Efficient Dictionary-based String Compressing System

Jingmao You
Inria CEDAR and Ecole Polytechnique
Paris, France
Email: jingmao.you@polytechnique.edu

***Abstract*—Nowadays, there are some researches have shown that the usage of a data compression scheme could improve query processing performance in the database system. One such scheme is to use a dictionary in order to replace long values (variable length) with shorter (fixed-length) integer code[1].**

At the beginning of the internship, I did literature survey about column-oriented database system, data compression algorithms and prefix-tree indexes building. Then I designed and implemented the algorithms and data structures of a dictionary-based string compression system for the database system, based on the approach from one paper[1] After that, I add new features to my system apart from the ones existed in some older systems. Those new features include the suffix-match encoding, parallelization lookup operation and file generation. The main features of my data compression system include the data structure that can efficiently support the dictionary, efficient indexes that support the decoding (get the string from the given code) and exact match, prefix-match and suffix-match encoding (get the code from the given string), doing the lookup operation (encoding and decoding) concurrently, generation dictionary file on the disk and loading the file into the memory for the usage of it.

I. Introduction

In the modern data analysis, researchers are dealing with millions of GBs of data, and many are trying to use the technology named data compression to make a file, message, or any chunk of data smaller. Data compression can significantly decrease the amount of storage space a file takes up. There are two kinds of compression, lossless and lossy. Lossy compression loses data, while lossless compression keeps all the data. Lossless compression can compress the data whenever redundancy is present. Therefore, lossless compression takes advantage of redundancy.[1]

This time, I would like to implement a data compression system for a database system in C++. More specifically, for a column-oriented database system. Recent work has shown that the dictionary encoding, as a lightweight compression scheme, could lead to the great improvement of query processing on top of compressed data.[1][2][3] Dictionary encoding replaces long values(variable length) of strings with shorter integer codes(fixed length)[1]. Usually, for the column-based database system, we need to keep a dictionary for each column, using an array to save all the string values and use another to store the corresponding integer codes. Moreover, the values of the dictionary are usually order preserved, in order to decrease the cost of some expensive query operations, such as sorting or searching[1].

Considering these requirements, I did a literature survey related to dictionary compression scheme that preserves order. In the previous work, the researchers implemented serial encoding (prefix matching and exact matching) and decoding processes, using indexes to improve lookup speed[1].

Based their approach, I have first implemented a dictionary and indexes then I have done some improvement and extension, such as the data structure of the index, the suffix matching, the loading of the dictionary form the main memory, and the reading of the dictionary file from the disk.

After the implementation, I have done several experiments in order to test the validity and the performance of the system. All the experiments show that the new extended methods work well under different testing datasets.

The organization of the report will be such:

(1) In section 2, I will generate the result of the literature survey and illustrate the importance of data compression in the database system, and why dictionary-based compression system is an optimal choice for the column-oriented database system.
(2) In section 3, I will explain the Leaves dictionary data structures that I have implemented based on the referenced paper. The dictionary contains multiple pages, each page is called a leaf. A leaf is modeled as a table, however, it is designed in efficient way so it does not require a pointer to point from value to code.
(3) In section 4, I will explain the implementation of the indexes, which is used to locate the leaf which contains the target value or the targe code. The index is modeled as the trie, however, it avoids the massive usage of pointers. Each index contains multiple nodes. I implemented the data structure of the node and the traverse algorithm of the index.
(4) In section 5, I will explain the encoding and decoding methods in the dictionary class based on the indexes that I have implemented. The encoding method includes the exact matching, the prefix matching, which have been mentioned by the referenced paper. Based on that, I have added the suffix matching due to the need of our database system.
(5) In section 6, I will explain the parallelization for the encoding and decoding process, which is designed and implemented by myself. The paper just does every lookup operation under one single thread.

(6) In section 7, I will explain another novel design of my compression system. In my system, I assume the dictionary and indexes are stable, so I generate a binary file and put it in the disk. Then, I could load the dictionary and indexes from the file from the disk when I need to do the encoding and decoding instead of creating the dictionary again.
(7) In section 8, my experiments evaluate all the functionalities of the program, which includes: the building of the dictionary, the encoding and the decoding process, the parallelization and the file generation under the different workload. I also show a detailed analysis of their performance. As a result, all the methods work as intended.
(8) In section 9, I present the conclusion of my work and future work.

II. State of the Art

During recent years, the column-oriented database system has raised the interest of more and more scientists. In column-oriented systems, all the columns are stored individually in the disk, this feature enables queries to read just the attributes they need, rather than having to read entire rows from disk and discard unneeded attributes once they are in the memmory[4] .

One problem with database systems is that large amounts of data are repetitive, which will cause the waste of the storage space. One way to solve it is to introduce a data compression system in the database system. Since we need to be able to compress the data and restore them, the data compression algorithm needs to be lossless.

The common lossless data compression algorithms include run-length encoding, bit-vector encoding and dictionary encoding. Among them, dictionary encoding works well for distributions with a few very frequent values, and can also be applied to strings[4]. Moreover, dictionary can be built for just a part of the whole database system. For example, each column could have its own database system.

One problem of the dictionary encoding is that reading operation can be costly, so the design of the dictionary encoding should assure the efficient reading operation. Some researchers present some dictionary data structures[1][2][3] and others present some indexes[5][6][7] to assure efficiency.

III. Dictionary Structure

In this section, there are two parts that need to be explained. The first part is the composition of a dictionary. What is in the dictionary and why? The second part is the data structure that I have implemented in order to make the dictionary useful for data compression.

A. Dictionary Overview

The dictionary is where I store all the values and corresponding codes. Each value in the dictionary is unique and its code is also unique. A dictionary has many pages, each page will store some values and codes, we call such a page a leaf. A leaf is modeled as a table, where a value can be mapped to a unique code.

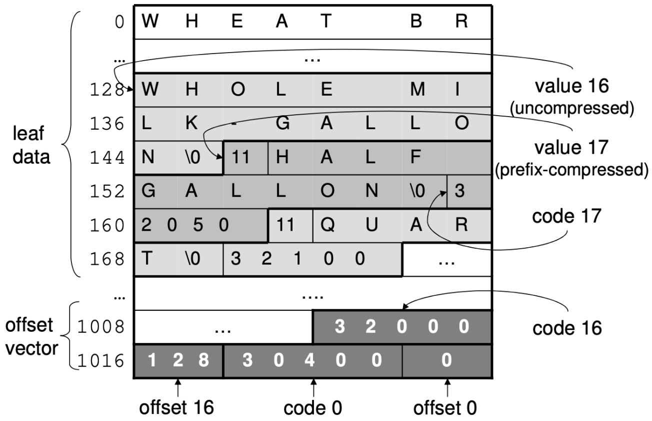

Fig. 1. Leaf data structure.

B. Leaf Data Structure

A dictionary is a list of leaves, so the data structure of a leaf is quite important. When it comes to the encoding and decoding operations, I will locate one leaf through index and then I will do the lookup in the leaf. A leaf is modeled as a table.

Value	Code
aab	0
aae	1
aaf	2
aaz	3

In order to achieve the efficient storage of values and codes as well as an efficient search algorithm, I have implemented a leaf class. The memory layout of a leaf structure is shown Fig. 1[1]. The compression idea here is called incremental-encoding, while each n-th string is not compressed to enable an efficient lookup of strings without decompressing the whole leaf data[1][8]. For example, in the figure, each 16-th string is not compressed, where the strings in between are compressed. The uncompressed value 16 is ”WHOLE MILK - GALLON”, and the value 17 should be ”WHOLE MILK - HALF GALLON”. However, since the first 11 characters of value 17 is as same as the value 16. The common prefix length of value 17 is 11(meaning the first 11 characters of value 17 is the same as value 16), and the uncompressed part of value 17 is ”HALF GALLON”. In order to locate the uncompressed value, an offset vector is stored at the end of the leaf that holds the code of uncompressed value. There are some parameters in the leaf class to build the data structure[1]:

- **leafSize:** Define the memory allocated for a leaf. Use a pointer to point to the allocated memory. This is predefined as 1024 in order to fit the cache size of a CPU.
- **offsetSize:** Define the number of bytes that are used to store the offset of a value in a leaf. All the offsets are stored in the offset vectors at the end of each leaf. The offset is just used to locate the uncompressed value. The

offsetSize is predefined as 3, the maximal leaf size is $2^{24} - 1$ bytes.

- **codeSize:** Define the number of bytes that are used to store the integer codes. This number represents the maximal numbers of values in the dictionary. The number is predefined as 4, meaning there could be $2^{32} - 1$ values in the dictionary.
- **prefixSize:** Define the number of bytes that are used to store the common prefix length. For example, the common prefix length of value 17 is 11. This number is predefined as 1, meaning the maximal prefix length is $2^{8} - 1$.
- **decodeInterval:** Define the interval that is used to store the uncompressed value. The number is predefined as 16, meaning each 16-th value is not compressed.

All the parameters above could be changed when a new leaf object is being initialized. However, there are other parameters that are generated during the building of the dictionary:

- **value_num:** Record the number of values in the leaf object, including the number of uncompressed values and that of compressed values.
- **string_num:** Record the number of uncompressed values.
- **node_first:** Record the first code of value in this leaf.
- **node_last:** Record the last code of value in this leaf.
- **pointerLeaf:** Pointer of the allocated memory of the leaf.
- **next:** Pointer to the next leaf.
- **leaf_data:** The offset to the end of the leaf data(the compressed data), as Fig.1 suggests.

It has been well discussed that most CPUs have a better performance when the data is byte-aligned, and other variants might lead to an increase in the memory consumption of the leaves[1].

C. Leaf Operations

There are three methods that are really important in the leaf class. The first is the insertion method, it will insert several values into a leaf while generating related codes. The second is the encoding method, which is used to find code with the given value. The third is the decoding method, which is used to find value with the given code. The other methods are mainly used to support the methods above.

1) Insertion Method: The final dictionary is actually divided into two parts. The first part is a dictionary built for the exact matching and the prefix matching encoding, and the second part is a dictionary built for suffix matching encoding. In order to build these two parts, I have implemented two different bulk-load methods. The design and the algorithm of insertion methods are not mentioned in the referenced papers, so this part is purely designed and implemented by myself.

(1) **bulkload_initial:** Insert multiple strings into a leaf, the method will generate the codes of the strings automatically. The algorithm is shown in Algorithm 1.

(2) **bulkLoad_initialSuffix:** This is a method to create the leaf for the suffix encoding method. All the values in the second part of the dictionary are the reverse version of the first part. For example, the value ”aab” in the first part should be ”baa” in the second part. However, these

Algorithm 1 Insert values into the leaf

```
Input: pointer that points to the allocated memory of leaf: leafpointer, pointer of input strings: str, the number of input strings: i the code of the first value in this leaf: code_start.
Output: if the insertion method succeeds, then return 0, else return -1.
Put all the input strings into a vector.
Generate codes for all the input strings, from code_start to code_start + i - 1.
off_vector = ceil(i/decodeInterval) is the number of uncompressed values in the leaf, which is also the number of offset vectors
Generate offset vectors array in order to store the offset and the code of the uncompressed values.
//Calculate all the prefix length for every compressed value.
for $j = 0;j < string\_num;j++$ do
    Calculate prefix_length: prefix length of the uncompressed string and the string next to it;
    for all compressed strings in interval do
        Calculate prefix_length1: prefix length of the uncompressed string and other compressed string;
    end for
    if $prefix\_length >= 255$ then
        //The length is limited by the prefixSize
        $prefix\_length = 255;$
    end if
end for
Based on the prefix_length, compressed all the strings, except the first string of each interval. Each compressed string is composed with: prefix_length + uncompressed part.
//Insert the uncompressed values, compressed values, codes and offset vectors into the leaf.
for $j = 0; j < string\_num; j++$ do
    Insert the uncompressed value into the leaf;
    // leaf_off is used to know the offset of the end of leaf data.
    $leaf\_off+ = size(uncompressed\_value);$
    for $t = 1;(t < decodeInterval)$ and $(decodeInterval*j+t < this->value\_num);t++$ do
        //Insert all the compressed values in this interval into the leaf;
        Insert the prefix_length;
        Insert the compressed value;
        Insert the code of the compressed value;
        $leaf\_off+ = prefixSize + codeSize + size(compressed\_value);$
    end for
    Store the $leaf\_off$ as the offset of the next uncompressed string;
end for
Insert the offset vectors from the end of the leaf;
```

two values need to have the same code to make sure the correctness of suffix matching encoding. So the input of these insertion algorithm will also include the codes of the strings.

2) Encoding Method: The encoding method in the leaf class contains two steps. The first step is the binary search through the offset vectors. The second step is the sequential search though compressed values between two uncompressed values in the leaf data part.

(1) **binarySearch** The binary search is usually used to search an integer number, I use it to search a string. However, there might not be an exact string the same as the input one. Instead, if there are two strings: $value1$ and $value2$, and the input string is $value$. Then the condition is $value1 <= value$, and $value < value2$.

(2) **sequentialSearch** The algorithm of the sequential search is in the paper[1]. The idea is using the result of the binary search to locate the uncompressed value. Then start decompressing the compressed values after the uncompressed value, noted as $value_n$, and compare them with the input string $value$. If it could reach a point where $value_n == value$, then return the code of $value_n$, else return -1, meaning $value$ is not found.The algorithm of it is detailed in the paper[1].

3) Decoding Method: The decoding method is similar to the encoding method and also has two steps. The first step is a binary search and the second step is sequential search.

(1) **binarySearch** The input of the binary search is an integer number $code$. It will try to find this $code$ in the offset vectors. If it managed to find a $code_{uncompressed}$ and $code_{uncompressed} == code$, then return the uncomresssed string. Else it will try to find $code_{uncompressed1}$ and $code_{uncompressed2}$, so $code_{uncompressed1} < code$ and $code < code_{uncompressed2}$. Or it will return en empty string to show that there is no value corresponding with the input code.

(2) **sequatialSearch** The sequential search is used to find $code_n$ when it manages to find $code_{uncompressed1}$ and $code_{uncompressed2}$ in the binary search. Then it will find the $code_n$ between those two values, where $code_n == code$. It will return the $value$ corresponding with the $code_n$.

D. Dictionary Building

The dictionary is this system is a list of leaves. It has two parts. The first part of the dictionary is an ordered dictionary built upon input strings, the second part of the dictionary is a reversed order dictionary built upon the reversed input strings. The most important part here is to decide how many strings that could be inserted into one leaf, and then build the leaves one by one.

1) Ordered Dictionary: All the values in the dictionary have to be unique, and they need to be sorted alphabetically. I implemented a alphabeticallySort method to achieve that. This method inserts all the strings into a $map < string, int >$ data structure. First, it inserts the $pair < value, 1 >$ into the map, due to the map data structure in C++, all the duplicated strings are deleted, and they are sorted alphabetically.

Algorithm 2 binarySearch_prefix

Input: pointer that points to the allocated memory of leaf: leafpointer, the starting point: start, the ending point: end, the string value to be encoded: value.

Output: the corresponding code of the value, else return -1.

```
;
if $end >= start$ then
    $mid = start + (end - start)/2$;
    $offset\_1 = (offsetSize + codeSize) * mid\_off$; //the offset vector of the middle uncompressed value
    $offset\_2 = (offsetSize+codeSize)*(mid\_off+1)$;
    $offset1 = getOffset(offset\_1)$; //the offset of the middle uncompressed value
    $offset2 = getOffset(offset\_2)$;
    $value1 = leafpointer[offset1]$;
    $value2 = leafpointer[offset2]$;
    if $value1 < value$ then
        if $value < value2$ then
            return $mid$;
        end if
        if $mid == string\_num - 1$ then
            return $mid$;
        end if
    end if
    return $binarySearch\_prefix(leafpointer, mid + 1, end, value)$;
    if $value1 == value$ then
        return $mid$;
    end if
    if $value < value1$ then
        return $start$;
    end if
else
    return $binarySearch\_prefix(leafpointer, start, mid - 1, value)$;
end if
return $-1$;
```

After all the values are ready to be inserted, it will calculate the number of offset vectors, in other words, the number of intervals. The first value in each interval will stay uncompressed, the rest will be compressed. So it calculates the prefix length and the size of compressed values in each interval.

In every interval, it has uncompressed value, compressed values, the codes, the prefix length and one offset vector. All those values are added and stored in an array $size\_interval[]$. Suppose there are n intervals when the size of those intervals reach the size of one leaf, it will insert the strings in those intervals into a leaf using the bulkload initial method in the Leaf class. Then it will do the same operation to the rest of the intervals until all the values have been inserted into the dictionary.

2) Reversed order dictionary: It reverses all the input strings and the rest of the process is the same as the building

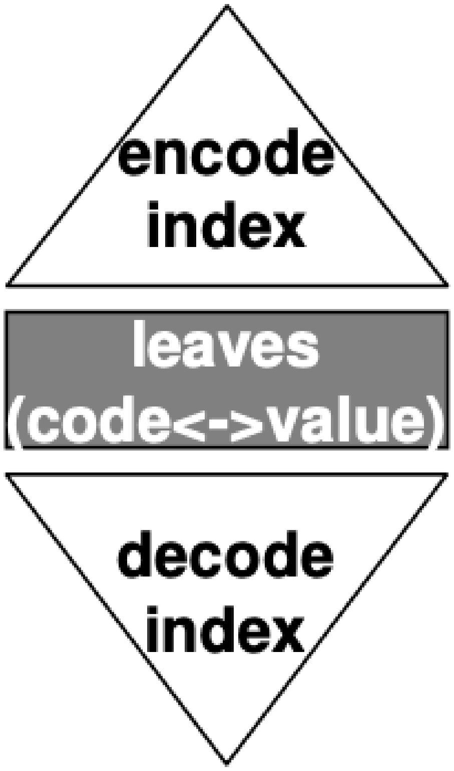

Fig. 2. Shared leaves indexes.

of the ordered dictionary.

IV. Index Structure

This section is divided into several parts. The first part illustrates the general idea of the index, including the purpose of building an index, the indexes that I have implemented in the dictionary data compression system. The second part is the data structure of the index, I will explain the details of the implementation and why this is a good way to implement it. The third part is the operations related to the index, here I will explain how to build an index from the dictionary. As for the usage of indexes for the encoding and decoding process, I will explain them in the next section.

A. Index Overview

The idea of building shared leaves indexes based on the dictionary is mentioned in the paper[1]. The general idae of shared leaves indexes is shown in Fig. 2[1] The encoding and decoding indexes are built on the same dictionary. In some other traditional implementations, there are two dictionaries for two indexes, which could waste the storage space. Since both values and codes in the dictionary are unique, I use one index to locate the value(encoding index) and another index to locate the code(decoding index). This implementation is called the shared leaves indexes[1].

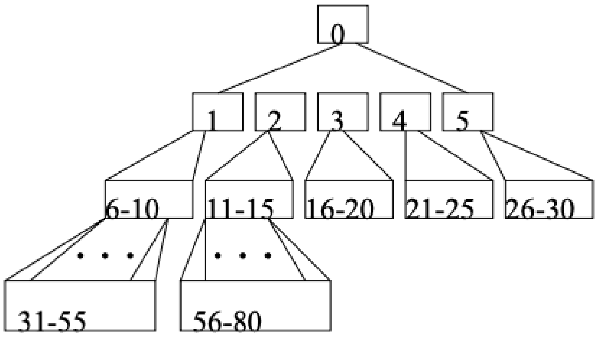

Fig. 3. CSS-Tree.

The index is modeled as a trie, which is also called a digital tree or prefix tree. Usually, the implementaion of a trie will use many pointers from a parent node to all his children nodes. Here, I implement a cache sensitive search tree[5], noted as CSS-Tree. The CSS-Tree is shown in Figure 3.

In the CSS-Tree as shown in Fig. 3[1], each parent node just have five children, and it only just use one pointer to point to their first child. This method is more cache sensitive than the traditional implementation. However, I purposed a new way of implementation. Take the CSS-Tree in Figure 3 as an example, if the size of each node is 1 byte and there are 4 values in 1 node, and the values are sorted from the smallest to the largest. For node 2, if the input $value$ is less than the first value of node 2 and larger than the last value of node 1, then the next search node should be node 11. If the $value$ is larger than the first value of node 2 and less than the second value of node 2, then the next search node should be node 12. If all the nodes are stored in contiguous memory space, then the offset between node 2 and node 11 is 9 bytes, the offset between node 2 and node 12 is 10 bytes. So if I could calculate the offset in advance and store them in the node, I could avoid the usage of extra pointers and the traverse of the CSS-Tree could be more efficient.

B. Index Node Data Structure

One index contains multiple nodes, the values that store in the node is used for comparison. In this part, I will explain the data structure of the node in the encoding index and the node in the decoding index.

1) Encoding Index Node Data Structure: The size of the node is predefined as 32 bytes[1], however, it could be changed easily when a new object is created.

Offset to 1st child	No of keys	Offset vector	Keys(reversed)
0	3	[29,25,22]	[bc,amq,am]

A node is divided into 4 parts:

(1) **Offset to 1st child:** The value stored in this part is used to locate the 1st child of this node.

(2) **No of keys:** The number of keys in this node. The key of the encoding index node is the prefix value. For example, this node has three prefix values: bc, amq and am.
(3) **Offset vector:** The offset vectors are used to locate the keys in this node, since the size of keys in the node is not fixed. For example, the first offset vector here is used to locate the first key in the node(the smallest key), which is ”am”.
(4) **Keys(reversed):** The value of the keys in the node. The value is inserted from the end of the node. Because the size of the node is fixed while the size of key is not, the key is inserted from the end and the insertion stops when this node is full.

2) Decoding Index Node Data Structure: The data structure of the decoding index node is not mentioned in the paper[1], I design and implement them. The data structure is similar to that of encoding index node. However, the size of the code in the dictionary is fixed, so offset vector is not needed.

Offset to 1st child	No of keys	Keys(ordered)
0	3	32 64 96

(1) **Offset to 1st child:** The value is used to locate the first child node.
(2) **No of keys:** The number of keys in this node.
(3) **Keys(ordered):** The keys in this node, each key is an integer number, is the code from the dictionary. Since the predefined size of node is 32 bytes, the space for a key is 24 bytes, each node could store 6 keys.

C. Index Node Operations

Apart from the data structure of the node, there are some methods in the node for the building of the index.

1) Encoding Index Node Operations: The index is modeld as a trie, so the nodes in the trie have different levels. In this implementation, specifically, I define the level of the nodes in the buttom is level 0. The root node has the highest level.

The parameters in the class:

(1) **level:** The level of this node.
(2) **level_num:** The number of nodes in this level.
(3) **indexNodesize:** The size of this node. This figure is used to decide how many keys can be inserted into the node.

Some methods in the class:

(1) **insertLeafnum:** The method is used to insert the offset to 1st child. For the nodes in level 0, this value is the number of leaves in the dictionary.
(2) **insertKeys:** Insert a key into the node. The key will be inserted from the end, the number of keys will be added 1, and the offset vector will also be added. If the end of the offset vectors does not reach the beginning of the keys, then the insertion succeeds.
(3) **setLevel:** Set the level parameter. Starting from 0.

2) Decoding Index Node Operations: The methods in decodingindexNode class is similar to the ones in encodingindexNode. Except the insertKey method does not need to add the offset vector and does not insert the key from the end of the node.

D. Index Building

The index is built from the bottom. The nodes in level 0 will return the number of a leaf from the dictionary. The building process ends when there is only one node in the level, this node is the root node, and the level of it is the highest

1) Encoding Index Building: The first step is generating the keys from the dictionary. All the values in the dictionary have been sorted, if two leaves are supposed to be distinguished by a key, then the key needs to be generated from the largest value of the first value and the smallest value from the second leaf. The key generation algorithm is shown in Algorithm 2.

Algorithm 3 Generate Encoding Index Keys

Input: string str1: the largest value from the former leaf, string str2: the smallest value from the later leaf.
Output: string key: the generated key.

```
$key\_length = min(str1.length, str2.length)$;
for $i = 0$;$i < key\_length$;$i++$ do
    if $str1[i] == str2[i]$ then
        $key[i] = str1[i]$;
    end if
    if $str1[i] < str2[i]$ then
        $key[i] = str2[i]$;
        Break;
    else
        //Do nothing
    end if
end for
return $key$;
```

After having all the keys, they will be inserted into the nodes one by one. If the number of nodes in level 0 is larger than 1, then a new level needs to be created above level 0. The building of nodes in level 0 is shown in Algorithm 3.

Algorithm 4 Encoding Index Building: Level 0

```
$node\_tmp; tree1;$ // index node,tree used to store the nodes in one level
$offset = 0; level = 0; level\_num = 0;$
for $i = 0, j = 1; j < leaves\_num; i++, j++$ do
    $leaf1 = leaf[i];$
    $leaf2 = leaf[j];$
    $str\_small = leaf1.largestvalue;$
    $str\_large = leaf2.smallestvalue;$
    $key = generateKey(str\_small, str\_large);$
    $node\_tmp.insertLeafnum(offset);$
    $test = node\_tmp.insertKeys(key);$
    $node\_tmp.setLevel(level);$
    if $test == -1$ then
        //This node is full, needs to add one node;
        $tree.Insert(node\_tmp);$
        $offset+ = node\_tmp.getKeyum;$ //add the number of keys in the node to the offset
        $newnode\_tmp;$
        $node\_tmp.insertLeafnum(offset);$
        if $j == leaves\_num - 1$ then
            $level\_num++;$
            $node\_tmp.setLevelnum(level\_num);$
            $tree1.Insert(node\_tmp);$
        end if
    else
        if $j == leaves\_num - 1$ then
            $level\_num++;$
            $node\_tmp.setLevelnum(level\_num);$
            $tree1.Insert(node\_tmp);$
        end if
    end if
end for
```

Then I need to build the levels above level 0, until it reaches a point where only one node in the current level. As it is shown in Algorithm 4.

Algorithm 5 Encoding Index Building: Level 1 to n

```
while $level\_num > 1$ do
    $node\_tmp2; level++; level\_num = 0;$
    $offset1 = 0; offset = 0;$
    for $i = num\_nodes - level\_num1 + 1; i < num\_nodes; i++$ do
        // num_nodes is the number of nodes in tree1,level_num1is the number of nodes in level n-1, suppose current levelis level n
        $node\_tmp1 = tree1.getNode(i)$
        $key = node\_tmp1.getKey$
        $node\_tmp2.insertLeafnum(offset);$
        $node\_tmp2.setLevel(level);$
        $test = node\_tmp2.insertKeys(node_i.smallestKey)$
        if $test == -1$ then
            $level\_num++;$
            $node_tmp2.setLevelnum(level_num);$
            $tree1.Insert(node\_tmp2);$
            $offset+ = node\_tmp2.getKeynum;$
            $newnode\_tmp2;$
        else
            if $i == num\_nodes - 1$ then
                $level\_num++;$
                $node\_tmp2.setLevel(level);$
                $node\_tmp2.setLevelnum(level\_num);$
                $tree1.Insert(node\_tmp2);$
            end if
        else
            if $i == num\_nodes - 1$ then
                //if i reaches the last key in this level,we need to push it into a node anyway/
                $level\_num++;$
                if $level\_num == 1$ then
                    $offset = 0;$
                    $node\_tmp2.insertLeafnum(offset);$
                end if
                $node\_tmp2.setLevelnum(level\_num);$
                $tree1.Insert(node\_tmp2);$
            end if
        end if
    end for
end while
```

Now all the nodes have been created. Note that all the nodes are inserted into the list tree1 from the bottom, however, when the index is used, it requires to traverse the tree from the root node. The last step of the building process is to reverse the order of nodes and insert them into the final index.

Algorithm 6 Encoding Index Building: Final Step

```
$indexnode\_tmp = newencodingindexNode$;
$levelnum = tree1- > getNode(num\_nodes - 1)- > getLevelnum()$;
//The number of nodes in level 0.
$nodes\_left = num\_nodes$;
//The nodes ready to be inserted.
$nodes\_inserted = 0$;
while $nodes\_left > 0$ do
    $levelnum = tree1.getNode(nodes\_left - 1).getLevelnum()$; // The number of nodes in the current level.
    $nodes\_left- = levelnum$; //The left number of nodes.
    for $i = nodes\_left$;$i < nodes\_left + levelnum$;$i++$ do
        //Insert the nodes in the level into the final index.
        $node\_tmp = tree1.getNode(i)$;
        $indexnode\_tmp.insertNode(node\_tmp)$;
        $indexnode\_tmp.insertLevelnum(levelnum)$;
        $index.push\_back(indexnode\_tmp)$;
    end for
end while
```

All the nodes have been inserted into the index in the right order. The usage of this index in the encoding process will be explained in the next section.

2) Decoding Index Building: The building of the decoding index is nearly the same. However, when I need to generate the key from two leaves, I just need to take the smallest code in the second leaf as the key.

V. Encoding and Decoding Methods

In this section, I will explain the key functionalities of the system: encoding and decoding methods. Encoding method will return the codes based on the input string values. Decoding method will return the string values based on the input codes.

A. Encoding Method

The paper[1] explains the idea of exact matching and prefix matching, I added suffix matching based on that. All the algorithms are designed and implemented by myself.

1) Exact Matching: The meaning of exact matching is: for a given string, the method will try to find the same string in the dictionary, it will return the related code if it succeeds, or it will return -1. The algorithm is based on the encoding index. The index will generate the possible leaf number and the encoding method in the leaf class, which is mentioned in Section 2 will try to return the corresponding code.

2) Prefix Matching: The meaning of the prefix matching is: for a given string, the method will try to find all the string values that use the input string as the prefixes. Then it will return all the corresponding codes.

3) Suffix Matching: The meaning of the suffix matching is: for a given string, the method will try to find all the string values that use the input string as the suffix. Then it will return all the corresponding codes.

First I need to choose the right index. There are two indexes, one is for prefix matching, the other is for suffix matching. Then the index needs to be traversed in order to find the leaf number. The traverse starts from the root node, when it reaches $value < key$, it will jump to the lower level, until it reaches the level 0 of the index. The algorithm of generating the leaf number is shown in Algorithm 7. Having found the leaf number of the dictionary, the input value will be searched in that specific leaf. The algorithm of encoding based on a leaf number from the index is shown in Algorithm 8.

Algorithm 7 Encoding Method: Choose the right index

```
Input: $value$: the string value ready to be translated to code, $prefix\_flag$: used to indicate the encoding methods, 0 represents the exact matching, 1 represents the prefix matching and 2 represents the suffix matching.
if $prefix\_flag == 2$ then
    $index = index\_suffix$; //Choose the suffix index.
    $reverse(value)$;
else
    $index = index\_prefix$; //Choose the prefix index.
end if
```

Algorithm 8 Encoding Method: Generate the leaf number

Input: $value$: the string value ready to be translated to code, $prefix\_flag$: used to indicate the encoding methods, 0 represents the exact matching, 1 represents the prefix matching and 2 represents the suffix matching.

Output: $result$: the method will return the codes based on the input value and choice of encoding method.

```
$level\_tmp = index[0].getLevel$; //get the level from the root node
while $level\_tmp >= 0$ do
  $offset1 = offset$; //offset from the index node.
  $levelnum\_count1 = levelnum\_count$; //number of nodes in this index level.
  $levelnum\_count+ = nodes\_num$; //add the number of nodes in the current level.
  for $i = level\_start, m = 0$;$i < nodes\_num$;$i{+}{+}, m{+}{+}$ do
    //compare the input value with keys from nodes($nodes\_num$) in the same level.
    $offset\_tmp1+ = m$
    for $j = 1$;$j <= key\_num$;$j + +$ do
      //Loop the keys in one node.
      $leaf\_num1 = index[offset\_tmp1].getOffset + j - 1$; //
      if $value < key$ then
        renew the $offset\_tmp$; $break\_point$; $level\_start$
        if $level\_tmp == 0$ then
          $leaf\_num = leaf\_num1$; //get the leaf number.
        end if
      end if
      if $value >= key$ then
        renew the $offset\_tmp$; $break\_point$; $level\_start$
        if $level\_tmp == 0$ then
          $leaf\_num = leaf\_num1$; //get the leaf number.
        end if
        else
          if $value <= key$ then
          renew the $offset\_tmp$; $break\_point$; $level\_start$
          if $level\_tmp == 0$ then
            $leaf\_num = leaf\_num1$; //get the leaf number.
          end if
          $break$;
        end if
      end if
    end for
    if $break\_point == 1$ then
      $break\_point = 0$;
      $break$;
    end if
    $level\_tmp - -$;
  end for
end while
```

Algorithm 9 Encoding Method: Encoding based on the leaf number

Input: $value$: the string value ready to be translated to code, $prefix\_flag$: used to indicate the encoding methods, 0 represents the exact matching, 1 represents the prefix matching and 2 represents the suffix matching.

Output: $result$: the method will return the codes based on the input value and choice of encoding method.

```
if $prefix\_flag == 0$ then
  //The exact matching.
  $result = getLeaf(leaf\_num).lookup(value)$; //encoding method in the leaf, explained in section 2.
else
  //The prefix matching and the suffix matching.
  $result = getLeaf(leaf\_num).lookup\_prefix(value)$; //prefix or suffix matching in the leaf.
  //For the prefix and suffix matching, the values in the next leaf might also need to be tested.
  $length = size(value)$;
  $leaf\_num + +$;
  if $leaf < Leaves\_num$ then
    $next\_value = getLeaf(leaf\_num).smallestvalue$;
    while $(next\_value.find(value)) and (leaf\_num < Leaves\_num)$ do
      $result1 = getLeaf(leaf\_num).lookup(value)$; //Do the prefix matching in this leaf.
      $leaf\_num + +$; //Try to do the matching in next leaf.
      $next\_value = getLeaf(leaf\_num).smallestvalue$;
    end while
  end if
end if
$result = result + result1$;
return $result$;
```

B. Decoding Methods

The input of the decoding process is the code. The method will use the decoding index to find the leaf that might contain this code, and then use the decoding method in the leaf class in order to find the corresponding string value.

VI. Parallelization

The parallelization mentioned in the paper is used for the building of the dictionary. However, in this system, it is a fixed dictionary which is just used for encoding and decoding. The focus of parallelization is doing the encoding and decoding process through multiple threads. In this part, I use the thread library to create multiple threads, in order to run the encoding and decoding process concurrently.

A. Encoding Parallelization

Each thread will run a worker method, it is divided into two parts:

(1) **Lookup in the Dictionary:** The dictionary will be read from the memory, then the worker will try to find the codes based on the input string values.

(2) **Write the result:** Having generated the codes from the dictionary, all the results need to be inserted into a shared space. It requires to use the lock and unlock to avoid conflicts.

Algorithm 10 concurrent_worker_encode

```
Input: $strings$:the strings to be encoded. $mode$:choose the encoding methods,exact,prefix or suffix matching.
Output: $void$
$size = strings.size$;
for all $string$ in $strings$ do
    $tmp\_code = lookupIndex(string, mode)$;//lookup a string
    $tmp\_codes.insert(tmp\_code)$;//$tmp\_codes$stores the results of strings
end for
$exclusive.lock$//before writing the result of the worker into the final result space
$result.insert(tmp\_codes)$
$result.unlock$
```

Then there is a method that could create multiple threads, each thread will start a worker. The method could generate the number of threads based on the hardware.

Algorithm 11 concurrent_task_encode

```
Input: $strings$:the strings to be encoded. $mode$:choose the encoding methods,exact,prefix or suffix matching.
Output: $void$
$concurrent\_count = thread :: hardware\_concurrency$;
$size = strings.size$;
$range = size/concurrent_{c}ount$;//number of strings in each thread.
for $t = 0$;$t < concurrent_{c}ount - 1$;$t + +$ do
    $tmp = range * t$;
    $strings\_tmp = strings(begin + tmp, begin + tmp + range)$;//$strings\_tmp$stores the strings for 1 thread.
    $threads.push\_back(this- > concurrent\_worker\_encodeThread(strings\_tmp, mode))$;
end for
$strings\_tmp = strings(begin + (concurrent\_count - 1) * range, end)$;
for all $t$ in $threads$ do
    $t.join$;
end for
```

B. Decoding Parallelization

The decoding parallelization is similar to encoding parallelization. There are multiple threads and each thread starts a worker. The worker will decode part of the input codes.

VII. File Generation and Reading

The dictionary and the indexes that are created are stored in the main memory, which means they are supposed to be created every time. In this system, I design a method to store the dictionary in a binary file and store the file on the disk. I have also designed a method to read the file from the disk and restore the dictionary and the indexes.

A. File Generation

The file generation is divided into two parts:

(1) **Dictionary Part:** The dictionary needs to be stored in the file. Some parameters, such as the number of leaves in the dictionary, need to be written into the file to restore the dictionary later. Then all the leaves in the dictionary need to be inserted into the binary file one by one.

(2) **Index Part:** There are three indexes that need to be written into the file. The number of nodes in each index need to be written, then the data in each node needs to be written.

Algorithm 12 generateFile

```
Input: $filename$:the name of the generated file.
Output: $void.$
$ofstreamoutfile(filename)$;
$intparameters[]$;
$outfile.write(parameters)$;
$intleafp[]$; // used to store the parameters of a leaf.
$parameters[0] = dic1$; //The number of leaves of the dictionary for prefix matching.
$parameters[1] = dic2$; //The number of leaves of the dictionary for suffix matching.
for $t = 0$;$t < leaves\_num$;$t + +$ do
    $leaf\_tmp = getLeaf(t)$;
    $leafp[0] = leaf\_tmp.getleafSize$;
    $\ldots$ //Insert other parameters, such as the leafSize, the codeSize, the offsetSize.
    $outfile.write(leafp)$;
    $outfile.write(leaf\_tmp.pointer)$; //Write the data in the leaf into the file.
end for
$intsize[]$; //Store the size of each index
$size[0] = prefix\_index.size$
$size[1] = suffix\_index.size$
$size[2] = decode\_index.size$
$outfile.write(size)$;
for $t = 0$;$t < size[0]$;$t + +$ do
    $outfile.write(prefix\_index.node[t])$;
end for
for $m = 0$;$m < size[1]$;$m + +$ do
    $outfile.write(suffix\_index.node[m])$;
end for
for $n = 0$;$n < size[2]$;$n + +$ do
    $outfile.write(decode\_index.node[n])$;
end for
$outfile.close$; //
```

B. File Reading

The file reading is divided into two parts:

(1) **Dictionary Part:** From the parameters in the file, the number of leaves in the dictionary could be got. Then read the data in each leaf and restore the dictionary.

(2) **Index Part:** From the file read the number of nodes for each index, read the data of the nodes and restore the indexes.

The algorithm is like the reverse version of the file generation algorithm.

```
Algorithm 13 readFile
Input: $filename$:the name of the generated file.
Output: $void.$
$ifstreaminfile(filename);$
$intparameters[];$
$infile.read(parameters);$
for $t = 0;t < parameters[1];t++$ do
    $intleafp[];$
    $infile.read(leafp);$
    $newleaf\_tmp;$
    $infile.reaf(leaf\_tmp);$ //read the data of a leaf from the file
    $Insert(leaf\_tmp);$ //Insert a leaf into the dictionary.
end for
$intsize[];$ //Read the size of index
$infile.read(size);$
for $m = 0;m < size[0];m++$ do
    $infile.read(prefix\_index.node[m]);$ //read the prefix_index from the file
end for
...// read suffix_index and decode_index in the same way.
$infile.close;$
```

VIII. Experiments

This section shows the results of my validity experiments and performance experiments with the prototype of my string-dictionary. The dictionary has been implemented with the data structure discussed in section 2, and the indexes have been implemented with the data structure discussed in section 3. The encoding and decoding methods have been implemented as the algorithms discussed in section 4. All the experiments are divided into two parts. The first part is called Validity Experiments. I tested the functionalities of the system, including encoding and decoding methods, file generation and file reading. The second part is called Performance Experiments. I tested the time consumption of concurrent lookup and that of serial lookup. I changed the node size in indexes, try to test the performance of indexes with different node sizes. Then I build a simple dictionary using the map structure in C++, comparing with this efficient system.

I implemented the data structures and algorithms in C++ and tested them in 64-bit Macbook Pro(macOS Catalina) with 2,3 GHz 8-Core Intel Core i9 and 32 GB 2667 MHz DDR4 main memory.

In order to generate a meaningful workload, I built my own workload generator instead of using TPC-H data generator dbgen[1]. The data generated by dbgen might follow a certain pattern(the customer name is composed of the prefix'Customer') or the domain size of each attribute is too low(the name of the country)[1]. The properties of the workloads that I have generated will be shown for each experiment individually.

A. Validity Experiments

The validity experiments are divided into two parts: the first part is building a dictionary and test the decoding the encoding methods. the second part is to store the dictionary in the file and read the dictionary from the file, then do the encoding and decoding methods to test whether the file has been generated and read successfully.

Parameter	Value
leafSize	1024 Bytes
offsetSize	3 Bytes
codeSize	4 Bytes
prefixSize	1 Bytes
decodeInterval	16
string-length	(1)10(2)15(3)20
string-number	(1)6400(2)9600(3)16000
distribution	unsorted
duplication	32 times for each string

The testing result of encoding method:

Mode	Input Value	Result Code
exact matching	abxililnqdxjvtliaqkn	1
prefix matching	ab	1
prefix matching	a	0,1,. . . ,29
suffix matching	aqkn	1
suffix matching	kn	115 1

The testing result of decoding method:

Input Code	Result Value
1	abxililnqdxjvtliaqkn
115	dfhvquuirbnzrkn
29	axyigtcrgdbdceuylcqj
30	baqqzlmblbuzwpsdmigh

The result of decoding method could be used to prove that the result in encoding method is right. It is clear that the code of value abxililnqdxjvtliaqkn is 1, so the exact matching is correct. For the prefix matching, the value of code 29 has a value with the prefix ”a”. For the suffix matching, the value of code 115 has a value with the suffix ”kn”. One more thing is that, the values in the dictionary should be ordered, so the value of code 30 should have a prefix larger than ”a”, and here it is clear that the prefix is ”b”.

As for the second part, I write all the indexes and dictionary in one binary file. I start a new dictionary and read the binary file from the disk in order to do restore the dictionary and indexes. Then I do the same test as in part one and all the results are the same. The functionalities of the system have been proved and it runs as expected.

B. Performance Experiments

The performance experiments are divided into three parts: The first part is testing the effectiveness of parallelization, I run the serial version and concurrent version of decoding

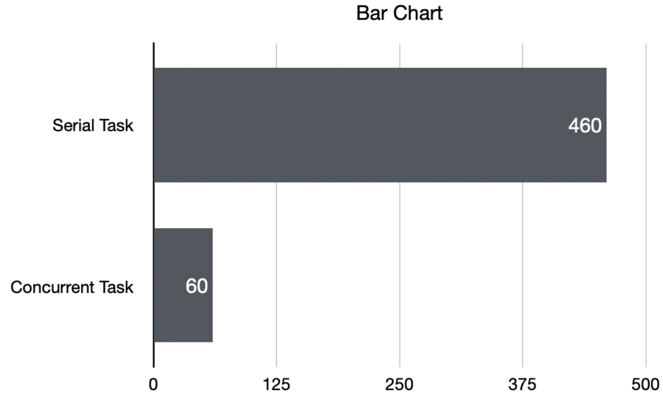

Fig. 4. Encoding Performance: Serial vs Concurrent

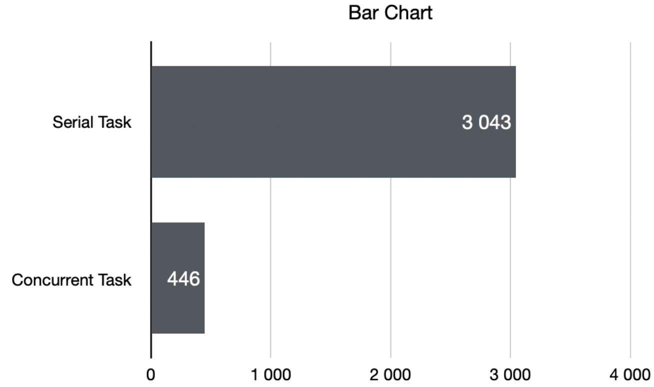

Fig. 5. Decoding Performance: Serial vs Concurrent

and encoding methods, and compare the time consumption of them. The second part is testing the size of nodes in the index. Try to see the time consumption under different node sizes. The third part is to try to compare my implementation with a naive dictionary compression system. I will try to do the same encoding and decoding processes and compare the time consumption.

1) Parallelization Experiment: I try to test whether the parallelization can actually reduce the time consumption.

Parameter	Value
leafSize	1024 Bytes
offsetSize	3 Bytes
codeSize	4 Bytes
prefixSize	1 Bytes
decodeInterval	16
string-length	(1)10(2)15(3)20
string-number	(1)6400(2)9600(3)16000
distribution	unsorted
duplication	32 times for each string

The encoding test:

mode	dataset size
prefix matching	32000

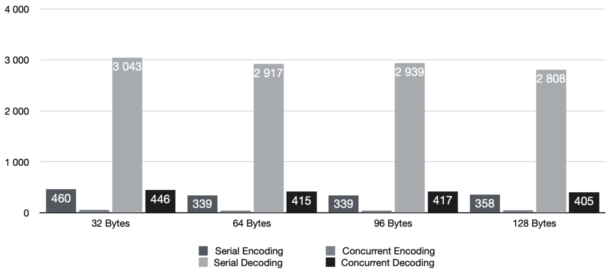

Fig. 6. Performance Under Different Index Node Sizes

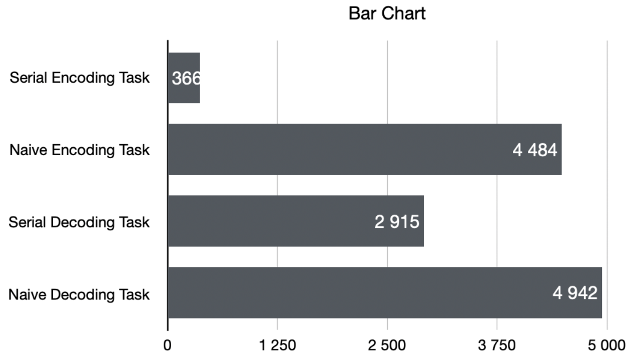

Fig. 7. Performance Compare With Naive Dictionary

The decoding test:

dataset	size
[0–999] repeat 100 times	100000

The testing results are shown in Fig. 4 and Fig. 5. The unit measurement is ”ms”. However, the program detects that there are 16 threads can be created, the actual time consumption is 1/8 instead of 1/16. This is because the hyper-threading can not act as two threads here, so the actual number of threads is 8, due to the fact that the machine has 8 cores.

2) Index Node Size Change: In this part, I change the size of the node to see the change in performance. The larger the index size is, the more keys could be stored in one node. The result is shown in Fig. 6 and the unit measurement is ”ms”. The result shows that the time consumption decreases when I change node size from 32 to 64 bytes. But it stays still when I change it to 96 or 128 bytes. In conclusion, if there are too many keys in one node, then it is like a loop all the values or codes in the dictionary. Else if there is only one key in one node, then there will be too many levels in the trie. Both situations will cause more time consumption and decrease performance.

3) Comparison with Naive Dictionary: In order to make sure out implementation has good performance, I have also implemented a naive version dictionary which can also do the encoding and decoding operations. The naive dictionary uses a map to store all the values and codes. Once I need to do the encoding or decoding operation, I need to loop the whole dictionary. The encoding mode here is prefix matching. The result is shown in Fig. 7. It is really clear that the naive dictionary compression is much slower than my implementation.

IX. Conclusions and Future Work

In this system, I have designed and implemented an efficient dictionary-based data compression system which is used to column-oriented database system. This implementation gathered work from the previous job of other researchers and add new features that will make this system more practical for the column-oriented database system. Since the implementation is written in C++, it could be easily optimized and adapted to other platforms. It is also expected to be integrated with column-oriented database systems in the future.

References

[1] C. Binnig, S. Hildenbrand, and F. Färber, "Dictionary-based order-preserving string compression for main memory column stores," in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, 2009, pp. 283–296.

[2] D. Abadi, S. Madden, and M. Ferreira, "Integrating compression and execution in column-oriented database systems," in *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, 2006, pp. 671–682.

[3] M. Zukowski, S. Heman, N. Nes, and P. Boncz, "Super-scalar ram-cpu cache compression," in *22nd International Conference on Data Engineering (ICDE'06)*. IEEE, 2006, pp. 59–59.

[4] D. Abadi, P. Boncz, S. H. Amiato, S. Idreos, and S. Madden, *The design and implementation of modern column-oriented database systems*. Now Hanover, Mass., 2013.

[5] J. Rao and K. A. Ross, "Cache conscious indexing for decision-support in main memory," 1998.

[6] ——, "Making b+-trees cache conscious in main memory," in *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, 2000, pp. 475–486.

[7] J. Goldstein, R. Ramakrishnan, and U. Shaft, "Compressing relations and indexes," in *Proceedings 14th International Conference on Data Engineering*. IEEE, 1998, pp. 370–379.

[8] I. H. Witten, I. H. Witten, A. Moffat, T. C. Bell, T. C. Bell, and T. C. Bell, *Managing gigabytes: compressing and indexing documents and images*. Morgan Kaufmann, 1999.