

Efficient compression of text attributes of data warehouse dimensions

Jorge Vieira¹, Jorge Bernardino², Henrique Madeira³

¹Critical Software S.A.

jvieira@criticalsoftware.com

²CISUC-ISEC, Instituto Politécnico de Coimbra

jorge@isec.pt

³CISUC-DEI, Universidade de Coimbra

henrique@dei.uc.pt

Abstract. This paper proposes the compression of data in Relational Database Management Systems (RDBMS) using existing text compression algorithms. Although the technique proposed is general, we believe it is particularly advantageous for the compression of medium size and large dimension tables in data warehouses. In fact, dimensions usually have a high number of text attributes and a reduction in their size has a big impact in the execution time of queries that join dimensions with fact tables. In general, the high complexity and long execution time of most data warehouse queries make the compression of dimension text attributes (and possible text attributes that may exist in the fact table, such as false facts) an effective approach to speed up query response time. The proposed approach has been evaluated using the well-known TPC-H benchmark and the results show that speed improvements greater than 40% can be achieved for most of the queries.

1 Introduction

Over the last decades, the volume of data stored in databases has grown at a very high pace. Despite the fact that the evolution of storage capacity has followed this growth, a similar increase of disk access performance has not happened. On the other hand, improvements in speed of RAM memories and CPUs have outpaced improvements in physical storage devices by orders of magnitude. This technological trend led to the use of data compression, trading some execution overhead (to compress and decompress data) for the reduction of space occupied by data.

In a compressed database data is stored in compressed format on disk and is either decompressed immediately when read from disk or during query processing. In databases, and particularly in data warehouses, the reduction in the size of the data obtained by compression represents normally a gain in speed, as the extra cost in execution time (to compress and decompress the data) is compensated by the reduction in size of the data that have to be read/stored in the disks.

Data compression in data warehouses is particularly interesting for two main reasons: **1)** the amount of data normally stored in the data warehouse is very large,

potentially allowing considerable gains in data size, and 2) the data warehouses are used for querying only (i.e., only read accesses, as the data warehouse updates are done offline), which means that compression overhead is not relevant. Furthermore, if data is compressed using techniques that allow searching over the compressed data, then the gains in performance could be quite significant, as the decompression operation are only done when is strictly necessary.

In spite of the potential advantages of compression in databases, most of the commercial relational database management systems (DBMS) either do not have compression or just provide data compression at the physical layer (i.e., database blocks), which is not flexible enough to become a real advantage. Flexibility in database compression is essential, as the data that could be advantageously compressed is frequently mixed in the same table with data whose compression is not particularly helpful. Nonetheless, recent work on attribute-level compression methods has shown that compression can improve the performance of database systems in read-intensive environments such as data warehouses [1, 2].

Data compression and data coding techniques transform a given set of data into a new set of data containing the same information, but occupying less space than the original data (ideally, the minimum space possible). Data compression is heavily used in data transmission and data storage. In fact, reducing the amount of data to be transmitted (or stored) is equivalent to the increase of the bandwidth of the transmission channel (or the size of the storage device).

The first data compression proposals appeared in the 40's, namely proposed by D. Huffman, but these earlier proposals have evolved dramatically since then. In [3] is presented a survey of data compression covering all the period from the first proposals until the end of the 80's. More recent proposals can be found in [4].

The main emphasis of previous work has been on the compression of numerical attributes, where coding techniques have been employed to reduce the length of integers, floating point numbers, and dates [1, 5]. However, string attributes (i.e., attributes of type CHAR(n) or VARCHAR(n) in SQL) often comprise a large portion of database records and thus have significant impact on query performance.

In this paper we propose a flexible compression approach that allows the compression of data in Relational Database Management Systems (RDBMS) using existing text compression algorithms. This solution is specially appropriated for the compression of large dimension tables in data warehouses, as these tables usually have a high number of text attributes. The high complexity and long execution time of most data warehouse queries make the compression of dimension text attributes an effective approach to speed up query response time as shown by the experimental results obtained using TPC-H benchmark, where speed improvements greater than 40% have been observed.

The structure of the paper is as follows. Section 2 briefly summarizes the state of the art of data compression and coding techniques. Section 3 describes the proposed technique and section 4 evaluates experimentally the gain in space and performance obtained with the proposed technique. Section 5 concludes the paper.

2 Related Work

Data compression has been a very popular topic in the research literature and there is a large amount of work on this subject. The most obvious reason to consider compression in a database context is to reduce the space required in the disk. However, a maybe more important issue is whether the processing time of queries can be improved by reducing the amount of data that needs to be read from disk using a compression technique. There has been much work on compressing database indexes [6, 7] but less on compressing the data itself. With respect to compression of non-index data, traditional techniques such as Huffman coding [8] and Lempel-Ziv [9] work well for compressing certain types of data, such as medical images [10] or text [11], but are not advantageous to compress string fields in a database due to high execution costs. There are other algorithms for compressing numeric data [2, 12]. However, despite the abundance of string-valued attributes in databases, most existing work has focused on compressing numerical attributes [2, 6, 7, 12].

Recently, there has been a revival of interest on employing compression techniques to improve performance in a database. The data compression currently exists in the main databases engines, being adopted different approaches in each one of them.

Data compression in Oracle [13] is based on an algorithm specifically designed for relational databases that uses compression at data block level. In each block is created a table of symbols with correspondence to a dictionary of the attributes that are the compression target. The attributes are replaced in the block by pointers (links) for the table of symbols. The compression is only achieved in complete columns, however it can be used compression between columns (use the same value of the dictionary for different columns of the block) or of sequences of columns, when such could be advantageous. However, the use of this type of compression has a significant impact in data loading time and update time that increase significantly. To optimize these operations a table can be divided in temporal partitions and compressing only a partition when all estimate updates in this partition are done.

Teradata [14] presents a compression algorithm at the attribute level where compression candidates are all the attributes of fixed size that are not part of the existing index on the primary key of the table, with special advantage for columns with low cardinality. The compressed values are stored in a dictionary in the table header. This dictionary is constructed when tables are created or columns are added to an existing table, being the values to compress indicated by the user. The main advantage of this type of compression is the reduction of occurrences of decompression of the values, as decompression is only done when the compressed attributes are necessary for the construction of the query result (lazily decompressed).

IBM DB2 [15] is based on Lempel-Ziv compression algorithm, to make compression of lines. The compression algorithm is adapted accordingly to the data to compress, being the analysis of the data carried on samples. The compression can be made on all the tablespace or only on a partition. Each page has an anchor that indicates if that page is compressed or not. In the same way, each line has a bit that indicates if it is compressed or not. All the lines that are necessary for the execution of a query must be decompressed. In order to optimize the compression and decompression of data, a processor of compression by hardware can be used.

Sybase IQ [16] stores the data in a different form of the usual RDBMS. The data of one table is stored by columns instead of the traditional storage per line. The data is indexed in arrays of bits for each column called Bit-Wise indexes. The index can contain all the distinct values if the values are just a few (less than 1000), or slices of values for columns with many distinct values. The particular form of storage of data is in itself the technique of compression used in Sybase IQ. This compression is obtained by two forms: elimination of the repeated values and reduction of the space occupied by the indexes. With Sybase IQ we only access data columns needed for the query. The data loading operations become heavier due to different form to store the data, however do not exist performance loss due to compression of the data.

The compression of data in databases offers two main advantages: less space occupied by data and potentially better query response time. If the benefit in terms storage is easily understandable, the gain in performance is not so obvious. This gain is due to the fact that less data had to be read of the storage, which is clearly the most time-consuming operation during the query processing.

The most interesting use of data compression and codification techniques in databases is surely in data warehouses, given the huge amount of data normally involved and its clear orientation for the query processing. As in the data warehouses all the insertions and updates are done during the update window [17], when the data warehouse is not available for users, off-line compression algorithms are more adequate, as the gain in query response time usually compensates the extra costs to codify the data before being loaded into the data warehouse. In fact, off-line compression algorithms optimize the decompression time, which normally implies more costs in the compression process. The technique presented in this paper follow these ideas, as it takes advantage of the specific features of data warehouses to optimize the use of traditional text compression techniques.

3 Attribute Compression

As mentioned before, some of the major commercial databases already offer compression mechanisms. However this compression features are generic and, consequently, not optimized to any specific scenario. Our approach is especially designed for data warehouses, particularly to compress dimension tables, which are usually composed by a large number of textual attributes with low cardinality. This approach also has the benefit of being independent from the RDBMS used. Therefore, using our approach we can implement data compression in databases that do not offer this option, such as Microsoft SQL Server or PostgreSQL. The use of this technique can also be combined with compression techniques already existing in some database engines, as the compression offered by Oracle 9iR2 and 10g.

The main objective of this technique is to allow the reduction of the space occupied by dimension tables with high number of rows, reducing the total space occupied by the data warehouse and leading to a consequent gains on performance. In fact, by reducing the size of large dimensions the star model becomes closer to the ideal star model proposed by R. Kimbal [17], with a consequent speed improvement.

The proposed approach aims to compress two different types of database attributes:

- Text attributes with low cardinality (referred further on as *categories*).
- Attributes of free text (comments or notes) that are mainly used for visualization (referred further on as *descriptions*).

Categories are textual attributes with low cardinality. Examples of category attributes are: city, country, type of product, etc. The codification of categories is made using 1 or 2 bytes, depending on the category cardinality. For categories with less than 256 distinct values the use of 1 byte is enough, the use of 2 bytes allows to codify categories with a maximum of 65791 (65535+256) distinct values, where the 256 most used are represented by an one byte code and the remaining less used values are represented by a two bytes code.

A description is a text attribute that is mainly used for visualization. In other words, it is not usually used to restrict, to group, or to order a query. An example of a description attribute is the comment attribute, which is frequently found in dimension tables. This type of attributes has the particularity of having a low access frequency and it is only necessary to decode it when the final result is being constructed. As this is a free text attribute its cardinality tends to be very high (attribute values are normally different one from each other), therefore using a codification similar to the one used for categories would result in an increase of the space occupied by the data. Therefore, for this type of attributes we propose the use of searchable text compression algorithms, which allow reducing the size of this type of attributes maintaining the ability of querying using this attributes to restrict the results. It is worth nothing that the codification of attributes does not imply any modifications at the physical structure of the database. The only change that may have to be made is the transformation of the CHAR attributes into VARCHAR attributes.

In order to guarantee that the data analysis applications (On-Line Analytical Processing - OLAP tools) continue to work transparently we propose the implementation a middleware to handle the compression and decompression of the attributes (using an approach similar to the one we have used in [18, 19]). That is, this middleware will perform query rewriting based on the metadata about the compressed attributes. The query originally received from the OLAP tool is intersected by the middleware and the values corresponding to compressed attributes are compressed before submitted the (modified) query to the DBMS. The query is executed normally and the result is decompressed only in the cases where compressed dimension attributes appear in the Select statement in the query (this will be detailed further on). This means that from the database point of view no changes are required as the data warehouse store searchable compressed attributes and the DBMS receives queries from the middleware with the values of attributed that are stored in a compressed form also compressed, which assure that the query is executed correctly.

3.1 Categories Coding

Categories coding is done through the following steps:

1. The data in the attribute is analyzed and a frequency histogram is build.
2. The table of codes is build based on the frequency histogram: the most frequent values are encoded with a one byte code; the least frequent values

are coded using a two bytes code. In principle, two bytes are enough, but a third byte could be used if needed.

3. The codes table and necessary metadata is written to the database.
4. The attribute is updated, replacing the original values by the corresponding codes (the compressed values).

Table 1 presents an example of typical attributes of a client dimension in a data warehouse, which may be a large dimension in many businesses (e.g., e-business). For example, we can find several attributes that are candidates to coding, such as: CUST_FIRST_NAME, CUST_LAST_NAME, CUST_MARITAL_STATUS, CUST_POSTAL_CODE and CUST_CITY.

Assuming that we want to code the CUST_CITY attribute, an example of possible resulting codes table is shown in Table 2. The codes are represented in binary to better understand the idea. As the attribute has more than 256 distinct values, we will have codes of one byte to represent the 256 most frequent values (e.g. Berlin and Copenhagen) and codes of two bytes to represent the least frequent values (e.g. Dublin and Oporto). The values shown in Table 2 (represented in binary) would be the ones stored in the database, instead of the larger values. For example, instead of storing “Copenhagen”, which corresponds to 10 ASCII chars, we just stores one byte with the binary code “00000111”.

Table 1. Example of typical customer attributes and cardinality

Attribute	Type	Cardinality
CUST_ID	NUMBER	100000
CUST_FIRST_NAME	VARCHAR2(20)	800
CUST_LAST_NAME	VARCHAR2(40)	640
CUST_GENDER	CHAR(1)	2
CUST_YEAR_OF_BIRTH	NUMBER(4)	74
CUST_MARITAL_STATUS	VARCHAR2(20)	5
CUST_STREET_ADDRESS	VARCHAR2(40)	99435
CUST_POSTAL_CODE	VARCHAR2(10)	623
CUST_CITY	VARCHAR2(30)	620
CUST_COMMENT	VARCHAR2(200)	95444

Table 2. Codes Table example

Values	Code (in binary)
Berlin	00000101
Copenhagen	00000111
Dublin	00000001 00000010
Lisbon	00001001
London	00001011
Oporto	00000001 00000011
...	...

3.2 Descriptions Coding

Descriptions coding is very similar to the categories coding with the major difference that in this case the value in the attribute is not regarded as a single value, but as a set of values (an ASCII string). Any text compression algorithm can be used to perform this type of compression, provided that it allows approximated (i.e., using text wild card) and exact search without decompression. Some text compression formats, like the ones presented in [20] and [21], allow this type of compression.

The key point to understand the coding approach used is that the compression algorithm includes the construction of a codes table similar to the one used in categories coding. Table 3 presents an example of the comment attribute of a typical customer table. In order to compress this attribute we first have to merge the values of all rows into a single text value, and then apply the compression algorithm in order to obtain a codes table, similar to the one presented in Table 4. Putting all the values in the comment attribute in a virtual same text file is needed to facilitate the determination of the frequency of each word. Again, as in category coding, we represent the most frequent word by one byte and the less frequent words by two bytes. If needed, we use a third byte for the even less frequent words.

After obtaining the codes table we must apply it, compressing the values in the table. The final result will depend on the compression algorithm used.

Table 3. Description attribute example

CUST_ID	CUST_COMMENT
1	The amount of time is not enough for processing
2	A quiz should be sent to this client
3	The client address is not complete
4	This client usually buys items in more than one store
5	This client should be deleted
6	There is something wrong whit the name
...	...

Table 4. Description codes table example

Values	Code
The	00000101
On	00000111
client	00001001
Name	00001011
This	00000001
...	...

As this type of attributes, when they exist, tend to occupy a large part of the table total space, their compression allows in general an impressive reduction in the size of the target table, while keeping the ability to search in the compressed values.

3.3 Query execution

As mentioned, when using this compression approach the queries must be executed through a middleware that performs query rewriting and data decompression when necessary. In order to optimize the tasks of query rewriting and decompression the middleware has the codes metadata tables loaded in memory. In fact, this work as a small dictionary to allow translation from uncompressed values to compressed values and vice-versa.

Query rewriting is necessary in queries where the coded attributes are used in the WHERE clause for filtering. In these queries the values used for filter the result must be replaced by the correspondent coded values. Following are some simple examples of the type of query rewriting needed.

Example 1. The value 'LONDON' is replaced by the corresponded code, fetched from the codes table, shown in Table 2.

Original Query	Modified Query
SELECT CUST_NAME FROM CUSTOMERS WHERE CUST_CITY = 'LONDON'	SELECT CUST_NAME FROM CUSTOMERS WHERE CUST_CITY = 00001011

Note that the code is represented in binary (1 byte).

Example 2. The value 'L%' is replaced by the set of codes that exist in the codes table of Table 2 and that verify the condition.

Original Query	Modified Query
SELECT CUST_NAME FROM CUSTOMERS WHERE CUST_CITY like 'L%'	SELECT CUST_NAME FROM CUSTOMERS WHERE CUST_CITY in (00001001,00001011)

Example 3. The values 'the' and 'client' are replaced by the correspondent compressed values, fetched from the codes table, shown in Table 4.

Original Query	Modified Query
SELECT CUST_NAME FROM CUSTOMERS WHERE CUST_COMMENT like '%the%client%'	SELECT CUST_NAME FROM CUSTOMERS WHERE CUST_COMMENT like '%00000101%00001001%'

In queries where the coded attributes are not used for filtering, it's not necessary to perform query rewriting.

3.4 Decompression

The decompression of the attributes is only made when the coded attributes are in the query select list. In these cases the query is executed and after that the result set is

processed in order to decompress the attributes that contain compressed values. As the typical data warehousing queries return small result sets the decompression time will represent a very small amount of the total query execution time.

4 Experimental Results

The goal of the experiments performed is to measure experimentally the gains in storage and performance obtained using the proposed technique. We have implemented a simplified version of the middleware needed to compress and decompress the data and we used the TPC-H [22] performance benchmark as experimental setup. The size of the database was 1 GB (scale factor 1 in the TPC-H scaling rules). After analyzing the TPC-H schema we compressed the biggest dimensions (Orders, Part and Customer) and the fact tables (Lineitem and Partsupp).

The experiments were divided in two phases. In the first phase only categories compression was used. In the second phase we used categories compression in conjunction with descriptions compression.

4.1 Categories Coding Results

The three biggest dimensions (Orders, Part and Customer) and the Lineitem fact table were compressed using categories compression. Lineitem table was compressed, although it is a fact table, because this table has two degenerated dimensions that are text attributes with very low cardinality.

Table 5 presents the storage gains obtained after compressing the tables using categories compression. An average compression gain of 24.5% was obtained.

Table 5. Categories compression gains

<i>Table</i>	<i>Initial Size (MB)</i>	<i>Categories Compression</i>	
		<i>Size (MB)</i>	<i>Gain (%)</i>
Lineitem	856	640	25.2%
Orders	192	152	20.8%
Part	31	17	45.1%
Customer	28	26	7.1%
Total	1107	835	24.5%

4.2 Descriptions coding results

In TPC-H schema all the tables have a textual field used for store comments. As we saw before this kind of data can be compressed using searchable text compression algorithms. As these fields are substantially large, they represent a huge percentage in the total size of the tables.

Table 7 presents the gains obtained in storage after applying the descriptions compression in the biggest tables. This compression was applied after the categories compression and has resulted in an average compression ratio of 39.9%.

The text compression was made using a simple algorithm where the 127 most frequent words are coded with a character in the range of 0 (00000000) to 127 (01111111) and the less frequent words are coded with two characters, the first one in the range of 128 (10000000) to 192 (11000000) and the second one in the range of 193 (11000001) to 255 (11111111). This algorithm is limited for the compression of 4223 different words ($127 + 64 \cdot 64$) and enables the search of words within the compressed text. Obviously, it is easily extended to the compression of more than 4223 different words by using a third coding character.

Table 6. Categories and descriptions compression gains

<i>Table</i>	<i>Initial Size (MB)</i>	<i>Attributes Compression</i>		<i>Descriptions Compression</i>	
		<i>Size (MB)</i>	<i>Gain (%)</i>	<i>Size (MB)</i>	<i>Gain (%)</i>
Lineitem	856	640	25.2%	536	37.3%
Partsupp	136	136	0%	72	47%
Orders	192	152	20.8%	104	45.8%
Part	31	17	45.1%	15	51.6%
Customer	28	26	7.1%	19	32.1%
Total	1243	971	21.8%	746	39.9%

This simple algorithm was chosen for its easiness of implementation and it does not offer the best compression ratio possible. As it is not optimal, the speedup that may be obtained will be a conservative value. Other existing compression algorithms, like the ones presented in [20] and [21] can be used to optimize the compression ratio.

4.3 Performance results

In order to evaluate the performance speedup obtained with the compression performed a subset of the TPC-H queries were executed with the following configurations:

1. No compression
2. Categories compression
3. Categories compression and descriptions compression

Table 7 presents the execution time in seconds of each query in the three configurations. We do not show the SQL of the TPC-H queries used for space reasons (some queries are quite long) but all the queries can be found in [22]. The use of categories compression resulted in an average speedup of 24,6%. The use of categories compression and descriptions compression resulted in an average speedup of 41,8%.

As can be observed in the results shown in Figure 1, all the queries suffered a reduction in execution time when we use categories compression and the query execution time is considerably reduced when we use categories in conjunction with description compression.

Table 7. Queries execution times in seconds

Query	<i>No compression</i>	<i>Categories</i>	<i>Categories+Descriptions</i>
Q1	47	38	30
Q2	46	34	25
Q3	50	38	28
Q4	39	28	20
Q5	52	40	31
Q6	49	35	28
Q7	52	43	28
Q8	40	31	24
Q9	80	60	46
Q10	52	34	33
Total time	507	382	295
Speedup(%)	-	24,6%	41,8%

5 Conclusions

This paper proposes and evaluates a general approach that allows the compression of data in RDBMS, which is particularly advantageous for the compression of medium size and large dimension tables in data warehouses. In fact, large dimensions usually have a high number of text attributes and a reduction in the size of middle or large dimension have a big impact in the execution time of queries that join that dimension with the fact tables. In general, the high complexity and long execution time of most data warehouse queries make the compression of dimension text attributes and possible text attributes that may exist in the fact table an effective approach to speed up query response time.

The proposed technique includes coding of two types of dimension attributes: categories and descriptions. The former are attributes with low cardinality (typically, text attributes) while the latter are attributes such as comments and descriptions (free text attributes). This approach also has the benefit of being independent from the RDBMS used. Therefore, using our approach we can implement data compression in databases that do not offer this option, or combined with compression techniques already existing in some database engines.

The proposed approach has been evaluated using the well-known TPC-H benchmark and the results have shown that it is possible to obtain a significant reduction of approximately 40% in the space occupied by TPC-H tables. The results also show a speedup improvement better than 40% for most of the queries.

References

1. Goldstein, J., Ramakrishna, R., Shaft, U.: Squeezing the most out of relational database systems, In Proc. of ICDE (2000) page 81.
2. Westmann, T., Kossmann, D., Helmer, S., Moerkotte, G.: The Implementation and Performance of Compressed Databases, ACM SIGMOD Record Vol 29, No 3 (2000) 55-67
3. Lelewer, D., Hirschberg, D.: Data Compression, ACM Computing Surveys (1987)
4. Data Compression Conference. DCC Home page: <http://www.cs.brandeis.edu/~dcc/index.html>.
5. Roth, M., Horn, S.: Database compression, SIGMOD Record, 22(3):31-39 (1993)
6. Gray, J., Reuter. A.: Transaction Processing: Concepts and Techniques, Morgan Kaufmann (1993)
7. Ramakrishnan, R., Gehrke, J.: Database Management Systems, McGraw Hill (2000)
8. D Huffman: A Method for the Construction of Minimum Redundancy Codes, Proc IRE, 40(9) (1952) 1098-1101
9. Ziv, J., Lempel, A.: A Universal Algorithm for Sequential Data Compression, IEEE Transactions on Information Theory, 22(1) (1977) 337-343.
10. Karadimitriou, K., Tyler, J.: Min-Max Compression Methods for Medical Image Databases, SIGMOD Record, Vol 26, No 1 (1997)
11. Moffatt, A., Zobel, J.: Text Compression for Dynamic Document Databases, IEEE Transactions on Knowledge and Data Engineering, Vol 9, No 2 (1997)
12. Chen, Z., Gehrke, J., Korn, F.: Query Optimization In Compressed Database Systems, ACM SIGMOD (2001) 271-282
13. Poess, M., Potapov, D.: Data Compression in Oracle, Proceedings of the 29th VLDB Conference (2003)
14. Morris, M.: Teradata Multi-Value Compression V2R5.0, Teradata White Paper (2002), available at <http://www.teradata.com/t/page/86995/>.
15. IBM Redbooks: DB2 V3 Performance Topics (1994) 75-90, available at <http://publib-b.boulder.ibm.com/Redbooks.nsf/RedbookAbstracts/gg244284.html?Open>.
16. Krneta, P.: Sybase Adaptive Server® IQ with Multiplex, Sybase White Paper (2000), available at <http://www.sybase.com/products/databaseservers/asiq>.
17. Kimball, R., Ross, M.: The data warehouse toolkit, 2nd edition, Ed. John Wiley & Sons, Inc (2002)
18. Bernardino, J., Furtado, P., Madeira, H.: Approximate Query Answering Using Data Warehouse Striping, Journal of Data and Knowledge Engineering, Volume 19, Issue 2, Elsevier Science Publication (2002)
19. Costa, M., Vieira, J., Bernardino, J., Furtado, P., Madeira, H.: A middle layer for distributed data warehouses using the DWS-AQA technique, 8th Conference on Software Engineering and Databases (2003)
20. Brisaboa, N., Iglesias, E., Navarro, G., Paramá, J.: An Efficient Compression Code for Text Databases, ECIR (2003) 468-481
21. Silva da Moura, E., Navarro, G., Ziviani, N., Baeza-Yates, R.: Compression: A Key for Next-Generation Text Retrieval Systems, ACM transactions on informations systems (2000) 113-139
22. TPC-H benchmark. <http://www.tpc.org/tpch/spec/tpch2.1.0.pdf>