Semantic Database Compression System Based on Augmented Vector Quantization

Saad M. Darwish[1*], Saleh M. El-Kaffas[2], Omar A. Abdulateef[1]

[1] Department of Information Technology, Institute of Graduate Studies and Research, Alexandria University, 163 Horreya Avenue, El-Shatby 21526, P.O. Box 832, Alexandria, Egypt.
[2] College of Computing & IT, Arab Academy for Science, Technology & Maritime Transport (AASTMT), Egypt.

* Corresponding author. Tel.: +20122632369; email: saad.darwish@alex-igsr.edu.eg
Manuscript submitted March 1, 2015; accepted June 27, 2015.
doi: 10.17706/jsw.10.10.1127-1139

Abstract: In the last years, that amount of data stored in databases has increased extremely with the widespread use of databases and the rapid adoption of information systems and data warehouse technologies. It is a challenge to store and recover this increased data in an efficient method. This challenge will potentially appeal in database systems for two causes: storage cost reduction and performance improvement. Lossy compression in databases can return better compression ratios than lossless compression in general, but is rarely used due to the concern of losing data. For relational databases, using standard compression techniques like Gzip or Zip don't take advantage of the relational properties; since these techniques don't look at the nature of the data. In this paper, we propose a database compression system that takes advantage of attributes semantics and data-mining models to find frequent attribute pattern with maximum gain to perform compression of massive table's data. Furthermore, the suggested system relies on augmented vector quantization (*AVQ*) algorithm to achieve lossless compression version without losing any information. Extensive experiments were conducted and the results indicate the superiority of the system with respect to previously known techniques.

Key words: Lossless database compression, semantic encoding, augmented vector quantization.

1. Introduction

With the widespread use of databases and data warehousing technologies, the amount of data stored in databases has increased tremendously. It becomes attractive to compress data in database systems. Relational data is quite different from the text and multimedia data, because many semantic structures (e.g., data dependencies and correlations) exist within relational data. Database compression techniques can have a positive cost effect not only on the storage and transfer of data, but also in the areas of database security, backup and recovery procedures, as well as enhancing database performance. Yet, database compression introduces several problems such as data property disruption, portability, and system complexity. Database compression adds a layer of complexity to the design, implementation, and operation of a database system. Designers are reluctant to accept the additional complexity [1], [2].

As with any compression technique, the effectiveness of compressing the data not only depends on the system, but also depends on the characteristics of the data in the system. The following data characteristics offer a brilliant environment for the application of database compression techniques [1]: 1) Sparseness, in sparsity database fields tend to have large clusters of zeros, blanks, or missing data indicators. Sparseness is

the most important property in determining the overall compression percentage that could be achieved in the database. 2) Frequency, in large textual databases characters with alphanumeric attributes don't occur with the same frequency. This can be taken advantage of redundancy codes. In general, effective table compression techniques that are semantic in nature exploits both 1) the meanings and dynamic ranges of individual attributes (e.g., by taking advantage of the specified error tolerances); and, 2) existing data dependencies and correlations between attributes in the table [3].

Data redundancy occurs when strings of data patterns are predictable, and therefore carry little or no new information. Finding and exploiting this redundancy in databases is the basis for any data compression technique. There are four basic types of redundancy present in databases that include: character distribution, character repetition, high-usage patterns, and positional redundancy; with some databases having a mixture of types; see [1] for more details. To apply data compression to databases, the compression scheme must satisfy the following requirements: 1) lossless compression, as the exact information must be preserved; 2) fast compression/decompression for databases in active uses [2]. Our proposed compression system relies on the data mining approach to extract high-usage patterns in the table; so patterns that will appear with relatively high frequency can be represented (coded) with fewer bits.

Traditional "syntactic" database compression methods, such as Lempel-Ziv, simply treat the table as a large byte string and operate at the byte level. Thus, they fail to exploit the semantic structures in the relation (table) [2], [4]. The tradeoff in such case is usually between the ease of retrieval (the ease with which one can retrieve a single tuple or attribute value without decompressing a much larger unit) and the effectiveness of the compression. Typically, compressing a database in this fashion can produce a compression factor between 2 and 4, i.e. the compressed database is one-half to one-quarter its original size [5]. The use of semantic compression has generated considerable interest and motivated certain recent works.

Semantic compression uses relationships within the data to compress it rather than regarding the table as a file. Semantic compression can be done in a row-based manner, which involves clustering rows together with a certain amount of variation, or a column based manner, which involves removing columns that can be predicted using other columns and storing any exceptions that exceed a certain error bound separately [4]. While row-wise semantic compression in general shows better compression ratios, column-wise compression has its own advantages. Semantic compression can be divided into two separate techniques: semantic independent and semantic dependent. Semantic independent techniques can be used for any data type, with varying degrees of effectiveness, and don't use any information regarding the content of the data. Semantic dependent techniques depend, and are based, on the context and semantics of the data. Most the data compression techniques discussed in the literature of this latter type [1], [6].

1.1. Our Contribution

In this paper, we describe the architecture of a new system that takes advantage of attributes' semantic to perform lossless compression, most commonly chosen type of compression for the sake of not losing data, of massive data tables. We base our compression technique on the observation that sets of some attribute values occur frequently in a relational table. The proposed semantic independent compression system is based on a novel idea of exploiting data correlations for individual attributes using data mining to discover or learn redundancy of high-usage patterns inside table for achieving a higher compression ratio.

This work primarily focuses on the underneath component of the compression framework, that is to efficiently find dependency patterns in relational data to optimize the compression ratio through utilizing the concept of compression gain. Our system is similar to clustering, with each considered representative pattern like a cluster representative. The proposed system uses a lossless version of vector quantization

called augmented vector quantization (*AVQ*) that is appropriate for database compression to reduce database space storage requirements and improve disk I/O bandwidth. One advantage of this approach is that compression/decompression is performed at the table-level, which is desirable for increasing the compression ratio and security.

The outline of remainder of this paper is as follows. Section 2 presents a summary of the state-of-the-art semantic database compression approaches. Section 3 describes the proposed data-mining-based semantic compression system. The test results and discussion of the meaning are shown in Section 4. A short summary of this paper and outlook of future work is given in Section 5.

2. Background and Literature Survey

Compression can be applied to databases at relation level, page level and tuple or attribute level [7], [8]. In tuple level, storing data in column presents a number of opportunities to improve performance for compression algorithms when compared to row-oriented architectures. In a column-oriented database, compression schemes that encode multiple values at once are natural. In a row-oriented database, such schemes do not work as well because an attribute is stored as a part of an entire tuple, so combining the same attribute from different tuples together into one value would require some way to mix tuples. In page level (Block-oriented) compression methods, the compressed representation of the database is a set of compressed tuples. When access to a tuple is required, the corresponding page is transferred to the memory and the only decompression is done to obtain the decompressed tuple.

In the literature, there are three types of block-oriented database compression techniques [6], [9]: 1) Bit compression (BIT), 2) Adaptive Text Substitution (ATS), and 3) Tuple Differential Coding (TDC). Two of them, BIT and ATS, are adaptations of conventional data compression techniques. The third one exploits the redundancy among tuples differently to achieve compression. This allows decompression at the field level; this technique is only useful for records with low-cardinality fields. The original work describing TDC discussed practical details such as how to handle textual attributes in TDC [9]. It demonstrated that TDC is superior to other database compression methods currently in use, and provides both better compression ratios as well as faster query response times.

Although semantic compression that operates on both relation and block level has several advantages over syntactic compression, the two types of compression are not mutually exclusive. In fact, it has been stated in [4] that applying semantic compression before syntactic compression results in better compression performance than from either syntactic compression or semantic compression. Still, syntactic compression used in the second phase will invalidate the fast retrieval benefit discussed earlier for semantic compression.

Now, we present a brief sketch in the previously known semantic compression algorithms that are related to our work. The fascicles algorithm presented in [10] is the first semantic compression algorithm developed for tables. Given a table of m columns and a user-specified value of u ($u \leq m$), the algorithm extracts a model M consisting of w fascicles, each of which is represented by a u-tuple. The u columns are called compact attributes because these are columns with very similar values (i.e., values within the error tolerance) for all the rows assigned to the fascicle. While the fascicles algorithm determines the u compact columns locally on a per fascicle basis, SPARTAN [3] tries to separate the m columns into a set of predictor attributes and a set of predicted attributes globally for the entire relation. The model M, in this case, is simply the set of the predictor attributes. SPARTAN identifies the predictor columns by constructing Bayesian Network and CaRTs (classification and regression trees). As seen in the fascicle algorithm and SPARTAN, the key aspects that differentiate one semantic compression algorithm from another are the exact

definition of the model *M* used to compress the database and how it is constructed. SPARTAN was shown to achieve better compression ratios than gzip and fascicles on selected datasets.

In column-wise semantic database compression, columns are compressed based on their associations with other columns by using predictive models [11], [12]. These predictive models could be used for more than just bulk decompression. ItCompress [4] employed the approach of Fascicles but improved the algorithm by applying the row-wise compression algorithm iteratively that achieves better compression ratios than even SPARTAN. While ItCompress achieved a better compression ratio than Spartan, Spartan's usage of compression in a more transparent manner makes it a more attractive choice for integration into a widely used DBMS. This transparency can be useful, as the query engine can take these into account when running queries, and using these cleverly could yield significant performance improvements.

In the literature, many lossless semantic compression algorithms for relational databases are suggested. For example, researchers in [2] proposed a semantic compression technique that exploits frequent dependency patterns embedded in the table. One advantage of this approach is that compression/decompression is performed at the tuple-level, which is desirable for integrating the compression technique into database systems. The key algorithmic problem is to compute an optimal set of frequent patterns in the input relational table such that the overall storage of the compressed table is minimized. They showed that it is hard to compute an optimal compression solution. Therefore, an iterative greedy compression framework is offered to solve this problem.

With the purpose of improving the efficiency of lossless compression methods on a column-by-column basis, some authors employed methods such as dictionary encoding and run-length encoding. However, these methods were implemented in C-Store, which is a column-oriented DBMS [13]-[15]. While using a system such as C-Store could certainly make column-wise semantic compression easier to use, the methods described there do not adapt easily to row-oriented DBMS as well (run-length encoding, for example) [12]. In addition, the authors in [13] have made the case for integrating lossless compression into DBMS products as a means of improving price performance. Tradeoffs were made to solve numerously architectural problems encountered during this integration. They chose the row as a unit of compression. The selected compression algorithm is a non-adaptive variant of the Ziv-Lempel algorithm using extension symbols. The Ziv-Lempel parse tree is determined by building a large parse tree with sampled rows and trimming it to the desired number of nodes.

In [16] a method was described to allow the encoding and compression of one or more tables of data by splitting each table into two or more sub-tables, followed by permutation of the sub-tables. Here, the basic idea in semantic encoding is to store some projections of a table, rather than the table itself. W. Ng et al. [17] designed and implemented a novel database compression method based on augmented vector quantization (*AVQ*) that does not incur any of the computational overheads of conventional *VQ*. The output vectors computed without resorting to any codebook computation algorithms. There is no need for code words as the each vector is associated with a disk block, and no searching of the codebook is necessary. These features make *AVQ* more computationally efficient than the conventional *VQ* in terms of coding and decoding. However, this scheme is only applicable to discrete finite domains where the attribute values known in advance, and the cardinality of each domain is small.

The work suggested in [18] reports a lossless compression technique called non-differential augmented vector quantization. The technique maps a database relation into a static bitmap index cached access structure. Consequently, anyone was able to achieve substantial savings in space by storing each tuple as a bit value in the computer memory. Important distinguishing characteristics of their technique are that tuples can be compressed and decompressed individually rather than a full page or entire relation at a time. Furthermore, the information needed for tuple compression and decompression can reside in the memory.

The algorithm supports standard database operations, permits very fast random access and atomic decompression of tuples in a large collection of data with low decompression cost. For a short survey of database compression techniques for column oriented database, readers can refer to [11].

Following this recent development, this paper presents a new approach for lossless semantic database compression based on a modified version of *AVQ* that exploits semantic structures in a table. The proposed system utilizes data mining technique to extract frequent pattern with maximum gain embedded in table to be used as a representative tuple. The system relies on the semantic independent concept and employs predictive data correlations for individual attributes to construct concise and accurate compression model. For evaluation purposes, we compare our scheme to traditional lossless–semantic compression suggested by H. Huang [2], and find that our scheme reduces the database size by a greater extent in most cases.

3. Proposed Method

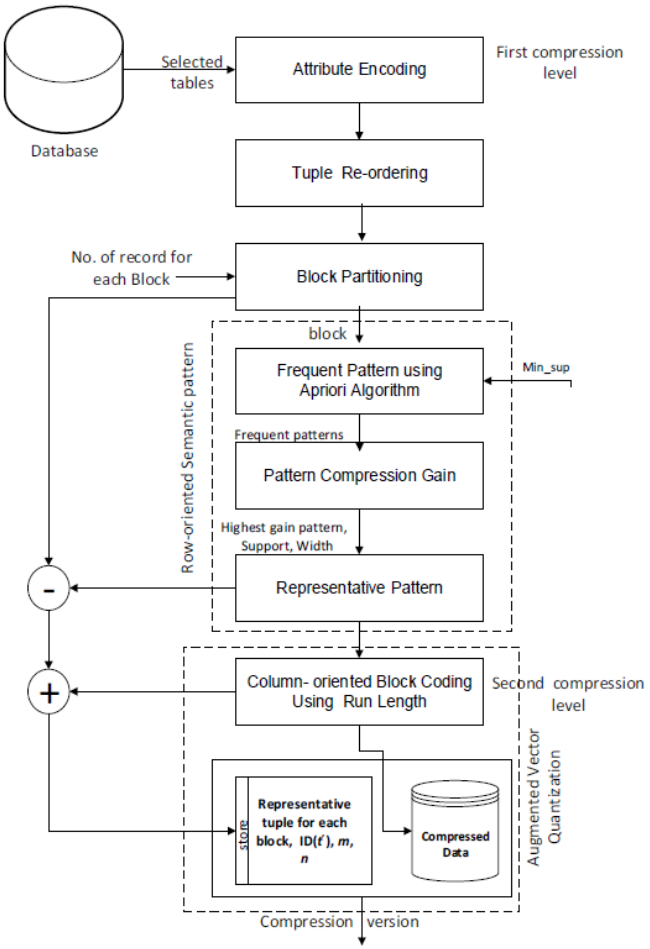


Fig. 1. Proposed lossless semantic database compression.

Semantic encoding simultaneously compresses a table by transforming it into a collection of components that can be reassembled in an extremely large number of meaningful ways. This transformation can compress the table by a much larger factor than can be achieved by syntactic compression methods [19]. Here, the compression achieved by semantic encoding result from the fact that when a table is projected, many repeated entries are eliminated. Fig.1 shows the general data flow diagram of the proposed lossless semantic compression system for relational database based on augmented vector quantization, which evolved with better compression ratio and performance.

The system tries to derive a descriptive model of the database by taking into account the semantics of the attributes. The key of the proposed method is to compress a large table based on an Apriori data mining algorithm to find frequent patterns that take the semantics of the table into consideration during compression. So, complex correlation and data dependency between the data attributes can be exploited. Note that this is not supported in case of syntactic compression methods since the database is viewed as a large byte string in such methods. The system consists of five steps: 1) attribute encoding, 2) tuple re-ordering, 3) block partitioning, 4) semantic patterns extraction, 5) compression gain calculation and 6) block coding.

3.1. Attribute Encoding

The first pre-processing step in the suggested system encodes each attribute value to a number. For discrete finite domains where all the attribute values are known in advance, each attribute value is mapped to its ordinal position in the domain. For other domain types, more work is needed. For alphanumeric strings, we may construct a table containing the set of these strings and replace each attribute by an index into the table. Other schemes may be used. Observe that this step by itself achieves compression (first level) because an attribute value that consists of a long string of ASCII characters is mapped to a short number.

3.2. Tuple Re-ordering

The next pre-processing step is to re-order the tuples by an ordering rule $\varphi : R \rightarrow N_{\Re}$ defined as [17]:

$$\phi(a_1,\ldots,a_n) = \sum_{i=1}^{n}\left(a_i \prod_{j=i+1}^{n}\left|A_j\right|\right), \quad (1)$$

φ is an n-dimensional to 1-dimensional mapping that maps a tuple $t \in R$ uniquely into its ordinal position in the $\Re$ space. A relation $R = \langle\langle A_1, A_2, \ldots, A_n\rangle\rangle$ is a subset of $\Re$ and corresponds to the set of input vectors to be coded. t is an n-dimensional vector. $N_{\Re} = \{0,1,\ldots,\|\Re\|-1\}$ be a set of integers that correspond to $\mathcal{R}$ space, where $\|\Re\| = \prod_{i=1}^{n}|A_i|$ is the size of the $\Re$ space. In our case, $\langle a_1, a_2, \ldots a_n\rangle \in \Re$ is randomly chosen based on the number of attribute. The role of $N_{\Re}$ is to lexicographically reorder the tuples based on all attributes A_j taking into account the correlation between them.

3.3. Block Partitioning

In this step, we partition the re-ordered table into k disjoint subsets of tuples $B_i, B_i, \ldots, B_k$, so that coding and decoding is performed at the granularity of data objects (block level). In our case, the user determines the partition size depending on the number of tuples in the table. Other schemes may utilize the size of a memory page or disk sector as the partition size as it is the unit of I/O transfer. This step simplifies the compression because coding and decoding is localized, i.e. if tuples in the block are coded, then decoding need only be performed on the block. Furthermore, after tuple reordering and block partitioning, tuples in a block form a cluster.

3.4. Semantic Patterns Extraction

When correlations exist between the attribute domains in R the entropy of the database is lower; so that compression methods tend to perform better in such cases [6]. Our algorithm extracts frequent patterns from the relational data by employing data mining techniques, and uses such knowledge to extract the representative row to minimize storage requirements through tuple difference coding. Successive tuples are

differenced, and the differences are stored instead of the original tuples themselves.

Given B_i, $i = 1 to k$ where there are m tuples with n attributes. A tuple t can be viewed as a set of <attribute, attribute-value> pairs. A subset $P \subseteq B_i$ is called an itemset. The support or support count of an itemset B_i, denoted as $Sup(P)$, is the number of tuples in the block B where P occurs as a subset. If the support of an itemset is greater than or equal to a user-specified support threshold *min_sup*, the itemset is called a frequent itemset. A frequent itemset is also called a frequent pattern. We refer to the number of items in a pattern P as the width of pattern P, denoted as $|P|$. The proposed system employs Apriori algorithm as a pioneer association rules mining method to extract large itemset from B_i for its simplicity [2].

3.5. Pattern Compression Gain

The key algorithmic problem is to compute an optimal set of frequent patterns for the input$B_i$such that the overall storage of the compressed table is minimized. This is a challenging problem because, not only is finding frequent patterns in the relational data a computation-intensive task, but also is selecting a subset of frequent patterns for optimal compression NP-hard. Here, we utilize the compression gain concept to solve the problem of selecting an optimal frequent pattern. The compression gain of a pattern P is defined as[2]:

$$\text{gain}(p) = \begin{cases} |P|.\text{Sup}(P) & \text{if } |P|>1, \text{ and } \text{Sup}(P)>1 \\ 0 & \text{if } |P|=1, \text{ or } \text{Sup}(P)=1 \end{cases}, \quad (2)$$

Essentially, the idea of using compression gain is to measure the contribution of a pattern to the storage saving by its coverage over the block; because patterns that 'cover' larger area in the block will lead to better compression [2]. Consider the three patterns show in Table 1, while the support decreases monotonically with the increasing width, the gain can either increase or decrease, depending on the actual value of |P| and Sup(P). The width of a pattern is bounded by the maximum number of items in a record, (i.e. the total number of attributes in the table). Given number of frequent patterns, the output of this step is the highest gain pattern that is used as a representative pattern $\hat{p}$.

Table 1. Non-monotonic Behavior of Pattern Gain Computation

Itemset	1,2	1,2,3	96,4,97,6
Width	2	3	4
Support	70	60	50
Gain	140	180	200

A block B_k B_k now consists of a set of ordered tuples $B_k = (t_{k,1}, t_{k,2}, \ldots t_{k,i}), t_{k,i} \in R$ with $t_{k,i} < t_{k,j}$ $for\ i < j$. The first tuple of the representative pattern in each block B_k is chosen as the representative tuple $\hat{t}$ of the block (has the same values of the remaining tuples in $\hat{p}$). Thus, every tuple $t_{k,i} \in B_k$ is mapped to $\hat{t}_k$ such that the total distortion $\sum_{i=1}^{m} |\varphi(t_{k,i}) - \varphi(t)|$ is minimized and the block is sparse, so that it contains zeros as much as possible. Here, a new semantic compression scheme based on the selection of representative tuple of each block is utilized.

3.6. Block Encoding

The idea of redundancy (repetition) of values within a column in a relational table can also be extended

between columns; this constitutes stage two of our compression algorithm. Using the first tuple in representative pattern of each block as a reference, each tuple in the block is replaced by its difference with respect to its reference tuple. So that attribute values are defaulted to be zero if it is the same with the reference unless the actual value differs from the reference value. In such cases, outlying values are specifically stored for the row without any change. Formally, for each block B_i given two tuples $\hat{t}, t_j$, where $\hat{t}$ is the representative tuple and $t_j, j = 1 \text{ to } m$ is any tuple inside block such that $\hat{t} \neq t_j$, the difference between them may be defined as:

$$d_B(\hat{t}_i, t_j) = \begin{cases} 0 & A_i(\hat{t}) = A_i(t_j) \ \ i = 1 \text{ to } n \\ A(t_j) & \text{otherwise} \end{cases} \tag{3}$$

For coding, the concept of vector quantization (VQ) with lossless capability is employed [17]. The quantizer Q can be seen as a combination of two functions: a coder and a decoder. The coder ξ is a mapping of $\Re^n$ into the index or codeword set $j = \{1, 2, \ldots, m_\xi\}$ and the decoder $\wp$ is a mapping of j into the output set Y. The optimal quantizer Q is the one that minimizes $\sum_{j=1}^{m} d_B(\hat{t}, t_j)$ for all input vectors (tuples). A direct application of *VQ* to encode table would be to replace each tuple in B_i with a codeword or index that indicates the representative tuple $\hat{t}$. Unfortunately, this method of coding is lossy; the original tuples are no longer completely recoverable. To deal with lossless compression, augmented *VQ* is used. Instead of replacing each tuple in a block only by its codeword as *VQ* does, it includes the difference between the tuple and its representative tuple [17].

In formal, *AVQ* encodes a tuple $t \in B$ by the pair $\langle \xi(d_B(\hat{t}_i, t_j), A_i(\hat{t}) \rangle$, where ξ is the coder (run length encoder RLE in our case) that produces the codeword (or index into the codebook). The compression efficiency of *AVQ* depends on the choice of the codebook (representative tuple in our case). If the codebook is properly designed, the average difference between a tuple and its representative tuple will be small enough that it takes fewer bits to encode than the original tuple. Run-length encoding is very effective in databases where there are long sequences of repeated zeros or missing values [20]. Simply, run-length encoding replaces sequences of identical values by a count field, followed by an identifier for the repeated value. The count field must be flagged so that it can be recognized from the other data values, and the selected sequence must have enough repeated values to warrant its replacement by the count and character fields [1]. Procedure RLE illustrates the steps of encoding D_B (difference) array that accumulates $d_B(\hat{t}_i, t_j)$ for all tuples inside block. The coding is completed when these encoded tuples of each block are concatenated as a single stream of data, along with the metadata that includes representative tuple $\hat{t}$, location of $\hat{t}$ inside block, number of tuples *m*, and number of attribute *n* for each B_i that are being placed in the front of the data steam. An example of the stream S_1 for the block B_1 is as follows:

S_1= {10, 1,2,3,4, - , #, 8, 10, 20, 30, 41, 58, 79, 83, 90, 9/0, 9, 14, 21, 35, 44, 61, 82, 86, 93, 9/0}

Procedure RLE ($\forall A_{D_b,i} \in D_{B,j}, i = 1, \ldots n \ , j = 1, \ldots k$)

Start on the first element of input $A_{D_b,i}[t], \ t = 1, \ldots m$

// *coding column-by-column*
Examine next value
If same as previous value
Keep a counter of consecutive values
Keep examining the next value until a different value or
End of input then output the value followed by the counter.
Repeat
If not same as previous value
Output the previous value followed by '1' (run length)
Repeat.

In encoding phase, each block is coded into a stream S_i that consists of numbers and signs. The first value in the stream reflects the number of tuples m. The next set of values signify the representative tuple $\hat{t}$ that is displayed between the first value in the stream and the symbol '-'; the count of these values represent the number of attribute (i.e. the stream contains all values belongs to $\hat{t}$). Excluding the value $A_{D_b,i}[t] \in \hat{t}, i = 1 to n$ in the encoded column that is represented in the stream as '#' symbol, the remaining coding is as follows: if the value $A_{D_b,i}[t]$ is not repeated, it will be shown in the stream as it else the RLE is employed to represent the repeated values in the form a/b where a denotes the counter of the repetition and b symbols the value itself. After finishing encoding the first column, the same coding style is utilized for the residual columns in the block.

3.7. Decompression Step

For decompression, the decoder follows a slightly different algorithm than the encoder. Given the compressed vector for each block S_i; the decompression process mainly divided into four levels: in the first level run length decoding is engaged to extract a/b pattern; repeating the b's value a times. The second level includes building the block's matrix through putting the stream values $A_{D_b,i}[t]$ after '-' symbol to the matrix column by column according to the first value in S_i. Regarding '#' symbol, the decoder replaces the location of '#' symbol with representative tuple $\hat{t}$ in a row manner. In the third level the values of zero in the matrix are replaced with their equivalent in the representative tuple $\hat{t}$. Finally, in the fourth level, the obtained values in the matrix are decoded according to the first step in the coding process (attribute encoding).

Procedure Decoder $(S_i, i = 1 \text{to } k)$

m = S_i {1}, $\hat{t}$ = substring from S_i {2} to S_i {index of symbol '-' -1}
S_{RLE} = substring from S_i beginning from '-' to end of the stream
n = number of $\hat{t}$ values, $M_{m \times n}$ = block matrix of zeros, *id*= Index of '#' symbol inside S_{RLE}
Start on the first value of S_{RLE}
1- Extract a/b pattern; repeating the b's value a times.
2- $M_{id,n} = A_{D_b,i}[t] \in \hat{t}$ // *put* representative *tuple inside M in row manner*
3- For each consecutive m values of S_{RLE} ($mS_{1 \times m,i}, i = 1 \text{ to } n$) not include '#' symbol.
4- $M_{i,j} = mS_i[1, j], \; j = 1 \text{ to } m, i \not\subset id$ // *put* $mS_{1 \times m,i}$ inside *M in column manner*
5- Decode $M_{i,j}$ values according to attribute encoding step

4. Performance Analysis and Results

In this section, we present performance study for our algorithm. The performance study consists of two parts. First, a substantial storage saving can be achieved by using the proposed semantic compression

algorithm. The compression performance is compared with state-of-the–art semantic lossless compression algorithm (SPACK) suggested by H. Huang [2] on three real life datasets, which are summarized in Table 2. Second, we report the efficiency of the algorithm.

Table 2. Datasets Characteristics

Dataset	Records	Attributes
Census	99,762	41
Connect4	67,557	43
Retail	260,000	28

All the experiments are performed on an Intel(R)Processor Core(TM) i3-2330M CPU @ 2.20GHz (4 CPUs), 2.2GHz with 4 GB of main memory, running Microsoft Windows 8. The algorithms are coded in PHP and MySQL. Our work focuses primarily on optimizing the compression ratio that achieves the maximum possible reduction in the size of the data. The larger the compression ratio, the more effective the compression. The most popular method of measuring the performance of a compression technique is the compression ratio, *CR* [1]:

$$CR = \left(1 - \frac{T_O}{T_C}\right)\%, \tag{4}$$

where T_O symbols original table size and T_C is the compressed table size. Fig. 2 shows the compression ratios for the proposed system and *SPACK* system on the three real datasets. The figure clearly indicates that our system outperformed *SPACK* on all datasets. We can explain these results on the basis that the proposed system uses Apriori algorithm to extract frequent pattern having the maximum gain and employs two level of coding (attribute encoding and run length coding) to increase *CR*. In contrast, *SPACK* algorithm iteratively compresses the data with the pattern having the maximum gain through utilizing *FP* growth algorithm. Both approaches are effective for handling the large datasets that has many attributes.

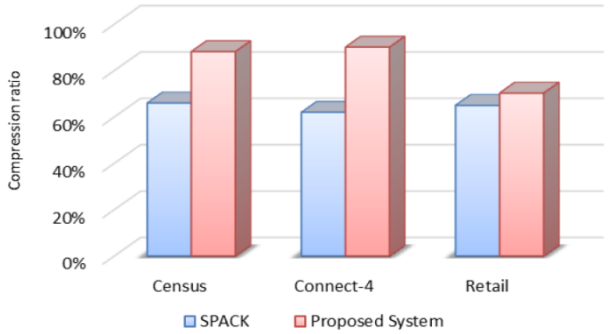
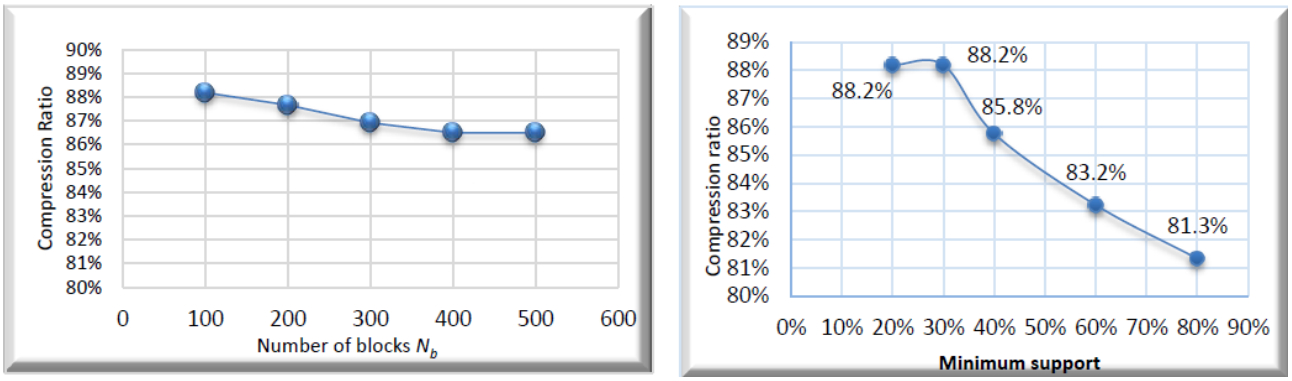


Fig. 2. Comparative results.

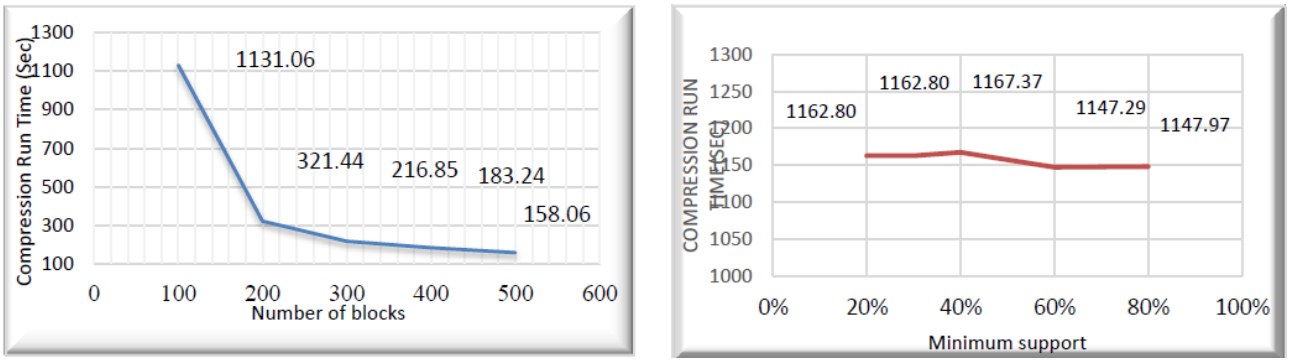
Having compared our system to another compression algorithm, we will next investigate the effect that different parameter settings have on our system, which includes both numbers of blocks N_b and *min_sup*. To keep the number of parameter setting combinations small, we will only vary the setting for one parameter at a time while keeping the setting for another parameter to its default value. In Fig. 3 (a), we vary N_b from 100 to 500 and look at the compression ratio achieved by our system on the Census dataset. From the graph, we can see that increasing N_b will diminish the compression ratio. This decreasing is negligible. This lessening is due to the fact that higher number of blocks leads to reduce the number of tuples in each block

and thus reduces the chances of repeating tuples used to mark the frequent patterns. However, we also note that the storage for representative rows will increase with N_b. Thus if N_b is subsequently increased to an extreme value where the reduction in outlying values is not enough to offset the additional storage need for the representative rows, then the compression ratio will decrease instead. Fig. 3 (b) shows the effect of increasing *min_sup* ratio on compression ratio; as we can see their exit an inverse relationship.



(a) (b)

Fig. 3. (a) Effect of N_b on compression ratio. (b) Effect of *min_sup* on compression ratio for census dataset.



(a) (b)

Fig. 4. (a) Effect of N_b on compression run time. (b) Effect of *min_sup* on compression run time for censzzus table under 100 blocks.

Having seen how the parameter settings affected the compression ratio of the system, we will examine the effect of parameter settings on the compression running time. Here, all running times are obtained from the same set of experiments. Fig. 4 (a) shows the effect of changing the number of blocks N_b on the time used for compression in the case of *min_sup* =20%. We note that the time required for the compression decreases with increasing the number of blocks. This is because, increasing the number of blocks resulting in a small number of tuples within the block and therefore less processing is required. But there is no noticeable effect on the compression time in the case of *min_sup* percentage change, as shown in Fig. 4(b), this is because the number of frequent patterns within each block are small and converged in time that are needed for the extraction process. In general, the required time for compression is reasonable for online applications.

5. Conclusions and Discussions

Semantic encoding is a new, patented technology that greatly increases the compression ratio of database. This technology utilizes frequent dependency patterns embedded in the relational table to reduce storage requirements. In this paper we have described a novel database compression system that exploits attribute

semantics and data-mining model to effectively compress massive data tables. In order to apply frequent patterns effectively on the semantic compression, the compression gain of a frequent pattern is defined as the coverage of the pattern over the table, and is used for pattern selection in the compression framework.

To increase compression ratio, this paper reports a lossless compression technique based on augmented vector quantization (*AVQ*). *VQ* is a data compression technique with wide applicability in speech and image coding, but it is not directly suitable for databases because it is lossy. We show how anyone may use a lossless version of vector quantization to reduce database space storage requirements and improve disk I/O bandwidth. The new design does not impose any changes on the tables in the database structure. The primary benefits of the proposed system are: (1) the system performs compression/decompression on the block-level. For this reason, compression can be easily integrated into database systems. (2) The system provides significant compression ratio so that making it a cost-effective compression technique. (3) The compression efficiency of the proposed system not depends on the choice of the codebook. (4) The system decreases the time overhead for both compression and decompression for large tables.

The proposed system achieves a better result than others lossless semantic compression in the literature. However, semantic compression is limited by the dataset being compressed, as some datasets may not show relationships between columns, although this tends to be the exception rather than the common case. There are some interesting research issues related to integrate the compression technique proposed herein into database systems, including the incremental updates, the design of the physical layout of the data page of the compression data, and the indexing structure on the compressed table.

References

[1] Roth, M. A., & S. Horn, J. V. (1993). Database compression. *Journal of Association for Computing Machinery*, *22(3)*, 31-39.

[2] Huang, H. (1998). *Lossless Semantic Compression for Relational Databases*. Ph.D. Dissertation, Simon Fraser University, Canada.

[3] Babu, S., Garofalakis, M., & Rastogi, R. (2001). SPARTAN: A model based semantic compression system for massive data tables. *Proceedings of the International Conference on Management of Data* (pp. 283–294).

[4] Jagadish, H. V., Ng, R. T., Ooi, B. C., & Tung, A. H. (2004). It compress: An iterative semantic compression algorithm. *Proceedings of the International Conference on Data Engineering* (pp. 646 – 657).

[5] Benjamin, D. P., & Walker, A. (2005). Semantic encoding of relational databases in wireless networks. *Proceedings of the International Conference on Data Mining, Intrusion Detection, Information Assurance and Data Networks Security* (pp. 255-262).

[6] Wu, W. B., & Ravishankar, C. V. (2003). The performance of difference coding for sets and relational tables. *Journal of the ACM*, *50(5)*, 665-693.

[7] Habib, A., Hoque, A. L., & Rahman M. S. (2012). High performance query operations on compressed database. *International Journal of Database Theory and Application*, *5(3)*, 1-14.

[8] Abadi, D., Madden, S. R., & Ferreira, M. C. (2006). Integrating compression and execution in column-oriented database systems. *Proceedings of the International Conference on Management of Data* (pp. 671-682).

[9] Ng, W. K., & Ravishankar C. V. (1997). Block-oriented compression techniques for large statistical databases. *IEEE Transaction on Knowledge and Data Engineering*, *9(2)*, 314-328.

[10] Jagadish, H. V., Madar, J., & Ng, R. T. (1999). Semantic compression and pattern extraction with fascicles. *Proceedings of the International Conference on very Large Data Bases* (pp. 186-198).

[11] Raichand, P., & Aggarwal, R. R. (2013). A short survey of data compression techniques for column

oriented database. *Journal of Global Research in Computer Science*, *4(7)*, 43-46.

[12] Karumbunathan, A. (2013). *Using Predictive Models for Compression in Database Systems* (Report No. CIT 367). Brown University, USA.

[13] Iyer, B. R., & Wilhite D. (1994). Data compression support in databases. *Proceedings of the 20th International Conference on very Large Data Bases* (pp. 695-704).

[14] Tamrakar, A., & Nanda, V. (2012). Comparison of different security and compression techniques of relational database. *International Journal of Advanced and Innovative Research*, *1(2)*, 1-4.

[15] Muthukumar, M., & Ravichandran, T. (2012). Analysis compression performance for real time database systems. *International Journal on Advanced Computer Theory and Engineering*, *1(2)*, 17-22.

[16] Benjamin, D. P., & Walker, A. (2004). *Semantic Encoding and Compression of Database Tables*. U.S. Patent 6,691,132 in Reengineering LLC, USA.

[17] Ng, W. K., & Ravishanka, C. V. (1995). Relational database compression using augmented vector quantization. *Proceedings of the Eleventh International Conference on Data Engineering* (pp. 540 – 549).

[18] Atayero, A. A., Alatishe, A. A., & Olugbara, O. O. (2011). Compression of high-dimensional data spaces using non-differential augmented vector quantization. *International Journal of Information and Communication Technology Research*, *1(8)*, pp. 329-336.

[19] Bettini, C. (2001). Semantic compression of temporal data. *Proceedings of the Second International Conference on Advances in Web-Age Information Management* (pp. 267-278).

[20] Murugesan, M., & Ravichandran, T. (2013). Real time database compression optimization using iterative length compression algorithm. *Proceedings of International Conference on Computer Science & Information Technology* (pp. 99–105).

Saleh Mesbah Elkaffas is an associate professor at the Department of Information System, College of Computing & IT, Arab Academy for Science, Technology & Maritime Transport (AASTMT). He is currently the director of the Remote Sensing & Spatial Studies Unit, AASTMT. He received his B.Sc. in electronics & telecommunications engineering, M.Sc. and Ph.D. from Alexandria Univ. His research interests include digital image processing, satellite remote sensing, and GIS.

Saad M. Darwish received his Ph.D. degree from the Alexandria University, Egypt. His research and professional interests include image processing, optimization techniques, security technologies, and machine learning. He has published in journals and conferences and severed as TPC of many international conferences. Since Feb. 2012, he has been an associate professor in the Department of Information Technology, Institute of Graduate Studies and Research, Alexandria University, Egypt.

Omar A. Abdulateef received the B.Sc. degree in computer science from the Department of Computer Science, College of Computers, University of Alanbar, Iraq in 2005. Currently he is a M.Sc. student in the Department of Information Technology, Institute of Graduate Studies and Research, Alexandria University, Egypt. His research and professional interests include database compression and intelligent systems.