An Efficient Bulk Loading Approach of Secondary Index in Distributed Log-Structured Data Stores

Yanchao Zhu[1], Zhao Zhang[1,2(✉)], Peng Cai[1], Weining Qian[1], and Aoying Zhou[1]

[1] School of Data Science and Engineering, East China Normal University, Shanghai, China
yczhu@stu.ecnu.edu.cn, {zhzhang,pcai,wnqian,ayzhou}@sei.ecnu.edu.cn

[2] School of Computer Science and Software Engineering, East China Normal University, Shanghai, China

Abstract. How to improve reading performance of Log-Structured-Merge (LSM)-tree gains much attention recently. Meanwhile, constructing secondary index for LSM data stores is a popular solution. And bulk loading of secondary index is inevitable when a new application is developed on an existing LSM data stores. However, to the best of our knowledge there are few studies on research of bulk loading of secondary index in distributed LSM-tree. In this paper, we study the performance improvement of bulk loading of secondary index in *distributed* LSM-tree data stores. We propose an efficient bulk loading approach of secondary index in Log-Structured Data Stores. Firstly, we design secondary index structure based on distributed LSM-tree to guarantee the scalability and consistency of secondary index. Secondly, we propose an efficient framework to handle bulk loading of secondary index in a distributed environment, which can provide a good load balancing for query processing by using *equal-depth histogram* to capture data distribution. Analysis of theoretical and experimental results on standard benchmark illustrate the efficacy of the proposed methods in a distributed environment.

Keywords: Secondary index · Distributed bulk loading · Load balancing

1 Introduction

Recently, NoSQL databases are becoming more and more popular to support scale-out applications, such as BigTable [11], LevelDB [3] and Hbase [1], etc. Most NoSQL storage engines are implemented by *distributed LSM-tree* [13], which is a tree-like data structure that has high performance in write-intensive workloads.

For distributed LSM-trees, fast read operations are challenging because the data is partitioned by the primary key. Each partition is distributed to different nodes in a distributed environment. Furthermore, in order to get consistent query results, the corresponding relation tuples will be merged from disk stores and in-memory stores [13]. The system must scan the whole data set distributed to different nodes in order to answer queries on non-primary key attributes.

Building a secondary index is a popular solution to decrease response time of the queries on non-primary key attributes. For an empty LSM-tree data store, several methods have been proposed to build secondary indices efficiently. For example, Tan et al. [14] introduce the *Diff-index* structure to support index maintenance schemes. However, constructing a secondary index on an existing distributed LSM data store is very inefficient if traditional insertion operator is adopted. Unfortunately, bulk loading of a secondary index is inevitable when a new application is developed on a given LSM-tree data store.

In this study, we address the issues of constructing secondary index for a given LSM-tree data set, i.e. bulk loading of the secondary index on distributed LSM-tree data stores. The major difficulties include (1) the data is distributed to several nodes; and (2) the data store in the memory and in the disk at the same time. If each node maintains its own local index, high selective queries will be inefficient.

To construct the secondary index on an existing LSM-tree data store, the key task is to do efficient global sorting on search keys of the secondary index for bulk loading. An approach has three stages, including local index construction, index partition division and global index construction. Firstly, each data node builds its own local index and sends statistic information of search keys to the coordinator. Secondly, the coordinator generates multiple uniform ranges on search keys based on statistic information from data nodes, and sends the ranges to corresponding data nodes. Finally, data nodes receive ranges from the coordinator to create index partitions by shuffling data. Particularly, we propose *equal-depth histograms* to capture data distributions on search keys in Stage 1.

We have made the following contributions in this paper.

- We design a global sorting approach for bulk loading of secondary index on distributed LSM-tree.
- We employ *sampling equal-depth histogram* to capture data distribution.
- We integrate the bulk loading algorithm into our distributed database CEDAR [2] and measure the system performance using a standard benchmark. Experimental results show the efficacy of the proposed methods.

The rest of paper is organized as follows: Sect. 2 introduces background knowledge related to our work. Section 3 gives an overview of index construction. Discussion of details of bulk loading is in Sect. 4. We then provide an evaluation of the approach in Sect. 5. Finally, we describe related work in Sect. 6, and conclude our paper in Sect. 7.

2 Background

We begin with an overview of the LSM-tree. An implementation of LSM-tree is shown in Sect. 2.1. And the index structure based on the LSM-tree in distributed systems is introduced in Sect. 2.2.

2.1 System Model

The LSM-tree is a data structure that is optimized for frequent updates. It comprises a tree-like in-memory store and several tree-like immutable disk stores. Writing into LSM-tree equals to an insertion into the in-memory store. When the size of the in-memory store reaches a threshold, its content will be flushed to the disk. Multiple disk stores will be generated after several data merge processes. Thus read operations need to merge disk stores with the in-memory store to get consistent data, which greatly affects the performance of query. In order to improve the performance of query processing and to save disk space, multiple small disk stores can be merged into a large disk store periodically.

A typical implementation of a distributed storage system employing the LSM-tree is shown in Fig. 1. A table in the system is divided into partitions by a continuous primary key. Each partition is denoted by a range of primary keys $[start\_key, end\_key)$. Partitions are stored on the SSD on data nodes in a replicated way. The system meta data is stored on a node called the coordinator. Modifications of the database are stored in the memory of a node called the transaction node. When the size of in-memory store reaches a threshold, a merge process is launched. A new in-memory store will accept modifications and the old in-memory store will be merged with disk stores on data nodes, applying changes of data to the disk. After the merge process, new partitions are in service and old partitions are deleted. Read operations will first ask the coordinator for data location and then merge the corresponding data from disk stores and the in-memory store for response.

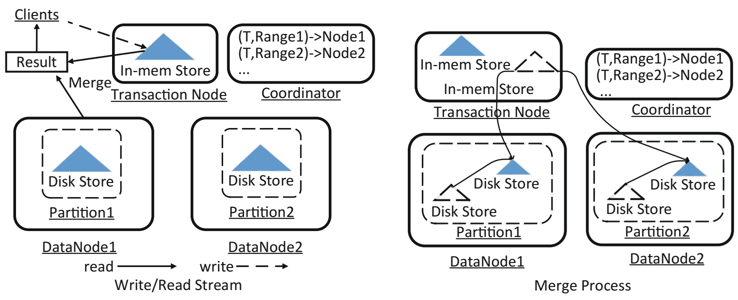
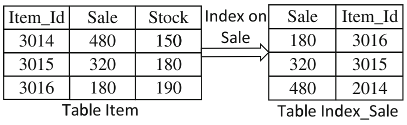


Fig. 1. Implementition of LSM-tree

2.2 Index Structure

A common structure of an index is an index table [15], which is partitioned and stored on a cluster of nodes as an ordinary one. A record in the index table is combination of an index column (*search_key*) and a primary key column of the data table, like as (*search_key*, *primary_key*). Figure 2 shows an example of the index. The primary key of the item table is column "Item_Id". If we create an index on the column "Sale", the schema of index table is shown in Fig. 2. A query is executed by accessing the index table to get the primary key of the data table, and then getting results from the data table according to the aforementioned primary key. Certainly, we also permit users to build the covering index, which is an index that contains all, and possibly more, the columns that you need for your query.

We organize the secondary index as the ordinary data table for three reasons. Firstly, modifications of indices are not subject to the CAP theorem [10] since modifications of the data and the index table are on the same node, which is important to a distributed write-intensive system. Secondly, bulk loading of indices can be done without blocking transactions since modifications of indices and data tables are stored in the in-memory store. Thirdly, the management of index data can reuse components for the data table, such as load balancing, scalability and high availability, etc.



Item_Id	Sale	Stock
3014	480	150
3015	320	180
3016	180	190

Table Item

Index on Sale

Sale	Item_Id
180	3016
320	3015
480	2014

Table Index_Sale

Fig. 2. Index structure

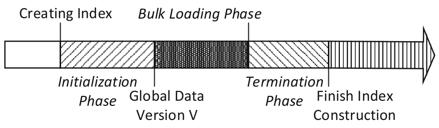


Fig. 3. Index construction process

3 Overview of Index Construction

In this section, a general process of secondary index construction will be described. The procedure of index construction has three phases: (1) Initialization phase, preparing for the start of the index construction. (2) Bulk loading phase, creating uniform index partition. (3) Index termination phase, implementing the replication of index partitions. The Fig. 3 shows the more details on above three phases. The more explanations on process of index construction will be described as follows.

3.1 Initialization Phase

When receiving the command of creating index, the system enters the initialization phase. In this phase, the system waits for a time point when the system reaches a global consistent data version V. After the time point, modifications of indices will be maintained in the in-memory store. Since modifications of indices and data tables are maintained in the in-memory store, constructing indices on data version V will guarantee the consistency of indices and data tables without blocking transactions. As is mentioned in Sect. 2, after data merge processing, data in the old in-memory store is merged with disk stores, modifications of system are maintained in the new in-memory store. Hence, the completion of the merge processing means that initialization phase finished.

3.2 Bulk Loading Phase

The system will complete the construction of index partitions in this phase. Furthermore, the construction of the index is divided into three stages: local index construction, balanced index partition division and global index construction. In the first stage, each data node builds its own local index and sends statistic information of search key to the coordinator. In the second stage, the coordinator generates multiple uniform ranges on search keys based on statistic information from data nodes, and sends the ranges to corresponding data nodes. In the third stage, data nodes receive ranges from the coordinator and create index partitions by shuffling data. Algorithm 1 is the main program. Algorithm 1 calls local index construction routine (Algorithm 2), index partition division routine (Algorithm 3) and global index construction (Algorithm 4) routine in the proper order.

Local index construction. The first key task of bulk loading is efficient global sorting on search keys of the secondary index. On the one hand, construction of local index can reduce communication overhead when global sorting on search keys is executed. On the other hand, the statistic information denoted by a equi-depth histogram collected on the local index can make partition division more uniform, which can guarantee good balancing for query processing.

Balanced index partition division. If the system wants to get uniform index divisions, the coordinator must understand the distributions of search keys on the whole data set and the number of index partitions. So, statistic information received from data nodes is summarized at the coordinator node. And the the number of index partitions is computed. Finally, the coordinator uniformly divides the index ranges according to the number of index partitions and sends the ranges to corresponding data nodes.

Global index construction. The data nodes are executors of global index construction. The data node communicates with other data nodes according to index ranges received from the coordinator to create index partitions in the global search key order.

3.3 Index Termination

After the bulk loading phase, the coordinator will schedule the task for replication of the index for high availability of the index. The replication mechanism of index table is the same to the original table since an index is organized as an ordinary table. Index construction is completely finished after index replication. And then the index is available for query.

4 Bulk Loading Process

The bulk loading phase has three stages, including local index construction, index partition division and global index construction. Firstly, each data node builds its own local index and sends statistic information of search keys to the coordinator. Secondly, the coordinator generates multiple uniform ranges on search keys based on statistic information from data nodes, and sends the ranges to corresponding data nodes. Finally, data nodes receive ranges from the coordinator create index partitions by shuffling data. Note that Algorithm 1 is the main program. Algorithm 1 calls local index construction routine (Algorithm 2), index partition division routine (Algorithm 3) and global index construction (Algorithm 4) routine in proper order.

Algorithm 1 illustrates the framework of the bulk loading approach in distributed LSM-tree data stores. We assume we will construct a secondary index on column search_key of table T. And partitions P of table T are distributed to multiple data nodes. So, all data nodes contained $p \in P$ run the function RunLocalIndex(p, interval, serach_key, storing) (Algorithm 2) to build local index and report the histogram H containing search_key distribution information to the coordinator (at line 6). Afterwards, the coordinator runs RunPartitionDivision(H,N) (Algorithm 3) to get balanced index data partitions based on histogram H and the number of partitions of index data (at line 9), and assigns partition ranges to appropriate data nodes (at line 10). Finally, the data nodes receive range information from the coordinator and run RunGlobalIndex(L) (Algorithm 4) to build the global index based on index partition range list L (at line 12).

4.1 Local Index Construction

Local index construction starts after the initialization phase. Algorithm 2 illustrates the process of local index construction. Note that Algorithm 2 must run at data nodes. Each data node scans the original data table partitions located in itself in order to construct index entries (at lines 6–14). If we need to build the covering index, i.e. storing columns are not NULL, an index entry needs to contain storing columns besides the search key and primary key of the original table (at line 8). Otherwise, an index entry only contains the search key and primary key (at line 11). And an equi-depth histogram H_i is adopted to capture data distributions on search keys, where each bucket is defined by an interval which is left-closed and right-open, i.e. $[start\_key, end\_key)$ (at lines 15–26). And *interval* represents the depth of the bucket h. The size of depth of bucket can be adjusted. Smaller bucket depth means more accurate information of data distributions, but it also needs more space to store more statics information. Finally, the local index records are written to the disk (at line 25).

Algorithm 1. Index Bulk Loading

```
1  Let $P$ denote partition set of table $T$ which needs to construct index;
2  Let $h_i$ and $H$ denote a bucket and an array of equi-depth histogram
     respectively;
3  Let $R$ denote an array of range /* construct local index                 */
4  foreach partition $p \in P$ in all data nodes do
5      $h_i \leftarrow$ RunLocalIndex($p$, $depth$) ;
6      $H.add(h_i)$;
7  end
8  report $H$ to Coordinator;
   /* divide index partition range                                        */
9  $R \leftarrow$ RunPartitionDivision($H$, $N$) ;
10 Coordinator sends the partition range $R[i]$ to all data node with respect to $R[i]$ ;
   /* construct index partition                                           */
11 foreach datanode correlated to $R[i]$ do
12     $index\_partition \leftarrow$ RunGlobalIndex($R[i]$);
13 end
```

Example 1. To help explain this process, we refer to Fig. 4 as our running example. Table Item has 16 records and contains three partitions, i.e., partition1, partition2 and partition3 on DataNode1, DataNode2. Local index construction in Fig. 4 describes the process local index construction. Each record in a partition is mapped to an index record and index records are sorted by the primary key of the index table. In our example, the primary key of the index table is $(Sale, Item\_Id)$. After local index construction, three *local_indexes* $local\_index1$, $local\_index2$, $local\_index3$ are constructed. The equal-depth histogram of partition1 is shown in Fig. 5. The depth of bucket is 2. Thus three buckets are sampled. See $<(101, 3001), (201, 3002)>$, $<(201, 3002), (400, 3004)>$ and $<(400, 3004), (600, 3006)>$ in Fig. 5.

4.2 Balanced Index Partition Division

After local index construction, the coordinator will receive equal-depth histograms of all *local_index* and then divide the index data into multiple uniform partitions. It is very important to decide the number of partitions and the division strategy of the index data. Subsequently, more details will be explored.

Generally, the larger size of partition means less efficient for high selective queries, while the smaller size of partition means more space for meta data in a distributed database. Thus, we have to set an appropriate size of partition for the index data. Considering a table T$(col_1, col_2...col_n)$ with an index I on col_j. Let $size(T)$ and M denote the size of T and the maximal limitation of partition size of T respectively. In fact, the partition number P of T can be defined by Eq. (1). Similarly, the partition number of the index data on table T denoted by P' can be defined by Eq. (2). The relationship between the number

Algorithm 2. RunLocalIndex

```
Function RunLocalIndex($P$, $search\_key$, $storing$) /* local index construction */
    Let $r$ and $r'$ denote a record of data table and index table respectively ;
    Let $P'$ denote the set of index records sorted by search_key ;
    Let $H$ and $h[j]$ denote a equi-depth histogram and a bucket respectively ;
    int $i, j \leftarrow 0$; $P' \leftarrow \emptyset$; $Flag \leftarrow TRUE$ ;
    for $each\ r \in P$ do
        if $storing\ is\ NULL$ then
            $r' \leftarrow (r.search\_key, r.primary\_key, storing)$ ;
        end
        else
            $r' \leftarrow (r.search\_key, r.primary\_key)$;
        end
        $P' \leftarrow P'$ inserted $r'$ based on the order of search-key ;
    end
    while $i <> |P'|$ do
        i++ ;
        if $Flag$ then
            $h[j].start\_key \leftarrow r'[i-1].search\_key$ ; $Flag \leftarrow FALSE$ ;
        end
        if $i\ mod\ interval == 0$ then
            $h[j].end\_key \leftarrow r'[i].search\_key$ ;
            $Flag \leftarrow TRUE$; j++;
            $H \leftarrow H \cup h[j]$ ;
        end
        write $r'$ to local index partition on disk ;
    end
    return $H$
end
```

of index data partitions and the number of original data table partitions can be defined by Eq. (3). Furthermore, the index partition number can be calculated by Eq. (4) for fixed-length storage systems. However, we have to capture the size of index records during local index construction to calculate the partition number of the index by Eq. (2) for varying-length storage systems. In practice, we set the number of partitions of the index data same to the number of partitions of the original table data in order to reduce query response time.

$$P = \frac{size(T)}{M} \tag{1}$$

$$P' = \frac{size(I)}{M} \tag{2}$$

$$P' = P \times \left(\frac{size(I)}{size(T)}\right) \tag{3}$$

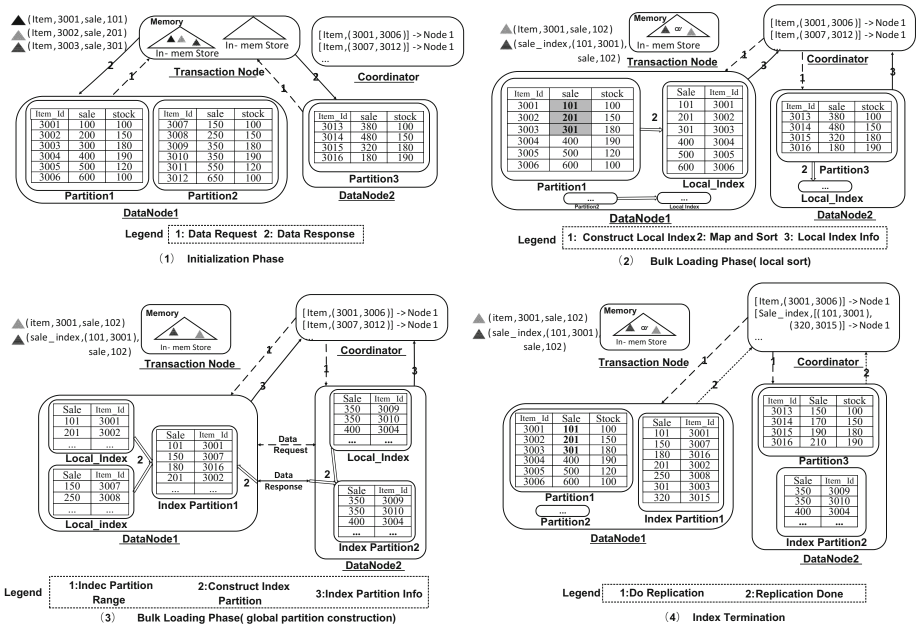

(1) Initialization Phase
(2) Bulk Loading Phase(local sort)
(3) Bulk Loading Phase(global partition construction)
(4) Index Termination
Transaction Node
Coordinator
DataNode1
DataNode2
Legend 1: Data Request 2: Data Response
Legend 1: Construct Local Index 2: Map and Sort 3: Local Index Info
Legend 1:Indec Partition Range 2:Construct Index Partition 3:Index Partition Info
Legend 1:Do Replication 2:Replication Done

Fig. 4. Index bulk loading

$$P' = P \times \left(\frac{size(search\_key, primary\_key)}{size(col_1, col_2 ... col_n)} \right) \tag{4}$$

Algorithm 3 illustrates the procedure where the coordinator divides the index data into multiple uniform partitions. So, Algorithm 3 must run at the coordinator node. First, the coordinator puts together a whole histogram H' with the histogram h_i received from each individual data node based on the order of bucket (at line 5). Afterwards, The coordinator will divide the H' into $\lceil (|H'| - 1)/N \lceil$ ranges, where $|H'|$ represents the number of buckets and N is the number of index data partitions (at line 6). Note that each index data partition is defined by an interval [start_key,end_key) which is left-closed and right-open. Furthermore, the range is continuous, such as $(MIN, index\_key_1]$, $(index\_key_1, index\_key_2]...(index\_key_p, MAX)$. After the division, the coordinator will allocate ranges to data nodes and data nodes will construct corresponding partitions, which are described in the main program (Algorithm 1).

The key point of Algorithm 3 is to guarantee each data node maintains almost same size of the index, aiming at providing a good load balancing for query processing. The coordinator adopts two strategies to assign index ranges to data nodes: (1) **overlap priority**, it means that the coordinator assigns the index range to the data node which has maximal overlaps with the range. (2) **Load priority**, it means that the coordinator assigns the index range to the data node which has the lightest load. The overlap priority can reduce communication overhead, and the load priority can provide a quicker response for queries.

Algorithm 3. RunPartitionDivision

```
Function RunPartitionDivision($H$, $N$)
    Let $H'$ denote a equi-depth histogram which bucket is sorted by its start_key ;
    Let $R$ denote an array of range defined by [start_key, end_key) ;
    Let $h[i]$ be the i-th bucket of $H'$ ;
    Load all buckets of $H$ based on the order of bucket.startkey into $H'$ ;
    $n \leftarrow \lceil (|H'| - 1)/N \rceil$ ;
    $R \leftarrow \emptyset$; $i, j \leftarrow 0$; $Flag \leftarrow FALSE$ ;
    $R[1].start\_key \leftarrow h[1].start\_key$ ;
    while $i <> |H'|$ do
        i++ ;
        if $Flag$ then
            $R[j].start\_key \leftarrow h[i-1].end\_key$; $Flag \leftarrow FALSE$ ;
        end
        if $i \ mod \ n == 0 || i == |H'|$ then
            $R[j].end\_key \leftarrow h[i].end\_key$ ;
            $Flag \leftarrow TRUE$ ; j++ ;
        end
    end
    return R;
end
```

The end users can adopt different policies according to application scenarios. Certainly, the communication overhead cannot be totally avoided even when the overlap priority is adopted. Fortunately, communication overhead is generated in the offline stage. It has no effect on the response of query processing.

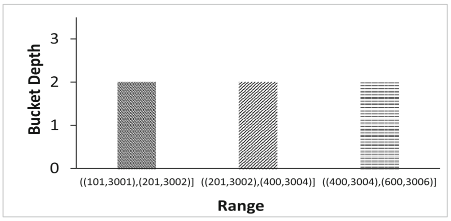


Fig. 5. Equal-depth histogram

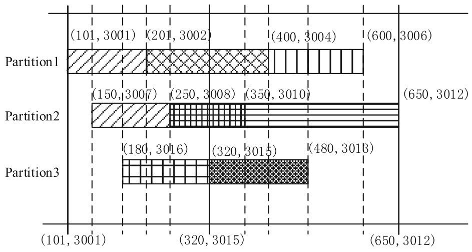


Fig. 6. Partition division

Example 2. Take the example in Fig. 4. After local index construction, the coordinator receives three histograms and will divide ranges of index partitions. The process of division is shown in Fig. 6. Since the partition number of the data table is 3. For fixed-length storages, the partition number of the index is 2. Thus, ranges of partitions are: (MIN,(320,3015)] and ((320,3015),MAX). The first partition includes 7 records and the second partition includes 9 records.

4.3 Global Index Construction

Global index construction starts after the division of index partitions. During global index construction, a data node may receive two types of information. One is the control information from the coordinator node. Another is the data information from other data nodes. The former is the index partition construction command with a list of index partition ranges, which means the data node will maintain the index data corresponding to ranges received from the coordinator. The latter is index records loaded from other data nodes in the given range. Certainly, the index records are sorted by the same search key. After getting all the index records in the given range, the sorted index data will be flushed to the disk serving as an index table partition.

Algorithm 4 illustrates the procedure of global index construction. It is worth noting that Algorithm 4 must run at all data nodes received construction index messages from the coordinator. Firstly, the algorithm prepares the data for each range according to a range list L received from the coordinator (at line 4, line 8). Meanwhile, it may receive requests from other data nodes for index records and it will response to the these requests. After getting all the records of P, it will sort these records and then flush them to disk as an index partition P (at line 5, line 9). In practice, the procedure of constructing index partitions can be implemented by multi-thread technique, i.e. each range can be assigned a thread.

Algorithm 4. Global Index Construction

```
1  Function RunGloalIndex($L$)
2      Let $P$ denote a index partition ;
3      for each $l_i \in L$ do
4          if the current data node contains all index data in range $l_i$ then
5              Flush index data to disk to construct index partition $P$ ;
6          end
7          else
8              shuffle and sort index data which is not located in current nodes
                 with other data nodes correlated to $l_i$ ;
9              Flush index data to disk to construct index partition $P$ ;
10         end
11     end
12 end
```

Example 3. Take the example of global index construction in Fig. 4. After index partition division, DataNode1 receives a partition range $(MIN, (320, 3015)]$ and DataNode2 receives a partition range $((320, 3015), MAX]$. After getting all the records of a range, index records are flushed to the disk serving as an index partition, such as index partition1 and index partition2 in this example.

5 Experimental Evaluation

5.1 Experimental Setup

For experimental analysis, we integrate the approach with our system CEDAR based on OCEANBASE 0.4.2 [4], which is a scalable open source RDBMS developed by Alibaba. It consists of four modules: master server (Coordinator), update server (TransactionNode), baseline data server (DataNode) and data merge server (MergeServer). Now we describe our experimental setup and give a brief overview of the database employment we have used for evaluation.

Cluster platform: We run the experiments on a cluster of 9 machines for most of our experiments except scalability. Each machine is equipped with an Intel(R) Xeon(R) CPU E5-2620@2.00 GHz (a total of 12 physical cores), 96 GB RAM and 3TB Raid5 while running CentOS version 6.5. All machines are connected by a gigabit Ethernet switch.

Database deployment: The database is configured with four MergeServe and four DataNode and each of them is deployed on a single machine in the cluster. The cluster is also configured with a Coordinator and a TransactionNode and they are deployed on a same machine. For better performance, we cache the index and the data table in memory. We choose the load priority policy for guarantee load balancing of index.

Benchmark: Sysbench [8] is a popular open source benchmark to test open source DBMSs. We extend Sysbench [8] by adding an item table in which each row has a unique *item_id* as the primary key and 3 columns. Among them, *item_price* is the column to index. The rest 2 columns are *item_desc* fed with 100 Byte long random byte and *item_title* fed with 92 Byte long random byte, Altogether each row is approximately of 200 Byte in size. We change the workload by varying client threads. For tests of scalability, we vary DataNode from 1 to 4. We test the performance of the index with a query with a predicate on the item price attribute: *Q1: SELECT item_id, item_desc FROM item WHERE item_price = "?"*. By adjusting the size of return size, we can define the selectivity of the test queries.

5.2 Index Construction

We first evaluate the efficiency of index construction. We generate 50 million to 125 million and load data to the system. We compare our approach with a common approach to construct index table denoted as "Read-Insert" where the index table is constructed by first reading from the data table and then inserting into the index table. We launch 100 threads for the "Read-Insert" approach and 10 threads for our bulk loading approach on each DataNode. As is shown in Fig. 7, our approach constructs index much faster than the common approach, for there is less network overhead and no random disk I/O in our approach.

We then test the index partition strategy. We vary the distribution of data on the index key and test three types of distribution: Uniform distribution, Gaussian

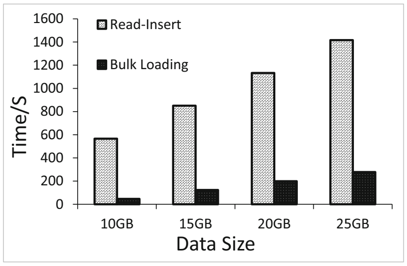


Fig. 7. Index construction time

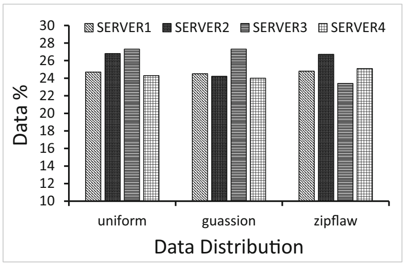


Fig. 8. Partition distribution

distribution, Zipf distribution (Zipf factor = 2). We generate 50 million records for each distribution and create indices on *item_price* for them respectively. After index construction, data distributions of indices are shown in Fig. 8. As we can see, benefiting from balanced index partition strategy, the index data on each node relatively equals to each other.

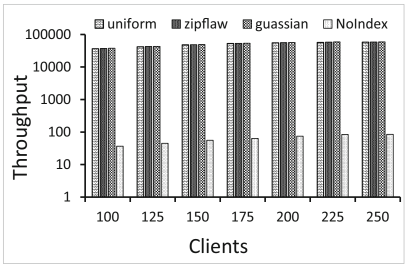


Fig. 9. Query throughput

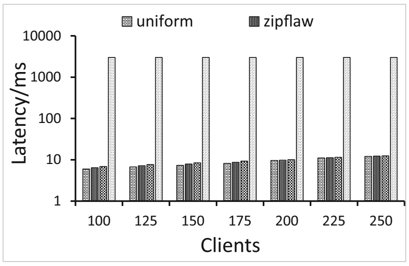


Fig. 10. Query latency

5.3 Query Performance

In this test, we first study the query performance of indices with different data distribution properties. Since the index key has different distribution properties, the result size of test queries may differ, however, since the query distribution is uniform, average result size for queries is relatively the same under different data distribution properties. Figures 9 and 10 demonstrate that query throughput and latency of the system under different data distribution properties are almost same, for our approach captures the different properties of data and guarantees the load balancing of index.

We then test the latency of queries with different selectivities, using concurrent client threads from 50 to 275. We vary selectivity from 0.0001% (50 rows in result) to 0.001% (500 rows in result). As is shown in Figs. 11 and 12, for queries with higher selectivities, the system need less time for query processing

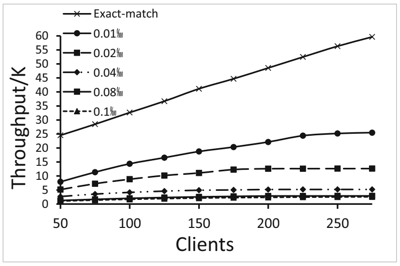


Fig. 11. Query throughput (selectivity)

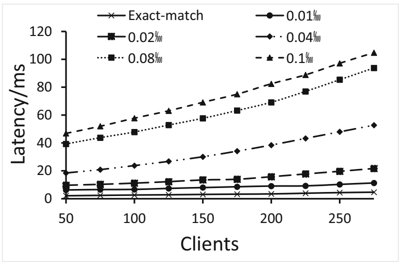


Fig. 12. Query latency (selectivity)

and network communication, thus the system has better latency and throughput with higher selectivity. Figure 11 also shows that the latency of queries with high selectivity is less affected by the workload than that of queries with low selectivity. Queries with lower selectivity requires more system resources such as CPU and Network than queries with high selectivity. As the workload increases, more queries will compete for CPU. Thus, the throughput keeps unchanged and the latency of queries increases.

5.4 Scalability

In this test, we test the scalability of index construction and query performance with different selectivities. We vary the size of data node from 1 to 4. For index construction, we measure the construction time for index. For query performance, we measure the latency and throughput of exact match queries and queries with different selectivities.

As we can see in Fig. 13, time for index construction reduces with the increment of the number of data nodes, this is due to the fact that we allocate index partitions to data nodes uniformly. However, time for index construction does not reduce linearly because adding nodes may cause more network communication.

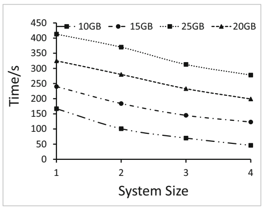


Fig. 13. Index construction

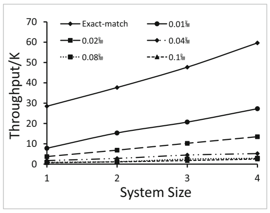


Fig. 14. Query throughput

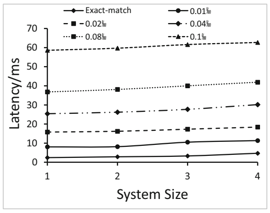


Fig. 15. Query latency

As can be seen from Figs. 14 and 15, the system scales well with nearly flat query latency when the size of data node increases due to the fact that the index is organized as a normal table and can take advantage of the load balancing of the system. In this test, the workload submitted to the system is proportional to the data node size. As we can see, by adding data node, more workload can be handled. The latency for queries with a specific selectivity remains essentially unchanged with different number of data nodes.

6 Related Work

There are two types of secondary index models on distributed data stores: local index models and global index models. In local index models, the index is built on each node which indexes local data. Global indices support high selectivity queries better. Huawei's Hindex [6] realizes the local index on the LSM-based storage system Hbase [1], however, load balancing of index is not supported. Diff-index (Differentiated Index) [14] realizes the global index on Hbase which supports index creation on an empty table.

There have been several existing bulk loading approaches of secondary indices in distributed log-structed data stores. Phoenix [5] use map-reduce [12] to construct distributed index, modifications of index are maintained in memory. However, when the data set is huge, it needs to create index in an asynchronous way by an external tool. AsterixDB [9] presents a bulk loading approach for converting existing index structures to LSM-based index structures. The approach is for AsterixDB and employs multiple data structures realizing bulk loading of indices, which means it is hard to do load balancing of indices. Another approach is by external storages such as solr [7]. Indices are stored in an external system. Thus, the construction of indices needs to read from the database and insert into the external system.

7 Conclusion

We introduce a new bulk loading approach of secondary index in distributed log-structed data stores. The approach supports efficient bulk loading of index by taking rational use of resources in a cluster. Whats more, by using equal-depth histogram, load balancing of index is guaranteed. We perform an extensive evaluation of our approach on a LSM-based distributed system. The results from our experiments show that our approach take rational use of cluster for bulk loading of index and guarantee the load balancing of index. In addition, since an index is organized as a normal table and is integrated in the load balancing of the system, the index is scalable and adding nodes to the system will improve the performance of the index.

Acknowledgements. This work is partially supported by National High-tech R&D Program (863 Program) under grant number 2015AA015307, National Science Foundation of China under grant numbers 61402180, 61432006 and 61672232, Natural Science Foundation of Shanghai under grant numbers 14ZR1412600, and Guangxi Key Laboratory of Trusted Software (kx201602). The corresponding author is Zhao Zhang.

References

1. Apache HBase website. http://hbase.apache.org/
2. CDEAR website. https://github.com/daseECNU/Cedar/
3. LevelDB website. http://leveldb.org/
4. OceanBase website. https://github.com/alibaba/oceanbase/
5. PHOENIX website. http://phoenix.apache.org/
6. Secondary Index for HBase. https://github.com/Huawei-Hadoop/hindex
7. SOLR website. http://lucene.apache.org/solr/
8. Sysbench website. http://dev.mysql.com/downloads/benchmarks.html
9. Alsubaiee, S., Asterixdb, A., et al.: A scalable, open source bdms. Proc. VLDB Endowment **7**(14), 1905–1916 (2014)
10. Brewer, E.: Pushing the cap: strategies for consistency and availability. Computer **45**(2), 23–29 (2012)
11. Chang, F., Dean, J., Bigtable, G., et al.: A distributed storage system for structured data. ACM Trans. Comput. Syst. (TOCS) **26**(2), 4 (2008)
12. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. In: Conference on Symposium on Opearting Systems Design & Implementation, pp. 107–113 (2004)
13. ONeil, P., Cheng, E., Gawlick, D., O'Neil, E.: The log-structured merge-tree (lsm-tree). Acta Informatica **33**(4), 351–385 (1996)
14. Tan, W., Tata, S., Tang, Y., Fong, L.L.: Diff-index: differentiated index in distributed log-structured data stores. In: EDBT, pp. 700–711 (2014)
15. Zou, Y., Liu, J., Wang, S., Zha, L., Xu, Z.: CCIndex: a complemental clustering index on distributed ordered tables for multi-dimensional range queries. In: Ding, C., Shao, Z., Zheng, R. (eds.) NPC 2010. LNCS, vol. 6289, pp. 247–261. Springer, Heidelberg (2010). doi:10.1007/978-3-642-15672-4_22