pLSM: A Highly Efficient LSM-Tree Index Supporting Real-Time Big Data Analysis

Jin Wang, Yong Zhang, Yang Gao, Chunxiao Xing
Research Institute of Information Technology
Tsinghua National Laboratory for Information Science and Technology
Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China
{wangjin12, yang-gao10}@mails.tsinghua.edu.cn, {zhangyong05,xingcx}@tsinghua.edu.cn

***Abstract*—Big Data boosts the development of data management and analysis in database systems but it also poses a challenge to traditional database. NoSQL databases are provided to deal with the new challenges brought by Big Data because of its high performance, storage, scalability and availability. In NoSQL databases, it is an essential requirement to provide scalable and efficient index services for real-time data analysis. Most existing index solutions focus on improving write throughput, but at the cost of poor read performance. We designed a new plug-in system PuntStore with pLSM (Punt Log Structured Merge Tree) index engine. To improve read performance, Cache Oblivious Look-ahead Array (COLA) is adopted in our design. We also presented a novel compact algorithm in bulk deletion to support migration of data from temporary storage to data warehouse for further analysis.**

Keywords--Big Data; index; Log Structured Merge Tree; Cache Oblivious; bulk deletion; write performance

I. INTRODUCTION

In the process of informatization, the amount of data has been increasing at a high speed for both individuals and organizations. Data has played a significant role in every industry and business function field, bringing us into the era of data. Therefore such problems brought by "Big Data" have been a common concern of the field. The requirements for Big Data management and analysis are widely different from those of traditional data management. Huge data volumes need to be kept online for querying and analyzing. In addition, queries need to be answered immediately to enable real-time analysis and decision making.

To tackle such problems, we designed a plug-in system called PuntStore. PuntStore makes optimization in storage, distribution, scalability, heterogeneity and security. To satisfy the needs for temporary storage and make real-time analysis, we designed a non-relational database PuntDB. Just like the SAP HANA database [2], PuntDB provides the foundation for other high-level applications in PuntStore. To support real-time data analysis well, index service in PuntDB must be able to deal with large-scale, complex data and provide immediate availability of operational data. A highly efficient index could improve the speed of data retrieval.

We must organize an index into a particular kind of data structure in order to make use of it. B-Tree is a general type of data structure to support index in relational databases. However, B-Tree has a problem of aging and suffers from disk-seek bottlenecks when faced with large volume of data [10]. Generally speaking, the analysis of Big Data relies on a storage type for analytical workloads. The analytical workloads emphasize write throughput and sequential reads over random access. Thus a data structure supporting better write performance is needed.

The Log Structured Merge Tree (LSM-Tree) is a general model to reach write optimization [9]. But LSM-Trees tradeoff read performance for improving write throughput. Cache Oblivious Streaming B-Tree is a dictionary that implements efficiently insertions and range queries [5]. A widely used kind of Cache Oblivious Streaming B-Tree is Cache Oblivious Look-ahead Array (COLA). It enables the logically contiguous elements to be stored in the same block instead of scattered on disk. Therefore the seek time will be much shorter. However, COLA can't make good use of memory as buffer. Moreover, it doesn't support efficient deletion. Thus, we design a new index Punt LSM (pLSM) which could satisfy the needs for performing index probes in write optimized systems.

In the following sections, we will discuss the specific implementation of pLSM index. Section 2 provides the necessary background of our pLSM structure. Section 3 is an overview of the index structures with the similar goal of our pLSM. Section 4 describes our design of pLSM in detail. We also presented a new compact algorithm to solve the problem of deletion for COLA. We offered the analysis of advantages of pLSM in bulk deletion. Section 5 is about the testing of our design of pLSM, showing how it outperformed B-Tree and the original LSM-Tree on different kinds of workloads.

II. BACKGROUND

Nowadays, traditional magnetic disk still acts as the main storage media, so the performance of storage system is decided by hard disk seek time. Storage capacity per disk will continue to grow quickly, but it seems that seek time will change slowly. To make full use of the bandwidth and cope with random I/O, proper data structures for external memory are needed such as index for highly efficient query and inserting.

0730-3157/13 $26.00 © 2013 IEEE
DOI 10.1109/COMPSAC.2013.40

A. B-Tree

B-Tree is a balanced search tree structure designed for external memory [4]. A large number of precious work shows that B-Tree provides optimal random reads. However, when a large number of elements are added into the tree, the external memory will become fragmentized and more random I/Os will occur, which will lead to long seek time. In order to improve write performance, a write-optimized structure is needed.

B. LSM-Tree

A LSM-Tree is an index that consists of one smallest component in memory and several larger B-Tree components on disk [9]. Figure 1 shows the structure of a typical LSM-Tree. The in-memory component is defined as C0 and could be updated in place. The newest data is inserted into C0. After C0 is full, it is merged with the next smallest component C1. Other components on disk increase in size exponentially to ensure that the merge cost is minimized.

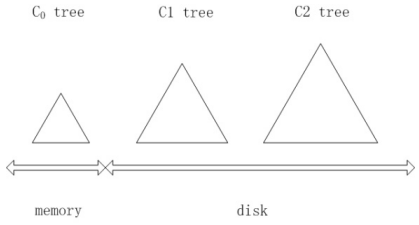

Figure 1. A general structure of LSM-Tree

The fundamental mechanism of LSM-Tree is to defer the process of putting index changes into disk and deal with the write operation in batch. By streaming data in and flushing it back to disk as one streaming operation, we can amortize the disk I/O cost. Moreover, since all components on disk are produced by merging with the next smaller component, LSM-Tree is updated by eliminating random I/O. However, the scan operations in LSM-Tree were not as efficient as those in B-Tree.

C. Cache-Oblivious Data Structure

The B-Tree mentioned above is typically analyzed in the Disk Access Machine model. This model makes an assumption that the memory of size M is organized into blocks of size B and the external memory is arbitrarily large. Within the DAM model, searching is optimal and insertion costs $O(\log_B N)$ block transfers.

Within the Cache Oblivious model by M.Frigo [5], the block size B is unknown to the algorithm so that any memory-specific parameterization could be avoided. Of the extant cache oblivious structures, the most widely mentioned is Cache Oblivious Look-ahead Array (COLA), which supports insertion of N elements in $O(\log N/B)$ block transfers and searches in $O(\log N)$ transfers [6]. Moreover, the COLA consists of $\log_2 N$ arrays, each of which is either completely full or completely empty. Each full array is sorted so that we can use binary search to accelerate searching in one array. The kth array is of size 2^k and arrays are stored in contiguously in memory. An example of COLA is shown in Figure 2. When an element is inserted into a full array, the array will be merged into the next array until all the elements are contained in full arrays.

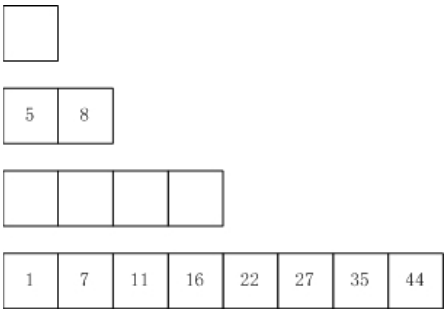

Figure 2. An example of Cache Oblivious Look-ahead Array

III. RELATED WORK

Although original LSM-Tree could improve write performance by eliminating random I/O, it scarifies read performance since in the worst case it involves a maximum of n I/Os (n is the number of components on hard disk) as each component of the tree needs to be checked. Besides, it also brings long write latency because of the asynchronous merge procedure between different components on disk.

The bLSM-Tree designed by Sears is a new LSM-Tree variant [3]. It improves excessive read amplification by protecting C1, C2…CN tree component with Bloom Filters. This method could efficiently reduce query time when the element doesn't exist. In order to eliminate long write pauses and provide optimal write performance, bLSM-Tree implements a mechanism called *spring and gear schedule* to replace the traditional partition schedule [7] and ensure the completion of merge processes at the same time. The primary limitation of bLSM is that the time of merging process is bound with the timing insertions into C0. To avoid blocking caused by this bound, we must halve the size of memory. Besides, we need to estimate the costs of future merge in order to apply this approach. This is not always possible in real-time systems.

Similar work has been done to improve read performance. TokuDB is a store engine for MySQL[1]. It uses an efficient index to speed up query and attain high scalability. By leveraging write-optimized compression, TokuDB achieves up to a 90% reduction in HDD and flash storage requirements, without impacting performance [10]. Fractal Tree Index implemented by TokuDB is an efficient solution to improve writes performance with less tradeoff on read performance. It could run 10 to 100 times faster inserts than B-Tree. Although the query time for Fractal Tree is theoretically no better than B-Tree, it in fact performs much better since Random I/Os are avoided. However, many problems remain to be solved by TokuDB. For instance, TokuDB can't support efficient deletion and updating. And its record's size shouldn't be too large, so

TokuDB can't be used to store BLOB (Binary Large Object).

IV. THE IMPLEMENTATION OF PLSM

As mentioned in the previous section, it is crucial to ensure lookup and scan performance while seeking for write optimization. To attain that goal, we present a new LSM-Tree variant pLSM to address the limitation of LSM-Tree mentioned above. Just like the bLSM implementation is based upon Rose, a column-compressed, log-structured replication [8], we base pLSM on our PuntStore system. We implemented Skip List as the in-memory component for pLSM, which has a fast insertion and can improve the speed of random access indexed lookups. For components on disk, we implements Cache Oblivious Look-ahead Arrays to overcome the shortcomings of "aging" in B-Trees. To fulfill the task of bulk deletion, we design a new compact algorithm to make full use of disk space.

A. The Design of PuntDB

PuntDB is an optimized NoSQL database to support the storage and analysis of Big Data. PuntDB provides the high-performance data storage and processing engine within Punt Store. Originally, PuntDB uses B-Tree as its index structure. However, B-Tree index doesn't perform well in the task of wireless sensor network for data collection. When large volume of data swarms into the database at a high speed, the B-Tree index can't perform so well as to support real-time insertion and query as expected before.

The reason is the "aging" problem of B-Tree. As we know, the B-Tree performance is disk-bandwidth limited. High entropy insertion in B-Tree has a poor data locality, thus causing more random I/O. This also happens in the case of range query. In a new-built B-Tree, range queries have good locality since leaf nodes are laid out sequentially on disk during this period. But when B-Tree becomes aged, the leaf blocks are scattered across the disk because the usage of bandwidth will drop to as little as 1%.[10] To avoid aging and eliminate random I/O, we designed pLSM index instead of B-Tree to support range query in the background of Big Data.

B. COLA For Disk Component

To achieve high read performance of our pLSM-Tree, we implement Cache Oblivious Look-ahead Array as the exponent on disk. To speed up searching in external memory, we protected each component on disk with a Bloom Filter, just as bLSM does. Considering the balance between insertion throughput and lookup cost, pLSM consists of three components: C0 in memory and C1, C2 on disk. In this section we describe the improvements made by approaching COLA by M.A. Bender as the component on disk instead of B-Tree [6].

As is analyzed in the previous section, the block transfer for insertion or deletion needs much fewer disk seeks. When an element is added into the COLA while the first array is full, a series of merge will happen to maintain the structure of COLA. As each merge is performed between the adjacent arrays, an element is at most involved in $\log N$ merges. Since the process of merge sort is very I/O efficient, such merge operation will not lead to extra disk I/Os. The cost for merging per element is $O(1/B)$, on average there are $O(\log N)$ elements to be merged. So with the algorithm in Figure3, average insertion cost is $O(\log N / B)$, which improves greatly from $O(\log_B N)$ for B-Tree. To speed up the insertion process, we added another array of each size called *shadow array* for temporary storage. At the beginning of each step, the shadow array is empty. As is shown in Figure 6, during the merging process each array will be merged to the shadow array with the same size and then to the next array. This measure trades off space for time and ensures an efficient insertion performance.

Algorithm 1 Insert(*Item*)

```
Parameter: Item: the item to be inserted into COLA.
1:  curnum: the current number of items in COLA; L1, L2, result: temporary lists used to
help do merging
2:  if COLA contains Item then
3:      return
4:  if COLA is full then
5:      call for Rolling merge;
6:      return
7:  Let cur be the first empty level in COLA;
8:  if cur=0 then
9:      Insert Item into level [0] of COLA;
10:     Increase curnum by 1;
11: else
12:     Insert Item into L1;
13:     copy level [0] of COLA into L2;
14:     mergeSort L1 and L2 into result;
15:     set level[0] to be empty;
16:     for k=1 to cur-1 do
17:         Replace L1 with result;
18:         Replace L2 with level[k] of COLA;
19:         mergeSort L1 and L2 into temporary result;
20:         Set level[k] to be empty;
21:     Replace level [cur] of COLA with the list result;
22:     Set level[cur] to be full;
23:     Increase curnum by 1;
24: return
```

Figure 3. The Insert Algorithm for COLA

Bloom Filter is a random data structure with high space efficiency. By implementing Bloom Filter to the components on disk, we could reduce the cost of point query from N to 1+N/100.(N is the number of component on disk) The amount of memory it required is related to the number of elements to be added, not the size of them, so the memory overhead of Bloom Filter is insignificant.

One limitation of Bloom Filter is false positive, we may wrongly judge a non-exist element as existing in the set. Another limitation is that Bloom Filter doesn't support deletion. To support deletion in our pLSM, some modifications need to be implemented.

The LSM-Tree could transform random I/O to sequential I/O by performing batch write. A COLA with N elements have log N arrays, one array for each power of two. This mechanism could ensure that all the elements in one COLA could fill up one or several blocks, which provide great

convenience for lookups and scans. So we chose COLA as the component of external memory.

Another advantage of COLA is that each array is sorted, so we can easily use binary search on each of them to attain $O(\log N)$ time cost for each array. Since there are totally $\log N$ arrays, the total cost of searching is $O((\log N)^2)$.

Unlike B-Tree, COLA does not theoretically support range query well. In order to do a range query, we need to scan each full level to judge whether a key of the element is in the range. But when dealing with mass amount of data, the time cost for range query in COLA is generally less than that in B-Tree. Because all the elements of COLA are centralized stored in consecutive blocks, while in an aged B-Tree even elements on contiguous blocks may scattered on the disk. However, in our experiment the range query of pLSM doesn't perform well due to limited data size.

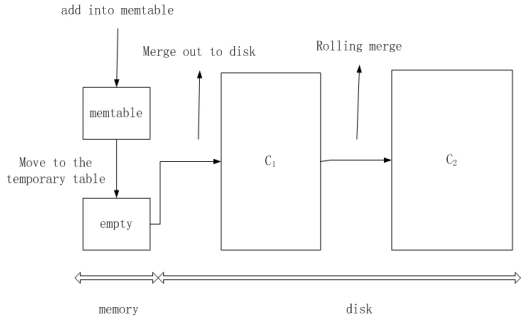

Figure 4.The process of Rolling Merge

In a LSM-Tree when Ci component is full, it will be merged into the next smaller component Ci+1 in a process called rolling merge, as is shown in Figure 4. This merge process is closely related to online B-Tree merging. B-Tree merging addresses situations where contents of a single table index have been split across two physical B-Trees that now need to be reconciled [12]. This process could reduce the storage requirements but add complexity of the merging algorithm. Besides, since B-Tree needs to maintain a global ordered array of leaf nodes, extra cost would be needed in the process of rolling merge. This procedure needs complex mechanism of concurrent control and locking operation, which could not meet the need of availability for NoSQL databases.

If COLA instead of B-Tree is used in pLSM, the rolling merge process could be simplified. COLA doesn't call for a global ordered structure, so what we need to do is just insert all the elements into Ci+1, with the disk I/O efficiency of merge process. It is also much easier to schedule the merge process between components.

C. Dealing with Bulk Deletion

One of COLA's limitations is that it hasn't yet supported efficient bulk deletion operation. This is because once an element is removed, all of the other elements need to be reallocated to the upper levels to maintain the whole structure since no partly full arrays is permitted in COLA. So there has not been an efficient way of deletion for TokuDB.

To support real-time data warehouse, we must ensure the non-stop availability and rapid updating [14]. Since updating is a deletion followed by an insertion, the ability to efficiently delete bulk of data from a table is very important. Therefore, it is necessary for our pLSM index to support efficient bulk deletion. T. Lilja [13] provided an efficient two-phase online bulk deletion algorithm for B-Tree index on a multi-attribute key, which could be performed similarly in our pLSM. To begin with, we could mark elements in arrays of COLA as deleted in the first phase. These marked elements still take up spaces and will be merged to next arrays once new elements are added. But they are invisible for query operation since they have been marked as deleted. These elements will be physically deleted when the component is full and the process of rolling merge begins. Or after the *empty rate* of a COLA reaches its threshold value, the compact algorithm will start to clear the deleted item. In the implementation of COLA, there is no structure modification at all. What we need to do is just adding the elements that are not marked as deletion to the next component and discarding the marked ones. And since the size of arrays in COLA is a power of 2, we could take advantage of this feature and merge them into the next component in batch. In order to deal with deletion in the largest component, the pLSM needs to be regular compacted. The compact algorithm will be shown in Section E later.

D. A Variant of Bloom Filter

In the previous section, we have discussed about using Bloom Filter to speed up judging whether a given element is in the LSM Tree. However, the original Bloom Filter doesn't support deletion because one bit in the Bloom Filter is shared by several keys. If we simply set a bit related to one key from 1 to 0, many other keys will be influenced. This makes components in external memory append-only ones. To solve this problem, we created a variant of Bloom Filter as *Delete Filter* and protected the on-disk components with it together with Bloom Filter.

The *Delete Filter* is the same as Bloom filter, except that we add the operation of delete. In the operation of delete, we set the bit corresponding to the given key from 1 to 0. This operation is an obvious mistake in Bloom Filter, but when we combine the usage of Bloom Filter and Delete Filter, this mistake will not affect the accuracy of query and only brings some extra overhead in searching a non-existing element in COLA. When an element is marked as deleted, we will add it into the Delete Filter. When judging whether a component contains a given key, we first search it in the Delete Filter. If it is in the Delete Filter, we will conclude that it doesn't exist. If it is not in the Delete Filter, we will turn to the Bloom Filter as we usually do.

When we update a record with a given key, we need to delete the record and re-insert it with the same key but different value. Then we would first set the corresponding

bit of the key from 1 to 0 in Delete Filter. Thus it will inevitably influence other keys. For instance, one bit in Delete Filter is related to three keys A, B and C. When we re-insert key A and set this bit to 0, if we continue to judge key B, the Delete Filter will wrongly judge that B exists in the component while in fact B has been deleted. But according to our Search algorithm (in the chart of algorithm 2), even we wrongly judge B is in the component, since we can't find it in any array of COLA, the search algorithm will return null value in the end. The mistake caused by Delete Filter only results in extra search cost for looking up a non-exist element. But at the same time, we could support deletion as well as insertion for on-disk components protected by Bloom Filter. And since the number of updated elements comprises only a small part, the overhead brought by Delete Filter will be very tiny. Thus it is a reasonable solution to use Delete Filter to support deletion in pLSM while implementing Bloom Filter to speed up searching.

E. Compact Algorithm for COLA

Algorithm 5 Compact()

```
Parameter:
1: num: current number of existing items; curlv: the max level after compact; tmpls: the
temporary list of items
2: delnum: the number of deleted elements in COLA; curnum: the current number of items in
COLA
3: for i= maxlevel-1 to 0 do
4:     if capacity of level[i] <= num then
5:         curlv=i,break;
6: for each level level[q] in COLA do
7:     if q!=curlv and level[q] is full then
8:         for each item level[q][k] in level[q] do
9:             if level[q][k] is not deleted then
10:                if curlv is empty then
11:                    add level[q][k] into level[curlv];
12:                else if
13:                    curlv is full and there are deleted items in level[curlv] then
14:        Replace the deleted item in level[curlv] with level[q][k];
15:        Find the next deleted item in level[curlv];
16:            else
17:                insert level[q][k] into tmpls;
18: if curlv is empty then
19:     set curlv to be full;
20: set curnum=num;
21: set delnum=0;
22: do mergeSort of level[curlv] to keep it in order;
23: for each item it in tmpls do
24:     re-insert it into COLA;
```

Figure 5. The Compact Algorithm for COLA

When more elements are logically deleted, there will be many "holes" in COLA. In order to reduce system overhead, C1 component must be compacted before being merged into C2. C2 component also needs compacting to physically remove marked elements.

To solve the above problem, we design a *compact algorithm* for our pLSM. With the help of compacting, we could and improve space utilization in pLSM. To describe the degree of fragments in COLA, we present a parameter *empty rate*, which is the ratio of current number to that of marked elements in COLA. When the empty rate reaches a threshold *max empty rate* the compact procedure will begin to work. To describe the algorithm, we define *deleted elements* as the marked elements, *existing elements* as the elements that are not marked.

The basic idea is to find a largest array *current array* according to the number of existing elements, and replace the deleted elements with elements in other arrays. When all the elements in current array are existing elements, we store the remained existing elements in a temporary array and clear up all but the current array in COLA. Then we perform a bulk load of all remaining elements into COLA. The specific algorithm is described in Figure 5.

V. EXPERIMENT

In this section, we compare the performance of pLSM with B-Tree and original LSM-Tree (we compare the B-Tree index used by our PuntDB).

A. Experiment Setup

Our experimental setup consists of a server machine with 32GB RAM and a 2.40GHz Intel(R) Xeon E5620 CPU with 2 cores. The environment for our workbench is Windows server 2008. The data set for insertion and deletion experiments is key-value pairs in String format that are randomly generated. We set the size of in-memory component C0 of pLSM to be 128MB, the C1 component to be 2GB and the C2 component to be 16GB. For B-Tree, we use the 128MB RAM as its buffer.

B. Insert Performance

To test the insert performance, we tested the elapsed time for different index structures as records are inserted. We used two kinds of data set: one with random keys and the other with ordered keys. Since the index buffer is 128MB, the time will begin to grow rapidly after the index size reached 128MB in this experiment. From the result of the experiment, we could see that in both data sets B-Tree has an obvious problem of aging. By contrast, pLSM has a good performance for random insertion. Even pLSM meets the worst insertion case; the insertion time is still acceptable because there is no need to deal with reorganization since the structure of COLA will never change as B-Tree does once it is initialized.

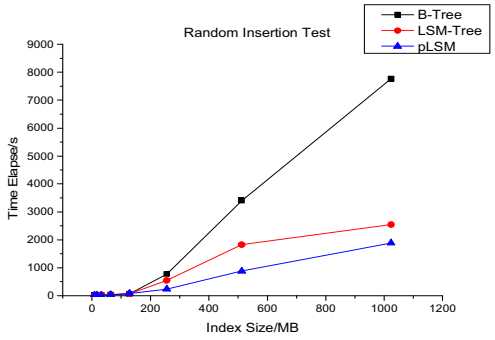

Figure 6.The result of random insertion

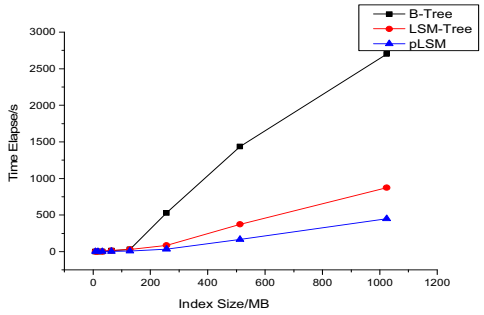

Figure 7.The result of ordered insertion

C. Query Performance

We tested the query performance for each structure by pausing insertion for a while to query the data. To test the raw query efficiency of different index structures, we tested three types of query:

1. Query Set 1: This is a query set of point query. A point query will exactly return the corresponding value. The keys are randomly chosen from the inserted ones.

2. Query Set 2: This is a query set of range query. Many small ranges (contains 100 items each) are randomly chosen to be tested.

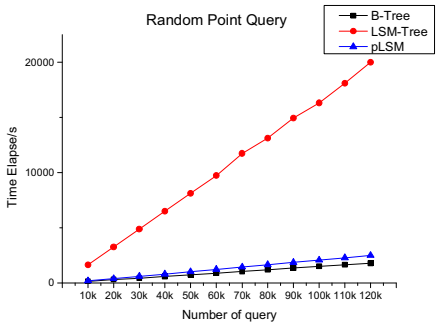

Figure 8. The result of random point queries.

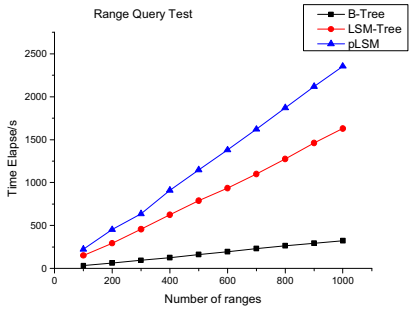

Figure 9. The result of random range queries.

We could conclude from Figure 8 and Figure 9 that pLSM is optimized for point query. But since there is no efficient range query algorithm for pLSM, it has a worse performance in range query.

VI. CONCLUSION

The implementation of pLSM is cost-effective in dealing with Big Data and offering real-time response for lookups and insertions. To attain the goal of write optimization, we use the LSM-Tree model. To accelerate the query response, we use COLA as the on-disk component. We also make a variant of Bloom Filter and two-phase bulk deletion algorithm to support efficient deletion in pLSM. To make full use of space and reduce the frequency of rolling merge, we put forward a compact algorithm for COLA.

Furthermore, a number of technical issues remain to be done. To achieve better performance of COLA, the PuntDB must have better data compression mechanism; in some cases the whole COLA component is too large to be created in the memory at one time, we need to improve the dynamic expand mechanism. Moreover, how to define the threshold of *empty rate* in compact algorithm needs to be further studied.

Acknowledgement

Our work is supported by National Basic Research Program of China (973 Program) No.2011CB302302, and Tsinghua University Initiative Scientific Research Program.

REFERENCES

[1] http://www.tokutek.com/

[2] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, W. Lehner. SAP HANA Database - Data Management for Modern Business Applications. SIGMOD Record, 2011.

[3] R. Sears, R. Ramakrishnan. bLSM: A General Purpose Log Structured Merge Tree. In SIGMOD,2012.

[4] Bayer, R.; McCreight, E. (1972), "Organization and Maintenance of Large Ordered Indexes", Acta Informatica 1 (3): 173–189.

[5] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache Oblivious algorithms. In Proc. 40th Annual Symp. On Foundations of Computer Science (FOCS), pages 285-297, New York, Oct. 1999.

[6] M. A. Bender, M. Farach-Colton, J. T. Fineman, Y. R. Fogel, B. C. Kuszmaul, and J. Nelson. Cache-oblivious streaming B-trees. In SPAA, 2007.

[7] C. Jermaine, E. Omiecinski, and W. G. Yee. The partitioned exponential file for database storage management. The VLDB Journal, 16(4), 2007.

[8] R. Sears, M. Callaghan, Eric Brewer. Rose: Compressed, log-structured replication. In PVLDB, 2008.

[9] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (LSM-tree). Acta Informatica,33(4):351-385, 1996.

[10] M. A. Bender. Performance of Fractal-Tree Database. Technical Report, Tokuteck,2009

[11] P. William. Skip lists: a probabilistic alternative to balanced tree. Communications of the ACM 33 (6): 668–676. 1990.

[12] X. Sun, R. Wang, B. Salzberg, and C. Zou. Online B-tree merging. In SIGMOD, 2005.

[13] T. Lilja, R. Saikkonen, S. Sippu, E. Soisalon-Soininen. Online bulk deletion. In ICDE, 2007.

[14] R.J. Santos, J. Bernadino, M. Vieira. Leveraging 24/7 Availability and Performance for Distributed and Real-Time Data Warehouses. In COMPSAC, 2012