The Key as Dictionary Compression Method of Inverted Index Table under the Hbase Database

Pengsen Cheng
Chengdu University of Information Technology, Chengdu, P.R.China
Email: cps11@163.com

Junxiu An
Chengdu University of Information Technology, Chengdu, P.R.China
Email: anjunxiu@cuit.edu.cn

***Abstract*—Starting with Hbase's own characteristics, this paper designs an inverted index table which includes key word, document ID and position list, and the table can saves a lot of storage space. After then, on the basis of the table, the paper provides key as dictionary compression with high compression ratio and high decompression rate for the data block. At last, this paper tests the effectiveness of the compression method by comparing it with Lzo and Gzip which supported by Hbase.**

***Index Terms*—Hbase, inverted index table, key as dictionary compression**

I. INTRODUCTION

The speed of reading data from disc into memory can be improved by compressing data. The high efficient decoding algorithms run fast on modern hardware. The total time of reading compressed data from disc into memory and decompressing the data is less than reading the same block of uncompressed data. For example, it can reduce the I/O time consuming by loading a compressed position list, even need to add some additional decompression time. So, in the most cases, the search engine which introduces compressed inverted index is more efficiency than which not introduces it. If the target of compression is to save disc space, the compression algorithms usually loss part of compress efficiency. But if the algorithms aim at improve disc transfer efficiency, they're uncompress efficiency must be higher [1].

Traditional compression method based on different compress principle can be divided into the following three classes:

a. Compression method based on statistics, the core principle is the Shannon information theory [2, 3] . In practice, the symbols in message source encode to different length according to its frequency. The symbols with higher frequency encode to short codes, whereas the symbols with lower frequency encode to long codes. Thanks to large probabilities appear more often and small probabilities appear few, after encoding a message can be lossless represented with few symbols.

b. Compression method based on integer transform, this method of compression that applied to a series of integers, is the most used method in inverted index compression [4-9] . In current mainstream computers usually use 32 bits to represent an integer, but for a large part of the number don't need so many bits to be represented. Compression method based on integer transform takes advantage of integer factorization or combinations to represent one or a set of integers with appropriate bits. Those methods represent smaller integers with fewer bits and larger integers with more bits. It achieves less storage space by sacrificing the total number of integer represented with fixed bits.

c. Compression method based on dictionary, this method introduced a dictionary indexing mechanisms into it [10-12] . In other systems, it often needs extra space to store the indexes. However, if it requires additional space in the compression method to store dictionary information, it reduces the compression ratio and even may bring about volume expansion. Therefore, in the dictionary-based compression method, the compressed data make up of index information. This means that data appeared first as a dictionary, later if the same data, record the length of the data and their distance with the dictionary data; if this data don't appear in the previous, the data is recognized as dictionary data.

These compression methods have improved compress ratio or decompress efficiency from different ways, but a common problem in search engine is that these methods focus either on compress ratio or decompress efficiency. The query efficiency is a great important of search engine. In querying inverted index, reading file and decompressing data is a unified process and the process should be consider together rather than focus on one side. In addition, different from traditional row of relational database, Hbase is a sparse, column-oriented and distributed database which physical file not only contains user required information but also contains a lot of redundant information. Byte is base data type in Hbase, therefore there are 256 symbols. If it uses statistics-based compression method, the code tree will be too big. Thanks to block is operation units in compression, there are lot of bit operation and code tree querying operation, and compression ratio is poor and decompress efficiency is normal. The purpose of compression methods based on

doi:10.4304/jsw.8.5.1086-1093

integer transform is to represent a smaller integer with fewer bits, and data in Hbase is the shortest of 8-bit bytes. In compression which operation unit is byte, raising the compression ratio is difficult and different compression methods with different operational size own different decompress efficiency. Inverted index data file in Hbase is a kind of redundant information and integer sequences arranged in alternating files. Large amount of redundant information in the data is suitable for dictionary compression. It needs to compress the list position of invert index before the list be inserted in Hbase, for the query efficiency of invert index can be increased by querying as decompressing the list.

Hbase database currently only supports Gzip compression and Lzo compression [13, 14] . The Gzip compression provides a higher compression ratio, but efficiency of decompression is low [15] ; The Lzo has a high decompress efficiency, but compression ratio is low [16] . Aiming at storage structure of Hbase and the characteristics of data block in the files, this paper comes up with key as dictionary compression method for data block. The way try to raise the compression ratio under the premise of higher decompress efficiency, achieve the unity of file reading and data decompression, and increase the query efficiency of inverted index.

II. The File Storage System In Hbase [13, 14]

Hbase is a column-oriented of distributed database and constructs on Hadoop that is a kind of distributed file system. The file system of Hbase continues to use Hadoop's HDFS (Haddop file system). There would be more than one column family in one table under Hbase, and one column family corresponds to a folded on the physical storage structure. One folder contains several Hfiles. Hfile's structure shown in the following figure:

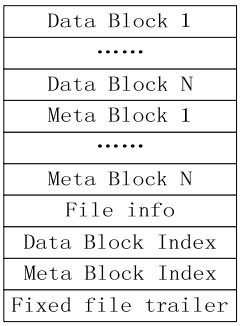

Fig.1.The structure of Hfile

Hfile is variable length file. Every record saves a key/value in the data block, and other parts of Hfile save location information and index information. Hbase can automatic sorts the data in memory to lexicographic order by row key. When writing file, Hbase writes all data blocks in the memory to the files, and in the end of the file adds index information of data blocks. When reading file, it reads interrelated data blocks through accessing index information rather than reads all data at once.

Through analysis on source code, internal compression mechanism in Hbase, it only can compresses on data block and meta block, and compression process occurs in writing data from memory to disk. Each time it only compresses one block. That means every block of compression is needed to re-initial of compression algorithm. And it performs the decompression operation when reading data from disc to memory.

As shown in figure 2, it is a record structure in data block. It begins with two fixed length numbers that represent the length of key and value respectively. The key begins with one fixed length number that represents the length of row key. It followed by the row key, and then is a fixed length number that represents the column family's length, then the column family, then the qualifier, and at last are two fixed length numbers that represent time stamp and key type. The principal original data types are integer and string in key data, but in order to simplify data access procedures, Hbase will automatic change data type to byte. Therefore, the key data is a byte array from software manufacturing perspective. Compared with the key, there are thousands types of the value in practice, but in the same reason, the value also simplify to a byte array.

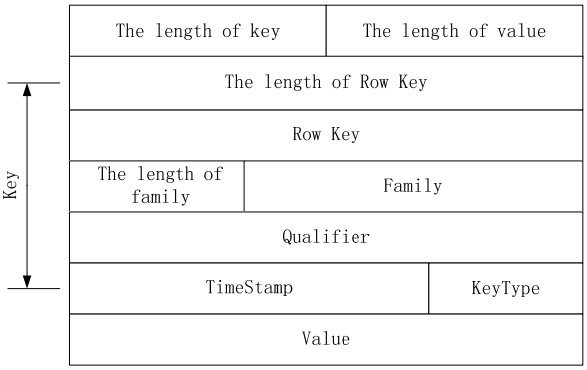

Fig.2.The structure of a record in data block

III. The Design Of Inverted Index Table

Inverted index that adopt word-level inverted table, is structured as follows:

$$< T, (doc_1, pos_1, pos_2, ..., pos_{f_1}),$$
$$(doc_2, pos_1, pos_2, ..., pos_{f_2}), ...,$$
$$(doc_n, pos_1, pos_2, ..., pos_{f_n}) >$$

The T represents the index, namely a word; . $doc_i (i = 1, 2, ..., n)$. represent the ID of documents contained the T ; $pos_i (i = 1, 2, ..., f)$ represent the positions that T locates in $doc_i (i = 1, 2, ..., n)$.

Currently, it is bad, inefficient and there are many system bugs on Hbase's multiple column families. Official recommends that the column family should be controlled at up to 2 or 3 [13] . Because every column family stored in a specific folder and different column families in a same row data are stored in different file, it creates a lot of redundant information. Just the data like row key, time stamp is duplicating saved many times as the number of column family increases that led to a large

number of I/O operation and inefficient when reading data. Reading inverted index file should be in real-time. That means reading file should minimize unnecessary I/O operations. And reading inverted index has the characteristics of fetching the entire row. For a specific keyword, all of the information is needed in the follow-up operation, include document ID, position record list and frequency of the word which hidden in position record. For those reasons, it is not suitable for multiple column family stores. To make full use of storage format of Hfile, this paper used document ID as qualifier in table. In data block, qualifier filed is a space designed to store the column name. Storing document ID in it can save storage space and also in line with people's habits of logical thinking and Hbase provide the interface to obtain the value in qualifier filed. In this paper, the length of the document ID is 4 bytes or 32 bits.

For those reasons, after changed the word-level inverted table to index table under Hbase, the logical view as shown in table I:

TABLE I
LOGICAL VIEW

Row key	Time stamp	position			
		Doc*i*	Doc*j*	…	Doc*k*
W	t*n*	p*1*…p*f*			
	t*n-1*		p*1*…p*f*		
	…				
	t*1*				p*1*…p*f*

In table I, row key refers to a specific word and corresponds to the index entry T in word-level inverted table; time stamp is a unique identification generated by Hbase when insert a data into it. Position is name of column family. This column family has indeterminate number of qualifier. Each qualifier is document ID that T appeared in it, like Doci, Docj, Dock in table, and it corresponds to doc_i in word-level inverted table. One column in a specific row has one value at most under the column family of position. This value is the compressed position record list about T in doc_i.

Hbase is sparse storage database. Section of empty value is directly ignored. At the time of the actual storage, because one column of a specific row is maximum one value, the storage number equal to the number of not empty value.

TABLE II
PHYSICAL VIEW

Row key	Time stamp	Column family	*Column*	*Value*
W	t*n*	position	Doc*i*	p*1*…p*f*
W	t*n - 1*	position	Doc*j*	p*1*…p*f*
……	……	……	……	……
W	t*1*	position	Doc*k*	p*1*…p*f*

IV. KEY AS DICTIONARY COMPRESSION

Taking the characteristics of Hbase and Hfile into consideration, this paper designed key as dictionary compression method (KADC) to compress the all key; the paper used variable byte codes (VBC) to compress value and the integers which stand for length. In order to synchronize querying and decompressing of position list, it needs to compress the list before the position list is inserted into Hbase. Then, the compressed list is the value in Hbase, and it is unnecessary to compress the value again.

A. Compression Algorithm

The length of array and array was expressed by $(n, key)_k$, and the k meat which row in the block. The array $key = a_1, a_2, a_3, ..., a_n$, and key made up by 7 short array: the length of row: $rowlen = a_1, a_2$; row key: $row = a_3, ..., a_{3+i}$; the length of column family: $famlen = a_j$; column family: $fam = a_{j+1}, ..., a_{j+8}$; qualifier: $col = a_{j+9}, ..., a_{j+12}$; timestamp: $time = a_{j+13}, ..., a_{j+20}$; key type: $type = a_{j+21}$, and $j = i + 4$, $i \in [0, 255]$.

Definition 1: In $\forall(n, key)_r$ and $\forall(n, key)_k$ $(r \neq k)$, the $famlen_r \equiv famlen_k$, $fam_r \equiv fam_k$.

Definition 2: In $\forall(n, key)_r$ and $\forall(n, key)_k$ $(r \neq k)$, if $row_r = row_k$, $rowlen_r \equiv rowlen_k$.

Definition 3: In $\forall(n, key)_r$ and $\forall(n, key)_k$ $(r < k)$, if $row_r = row_k$, there would not be a $(n, key)_l, l \in (r, k)$ which the $row_l \neq row_k$.

Corollary 1: Array's length $n = j + 21 = i + 25$, as a result $n \in [25, 280]$.

Hbase automatically sorts records to lexicographic order by row key and then sorts data in same row keys by time stamp from large to small, which the definition 3 means. As the data growing, one data block will be gradually filled up by the data with same row key. As known from definition 1 and definition 2, the length of row key, row key, the length of column family and column family were all same in records which row key are same. It only needs to store same filed once in records with same row key in a block. Therefore, it constructs the first four filed to be a dictionary data.

In the inverted index, the least used operation is deleting. The value of key type filed is input type for most and even all key/value data, thus dictionary data includes key type filed. The value of time stamp and qualifier is in wide range. But the time stamp with 8 bytes is longer than key type, and the change in its high byte is not obvious and slow because the change is progressively increasing. Therefore, the time stamp is included in the dictionary. In practice, time stamp in different key/value will be same when insert data into Hbase in a very short

period of time. To increase compress ratio as far as possible includes qualifier filed into dictionary. As a result, a complete dictionary data was a complete key data.

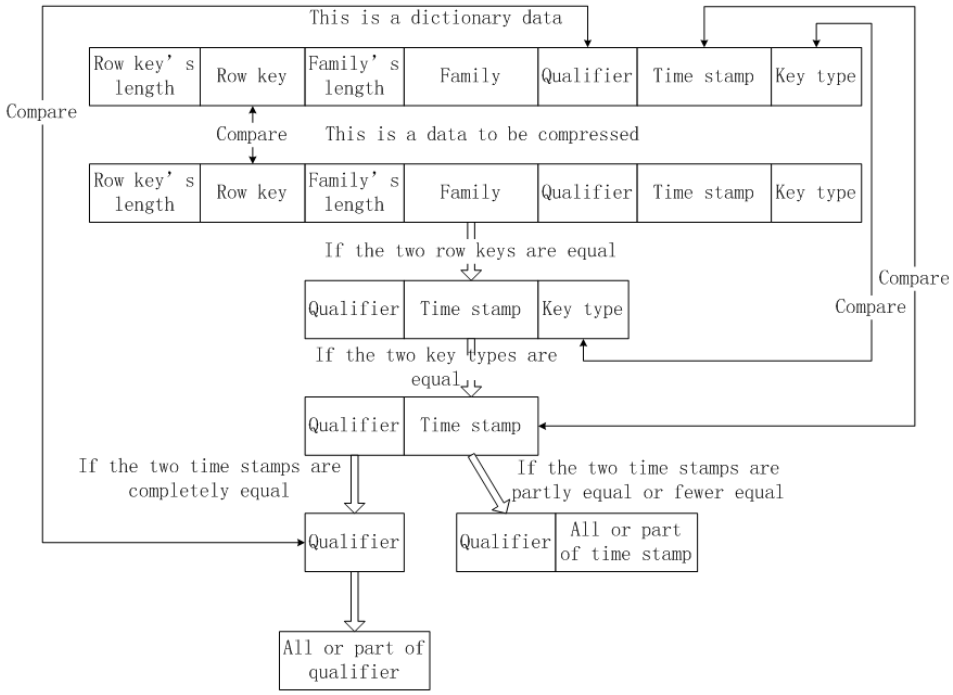

Fig.3.The process of compression method

Compression process shown in figure 3, the compare means to compare the bytes from high byte to low byte, and the process ends when the determine condition is false, and the compressed data is the current data. Hbase is programmed by Java which is big endian. To meet the common comparative method, the qualifier and time stamp were transformed to litter endian. This process compresses data in the order of row key, key type, time stamp, and qualifier. The data do not be compressed when its row key isn't same with dictionary's. That mean the records with former row key are all compressed and the current data is not only the compressed data of itself, but also it is the dictionary data with next row key.

Algorithm 1.Compression algorithm

```
Input: Byte array dict, Byte array data
     Output: Byte array dict, Byte array compressed_data, Integer length_of_compressed_data
     //transform the first two bytes in array to length of row key
1    length_of_rowkey_in_dict = bytesTointeger(dict, 0, 2);
2    length_of_rowkey_in_data = bytesTointeger(data, 0, 2);
     //get the row key which begin from the third byte of array
3    rowkey_in_dict = GetbytesFrombytes(dict, 2, length_of_rowkey_in_dict);
4    rowkey_in_data = GetbytesFrombytes(data, 2, length_of_rowkey_in_data);
   //transform the endian of qualifier and time stamp in data to little endian
5    data = ToLittleEndian(data);
6    if rowkey_in_dict != rowkey_in_data then
7         dict = data
8         compressed_data = data
9         length_of_compressed_data = 2 + rowkey_in_data + 1 + 8 + 4 + 8 + 1;
10   else
          //get the length of dict
11        length_of_dict = 2 + rowkey_in_dict + 1 + 8 + 4 + 8 + 1;
          //get the maximum possible length of compressed data in the condition
12        length_of_compressed_data = 4 + 8 + 1;
          //let i express the last byte of array
13        i = length_of_dict - 1;
14        while dict[i] != data[i] do
15             i--;
16             length_of_compressed_data--;
17      compressed_data = GetbytesFrombytes(data, 2 + rowkey_in_dict + 1 + 8, length_of_compressed_data);
```

B. Decompression Algorithm

The length of compressed array and compressed array was expressed by $(cn, ckey)$. The array $ckey = c_1, ..., c_{cn}$, and $ckey$ made up by 3 short array: qualifier: $ccol = c_1, ..., c_{ci}$, timestamp: $ctime = ..., c_{cj}$, $ctype = c_{ck}$, and $ci \in [1,4]$, $cj \in [0,8]$, $ck \in [0,1]$.

Corollary 2: The compressed array' length $cn = ci + cj + ck$, as a result $cn \in [1,13]$.

Corollary 3: $\forall(xn, xkey)$, if $xn \geq 25$, $(xn, xkey)$ was (n, key) data type. If $xn \leq 13$, $(xn, xkey)$ was $(cn, ckey)$ data type.

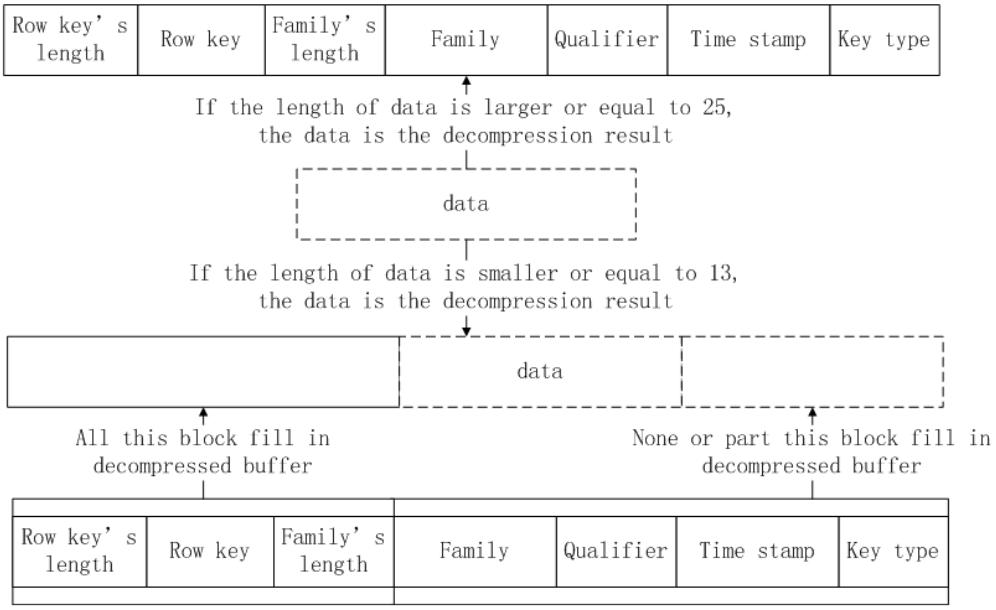

Fig.4.The process of decompression method

Decompression process shown in figure 4, the dictionary data and compressed data can be distinguished by their length, as it proved by the corollary 3. If the length of data larger or equal to 25, that means it is the dictionary, and itself is the decompressed data. If the length of data smaller or equal to 13, it means it is the compressed data. The decompression data needs to get the missing part from dictionary data.

Algorithm 2.Decompression algorithm

```
        Input: Byte array dict, Byte array data, Integer length_of_data
        Output: Byte array dict, Byte array decompressed_data, Integer length_of_decompressed_data
1       if length_of_data > = 25 then
2           dict = data
3           decompressed_data = data
4           length_of_decompressed_data = length_of_data
5       else
            //transform the first two bytes in array to length of row key
6           length_of_rowkey = bytesTointeger(dict, 0, 2);
            //Calculate the length of the first four filed
7           length_of_fisrt_four_filed = 2 + length_of_rowkey + 1 + 8;
            //put the bytes of the first four filed into the decompressed_data
8           decompressed_data = GetbytesFrombytes(dict, 0, length_of_fisrt_four_filed);
            //add the data to the following valid buffer of decompressed_data
9           decompressed_data += data;
10          length_of_dict = length_of_fisrt_four_filed + 4 + 8 + 1;
            //current the number of valid bytes in decompressed_data
11          length_of_decompressed_data = length_of_fisrt_four_filed + length_of_data;
12          if length_of_decompressed_data < length_of_dict then
                //add the last part to the following valid buffer of decompressed_data
13                  decompressed_data += GetbytesFrombytes(dict, length_of_decompressed_data - 1, length_of_dict -
length_of_decompressed_data);
14              length_of_decompressed_data = length_of_dict;
            //restore the endian of qualifier and time stamp to big endian
15      decompressed_data = ToBigEndian(decompressed_data);
```

C. Algorithm Analysis

In the compression algorithm, the main cost is byte compression. When the row key of dictionary and of data is different, it needs the minimum number of comparisons that it only needs to compare the number of the length of shorter row key. When the row keys are same between dictionary and data, it maybe needs the maximum number of comparisons that it needs to compare the whole key part. So, the time complexity of compression one recode is $O(n)$, and the complexity of compression one block is $O(n \times recodes)$. The total complexity of compression is $O(N)$ which N means the total number of bytes in one block. In the process of compression, it needs extra space to save dictionary data. So the space complexity of compression is $O(M)$ which M means the length of dictionary data.

In the decompression algorithm, the main cost is to change little endian to big endian. The bytes of qualifier are 4 and time stamp filed are 8. Every time, changing a 32bits digit from little endian to big endian needs 4 shift operations, and changing a64bits digit needs 8shift operations. Each time, decompression one recode needs 12 shift operations. So the time complexity of decompression one block is $O(12 \times recodes)$. The total complexity of decompression is $O(N)$ which N means the total number of recodes in one block. In the process of decompression, it needs extra space to save dictionary data. So the space complexity of decompression is $O(M)$ which M means the length of dictionary data.

D. The Compression of A Key/Value

A key/value data can be divided into 4 parts. Except the key and value, there still are two filed at the beginning to record the length of key and value. The length is compressed by variable byte codes(VBC)[5]. The compression solution is shown in follow figure:

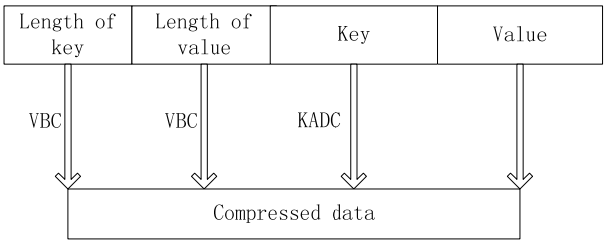

Fig.5.The hybrid compression method of a key/value

The complete process of a key/value as follow:

First, the key was compressed in the row with the dictionary compression method. The result and the length of compressed data were both saved in catch.

Second, the compressed length of key was written into the file, then the compressed length of value was written into the file, and last the compressed key and value were written into the file in order.

V. Experimental Methods And Results

Thanks to this paper only involved about compressing on data block and to simplify experimental and to enhance effectiveness of the experimental results, part experiments in this paper were not doing under the environment of Hbase. But in order to verify the solution, the processed data of Hbase was put into the program which simulated the reading and writing processing in Hbase.

There were 2539 web pages used as the experimental data in this paper. They all came from the Internet. After web purification and segmentation processing, these data was inserted into Hbase database in stand-alone environment, and then the Hbase services were immediately stopped. The purpose of that was to make Hbase written its data in memory to disc. A totally generated 3 Hfile, and their size were 11873280 bytes, 15609856 bytes and 46880768 bytes. And then these files were processed. The way was to extract all data blocks from them and save this data to file A, B and C. their size was 11863262 bytes, 15595881 bytes and 46843112 bytes.

The basic information of experimental computer as follow:

The version of system operation: RedHat CentOS release 5.5;

The type of CPU: Intel(R) Xeon(R) CPU E5345 @ 2.33GHz (there were 2 CPUs);

The capacity of memory: 4046280 kB.

The name of the compression in the paper was kadc. It was written by ASCI C and compiled to static link library with O2 compilation optimization by GCC compiler. The comparative experiments adopted the library of LZO-2.05 [16] and library of ZLIB-1.2.5 [15] . The default compilation optimization level of them was O2 and O3. Every comparative experiment was the same framework. The only difference was to call different compression and decompression library. All operations of reading and writing file, compressing and decompression were in a data block units.

TABLE III

THE RESULTS OF EXPERIMENT

Method	Compression ration	Compression efficiency (MB/s)	Decompression efficiency (MB/s)
KADC	20.3%	468	891
LZO	26.0%	374	818
GZIP	15.8%	25	239

From table III, it can be found that the key as dictionary compression method is superior to LZO on the compression ratio of inverted index table. And the compression efficiency and decompression of key as dictionary is superior to GZIP's and LZO's. The Calculation method of compression efficiency and of decompression efficiency is:

$$Compression\ efficiency = \frac{The\ size\ of\ original\ file}{Compression\ time},$$

$$Decompression\ efficiency = \frac{The\ size\ of\ original\ file}{Decompression\ time}.$$

The compression time and decompression time don't include time of I/O consuming. And in the calculation method of decompression efficiency, the size of original file is the size of decompressed file rather than compressed file's.

TABLE IV
COMPRESSION TIME (S)

Method	11M	15M	45M
KADC	0.031	0.046	0.119
LZO	0.044	0.048	0.131
GZIP	0.465	0.593	1.767

TABLE V
DECOMPRESSION TIME (S)

Method	11M	15M	45M
KADC	0.012	0.018	0.061
LZO	0.014	0.019	0.073
GZIP	0.050	0.067	0.188

In table IV and table V, the file size is the original file size. The time means the value of compression or decompression time plus the time of I/O consuming. It could be found that the KADC's consuming time is less than others.

TABLE VI
COMPRESSION RATION AND QUERYING TIME(S) UNDER THE HBASE REAL ENVIRONMENT

Method	Compression ration	Querying time (s)
No compression	100%	7.91
KADC	20.3%	4.32
LZO	26.0%	5.60
GZIP	15.8%	7.02

In table VI, all the result is monitored under the real Hbase environment. In the compression ration test, it closed the compression on meta data of LZO and GZIP. In the querying time test, the querying time means scanning all data in index table that the Hbase reads all data on disc to memory without any process.

VI. EXPERIMENTAL CONCLUSION

Inverted index is principally applied in the technology of search engine. Since, to meet the technical characteristics of search engines, it greater emphasis on the immediate accessibility of inverted index. The need reflected in this article is mainly the execution time of decompression as short as possible.

It assumes the size of source data is S(MB), compression ratio is C, decompression efficiency is D(MB/s), the average ratio of reading data from disc to memory is T(MB/s), and the time of whole decompression process is t. Therefore:

$$t = \frac{S}{D} + \frac{SC}{T}$$

The experimental data was taken into the above equation. It would be found that: $C_{kadc} < C_{lzo}$ and $D_{kadc} > D_{lzo}$. Therefore, the following conclusions can be drawn:

$$t_{kadc} < t_{lzo}.$$

Now, it assumes that:

$$t_{kadc} \geq t_{gzip}. \quad 1$$

From 1, it can get that:

$$T \leq \frac{(C_{kadc} - C_{gzip})D_{gzip}D_{kadc}}{D_{kadc} - D_{gzip}}. \quad 2$$

From 2, it can get that only when $T \leq 14.69MB/s$, the 1 is right. But current, the T is much larger than this number. So in the most cases, it can get the result that the $t_{kadc} \geq t_{gzip}$ is true.

This conclusion can be verified from table V and table VI. Therefore, Kadc than Gzip or Lzo was more suitable of inverted index compression.

Above conclusions apply to the stand-alone environment. If in a distributed environment, it needed to add the time of network transmission. The time for consumed of whole decompression process is t:

$$t = \frac{S}{D} + \frac{SC}{T} + t_{net}.$$

t_{net} was the time of network transmission. It was also related to the size of compressed file and related to the compression ratio in the case of same source size. So, time of network transmission did not have an impact on experimental conclusion, but it would increase gap between Kadc and Lzo methods.

VII. CONCLUSION

Through analysis on experiment and experimental results, the key as dictionary compression method on particular inverted index table under the Hbase database has a high immediacy, and it meets the requirements of search engine for instant response. The key as dictionary compression method for key also can apply to other big table design under Hbase. But the source of Hbase just give the option of Lzo and Gzip, so the source must be modify to use this method in Hbase and at the same time the Java interface of this method should be gave. At last, the meta block also can be compressed except data block in Hbase's Hfile.

VIII. ACKNOWLEDGEMENT

This work was supported by 2012 China National Social Science Fund (No. 12XSH019) and 2010 Natural Science and Technology Development Fund of Chengdu University of Information Technology (No. CSRF201002).

REFERENCES

[1] Manning, C.D., P. Raghavan and H. Schütze, *Introduction to Information Retrieval*. 2009. http://nlp.stanford.edu/IR-book/.

[2] Shannon, C.E., A Mathematical Theory of Communication. Bell System Techical Journal, 1948. 27: p. 379-423, 623-656.

[3] FengGui, LingQiwei and ChenDonghua, Information Theory and Coding Techniques. 2007, Beijing China: TsingHua Uniersity Press.

[4] LiuXingyu, A Research of Full-Text Retrieval Based on Inverted Index. 2004, Huazhong University of Science and Technology: Wuhan China.

[5] PanShengyi, A Study on Compression Algorithm Performance Based Inverted Index. 2009, Hangzhou Dianzi University: hangzhou China.

[6] A, M. and S. L, Binary interpolative coding for effective index compression. Information Retrieval, 2000. 3(1): p. 25-47.

[7] Williams, H.E. and J. Zobel, Compressing Integers for Fast File Access. The Computer Journal, 1999. 42(3): p. 193-201.

[8] V, A. and M. A, Inverted index compression using word-aligned binary codes. Information Retrieval, 2005. 8(1): p. 151-161.

[9] WangHu and WangQianping, Research and Analysis on Five Inver ted Files Compression Techiques. Computer Engineering and Applications, 2006(07): p. 169-173.

[10] Ziv, J. and A. Lempel, A Universal Algorithm for Sequential Data Compression. IEEE Transactions on Information Theory, 1977. IT-23(3): p. 337-343.

[11] Storer, J.A. and T.G. Szymanski, Data compression via textual substitution. Journal of the ACM , 1892. 19(4): p. 928-951.

[12] CaoDengjun, Study of Real-time and Lossless Compression Encoding. 2004, National Central University: Taoyuan Taiwan China.

[13] The Apache Software Foundation, The Apache Hbase Book. 2010. http://hbase.apache.org/book/book.html.

[14] The Apache Software Foundation, The Source of Hbase. 2011. http://hbase.apache.org/.

[15] Roelofs, G., J. Gailly and M. Adler, A Massively Spiffy Yet Delicately Unobtrusive Compression Library. 2010. http://zlib.net/.

[16] Oberhumer, Lzo. 2010. http://www.oberhumer.com/opensource/lzo/.