Improving Efficiency of Data Intensive Applications on GPU Using Lightweight Compression

Piotr Przymus[1] and Krzysztof Kaczmarski[2]

[1] Nicolaus Copernicus University, Chopina 12/18, Toruń, Poland
`eror@umk.mat.pl`
[2] Warsaw University of Technology, Plac Politechniki 1, 00-661 Warszawa, Poland
`k.kaczmarski@mini.pw.edu.pl`

Abstract. In many scientific and industrial applications GPGPU (General-Purpose Computing on Graphics Processing Units) programming reported excellent speed-up when compared to traditional CPU (central processing unit) based libraries. However, for data intensive applications this benefit may be much smaller or may completely disappear due to time consuming memory transfers. Up to now, gain from processing on the GPU was noticeable only for problems where data transfer could be compensated by calculations, which usually mean large data sets and complex computations. This paper evaluates a new method of data decompression directly in GPU shared memory which minimizes data transfers on the path from disk, through main memory, global GPU device memory, to GPU processor. The method is successfully applied to pattern matching problems. Results of experiments show considerable speed improvement for large and small data volumes which is a significant step forward in GPGPU computing.

Keywords: lightweight compression, data-intensive computations, GPU, CUDA.

1 Introduction

In all branches of science and industry amount of data which needs to be processed increase every year with enormous speed. Often this analysis involve uncomplicated algorithms but working on large data sets which in most cases cannot be efficiently reduced. This kind of applications are called data-intensive and are characterized by the following properties:

1. data itself, its size, complexity or rate of acquisition is the biggest problem;
2. require fast data access and minimization of data movement;
3. expects high, preferably linear, scalability of both hardware and software platform.

One of the most typical solutions to process large volumes of data is the map-reduce algorithm which gained huge popularity due to its simplicity, scalability and distributed nature. It is designed to perform large scale computations which may last from seconds to weeks or longer and involve from several to hundreds or thousands of machines processing together peta-bytes of data.

On the opposite side we can find problems which reside in a single machine but are large enough to require high processing power to get results within milliseconds. GPU programming offers tremendous processing power and excellent scalability with increasing number of parallel threads but with several other limitations. One of them is obligatory data transfer between RAM (random-access memory) and the computing GPU processor which generates additional cost of computations when compared to a pure CPU-based solution. This barrier can make all GPU computations unsatisfactory especially for smaller problems.

Time series matching is a popular example of this kind of data intensive applications in which a user may expect ultra fast results. Therefore in the rest of this work we use this example as a proof of concept application.

1.1 Motivation

Goal of this work is to improve efficiency of data transfer between disk, through RAM, global GPU memory and processing unit, which is often a bottleneck of many algorithms and gain noticeable higher speed up of algorithms when compared to classical CPU programming. We focus on memory bound data intensive applications since many computation intensive applications already proved to be much faster when properly implemented in parallel GPU algorithms.

Let us consider a problem of matching whole patterns in time series. As a distance function we can use the Hamming distance. Due to the lack of complex calculations main limitation of this problem is data transfer. To facilitate the transfer of data we can use either hardware solutions or try to reduce the data size by compressing or eliminating data. Since the hardware is expensive and the elimination of data is not always possible, data compression is usually the only option. Classic compression algorithms are computationally expensive (gain from the transfer data does not compensate the calculations [1]) and difficult to implement on the GPU [2]. Alternatives such as lightweight compression algorithms which are successfully used for the CPU are therefore very attractive.

1.2 Related Works

Lossless data compression is a common technique for reducing data transfers. In the context of SIMD (Single Instruction Multiple Data) computations data compression was successfully utilized by [3] in tree searching to increase efficiency of cache line transfer between memory and processor. The authors indicate that GPU implementation of the same search algorithm is computational bound and cannot be improved by compression. In [4] authors present interesting study of different compression techniques for WWW data in order to achieve querying speed up. A general solution for data intensive applications by cache compression is discussed in [1]. Obviously efficiency may be increased only if decompression speed is higher than I/O operation. In our case we show that decoding is really much faster and eliminates this memory bottleneck. The compression schemes proposed by Zukowski et al. [1] offer good trade-off between compression time and encoded data size and what is more important are designed especially for super scalar processors which means also very good properties for GPU. Delbru et al. [5] develop a new Adaptive Frame of Reference compression algorithm in order to achieve significant speed up of compression time, which in turn allows for effective updates of data.

All these works are in fact entry points to our solution. Our contribution is that the compression methods have been redesigned for GPU architecture and may offer excellent encoding and decoding speed for all compression ratios. Our solution may be utilised in any data intensive algorithms involving integer or fixed-decimal number data. We also explain how this method may be extended to other algorithms.

As the preliminary work we checked if lightweight compression algorithms can be used for different kinds of common time series databases. We collected data from various data sets covering: star light curve (TR1, TR2 – [6]), CFD (contract for difference) and stock quotations for AUDJPY, EURUSD, Shenzen Development Bank respectively (IN1, IN2, TS1 – [7,8]), radioactivity in the ground at 2 hourly intervals over one year (TS2 – [8]) and sample ECG (electrocardiogram) data (PH1,PH2 – [9]). The data used in this experiment is the data with fixed decimal precision that can be stored as integers, which allows to use lightweight compression. Efficiency of conventional compression algorithms (gzip, bzip2 - with default settings) compared to the lightweight compression methods (PFOR, PFOR-DIFF) is presented in figure 1. We can notice that lightweight compression, although not achieving as high compression ratio as gzip, may also obtain similar results, bzip2 is always much better but also significantly slower. This is also proved by L. Wu et al. [2] who showed that conventional compression is too slow to increase overall algorithm efficiency for GPU.

Due to its specificity, time series usually have a good compression ratio (Fig. 1). The most important benefits of compression can be summarized as follows:

- shorter time to load data from disk
- shorter time to copy data from RAM to device
- possibility to fit larger set of data directly on the GPU (GPU DRAM is often limited)
- reduced data size in storage

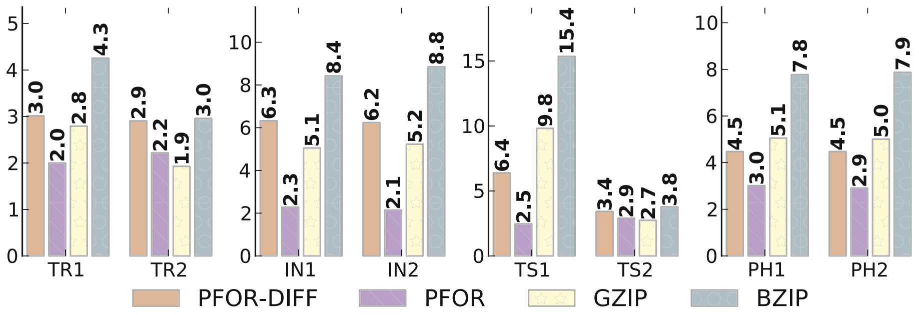


Fig. 1. Achieved compression level of lightweight compression and conventional compression methods for various datasets (higher is better). Data sets: TR1, TR2 – star light curve; IN1, IN2, TS1 – CFD and stock quotations; TS2 – radioactivity in the ground; PH1, PH2 – ECG data.

We conclude that lightweight compression may be used in time series data intensive applications achieving compression ratio from 2 to 7. This initial investigation lets us to predict that data transfer cost accompanied with decompression time may be noticeably decreased.

1.3 Research Hypothesis and Methodology

The main research hypothesis for this project is as follows.

- Data-intensive applications may increase their efficiency by utilization of lightweight compression methods in GPU global and shared memory.
- Cost of data decompression can be amortised by fewer global memory reads.
- Additional benefit may be obtained by proper utilisation of shared memory decompression.
- Data-intensive applications may benefit from GPU computing when applied for smaller data sets than without the method.

In order to verify the hypotheses we implement a prototype which will be used as a proof of concept equipped with several real-time and real-data measurements performed by fine grained timers and memory access analysis done by a professional profiler. Checking the hypotheses will involve a few kinds of experiments including data intensive application using:

1. GPU algorithm without decompression compared to CPU algorithm without decompression. This will allow to estimate possible general speed up of an algorithm when run on a GPU and also will be a starting point for other experiments by registering time necessary to perform pure computations without any decompression overhead.
2. GPU algorithm with decompression in global memory compared to CPU algorithm with decompression. This will show potential speed up when data transfer is minimised by data compression.
3. GPU algorithms with decompression in shared memory compared to GPU with decompression in global memory and CPU algorithm with decompression. This will show the speed up when minimising GPU global memory reads.

As an initial set up for this research we use time series matching problems as an exemplar of data intensive applications. Because of efficiency requirements mentioned in the previous section we utilise lightweight compression methods: PFOR and FOR-DIFF algorithms.

2 Method Description

2.1 Lightweight Compression

In this section we discuss the algorithm FOR and FOR-DIFF in different variations, as well as discuss problems and solutions when using this approach on the GPU. FOR determines range of values for the frame, and then maps the values in this interval using a minimum number of bits needed to distinguish the data [10]. A common practice is to convert the data in the frame to the interval $\{0,\ldots,max-min\}$. In this situation, we need exactly $\lceil log_2(max-min+1)\rceil$ bits to encode each value in the frame.

The main advantage of the FOR algorithm 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. Additionally functions are loop-unrolled and use only shift and mask operations. This implies that there are dedicated compression and decompression routines prepared for every bit encoding length.

The compression algorithm works as follows, for the data frame of length n loop is performed each m values (where m is a multiple of 8) and compressed using the same function at each iteration step. Decompression is similar to compression. We iterate through the m-coded values, and we use a function that decodes m values.

FOR-DIFF algorithm is similar to FOR, however, stores the differences between successive data points in frame. Compression needs to calculate the difference, then compresses them using the FOR compression scheme. Decompression begins by decompressing and then reconstructs the original data from the differences. This approach can significantly improve the compression ratio for certain types of data.

The main drawback of FOR is that it is prone to outliers in the data frame. For example, for the frame $\{1,2,3,3,2,2,2,2,3,3,1,1,64,2,3,1,1\}$, if not the value 64 we could use the $\lceil log_2(3-1+1)\rceil = 2$ bits to encode the frame, but due to the outlier we have to use 6-bit encoding ($\lceil log_2(64-1+1)\rceil$) thus wasting 4 bits for each element.

Solution to the problem of outliers has been proposed in the work [1] in the modified version of the algorithm, called Patched FOR or PFOR. In this version of the algorithm outliers are stored as exceptions. PFOR first selects a new range mapping for the data in order to minimise the size of compressed data frames taking into account the space needed to store exceptions. Therefore, compressed block consists of two sections, within the first section the compressed data are kept and in the second exceptions are stored (encoded using 8, 16 or 32 bits). Unused slots for exceptions in the first section are used to hold the offset of the following exception 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 [1]. 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 of various solutions have been proposed, such as reducing the frame size [1], or algorithms that do not generate compulsive exceptions, such as Adaptive FOR [5] or modified version of PFOR [4]. Our publication is based on PFOR algorithm presented in [4]. Compressed block consists of three parts: the first in which compressed data are kept, second section where exceptions offsets are stored, and the last section which holds remainders of exceptions. When the outlier is processed, its position is preserved in an offset array, and the value divided into bits that are stored in the first section and the remainder to be retained in the third. Second and third section are then compressed using FOR(separately).

Decompression proceeds as follows: first decompresses the data section. Then decompresses the offset and exceptions array. Then iterates over the array offset, and restore the value of by applying patch from exception array.

2.2 GPU Implementation

To be able to decompress a data frame, decompression functions must be specified (for block of values, positions and supplements). In the case of the CPU one can use function pointers. On GPU this is dependent on the GPU computation capabilities (cc). For cc lower than the 2.x is not possible to use function pointers, which were introduced in version 2.x and higher. So far we used solution compatible with both architectures. In future work a version that uses features introduced in 2.x cards may simplify the code and make it more flexible.

Our current implementation is based on macros and requires information about the compressed data at compile time. This is not a serious limitation in the case of data that we analysed, as the optimum compression parameters were found to be constant within particular sets of data. Usual practice is to determine the optimal compression parameters based on sample data [1] and use them for the rest of the data set.

Another challenge was the issue of optimal reads of global memory during decompression: first decompression threads do not form coalesced reads leading to a drop in performance, and secondly the CUDA architecture can not cope well with the types of readings less than 32 bits in length. Our solution involves packaging data in texture memory, which solves the first problem. The second problem is solved by casting compressed data (char array) to array of integers. Decompression requires the reverse conversion. Texture, however, introduces some limitations, the maximum size is 2^{27} elements in case of one dimensional array. In future we plan to abolish the need for packaging data in the texture by modifying the compression and decompression algorithm. In this way readings of compressed data will allow for coalesced reads.

We prepared two versions of the experiments for GPU. In the first one we decompresses into the shared memory and in the second one into global memory. Algorithm based on global memory, as expected, is considerably slower than the algorithm based on shared memory (global memory is hundreds of times slower than shared), but this is still good choice for the considered experiment. On the other hand using shared memory is not always possible, so the algorithm based on global memory is still an alternative.

In the case of PFOR-DIFF algorithm we still need to perform reconstruction step, which involves iterating over decompressed buffer. In our opinion the best solution for this step, is to use properly implemented parallel prefix sum.

To decompress m (where m is a multiple of 8) values, we use $m/8$ threads. Each of the threads decompresses 8 values. Assuming that we have k exceptions (where k is a multiple of 8) we use the $k/8$ threads to decompress the offset and exception array. Then we patch 8 values as indicated in offset array using exceptions. This scheme is easily customisable when we have fewer threads. For performance reasons, we still need an array of 24 integers (safely located within threads registers) - ensuring best performance. After the decompression phase, array can be safely used for other purposes. For the convenience we have a macro that entered at the beginning of the kernel will prepare the appropriate code for uncompressing data, making use of decompression in other solutions easy.

We prepared uncompressing data code for both algorithms, ie. PFOR and PFOR-DIFF. In the experimental part we used the PFOR-DIFF algorithm as it is computationally more complex.

3 Preliminary Results

3.1 Experiment Results

Experiment Settings. In our experiment we used the following equipment: two Nvidia cards - Tesla C2050 / C2070 with 2687 MB and GeForce 2800 GTX with 896 MB (from CUDA Capability Major 2.0 and 1.3) - 2 x Six-Core processor AMD Opteron (tm) Processor with 31 GB RAM, Intel (R) RAID Controller RS2BL040 set in RAID5, 4 drives Seagate Constellation ES ST2000NM0011 2000 GB. We used the Linux operating system kernel 2.6.38-11 with the CUDA driver version 4.0.

Based on the observations that we made when analysis of example time series, we generated test data sets to ensure equal sizes of test data at different compression ratios. Each data set consists of 5% of outliers (which is consistent with the analysed data).

Experiments were conducted on different sizes of data ($2MB$, $4MB$, $10MB$, $50MB$, $100MB$, $250MB$, $500MB$, $750MB$, $1GB$). For each size, 10 iterations were performed and the results were averaged. In addition, to ensure the same disk read times for the same size of data we averaged disk read times for each data size. For the experiment with 1GB of data we present detailed results in table 1 which is visualised in figure 2. For the remaining data sizes we present average speed up in figures 3b and 3a.

Table 1. Measured times in seconds for 1GB of data. **Level** – compression level of input data. **IO** – time of disk IO operations. **Mem** – time of RAM to GPU Device data copying. **Computation** – algorithm computation time including data decompression.

Method type	Level	IO	Mem	Computation	Summarized
No compr. CPU	1	67.295	0.0	2.410	69.705
No compr. GPU	1	67.330	0.581	0.010	67.922
Compr. GPU shar.	2	33.448	0.286	0.085	33.821
Compr. GPU glob.	2	33.495	0.286	0.426	34.208
Compr. CPU	2	33.949	0.0	7.690	41.640
Compr. GPU shar.	4	16.778	0.104	0.083	16.966
Compr. GPU glob.	4	16.785	0.105	0.427	17.318
Compr. CPU	4	17.009	0.0	7.203	24.213
Comp. GPU shar.	6	11.192	0.095	0.082	11.369
Comp. GPU glob.	6	11.178	0.095	0.428	11.701
Comp. CPU	6	11.345	0.0	6.961	18.307

3.2 Discussion

Figure 2 presents final results of our experiments which were run on whole pattern matching algorithm. The bars are grouped by levels of compression, from 2 to 6, plus no compression. On the left we may compare efficiency of global and shared memory decompression. The right side shows CPU performance.

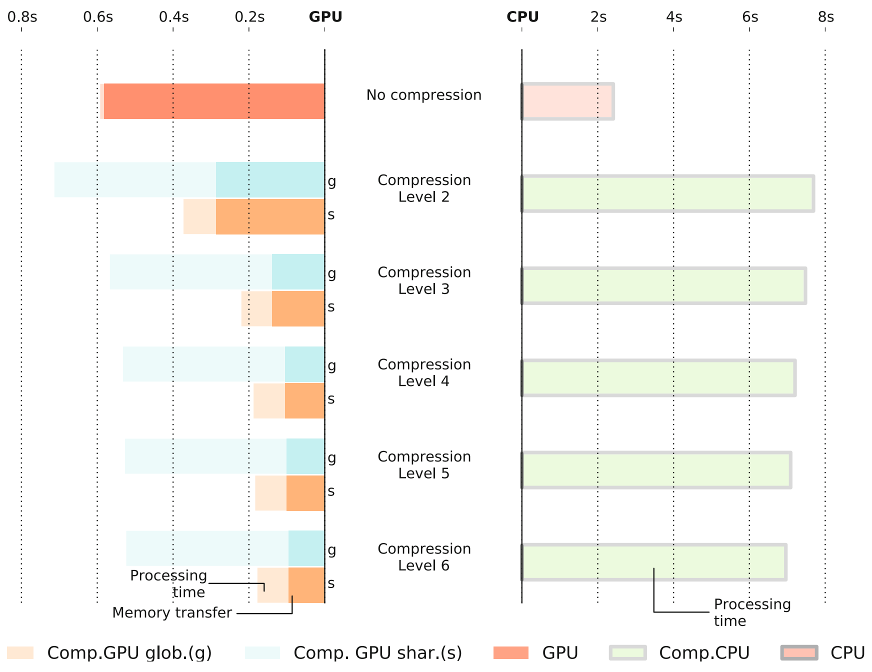


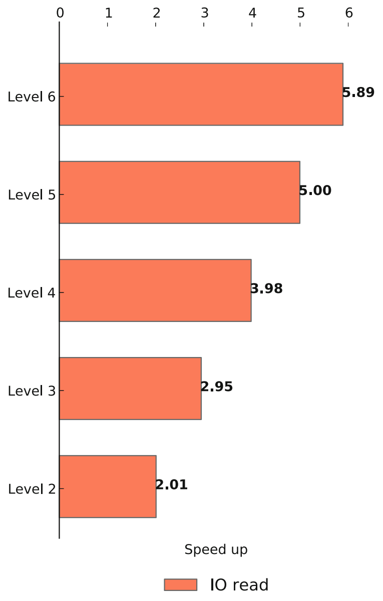
Fig. 2. Performance of a whole pattern matching algorithm with and without compression for 1GB of data, excluding IO times (lower is better). g – global memory decompression, s – shared memory decompression.

CPU performance is highly influenced by data decompression time. We can see that decompression takes from 4 to almost 6 seconds which is about 200% to 300% of the algorithm without decompression. The only possible improvement can be then observed on IO operations.

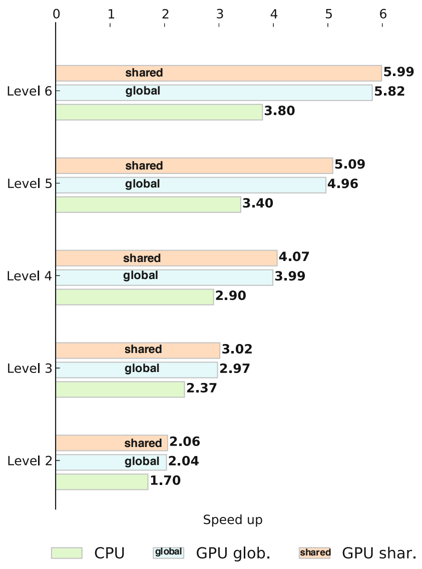
Thanks to ultra fast decompression GPU implementation of the algorithm performs much better. We can observe that for compression level 2 decompression in shared memory improved overall execution time by almost 2 times. For level 6 it is almost 3 times better. Decompression in global memory although also very fast can be slower for lower levels.

We must point out here that architecture of GPU shared memory limits possible number of algorithms which may use shared memory decompression method. Currently, subsequent kernel calls is the only method for global synchronisation. However, GPU cannot store shared memory state between kernel calls. As the consequence data decompression and algorithm execution must be done in single kernel. We are conscious that this may be improved by more complex kernels and data decompression strategy which will be addressed in our future research.

Figures 3a and 3b show interesting results concerning our shared memory decompression method. First we must notice that data compression alone increases application speed by reducing necessary data transfer. Fig. 3a demonstrates that pure IO operation speed up corresponds data compression ratio, which is an obvious result.



(a) Speed up on IO read time (higher is better)



(b) Computations speed up when using lightweight decompression compared to single threaded CPU version without decompression. This includes IO. (higher is better)

Fig. 3. Performance improvments when using lightweight compression

However, fig. 3b shows that this speed up may be significantly better if decompression is performed in shared memory. We achieved speed up improvement from 2% with compression ratio 2 up to 10% with compression ratio 6. These results prove our hypothesis that shared memory decompression increases efficiency of data intensive applications.

4 Conclusions and Future Work

In this paper we presented research devoted to improvement of GPU algorithms by utilisation of shared memory decompression. The hypothesis was evaluated and proved in many experiments. The contribution of our work may be summarised as follows.

We developed two highly optimised GPU parallel decompression methods: one dedicated for global and one for shared memory. We improved original algorithms by adding ability to deal with time series data, include negative values and fixed point values. We have no information about any other GPU implementations of lightweight compression with similar abilities.

We tested lightweight compression for time series and applied our novel methods to data intensive pattern matching mechanism. Our evaluation proved that the method significantly improved results when compared to other methods.

We showed that data decompression in GPU shared memory may generally improve performance of other data intensive applications due to ultra fast parallel decompression procedure since time saved by smaller data transfer is not wasted for decompression.

We analysed relationship between global and shared memory decompression showing that although shared memory may limit number of possible applications due to lack of global synchronisation mechanism, it significantly improves performance.

We plan to design a general purpose library which could be used in a similar way to CUDA Thrust library. Utilisation of common iterator pattern with memory blocks overlapping may offer interesting abilities breaking barrier of inter block threads communication. Such an iterator would require to define size of a shared memory buffer which would be common for more than one block in parallel threads execution. Description of such an extended iterator pattern for overlapping shared memory will be the next step of this research. During the preparation of this work, we managed to locate several problems of the current solution. In the future we also plan to prepare a new version of the algorithm which will allow for better use of CUDA 2.x computation capabilities and remove the constraints which we mentioned in the implementation description.

This work is a part of a larger project which aims to create a GPU based database for scientific data (such as time series, array, etc.).

References

1. Zukowski, M., Heman, S., Nes, N., Boncz, P.: Super-scalar ram-cpu cache compression. In: Proc. of the 22nd Intern. Conf. on Data Engineering, ICDE 2006, pp. 59–59. IEEE (2006)
2. Wu, L., Storus, M., Cross, D.: Cs315a: Final project cuda wuda shuda: Cuda compression project. Technical report. Stanford University (March 2009)
3. Kim, C., Chhugani, J., Satish, N., Sedlar, E., Nguyen, A.D., Kaldewey, T., Lee, V.W., Brandt, S.A., Dubey, P.: Fast: fast architecture sensitive tree search on modern cpus and gpus. In: Proc. of the 2010 Intern. Conf. on Management of Data, pp. 339–350. ACM (2010)
4. 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)
5. 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)
6. Harvard IIC. Data and search interface, time sries center (2012), `http://timemachine.iic.harvard.edu/`
7. Integral. Truefx (2012), `http://www.truefx.com/`
8. Hyndman, R.J.: Time series data library (2012), `http://robjhyndman.com/tsdl`
9. Goldberger, A.L., et al.: Physiobank, physiotoolkit, and physionet: Components of a new research resource for complex physiologic signals. Circulation 101(23), e215-e220
10. Goldstein, J., Ramakrishnan, R., Shaft, U.: Compressing relations and indexes. In: Proc. of the 14th Intern. Conf. on Data Engineering, pp. 370–379. IEEE (1998)