Fixed-length String Compression for Direct Operations in Column-oriented Databases

KeYan  
School of Computer Science and Technology  
Huazhong University of Science and Technology  
Wuhan, China

Meiyi Xie  
School of Computer Science and Technology  
Huazhong University of Science and Technology  
Wuhan, China

Hong Zhu  
School of Computer Science and Technology  
Huazhong University of Science and Technology  
Wuhan, China

Abstract—Compression is one of the most important techniques in column-oriented database systems development. For fixed-length string typed columns, both heavyweight and lightweight compression schemes have limitations. In this paper, we propose a compression scheme, called FSC (Fixed-length String Compression), to achieve good compression ratio and support direct queries on compressed data without decompression in advance. The main idea of FSC is to vertically partition a fixed-length string typed column into sub-columns, which are compressed by different lightweight compression methods. Moreover, we present a search method, which are called FSC-search, to search on compressed data directly. Intensive experiments show that FSC not only achieves good compression ratio, but also improve query performance by supporting direct searching on compressed data.

Keywords—column-oriented database; fix-length string; query; compression; decompression

I. INTRODUCTION

In a column-oriented database system, each attribute is stored in a separate column such that successive values of that attribute are stored consecutively on disk. Column-stores are more suitable to compress than row-stores [1,2]. Compression techniques gradually become indispensable techniques for column-oriented database systems.

Several compression schemes are improved and applied to compress string typed columns for column-stores [3,4,5]. Nevertheless, there are some limitations in these schemes. On the one hand, heavyweight compression methods, such as Huffman [6] and LZ (Lempel-Ziv Encoding) methods [7], achieves good compression ratios. However, it is difficult to directly access compressed data before decompression [8]. Decompression of whole column leads to more CPU overhead, which may offset the saved cost of I/O from compression and degrade query performance. On the other hand, lightweight compression methods (including Run Length Encoding [9], Dictionary encoding [10], Frame of Reference encoding [11] and Bit Vector encoding [12]) support direct queries on compressed data. However, one kind of lightweight method is impossible to fit all kinds of string typed columns. For example, Dictionary encoding and Bit Vector encoding are only useful for columns with low cardinality. Especially, there are some data columns cannot be compressed by any lightweight compression schemes.

In this paper, we introduce a compression method, called FSC (Fixed-length String Compression), for fixed-length string, to obtain good compression and searching performance. Main contributions of this study are:

- We proposed and implemented FSC, a compression method for fixed-length string typed columns in column-oriented database systems. In FSC, a column is vertically partitioned into several sub-columns; then each sub-column is compressed using the fittest lightweight compression algorithms. We defined the fitness of a lightweight algorithm for a sub-column.

- We developed a searching method, called FSC-search, to support direct queries on columns compressed by FSC. Direct searching on sub-columns is supported since sub-columns are compressed using lightweight algorithms. Thus, results of searching on sub-columns composed the result of searching on the whole columns.

- Experiments are conducted to evaluate performance of compression and searching. Compression ratio, time for compression and decompression of FSC, and time for searching are evaluated in our experiments. Performance of FSC is compared with zilb.

FSC could be used to compress all kinds of fixed-length string typed data columns, especially for some columns that could not be compressed by any lightweight methods. FSC could support direct searching on compressed data, which could not be implemented in heavyweight compression methods. Our experiment results present the good performance of FSC in compression and searching.

II. RELATED WORK

Run Length Encoding (RLE) [9], where repeats of the same element are expressed as (value, run-length) pairs, is only suited for columns that have reasonable-sized runs of the same entries. Frame of Reference encoding (FOR) [11] uses a base value, a frame of reference and stores all entries as offsets from the reference value. This method is useful for numeric typed columns without wide spread values. Bit

Huffman [6] and Lempel-Ziv Encoding (LZ) [7] are typical methods in heavyweight compression scheme. Common compressors like zlib are based on these two algorithms. Although, zlib achieves better compression performance than lightweight compression schemes, and LZ join [13] support direct join operation on columns compressed by LZ compression methods, direct searching operation on columns compressed by heavyweight algorithms are difficult to implement.

III. FSC ARCHITECTURE

Given a fixed-length string typed column T with n entries, where \(|T|=n\). Each entry of T, denoted as \(T[i]\), is a string with s characters, that is \(|T[i]|=s\), where \(i=1, 2, \ldots, n\). The \(i^{th}\) entry of T can be described as: \(T[i] = <a_1, a_2, \ldots, a_n>\), where \(a_i\) is the \(j^{th}\) character in \(T[i]\) and \(j=1, 2, \ldots, s\).

A. Partitioning Column T into Sub-columns

In FSC, each entry in column T is divided into s sub-columns since characters are the basic unit in a string. A sub-column is the string that is composed of characters in the same location of different entries. Therefore, the length of a sub-column is \(n\) since there are n entries in column T. That is, the \(j^{th}\) character in each entry of T composes the \(j^{th}\) sub-column. Each sub-column can be denoted as \(T_{sub}[j] = <a_1, a_2, \ldots, a_n>\), where \(j=1, 2, \ldots, s\).

For example, a column T of telephone number has six entries, and each entry has ten characters. Thus, \(n=|T|=6\), \(s=|T[i]|=10\). As shown in figure 1, T is vertically partitioned into ten sub-columns and each sub-column has six characters. Such as, \(T_{sub}[1]=<’2’,’1’,’1’,’2’,’2’,’2’>\).

![Figure 1. A column T is vertically partitioned into sub-columns](image)

B. Compression of Sub-columns

After partitioning column T into sub-columns, the most important issue is how to compress these sub-columns. The purpose of compression is to improve performance of queries. Time required by a query on a compressed sub-column includes the time for reading data from disk, the time for searching on compressed data and the time for decompression. Good compression ratio leads to less I/O cost.

In RLE compression scheme, a run is replaced with a binary-tuple: (value, run-length), where each element of the binary-tuple is given a fixed number of bits. When \(T_{sub}(i)\) with \(n\) characters is compressed using RLE, the element value is a character (1 byte), and the element run-length is assumed to be 4 bytes. Thus, compression ratio of RLE is:

\[
\text{ratio}_{\text{rle}} = \frac{5 \times r \times c}{n}
\]

(1)

FOR uses a base value, a frame of reference and stores all values as offsets from the base value. Thus, when \(T_{sub}(i)\) is compressed using FOR, a base value is a character (8 bits), and each offset from the base value need \(\log_2 d\) bits. Thus, compression ratio of FOR encoding can be defined as:

\[
\text{ratio}_{\text{for}} = \frac{\log_2 d \times n + 8}{8 \times n}
\]

(2)

In Dictonary encoding, each character in \(T_{sub}(i)\) stores as a dictionary code. Each dictionary code needs \(\log_2 c\) bits. There are \(c\) elements in dictionary and each element is a character. That is, dictionary table needs \(c \times 8\) bits. Thus, compression ratio of Dictionary encoding is:

\[
\text{ratio}_{\text{dic}} = \frac{\log_2 c \times n + c \times 8}{8 \times n}
\]

(3)

In Bit Vector encoding, a bit-string is associated with each value with a ‘1’ in the corresponding position if that value appeared at that position and a ‘0’ otherwise. There are \(c\) bit-strings since there are \(c\) different values in the column, and each bit-string has \(n\) bits. In Bit Vector encoding, \(c\) different values and \(c\) bit-strings are stored. Thus, the compression ratio is:

\[
\text{ratio}_{\text{bv}} = \frac{c \times n + c \times 8}{8 \times n}
\]

(4)

2) Compression of Sub-columns: \(\alpha\) is a constant factor regarding disk I/O capability. \(\beta_{\text{rle}}, \beta_{\text{for}}, \beta_{\text{dic}}\) and \(\beta_{\text{bv}}\) denote the cost of retrieving on data compressed by RLE, Dictionary, FOR and Bit Vector encoding respectively. Thus, according to the equation (1), (2), (3) and (4), an approximation of the time for searching on a sub-column compressed by four kinds of compression methods can be obtained:

\[
S(X) = \text{ratio}_{x} \times n \times (\alpha + \beta_{x})
\]

(5)

where X is RLE, FOR, Dictionary or Bit Vector encoding. \(\alpha\) and \(\beta_{x}\) depend on the machines used in experiments. These
factors will be evaluated before using FSC to compress data columns.

We choose X to compress $T_{sub[i]}$ where X leads to minimum $S(X)$. There are three steps to compress a sub-column $T_{sub[i]}$:

- Scan $T_{sub[i]}$ to obtain three characteristic values, including $r_i$, $c_i$ and $d_i$
- Work out the minimum of $S(X)$ according to the equation (5).
- Choose the corresponding method X to compress $T_{sub[i]}$

3) Analysis of Time Complexity: Compression time of a column $T$, denoted as $C_T$, is the sum of compression time of all sub-columns of $T$, which are denoted as $C_T[i]$, where $i=1, 2, \ldots, s$. Thus,

$$C_T = \sum_{i=1}^{s} C_T[i]$$  \hspace{1cm} (6)

$C_T[i]$ includes the time of evaluating compression ratios and the time of compression using lightweight methods. Time complexity of evaluating compression ratios are O(n), because it needs to scan all characters in sub-column $T_{sub[i]}$. And, Time complexity of compressing using lightweight methods is O(n). Thus, $C_T[i]$ is O(n). According to equation (6), $C_T$ is $O(s \times n)$.

The decompression time of column $T$, denoted as $D_T$, is the sum of decompression time of all sub-columns of $T$, which are denoted as $D_T[i]$, where $i=1, 2, \ldots, s$. That is,

$$D_T = \sum_{i=1}^{s} D_T[i]$$  \hspace{1cm} (7)

Clearly, $D_T[i]$ is O(n), since decompression time for each lightweight compression scheme is O(n). Thus, according to equation (7), $D_T$ is $O(s \times n)$.

IV. SEARCHING ON COMPRESSED DATA

Query is one of the commonly used operations in column-oriented database systems. For example, figure 2 shows an example of an operation using the operator ‘=’ and an example of an operation using the operator ‘like’ in SQL expressions.

<table>
<thead>
<tr>
<th>category</th>
<th>pattern</th>
<th>implication</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exact Match</td>
<td>$x$</td>
<td>$s=p$ and $i=1, 2, \ldots, p$</td>
</tr>
<tr>
<td>Fixed Part Match</td>
<td>$x^*$</td>
<td>$s=p+q$ and $t_i=x_i$, where $i=1, 2, \ldots, p$</td>
</tr>
<tr>
<td>Pattern Match</td>
<td>$%x$</td>
<td>$s=p$ and $t_{i+p}=x_i$, where $i=1, 2, \ldots, p$</td>
</tr>
</tbody>
</table>

A. Categories of Searching on String

Assume that x is the searching pattern in a query, x is denoted as $<x_1x_2\ldots x_p>$. An entry t in a column T is denoted as $<t_1t_2\ldots t_s>$. There are 12 basic kinds of searching, which are listed in table 1. We group these searching into three categories: exact match, fixed part match and pattern match, according to whether the matching position is fixed or not.

<table>
<thead>
<tr>
<th>TABLE I. TWELVE BASIC KINDS OF MATCHES</th>
</tr>
</thead>
<tbody>
<tr>
<td>category</td>
</tr>
<tr>
<td>----------</td>
</tr>
<tr>
<td>Exact Match</td>
</tr>
<tr>
<td>Fixed Part Match</td>
</tr>
<tr>
<td>Pattern Match</td>
</tr>
</tbody>
</table>

B. Exact Match

In exact match, each character in an entry needs to be compared with each character in x. We don’t consider any optimization strategies in searching process, such as index. It is outside the scope of this paper. Optimization strategies can be used directly on data compressed by FSC since FSC support direct searching on compressed data.

Sub-columns $T_{sub[1]}, \ldots, T_{sub[s]}$ of a column $T$ are compressed to $T_{comp[1]}, \ldots, T_{comp[s]}$. If s is not equal to p, there is no entries in T exact matched with x. Thus, we only consider the case of s equal to p. That is, it is necessary to compare $x_i$ with each element in $T_{comp[i]}$, where $i=1, 2, \ldots, p$. Algorithm of exact match are described in figure 3. In the
algorithm, flag is an array with n elements. If flag[i] is 1, the ith entry is exact matched to x; otherwise, the ith entry is not exact matched to x.

C. Fixed Part Match

The difference between exact match and fixed part match is matching positions (including starting position and end position of matching). For example, in exact match, the starting position and end position of matching in t are 1 and p respectively; in *x pattern, the starting position and end position of matching in t is q + 1 and q + p + 1 respectively. Thus, all types of fixed part match could be processed by changing the algorithm described in Figure 3 according to the starting position and end position of matching, which are listed in table 1. The procedures of fixed part match are not repeated in this section, due to space limitations in the article.

D. Pattern Match

Pattern match is much different with exact match and fixed part match. We improved KMP [14] method to process pattern match %x% in. In figure 4, we describe the algorithm of pattern match. An array called next is calculated like KMP method in advance. The other array called flag is the comparison location in x. That is, the jth element in Tcomp[i] need to be compared with xflag[j]. After the comparison, flag[j] will be changed according to the comparison result and array next. At last, if flag[j] is equal to 0, the jth entry is pattern matched to x; otherwise, the jth entry is not pattern matched to x.

algorithm 2: pattern match x in T compressed by FSC
Data: compressed column T (Tcomp[1],…, Tcomp[s]) and x (x1,…,xs)
Result: flag[1],……,flag[n]
1: calculate array next according to KMP method
2: initialize each element in array flag to 1
3: for each character in x
4: for each element in Tcomp[i]
5: if flag[j] != 0//the comparison location in x is legal
6: if the jth element in Tcomp[i] is equal to xflag[j]
7: if xflag[j] is the last character in x
8: flag[j]=0//the jth entry is pattern matched to x
9: else
10: flag[j]=flag[j]+1 //modify the location
11: else
12: modify flag[j] according to next[flag[j]]

Figure 4. Pattern match on compressed column T

V. EXPERIMENTS

Our experiments are executed on 2.5 GHz Pentium Dual-Core, running Windows XP, with 3GB of main memory. A data generator called dbgen is used to generate an instance of the TPC-H [15] data set at scale 10, which yields a total database size of approximately 10 GB with 6 tables, named Supplier, Customer, Part, Partsupp, Orders and Lineitem respectively. We ran experiments on each fixed-length string typed column in these tables.

Before the evaluating the performan FSC, we found that, \( \beta_r \), \( \beta_{loc} \), \( \beta_{ac} \) and \( \beta_{bit} \) are the same with each other since we retrieve each element of a sub-column sequentially. \( S(X) \), which can be calculated according to equation (5), is only depends on ratioX, where X is one method of RLE, FOR, Dictionary or Bit Vector encoding. ratioX can be calculated according to equation (1), (2), (3) and (4).

A. Data Columns

There are 7 fixed-length typed columns in the instance of the TPC-H data sets in all. For simplification, we give each data column an identifier. In table 2, we list features of each column, including identifier, attribute name, source (i.e. which data set a column belongs to), scale (i.e. the number of entries in a column) and length (i.e. the length of each entry in a column). For example, the column with identifier 1 is an attribute called s_phone in data table Supplier; this column contains 100000 entries, and each entry contains 15 characters.

<table>
<thead>
<tr>
<th>identifier</th>
<th>name</th>
<th>source</th>
<th>scale</th>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>s_phone</td>
<td>supplier</td>
<td>100000</td>
<td>15</td>
</tr>
<tr>
<td>2</td>
<td>s_name</td>
<td>supplier</td>
<td>100000</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td>p_brand</td>
<td>part</td>
<td>200000</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>c_phone</td>
<td>customer</td>
<td>1500000</td>
<td>15</td>
</tr>
<tr>
<td>5</td>
<td>c_name</td>
<td>customer</td>
<td>1500000</td>
<td>18</td>
</tr>
<tr>
<td>6</td>
<td>p_mfgr</td>
<td>part</td>
<td>2000000</td>
<td>14</td>
</tr>
<tr>
<td>7</td>
<td>o_clerk</td>
<td>orders</td>
<td>1500000</td>
<td>15</td>
</tr>
</tbody>
</table>

Lightweight compression schemes are not suitable to be used directly in string typed columns. Dictionary encoding is improved to compressed string typed columns, such as [3]. However, it is useful for columns with low cardinality. Zlib is a commonly used heavyweight compression scheme that can be used directly in column-oriented database systems and obtain good compression performance. Thus, we use FSC and zlib to compress and decompress each column separately, and evaluate the time for compression, decompression and searching.

B. Compression Ratios

We use FSC and zlib to compress all of the data columns listed in table 2 respectively. Source code of zlib is downloaded from [16]. Figure 5 shows the sizes of each column that compressed by zlib and FSC respectively. It can be concluded that FSC achieves better compression ratio than zlib for fixed-length string typed columns.

Figure 5. Data size of compressed columns
C. Time for Compression and Decompression

Besides compression ratio, encoding and decoding speed are important factors in many applications. Thus, we evaluate the compression and decompression time of zlib and FSC scheme respectively. In order to clearly show the performance comparison between zlib and FSC, compression, decompression and searching time of the first two columns in latter figures are higher than actual.

1) Time of Compression: Time for compression is shown in Figure 6. There is no doubt that FSC cost less time for compression than zlib.

![Figure 6. Time for compression](image)

2) Time of Decompression: Time for decompression is shown in Figure 7. Decompression speed of FSC is slower than zlib for most columns. FSC decompresses column 1 and column 4 faster than zlib. These two columns are telephone number columns, and entries are much different with each other. The other columns are name columns, and entries in these columns are very similar. For example, each entry in column 3 is “Brand#××”, and only the last two characters are different.

![Figure 7. Time for decompression](image)

Although FSC spent more time for decompression than zlib, it can be offset by saved cost of I/O and direct retrieving on compressed data. These are illustrated in experiments latter.

D. Time for Searching

Operations of searching on strings are grouped into three categories in section 4. Thus, in this section, we implement these categories of searching on each column and measure the time of searching. For FSC, we search on compressed data directly; and for zlib, compressed columns are decompressed before being searched.

1) Searching Time of Exact Match: Searching time of exact match on string columns compressed by FSC includes the time of reading compressed data from disk and the time of retrieving on compressed data. However, for searching on string typed column compressed by zlib, it is necessary to decompress before retrieving.

![Figure 8. Searching time of exact match](image)

Figure 8 presents that FSC achieves better performance than zlib for exact match. There are three reasons:
- FSC obtains better compression ratio than zlib, which saves cost of I/O;
- There is no need to decompress for FSC, since data columns compressed by FSC can be operated directly;
- Small size of data leads to high cache hit rate. Thus, direct operations on compressed data leads to less time of retrieving.

![Figure 9(a) Selectivity below 30%](image)

![Figure 9(b) Selectivity of 100%](image)

Figure 9. Searching time of fixed part match with different selectivity

2) Searching Time of Fixed Part Match: In fixed part match, after retrieving on columns compressed by FSC, matching entries need to be decompressed. Thus, the more matched entries there are, the more time needed to decompress. Thus, we use selectivity to denote the percentage of the size of result of searching to the size of data column. For searching on a data column, which has 100 elements, there are 30 elements satisfying the matching condition. The selectivity of this searching is 30%. We conduct two experiments on different selectivity.

Figure 9(a) shows the searching time with selectivity below 30% and figure 9(b) presents the searching time with
selectivity of 100%. These two figures illustrate that, for fixed part match, FSC achieves better query performance than zlib. Fixed part match is much similar to exact match. Although the time for decompression in FSC is more than that in zlib, for FSC, the time saved from I/O and retrieving on compressed data offset the high decompression cost. Thus, FSC achieves better query performance than zlib in fixed part match.

3) Searching Time of Pattern Match: Figure 10(a) shows the searching time of pattern match with selectivity below 30%. And, figure 10(b) presents the searching time with 100%. FSC is slower than zlib for column 6. There are two reasons: firstly, in this column there are many same strings; secondly, zlib perform quick decompression for this kind of columns. Thus, for FSC, time saved from I/O and retrieving does not offset the time cost more than zlib for decompression. However, it is a special column. For most of columns, FSC achieves better performance of pattern match.

![Graphs showing searching time](image)

(a) Selectivity below 30%

(b) Selectivity of 100%

Figure 10. Searching time of pattern match with different selectivity

VI. CONCLUSION AND FUTURE WORK

Compression is one of the most important techniques for column-stores to improve performance significantly. In this paper, a new compression scheme, called FSC, is proposed as an efficient technique to compress fixed-length string typed columns. In FSC, a data column is vertically partitioned into several sub-columns, and each sub-column is compressed by a lightweight compression method, which leads to the best compression ratio. Query methods searching directly on the compressed columns are designed and implemented. Experiment results of FSC and the corresponding searching methods show the better performance of FSC.

As a part of our future work, we plan to integrate more compression scheme in FSC for sub-columns and leads to more significant compression ratio. Variable length string is another basic type in column-oriented database system, and FSC could be used to compress this type of columns in future.

In addition, parallel processing and index on sub-columns could be introduced to improve query performance.

REFERENCE
