dynamic (*DP) plans. I stands for indices and V for values.

Scale was used instead of Delta. We may observe, that for timestamp arrays, compression ratio of a dynamic compression plan was equivalent to best static compression plan. This situation appeared because all time series were evenly sampled in this case. Therefore one static plan for all metrics generated the same results as a dynamic plan, selected for each time series separately. Note that in real systems, some measurements may be event-driven and thus dynamic plan could generate better results.

For data values, a dynamic compression plan almost doubles compression ratio of the best static compression plan which means that dynamic tuning was much better than selection of one static plan for the whole buffered column family. Obviously, this is heavily data dependant, but as a general rule a dynamic compression plan will never generate a compression plan worse than the best static plan (as it always minimizes locally). Additionally hints system may be used to enforce a static compression plan for cases when a dynamically generated compression plan does not produce satisfactory profits.

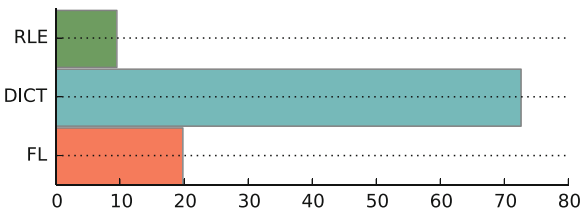

Fig. 10. Statistics calculation speed-up on sample data with 8 millions values compared to single threaded GPU. This includes GPU memory transfer (higher is better).

7.3 Evaluation of Statistics Calculations and Bandwidth of Compression Methods on GPU

In Fig. 10 on the right GPU statistic generator is compared to similar CPU version (implemented as a single thread). Generated statistics are then used to support the compression planner selection. A significant speed-up of factors from 10 to 70 was gained thus usage of GPU platform in statistics calculation step is justified.

Furthermore, GPU platform allows to archive high compression bandwidth for lightweight compression schemes (see Table 4 and results from [13]). We conclude that GPU may be used just as a kind of compression coprocessor even if there are no other computations done on a GPU side.

Table 4. Achieved bandwidth of pure compression methods (no IO).

Algorithm	Delta	Scale	(P)Dict	(P)For	(P)Fl	Rle	Pconst
GB/s	28.875	41.134	6.924	9.124	9.375	5.005	2.147

8 Conclusions and Future Research

Monitoring of complex computer infrastructure is already an important industrial problem. Time series database system try to address many of the problems which may appear, like: scalability, robustness, safety and availability. Our research focuses on a time series database system supported by GPU coprocessors which may be used for many purposes, from data compression to analysis and aggregation.

In this paper, touching lightweight compression methods we successfully extended results from [13,19]. We not only designed and implemented new patched compression algorithms on GPU (i.e. Patched Dict, Patched Const. and Patched Fixed Length) but also presented a dynamic compression planner. Our novel prototype system was adapted to time series compression in a NoSQL database. The compression method is composed of several nested algorithms.

Furthermore our compression planner uses dynamic data statistics calculated on the fly using a GPU device for the best possible lightweight cascaded compression plan selection. We believe that the resulting compression ratios and algorithms bandwidth (please refer to Table 4) combined with ultra fast decompression [13,19] on GPU are especially attractive for time series databases.

Our future work will concentrate on query optimization in hybrid CPU/GPU environment, query execution on partially compressed data and extending dynamic compression planner by introducing additional costs factors (i.e. decompression execution time [13] or potential of query execution on compressed data) leading to a full-flagged time series database system.

References

1. Apache HBase (2013). http://hbase.apache.org
2. OpenTSDB - A Distributed, Scalable Monitoring System (2013). http://opentsdb.net/
3. ParStream - website (2013). https://www.parstream.com
4. TempoDB - Hosted time series database service (2013). https://tempo-db.com/
5. Andrzejewski, W., Wrembel, R.: GPU-WAH: applying GPUs to compressing bitmap indexes with word aligned hybrid. In: Bringas, P.G., Hameurlain, A., Quirchmayr, G. (eds.) DEXA 2010, Part II. LNCS, vol. 6262, pp. 315–329. Springer, Heidelberg (2010)
6. Boncz, P.A., Zukowski, M., Nes, N.: Monetdb/x100: hyper-pipelining query execution. In: CIDR, pp. 225–237 (2005)
7. Breß, S., Schallehn, E., Geist, I.: Towards Optimization of Hybrid CPU/GPU Query Plans in Database Systems. In: New Trends in Databases and Information Systems, pp. 27–35. Springer, Heidelberg (2013)
8. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: Bigtable: a distributed storage system for structured data. In: OSDI'06: Seventh Symposium on Operating System Design and Implementation, Seattle, WA, November, pp. 205–218 (2006)
9. Chatfield, C.: The Analysis of Time Series: An Introduction, 6th edn. CRC Press, Florida (2004)
10. Cloudkick. 4 months with cassandra, a love story, March 2010. https://www.cloudkick.com/blog/2010/mar/02/4_months_with_cassandra/
11. Dean, J., Ghemawat, S.: Mapreduce simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2004)
12. Delbru, R., Campinas, S., Samp, K., Tummarello, G.: Adaptive frame of reference for compressing inverted lists. Technical report, DERI - Digital Enterprise Research Institute, December 2010
13. Fang, W., He, B., Luo, Q.: Database compression on graphics processors. Proc. VLDB Endowment **3**(1–2), 670–680 (2010)
14. Fink, E., Gandhi, H.S.: Compression of time series by extracting major extrema. J. Exp. Theor. Artif. Intell. **23**(2), 255–270 (2011)
15. Lees, M., Ellen, R., Steffens, M., Brodie, P., Mareels, I., Evans, R.: Information infrastructures for utilities management in the brewing industry. In: Herrero, P., Panetto, H., Meersman, R., Dillon, T. (eds.) OTM-WS 2012. LNCS, vol. 7567, pp. 73–77. Springer, Heidelberg (2012)
16. Marler, R.T., Arora, J.S.: Survey of multi-objective optimization methods for engineering. Struct. Mult. Optim. **26**(6), 369–395 (2004)
17. OpenTSDB. Whats opentsdb (2010–2012). http://opentsdb.net/
18. Papadimitriou, C.H., Yannakakis, M.: Multiobjective query optimization. In: Proceedings of the Twentieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, pp. 52–59. ACM (2001)
19. Przymus, P., Kaczmarski, K.: Improving efficiency of data intensive applications on GPU using lightweight compression. In: Herrero, P., Panetto, H., Meersman, R., Dillon, T. (eds.) OTM-WS 2012. LNCS, vol. 7567, pp. 3–12. Springer, Heidelberg (2012)
20. Przymus, P., Kaczmarski, K.: Dynamic compression strategy for time series database using GPU. In: New Trends in Databases and Information Systems. 17th East-European Conference on Advances in Databases and Information Systems, 1–4 September 2013 - Genoa, Italy (2013)

21. Przymus, P., Kaczmarski, K.: Time series queries processing with gpu support. In: New Trends in Databases and Information Systems. 17th East-European Conference on Advances in Databases and Information Systems, 1–4 September 2013 - Genoa, Italy (2013)
22. Przymus, P., Kaczmarski, K., Stencel, K.: A bi-objective optimization framework for heterogeneous CPU/GPU query plans. In: CS&P 2013 Concurrency, Specification and Programming. Proceedings of the 22nd International Workshop on Concurrency, Specification and Programming, 25–27 September 2013 - Warsaw, Poland (2013)
23. Przymus, P., Rykaczewski, K., Wiśniewski, R.: Application of wavelets and Kernel methods to detection and extraction of behaviours of freshwater mussels. In: Kim, T., Adeli, H., Slezak, D., Sandnes, F.E., Song, X., Chung, K., Arnett, K.P. (eds.) FGIT 2011. LNCS, vol. 7105, pp. 43–54. Springer, Heidelberg (2011)
24. Wu, L., Storus, M., Cross, D.: Cs315a: final project cuda wuda shuda: Cuda compression project (2009)
25. Yan, H., Ding, S., Suel, T.: Inverted index compression and query processing with optimized document ordering. In: Proceedings of the 18th International Conference on World Wide Web, pp. 401–410. ACM (2009)
26. Zukowski, M., Heman, S., Nes, N., Boncz, P.: Super-scalar RAM-CPU cache compression. In: ICDE'06. Proceedings of the 22nd International Conference on Data Engineering, pp. 59–59. IEEE (2006)

http://www.springer.com/978-3-662-45760-3

Transactions on Large-Scale Data- and
Knowledge-Centered Systems XV
Selected Papers from ADBIS 2013 Satellite Events
Hameurlain, A.; Küng, J.; Wagner, R.; Catania, B.; Guerrini,
G.; Palpanas, T.; Pokorný, J.; Vakali, A. (Eds.)
2014, IX, 125 p. 49 illus., Softcover
ISBN: 978-3-662-45760-3