Soft Comput manuscript No.
(will be inserted by the editor)

A meta-heuristic approach for RLE compression in a column store table

Jane Jovanovski · Nino Arsov · Evgenija Stevanoska · Maja Siljanoska Simons · Goran Velinov

Received: date / Accepted: date

Abstract Structured data are one of the most important segments in the realm of big data analysis, that have undeniably prevailed over the years. In recent years, column-oriented design has become the most appropriate choice of structured data organization in analytical systems. These storage systems, usually referred to as column stores, organize data in a column-wise manner. More specifically, column-oriented databases or warehouses and spreadsheet applications have recently become a popular and a convenient tool for column-wise data processing and analysis. At the same time, the volume of data increases at an extreme rate, which despite of the decrease in pricing of storage systems, has stressed the importance of data compression. Aside from emphatic performance improvements in large read-mostly data repositories, column-oriented data are easily compressible, which enables efficient query processing and pushes the peak of overall performance. Many compression algorithms, including the Run-Length Encoding (RLE), exploit the similarity among the column values, where repetitions of the same value form columnar runs that can be found in most database systems.

This article presents a comprehensive analysis and comparison of common and well known meta-heuristics for columnar run minimization, based on standard implementations by using real datasets. We have analyzed Genetic Algorithms (GA), Simulated Annealing (SA), Particle Swarm Optimization (PSO) and Tabu Search (TS). The first two have been most efficient and have therefore undergone sensitivity analyses for parameters fine-tuning on synthetic datasets. After improving, these heuristics have been tested on initial real datasets. Experiments show that the algorithms perform consistently well on both synthetic and real data, demonstrating higher run-reduction efficiency compared to existing approaches. Moreover, the results show that the applied fine-tuned heuristics exhibit quick convergence to approximately optimal solutions, accompanied by insignificant overhead. In addition, we provide comprehensive implementations of the heuristic RLE compression approaches based on common optimization methods. They have proven to be effective at physical compression to an extent of being suitable as every-day appliances. The experiments on real datasets also indicate that our implementations overcome the expected on-disk file compression ratio, in most cases being better than the respective reduction of runs.

Keywords Column stores · RLE compression · Columnar runs · Run reduction · On-disk size compression · Meta-heuristic optimization

J. Jovanovski (✉) · N. Arsov · E. Stevanoska · M. Siljanoska Simons · G. Velinov
Faculty of Computer Science and Engineering,
Ss. Cyril and Methodius University,
Rugjer Boshkovikj 16, P.O. Box 393, 1000 Skopje
E-mail: jane.jovanovski@finki.ukim.mk

1 Introduction

In this article we leverage meta-heuristic methods for maximizing the rate of the RLE compression algorithm. We then apply it to data stored in a column-wise manner and use the results to implement a physical compression tool based on the meta-heuristic methods we have analyzed. The following subsections delve into our motivation and contribution.

1.1 Motivation

In this section we describe the motivation for the presented approach to addressing the problem of optimal compression: we state the reasons for focusing on column-oriented data

storage, the reasons for improving RLE compression in particular, we argue the appropriateness of meta-heuristic optimization methods for solving the problem, and lastly, we justify the need for such tools.

Column-oriented data storage. Traditional database systems use row-oriented data storage, that is, values from different columns of a record (row) are stored together. This row-based data organization enables high performance writes, which is especially suitable and beneficial for OLTP applications. However, it does not work well with systems, such as data warehouses, which are oriented towards ad-hoc querying of large amounts of data. In this case, better performance is achieved by using a column-wise data organization to facilitate complex queries over large data sets (Abadi et al 2008). Over the years, spreadsheet software has marked an increase in its usage for data analysis as a part of the OLAP toolchain, very often for analyzing data organized in a column-oriented manner.

The idea of storing data column-wise originated in the seventies through investigations on how to decompose records into smaller subrecords and store them in separate files (Hoffer and Severance 1975). In the eighties a fully decomposed storage model was devised where each column is stored in a separate file (Copeland and Khoshafian 1985). The development of MonetDB, a column store pioneer, began in the early nineties at CWI (Holsheimer and Kersten 1994). Sybase launched Sybase IQ, the first commercial columnar database system, in 1996, and then later followed Vertica, Exasol, Paraccel, InfoBright and SAND. More recently, Microsoft presented SQL Server 2012 as the first general-purpose database system to fully integrate column-wise storage and processing into the system (Larson et al 2012).

Database systems which organize data in a column-wise fashion are usually referred to as column stores. In a nutshell, in a column store each column is stored separately, i.e. values from a single column in different records are stored contiguously, typically densely packed, whereas the traditional database systems store entire records one after another (Abadi et al 2013). This column-based organization reduces the data processed by a query because the query reads only the columns it needs.

Data compression. Column-oriented data are highly amenable to compression, enabling column stores to optimize their storage space and utilize the storage optimization to improve their performance for a read-mostly query workload. Moreover, the extremity of the rate at which volumes of data increase in database systems has swollen the need of compression for efficient large scale data processing. Big data (see Marz and Warren 2015) has established a basis for compression-based optimization of so-called big data analytical stacks (Tang et al 2015).

Storing data by columns greatly increases the similarity between adjacent column values, which enhances the compressibility of the data (Abadi et al 2006). Many compression algorithms exploit this similarity by minimizing the number of columnar runs, that is, the number of repeats of the same value in each column. There are two general approaches to minimize the number of columnar runs. The first is minimization by row reordering, which refers to rearranging the rows in such a way that the number of columnar runs in the table is as minimal as possible. The second is minimization by table sorting, based on the idea that sorting the table improves compression, and permuting the columns in the right order before sorting can reduce the number of columnar runs by a factor of two or more.

Meta-heuristic optimization. The problem of determining an optimal row reordering and the problem of determining an optimal column sorting order are both known to be NP-hard (Lemire and Kaser 2011). Given that these problems are computationally difficult and cannot be solved efficiently in theory, they are clearly good candidates for applying meta-heuristic optimization methods. They have proven convenient and effective as an optimization technique for lossless data compression, especially when it comes to image compression. For instance, (Chang et al 2009) have relied on GAs to optimize data embedding in images and their subsequent compression, while (Iyoda et al 2007) have exploited GAs as a first stage in optimization combined with unsupervised learning techniques. (Yimin et al 1999) generate optimal coding schemes for digital image compression using GAs. By virtue of the importance of data compression and the complexity of the quest to find an optimal compression, other methods have been developed or used for lossless data compression. For example, PSO in (Tan et al 2014) and self-configuration single particle optimizer (SCSPO) in (Ji et al 2013) are used for lossless DNA sequence compression.

Scarcity of compression tools. A further motivation for our work presented here is the somewhat obvious scarcity of convenient compression tools that enable approximately optimal compression in **reasonable time**. Existing algorithms, such as heuristic column reordering and sorting, are fast, but certainly not optimal. On the other hand, algorithms involving combined row and column reordering are complex and slow, while their effect on improving compression is negligible compared to simple heuristics. Thus, both variations are seemingly impractical and unsuitable for real systems where the preprocessing time is crucial.

1.2 Contributions

In this article we take advantage of meta-heuristic optimization methods concerning compression of structured data, organized column-wise, which is eventually conducive to RLE

compression. We address the problem of minimizing the number of columnar runs by sorting a column store table in an optimal column sorting order and its corresponding RLE-based table compression. Abadi et al (2006) recommended solving the problem by lexicographic sorting with "low cardinality columns serving as the leftmost sort orders". Lemire and Kaser (2011) justified this empirical recommendation and proved that the heuristic gives good results. However, often a meta-heuristic might lead to better results. In our previous work we developed a GA for solving the given problem and demonstrated that this approach exhibits improved run-reduction efficiency up to a factor of 4 compared to the reduction achieved by the given heuristic (Jovanovski et al 2013).

We have implemented four meta-heuristic optimization methods by using standard and recommended parameters and adjustments for combinatorial optimization. Then, the meta-heuristics have been compared against seven representative datasets. Using the quality of the solution, but more importantly, the speed of convergence to a nearly optimal solution, we have chosen two methods - GA and SA.

After they had undergone parameter sensitivity analysis by using numerous synthetic datasets (differing by distribution or by the number of rows and columns), GA and SA were fine-tuned and re-adjusted to use parameters that are most suitable for these types of problems. The re-adjusted meta-heuristic methods were then applied to real datasets, where they demonstrated improvement compared to standard parameters. The measurements show that the total improvement that these compression algorithms demonstrate goes up to 75% reduction of runs with respect to the initial solution, while the heuristic demonstrates consistently lower improvements randing from 1% and up to 75% less.

We also provide an implementation of meta-heuristic RLE compression based on the two most effective fine-tuned meta-heuristic optimization methods. We show that by taking into account various storage format optimizations, it yields a significantly better compression ratio than the run-reduction ratio itself, demonstrating up to 90% compression, i.e. on-disk size reduction.

Finally, a comprehensive contribution of our approach is the reasonable time which the meta-heuristic methods take to find a nearly optimal column and sorting order that results in a nearly optimal RLE compression. Our meta-heuristic optimization methods converge reasonably fast and demonstrate sound compression ratios on both synthetic and realistic datasets. In comparison, some of the much more complex algorithms surpass our solutions at a negligible rate of quality, while being much slower makes them applicable to only a limited number of scenarios. Therefore, our approach is a highly acceptable trade-off between the speed of convergence and the solution quality, making them a sound "off-the-shelf" tool applicable in a wide variety of real scenarios.

1.3 Problem definition

One of the primary advantages of column stores is data compression which helps to reduce storage space as well as I/O times. An attractive approach for compressing sorted data in a column store is the RLE where repeats (runs) of the same value are stored as a single data value and count, rather than as the original run. RLE performs lossless data compression, that is, it allows the exact original data to be reconstructed from the compressed data. In column stores, RLE speeds up many queries: sum, average, median, percentile, and arithmetic operations over several columns (O'Neil and Quass 1997).

Columns in column stores can be compressed by using the repetition and similarity among values within a column, where sequences of adjacent column cells with identical values form columnar runs. Each columnar run is stored as a RLE pair (value, run-length). However, accessing a random position in this kind of data structure requires an $O(r)$ operation for a column with r runs. Therefore, it might be useful to code some redundant data, such as storing the starting and/or the ending position of the run, and store each columnar run as a RLE triple (value, start-position, run-length) or (value, start-position, end-position). This data structure allows to binary search for a particular position, improving the search bounds to $O(log_2(r))$. Storing the columnar runs as RLE triples ensures preservation of the original row structure and eases the process of table reconstruction.

Obviously, the smaller the number of columnar runs in a table, the bigger the RLE-based table compression would be. This implies that the number of columnar runs can be used as a general model for RLE-based table compression, i.e. the problem of RLE-based compression of column store tables can be viewed as the problem of minimizing the number of columnar runs defined as follows: Given a column store table with M columns, find the optimal column sorting order which minimizes the total number of columnar runs $\sum_{i=1}^{M} r_i$, where r_i is the number of columnar runs in the i-th column, $i = 1, 2, ..., M$.

For example, Table 1 represents the table Customer whose total number of columnar runs is 23, however this number is not optimal. Table 2 shows the same table after sorting its columns in one possible optimal order when the number of columnar runs is minimized and is 15, whereas Table 3 displays the corresponding RLE-based compression of the table.

This approach enables usage of RLE-based compression for data with large runs of repeated values, which typically occur on sorted columns with a few number of distinct values. Unlike other compression approaches where additional data structures are needed for reconstructing the original ta-

Table 1 The original Customer table (before sorting)

Row	1	2	3	4
Col	Name	City	Color	Gender
1	Nicholas	Amsterdam	Blue	M
2	Margaret	Zurich	Purple	F
3	Tomas	Rome	Blue	M
4	Nicholas	London	White	M
5	Christian	Zurich	Blue	F
6	Tomas	Amsterdam	White	M

Table 2 The Customer table after sorting in an optimal column order

Row	1	2	3	4
Col	Name	City	Color	Gender
1	Nicholas	Amsterdam	Blue	M
2	Nicholas	London	White	M
3	Tomas	Amsterdam	White	M
4	Tomas	Rome	Blue	M
5	Christian	Zurich	Blue	F
6	Margaret	Zurich	Purple	F

Table 3 The Customer table compressed using RLE-based compression

Row	1	2	3	4
Col	Name	City	Color	Gender
1	Nicholas[1,2]	Amsterdam[1,1]	Blue[1,1]	M[1,4]
2	Tomas[3,4]	London[2,2]	White[2,3]	F[5,6]
3	Christian[5,5]	Amsterdam[3,3]	Blue[4,5]	
4	Margaret[6,6]	Rome[4,4]	Purple[6,6]	
5		Zurich[5,6]		
6				

ble rows, here only the original row order is not preserved and thus only one extra meta column is needed to store it.

1.4 Structural organization of the article

The article is organized as follows. Sect. **??** describes the design and implementation details of the proposed GA. Next, in Sect. 3 we analyze thoroughly the practical implementation of the RLE compression based on the GA and its additional optimization regarding actual size on disk after a column-store table has been compressed. In Sect. 4 we present the experiments performed both on synthetic data and realistic datasets and analyze the obtained results in terms of run-reduction efficiency and on-disk file compression ratio. Finally, we give some concluding remarks in Sect. 5.

2 Meta-heuristic optimization algorithms

We implement four optimization algorithms: GA, TS, SA and PSO. They all start from a random solution (or a set of random solutions) and use different methods of searching the neighbors in order to generate a near optimal solution. How 'good' a solution is, is determined by the value of the fitness function.

2.1 Common implementation details of the algorithms

In the following subsections the details of the implementations that are common for all four optimization algorithms used are described. These include the representation of the solution domain, the fitness function and the initialization of the initial solution(s).

2.1.1 Representation of the solution domain

The representation of the solution domain is problem dependent and should be chosen in a way that respects the structure of the search space. Each solution is represented through an abstract class, called chromosome (for GA), particle (for PSO) or just solution(for SA and TS).

From a computational point of view, the problem of finding an optimal column sorting order for a table with M columns consists of determining the particular permutation of the non-repeating sequence $1,2,...,M$ representing the order in which the columns are sorted such that the total number of columnar runs in the table is minimized. Each permutation element is assigned a flag value to represent the order of sorting (ascending, descending or none).

Therefore, the class that represents candidate solution of the problem under consideration consist of an ordered set $\{(c_i,s_i)\,|i=1,2,...,M\}$, where $\{c_1,c_2,...,c_M\}$ is a permutation on the set $\{1,2,...,M\}$ which represents the order in which the M columns of the table are sorted, and $\{s_1,s_2,...,s_M\}$ is a permutation with repetition of M elements on the set $\{0,1,2\}$ where $s_i=0$ denotes that the column c_i is not sorted, while $s_i=1$ and $s_i=2$ denote that the column c_i is sorted in ascending and descending order, respectively. Hence the search space of the given problem contains $M!3^M$ possible chromosomes, where $M!$ is the number of permutations of M elements and 3^M is the number of permutations with repetition of M elements on a set of 3 distinct elements.

For example, consider the following chromosome $\{(4,2),(1,1),(2,1),(3,0)\}$ which represents one optimal column sorting order for the table Customer, as shown in Table 2. Initially the sorting is done by the 4th column 'Gender' in descending order; if some table rows contain the same values for 'Gender', then they are sorted by the column 1 'Name' in ascending order; finally the rows who have same values for 'Name' are sorted by their values in the 2nd column 'City' in ascending order. The last gene of the chromosome indicates that the 3rd column 'Color' should not be considered for sorting. This example corresponds to the

results of the following SQL ORDER BY clause, which is used in a SELECT statement to sort results either in ascending or descending order:

```
SELECT *
FROM Customer
ORDER BY Gender DESC,
         Name ASC,
         City ASC
```

2.1.2 Fitness function

Every visited solution by the optimization algorithms is evaluated and assigned, by means of a fitness function, a measure of its goodness with respect to the given problem. The algorithms use this fitness value as the quantitative information to guide the search.

The fitness of a solution is the value of an objective function for its phenotype. In the given problem, the objective function is the total number of columnar runs in a table. If the table has M columns, the fitness function is defined as:

$$f(x) = \sum_{i=1}^{M} r_i(x) \ ,$$

where $r_i(x)$ gives the number of runs of identical values in the i-th column for a column sorting order corresponding to a solution x, for $i = 1,2,\ldots,M$.

2.1.3 Initial solutions

Any optimization algorithm starts with one or set of solutions. The starting solution(s) should be as diverse as possible, in order to enable wide exploration of the problem's search space. In order to achieve this, the initial population in most cases is chosen randomly.

One solution in our implementation is randomly generated as follows: a random permutation of M elements (corresponding to the indices of the M columns of the table) is generated using the Fisher Yates shuffle algorithm (**?**),where any permutation of M elements will be produced with probability of exactly $1/M!$, thus yielding a uniform distribution over all such permutations. Then, for each of the elements in the permutation a randomly selected element from the set $\{0,1,2\}$ is assigned.

We want to ensure that the initial population's best fitness value is at least as good as the fitness value of the original table and that no good results are being lost, so in the initial population used by PSO and GA we add the solution $\{(1,0),(2,0),...,(M,0)\}$ formed by the identity permutation which corresponds to the original table. In the multiple runs of TS and SA algorithms we make sure that one run starts with the solution which represents the original table.

.

2.2 The genetic algorithm

GA is a common heuristic optimization method based on the principle of natural evolution. It encodes a potential solution to a specific problem on a simple chromosome-like data structure and applies recombination operators to these structures so as to preserve critical information. An implementation of a GA begins with an initial population of (typically random) chromosomes. Each chromosome is evaluated using a fitness function that is specific to the problem being solved. Then, based upon the fitness values, the GA allocates reproductive opportunities in such a way that the chromosomes which represent a better solution are given more chances to reproduce than the chromosomes that represent poorer solutions.

The following subsections describe the parameters and the design details of the proposed GA for determining an optimal sorting order which minimizes the number of columnar runs in a given table. The GA approach combines permutation-based chromosome representation, problem-specific heuristic for the initialization process and traditional generational GA with overlapping populations.

2.2.1 Genetic operators

Every GA uses a selection mechanism to decide which individuals will comprise the mating pool and will be used to generate new offspring which will form the basis of the next generation. When dealing with a problem of minimization, the individuals with lower fitness values will have a better chance to be selected for the mating pool, whereas the individuals with high fitness values will be more likely to disappear (Lee and El-Sharkawi 2008). Once the mating pool is complete, the GA proceeds with the reproduction phase by applying crossover. Crossover is the process of randomly selecting two parent chromosomes from the mating pool, exchanging genetic material between the parents and producing new offspring chromosomes with the hope that the new chromosomes will inherit good features from their parents and therefore will be better than the parent chromosomes. This tends to increase the quality of the populations and force convergence. After crossover, the offspring chromosomes are subjected to mutation. Mutation allows for random modification of the genetic material. Applying mutation helps maintain genetic diversity in the population and prevents the GA to be trapped in a local minimum, hence it plays an important role in any GA. Replacement is the last stage of any breeding cycle in a GA. In the reproduction phase, two parents are drawn from the mating pool and they breed two children via crossover and mutation, but not all four chromosomes can survive to the next generation. Therefore, a replacement method needs to be chosen to determine which individuals will form the next generation.

The design choices for the genetic operators used in our GA are briefly described below.

Selection. The selection of individuals for the mating pool is performed by exploiting the tournament selection method, implemented by holding a tournament between $k \geq 2$ randomly chosen individuals. We made this choice because this algorithm is computationally more efficient (no sorting is required) and more amenable to parallel implementation compared to other selection algorithms (Mitchell 1998).

Crossover. There is a variety of crossover operators designed specifically for permutation based chromosomes, a good overview is given by Bäck et al (2000). Our GA implements the three most simple and commonly used crossover operators which ensure permutation chromosome feasibility: Cycle Crossover (CX), Order Crossover (OX) and Partially Matched Crossover (PMX) (Sivanandam and Deepa 2008; Lee and El-Sharkawi 2008). CX performs a recombination under the constraint that each gene comes from one of the parents. OX constructs an offspring by choosing a subsequence of one parent and preserving the relative order of the genes of the other parent, while PMX maps a portion of one parent's genes into a portion of the other parent's genes and exchanges the remaining information. Crossover is applied with a probability P_c, hence for a population of K_{pop} chromosomes, $\lceil P_c K_{pop} \rceil$ offspring chromosomes will be produced with crossover.

Mutation. An important parameter in the mutation technique is the mutation probability P_m which determines how often the offspring chromosomes will be mutated. If there is no mutation, offspring are generated immediately after crossover without any change. If mutation is performed, one or more genes of a chromosome are changed. If the population size is K_{pop} and the mutation probability is P_m, then $\lfloor P_m K_{pop} \rfloor$ offspring chromosomes will be subject of mutation. We implemented three mutation operators that preserve the ordering property of permutation chromosomes: Insertion Mutation (IM), Simple Inversion Mutation (SIM) and Swap Mutation (SM) (Sivanandam and Deepa 2008; Lee and El-Sharkawi 2008). IM removes a randomly chosen gene from the chromosome and reinserts it in a randomly selected location in the chromosome. SIM reverses the chromosome section between two randomly chosen cut points, while SM selects randomly two genes in the chromosome and exchanges (swaps) their content corresponding to the column permutation numbers.

Replacement. We used a traditional generational GA with overlapping populations, that is, an algorithm which uses generational updates of the population as well as the elitism strategy to improve the performance. This means that the best individuals of the parent population are retained in the new population, while the rest of the individuals of the parent population are completely replaced by the newly produced offspring chromosomes. Namely, $\lfloor (1-P_c) K_{pop} \rfloor$ of the best individuals of the parent population are retained in the new population, while the remaining $\lceil P_c K_{pop} \rceil$ chromosomes in the new population are produced by crossover from chromosomes in the parent population. This ensures that the best individuals from each population are not lost or destroyed (Mitchell 1998) and that the best solution is monotonically improving from one generation to the next, which enhances significantly the algorithm's performance. The potential downside is a premature convergence, but that can be overcome by an appropriate amount of mutation.

2.3 Particle swarm optimization

Similar to GA, PSO starts with initial set of feasible solutions. Each of these solutions is called a particle. At the beginning, these particles are randomly placed in the search space, and each particle evaluates the fitness function at its current location. In the next iterations, the particles "fly" through the solution domain with some velocity, in order to get closer to the good solutions found so far.

Let n be the dimension of the search space. Then, a particle is defined with three n-dimensional vectors: the current solution X,the best solution it found so far $\mathbf{P_p}$ and its velocity $\mathbf{V}$. PSO also keeps track of the best solution found by all particles so far, $\mathbf{P_n}$. The initial vectors X in the first iteration are randomly generated, in the way described in 2.1.3. The velocity V determines the amount of change each particle makes in one move, and can be seen as a step size. At the beginning, the initial velocity of each particle is a random number between V_{min} and V_{max}. In each iteration, the particles update the X and V vectors. The update of the velocity is defined by the following equation: $\mathbf{V_{new}} = \omega * \mathbf{V_{old}} + \eta_1 * rand() * (\mathbf{P_p} - \mathbf{X}) + \eta_2 * rand() * (\mathbf{P_n} - \mathbf{X})$ and consist of three parts: momentum, cognitive component and social component. The momentum (defined with the term $\omega * \mathbf{V_{old}}$ is the tendency of a particle to keep its previous direction. The cognitive component (defined with $\eta_1 * rand() * (\mathbf{P_p} - \mathbf{X})$) is the tendency to return to its previous best solution, and the social component ($\eta_2 * rand() * (\mathbf{P_n} - \mathbf{X})$) stands for the tendency to go to the best position found by all particles. The values ω, η_1 and η_1 are just the weights assigned to this three components. The $\mathbf{X}$ vector is updated by adding $\mathbf{V}$ to $\mathbf{X}$. This is represented with the following equation:$\mathbf{X_{new}} = \mathbf{X} + \mathbf{V_{new}}$. $\mathbf{P_p}$ and $\mathbf{P_n}$ are also updated if the new $\mathbf{X}$ is better than the previously found $\mathbf{P_p}$ and $\mathbf{P_n}$ solutions.

The previous paragraph describes the general version of PSO. But not all aspects of it are applicable in our implementation. In particular, in our problem the order of the columns is represented by a permutation of n numbers, where n is the number of columns. Therefore, the principle updating the $\mathbf{X}$ vector is not directly applicable. This principle assumes that

the elements of the solution and velocity are independent of each other, and the updates in the particle's velocity and its current solution are updated in the same manner. But with this procedure it is possible that after the update two elements of the list can take the same values. So, we need to find a way to deal the specificity of our representation of the solution.

2.3.1 Velocity

As stated above, the general equation for updating the X vector and the velocity are not directly applicable in our case, because it can lead to a solution that is not a permutation of n numbers. Therefore, it requires a new approach for updating $\mathbf{V}$. The new principle should follow the general idea of the velocity (step size): the higher the velocity of the particle, the more a particle is allowed to move and explore the search space. It also should contain the three parts of the velocity equation: momentum, cognitive component and social component.

In our implementation, instead of vector, the velocity and all of its components are floating-point values. At the end, we round the new found velocity to the nearest integer, and interpret this number as the number of changes (mutations) we make to the current particle's solutions. The equations that represent this concept for the new velocity are: $\mathbf{V_{new}} = \omega * \mathbf{V_{old}} + \eta_1 * \mathbf{V_{pop}} + \eta_2 * \mathbf{V_p}$ where $\mathbf{V_{pop}} = (1 - \mathbf{Val_n}/\mathbf{Val_{par}}) * \mathbf{V_{max}}$ and $\mathbf{V_p} = (1 - \mathbf{Val_p}/\mathbf{Val_{par}}) * \mathbf{V_{max}}$ where $\mathbf{Val_n} = C(\mathbf{P_n})$, $\mathbf{Val_{par}} = C(\mathbf{X})$ and $\mathbf{Val_p} = C(\mathbf{P_p})$. C is the fitness function used to evaluate the solutions, $\mathbf{Val_n}$, $\mathbf{Val_p}$, $\mathbf{Val_p}$ are the values of the fitness function for the best solution found by all particles, the best solution found by the particle which velocity we calculate and the current solution of the particle respectively. The velocity in our implementation is constrained by $\mathbf{V_{max}}$ and $\mathbf{V_{min}}$ parameters.

2.3.2 Neighborhood

In each iteration, we apply $\mathbf{V_{new}}$ mutations to each particle in order to get to its neighbors, where $\mathbf{V_{new}}$ is the new velocity of that particle. It is clear that $\mathbf{V_{new}}$ should be a positive integer. A change is defined as follows: first, we decide in which direction are we going to move (with probability p_1 we are moving towards the best solution of the current particle($\mathbf{P_p}$), and with probability $(1 - p_1)$ towards the best solution found by the all particles ($\mathbf{P_n}$)).(Note: we make all the changes based on the order array. The values in the flag array are updated simultaneously.) Then, we choose a random index i, and at that position at particle's current order vector we put the value that is on the same position in the best chosen solution. (Let that value be $best_i$). This provides us an order array which contains the $best_i$ element twice. Next, we want to find the original position where $best_i$ was

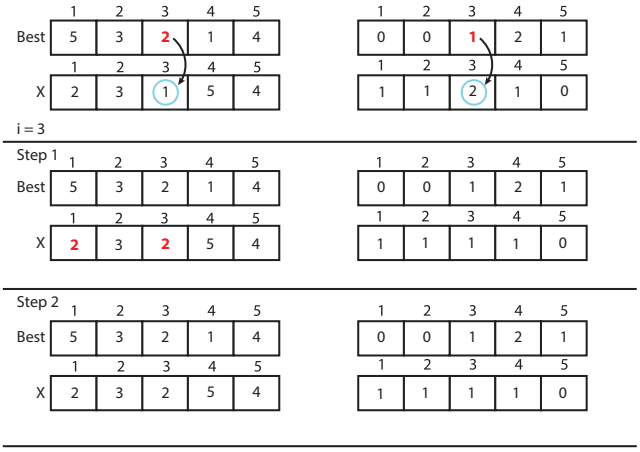

Fig. 1 Valid movement in the search space by PSO

stored, and put in that cell the value that was at the position i at the original array. The flag vector is also updated; the sorting flag for the $best_i$ column is copied from the best array.

The process is illustrated on Fig. 1. Let the chosen direction be towards the best solution found by that particle (*Best*) and $\mathbf{X}$ the particles current solution. We choose $i = 3$. In the first step, 2 is copied to the third position in the Order vector of the current solution, and 1 is copied to the third position in the flag vector. The order vector of X is not a valid permutation. We search for the index where value 2 was originally stored in the vector (position 1), and replace this value by the value that we replaced on the third position ($value = 1$).

This definition of the neighbors has one significant drawback: when a particle finds a solution that is better than the last $\mathbf{P_n}$, it will stay in this position forever. We introduce another form of mutation to avoid this behavior. Namely, when one particle finds a solution that is identical to the best one found so far, we mutate it as follows: first, we choose two random indexes and swap the elements on those indexes on the flag and order arrays. Then, we increase (modulo three) the element in the flag array at the position determined by the first index, and decrease (modulo three) the element at the second index.

2.4 Simulated annealing

The SA method follows principles of thermodynamics which apply when an alloy is heated and then slowly cooled down. The atoms first move a lot and then gradually settle into a low energy state, which allows them to find a low energy configuration.

2.4.1 Annealing schedule

In the SA method, we start from a single solution. In each iteration, we visit one of the current solution neighbors, and

evaluate it with the fitness function. Let S be the current solution, and S' its chosen neighbor. We denote by Δ the difference between the cost functions: $\Delta = C(S) - C(S')$.

If the new solution gives a better value for the fitness function ($\Delta < 0$ in the minimization problem, or $\Delta > 0$ in a maximization problem), it is accepted, and S' replaces the current solution S in the next iteration. But the main difference between SA and other optimization methods is that with SA a worse solution still has a chance to be accepted with a probability p. We use the following equation to calculate p: $p = \exp(-\Delta/T)$, where T is a control parameter called temperature), bounded by two input parameters T_{max} and T_{min}. The idea is to allow the algorithm to escape local optima. Sometimes it is necessary to move to a worse solution, in order to get to the better one.

In the beginning we are willing to let the algorithm be more exploring, but in the later iterations the probability of moving to a worse solution should be lower. We use the temperature parameter (T) to simulate this behavior. The probability of moving to a worse solution is directly proportional to the temperature T. Therefore, in the beginning (in the first iteration) the temperature starts with some high value T_{max}. In each step further, the temperature becomes gradually lower, thus lowering the probability of accepting a worse solution. The change in the temperature is defined by a cooling function (originally suggested in (Kirkpatrick et al 1983)). Since then, there are many different cooling function suggested. In our implementation, we define the temperature at step k as

$$T_k = T_{max} * \exp(T_{factor} * k/steps),$$

where $T_{factor} = -\log(T_{max}/T_{min})$.

In our implementation the parameter $steps$, T_{max} and T_{min}, together with the choice of cooling function completely define the behavior of the optimization, and are called annealing schedule.

2.4.2 Neighbors

Here a neighborhood of the solution S contains all the solutions that can be obtained by performing one of the two simple moves on S:

1. Choosing two random indexes, and swapping the elements on those position in the order and flag list
2. Increasing (modulo three) one element from the flag list.

2.5 Tabu search

TS is a heuristic optimization technique which uses adaptive memory in order to escape local optima. Namely, tabu search keeps a list of *moves* (mutations) that can not be performed on the current solution, called *forbidden moves*. The length of the tabu list (n) and the numbers of neighbors visited in each iteration (k), together with the choice of a structure that represents a forbidden move, are the only parameters that TS uses, and that makes it one of the most popular and easiest to implement heuristic optimization algorithms.

The search begins with one solution. The neighborhood has the same definition as the neighborhood we use in the SA optimization: we can either swap two elements in the order list (and corresponding elements in the flag list), or increase (modulo three) an element in the flag list. In each iteration, TS visits k neighbors of the current solution, and chooses one as a current solution for the next iteration. A solution is chosen if it has the best value for the fitness function, and it contains no tabu moves. If all visited solutions contain tabu elements we choose the one with the best fitness function.

In our implementation we keep two tabu lists that are updated after every iteration; the first one contains the last n swaps we have made, and the second stores the n last increments of the columns flags. If the newly chosen neighbor is constructed with the swap of two elements, we add two tuples in the "swap" list: $(idx_1, elem_1)$ and $(idx_2, elem_2)$. The idx_1 and idx_2 are the positions we choose for the swap, and $elem_1$ and $elem_2$ are the elements on those positions in the new solution (the chosen neighbor). If the new neighbor is constructed with incrementing an element in the flag array, in the tabu list we add the tuple $(index, flag)$. Index is the position where the flag was changed, and flag is the new flag element on that position.

When we check if a solution is tabu, we enumerate all the elements of the order list, and form list of tuples $(idx, elem)$. If at least one of the tuples is contained in the swap list, the solution is tabu. The similar logic is applied for checking tabu solutions based on the flag tabu list.

2.6 Complexity analysis

The complexity of the proposed meta-heuristic algorithms for a column-store table with M columns and N rows can be analyzed as follows:

Genetic Algorithm. Assuming that an initial population of K_{pop} chromosomes has been generated, the fitness function $f(x)$ is subject to minimization throughout G generations of the initial population produced by applying the described genetic operators. For each chromosome of the population, in each generation, the fitness of the chromosome is calculated by sorting the column-store table and then counting the columnar runs in the sorted table. Therefore, the complexity of the GA is predominantly influenced by two operations: The first operation takes $O(MN \log N)$ time. The second one runs in $O(MN)$ time, which represents the total number of field comparisons for calculating the value of the fitness function. However, the runs counting step is domi-

nated by the sorting step. The overall complexity of our GA is thus asymptotically estimated to $O(K_{pop}GMN\log N)$.

Particle Swarm Optimization. Let $K_{particles}$ be the number of particles in the initial pool of solutions, and I the maximum number of iterations. Same as before, the complexity of PSO is influenced by two parts: the sorting of the table ($O(MN\log N)$) and the movement of the particles $O(M)$. We estimate the overall complexity of the PSO to be $O(K_{particles}IMN\log N)$, which is very similar to the complexity of the GA.

Simulated Annealing. Let S be the number of steps performed by the SA. The movement from the current solution S to its neighbor S' is done in a constant time. Following the same thoughts as before, we estimate the complexity by $O(SMN\log N)$.

Tabu search. Let I be the number of iterations in TS, K number of visited neighbors in each iteration and T the length of the tabu lists. Same as in the SA, the movement to e neighbor is done in a constant time, and the sorting of the table is estimated by $O(MN\log N)$. Here we should also consider the complexity of the check if a certain solution is tabu. That is done in a $O(T)$, time, which is dominated by the complexity of sorting the table. The overall complexity is estimated by $O(IKMN\log N)$.

Existing heuristics. Related research in the area of minimizing the number of columnar runs in a table (Lemire et al 2012) suggests different methods for reordering the rows beyond the lexicographic order. Namely, heuristics such as the Multiple Lists heuristic, whose running time complexity is shown to be $O(c^2n+cn\log(nN_1N_2\ldots N_{c-1}))$, where N_i denotes the number of lists used for the i-th column (one list for each value in the first column) in each rotation of the columns. Equivalent to the notation used in this article, the complexity is $O(M^2N+MN\log(NN_1N_2\ldots N_{c-1}))$. Therefore, our optimization is a trade-off between time and compression ratio compared to the Multiple Lists heuristic. The parameters for the size of the initial population and number of iterations should be chosen to be relatively small, but big enough to produce diversity in each generation and allow a solution with a satisfactory value of the fitness function to be chosen in the final stage.

3 RLE implementation

We have developed two different versions of RLE compression based on our GA over data stored in a tabular manner: a Libre Office (further denoted as LO) extension implemented in Java and a considerably faster C++ application operable through the command line interface.

The Libre Office extension was developed using the *Libre Office 4.1 API*. In order to handle large amounts of data which is difficult to store in the main memory, the extension uses external memory resources instead. External memory resources refer to implementation techniques where instead of reading the whole input data into the main memory, data is stored on a physical disk and is being accessed and updated directly during run-time. External memory resources also refer to external sorting and storing supplementary data structures in memory-mapped files on a physical disk. Binary-indexed trees (also known as Fenwick trees) (Fenwick 1994) are stored in a file on the disk. Fenwick trees are used to index cells in tabular data and then quickly compute the length of a columnar run using external binary search in order to reduce the number of accesses to a physical disk.

The LO Java extension supports files containing up to a million rows due to the LO restrictions, but it provides minimal main memory consumption as a result of external memory resources. The default Open Document Spreadsheet format is represented by an archive containing a large quantity of XML tagging for a single spreadsheet and as such keeps a large amount of meta-data, which suppress the RLE effects. Therefore this extension supports raw tabular data instead, e.g. comma-separated values.

On the other hand, the C++ application does not use external memory resources. It runs entirely in the main memory and stores supplementary data structures there. This technique provides greater time performances due to minimal number of physical disk accesses (usually a single access to read the data into memory). It compresses an input file in a thread-per-column manner, where a separate thread is instantiated for each column, reducing time complexity by a factor equal to the number of columns, M. Although the C++ application does not use external memory resources, its memory consumption can be decreased by taking advantage of the Boost Flyweight data structure, provided by the Boost C++ Library. The C++ extension is compatible with both compact and immense data exceeding a few hundreds of millions of rows as long as it does not exceed the available RAM memory in a 64-bit system.

The source code for both of our implementations can be found at the following Git Hub repositories: the Genetic Algorithm for RLE compression in C++ (Arsov et al 2014a) and the Genetic Algorithm for RLE compression in Java (LO Extension) (Arsov et al 2014b). They are subject to the public Apache License v2 and are completely open to the public within the limitations of this license.

3.1 Storage format

After reordering the columns and sorting the data with respect to the outputs of the proposed GA, RLE is subsequently applied over reordered data. This ”squeezes” the data by eliminating duplicate cell values which appear sequentially in a single column. Whereas the LO Java exten-

Table 4 The Customer table after sorting in an optimal column order with the columnar runs $c_{i,j}$ consecutively enumerated

Row	1	2	3	4
Col	Name	City	Color	Gender
1	Nicholas_1	Amsterdam_5	Blue_{10}	M_{14}
2	Nicholas	London_6	White_{11}	M
3	Tomas_2	Amsterdam_7	White	M
4	Tomas	Rome_8	Blue_{12}	M
5	Christian_3	Zurich_9	Blue	F_{15}
6	Margaret_4	Zurich	Purple_{13}	F

sion generates a single compressed file, the C++ application using the thread-per-column technique results with a different storage format - RLE-compressed data in each column is stored in a single file on the disk.

A brief description of the storage format specified by the LO Java extension is given below.

Let $c_{i,j}$ denote the i-th columnar run in the j-th column. The data value which occurs in consecutive cells of the run $c_{i,j}$ is labeled by $v_{i,j}$. Let $L_{i,j}$ be the number of consecutive occurrences of a data value $v_{i,j}$ in a columnar run $c_{i,j}$, i.e. the length of run $c_{i,j}$. For example, in Table 4 the second run in the first column, $c_{2,1}$, corresponds to the two consecutive occurrences of 'Tomas'. This run is represented by the data value $v_{2,1}$ = 'Tomas'. Accordingly, the length $L_{2,1} = 2$.

After applying RLE over a single columnar run $c_{i,j}$, this run is stored in a simple format, $v_{i,j}\Delta L_{i,j}$, where Δ is a single byte delimiter. As this storage format suggests, storing a compressed columnar run produces an invariable overhead added to each distinct cell-value representative of a columnar run. Let λ denote a size measure in bytes. In terms of columnar runs, $\lambda_{c_{i,j}}$ represents the required number of bytes to represent the length $L_{i,j}$ of the run. In terms of data values, $\lambda_{v_{i,j}}$ represents the size of $v_{i,j}$ measured in bytes, t.e its length. Obviously, $\lambda_{\Delta} = 1$ for the reason that it is represented by a single ASCII character. So the overhead $\omega_{c_{i,j}}$ produced over a single run is equal to:

$$\omega_{c_{i,j}} = \lambda_{c_{i,j}} + \lambda_{\Delta}.$$

In most computer architectures, four bytes are often used to represent an unsigned integer, so $\lambda_{c_{i,j}} = 4$. Then the total overhead produced over input data consisting of M columns, each containing r_j columnar runs, $j = 1, 2, ..., M$, is equal to:

$$\sum_{j=1}^{M} \sum_{i=1}^{r_j} \omega_{c_{i,j}}.$$

For example, Table 5 is compressed by RLE and then stored in the described format. Looking at $c_{2,1}$, we have $\Delta =$ ' , ', $\lambda_{c_{2,1}} = 4$ and $\lambda_{v_{2,1}} = 5$. Accordingly, the overhead $\omega_{c_{2,1}} = 4 + 1 = 5$ and the total overhead throughout the Customer table is 75.

Table 5 The Customer table after RLE-based compression using the initial $v_{i,j}\Delta L_{i,j}$ storage format

Row	1	2	3	4
Col	Name	City	Color	Gender
1	Nicholas,2	Amsterdam,1	Blue,1	M,4
2	Tomas,2	London,1	White,2	F,2
3	Christian,1	Amsterdam,1	Blue,2	
4	Margaret,1	Rome,1	Purple,1	
5		Zurich,2		
6				

Table 6 The Customer table after RLE-based compression using an optimized storage format with Byte-Aligned Variable-Length Encoding

Row	1	2	3	4
Col	Name	City	Color	Gender
1	2Nicholas	1Amsterdam	1Blue	4M
2	2Tomas	1London	2White	2F
3	1Christian	1Amsterdam	2Blue	
4	1Margaret	1Rome	1Purple	
5		2Zurich		
6				

3.2 Further optimizations

The C++ application provides certain feasible optimizations regarding the storage format of a compressed columnar run. Primarily, the invariable overhead produced by the previously mentioned storage format can be reduced in terms of delimiter removal and subsequent inversion of the $(v_{i,j}, L_{i,j})$ pairs. This obviously reduces the total overhead by $\sum_{k=1}^{M} r_k$ bytes. Compressed columns are now stored in the format $L_{i,j}v_{i,j}$. In this case, $\omega_{c_{i,j}} = \lambda_{c_{i,j}}$. Moreover, the latterly reduced invariable overhead of each columnar run can in fact be revamped into a variable-sized overhead, depending on the actual size of each columnar run. We achieve this by a technique for encoding fixed-size unsigned integers, known as Byte-Aligned Variable-Length Encoding (Dean 2009). Namely, the unsigned integer value representing the length $L_{i,j}$ of a particular columnar run $c_{i,j}$ can be encoded using 7 bits per byte with an additional continuation bit. Given this, the initial constant value $\lambda_{c_{i,j}} = 4$ is replaced by the exact number of bytes required to represent the unsigned integer. This reduction is especially evident when ordered and sorted data contain only a small amount of runs longer than 2,097,152 (2^{21}).

For example, in Table 6, when storing compressed data in an optimal way, for the run $c_{2,1}$ we get $\lambda_{c_{2,1}} = 1$ and the corresponding overhead is $\omega_{c_{2,1}} = 1$. In this case the total overhead throughout the Customer table has been significantly reduced to 15.

Table 7 The modified Customer table (with the first occurrence of 'Zurich' replaced by 'Volgograd') after sorting in an optimal column order

Row	1	2	3	4
Col	Name	City	Color	Gender
1	Nicholas	Amsterdam	Blue	M
2	Nicholas	London	White	M
3	Tomas	Amsterdam	White	M
4	Tomas	Rome	Blue	M
5	Christian	Zurich	Blue	F
6	Margaret	Volgograd	Purple	F

Table 8 The modified Customer table (with the first occurrence of 'Zurich' replaced by 'Volgograd') after RLE-based compression

Row	1	2	3	4
Col	Name	City	Color	Gender
1	2Nicholas	1Amsterdam	1Blue	4M
2	2Tomas	1London	2White	2F
3	1Christian	1Amsterdam	2Blue	
4	1Margaret	1Rome	1Purple	
5		1Zurich		
6		1Volgograd		

3.3 Compression threshold

RLE compression over a single column produces overhead. In some cases, high cardinality columns are present in data. Compressing their runs certainly reduces their physical size, but adding overhead to each run may result in a compressed column with size even larger than the uncompressed column. This issue can be easily resolved with simple calculations prior to compression. Obviously, the compression threshold of the j-th column is prevailed when after reordering and sorting according to the GA, prior to compression, its size $\sum_{i=1}^{r_j} \lambda_{v_{i,j}} L_{i,j}$ is less than its total size, $\sum_{i=1}^{r_j} (\omega_{c_{i,j}} + \lambda_{v_{i,j}})$, after RLE compression, i.e after eliminating duplicates and adding an overhead.

For example, in Table 7, where the first occurrence of 'Zurich' has been replaced by 'Volgograd', the total size of the 'City' column is 43 bytes, whereas in Table 8, after the RLE-based compression, it has increased to 49. In this case, an additional indicator byte is stored along with the standard meta-data to indicate that this particular column has not been compressed. The rest can be safely compressed.

The thread-per-column technique we described earlier is now used to build Binary-indexed trees, often used to sum up N values in $O(\log N)$ time. We do this to avoid iterating over each cell in a column and repeatedly count its length and the length of the run containing it with the number of bytes required to represent this value. Instead, we only need $\lambda_{v_{i,j}}$, $L_{i,j}$ and $\lambda_{c_{i,j}}$. These values are then stored in a single node of the Fenwick tree. The whole tree is built in $O(r_j \log r_j)$ time, whereas the values required by the threshold above can be computed in $O(\log r_j)$ time. A column is compressed only when the threshold is prevailed.

4 Experimental results

In this section we report the results obtained on a set of experiments conducted to evaluate the performance of the proposed meta heuristic approaches and the corresponding RLE implementation in terms of run-reduction efficiency and on-disk size compression ratio. The experiments were done in three stages: first, we evaluated the four optimization algorithms on seven realistic datasets. We choose the two which gave the best results in terms of run-reduction efficiency and convergence time. Then, we generated synthetic datasets and made a sensitive analysis on the parameters of the two selected optimization algorithms. At the end, we again evaluated the performance of the two best algorithms with selected best parameters on realistic datasets in terms of run-reduction efficiency and on disc compression ratio, and compared this results with the performance of the existing heuristics H-LK. (Lemire and Kaser 2011).

4.1 Run-reduction efficiency

The first group of experiments is evaluating the run-reduction efficiency and time of convergence of the four proposed algorithms: SA, TS, GA and PSO. We used seven publicly available datasets representative of real-life data tables: BCUMB and CUMB1881 (Edwards 2010), Census-Income (Hettich and Bay 2000), Nursery and Poker-Hand (Frank and Asuncion 2010), Mushroom (Lichman 2013) and Dermatology data set (Lichman 2013). BCUMB contains records about the birth registrations in Cumberland County. CUMB1881 represents records from the Cumberland County Census 1881. Census-Income contains weighted census data extracted from the 1994 and 1995 current population surveys conducted by the U.S. Census Bureau. Nursery dataset was derived from a hierarchical decision model originally developed to rank applications for nursery schools in Slovenia (Olave et al 1989). In Poker-Hand each record is an example of a hand consisting of five playing cards drawn from a standard deck of 52 (Cattral et al 2002).Mushroom data set contains descriptions of hypothetical samples corresponding to 23 species of gilled mushrooms in Agaricus and Lepiota Family. In the dermatology data set the data about examination of patients with erythemato-squamus deseases are given, The set includes the results of a clinical evaluation, as well as the histopathological features of the skin samples.

The characteristics of these datasets are given in Table 9. The parameter ρ_0 shown in the last column, used by Lemire

et al (2012), is a simple measure of the statistical dispersion of the frequency of the table values, and for a table with N rows and M columns is computed as $\rho_0 = \sum_{i=0}^{M} \frac{f(v_i)}{NM}$, where $f(v_i)$ is the frequency of the most frequent value v_i within a column i, for $i = \overline{1,M}$.

All the experiments were performed on Intel(R) Xeon(R) CPU E5-2650L v3 @ 1.80 GHz with 4GB of RAM and 64-bit Linux (Debian). The parameters that proved to be robust in preliminary tests were the following: GA run with a replication rate of 10% i.e. crossover probability of 90 %($P_c = 0.9$) and a mutation rate of 50% ($P_m = 0.5$) over a population of 20 chromosomes ($K_{pop} = 20$), with maximum of 100 generations. The tournament size in the tournament selection was chosen to be $k = 2$ in all cases. We used SM and OX as the crossover and mutation operators respectively. In the SA algorithm we bounded the temperature with $T_{max} = 500$ and $T_{min} = 0.1$, and 500 steps ($s = 500$) were performed in each run. The bounds for the speed in the PSO algorithm were $V_{min} = 1$ and $V_{max} = N/4$, where N is the number of the columns in the table. The weights used in the calculation of the velocity were: $\eta_1 = 0.2$, $\eta_2 = 0.4$ and the inertia weight $\omega = 0.4$. The maximal number of iterations was set to 100. In the TS algorithm the length of the tabu list n was chosen to be $n = \sqrt{N}$. The algorithm was performed with maximum of 200 iterations, where 5 neighbors were visited in each iteration (k = 5). Each of the two allowed moves used for searching the neighborhood was performed with probability $p = 0.5$.

For each set and each algorithm 10 independent trials were performed and the standard deviation of the resulting data was observed in order to examine the stability of the algorithm. The overall performance measure used for comparisons of the sets was the new number of runs averaged over the 10 independent trials. These values were normalized and averaged over the 7 sets.

The reference point was the result obtained for the GA. We got following result for the number of runs compared to the number of runs return by the GA: for SA the obtained number of runs was $r = 1.0031$, for PSO $r = 1.0177$ and for TS $r = 1.0023$. Our conclusion is that four algorithms gave similar results in terms of the quality of the solution i.e. the difference between the run-reduction efficiency is insignificant.

We needed another criteria for evaluating the performance of the algorithms, the time needed to come to a near optimal solution. For that matter, we constructed plots for every set, where on the x-axis is the time (in seconds) and on the y axis is the best solution for the algorithm at time t. One example (for the bcumb dataset) is shown on Fig. 2. In order to make conclusions about the best algorithms in terms of time needed to find near optimal solution, we need to normalize the values obtained for each set and find the average

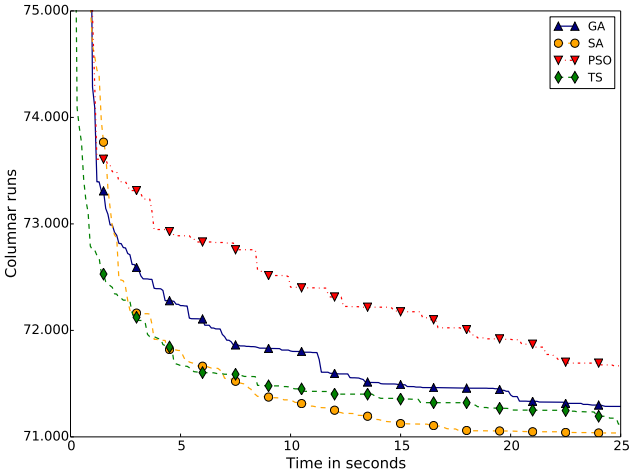

Fig. 2 Convergence on the BCUMB dataset

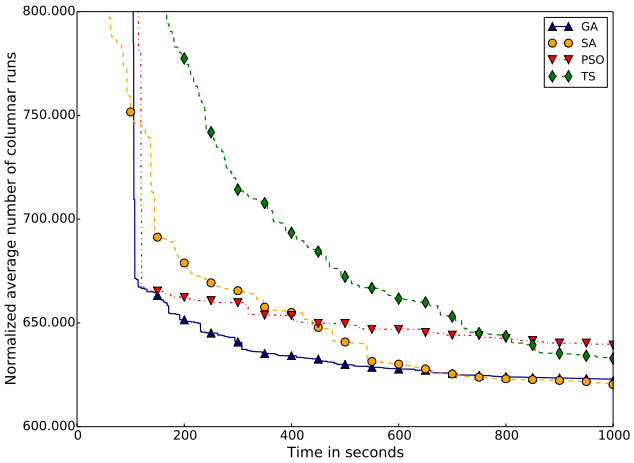

Fig. 3 Time convergence values normalized for all datasets

value of the fitness function at every time t. These summary result are shown on Fig. 3. TS was outperformed at almost every time t by all the algorithms, while GA has the best results after the first generation. Because GA and PSO operate on a set of solution (instead on just one) they need more time to evaluate the solutions in first iteration, but after that the returned results are better compared to the ones returned by TS and SA. After the first iteration, PSO does not have a significant improvement of the best found solution. Based on the fact that PSO is at every time t outperformed by GA and for low values of t SA achieves the best performance, we selected two algorithms: GA and SA as the algorithms with the best performance.

4.2 sensitivity analysis

Our next goal is to further improve the performance of the two selected algorithms, by performing a sensitivity analy-

Table 9 Characteristics of the realistic datasets used in the experiments

	rows	columns	min. card.	max. card.	ρ_0
Dermatology	366	33	2	61	0.66
BCUMB	5.953	20	34	1.357	0.12
Mushroom	8.124	22	2	12	0.58
‘ Nursery	12.960	9	2	6	0.32
CUMB1881	27.363	9	22	5.018	0.06
Census-Income	199.523	42	2	99.800	0.65
Poker-Hand	1.025.010	11	4	13	0.19

sis on their parameters. To achieve that, we constructed two sets of tables. In the first set, the generated tables contain uniformly distributed data. For uniformly distributed tables with M columns whose cardinalities are r_1, r_2, ..., r_M, any value within column i can take one of r_i distinct values with probability $1/r_i$, for $i = \overline{1,M}$. We generated 15 such tables, with 1000, 10000 and 30000 rows, using 10, 15, 20, 25 and 50 independently generated columns with uniform distribution of their values. The results obtained on these tables assume uniformity, thus there is a need to assess the reliability of the obtained results also for skewed data. The Zipfian distribution and its modifications are commonly used to model value distributions in databases (Eavis and Cueva 2007; Houkjær et al 2006). The frequency of the i-th value within a column is proportional to $1/i$. If the table has N rows, then each column can have N possible distinct values, however not all of them will normally appear. Following this direction, for the second set of tables we generated 15 Zipfian-distributed tables for the same number of rows and columns as in the uniformly distributed case. Each table column was generated independently with parameter varying between $a = 1.5$ and $a = 2.5$.

We wanted to find which combination of values for the parameters of GA and SA will give the best result on the two groups of tables generated in terms of run-reduction efficiency. For that purpose, we run the following experiments:

For the GA we experimented with different choices for crossover and mutation probability, as well as different mutation and crossover operators. We tested the following values: $0.5, 0.7, 0.9$ and $0.05, 0.1, 0.2, 0.5$ for crossover and mutation probabilities, and CX, OX, PMX and SM, SIM, IM for crossover and mutation operators, respectively. We performed 108 experiments on each group of tables with 10 independent trials each, totaling to 16200 experimental trials.

For the SA algorithm we experimented with the bounds of the temperature T_{max} and T_{min}. Our goal was to find which combination of the values for these parameters gives best results on the generated dataset. The tested values were $300, 400, 500, 700, 1000$ for T_{max} and $0.01, 0.1, 1, 10$ T_{min}

Fig. 4 shows the averaged reduction of the number of columnar runs attained by GA on Zipfian-distributed data for each combination of crossover and mutation operators. The results are averaged on all sets and all mutation and crossover probabilities. The highest reduction is achieved for PMX and SM operators. It is also clear that mutation operator has higher impact on the results than the crossover operator. The averaged run-reduction results obtained by GA for different crossover and mutation probabilities on Zipfian-distributed tables are shown on Fig. 5. These results show that the best crossover and mutation probability combination is $p_c = 0.9$ and $p_m = 0.5$. It can be concluded that the mutation probability has higher influence on the run-reduction efficiency than the crossover probability. The same combination of parameters was chosen as the best on the uniform distributed datasets as well. These values for the parameters will be used in all further experiments.

The averaged run-reduction results for SA algorithm with 20 combinations of values for the T_{max} and T_{min} parameters for Zipfian-distributed tables are shown on Fig. 6. The best results were obtained for $T_{max} = 300$ and $T_{min} = 0.01$. For uniformly distributed data tables, the best found values for temperature boundaries were $T_{max} = 400$ and $T_{min} = 0.1$, but the obtained value was nearly identical with the averaged value of SA with $T_{max} = 300$ and $T_{min} = 0.01$. Therefore, $T_{max} = 300$ and $T_{min} = 0.01$ will be used as our chosen parameter for SA algorithm in all further experimental applications.

4.3 Final experiments on the realistic datasets

In order to assess the improvement made by the sensitivity analysis in terms of run-reduction efficiency, we performed another set of experiments on the realistic datasets (the datasets are described in Section 4.1). The optimization process was run using the best parameters obtained by sensitivity analysis, namely, for the GA we use SM and PMX as mutation and crossover operators with mutation and crossover probability of 0.5 and 0.9 respectively. For the SA the temperature boundaries are $T_{max} = 300$ and $T_{min} = 0.01$. In Table 4.3 a comparison between the average run-reduction obtained with the first version of the best two optimization algorithms and these algorithms after the sensitivity analysis on the parameters is made. We can conclude that almost for all datasets we achieved better performance with the parameters obtained with the sensitivity analysis.

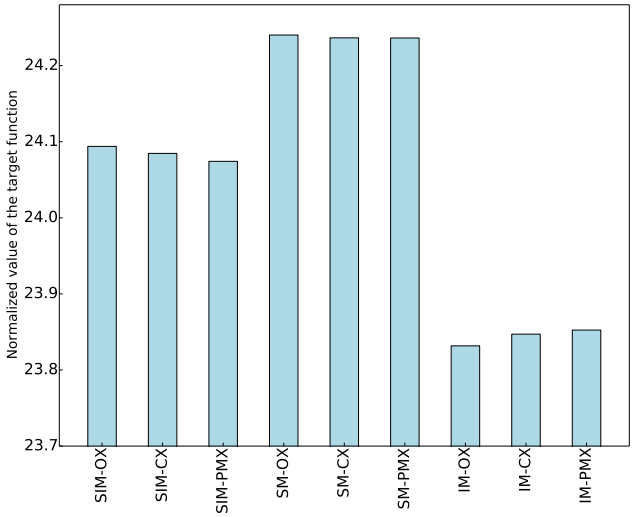

Fig. 4 Run-reduction for different crossover and mutation operators on Zipfian-distributed data

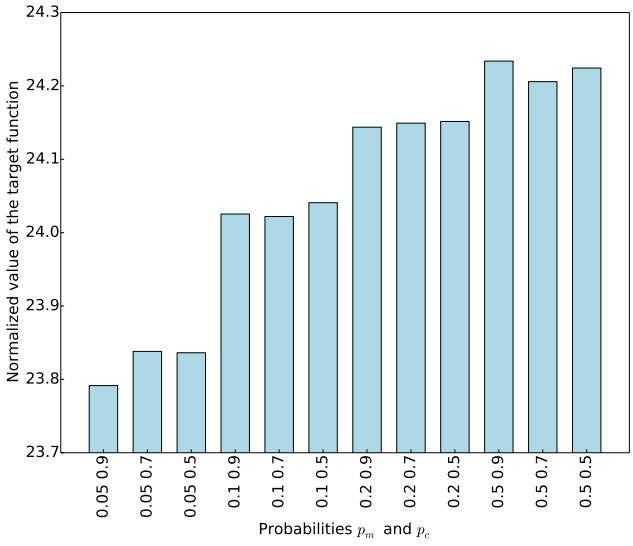

Fig. 5 Run-reduction for different crossover and mutation probabilities on Zipfian-distributed data

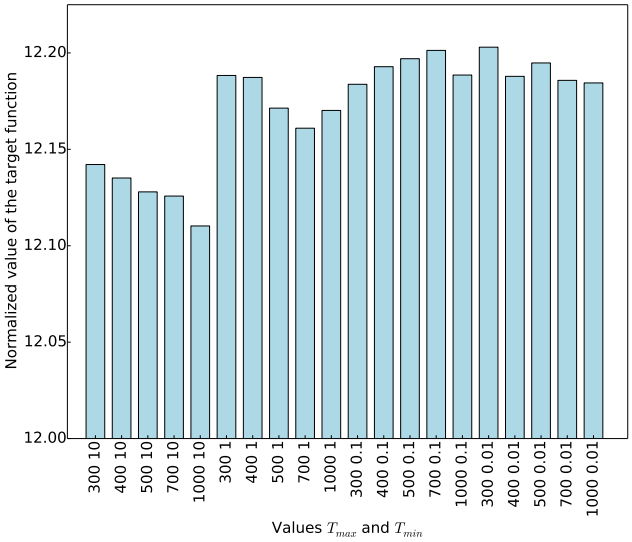

Fig. 6 Run-reduction for different T_{max} and T_{min} values on Zipfian-distributed data

We introduce here another improvement on our algorithms. So far, we generated randomly the initial population of the GA. However, sometimes a kind of heuristic can be used to seed the initial population (Sivanandam and Deepa 2008).For problems of minimization, this will result with low enough value for the mean fitness of the population and it may help the GA to find good solutions faster. An important problem-specific concept that was not taken into consideration in the first approach is the cardinality of the columns. The cardinality of a column denotes the number of distinct values in the column. There are three types of cardinality related to columnar value sets: low cardinality which refers to columns with few unique values (e.g. status flags, boolean values or major classifications such as gender), normal cardinality referring to columns with values that are uncommon but never unique (e.g. names or street addresses), and high cardinality which refers to columns with values that are very uncommon or unique (e.g. id numbers, emails or usernames). To formalize these notions, let $R_1, R_2 \in (0,1)$, $R_1 \leq R_2$. Given a table with N rows, a column with cardinality r is said to have low cardinality if $\frac{r}{N} \in (0, R_1]$ and high cardinality if $\frac{r}{N} \in (R_2, 1]$. Otherwise, the column has normal cardinality.

Columns with low cardinality are more likely to form longer runs of identical values and produce smaller number of runs upon sorting. On the other hand, high cardinality columns tend to form many short single-value runs, therefore sorting by them might not be useful. Following this reasoning, a heuristic based on column cardinalities can be employed to seed the initial population. One chromosome represents the original table and the other $K_{pop} - 2$ chromosomes are generated like in the first approach.We add another chromosome with flag defined by applying the following heuristic: Low cardinality columns are always considered for sorting and their corresponding permutation elements are assigned a flag value from the subset $\{1,2\}$, whereas high cardinality columns are not sorted and are assigned a flag value 0. Columns with normal cardinalities are randomly assigned any value from the set $\{0,1,2\}$. This heuristic approach is intuitive and has already been applied in state-of-the art DBMS, like Vertica DMBS (Lamb et al 2012), where it is combined with RLE compression to attain lower storage requirements and efficient query processing (ver 2015). Then, we sort the columns by the number of unique cells increasingly, and we use this sorting to initialize

the order vector of the chromosome.The used values for R_1 and R_2 were $= 0.1$ and 0.9. Similar logic is applied to the SA algorithm. Here, in the one of the ten trials we begin the optimization process with the heuristic-generated solution. This approach does not lead to significant improvements in terms of run-reduction efficiency, but instead we can achieve better convergence to a near-optimal solutions. We based this conclusion on the results shown on Fig. 7 and Fig. 8. On Fig. 7 the best value of the fitness function for every time t for GA-H and GA-R (normalized on all datasets) is shown. GA-H has lower values for the number of columnar run in the table since the beginning of the evolutionary process. The same conclusion can be drawn from Fig. 8. Based on these observations, we use GA-H and SA-H for all future experiments.

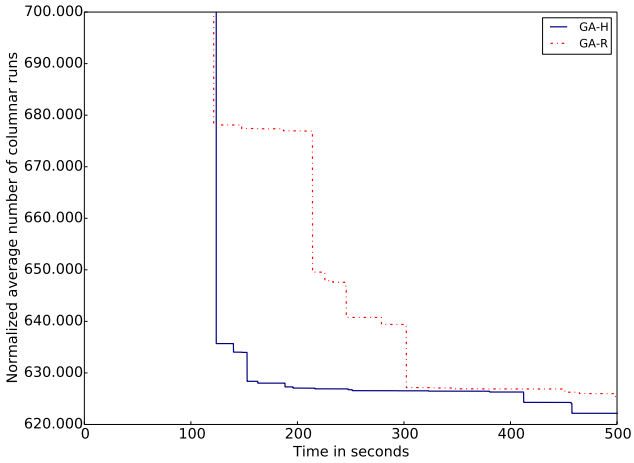

Fig. 7 Convergence comparison for GA-H and GA-R

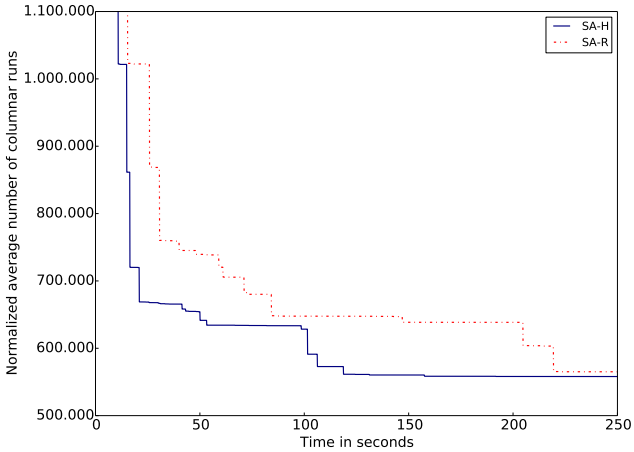

Fig. 8 Convergence comparison for SA-H and SA-R

The experimental results are summarized in Table 4.3 and Table 12. The first table (Table 4.3) gives a comparison between the initial fitness (initial number of columnar runs) of every dataset, and the average result found by H-LK, GA and SA in terms of number of columnar runs. They demonstrate that GA achieves higher run-reduction compared to H-LK and SA in all cases. The reduction improvements are up to 25.52 % in the best case. Most noteworthy are the improvements for the CUMB1881 dataset with a reduction of factor 4.38 (33.08 % vs 7.56 %) and for the BCUMB and Dermatology datasets with a reduction of a factor 2.23 (38.98% vs 17.48 %) and 1.87 (12.28 % vs 6.56 %).Table 12 compares the run reduction efficiency between H-LK, GA-H and SA-H. Ga-H again outperforms SA-H and H-LK, with similar performance to GA.

All these experiments performed on realistic datasets ascertain that our approach is better in terms of quality of the obtained solutions compared to the existing heuristic. However, it is fair to mention that the GA and SA approaches are more computationally expensive due to the fact that evaluating each candidate solution implies sorting of the table, unlike the existing heuristic which considers only a few easily computed statistics such as column cardinality. Nevertheless, in the case of data warehouses, any compression gain can be useful regardless of the computational time because the compression is done once and is used as such afterwards.

4.4 On-disk size compression ratio

Here we use the results described in previous section 4.3 to evaluate the GA-H'S performance in terms of on-disk size compression ratio by using the C++ version of the RLE implementation. The storage format used here corresponds to the one described in Sect. 3.2. Additionally, the threshold-based compression decision discussed in Sect. 3.3 is used to achieve an optimal on-disk actual size after compression.

The experiments were conducted on the previously used realistic datasets: Dermatology, BCUMB, Mushroom, Nursery, CUMB1881, Census-Income and Poker-Hand, exploiting the GA-H (PMX+SM operators, mutation probability 0.5 and crossover probability 0.9) variant of the GA. These experiments were focused on the actual size on the disk a table has after the RLE compression based on the GA and its reduction percentage compared to the reduction percentage of the number of columnar runs within the table.

The experimental results are summarized in Table 4.4. These experiments confirmed the storage format optimization discussed earlier. The results obtained show that the reduction of the actual1 on-disk size after compression is larger than the reduction of the number of columnar runs. With our implementation of RLE compression based on the described GA, the run reduction is outperformed by the

Table 10 Comparison between the initial result and the ones obtained with sensitivity analysis

	GA	GA-Sensitivity a.	SA	SA-Sensitivity a.
Dermatology	3.193	3.177	3.280	3.247
BCUMB	70.977	70.951	71.659	71.330
Mushroom	12.009	11.971	12.058	12.185
Nursery	17.702	17.679	17.707	18.901
CUMB1881	81.968	81.986	81.990	81.982
Census-Income	957.320	956.817	978.765	979.551
Poker-Hand	3.148.845	3.149.367	3.149.892	3.149.490

Table 11 Performance on realistic Datasets

	Initial fitness	Heuristic H-LK	GA	SA
Dermatology	5.207	4.296 (17.48 %)	3.177 (38.98 %)	3.247 (37.64 %)
BCUMB	80.880	75.570 (6.564 %)	70.951 (12.28 %)	71.330 (11.80 %)
Mushroom	69.235	13.129 (81.04 %)	11.971 (82.70 %)	12.185 (82.40 %)
Nursery	29.617	19.435 (34.38 %)	17.679 (40.30 %)	18.901 (36.18 %)
CUMB1881	122.510	113.248 (7.56 %)	81.986 (33.08 %)	81.982 (33.08 %)
Census-Income	3.764.757	1.054.198 (72 %)	956.817 (74.58 %)	979.551 (73.98 %)
Poker-Hand	9.157.488	3.180.362 (65.27 %)	3.149.367 (65.61 %)	3.149.490 (65.61 %)

Table 12 Run reduction efficiency of GA-H, SA-H and H-LK.

	H-LK	GA-H	SA-H
Dermatology	17.48%	38.8%	37.59%
BCUMB	6.56%	12.25%	11.16%
Mushroom	81.04%	82.76%	82.42%
Nursery	34.38%	40.17%	36.27%
CUMB1881	7.56%	33.06%	33.05%
Census-income	72%	74.75%	73.95%
Poker-Hand	65.27%	65.61%	65.6%

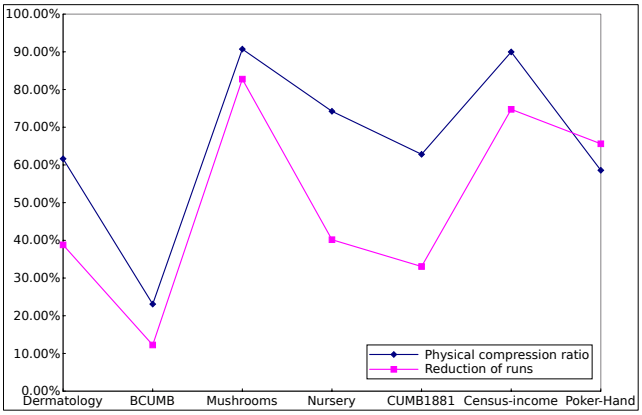

Fig. 9 Run-reduction efficiency vs. on-disk size compression ratio of GA-H(PMX+SM) on realistic datasets

compression ratio on disk. This is a consequence of short run lengths (represented with less than 4 bytes) together with wider columns, which is the critical point for saving disk storage. It is evident from Fig. 9 that this holds for all datasets except for Poker-Hands. This fact stands out especially for the Nursery dataset where compression ratio is almost twice as much as its run reduction. Table 4.4 indicates that the optimal solution of 40.17% of run reduction corresponds to 74.23% of physical size reduction. BCUMB and CUMB1881 are also characterized with such results, where the amount of physical reduction is almost double the amount of run reduction. Though Census-Income, Mushroom and Dermatology datasets do not follow the pattern of twofold physical reduction, it still shows a great improvement over run reduction.

In contrast, Poker-Hand differs from the rest of the datasets as its compressed rate is about 7% less than the run reduction. This anomaly occurs as a result of the average width of the columns in this dataset, that is equal to 1 byte across all columns - much less than in other datasets we have experimented on. Since the RLE length indicators' length varies from 1 to 4 bytes, the results above have helped us conclude that our algorithm achieves better compression rate in terms of physical on-disk size when the average width of the columns is larger. For small average column widths, much longer runs are required to achieve better compression rates.

All sets except Nursery and Poker-Hand contain empty cells. When each empty cell is replaced by an indicator the algorithm actually performs better in terms of compression rate and actual size on disk. Namely, when a columnar run of empty cells is compressed by RLE, a negative compression rate of the run is obtained due to the fact that the size of such columnar runs is 0 bytes and the compressed run's size is

Table 13 Reduction of runs and on-disc size reduction

	Size on disc	Compressed size	Reduction of runs
Dermatology	26.426	10.137 (61.64 %)	38.8 %
BCUMB	591.988	455.311(23.09 %)	12.25 %
Mushroom	381.888	35.450(90.72 %)	82.76 %
Nursery	1.072.349	276.304(74.23 %)	40.17 %
CUMB1881	1.243.068	462.139 (62.82 %)	33.06 %
Census-Income	95.876.138	9.614.420 (89.97 %)	74.75 %
Poker-Hand	25.152.051	10.417.766(58.58 %)	65.61 %

also 0 bytes. A minimum of 1 byte is added for the run length and therefore, the compression rate is negative. In the case when an indicator of 1 byte is used to mark each empty cell, it is obvious that the compression rate is positive whenever the run length is greater than 1. In such cases, both better compression rate and lesser size on disk can be achieved. The results presented in Table 4.4 come from experiments where each empty cell in a dataset was replaced (or marked) by a single-byte indicator.

5 Conclusions

We addressed the problem of RLE compression of a column store table based on minimization of the number of columnar runs in the table. We presented a generational GA and other meta-heuristic approaches such as PSO, SA, and TS for determining an optimal column sorting order which minimizes the number of columnar runs in a column store table. The algorithms employ permutation representation, pairwise tournament selection, elitism, and three different crossover and mutation operators. They were implemented and tested using both random and column cardinality-based heuristic generation of the initial population. The experiments performed on uniform and skewed data demonstrated that best results are achieved when using heuristic-based initial population with PMX crossover and SM mutation. Moreover, it was shown that fine-tuned genetic algorithm and simulated annealing perform consistently well also on realistic datasets and can achieve up to 4 times higher reduction compared to existing heuristics for solving the run minimization problem.

Furthermore, we have presented a comprehensive implementation of RLE compression in addition to the proposed meta-heuristics, which has proved effective at physical compression to an extent that it is capable of serving as an everyday tool. Rearrangement of the storage format for minimal overhead, encoding the integers to save space, marking the empty cells, and finally, an overhead threshold-based decision on compression have all contributed significantly to further savings of disk space. Experimental results obtained on realistic datasets showed that the implementation results are predominantly characterized with physical compression ratio on disk that exceeds the reduction of columnar runs by an average factor of 2, and sometimes even more.

All these experimental results demonstrated the effectiveness and robustness of the proposed GA and SA approaches and their performance in the experiments are an encouragement to merit further research into adding adaptive characteristics by mining experimental results.

References

(2015) Choosing sort order: Best practices - vertica online documentation. `http://web.archive.org/web/20080207010024/http://www.808multimedia.com/winnt/kernel.htm`, accessed 27 March 2016

Abadi D, Boncz P, Harizopoulos S, Idreos S, Madden S (2013) The design and implementation of modern column-oriented database systems. Foundations and Trends in Databases, 5(3): 197-280

Abadi DJ, Madden SR, Ferreira MC (2006) Integrating Compression and Execution in Column-Oriented Database Systems. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, USA, pp 671–682

Abadi DJ, Madden SR, Hachem N (2008) Column-Stores vs. Row Stores: How Different Are They Really? In: Wang JTL (ed) Proceedings of the ACM SIGMOD International Conference on Management of Data, ACM New York, NY, USA, Vancouver, Canada, pp 967–980

Arsov N, Siljanoska Simons M, Jovanovski J (2014a) GeneticAlgorithmRLE_Cpp: Genetic algorithm compression of tabular data. `https://github.com/ninoarsov/GeneticAlgorithmRLE_Cpp`, accessed 20 November 2014

Arsov N, Siljanoska Simons M, Jovanovski J (2014b) GeneticAlgorithmRLE_LOExtension Git Hub Repository: A Libre Office Calc Extension Implementing a Genetic Algorithm for RLE Compression of CSV data. `https://github.com/ninoarsov/GeneticAlgorithmRLE_LOExtension`, accessed 10 June 2014

Bäck T, Fogel DB, Michalewicz Z (2000) Evolutionary Computation 1: Basic Algorithms and Operators. CRC Press

Cattral R, Oppacher F, Deugo D (2002) Evolutionary Data Mining with Automatic Rule Generalization. In: Mastorakis N, Mladenov V (eds) Recent Advances in Computers, Computing and Communications, WSEAS Press, New York, pp 296–300

Chang CC, Chen YH, Lin CC (2009) A data embedding scheme for color images based on genetic algorithm and absolute moment block truncation coding. Soft Computing 13(4):321–331

Copeland GP, Khoshafian S (1985) A Decomposition Storage Model. In: Proceedings of the ACM SIGMOD Conference on Management of Data, Austin, Texas, pp 268–279

Dean J (2009) Challenges in building large-scale information retrieval systems. In: Keynote of the 2nd ACM International Conference on Web Search and Data Mining (WSDM)

Eavis T, Cueva D (2007) A Hilbert Space Compression Architecture for Data Warehouse Environments. In: Song I, Eder J, Nguyen T (eds) Data Warehousing and Knowledge Discovery, Lecture Notes in Computer Science, vol 4654, Springer-Verlag, Berlin, Heidelberg, pp 1–12

Edwards G (2010) Nova Scotia GenWeb Project, Cumberland County GenWeb. `http://www.rootsweb.ancestry.com/~nscumber/sources.html`, accessed 10 June 2014

Fenwick PM (1994) A new data structure for cumulative frequency tables. Software: Practice and Experience 24(3):327–336

Frank A, Asuncion A (2010) UCI Machine Learning Repository. `http://archive.ics.uci.edu/ml`, accessed 10 June 2014

Hettich S, Bay SD (2000) The UCI KDD archive. `http://kdd.ics.uci.edu`, accessed 10 June 2014

Hoffer J, Severance D (1975) The Use of Cluster Analysis in Physical Data Base Design. In: Proceedings of VLDB, Framingham, USA, pp 69–86

Holsheimer M, Kersten ML (1994) Architectural Support for Data Mining. In: Fayyad U, Uthurusamy R (eds) Knowledge Discovery in Databases: Papers from the 1994 AAAI Workshop, Technical Report WS-94-03, AAAI Press, Seattle, Washington, USA, pp 217–228

Houkjær L, Torp K, K W (2006) Simple and Realistic Data Generation. In: Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, pp 1243–1246

Iyoda EM, Shibata T, Nobuhara H, Pedrycz W, Hirota K (2007) Image compression and reconstruction using pi t-sigma neural networks. Soft Computing 11(1):53–61

Ji Z, Zhou J, Zhu Z, Chen S (2013) Self-configuration single particle optimizer for dna sequence compression. Soft Computing 17(4):675–682

Jovanovski J, Siljanoska M, Velinov G (2013) A genetic algorithm approach for minimizing the number of columnar runs in a column store table. In: Adaptive and Natural Computing Algorithms, 11th International Conference, ICANNGA 2013, Lausanne, Switzerland, April 4-6, 2013. Proceedings, pp 485–494, doi:10.1007/978-3-642-37213-1_50

Kirkpatrick S, Vecchi MP, et al (1983) Optimization by simmulated annealing. science 220(4598):671–680

Lamb A, Fuller M, Varadarajan R, Tran N, Vandiver B, Doshi L, Bear C (2012) The vertica analytic database: C-store 7 years later. Proceedings of the VLDB Endowment 5(12):1790–1801

Larson PÅ, Hanson EN, Price SL (2012) Columnar Storage in SQL Server 2012. IEEE Data Engineering Bulletin 35(1):15–20

Lee KY, El-Sharkawi MA (2008) Modern Heuristic Optimization Techniques: Theory and Applications to Power Systems, 3rd edn. Wiley-IEEE Press

Lemire D, Kaser O (2011) Reordering Columns for Smaller Indexes. Int J Inf Sci 181(12):2550–2570, doi:10.1016/j.ins.2011.02.002

Lemire D, Kaser O, Gutarra E (2012) Reordering rows for better compression: Beyond the lexicographic order. ACM Trans DB Syst 37(3):2550–2570

Lichman M (2013) UCI machine learning repository. URL `http://archive.ics.uci.edu/ml`

Marz N, Warren J (2015) Big Data: Principles and best practices of scalable realtime data systems. Manning Publications Co.

Mitchell M (1998) An Introduction to Genetic Algorithms, 3rd edn. The MIT Press, Cambridge

Olave M, Rajkovic V, Bohanec M (1989) Expert Systems in Public Administration: Evolving Practices and Norms, Elsevier Science Publishers, chap An application for admission in public school systems, pp 145–160

O'Neil P, Quass D (1997) Improved query performance with variant indexes. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, Tucson, USA, pp 38–49

Sivanandam SN, Deepa SN (2008) Introduction to Genetic Algorithms. Springer-Verlag Berlin Heidelberg

Tan L, Sun J, Tong X (2014) A hybrid particle swarm optimization based memetic algorithm for dna sequence compression. Soft Computing pp 1–14

Tang D, Liu T, Lee R, Liu H, Li W (2015) A case study of optimizing big data analytical stacks using structured data shuffling. In: Cluster Computing (CLUSTER), 2015 IEEE International Conference on, IEEE, pp 70–73

Yimin C, Yixiao W, Qibin S, Longxiang S (1999) Digital image compression using a genetic algorithm. Real-Time Imaging 5(6):379–383