Dynamic Compression Strategy for Time Series Database Using GPU*

Piotr Przymus[1] and Krzysztof Kaczmarski[2]

[1] Nicolaus Copernicus University, Poland
eror@umk.mat.pl
[2] Warsaw University of Technology, 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. GPU devices combined with fast compression and decompression algorithms open new horizons for data intensive systems. In this paper we present improved cascaded compression mechanism for time series databases build on Big Table–like solution. We achieved extremely fast compression methods with good compression ratio.

Keywords: time series database, lightweight lossless compression, GPU, CUDA.

1 Introduction

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 hours a day, seven days a week, generating terabytes of logs. Due to regression errors checking and early malfunction prediction these data must be kept with proper resolution including all details. Solutions like OpenTSDB [11], TempoDB [3] and others deal very well with these kind of tasks. Most of them work on a clone of Big Table approach from Google [5], a distributed hash table with mutual ability to write and read data in the same time.

Usually systems compress data before writing to a long-term storage. It is much more efficient to store data for some time in a memory or disk buffer and compress it before flushing to disk. This process is known as a table row rolling. Current systems like HBase [1], Casandra [6] and others offer compression optimization 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).

Similar problems appear in in-memory database systems. Solutions based on GPU processing (like ParStream [2]) tend to pack as many data into GPU devices global memory as possible. Efficient data compression method would significantly improve abilities of these systems. An average internet service with about 10 thousands of simultaneously working users may generate around 80GB of logs every day. After

* The project is funded by National Science Centre, decision DEC-2012/07/D/ST6/02483.

B. Catania et al. (eds.), *New Trends in Databases and Information Systems*,
Advances in Intelligent Systems and Computing 241,
DOI: 10.1007/978-3-319-01863-8_26,

compression they could fit into two Nvidia Tesla devices where average query can be processed within seconds compared to minutes in case of standard systems.

In case of time series compression ratio could be improved by a method tuned to types of data including its variability, span, differences, etc. However, tuning time slows down compression and often cannot fit into time window available in real time monitoring systems. This paper describes a dynamic compression strategy planner for time series databases using GPU processors with reasonable processing time and compression ratios. What is even more important, the resulting compressed data block can be decompressed very quickly directly into the GPU memory additionally allowing for ultra fast query processing, what we discussed in our previous publication [12].

The main contribution of this work is:

- three new implementations of patched compression algorithms on GPU
- a new dynamic compression planner for lightweight compression methods
- categorization for compression methods and reduction of configuration space for optimal plan searching
- evaluation of the achieved results on real-life data

Section 2.1 presents a general view of the system, section 2.2 contains the main contribution of our work: the dynamically optimized compression system. Experimental runtime results are contained in section 3 while section 4 concludes.

1.1 Motivation and Related Work

Optimal data 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 [9]. For industrial monitoring, lossy compression cannot be used due to possible degradation of anomalies.

In case of lossless compression one can use common algorithms (ZIP, LZO) which tend to consume lot of computation resources [4,15] or lightweight methods which are faster but not so effective. Our dynamic method attempts to combine properties of both approaches: is lossless but much faster than common algorithms, offers good compression ratios and may be computed incrementally. Also ability to decompress values directly into the processor shared memory should improve GPU memory bandwidth and enable it to be used in many data intensive applications.

An important challenge is to improve compression factor with an acceptable processing time in case of variable sampling periods. Interesting results in the filed of lossless compression done on GPU were presented by Fang et al. [8]. Using a query 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.

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. Comparing to [8] and [4] we can achieve better results by using dynamic compression planning methods with automated compression tuning upon processed time series data.

2 Dynamically Optimized Compression System

2.1 Time Series Database Architecture

General View A typical time series database consists of three layers: data insertion module, data storage and querying engine. Our compression mechanism touches all the layers working as a middle tier between the data storage and the rest of the system. In this work we shall focus on data compression mechanism assuming that decompression used by the query engine is an obvious opposite process.

2.2 Data Insertion

Data Collection. The data acquisition from ongoing measurements, industrial processes monitoring [10], scientific experiments [13], 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: *a*) number of data points (timestamps and their values) in one time series should not be limited; *b*) each time series should be identified by a name which is often called a *metric name*; *c*) each time series can be additionally marked with a set of *tags* describing measurement details which together with metric name uniquely identifies time series; *d*) 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.

Initial Buffering. 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 minutes, 2 hours, 24 hours – depending on the number of observations). This step directly predeceases dynamic compression which is described in the next section.

2.3 Compression Algorithms

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 [15] 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 [15]. 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 [15] or algorithms that do not generate compulsive exceptions [7,14]. The algorithms in this paper are based largely on those described by Yan [14]. In this version of the compression block is extended by two additional arrays - exceptions position and values. Decompression involves extracting data using the underlying decompression algorithm and then applying a patch (from exceptions values 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.

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.

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 minutes 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 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 [8]). 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.

2.4 Cascaded Compression Planer

Cascaded compression can significantly improve the compression ratio. However, there are two problems arising. First, there is a risk that cost of decompression will neglect benefits from lower transfer costs. Second problem is arising when searching for an optimal compression methods composition. Even relatively short plan of cascaded compressions (i.e. using 6 compositions out of 10 algorithms with repetitions) may generate a very large search space (in our example $\sum_{i=1}^{6} 10^i = 1,111,110$). Significant reductions must be done in order to achieve fast compression and best plan fitting in a reasonable time. We assumed that the time limit is set by corresponding CPU performance measured for one base compression step (see next section). Therefore in our method, the whole compression process including copying data to GPU, data statistics evaluation, optimal plan searching and final compression plan execution must be always faster than mentioned limit.

Stage One: Static Planner – Reduction of Plans Search Space. In the first static stage we determined acceptable transitions between compression algorithms which were divided into three categories: initial transformation, base compression, helper

compression. The complete compression schema is always composed of algorithms selected from these ordered categories with the following purposes:

1. **Transformation algorithms (SCALE, DELTA).** All algorithms in this section are optional but may be used together (if present must be applied in the given order). Goal: Improve properties of data storage and prepare for better compression.
2. **Base compression algorithms (PDICT, PFL, PFOR, RLE, PCONST).** Only one algorithm may be selected as the base algorithm. All algorithms in this section use two or more arrays. Some of them, may qualify for further compression using *Helper compression algorithms*.
3. **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. For example, exceptions and values arrays in all PATCH algorithms may only be compressed with FL.

Composition of all sensible paths between algorithms in these three categories leaves only 32 suitable compression plans out of former one million. The longest possible cascaded compression plan may be composed of six steps.

Stage Two: Hints System – Possibility of Manual Tuning. 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 hints are located in Table 1.

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

Hints	Meaning
SCALE, DELTA, (P)FL, (P)FOR RLE (P)DICT, PCONST	Enforces a specific compression algorithm in the plan.
SORTED	Specify whether the data is sorted.
TIMESTAMP	Automatically added by system to timestamps. Sets SORTED to True and SCALE to False.
DATA	Automatically added by system to time series values. If not otherwise specified sets SORTED to False and SCALE to False.

Stage Three: Dynamic Statistics Generator – Finding an Optimal Plan. In the last step, a maximal compression ratio plan is selected upon dynamically computed statistics. In our system they must be generated for each metric and rolled time period. Precomputing them and storing aside is not an optimal solution due to necessity of constant update and allocation of additional memory. Therefore all necessary estimations are calculated during this stage. Please note that if a plan contains a transformation algorithm then it must be applied before calculating statistics because it influences data.

Estimation results heavily depend on compression algorithms parameters. In [8] the choice of optimal parameters was straightforward, because 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 PATCH 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, 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. 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. This kind of computationally intensive task is ideal for parallel processing on a GPU device.

The following algorithms are used to calculate statistics.

- Bit histogram – used in size estimation of (P)FL and (P)FOR (includes estimation size of PATCH arrays with and without compression). Implemented with double buffering (registers and shared memory).
- Dictionary counter – used in size estimation of (P)DICT (includes estimation size of PATCH arrays with and without compression). As a side effect dictionary is generated for further usage if needed. Implemented with sort and reduction operations.
- Run length counter – used in RLE and PCONST. Implemented with reduction operation on key-value pairs.

All the above procedures were implemented using GPU parallel primitives mostly with CUDA Thrust library assuring the best performance. After statistics calculation step, the data is located in a GPU device memory and can be compressed without additional costs associated with the data transfer.

A complete plan evaluation must include base compression algorithm and dedicated helper algorithms sets. In case of all base algorithms, except for RLE, the helper compression algorithms appearing in the plan are already taken into account in the statistics. RLE requires to perform compression and then calculate statistics for the helper algorithms. For example, let us consider the following compression plan [[SCALE, DELTA], [PFL], [FL,FL]] (notation – [transformation algorithms, base compression, helper methods]), first we apply transformation algorithms before estimating base algorithm compression size. Let us denote the data after applying the transformation algorithms by $(x_i)_{i\in I}$. For $1 \leq j \leq 32$ let $g(j) = \#\{i \in I : j \text{ bits are sufficient to write } x_i\}$. The size of the data after compression using remaining part of plan (i.e. [[PFL], [FL,FL]]) is then estimated by

$$E := \min_{1 \le j \le 32} \left(\sum_{l=1}^{j} g(l) \cdot len(g) + \sum_{l=j+1}^{32} g(l) \left(\lceil \log_2 len(g) \rceil + last(g) - j \right) \right),$$

where $len(g) = \sum_{l=1}^{32} g(l)$ and $last(g) = \max_{1 \le l \le 32} \{l : g(l) \neq 0\}$. First sum estimates base algorithm compression size and second estimates compression size of two exception arrays compressed using FL algorithm. If we change PFL to PFOR similar estimation is made but in first step $\min (x_i)_{i \in I}$ is subtracted from all values. PDICT works on dictionary counter array and uses it to build an optimal dictionary with exceptions (i.e. PDICT generate three output arrays and each may be compressed using FL, optimal dictionary with exception is such that minimizes estimated compression size after applying PDICT algorithm and using FL helper algorithm). Detailed description of other evaluation functions is beyond the size limitation of this paper and will be published separately.

3 Runtime Results

We compared effectiveness of 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 91K data points in just a few time series (available at `www.mat.umk.pl/~eror/gid2013`). It was taken as a very short and limited sample from a telecommunication monitoring system which collects about 700.000 data points per day. Please note that in this case quality of the sample (its origin) is more important than its length.

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.

3.1 Evaluation of Compression Planer

The evaluation was divided into two parts. The first measured efficiency of 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 proving contribution concerning time efficiency.

Figure 1 on the right shows compression ratio (original size / compressed size) using several static plans (one compression plan for the whole column family) and 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 SCALE was used instead of DELTA. We may observe, that for timestamp arrays, compression ratio of 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

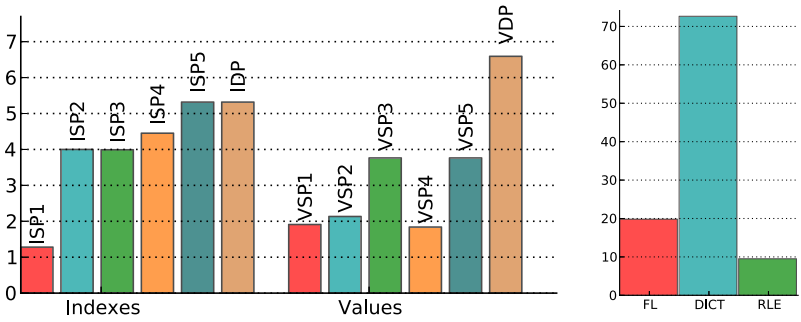

Fig. 1. Efficiency of the prototype dynamic compression system working on GPU. (left) Compression ratio for static (SP*) and dynamic (DP*) plans. I stands for index and V for values. (right) Statistics calculation speed-up including GPU memory transfer and using sample data with 8M values. (higher is better)

Table 2. 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

plan for all metrics generated the same results as 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, dynamic compression plan almost doubles compression ratio of 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 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 static compression plan for cases when using dynamically generated compression plan does not produce satisfactory profits.

In Fig. 1 on the left GPU statistic generator is compared to similar CPU version (implemented as a single thread). A significant speed-up of factors from 10 to 70 was gained which guarantees no slowdown in a lightweight compression application.

4 Conclusions and Future Research

We successfully extended results from [8,12] by introducing three new implementations of patched compression algorithms on GPU (i.e. Patched DICT, Patched Const. and Patched Fixed Length). Furthermore we presented a dynamic compression planner adapted to time series compression in a NoSQL database. Our planner uses statistics calculated on the fly for the best plan selection. Resulting compression ratios and algorithms bandwidth combined with ultrafast decompression [8,12] on GPU are attractive solutions for 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[8] or potential of query execution on compressed data).

References

1. Apache HBase (2013), http://hbase.apache.org
2. ParStream - website (2013), https://www.parstream.com
3. TempoDB – Hosted time series database service (2013), https://tempo-db.com/
4. Boncz, P.A., Zukowski, M., Nes, N.: Monetdb/x100: Hyper-pipelining query execution. In: CIDR, pp. 225–237 (2005)
5. 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 2006: Seventh Symposium on Operating System Design and Implementation, Seattle, WA, pp. 205–218 (November 2006)
6. Cloudkick. 4 months with cassandra, a love story (March 2010), https://www.cloudkick.com/blog/2010/mar/02/4_months_with_cassandra/
7. 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)
8. Fang, W., He, B., Luo, Q.: Database compression on graphics processors. Proceedings of the VLDB Endowment 3(1-2), 670–680 (2010)
9. Fink, E., Gandhi, H.S.: Compression of time series by extracting major extrema. J. Exp. Theor. Artif. Intell. 23(2), 255–270 (2011)
10. 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 2012 Workshops. LNCS, vol. 7567, pp. 73–77. Springer, Heidelberg (2012)
11. OpenTSDB. Whats opentsdb (2010-2012), http://opentsdb.net/
12. 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 2012 Workshops. LNCS, vol. 7567, pp. 3–12. Springer, Heidelberg (2012)
13. Przymus, P., Rykaczewski, K., Wiśniewski, R.: Application of wavelets and kernel methods to detection and extraction of behaviours of freshwater mussels. In: Kim, T.-h., Adeli, H., Slezak, D., Sandnes, F.E., Song, X., Chung, K.-i., Arnett, K.P. (eds.) FGIT 2011. LNCS, vol. 7105, pp. 43–54. Springer, Heidelberg (2011)
14. Yan, H., Ding, S., Suel, T.: Inverted index compression and query processing with optimized document ordering. In: Proc. of the 18th Intern. Conf. on World Wide Web, pp. 401–410. ACM (2009)
15. Zukowski, M., Heman, S., Nes, N., Boncz, P.: Super-scalar ram-cpu cache compression. In: ICDE 2006. Proc. of the 22nd Intern. Conf. on Data Engineering, p. 59. IEEE (2006)