Front. Comput. Sci., 2020, 14(3): 143604
https://doi.org/10.1007/s11704-019-8234-x

RESEARCH ARTICLE

Partition pruning for range query on distributed log-structured merge-tree

Chenchen HUANG, Huiqi HU (✉), Xing WEI, Weining QIAN, Aoying ZHOU

School of Data Science and Engineering, East China Normal University, Shanghai 200062, China

Abstract Log-structured merge tree (LSM-tree) is adopted by many distributed storage systems. It contains a Memtable and a number of SSTables. The Memtable is an in-memory structure and the SSTable is a disk-based structure. Data records are horizontally partitioned over the primary key and stored in different SSTables. Data writes on records are first served by the Memtable and then compacted to SSTables periodically. Although this design optimizes data writes by avoiding random disk writes, it is unfriendly to read request since the results should be retrieved and merged from both Memtable and SSTables. In particular, when the Memtable and SSTables are distributed on different nodes, it incurs expensive costs to serve range queries. A range query on non-primary key columns has to scan all partitions, which generates many network and I/O expenses. In this paper, we propose a partition pruning strategy to save cost for range queries. A statistics cache is designed to determine whether a partition contains the desired data or not, which enables read requests to avoid scanning useless partitions. As records can be updated in Memtable freely, to prevent incorrect filtering, a version-based cache synchronization strategy is proposed to ensure the queries to obtain the latest data state. We implement the proposed method in an open source distributed database and conduct comprehensive experiments. Experimental results reveal that the performance of range queries increased 30% ~ 40% with our partition pruning technique.

Keywords LSM-tree, table partitioning, statistics, cache consistency

Received July 5, 2018; accepted March 5, 2019

E-mail: hqhu@dase.ecnu.edu.cn

1 Introduction

With the rapid growth of data generated and accessed in distributed systems, the read and write performances of storage infrastructures are highly demanded in recent years. Log-structured merge tree (LSM-tree) [1] is designed as a write-optimized storage structure, which is widely used in distributed systems such as BigTable [2] and Cassandra [3]. Following the notation in [2], a typical LSM-tree organizes data records in two main components: a Memtable and an SSTable. The Memtable is a memory-based structure, which is used to perform data writes. The SSTable is a disk-based structure, which offers a large capacity of storage. Distributed LSM-tree can increase its capacity through expansion of SSTables. Then large data tables can be horizontal partitioned over its primary keys and stored in SSTables on different nodes. LSM-tree optimizes data writes by avoiding random disk writes. Data writes on records are first written into Memtable and then compacted into corresponding SSTables periodically. However, the structure is unfriendly to read operations. To access a single record, it should retrieve the Memtable and an SSTable since the Memtable may have recent updates of the record. The performance is more worse for range queries. A range query on a non-primary key column has to go through all the SSTables and Memtable, even if the dedicated record only exists in one structure, which generates many unnecessary network and I/O costs. As a result, optimizing read performance is extremely important to distributed LSM-tree.

There are some works [4–6] aiming at improving the read performance for LSM-tree. For instances, Sears and Ramakrishnan [4] attempted to avoid accessing useless partitions with the bloom filter technology [7]. Ahmad and Kemme [5] rely on major *Compaction* to merge multiple partitions together and reduce the number of visited partitions. However, these techniques have two limitations: (i) they are not suitable for range query since they cannot verify whether a range of records exist or not. In terms of range query over LSM-tree, it becomes less efficient due to the distributed partitions on the primary key. (ii) they only reduce the access on SSTables, but they can not prune the Memtable since it is always updated.

In this study, we target at improving the range queries for distributed LSM-tree system by partition pruning. The specific method is to make filter out the irrelevant SSTables and Memtable which do not intersect with the query range before scanning the data. In detail, there are two challenges: (i) how to determine whether an SSTable partition or Memtable partition contains the required range data without scanning the data? (ii) since the Memtable is changed dynamically, the data used to make partition pruning may be older than the latest data. As a result, using the determined result to execute read operations is hard to ensure the read consistency. To solve the above problems, we can gather the statistics of all the partitions in advance, then the p-nodes directly use the corresponding statistics to determine whether the desired data exist or not. Next, we need to design an update strategy for the statistics, which ensures that using the result determined by the statistics to scan data can guarantee read consistency. Our major contributions are summarized as follows.

(1) We propose a pruning mechanism and design statistics cache of SSTable and Memtable to prune the useless partition scanning for range queries.

(2) With respect to highly data updates, we propose a version-based cache synchronization strategy to ensure read consistency. It guarantees the corresponding data version of statistics got from cache satisfies the consistence of snapshot isolation [8], which also ensures that using the pruning result determined by these statistics to scan data reaches the same read consistency.

(3) We implement our partition pruning method on an open-source distributed database. Experiment results show that our proposed method outperforms existing approaches.

The rest of the paper is organized as follows. Section 2 introduces the architecture of the distributed LSM-tree that supports our partition pruning strategy. Section 3 describes the mechanism of partition pruning. Section 4 presents the implementation of partition pruning by using statistics cache. Section 5 proposes a version-based cache synchronization strategy to guarantee the data access achieves the consistence under snapshot isolation. Section 6 presents the experimental results and discusses their significance. Finally, we review related works in Section 7 and conclude in Section 8.

2 Architecture

In this section, we introduce the architecture of the distributed LSM-tree that supports our partition pruning strategy. As shown in Fig. 1, next we mainly from two aspects to describe the entire architecture: storage layer and query layer.

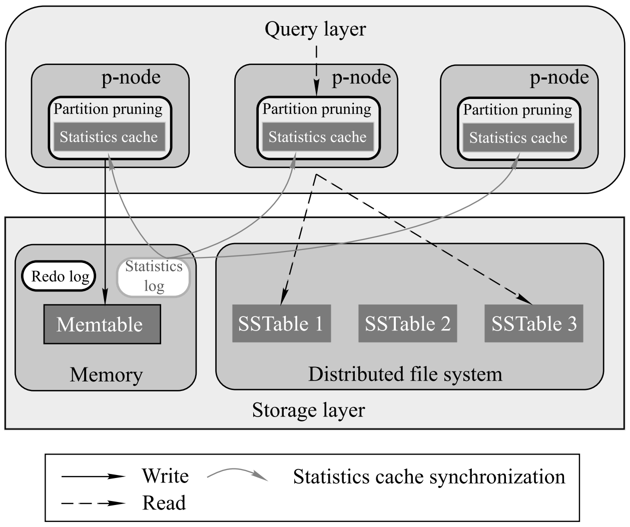


Fig. 1 Distributed LSM-tree supporting partition pruning

2.1 Storage layer

In the storage layer, the data of a table is partitioned and stored in a Memtable and several SSTables.

Memtable is an in-memory structure which services read and write requests. In the LSM-tree, data writes are only allowed to perform on the Memtable, it stores incremental data and resides in main memory to facilitate write performance.

SSTable is an on-disk and immutable structure where static data is stored in lexicographic order based on their primary key, it offers large storage capacity and services read requests only. LSM-tree can fit massive amounts of data through the expansion of nodes with SSTables. In the LSM-tree, records are first written into Memtable, and then are frozen and migrated to merge with the corresponding SSTables when the Memtable reaches a certain size.

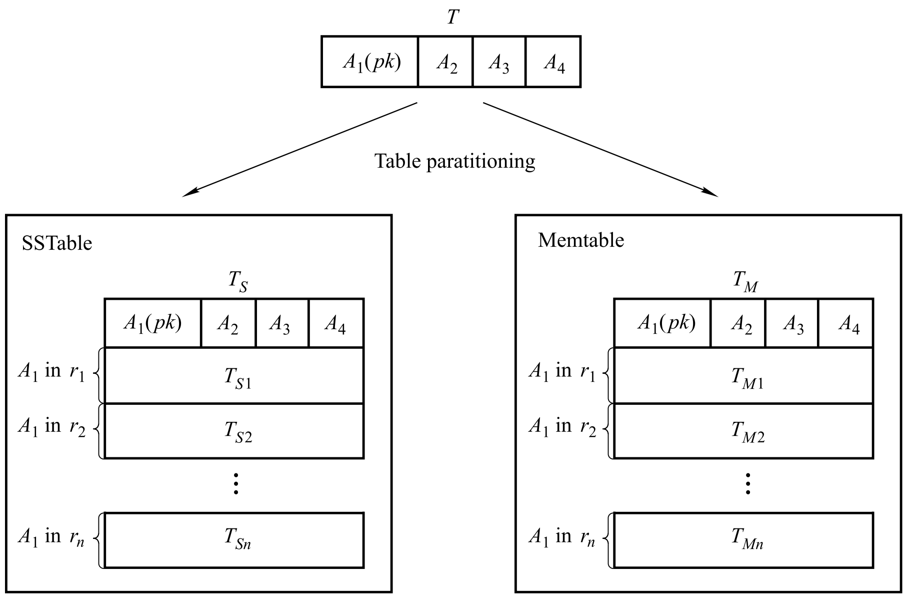


Fig. 2 Table partitioning on distributed LSM-tree

Table partitioning is a data organization scheme in which table data are divided into multiple data partitions on the values of one or more columns. These partitions can be distributed in different storage nodes or the same nodes. In this paper, we mainly talk about the table partitioning on distributed LSM-tree. Figure 2 presents a table T partitioned on its primary key A_1 on the distributed LSM-tree. In essence, it is a share-nothing partition, we can define it as follows:

Definition 1 (LSM-tree table partitioning) For a table T in distributed LSM-tree, we denote its static data and incremental data as T_S and T_M, which are stored in SSTable and Memtable respectively. Then, for its primary key whose rang is R, we can divide it into n disjoint sub ranges $r_1, r_2, \ldots, r_n$. Thus, T_S and T_M can be divided into n partitions $T_{S1}, T_{S2}, \ldots, T_{Sn}$ and $T_{M1}, T_{M2}, \ldots, T_{Mn}$ according to the n sub ranges $r_1, r_2, \ldots, r_n$.

2.2 Query layer

The query layer is mainly responsible for executing read and write requests, a node in the query layer interacts with the storage layer through network communication. For convenience, we simply call the processing node as p-node in the following. In this paper, we added a statistics cache on the p-node to implement our partition pruning strategy.

Write When a write operation arrives, it only performs on the Memtable. In order to avoid data loss, redo log is firstly constructed and then flushed into the durable device for recovery purpose. After the redo log has been flushed, its content is then written into the Memtable and published.

Read For a read request in LSM-tree, we have to retrieve the related incremental data and static data from memory and disks respectively and then merge them together. In Fig. 1, assume that SSTable 1, SSTable 2 and SSTable 3 store the static data of table T, and Memtable stores the corresponding incremental data. Then for a non-primary key range query for T, if there are no indexes on the non-primary attributes, we have to visit the Memtable and all SSTables to locate the required data, although the target values only exist in SSTable 2 and SSTable 3, which generates many useless disk accesses and network communications.

Statistics cache To avoid the unnecessary partition scanning in rang queries, we add a statistics cache on the p-node to determine whether a partition contains the desired data or not. As shown in Fig. 1, when a read operation arrives, it firstly use the statistics cache to filter the useless partitions, this process is called partition pruning. Then it only needs to visit SSTable 2 and SSTable 3, which avoid scanning Memtable and SSTable 1.

3 Partition pruning

In this section, we mainly present the mechanism of partition pruning on distributed LSM-tree.

For a range query Q, assume that its predicate conditions are only on the non-primary attributes of partitioned table T, then it requires invoking multiple remote calls to scan all the partitions of T. Intuitively, if a p-node is able to determine whether a partition contains the required range data or not, it can directly visit the dedicated structure without making other nonsense communications. This work aims at seeking efficient methods to prune useless partitions scanning for range query.

In practice, when a range query should scan the entire partitioned table T, the p-node firstly receives a scan request with the full rang R of primary key (used to partition T), then it divides R into n sub-ranges $r_1, r_2, \ldots, r_n$ by the rang of each partitions to pull data in parallel, instead of directly use R. For an arbitrary sub-range r_i, if we can determine whether the related SSTable partition T_{Si} and Memtable partition T_{Mi} should be scanned, we can determine for other sub-ranges in the same way. Therefore, we mainly focus on how to prune the useless scanning of related SSTable partition or Memtable partition for one specific sub-range. Before describing the partition pruning method, we first look at the different scan states for a sub-range.

Definition 2 (Scan state) Let T be a partitioned table, and r_i be the range of SSTable partition T_{Si} and Memtable partition T_{Mi}, then for a sub-range equals to ri, there are three different scan states:

(1) Only scan T_{Si}, the scan state can be denoted as Read SSTable (RS);

(2) Scan both of T_{Si} and T_{Mi}, we denote the scan state as Read SSTable and Memtable (RSM);

(3) Scan neither T_{Si} nor T_{Mi}, the scan state is defined as No Read (NR).

When a range query arrives, p-node first pulls data from all related SSTable and Memtable partitions, and then filters the data by predicate conditions and returns them to client. In the process, we can use the predicate conditions to prune unnecessary partitions before pulling data. Here, we denote the relation between the partition and the predicate conditions as follows:

Definition 3 (Relation between partition and predicate conditions) Let Q be a range query for the partitioned table T and $\mathcal{PC}$ be the set of its predicate conditions. For a non-empty partition T_i of T, if there exist data satisfy $\mathcal{PC}$, we define the relation between them as $T_i \widehat{\cap} \mathcal{PC} \neq \phi$; otherwise, $T_i \widehat{\cap} \mathcal{PC} = \phi$.

Usually, there is more than one predicate condition in a range query. In this case, we first determine the relation between the partition and every condition respectively, and then use the relations (and/or) between the conditions in the query to verify whether there exist the target values. For example, assume there are two predicate conditions pc_1, pc_2 and the relation between them is *and*. Then only when the partition has data that satisfy both pc_1 and pc_2, can we say that the target values exist. But if their relation is *or*, it only needs to satisfy one of the conditions.

Based on the above definitions, partition pruning is to make a p-node decide the scan state of a sub-range by identifying whether the related partitions satisfy the query predicate conditions. Thus, for a sub-range r_i, suppose that its corresponding Memtable partition and SSTable partition are T_{Mi} and T_{Si}, we can get the scan state by the following analysis process:

(1) if T_{Mi} is empty, it means that there is no update about r_i in Memtable, thus we should not scan T_{Mi}; if $T_{Si} \widehat{\cap} \mathcal{PC} \neq \phi$, T_{Si} should be scanned; the scan state is RS.

(2) if T_{Mi} is empty and $T_{Si} \widehat{\cap} \mathcal{PC} = \phi$, we should not scan both of T_{Mi} and T_{Si}, the scan state is NR.

(3) if T_{Mi} is not empty, $T_{Mi} \widehat{\cap} \mathcal{PC} = \phi$ and $T_{Si} \widehat{\cap} \mathcal{PC} \neq \phi$, we can not determine whether should scan T_{Mi}, because the merged result of T_{Mi} and T_{Si} could have some tuples satisfy $\mathcal{PC}$, thus we should scan both of them, the scan state is RSM.

(4) if T_{Mi} is not empty, $T_{Mi} \widehat{\cap} \mathcal{PC} = \phi$ and $T_{Si} \widehat{\cap} \mathcal{PC} = \phi$, the merge result of T_{Mi} and T_{Si} must not have tuple satisfy $\mathcal{PC}$, then we do not need to scan both of T_{Mi} and T_{Si}, the scan state is NR.

(5) if T_{Mi} is not empty and $T_{Mi} \widehat{\cap} \mathcal{PC} \neq \phi$, no matter the relation between T_{Si} and $\mathcal{PC}$, the merge result of T_{Mi} and T_{Si} must have tuples satisfy $\mathcal{PC}$, we should scan both of them, the scan state is RSM.

Combined with the above analysis process, Table 1 presents the scan states under different combination of three conditions ($T_{Mi} = \emptyset$, $T_{Mi} \widehat{\cap} \mathcal{PC} \neq \phi$, $T_{Si} \widehat{\cap} \mathcal{PC} \neq \phi$). The value of 0 indicates that the condition in this column is false, and 1 means true.

This section mainly talks about the schema of partition pruning. To implement it on a distributed LSM-tree, the major issue is how to confirm the relation between a partition and the predicate conditions. An intuitive solution is to gather some statistics of all the partitions in advance, and then each p-node uses the statistics to make a decision. Next section we will discuss what statistics are gathered to support partition pruning and how to maintain them.

Table 1   Scan state decision of a partition

$T_{Mi} = \emptyset$	$T_{Mi} \widehat{\cap} \mathcal{PC} \neq \phi$	$T_{Si} \widehat{\cap} \mathcal{PC} \neq \phi$	ScanState
1	0	1	RS
1	0	0	NR
1	1	0	Not possible
1	1	1	Not possible
0	0	1	RSM
0	0	0	NR
0	1	1	RSM
0	1	0	RSM

4   Statistics maintenance

In the process of implementing partition pruning, the first problem is how to determine the relation between a partition and the predicate conditions before scanning data, a straightforward method is to maintain the statistics of all the partitions, and then the p-node directly check the corresponding statistics to make a decision. In this section, we introduce the contents and gathering method of statistics. We cache all the Memtable partitions statistics and SSTable partition statistics in p-node to reduce the network load in range queries. Then we design an algorithm to present the partition pruning process based on statistics cache.

4.1   Statistics

Statistics type   For a range query, the predicate conditions can be roughly dived into two categories: (*i*) range conditions: such as >, <, *between* ... *and* ..., *etc*.; (*ii*) equal conditions: like =, *in*, *etc*. According to the kind of conditions, we can gather two types of statistics: min-max values and bloom filter.

- Min-max values: for a range condition, we can know the relation between a partition and it by determining whether there is intersection between the range of condition and the range of corresponding column. Here, the range of corresponding column can be obtained by gathering its min and max values.
- Bloom filter: for an equal condition, if its corresponding column is on the non-primary attribute, it needs to execute a range scan as well as range conditions. Then we can generate a bloom filter [7] using the data of its corresponding column to determine whether the target values exist. In this case, the determined results are not entirely accurate, because hash conflict may occur when the number of different values is large. But this not affects the correctness of query result, and bloom filter has used multiple hash functions to reduce the conflicts.

Statistics gathering   In the distributed LSM-tree, updates are first written into Memtable, and then are frozen and migrated to merge with the corresponding SSTables when the Memtable reaches a certain size. For convenience, we call the merging process as *Compaction*. On this basis, we respectively gather the statistics of SSTable and Memtable partitions as follows:

- SSTable partition statistics: for an SSTable partition, it may be changed after *Compaction*, so we can gather its statistics during every *Compaction*. All the SSTable partitions need to gather statistics at the time of the first *Compaction*. But during each of the following *Compaction*s, we can only gather statistics for the changed SSTable partitions to save cost. The update frequency is low because the interval between two *Compaction*s is always long.
- Memtable partition statistics: for the Memtable partition, it is changed by the data writes. When a write operation is executed on the Memtable, redo log are first prepared to avoid data loss caused by node failures, after the redo log has been flushed, its content is written into the Memtable and published. So we can gather statistics after the updates are written into Memtable. However, the frequency of the writes is very high, if we regather the statistics of entire Memtable partition after every change as well as SSTable partitions, the cost is very expensive. Thus, we can use the redo log to generate *statistics log* and then incrementally update the statistics, the specific process is described in Section 5.2.

4.2   Statistics cache

Considering the network traffic between p-node and SSTable or Memtable, we cache the statistics on the p-node.

In addition, hash table is chosen as the cache structure to further improve the speed of looking up statistics. Figure 3 shows the statistics cache of a partitioning table *Orders*. The table is divided into 3 SSTable partitions and 3 Memtable partitions by its primary key *order_id*, and the range of every partition is (min,500], (500,1000] and (1000,max]. As shown in Fig. 3(a) and Fig. 3(b), the SSTable and Memtable partition statistics are stored in two hash tables respectively, where the *key* is partition range and column name, the *value* is min-max values and bloom filter. Note that, compared with

Fig. 3(a), Fig. 3(b) lacks of the partition statistics of (min,500] and (500,1000], it means that there is no update about these ranges in Memtable.

Key			Value	
range	column_name		min_max	bloom_filter
(min,500]	total_money	→	39,13010	0 0 1 0 ··· 1 0 1 0
(500,1 000]	total_money	→	156,853	1 0 1 0 ··· 0 0 1 0
(1 000,max]	total_money	→	89,21380	1 1 0 0 ··· 0 0 1 0
(min,500]	user_name	→	able,orz	1 0 0 0 ··· 1 0 1 0
(500,1 000]	user_name	→	aim,yzn	0 1 0 1 ··· 1 0 0 0
(1 000,max]	user_name	→	arse,wim	1 0 1 0 ··· 0 0 1 0

(a)

Key			Value	
range	column_name		min_max	bloom_filter
(1 000,max]	total_money	→	39,6010	1 1 1 0 ··· 1 0 1 0
(1 000,max]	user_name	→	alex,otto	1 0 0 0 ··· 0 0 1 0

(b)

Fig. 3 Statistics cache. (a) SSTable partition statistics; (b) Memtable partition statistics

4.3 Partition pruning algorithm

Algorithm 1 presents the process of partition pruning based on statistics cache. The Input parameters are a sub-range r_i, a set of predicate conditions $\mathcal{PC}$ and a statistics cache SC. The Output parameter is the scan state of r_i. Firstly, we get the Memtable statistics from cache by r_i (line 3). (1) If there is no Memtable statistics about r_i in cache (line 4), it means that data in r_i have not been updated, then we need not scan Memtable. Next, we get the SSTable statistics from cache and determine whether they satisfy the predicate conditions $\mathcal{PC}$. If yes, we should scan the SSTable and the scan state is RS (lines 6–7); if no, the scan state is NR (lines 8–9). (2) If the cache has Memtable statistics about r_i (line 10), we should determine whether the statistics satisfy $\mathcal{PC}$: if yes, we need to scan both Memtable and SSTable, the scan state is RSM (lines 11–12); if no, we continue to get the SSTable statistics from cache and determine whether they satisfy $\mathcal{PC}$: if yes, we will scan both SSTable and Memtable, the scan state is RSM (lines 15–16); if no, we scan nothing and the scan state is NR (lines 17–18). Combined with the algorithm, we give an example to present how to use statistics cache to make partition pruning.

For example: Combined with Fig. 3, assume that there is a range query

$$select \times from\ Orders\ where$$
$$total\_money > 10000$$
$$and\ user\_name = \text{“}ese\text{”}.$$

To determine the scan state of sub-range (min, 500], we first get the Memtable statistics of predicate condition columns (*total_money*, *user_name*) from the cache, and find there is no corresponding value, which means that the data in range (min, 500] have not been updated and we need not to scan the Memtable. Then we continue to get the SSTable statistics from the cache, the min-max values of *total_money* are 39 and 13,010, which have intersection with the range condition $total\_money > 10,000$; but the value of the equal condition “*ese*” is not exist in the bloom filter, so we can determine that the SSTable partition is useless. In conclusion, we should scan neither Memtable nor SSTable, the scan state of (min, 500] is NR.

Algorithm 1 Partition pruning

```
   Input: Sub-range($r_i$);
          Predicate condition($\mathcal{PC}$);
          Statistics cache($SC$);
   Output: Scan state of $r$($state$);
1  $ss_r$ is the SSTablestatistics of $r_i$;
2  $ms_r$ is the Memtablestatistics of $r_i$;
3  get $ms_r$ from $SC$ by $r_i$;
4  if $ms_r$ not exist in $SC$ then
5      get $ss_r$ from $SC$ by $r_i$;
6      if $ss_r \widehat{\cap} \mathcal{PC} \neq \phi$ then
7          $state$ = RS;
8      else
9          $state$ = NR;
10 else
11     if $ms_r \widehat{\cap} \mathcal{PC} \neq \phi$ then
12         $state$ = RSM;
13     else
14         get $ss_r$ from $SC$ by $r_i$;
15         if $ss_r \widehat{\cap} \mathcal{PC} \neq \phi$ then
16             $state$ = RSM;
17         else
18             $state$ = NR;
19 return $state$;
```

In this section, we leverage partition pruning statistics cache to prune partitions. There is a challenge. Since the Memtable is stored in remote, its statistics has to be synchronized to p-nodes through network, as the Memtable services data writes, it is evolving over the time, the statistics cache on a p-node may not remain the same with the source one after the last synchronization. Therefore, we designs an efficient cache synchronization strategy in Section 5, which ensures that using the statistics got from cache still guarantees the consistency under snapshot isolation.

5 Consistence maintenance

Sometimes the statistics in cache may be not matched with the data in Memtable, which can lead errors in query result. For example, a partition data has been updated in Memtable, but the statistics in cache has been not synchronized. Thus, for a range query that requests the newest version data, if the data in Memtable is satisfy its predicate conditions, but the statistics in cache is not, then the partition will be filtered by the partition pruning and we will miss reading the latest target value. To address this problem, we propose a version-based cache synchronization strategy, which ensures that using the statistics cache in the range query can make the data access achieves snapshot isolation consistence. In this section, we firstly introduce the read consistency under snapshot isolation. Secondly, we describe the version-based cache synchronization strategy in detail. At last, we optimize the batch range query by designing an asynchronous update method to synchronize the statistics cache.

5.1 Consistence under snapshot isolation

In a database system, a query or an update operation usually can be viewed as a transaction. Snapshot isolation is a guarantee that all reads in the transaction will see the data with a consistent version. For example, for a transaction T with snapshot isolation, it will generate a version v_T at the beginning of T, and then all reads in T can only see the data with the max version smaller than v_T. To better understand snapshot isolation, we explain it in Fig. 4.

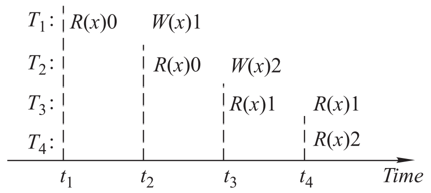


Fig. 4 Transactions with snapshot isolation

In Fig. 4, there are four transactions T_1, T_2, T_3 and T_4. Assume that the read consistence is *snapshot isolation*, and the version generated at the beginning of every transaction is their start time t_1, t_2, t_3 and t_4. Firstly, we define a variable x and its initial value is 0, $W(x)1$ is a write operation that updates x to 1, $R(x)1$ is a read request for x and the return value is 1. In T_1 and T_2, there are two write operations $W(x)1$ and $W(x)2$, which are committed in the interval $t_2 \sim t_3$ and $t_3 \sim t_4$ respectively. Then for T_3, its start time is t_3, which means that it can only read the latest writes committed before t_3. Therefore, no matter what the read request time(t_3 or t_4) for x in T_3, the result is always 1. However, in T_4, the start time is t_4, then it can read the updates of $W(x)2$ which is the latest write operation committed before t_4.

Continue with the above description, assume that there is a range query in T_4 and its predicate condition is $x > 1$, then we use the statistics cache to make partition pruning. If the corresponding data version of Memtable statistics is t_3 in the cache, it could miss scanning the updates of $W(x)2$ when we use the partition pruning decision to execute read operation, but the updated data $(x, 2)$ of $W(x)2$ satisfies the predicate conditions, which is not up to snapshot isolation. On the contrary, if the corresponding data version of statistics is bigger than t_4, we must be able to read the data $(x, 2)$. Thus, in order to ensure the data access achieve snapshot isolation consistency, we need to make sure that the corresponding data version of statistics got from cache is bigger than v_t.

When the statistics are synchronized from Memtable to p-node, we can design a version for the statistics by the latest data version,which will be sent to p-node and stored in cache. Thus, when we access the statistics cache, it can directly use the version to check the validity of statistics. For convenience, We call the designed version as *cache version* in following articles.

5.2 Version-based cache synchronization

Before describing the version-based cache synchronization strategy, we first introduce a lightweight synchronization method with *statistics log*, which can reduce the network overhead when the synchronization frequency is higher.

Lightweight synchronization with *statistics log* When the Memtable is changed by the write operation frequently, the synchronization frequency of statistics cache will be very high. In this scenario, if we send the statistics of entire Memtable to p-node for every synchronization, it will lead to higher network load. Thus, we can use the redo log to generate *statistics log* and then incrementally synchronize statistics cache, the process can be described as Fig. 5. When a write operation is executed on the Memtable, redo log entry is prepared to avoid data loss caused by node failures, it records the rowkey, column name and updated value of a cell. Before flushing the redo log entry into disk, we can map the rowkey into a partition range, and then use the range, column name and updated value to generate a *statistics log* entry, which can be directly applied to the update of local statistics. When a synchronization request arrives, *statistics log* are transported to p-nodes instead of entire statistics.

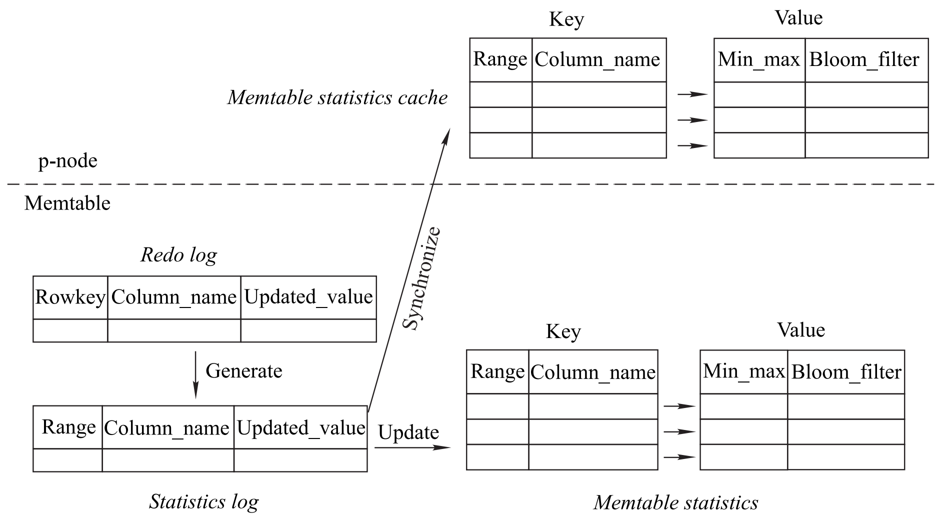


Fig. 5 Lightweight synchronization with *statistics log*

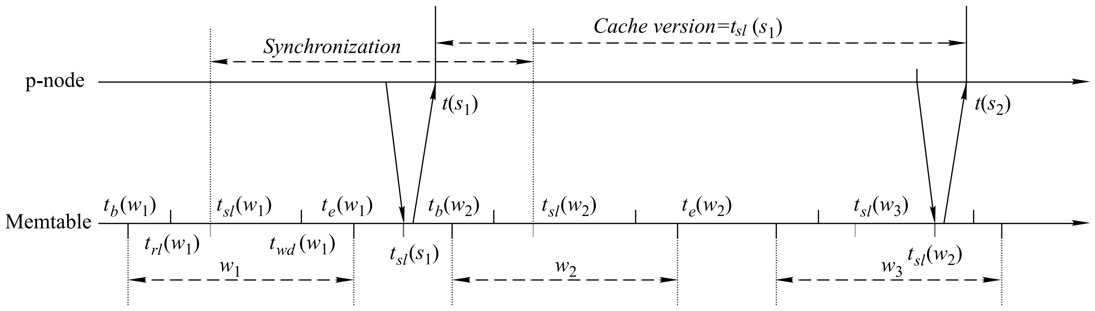


Fig. 6 Write operation and *cache version* design

Figure 6 shows the complete process of write operation which includes generating *statistics log*. Three write operations(w_1, w_2, w_3) are conducted on the Memtable. Assume that w_1 is a write operation, it is grouped by four steps:

(1) Generate redo log. Write operation generates redo log and buffered it in the main memory (e.g., time from $t_b(w_1)$ to $t_{rl}(w_1)$).

(2) Generate *statistics log*: using redo log to generate *statistics log* (e.g., time from $t_{rl}(w_1)$ to $t_{sl}(w_1)$).

(3) Write redo log into disk: redo log is written into the disk when the *statistics log* update finished (e.g., time from $t_{sl}(w_1)$ to $t_{wd}(w_1)$).

(4) Publish: after the write thread has finished disk writing. Data modifications of the write is applied into the Memtable (e.g., time from $t_{wd}(w_1)$ to $t_e(w_1)$).

Version-based synchronization The cache synchronization mechanisms in the literature can be grouped into two main categories: push based and pull based. Push-based mechanisms are mostly server-based, where the server informs the caches about updates, whereas pull-based approaches are client-based, where the client asks the server to update or validate its cached data. Combined with the above discussion, we choose a version-based pull method, where the *cache version* v_c is stored alongside the cache, and which can be used for range queries to determine the validity of statistics in cache. When a p-node accesses the statistics cache under snapshot isolation, it first confirms whether the *cache version* bigger than the query version v_q, if no, asks the Memtable to update cache. Next, we mainly describe how to design *cache version* to ensure the statistics got from cache is valid.

***Cache version* design** For a write operation w_1, after it is published, the Memtable is invariant before next write operation is published (e.g., time from $t_e(w_1)$ to $t_e(w_2)$). Based on the inference, we can draw the conclusion that statistics, gathered at $t_{sl}(w_1)$ that records the updates of w_1 for Memtable, is newest before next write operation is published (e.g., time from $t_{sl}(w_1)$ to $t_e(w_2)$). Therefore, if $t_e(w_2)$ can be determined, we can denote it as the version of the statistics. However, the time of w_2 published can not be determined, because the interval between w_1 and w_2 is unknown. Thus we can find a time that can be determined between $t_{sl}(w_1)$ and $t_e(w_2)$ as the version of statistics.

Assume that s_1 is a synchronization operation, $t_{sl}(s_1)$ is the time of s_1 arrived Memtable, and the updated time of the statistics accessed by s_1 is $t_{sl}(w_1)$, then we can determine $t_{sl}(w_1) < t_{sl}(s_1) < t_{sl}(w_2)$(the next time statistics updated). Combined with $t_{sl}(w_2) < t_e(w_2)$, we can conclude that $t_sl(w_1) < t_{sl}(s_1) < t_e(w_2)$. In conclusion, $t_{sl}(s_1)$ can be considered as the version of statistics. If the statistics is synchronized to the cache, the statistics *cache version* can be denoted as $t_{sl}(s_1)$.

Based on the above discussion, when a synchronous request arrived the Memtable, it first determines a *cache version* v_c, and then v_c and *statistics logs* are transported to p-node to update the statistics cache. For a range query with snapshot isolation, assume that the version generated at its start time is v_q. When it accesses the statistics cache, it first checks whether v_q is smaller than v_c, if yes, we can directly use the current statistics in cache to make decision; if no, the current statistics is invalid for the query and we need to send a synchronous request to obtain the latest statistics.

5.3 Asynchronous update for batch range query

Figure 7(a) shows the the procedure of statistics cache access for the single range query. When a query accesses the statistics cache and finds it is valid, the current statistics in cache can be directly used to make decision. Otherwise, it needs to send a synchronous request to Memtable to obtain the latest statistics. This access method is suitable for single range query, but it is unfriendly to batch range query. If multiple queries simultaneously access the cache and find it is invalid, they all will send the synchronous requests to Memtable. This causes repeated requests and wasted network bandwidth.

Asynchronous update To solve the above problem, we propose a asynchronous update strategy. Here, the statistics cache is synchronized by an independent thread at a regular interval, which can effectively avoid the repeat requests. In addition, we change the process of cache access to maximize the usage rate of invalid cache, which is shown in Fig. 7(b). When the p-node finds statistics cache is invalid but it is not the time to synchronize, we can first get the out-of-date statistics from cache to decide on a scan state, and then iterate through the cache until it has been synchronized. When we access the latest cache, we need to determine whether it has change compared with the previous version: if yes, we should get the latest statistics from cache and make a new decision; if no, we can directly use the scan state decided by the out-of-date cache to scan data.

6 Experiment

6.1 Experimental setup

The partition pruning scan method designed in this work has been applied in a well-known distributed relational database named Oceanbase [9], which is developed by Alibaba. In this system, the process of queries is split to two layers: query execution layer and data storage layer. The query execution layer is responsible for running query plans on query processing node named *query-server*, and data storage layer is in charge of retrieving data to provide query execution. The data storage layer node has two types server: *chunk-server* and *update-server*. The static data(SSTable) is stored on

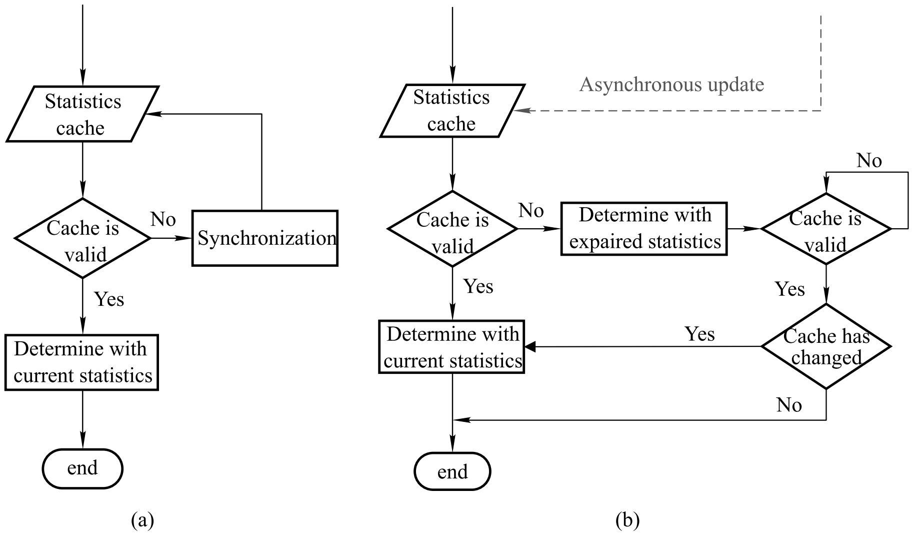


Fig. 7 Procedure of statistics cache access. (a) Single range query; (b) batch range query

chunk-server, and *update-server* record the incremental data in the Memtable. For a query with the predicate condition on non-primary attributes, the system original partition scan method is that all SSTable partitions are accessed, and Memtable partitions are also retrieved at the same time. For short, we denote the original partition scan method by OPS. We also have integrated our partition pruning scan method in the original system which is called as PPS.

All the experiments are conducted on a cluster of 9 physical machines (3 Mergeservers, 3 Chunkservers, 3 Updateservers (1 leader, 2 followers)), where each is equipped with an Intel(R) Xeon(R) CPU (E5-2620 0 @ 2.00GHz, with totally 24 cores, 500GB RAM and 1TB SSD while running an OS of Red Hat with version 4.4.7-4.

We choose four queries *Q-42,Q-52,Q-55,Q-96* from TPC-DS [10] benchmark as our workload: six tables, namely date_dim, item, store_sales, time_dim, store and household_demographics are included, each of them is partitioned on its primary key according to a certain size. Predicate conditions of the queries involve only non-primary keys and there are no any other indexes on non-primary columns, thus the scan operations require to scan all the partitions. Table 2 shows all the predicate conditions of four queries and Table 3 describes the scanned tables and corresponding predicate conditions of each query. Note that, for the equal conditions in Table 2, their corresponding columns are on the non-primary attribute and they need to execute a range scan as well as range conditions.

Table 2 Predicate conditions

No.	Predicate condition
1	date_dim.d_moy = random(11,12,uniform)
2	date_dim.d_year= random(1998,2002,uniform)
3	item.i_manger_id = 1
4	item.i_manger_id = random(1,100,uniform)
5	household_demographics.hd_dep_count = random(0,9,uniform)
6	time_dim.t_our = text("20",1,"15",1,"16",1,"8",1)
7	time_dim.t_minute >= 30
8	store.s_store_name = ‘ese‘

Table 3 Scanned tables and corresponding predicate conditions of each query

Q-42, Q-52		Q-55		Q-96	
Table	Conditions	Table	Conditions	Table	Conditions
date_dim	1,2	date_dim	1,2	-	-
store_sales	-	store_sales	-	store_sales	-
item	3	item	4	-	-
-	-	-	-	houshold_demographics	5
-	-	-	-	time_dim	6,7
-	-	-	-	store	8

6.2 Effectiveness

In this group of experiments, we demonstrate that effectiveness of PPS by comparing with OPS. Firstly, we verify whether the proposed strategies of PPS have worked from several aspects: I/O cost and Filtration. Next, we analyze the performance improvement of PPS by the crucial parameter(QPS).

I/O cost We show the I/O cost for OPS and PPS under default workload in Fig. 8. We calculate the averaged I/O cost of each query's scan operations. Take the Q-42 as an example, we find that OPS causes about 1000 IOPS (Input/Output Operations Per Second), and PPS may have about 850 Input/Output Operations Per Second. For the rest of Queries (Q-52, Q-55, Q-96), the IOPS of OPS is also superior to PPS, and it means PPS can significantly decrease the I/O costs for different queries.

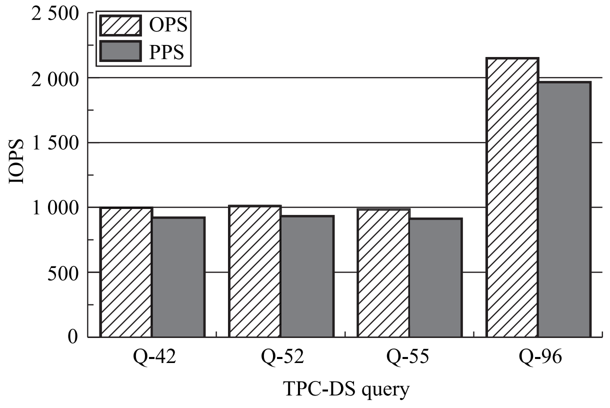
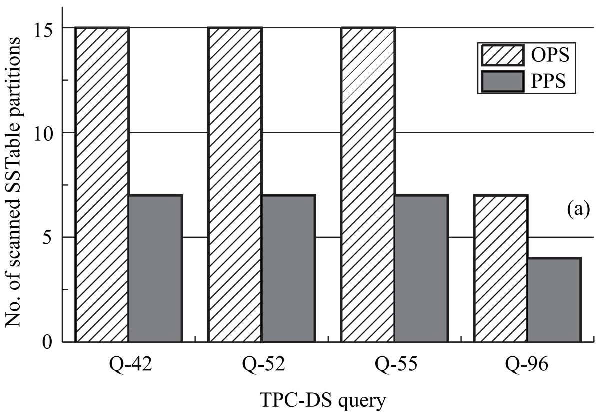


Fig. 8 I/O cost

Filtration We compare OPS and PPS by reporting the number of scanned SSTable partitions or Memtable partitions for different queries. Results are shown in Fig. 9(a) and Fig. 9(b). We can see that PPS access fewer number of SSTable partitions than OPS in Fig. 9(a). To retrieve data from partitions, OPS needs to access all related partitions (15 partitions) for any query (Q-42, Q-52, Q-55 and Q-96). However, the number of scanned SSTable partitions under

PPS, which take the partition-filter strategy, is significantly smaller than OPS. PPS only accesses 6, 6, 6 and 4 partitions for corresponding queries. For avoiding accessing redundant partitions, PPS also has an outstanding effect on accessing scanned Memtable partitions. As shown in Fig. 9(b), Q-42 with PPS only accesses 3 partitions, but OPS has to access all partitions. The other queries are also benefited from PPS. This group of experiments illustrates that PPS has a distinct filtration for accessing partitions.



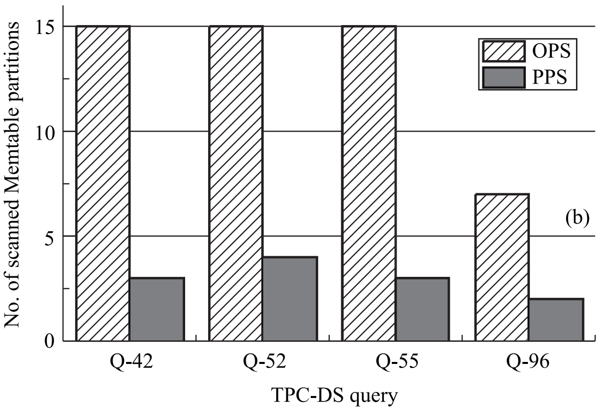


Fig. 9 Filtration. (a) Filtration of SSTable; (b) filtration of Memtable

Performance In Fig. 10, we present the performance of four queries with OPS and PPS respectively. For the perspective of QPS (queries per second), we obverse that the performance of PPS is superior to OPS under different queries. PPS can make Q-42 process about 600 queries during one second, but OPS only reaches about 450 QPS. This is because PPS can filter the redundant accessing partitions to save much CPU overhead that caused by disk I/O. Therefore, there more CPU resources can be taken to execute more queries concurrently. This means that can some progress of performance has been made in PPS.

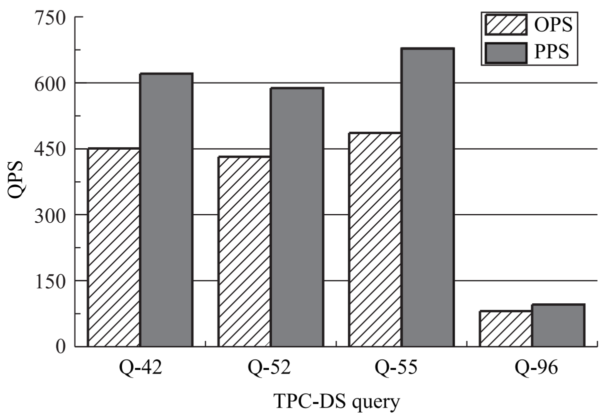


Fig. 10 Performance

6.3 Statistics cache

In this experiment, we make an evaluation for the space cost of statistics cache. We first analyze and give the statistics cache size of all the tables in Table 3, and then compare the overall cost of statistics cache and secondary index, which reflects the advantage of statistics cache in space cost.

Space cost In the process of gathering statistics, min-max values must be collected, and the bloom filer can be selectively constructed by users. In our experiment, we denote the size of a bloom filter as 1,024 bits. Then for a table (e.g., a table in Table 3), its cache size is fixed after the number of partitions has been determined, which is $n_p \times (\sum_{i=1}^{n_c}(2 \times s_i) + 1024/8 \times n_{bf})$ bytes. In the formula, n_p and n_c represent the number of partitions and columns respectively, s_i represents the size of data type in the ith column, and n_{bf} represents the number of columns that generate bloom filter. In Table 4, we give the statistics cache sizes of all the tables in Table 3. As shown in the table, the largest size is 4,700 bytes.

Overall cost For a non-primary key rang query, we can also improve its performance by building secondary index on the non-primary attribute. Compared with statistics cache, secondary index can be regarded as a more precise filtering of data, thus its read latency is relatively lower depending on the data distribution. However, its space consumption is much higher, because of it should record all the data in this column and their corresponding primary keys. According to the above analysis, it is meaningless to compare their performance or space separately, since different application scenarios have

Table 4 Space cost of statistics cache

Table	date_dim	store_sales	item	houshold_demographics	time_dim	store
Memory(bytes)	4,680	2,800	4,700	870	454	874

different requirements. Therefore, we use the cost function $C = P^r S$ [11] to make a comprehensive evaluation for them. In this function, P and S represent performance (latency of read) and space (memory) respectively, and r indicates the relative importance between P and S. To make it fair, assume that P and S are equally important in our experiment, which means that $r = 1$. Figure 11 shows the comparable results of Q-55 and Q-96 between the overall costs of using statistics cache and using secondary index. In the figure, the deeper the area's color, the higher the overall cost. As we can see, the read latencies of Q-55 and Q-96 using statistics cache are higher than using secondary index, but their overall costs are relatively lower, because their space costs are an order of magnitude lower than secondary index.

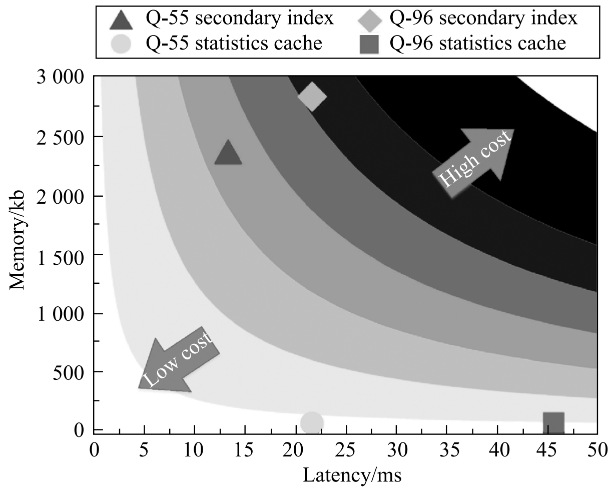


Fig. 11 Overall cost

6.4 Impact of data features

Based on the above results, we prove the effectiveness of PPS. Then, *Q-55* is executed under the dataset with different features to discover the relation between performance and two factors: *partition size* and *storage distribution*.

Partition size We vary the size of partitions for OPS and PPS. As reported in Fig. 12(a), according different sizes of partitions, when the size is 1.25 MB, OPS scans 12 SSTable partitions and PPS only needs to scan 6 SSTable partitions. With the decrease of partition size, the constant volume of data will generate more partitions, and OPS without filtration strategy has to scan all related partitions. For instance, under the other four partition sizes (1 MB, 0.75 MB, 0.5 MB, and 0.25 MB), OPS is involved in 15, 19, 28, and 55 partitions. The number of scanned SSTable partitions achieve a linear increase by decreasing the partition size under OPS. However, the number of scanned SSTable partition size increase slowly with the decrease of partition size. The reason for this phenomenon is that smaller partition size provides a finer granularity to filter redundant partitions. The partition size is smaller, the percent of filtered partitions is larger. As described in Fig. 12(b), the above analyzed principle applies in the Memtable environment. At last, we test the performance of OPS and PPS varying partition size from 1.25 MB to 0.25 MB under default distribution. The result is presented in Fig. 12(c), we find that performance of OPS quickly reaches the bottleneck when the partitioned size is 0.75 MB. It is because that decreasing partition size provides a higher parallelism for OPS to scan data before the 0.75 MB size.

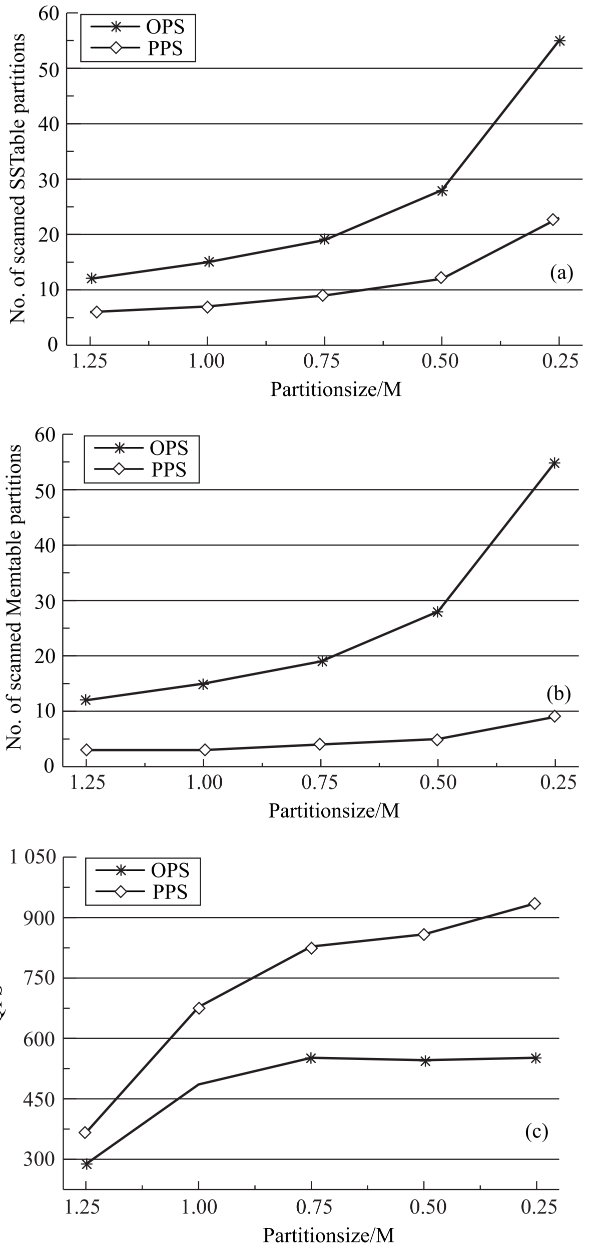


Fig. 12 Impact of partition size. (a) Filtration of SSTable; (b) filtration of Memtable; (c) performance

However, with decrease of partition size, more and more number of partitions reach the limitation of parallel scan, and restrict the performance of query. PPS filters many redundant partitions to achieve high performance while obtaining higher parallelism. Clearly, the partition size is key factor to improve performance of queries under PPS.

Storage distribution The original system is based on log-merge tree architecture, where the incremental data is stored in Memtable, and based data is stored on SSTable respectively. To verify the impacts of storage distribution on performance, we keep the volume of the data stored on disk, and adjust the percents of data on SSTable to resident on Memtable. Results are shown in Fig. 13. As presented in Fig. 13(a) and Fig. 13(b), with the percents change from 0 to 50%, the number of scanned SSTable partitions under OPS is no change, but the number of scanned SSTable partitions and Memtable partitions all increase. The reason is that OPS needs to scan all related partitions, regardless of how much data in Memtable; but the increase of data in Memtable may causes more Memtable partitions can not be filtered, which result in some SSTable partitions that not satisfy the predicate conditions should be scanned. To evaluate the impacts on performance, we test the performance of queries under different percents of data in Memtable in Fig. 13(c). We observe that PPS have a better performance than OPS. Even there is a great decrease of PPS's performance, which is caused by more scanned SSTable partitions are imposed, the worst performance of PPS is still better than OPS.

6.5 Impact of workloads

According to the above experimental analysis, we obverse that filtration is the key to improve the performance. The implementation of filtration method is based on accurate of statistics cache. In the next experiment, we use the timing synchronization of statistics cache to grantee read consistency, and test the impacts of synchronization workload and write workload on the performance of PPS.

Synchronization workload Figure 14 shows the synchronization overhead of statistics cache and performance of PPS under different synchronization intervals. Firstly, the time of one synchronization(a p-node from sending a synchronization request to receiving the response) always takes about 400μs when the synchronization interval varies within 1 ~ 10 ms, which is relatively short compared with the range query latency. Secondly, when the synchronization interval is too short (e.g., 1 ms, 2 ms, and 3 ms), the higher synchronization frequency results in higher overhead and causes the terrible performance (570, 590 and 623). However, longer interval always increase the time of wait for next synchronization if the current cache is invalid. For instance, when the interval is 10 ms, the value of QPS is only 480. In this situation, the out-of-date statistics may lead to a terrible filtration which can decrease the entire performance of queries. From these experiments, we find 6 ms is the best interval for synchronizing statistics cache.

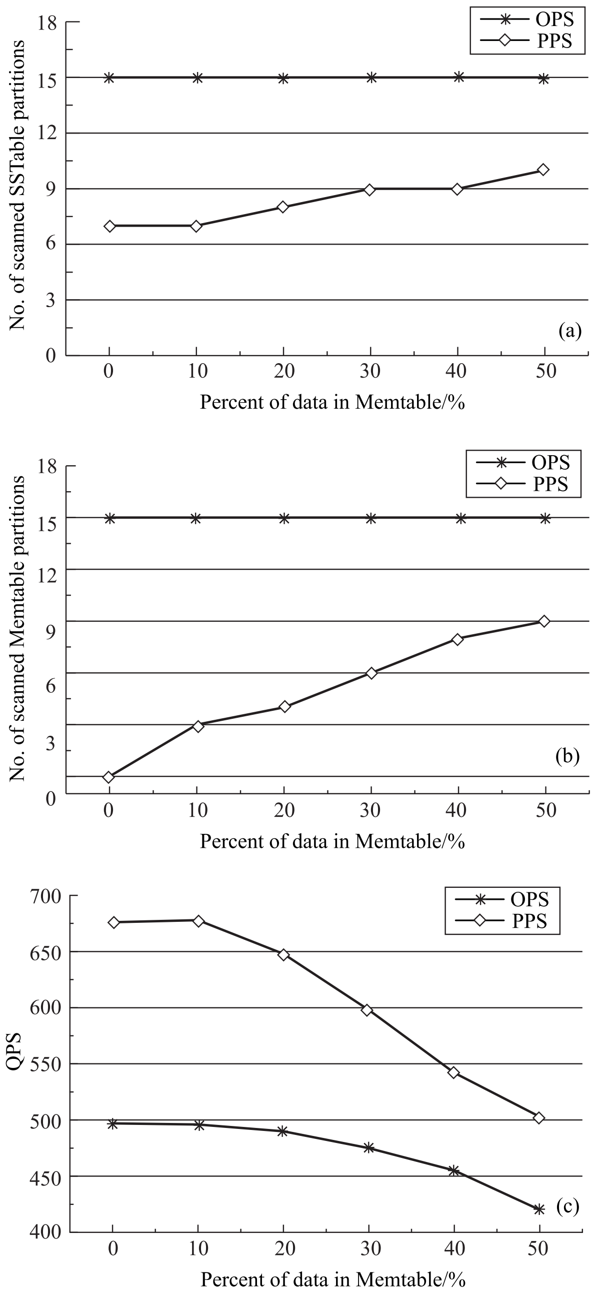


Fig. 13 Impact of storage distribution. (a) Filtration of SSTable; (b) filtration of Memtable; (c) performance

Write workload Figure 15 shows the performance of OPS and PPS by adding different write workload into the original workload. From this figure, we know that the performance of OPS and PPS all keep decreasing with the increase of write workload. This is because that write workload lead to the overloaded Memtable server. However, since PPS can filter some scanning of Memtable partitions, it can reduce the effect of Memtable server overload and achieve a better performance than OPS.

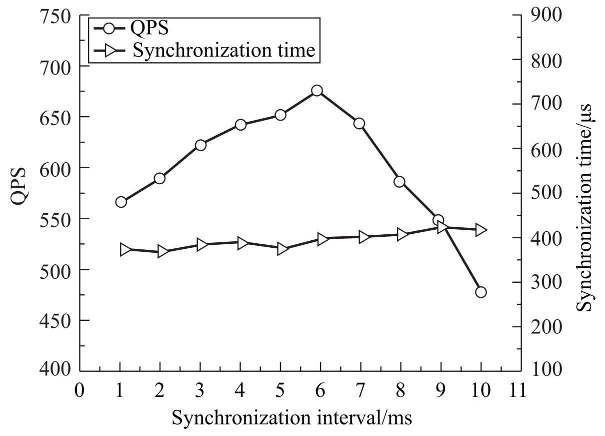


Fig. 14 Impact of synchronization workload

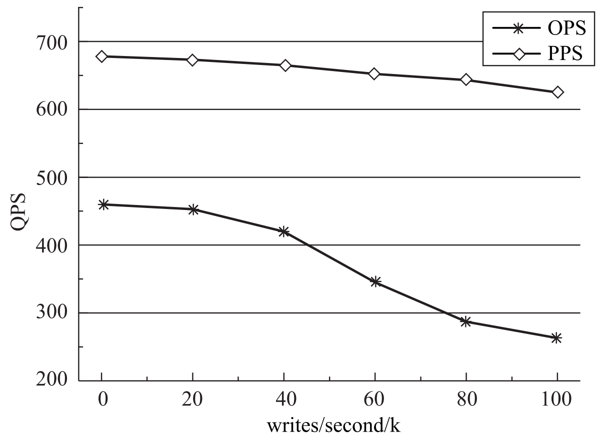


Fig. 15 Impact of write workload

7 Relate works

LSM-tree was born in 1996, it received a lot of attention since Google's Bigtable [2] was published. It first accumulates recent updates in memory, and then flushes the changes to the disk sequentially in batches and merges on-disk components periodically to reduce the disk seek costs. There are two architectures for the distributed LSM-tree: (i) The first is to assign the SSTable and Memtable that in the same range on the same node, such as [2, 3], it scales out for single-partitioned transactions, however it requires add an additional layer for distributed transactions [2]. (ii) The second is to distribute the SSTables and reside the Memtable in one node, such as [9, 12]. It fits massive amounts of data through the expansion of nodes with SSTables, and it can provide high transaction performance by avoiding expensive distributed transaction. But it adds the network communication between SSTable and Memtable. In this paper, we describe and formalize the pruning partition problem based on the second architecture. The problem is more important in the second architecture since more communication cost can be saved if one partition is pruned. As the formalization is similar, our solution also suits the first architecture.

LSM-tree optimizes data writes, however, it brings side effect for read operation. Some researches have been done to improve the read performance of LSM-tree. bLSM-tree [4] uses bloom filters to avoid unnecessary disk access on an SSTable, which is adopted in [2, 13]. Muhammad [5] improves the performance of major *Compaction* so that multiple SSTable files can be merged more quickly. VT-trees [14] is developed as a variant of LSM-tree that avoids unnecessary copies of SSTables during *Compaction* by adding pointers to old SSTables. These techniques only aim at reducing the SSTable access, none is able to answer whether to access the Memtable or not. Next work [15] uses Bloom filters to filter most unnecessary Memtable access without weakening the consistence. However, it can not optimize the range scans because the Bloom filter can not test whether a range of records exist or not. Secondary index [16] can be built on the LSM-tree to improve the performance of range queries, but its maintenance and space costs are higher.

Data partitioning is a well-known technique for achieving efficiency and scalability when processing large amounts of data. One of the well-known benefits of it is reduced scan time by scanning only the relevant parts. There are many DBMS (IBM DB2 [17], Oracle [18], Microsoft SQL Server [19]) support both single-level and multi-level partitioning, most of them implement partition elimination during query optimization, where selection predicates are used to determine which parts to scan. What is more, as Hadoop quickly becomes a popular ecosystem for big data analytics, many query engines for optimizer SQL on Hadoop support partition elimination, such as Cloudera's Impala [20], Facebook's Presto [21], and Hortonworks' Stinger [22].

8 Conclusion

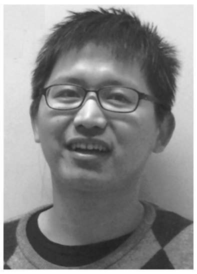
In this paper, we present an efficient partition pruning strategy that optimizes scan operations of range queries under distributed LSM-tree. The strategy mainly exploits the statistics cache to avoid redundant partitions scanning. Taking the write workload into consideration, we also propose a version-based cache synchronization strategy, which ensures that using the Memtable statistics cache in the range query can guarantee data access achieves snapshot isolation consistency. To evaluate the partition pruning strategy, we integrate it into an open source distributed database system. Experimental results show the proposed strategy outperforms the original system.

Acknowledgements This work was partially supported by the Youth Science and Technology – "Yang Fan" Program of Shanghai (17YF1427800), Youth Foundation of Natural Science Foundation (61702189), National Hightech R&D Program (863 Program) (2015AA015307), the National Natural Science Foundation of China (Grant Nos. 61432006 and 61672232).

References

1. O'Neil P, Cheng E, Gawlick D, O'Neil E. The log-structured merge-tree (LSM-tree). Acta Informatica, 1996, 33(4): 351–385
2. Chang F, Dean J, Ghemawat S, Hsieh W C, Wallach D A, Burrows M, Chandra T, Fikes A, Gruber R E. Bigtable: a distributed storage system for structured data. ACM Transactions on Computer Systems (TOCS), 2008, 26(2): 1–26
3. Lakshman A, Malik P. Cassandra: a decentralized structured storage system. ACM SIGOPS Operating Systems Review, 2010, 44(2): 35–40
4. Sears R, Ramakrishnan R. BLSM: a general purpose log structured merge tree. In: Proceedings of ACM SIGMOD International Conference on Management of Data. 2012, 217–228
5. Ahmad M Y, Kemme B. Compaction management in distributed key-value datastores. Proceedings of the VLDB Endowment, 2015, 8(8): 850–861
6. Wang J, Zhang Y, Gao Y, Xing C X. PLSM: a highly efficient LSM-tree index supporting real-time big data analysis. In: Proceedings of IEEE Computer Software & Applications Conference. 2013, 240–245
7. Bloom B H. Space/time trade-offs in hash coding with allowable errors. Communications of the ACM, 1970, 13(7): 422–426
8. Daudjee K, Salem K. Lazy database replication with snapshot isolation. In: Proceedings of International Conference on Very Large Databases. 2006, 715–726
9. OceanBase: an open source high performance distributed database system supporting massive data. Github Website
10. TPC-DS: a decision support benchmark that models several generally applicable aspects of a decision support system, including queries and data maintenance. TPC-DS Homepage
11. Zhang H C, Lim H, Leis V, Andersen D G, Kaminsky M, Keeton K, Pavlo A. Surf: practical range query filtering with fast succinct tries. In: Proceedings of the 2018 International Conference on Management of Data. 2018, 323–336
12. Zhu T, Zhao Z Y, Li F F, Qian W N, Zhou A Y, Xie D. Solar: towards a shared-everything database on distributed log-structured storage. In: Proceedings of 2018 USENIX Annual Technical Conference. 2018, 795–807
13. RocksDB: an embeddable persistent key-value store for fast storage. Wikipedia
14. Shetty P, Spillane R, Malpani R, Andrews B, Justin S, Erez Z. Building workload-independent storage with VT-trees. In: Proceedings of Usenix Conference on File and Storage Technologies. 2013, 17–23
15. Zhu T, Hu H Q, Qian W N, Zhou A Y, Liu M Z, Zhao Q. Precise data access on distributed log-structured merge-tree. In: Proceedings of Asia-Pacific Web (APWeb) and Web-Age Information Management (WAIM) Joint Conference on Web and Big Data. 2017, 210–218
16. Zhu Y C, Zhang Z, Cai P, Qian W N, Zhou A Y. An efficient bulk loading approach of secondary index in distributed log-structured data stores. In: Proceedings of International Conference on Database Systems for Advanced Applications. 2017, 87–102
17. DB2 Partitioned Tables. IBM Official Website
18. Baer H, Belden E, Dijcks J P, Fogel S, Hobbs L, Lane P, Lee S K. Oracle(R) Database VLDB and Partitioning Guide 11g Release 2. Oracle Corporation, 2011
19. Talmage R, Memtors S Q. Partitioned table and index strategies using SQL server 2008. Microsoft, 2009
20. Cloudera Impala: real-time queries in apache hadoop. Cloudera Official Website
21. Presto: Interacting with petabytes of data at facebook. Prestodb Official Website
22. Stinger: Interactive query for apache hive. Hortonworks Website

Chenchen Huang is a PhD candidate in the School of Data Science and Engineering, East China Normal University, China. Her research interests mainly include database system theory and implementation, query optimization and index structure of in-memory database.



Huiqi Hu is currently a lecture in the School of Data Science and Engineering, East China Normal University, China. He received his Phd Degree from Tsinghua University, China. His research interests mainly include database system theory and implementation, query optimization.

Xing Wei is a PhD candidate in the School of Data Science and Engineering, East China Normal University, China. His research interests mainly include database system implementation, query optimization, and in-memory computing technology.

Weining Qian is currently a professor in computer science at East China Normal University, China. He received his MS and PhD in computer science from Fudan University, China in 2001 and 2004, respectively. He served as the co-chair of WISE 2012 Challenge, and program committee member of several international conferences, including ICDE 2009/2010/2012 and KDD 2013. His research interests include Web data management and mining of massive data sets.

Aoying Zhou is a professor on computer science at East China Normal University, China where he is heading the Institute for Data Science and Engineering. He got his master and bachelor degree in computer science from Sichuan University, China in 1988 and 1985 respectively, and won his PhD degree from Fudan University, China in 1993. He is now acting as the vice-director of ACM SIGMOD China and Technology Committee on Database of China Computer Federation. He is serving as a member of the editorial boards of some prestigious academic journals, such as VLDB Journal, and WWW Journal. His research interests include Web data management, data management for data-intensive computing, and in-memory data analytics.