A High-Performance Key-Value Query Solution Based on Hash dictionary and Trie tree

Zhijia Yan
Zhejiang Yuying College of Vocational Technology
Hangzhou, China
y163110@126.com

Zhonghan Sun, Yidong Zheng, Jiajun Bu, Wei Wang
Zhejiang Provincial Key Laboratory of Service Robot
College of Computer Science, Zhejiang University
Hangzhou, China
{zhsun, 3100102842, bjj, wangwei_eagle}@zju.edu.cn

***Abstract*—With the development of Internet, tens of billions query were submitted every day. However, the old response mechanism of query which client submits query to server then server fetches data from database and sends data to client can't meet the requirements any more. We know the time of client to server request can't be cut down. To accelerate query speed, we have to avoid the costing time method that fetches data from database. Key-Value Store is popular today, but it has problems in Points barrels strategy and memory usage. In this paper, we propose a High-Performance Key-Value Query Solution which is based on hash dictionary and trie tree. Instead of fetching data from database, we construct a Key-Value dictionary in the memory to accelerate the whole query time. We use epoll of Linux to finish the asynchronous communication of client and server. For the dictionary, we design two different dictionaries like Hash dictionary, sorted dictionary based on trie tree. Hash dictionary uses Points barrels strategy and Minimal Perfect Hash Function. Sorted dictionary is based on trie tree. To illustrate performance of our solution, we test and record the performance of memory usage and query per second. It turns out that hash dictionary has more effective search time and sorted dictionary can hold more data.**

Keywords—key-value query; hash; tire tree

I. INTRODUCTION

E-commerce like Alibaba and Amazon plays an important role in current internet era. Every day, people search the goods they want. Tens of billions query was sent to E-commerce companies. The old response mechanism of query which client submits query to server then server fetches data from database and sends data to client can't meet the requirements [1]. With tens of billions goods records, one query in the database will cause too much time for real-time e-commerce transaction. Key-Value query[2,3] is a solution for slow database query. We can map the database as pair of Key-Value in the memory, but it's limited by the memory capacity.

In this paper, we propose a High-Performance Key-Value Query Solution which is based on hash dictionary [4,5,6] and trie tree [7]. Instead of fetching data from database, we construct a Key-Value dictionary in the memory to accelerate the whole query time. We use epoll of Linux to finish the asynchronous communication of client and server. For the dictionary, we design two different dictionaries like Hash dictionary, sorted dictionary based on trie tree. Unlike traditional Key-Value Query, hash dictionary uses bucket strategy and Minimal Perfect Hash Function to map the pairs of key-value to different index files then load the index files to memory. Sorted dictionary is based on trie tree. Hash dictionary has more effective search time and sorted dictionary can hold more data.

II. RELATED WORK

Key-Value [2,3] Store, a non-relational database, is a very popular big data solution. Relational database is made up of table which is consisting of records. And record has the same fields which are useful in table joins but cause much redundancy in the same time. The redundancy and table joins are the bottleneck of relational database. On the other hand, Key-Value Store is not certain or unchanged. Every record can store different fields. And for the Key-Value Store, we prefer using the pair rather than record. Just because the flexibility of Key-Value pair we can construct more useful table which will get much less redundancy and more data in the same memory capacity.

Key-Value Store is essentially a hash table for each key should one value. Table I shows a table which has the key named ID and value is consist of attribute1、attribute2、attribute3. The type of every attribute of value can be any type like number, text, image and so on. Generally, Key-Value pairs are serialized in files which keys are order lexicographically. Key-Value Store can handle big data with map-reduce in cluster of server.

TABLE I. STORAGE MODE OF KEY-VALUE PAIR

ID	Attribute1	Attribute2	Attribute3
1001	20	Yes	Image1
	21	No	Image2
	22	Yes	Image3

III. A HIGH-PERFORMANCE KEY-VALUE QUERY SOLUTION BASED ON HASH DICTIONARY AND TRIE TREE

In order to meet the needs of different scales of data, the server is designed to provide users with two types of Key-Value pairs, which are hash dictionary, sorted dictionary based on trie tree. We provide two different lengths of the key and

value which one is 8-bytes fixed-length and the other is 8-bytes key and variable-length value 0-255 bytes. Variable length dictionary is a dictionary of fixed-length simple modification, for simplicity, in the description of the two types of dictionary design algorithm, are described fixed length in the dictionary. Each dictionary is a map object, we design only load and find two interfaces, one for loading a dictionary and one for searching corresponding key in the dictionary.

When the amount of data is between 100M to 5G, hash dictionary is recommended. Hash dictionary is designed to hash keys of Key-Value into different files. Hash function as the index is stored in memory. In practice, combining hash bucket policy and Minimal Perfect Hash Function (MPHF) will greatly reduce the amount of memory required compared to traditional Key-Value Store. A hash function is called Perfect Hash Function, if all the elements of the set S are mapped into a set of integers without conflict [8]. A perfect hash function called minimal perfect hash function if all the elements of the set S are mapped to the n consecutive integers, where n is the number of elements of the set S [9].

All Key-Value hash dictionaries store in the disk, and only index functions which map key and file in the disk load into memory at query time. Hash dictionary solution contains three steps which are File point barrels, generating MPHF and file merging. If the Key-Value pairs are too much, we may not find a MPHF. We have to point Key-Value pairs into barrels. After that, we must merge index files to one file in order to reduce time of read disk. Details can be found in the Algorithm 1.

Algorithm 1 *Key-Value Query Based on Hash Dictionary*

```
Input: Key-Value pairs ;
Output: a set of Key-Value Files, Index File;
1.  If can Find MPHF for KV
2.      Return Generate Index file and Key-Value File by
MPHF
3.  Else
4.      Key-Value pairs point into uniform barrels as Set
F={f_i};
5.      Foreach f_i in F
6.      Generate Index files as Set I and Key-Value Files by
MPHF
7.  End
8.  Merge Set I as one File as Index File
9.  Return Index File, Key-Value Files
```

When the amount of data is more than 5G, we recommend using the sorted dictionary. Since the server no longer provide other types of dictionary, sorted dictionary can theoretically handle magnitude of more than TB using set multi-level index. Because multi-level index files can't be all loaded into memory, it will limit the query performance. If user has a greater magnitude of data requirement which is more than TB level, distributed platform is needed [10]. Sorted dictionary requires that the Key-Value pairs are already sorted. So sorted dictionary means that we should sort Key-Value by keys in files. Sorting can be done in many ways which a lot of research [9,10] had been done in this area. Here we mainly consider the way to reduce the use of memory using trie tree [7].

Trie tree can offer dense index in order to reduce the use of memory. The solution of sorted dictionary will always keep one trie index file in memory. Tire index is not a traditional Key-Value index which is bitwise trie index which is string with only 0 or 1. One bitwise string maps to one key. Once the value of key is required, we first search the key (character string) in the trie tree and get the bitwise string for the key(character string). Then we can get the address of the bitwise string which is representing the character string. The whole trie tree can be recursively defined as:

$$T=Len(L)+L+R \quad (1)$$

Where L is the left subtree , R is right subtree and Len(L) is the leaf number. So constructing the trie tree need to know the number of Key-Value pairs and left subtree left number. Because the dictionary is sorted, we can get the length of left subtree by binary search. After that, the length of right subtree is Len(T)-Len(L).So we can construct the trie tree recursively until now. Details can be found in the algorithm 2.

Algorithm 2 *Construct Trie Tree*

```
Name: ConstructTire
Input:  a set of sorted Key as K, length of left subtree as
        len_L;
Output: Trie Tree;
Initialization: len_L=len(K),
1.  If len_L=0 or len_L=1:
2.      Return "";
3.  Else:
4.      Length of left subtree
        len_Lsub=binary_search(K,len_L)
5.  L=K[0: len_Lsub]
6.  R=K[len_Lsub:len_L]
7.  Return len_Lsub+ ConstructTire(L,len_Lsub)
    + ConstructTire(R,len_L-len_Lsub)
```

After constructing trie tree, we can search key recursively. Details can be found blow in algorithm 3.

Algorithm 3 *Search Key in Trie Tree*

```
Name: TrieSearch
Input: target key K , length of trie tree, trie tree T;
Output: bitwise string bitK ;
1.  If len_L=0 or len_L=1:
2.      Return ""
3.  Else:
4.      If K[0] equal to character of 0:
5.          kLeft= K[1:-1]
6.          Len_left=T.leftChild
7.          Return
    "0"+TrieSearch(kLeft,Len_left,T.leftChild)
8.      Else:
9.          kRight= K[1:-1]
10.         Len_right=T.rightChild
11.         Return
    "0"+TrieSearch(kRight,Len_right,T.rightChild)
12.         Return
    "1"+TrieSearch(kRight,Len_right,T.rightChild)
```

When getting the bitwise string of character string, we can get the address of character string by dictionary value with the bitwise key.

IV. PERFORMANCE TEST

Here we only care two aspects, memory usage and query per second. Traditional Key-Value store will be used as a baseline. Memory usage and query per second can be found in Fig.1.

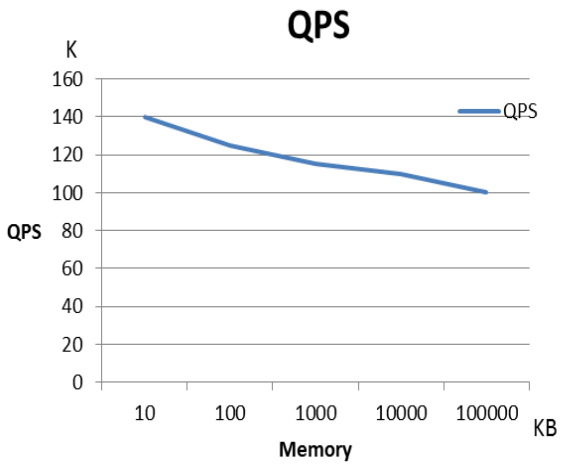

Figure 1 Traditional Key-Value

Traditional Key-Value store can offer 100 thousand query per second for memory less than 100M.

For Hash Dictionary Key-Value store, there are three aspects determining performance which are finding value of key in MPHF, computing the offset of the key in disk and read value from disk. First two steps cost constant time, the last step which is read value from disk costs the major time. Hash dictionary key-value store loads MPHF file into memory instead of data file. So we will do three different measurements for hash dictionary key-value store which are data size, memory usage and query per second. Data size and query per second can be found in Fig.2.

Figure 2 Hash Dictionary Key-Value

We can see that hash dictionary Key-Value store guarantees 10 thousand query per second for data size less than 5G. Compared with traditional Key-Value store, query per second of hash dictionary decreases to one percent of t Key-Value but has 50 times upper limit of key-value pairs.

Memory usage is related to the size of MPHF files. We will give the test of the size of MPHF files and memory usage. The size of MPHF file is one of ten of data size. Details can be found in Fig.3.

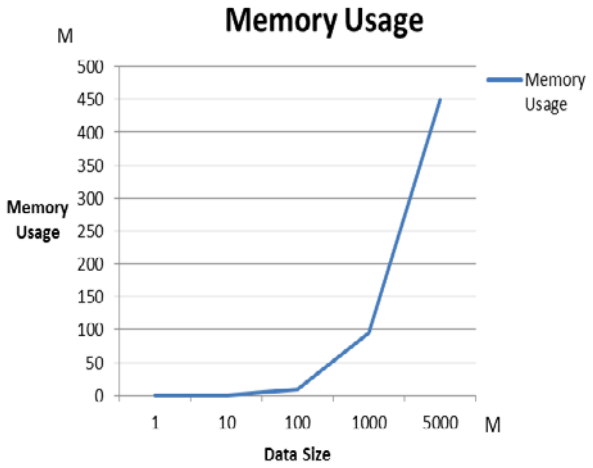

Figure 3 Memory Usage for hash dictionary Key-Value

For sorted dictionary Key-Value store, there are three aspects determining performance which are finding value of trie tree of key, computing the offset of the key in the disk and read the value of key from disk. We will give three tests which are data size, memory usage and query per second. Data size and query per second can be found in Fig.4.

Figure 4 Sorted Dictionary Key-Value

Sorted dictionary Key-Value store can offer 7 thousand query per second for data size less than 20G.Memory usage is related to the size of trie tree. We do the test for data size and memory usage. Details can be found in Fig.5.

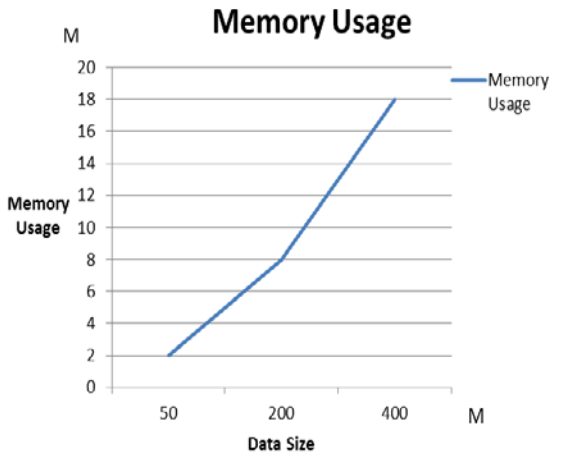

Figure 5 Memory Usage for sorted Key-Value

We just do the memory test for 400M because the computing power is the bottleneck of generating trie tree. But we pretty sure that it will not be bottleneck for practical application.

V. CONCLUSION AND FUTURE WORK

We propose a High-Performance Key-Value Query Solution which is based on hash dictionary and trie tree. Instead of fetching data from database, we construct a Key-Value dictionary in the memory to accelerate the whole query time. We use epoll of Linux to finish the asynchronous communication of client and server. For the dictionary, we design two different dictionaries like Hash dictionary, sorted dictionary based on trie tree. Unlike traditional Key-Value Query, hash dictionary uses bucket strategy and Minimal Perfect Hash Function to map the pairs of key-value to different index files then load the index files to memory. Sorted dictionary is based on trie tree. Hash dictionary has more effective search time and sorted dictionary can hold more data.

Our solution can handle TB level data. When the data scale is greater than TB, the multi-index solution maybe a good choice. We will try to design and optimization of multi-level index.

ACKNOWLEDGMENT

This work is supported by Detection Technology of the Web Accessible to the Disabled (2014 Professional Development Project of Domestic Visiting College Scholar of Zhejiang Provincial Department of Education, Grant No. FX2014178)

REFERENCES

[1] A. Anand, C. Muthukrishnan, S. Kappes, A. Akella, and S. Nath.Cheap and large CAMs for high performance data-intensive networked systems. In NSDI'10: Proceedings of the 7th USENIX conference on Networked systems design and implementation, pages 29–29. USENIX Association, 2010.

[2] B. Debnath, S. Sengupta, and J. Li. SkimpyStash: RAM space skimpy key-value store on flash-based storage. In Proc. International Conferenceon Management of Data, ACM SIGMOD '11, pages 25–36,2011.

[3] B. Debnath, S. Sengupta, and J. Li. FlashStore: High throughput persistent key-value store. Proc. VLDB Endowment, 3:1414–1425,September 2010.

[4] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. Communications of the ACM, 13(7):422–426, 1970.

[5] A. Badam, K. Park, V. S. Pai, and L. L. Peterson. HashCache: Cache storage for the next billion. In Proc. 6th USENIX NSDI, Apr. 2009.

[6] D. Belazzougui, F. Botelho, and M. Dietzfelbinger. Hash, displace, and compress. In Proceedings of the 17th European Symposium on Algorithms, ESA '09, pages 682–693, 2009.

[7] A. Andersson and S. Nilsson. Improved behaviour of tries by adaptive branching. Information Processing Letters, 46(6):295–300, 1993.

[8] F. C. Botelho, A. Lacerda, G. V. Menezes, and N. Ziviani. Minimal perfect hashing: A competitive method for indexing internal memory. Information Sciences, 181:2608–2625, 2011.

[9] E. A. Fox, L. S. Heath, Q. F. Chen, and A. M. Daoud. Practical minimal perfect hash functions for large databases. Communications of the ACM,35:105–121, Jan. 1992.

[10] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows,T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed stora