Front. Comput. Sci., 2019, 13(4): 760–777
https://doi.org/10.1007/s11704-018-7198-6

RESEARCH ARTICLE

Fault-tolerant precise data access on distributed log-structured merge-tree

Tao ZHU, Huiqi HU (✉), Weining QIAN, Huan ZHOU, Aoying ZHOU

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

© Higher Education Press and Springer-Verlag GmbH Germany, part of Springer Nature 2018

Abstract Log-structured merge tree has been adopted by many distributed storage systems. It decomposes a large database into multiple parts: an in-writing part and several read-only ones. Records are firstly written into a memory-optimized structure and then compacted into in-disk structures periodically. It achieves high write throughput. However, it brings side effect that read requests have to go through multiple structures to find the required record. In a distributed database system, different parts of the LSM-tree are stored in distributed fashion. To this end, a server in the query layer has to issues multiple network communications to pull data items from the underlying storage layer. Coming to its rescue, this work proposes a precise data access strategy which includes: an efficient structure with low maintaining overhead designed to test whether a record exists in the in-writing part of the LSM-tree; a lease-based synchronization strategy proposed to maintain consistent copies of the structure on remote query servers. We further prove the technique is capable of working robustly when the LSM-Tree is re-organizing multiple structures in the backend. It is also fault-tolerant, which is able to recover the structures used in data access after node failures happen. Experiments using the YCSB benchmark show that the solution has 6x throughput improvement over existing methods.

Keywords distributed data storage, log-structured merge tree, linearizability, fault tolerance

Received June 10, 2017; accepted November 3, 2017

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

1 Introduction

Modern applications store TB-sized or even PB-sized data in database systems. The amount of data is too huge to store in a single server entirely. As a result, it leads to the fast development of distributed database systems [1]. To better satisfy the storage requirement, many distributed database systems choose to implement the log-structured merge-tree (LSM-tree) [2]. The LSM-tree organizes records in multiple components: a Memtable and several SSTables, following the notations used in [3]. The Memtable is a memory-based structure, optimizing for high write throughput. The SSTable is a disk-based structure, offering large storage capacity and servicing read requests only. Records are written into the Memtable in the first, and then migrated into a SSTable in batch. The LSM-tree offers high write throughput compared with the conventional storage mechanism (e.g., B-tree). It has been widely implemented by distributed storage systems such as BigTable [3] and Cassandra [4], where the Memtable and SSTables are kept in the main memory and distributed file system (e.g., GFS [5]) respectively.

However, the LSM-tree sacrifices the read performance to some extent. To read a record, a read operation has to iterate over the Memtable and all SSTables until find the required data item. The dedicated record only exists in one structure and accessing the others is useless. The problem becomes even worse when the query layer is decoupled from the storage layer in distributed systems. Though systems using LSM-tree offer excellent performance, they lack some important features. Hence, some database systems (e.g., Megastore [6]

and Percolator [7]) build a query layer upon these storage systems to add SQL interface or transaction support. A node in the query layer interacts with the underlying storage layer through network communication. As a result, querying the distributed LSM-tree will issue many useless communications and increase the data access latency.

Realizing the limitation of LSM-tree in data access, some work [3] relies on *major compaction* to merge multiple SSTables together and reduce the number of SSTable to be visited. Some work [3, 8] tries to filter unnecessary SSTable read by using the Bloom filters [9]. These techniques only aim at reducing the SSTable access(see details in Section 2.1). But none is able to answer whether to access the Memtable or not. Reducing useless Memtable read is of great importance. Firstly, useless read requests only compete for shared software and hardware resources (e.g., latches, processors) of the Memtable server with write requests. Secondly, accessing Memtable is useless for most data entries. It is because the LSM-tree only stores recently committed data entries in the Memtable. Most data entries are held by SSTables.

This work targets at the distributed database system where Memtable, SSTables and query processing nodes, noted as p-node in the following, are deployed on different servers and proposes an effective way to precisely locate the storage structure for accessing. Before processing a read request, a p-node is able to identify the right structure for reading without contacting the storage layer. The problem is challenging as a p-node is difficult to determine whether the remote Memtable contains the required record especially when the Memtable keeps changing. Based on the designed precise data access mechanism, we consider two practical issues that make it to provide continuous services for applications: (i) ensuring consistent data access when the LSM-Tree is re-organizing its data storage in the backend; (ii) providing fault-tolerant data access service, which is able to recover all in-memory data structures required by our mechanism when node failures happen under a distributed system. We make the following contributions which can be summarized as follows:

1) A Bloom filter with low maintaining and synchronization overhead is designed to encode data existence for the Memtable.

2) A lease-based strategy is designed for p-nodes to keep a copy of the Bloom filter of Memtable, which helps the precise data access achieve the *linearizability* [10] consistency.

3) Our approach is designed to work robustly even node failures happens or the LSM-Tree compacts multiple index structures in the backend.

4) We conduct comprehensive experiments to evaluate our proposed method, results show that our method has 6x performance improvement over existing designs.

The paper is organized as follows. Section 2 revisits the mechanism of LSM-tree and formalize its precise data access problem. Section 3 presents a Bloom filter with low maintaining overhead in synchronizing data existence information from the Memtable server to p-nodes. Section 4 presents the lease-based Bloom filter synchronization. Section 5 discusses how to enable precise data access at the same time when the LSM-Tree compacts data storage. Section 6 discuss how to recover the structures used for precise data access in face of node failures. Experimental evaluations are showed in Section 7. We present related work in Section 8 and make conclusions in Section 9.

2 Preliminary

This section briefly reviews the design of LSM-tree in the distributed system and analyze its inefficiency in data access. Based on these, we formalize the precise data access problem.

2.1 Storage model

We target at a distributed database system built upon the log-structured storage. A typical structure is illustrated in Fig. 1. Basically, it consists of a Memtable, several SSTables and multiple p-nodes.

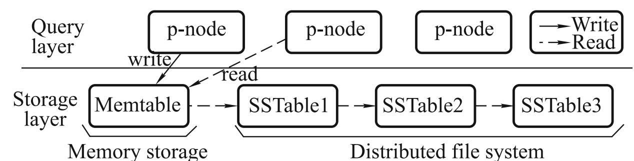

Fig. 1 Data storage and access on distributed LSM-tree

Memtable is the in-memory structure which services read and write requests. As in the LSM-tree, data writes are only allowed to perform on the Memtable, it resides in main memory to facilitate write performance. The Memtable can be implemented as any index structure that optimized for main memory access, such as bw-tree [11]. For a write request, if an entry is inserted, it is directly added into the Memtable; if an entry is updated, its new value is added into the Memtable.

Since the Memtable is kept completely in the main memory to improve write performance, data loss is possible when the Memtable server crashes. In order to avoid data loss, redo log entries [12] are forced into durable storage for recovery purpose. When a write operation arrives, its redo log entry

is firstly constructed and then be flushed into the commit log stored in durable device. After the redo entry has been flushed, its content is then written into the Memtable and published. As the current disk device only supports limited input/output operations per second (IOPS), it is extremely expensive to run a disk I/O for per redo log entry. An important technique named *group commit* [13] is used to improve I/O throughput by collecting multiple redo log entries together and then flushing them in a single disk write.

SSTable is the on-disk and immutable structure where data is stored in lexicographic order based on their primary key. The SSTable is generated by freezing an active Memtable. The frozen Memtable is transferred into distributed file system and becomes the SSTable. A new Memtable replaces the old one for servicing further write requests. Thus, the SSTable is a read-only data structure. In short, the log-structured storage firstly keeps written data in the Memtable. It then freezes and shifts the Memtable into durable storage when the Memtable reaches a certain size. As time goes by, it will generate several SSTables as illustrated in Fig. 1, where SSTable-1 is the latest created one and SSTable-3 is the oldest one.

Data access Figure 1 illustrates how a p-node in the query layer pulls a record from the underlying storage layer. To read a record entry with key k from the distributed LSM-tree, a p-node has to go through the 1st Memtable, 2nd SSTable, 3rd SSTable, ..., until seeing a record with k as its key. The data access is a limitation for the such storage model as a p-node has to visit multiple structures to locate a record. What is worse, Memtable and SSTables are stored in a distributed fashion so that a p-node has to issue many remote procedure calls to visit these structures. Basically, there are two kinds of optimizations taken by existing systems to reduce SSTable access namely *major compaction* and *Bloom filter*.

Major compaction To reduce the access overhead and avoid maintaining too many SSTables at the same time, major compaction is conducted in back-end to merge multiple SSTables together. A write process would read key-value entries from multiple SSTables and merge them into a single one. With the completeness of the major compaction, all old ones become expired and the new one comes into service. Major compaction helps control the number of SSTables in the LSM-tree within a proper size. With the help of compaction, data access can visit a single well-merged SSTable instead of multiple small ones.

Bloom-filter of SSTable Another optimization is to maintain a Bloom filter [9] for a SSTable. The Bloom filter has been proven to be efficient in testing whether an element is a member of a set. It has a 100% recall rate but allows only few probabilities of appearance of false positives. If an element is a member of a set, then all hashing bits in the Bloom filter should be **1**. After being constructed, the Bloom filter is immutable and can be easily cached on remote servers. When a server tries to read an entry from a SSTable, firstly, it can check the Bloom filter to judge whether the target entry exists or not. If the entry does not exist, it can simply ignore the SSTable and neglect the remote access.

Both the major compaction and the SSTable Bloom filter only reduce the SSTable access. However, each read request is still required to visit multiple structures as accessing the remote Memtable is inevitable. In this paper, we seek a way to precisely locate a single structure for reading so that a p-node can judge locally to determine visiting either the Memtable or one SSTable.

2.2 Precise data access

In a distributed database (e.g., Percolator [7], Megastore [6]) built upon distributed LSM-tree, a read request has to invoke multiple remote procedure calls. Intuitively, if a p-node is able to determine whether a Memtable or SSTable contains the required data entry precisely, it can directly visit the dedicated structure without making other nonsense communications. Previous works have designed methods to filter useless SSTable access. In a difference, this work aims at seeking efficient methods to filter useless Memtable access. As our work aims at filtering Memtable access, there is no essential difference between one SSTable or multiple. We assume there is a Memtable and one SSTable in the following.

The precise data access is to let a p-node determine visiting either the Memtable or one SSTable without contacting the underlying storage servers. Formally, it can be described as follows:

Definition 1 (Precise data access) Given an evolving structure Memtable m whose owned key set can be denoted as: $\mathcal{K}_m = \{k_1^m, k_2^m, \ldots\}$ and an immutable structure SSTable s whose owned key set can be denoted as set: $\mathcal{K}_s = \{k_1^s, k_2^s, \ldots\}$, a p-node is required to answer the following membership query $q(k)$ without contacting storage servers before accessing a data entry e indexed by k:

$$q(k) = \begin{cases} m, & k \in \mathcal{K}_m, \\ s, & k \notin \mathcal{K}_m \bigwedge k \in \mathcal{K}_s, \\ none, & \text{else}. \end{cases}$$

By getting the answer, the p-node can directly visit the target server to get the required data entry. In the above formula,

it is easy to answer whether $k \in \mathcal{K}_s$ by caching the Bloom filter of SSTable on p-nodes (as discussed in Section 1). But, answering whether $k \in \mathcal{K}_m$ is of much more difficulties.

The kernel problem is to answer *whether a remote evolving set contains a typical element or not*. An intuitive solution is to maintain a Bloom filter for the Memtable as well, and synchronize the structure to multiple p-nodes. Each p-node checks its local Bloom filter to determine whether an entry exists in the Memtable or not. However, there are two difficulties here. Firstly, since the Memtable is stored in *remote*, its Bloom filter has to be synchronized to p-nodes through network. But the Bloom filter is of large size. Direct synchronization tends to exhaust the network bandwidth. Secondly, as the Memtable services data writes, it is *evolving* over the time, a copy of its Bloom filter on a p-node may not remain the same with the source one after the last synchronization. The potential difference between the primary Bloom filter and its copies tends to lead to inconsistency read (i.e., if an entry e is newly created on the Memtable and it changes its bit in the Bloom filter from 0 to 1. But the bit in the copy is still 0 before re-synchronized. A read operation is not aware of the existence of e by examining the copy of the Bloom filter. It would neglect the Memtable access and return a stale version of the record stored in the SSTable.).

To address these problems, Section 3 discusses how to maintain a Bloom filter on the Memtable and synchronize that to a p-node. Section 4 presents a lease-based mechanism to guarantee that using a copy of the Bloom filter on a p-node still guarantees the linearizability consistency.

3 Entry existence identification

The first problem is to determine whether the Memtable contains an entry or not. A straightforward method is to maintain a Bloom filter ($\mathcal{B}_m$) for Memtable in the same way for SSTable($\mathcal{B}_s$). For each entry in the Memtable, its corresponding hashing bits in $\mathcal{B}_m$ are marked as **1**. *A problem is that the Bloom filter does not allow any element to be deleted. It is worthwhile to remind that the Memtable also never delete any data items*. Even though a query tries to delete a record from the database, the record is not directly removed from SSTable or Memtable, instead, an entry with a deleted flag is inserted into the Memtable for the record to be deleted. Hence, the *DELETE* query is treated as a special *UPDATE* and the Memtable never directly remove any existing entry.

As mentioned before, the Bloom filter would be synchronized from the Memtable server to multiple p-nodes. It is critical to reduce the network overhead. However, the method involves much synchronization overhead for *INSERT* queries. Once an entry is inserted into Memtable, its hashing bits in $\mathcal{B}_m$ are set to **1**. The change of these bits must be synchronized to each copy of $\mathcal{B}_m$, introducing considerable overhead. We propose an efficient method which totally avoids synchronization cost for data insertion by carefully designing the maintaining policy of $\mathcal{B}_m$ and its usage in data access.

3.1 Overview

Figure 2 illustrates how $\mathcal{B}_m$ is maintained and synchronized. When write operations are executed on the Memtable, redo log entries are prepared to avoid data loss caused by node failures. Before flushing a group of redo entries into disk, updates on $\mathcal{B}_m$ are generated from redo log entries based on the policy in Section 2. These updates act as the modification log entries of the $\mathcal{B}_m$ (short for bf-logs in the following). To reduce synchronization cost, $\mathcal{B}_m$ is not directly sent to p-nodes, instead bf-logs are transported to p-nodes and a copy of $\mathcal{B}_m$ on a p-node catches up with the source by applying (replaying) the identical bf-logs (See Section4).

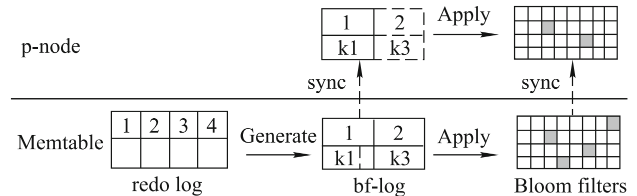

Fig. 2 The Bloom filter maintenance and synchronization

3.2 Maintenance of $\mathcal{B}_m$

The Memtable can be probably changed by the following types of operations: *INSERT*, *UPDATE*, *DELETE*. We discuss with the policy that each operation modifies the Bloom filter of Memtable. Considering an entry e with key k:

UPDATE A *UPDATE* operation modifies an existing record entry e. There are two situations: (i) If $k \notin \mathcal{K}_m$, then e is newly created in the Memtable. A bf-log is generated for k, which adds existence of k into $\mathcal{B}_m$; (ii) If $k \in \mathcal{K}_m$, then some previous operation has added k into $\mathcal{K}_m$ and handled its update on $\mathcal{B}_m$, the current one does nothing.

DELETE A *DELETE* operation removes an existing record entry e. As discussed in the beginning of the section, It is treated as special *UPDATE* operation, which adds an deleted flag for k. A read operation can identify whether the entry is deleted by reading the flag. Thus, the *DELETE* operation is treated the same to the *UPDATE*. (i) If $k \notin \mathcal{K}_m$, then k is modified for the first time. An entry with the deleted flag will

write into the Memtable. It is required to add the existence of k into $\mathcal{B}_m$; (ii) If $k \in \mathcal{K}_m$, then some previous operation has updated or deleted the entry e and processed its modification on $\mathcal{B}_m$, thus the current one does nothing.

INSERT An *INSERT* operation writes a non-existing record entry e into the Memtable. (i) If $k \notin \mathcal{K}_m$, then $k \notin \mathcal{K}_s$ must also stand. To read e, a p-node can easily find $k \notin \mathcal{K}_s$ by checking $\mathcal{B}_s$ and then e can only be found in the Memtable. The p-node can infer the fact without querying $\mathcal{B}_m$. Thus, there is no necessity in modifying $\mathcal{B}_m$ when inserting e into Memtable. (ii) If $k \in \mathcal{K}_m$, this means the entry must be tagged with a deleted flag, its bits in $\mathcal{B}_m$ must have been already processed by a completed *DELETE* operation. Therefore, we do not need to modify $\mathcal{B}_m$ again.

Table 1 summarizes when an operation should update $\mathcal{B}_m$. It is only modified when an entry is newly created on the Memtable by a *UPDATE* or *DELETE* operation.

Table 1 The maintenance policy of $\mathcal{B}_m$

State	*INSERT*	*UPDATE*	*DELETE*
$k \in \mathcal{K}_m$	×	×	×
$k \notin \mathcal{K}_m$	×	√	√

3.3 Data access based on $\mathcal{B}_m$

Considering the query $q(k)$ in Definition 1, we denote $k \in \mathcal{B}_m$ or $k \in \mathcal{B}_s$ when all hashing bits of the key k are **1** in the (copy of) $\mathcal{B}_m$ ($\mathcal{B}_s$) respectively. Temporarily, we assume there is no false positive in the Bloom filter and leave the issue in the next. There are totally four situations here:

(1) If $k \notin \mathcal{B}_m$ and $k \notin \mathcal{B}_s$, then e is either non-existing or newly inserted into Memtable. In this situation. A p-node will access Memtable. If existed, a p-node can read e from the Memtable. If not, a p-node announce its non-existence. Since reading a non-existing entry is rare in real workload and does not affect the performance a lot.

(2) If $k \notin \mathcal{B}_m$ and $k \in \mathcal{B}_s$, then e never receives any modification after it is written into the SSTable. A p-node will directly access the SSTable.

(3) If $k \in \mathcal{B}_m$ and $k \in \mathcal{B}_s$, e is stored on SSTable at first and then get modified. A p-node should visit the Memtable to read the entry.

(4) $k \in \mathcal{B}_m$ and $k \notin \mathcal{B}_s$ is infeasible to appear. Since $k \notin \mathcal{B}_s$ is 0, the entry does not exist in the SSTable. Hence, it does not exist in the database before it is added into the Memtable. Thus, the entry can only be brought into the Memtable by *INSERT* operation. However, *INSERT* never adds the key filed of an entry into the $\mathcal{B}_m$. On the other hand, after k is added into $\mathcal{K}_m$, no operation would modify $\mathcal{B}_m$ for the entry again. We should have $k \notin \mathcal{B}_m$, which contradicts with the assumption that $k \in \mathcal{B}_m$. Thus, the case is not possible.

Based on the above discussion, Table 2 summarizes the location of an entry under different combinations of Bloom filter states. Note that the false positive is not considered in the table.

Table 2 The Bloom filter state and the access matrix

$k \in \mathcal{B}_m$	$k \in \mathcal{B}_s$	Location	Comment on the entry
0	0	Memtable	Not existed or newly *INSERT* erted into Memtable
0	1	SSTable	Stored in the SSTable
1	0	-	Not possible
1	1	Memtable	Stored in SSTable and has updates in Memtable

Handling false positives The Bloom filter permits false positives to happen. Two kinds of false positives can happen here:

a) *The false positive of $\mathcal{B}_m$: there is $k \in \mathcal{B}_m$, but the entry is not found in the Memtable.* In this case, reconsider the situation (3) above, if $k \in \mathcal{B}_s$ stands, it is required to further access the SSTable.

b) *The false positive of $\mathcal{B}_s$: there is $k \in \mathcal{B}_s$, but the entry is not found in the SSTable.* Reconsider the situation (2) above, the entry is either non-existing or newly inserted into the Memtable. Accessing the Memtable is required for the case.

Precise data access algorithm The pseudocode of precise data access is shown in Algorithm 1. As discussed, the Memtable is selected as the destination *if and only if* k is contained by $\mathcal{B}_m$ or not by $\mathcal{B}_s$ ($k \in \mathcal{B}_m$ or $k \notin \mathcal{B}_s$). 1) If the condition stands, a p-node pulls e from Memtable in line 2–3. If it is not found and k is also contained by $\mathcal{B}_m$, the p-node continues to access the SSTable in case of false positives in line 4–5 and 9–10. 2) If the condition fails, then a p-node pulls e from the SSTable in line 7. When nothing is returned, the p-node access Memtable in case of false positives in line 8–10. As Bloom filter potentially contain false positives, an entry may not exist in Memtable or SSTable even if $\mathcal{B}_m$ or $\mathcal{B}_s$ confirms its existence respectively. When the entry is not found, re-access is used to handle any potential false positive in line 9–10.

Algorithm 1 Precise data access algorithm

```
Input:
  1. Query key k          2. Memtable m_1, and B_m
  3. SSTable s_0, and B_s
Output: Entry e
1  if k ∈ B_m or k ∉ B_s then
2  |  read e from m_1;
3  |  if k ∈ B_s then
4  |  |  alter = s_0;
5  else
6  |  read e from s_0;
7  |  alter = m_1;
8  if e is null then
   |  /* re-access in case of false positives */
9  |  read e from alter;
10 return e;
```

3.4 Log-based synchronization

A p-node is required to synchronize the remote Bloom filter of the Memtable to the local. A straight way is to send $\mathcal{B}_m$ from Memtable server to each p-node. Generally, the Bloom filter could occupy several MBs in order to encode more than a million elements. Apparently it is not proper to synchronize such a large object frequently among servers.

Lightweight synchronization We propose a lightweight method by simply sending the bf-logs, which act as the modification logs of $\mathcal{B}_m$. A copy of $\mathcal{B}_m$ replays bf-logs to catch up with the source. As discussed above, only a small part of operations generate bf-logs, the number of bf-logs is much smaller than that of redo log entries. It means that the bf-log synchronization has lower network overhead than the log replication [14], which is essential for implementing a fault-tolerant service.

Implementation Each bf-log is indexed by a monotonically increasing serial number. The Memtable server keeps the newest bf-logs in a circular buffer. In synchronization, a p-node pulls bf-logs from the remote by sending the largest serial number N ever received. The Memtable server replies with all bf-logs whose serial number is bigger than N. Since the circular buffer has limited memory space, some new bf-logs may overwrite the oldest ones in the buffer. When any required bf-log has been overwritten, the p-node would be directly replied with the whole $\mathcal{B}_m$. It happens when a p-node is firstly connected into the system and it tries to synchronize its local empty Bloom filter with $\mathcal{B}_m$ the first time.

4 Consistence maintenance

A copy of $\mathcal{B}_m$ (noted as $\mathcal{B}'_m$) kept by a p-node may differ with the source after synchronization. When $\mathcal{B}'_m$ is out-of-date, a record has a newer version in the Memtable, but a p-node (using Algorithm 1) may skip it and reads the record from the SSTable and returns a stale version to clients. To address the problem, we discuss how to guarantee strong consistency, namely the *linearizability* [10], for the precise data access. Figure 3 is used throughout this section to explain our design.

4.1 Linearizability

Linearizability is a consistency model. It requires that 1) each operation has a linearization point at some instant between its start time and complete time, and 2) each operation appears to occur instantly at that point. If all read operations visit both Memtable and SSTable, the property is easily guaranteed. For a read operation, its linearization point is the time when it reads the Memtable. For a write operation, its linearization point is the time when its data modification is applied into the Memtable. With these linearization points, all operations behave properly, i.e., each read operation always returns the value created by the last write operation that occurs before it.

However, the precise data access changes the behavior of read operations. Consider that a read operation $R_x(e)$ tries to retrieve record e and it determines to ignore the Memtable access after checking $\mathcal{B}'_m$. For $R_x(e)$, its linearization point can be the time when it checks $\mathcal{B}'_m$. At that point, (i) if the Memtable does not contain e, no write operation happened before the time point. R_x is still linearizable because it can return the latest version of e by accessing the SSTable only. (ii) If the Memtable contains e, a write operation must occur before the time point. R_x is non-linearizable because it misses a newer version of e stored in the Memtable. In all, R_x is

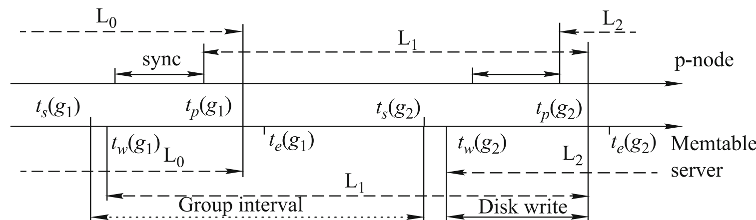

Fig. 3 Group commit and lease management

linearizable if e is not in the Memtable when R_x checks $\mathcal{B}'_m$. Before we discuss how to ensure the linearizability of R_x, we define the *well-constructed* relationship between a Bloom filter and a Memtable in Definition 2.

Definition 2 A Bloom filter $\mathcal{B}$ is well constructed for a Memtable m if: for each record in m, its key *is or is not* added into $\mathcal{B}$ using the policy in Table 2.

Actually, the record e is guaranteed to be not in the Memtable if $\mathcal{B}'_m$ is *well constructed* for the Memtable. Its correctness is proofed in the following.

Proof Assume $\mathcal{B}'_m$ is well constructed for the Memtable at R_x's linearization point, and e exists in the Memtable.

Since R_x ignores the Memtable access, $\mathcal{B}'_m$ must deny the existence of e and R_x must fetch e from the SSTable successfully in Algorithm 1. As e does exist in the SSTable, it can only be brought into the Memtable by a *UPDATE* or *DELETE* operation. In both cases, its key must be added into $\mathcal{B}'_m$ according to the policy in Table 2, which contradicts with the fact that $\mathcal{B}'_m$ denies the existence of e. The conjecture is proved by contraction.

Hence, if $R_x(e)$ ignores the Memtable access when using a well-constructed $\mathcal{B}'_m$ in precise data access, e is guaranteed to be not in the Memtable and then R_x is linearizable.

Clearly, The key to support linearizability is to refresh $\mathcal{B}'_m$ properly and make sure that it is well constructed for the remote Memtable when it is used in precise data access. In the following, we present a lease-based mechanism to refresh $\mathcal{B}'_m$ properly. To begin our discussion, we study how Memtable and $\mathcal{B}_m$ are changed by write operations in the first.

4.2 Group commit

In the Memtable, write operations are committed in group. Group commit is carried out with following steps:

1) Generation. Redo log entries are generated for write operations and buffered in the main memory. A write thread would force these redo entries into the disk in a fixed period, called the group interval (e.g., time from $t_s(g_1)$ to $t_s(g_2)$ in the figure).

2) Start phase begins at a time $t_s(g_x)$ with a group of redo entries formed. Then, bf-logs are generated from the group and applied into $\mathcal{B}_m$ (e.g., from $t_s(g_1)$ to $t_w(g_1)$). After that, the write thread starts to flush the group into the disk.

3) Write phase begins at a time $t_w(g_x)$. The write thread is writing redo entries into the disk (e.g., from $t_w(g_1)$ to $t_p(g_1)$ in the figure). Generally, it takes several milliseconds to finish a disk write under hard disk driver (HDD, see Wikipedia).

4) Publish phase begins at a time $t_p(g_x)$ after the write thread has finished disk writing. Data modifications of the group are applied into the Memtable in the phase (e.g., from $t_p(g_1)$ to $t_e(g_1)$ in the figure). After the group is published, it ends at a time $t_e(g_x)$.

Invariance In the above procedure, both $\mathcal{B}_m$ and the Memtable keep invariant during a period. Considering two successive groups g_1 and g_2, $\mathcal{B}_m$ is invariant from $t_w(g_1)$ (i.e., after bf-logs of g_1 are applied) to $t_s(g_2)$ (i.e., before bf-logs of g_2 are applied) and Memtable is invariant from $t_e(g_1)$ (i.e., after g_1 is published) to $t_p(g_2)$ (i.e., before g_2 begins to publish). By taking advantage of the invariance of Memtable and $\mathcal{B}_m$, we design a lease-based mechanism to ensure the read consistency when a p-node uses a copy of $\mathcal{B}_m$ in data access.

4.3 Lease management

Definition 3 A lease L_x is a contract given by the Memtable server and held by each p-node. It contains an invariant Bloom filter $\mathcal{B}'_m$ (a version of $\mathcal{B}_m$ at some time point) and an expiration time t_x, which ensures that $\mathcal{B}'_m$ is well constructed for the remote Memtable before t_x is reached. Therefore, during the lease, the precise data access provides the linearizability consistency if $\mathcal{B}'_m$ is used in Algorithm 1.

Basic design As discussed above, the Memtable keeps invariant after a group ends and before the next group begins to publish. A direct design is to give a lease which starts at the end of a group and finishes before the publish phase of the next group, so that the Memtable does not change during the lease. Considering two successive groups: g_1 and g_2 in Fig. 3, a p-node can use $\mathcal{B}_m$ at $t_w(g_1)$ between $t_e(g_1)$ and $t_p(g_2)$. Obviously, the version of $\mathcal{B}_m$ is well constructed for the Memtable during the period. But a limitation is that each lease starts after the previous one ends. When a lease is expired on a p-node, the new one is just generated on the Memtable server. Since it takes time to deliver a new lease from the Memtable server to a p-node, there is no valid one on the p-node before the new one is received. Thus, it is impossible to provide p-nodes with a valid lease all the time, making such design impractical. To this end, we consider designing lease with overlap.

Lease design A lease can begin after a group has updated $\mathcal{B}_m$, i.e., $t_w(g_x)$, and end before the next group begin to publish, i.e., $t_p(g_{x+1})$. For example, L_1 can last from $t_w(g_1)$ to $t_p(g_2)$ and $\mathcal{B}'_m$ is the version of $\mathcal{B}_m$ at $t_w(g_1)$. We proof that such design generates valid leases as well.

Correctness Between $t_w(g_1)$ and $t_p(g_2)$, the Memtable has

two versions while $\mathcal{B}'_m$ is constructed with all bf-logs generated by g_1 and all previously ended groups. 1) Before $t_p(g_1)$, the Memtable m_0 contains records committed by all groups those end in prior to g_1. $\mathcal{B}'_m$ is well constructed for m_0 because all bf-logs generated by these groups have been applied in $\mathcal{B}'_m$. Actually, bf-logs of g_1 are also applied in $\mathcal{B}'_m$. And it does not change the well-constructed relationship between $\mathcal{B}'_m$ and m_0. 2) After $t_p(g_1)$, the Memtable m_1 contains records committed by g_1 and all previously ended groups. Now $\mathcal{B}'_m$ is still well constructed for m_1. In both bases, $\mathcal{B}'_m$ is well constructed for the Memtable.

Since two successive leases have overlap in the time-line under such design, a new lease is available for acquisition before the in-using one is going to be expired. In Fig. 3, L_0 and L_1 overlap with each other between $t_w(g_1)$ and $t_p(g_1)$ (i.e., g_1 is in writing phase), which lasts several milliseconds. When L_0 is going to be expired, L_1 is already available for acquisition. It is sufficient for a p-node to extend L_1 from the Memtable server before L_0 is totally expired.

Note that in the overlap of two leases, both their $\mathcal{B}'_m$ work correctly in accessing Memtable. Taking L_0 and L_1 as an example, they have overlap from $t_w(g_1)$ to $t_p(g_1)$. In the period, $\mathcal{B}'_m$ of L_0 is well constructed for the Memtable. On the other hand, $\mathcal{B}'_m$ of L_1 also works for the Memtable, which is discussed by the *Case 0* in the above proof.

4.4 Lease implementation

In order to support the lease management, clocks of all servers are synchronized using the precision time protocol (PTP), which achieves clock inaccuracy within 50 μs on a local area network. Overall, the lease management is done with the following steps. 1) A p-node tries to acquire a lease by sending an acquisition request to the Memtable server. 2) When the request is received by the Memtable server, it replies the p-node with the newest generated lease and all bf-logs required to refresh the copy of $\mathcal{B}_m$ on the p-node. 3) After the p-node receives the reply, it updates its local copy of $\mathcal{B}_m$ and renew the expiration time of the lease. 4) The p-node tries to send a new acquisition request when it considers the current lease is going to be expired. In the following, we discuss how to generate a lease, to acquire a lease and to check the expiration of a lease in detail.

Generation A lease L_x is generated at the time $t_w(g_x)$, containing the current largest bf-log serial number N and the expiration time t_x. Its $\mathcal{B}'_m$ is created by replaying all bf-logs whose serial number is small than N. Its expiration time t_x is allowed be set as $t_p(g_{x+1})$, the time that the next group begins to publish. However, it is not known in advance, but can be inferred by adding the current time, the group interval and time spent on disk writing together (e.g., L_1 in Fig. 3). Local processing time ($t_s(g_x)$ to $t_w(g_x)$) is ignored because it is relatively very short. Group interval is given by system configuration. Disk write time can be estimated from the time used for previous groups.

Commit wait Since t_x is *estimated*, it can be smaller or bigger than its real value $t_p(g_{x+1})$. 1) If $t_x > t_p(g_{x+1})$, the Memtable should not allow g_{x+1} to publish its content. Otherwise, inconsistent read may happen since L_x is not expired now. As a result, the publish phase of g_{x+1} is blocked until t_x is reached. It is called as *commit wait*. 2) If $t_x < t_p(g_{x+1})$, the lease is expired early than the next group begins to publish. Thus, it does not block the next group from publishing. To avoid *commit wait*, we prefer to use the lower bound of the estimated disk write time in determining the t_x, making $t_x < t_p(g_{x+1})$. Note that choosing a small t_x does not harm correctness because $\mathcal{B}'_m$ of L_x is always usable before $t_p(g_{x+1})$.

Lease acquisition In each synchronization, a p-node pulls a lease and bf-logs whose serial numbers are in $(N_1, N_2]$ from the Memtable server (N_1 the largest serial number ever received, N_2 is the one specified by the lease). Synchronization is required when the lease is going to expire soon. Generally, a p-node tries to acquire a new lease when the in-using one will be expired in 400 μs. The number is chosen based on the time used for the round-trip communication, which takes about 200 μs under the 1 Gigabit Ethernet. Since the communication is fast, a p-node receives response early than the current lease is expired.

In Algorithm 1, a p-node first checks whether the Bloom filter is usable by confirming that the expiration time of the lease is not reached. If that is true, it is safe to use $\mathcal{B}'_m$ in data access. A problem is the clock inaccuracy between servers. Since the expiration time t_x of a lease L_x is specified by the Memtable server, to check whether a lease L_x is expired, t_x ought to be compared with t_m (i.e., the system time of the Memtable server). Since the comparison is done by the p-node, it is required for the p-node to calculate the upper bound of t_m. Actually, a p-node can calucate the upper bound of t_m as t_l + *DELETEta*. Here t_l is the time of its local clock and *DELETEta* is the maximal amount of clock difference under the PTP.

5 Compaction

The size of Memtable increases when more and more data are accumulated. To release the utilized memory, the current

active Memtable has to be compacted into a SSTable when its size reaches a threshold. To distinguish the concept of compaction with SSTables in Section 2, we call the process that compact the Memtable into the SSTable *minor compaction*. In this section, we first introduce the process of minor compaction, then we discuss how to achieve precise data access when the minor compaction happens.

5.1 Minor compaction

The minor compaction consists of three general steps: 1) freeze the current active Memtable m_1 and make it immutable; 2) create a new Memtable m_2 to serve future write requests; 3) the immutable Memtable m_1 is converted into SSTable format and moved into a distributed file system. To freeze m_1, the server writes a compaction log entry (short for CLE) into the redo log. The compaction log entry acts as the boundary for data writes. For redo entries written before the CLE, their corresponding data are written into m_1. For redo entries written after the CLE, their data are written into m_2.

Respectively, before flushing the CLE into the disk, the Memtable server appends a special bf-log entry, noted as *bf-change*, into the bf-log buffer. When the special log entry is received by a p-node, the p-node creates a new empty Bloom filter for m_2. For any bf-log entry received after the *bf-change*, it would be replayed into the new Bloom filter. Clearly, the usage of *bf-change* entry is similar with that of the CLE, which acts as the boundary for bf-log entries belonging to different Bloom filters. The CLE and *bf-change* entry ensure that if a redo entry applies its writes into m_1, its bf-log entry is added into $\mathcal{B}_m$ of m_1, otherwise, its bf-log entry is added into $\mathcal{B}_m$ of m_2. Hence, each Memtable has its $\mathcal{B}_m$ constructed correctly.

After the CLE is flushed, a new Memtable m_2 is created. For redo entries written after the CLE, they write their data into m_2. And the m_1 becomes an immutable Memtable now. A daemon thread moves m_1 into the distributed file system and stores that as a new SSTable file. Once m_1 has been totally stored as SSTable, the Memtable server persists the position of the latest CLE into a file, named as *replay-position*. The file informs the recovery procedure the start position for replaying the redo log when the Memtable server is restarted after node failure.

5.2 Data access of minor compaction

Being different from normal cases, there are an active Memtable m_2, an immutable Memtable m_1, and the old SSTable s_0 in the system during the minor compaction, where m_1 and m_2 are stored in the same Memtable server, s_0 is stored in the distributed file system. Since m_1 and m_2 are kept in the main memory of the same server. *A p-node can access both of them efficiently by communicate with the Memtable server once.*

A p-node maintains three Bloom filters at the same time: $\mathcal{B}_s$ for s_0, $\mathcal{B}_{m1}$ for m_1, and $\mathcal{B}_{m2}$ for m_2. Given a query key k, the precise data access is carried out with the following steps.

In the first, we infer whether the entry exists in s_0 by examining $k \in \mathcal{B}_s$ or not. Two cases are possible:

Case 1: if $k \notin Bs$, then the entry is guaranteed to be not in s_0. Hence, the entry can only be found in m_1 or m_2. It accesses m_2 in the first, if the entry is not found, it then accesses m_1. As m_1 and m_2 are stored in the main memory of the same server, accessing m_1 and m_2 can be serviced by a single network communication.

Case 2 : if $k \in Bs$, then the entry may exist in s_0. The p-node examines $\mathcal{B}_{m1}$ and $\mathcal{B}_{m2}$ before issuing remote data access.

a) If $k \in \mathcal{B}_{m1}$ or $k \in \mathcal{B}_{m2}$, then the entry may be stored in m_1 or m_2. The p-node accesses both m_2 and m_1 to find the entry. (i) If the entry is found in m_1 or m_2, then p-node only access Memtable; (ii) If the entry is not found due to false positives, the p-node tries to access s_0 again.

b) If $k \notin \mathcal{B}_{m1}$ and $k \notin \mathcal{B}_{m2}$, then the p-node tries to access s_0 in the first. (i) the entry is found in s_0. In this situation, *the entry is guaranteed to be not in* m_0 *and* m_1. This can be proved as: assuming *op* to be the first operation that brings the entry into Memtables, *op* can only be **Update** or **Delete** because **Insert** only operates on non-existing entry. As a result, *op* must update $\mathcal{B}_{m1}$ or $\mathcal{B}_{m2}$ before writing the entry into Memtables, contradicting with the fact that $k \notin \mathcal{B}_{m1}$ and $k \notin \mathcal{B}_{m2}$. (ii) the entry is not found in s_0, which results from the false positive of $\mathcal{B}_s$. An **Insert** operation may write the entry into Memtables. It does not update $\mathcal{B}_{m1}$ or $\mathcal{B}_{m2}$ as discussed in Section 3. Hence, the p-node is required to access m_0 and m_1.

Algorithm 2 summarizes the data access methods described in the above. Line 1 checks whether the entry is possibly stored in m_1 or m_2 using the rules discussed in the above. If the condition stands, line 2 tries to read e from m_1 and m_2. A p-node reads the m_1 and m_2 at the same time using a single network communication with the Memtable server. If the condition of line 1 does not stand, line 6 tries to read e from s_0. A p-node reads the s_0 by communicating with the

SSTable server. Line 8 checks whether e is not found in the first remote data access, which happens due to the false positive of the Bloom filter. In this case, line 9 accesses the rest structures to find the entry.

Algorithm 2 Data access during minor compaction

```
Input:
   1. Query key k                 2. Memtable m1, and B_m1
   3. Memtable m2, and B_m2       4. SSTable s0, and B_s
Output: Entry e
1  if k ∈ B_m1 or k ∈ B_m2 or k ∉ B_s then
2  |  read e from m1 and m2;
3  |  if k ∈ Bs then
4  |  |  alter = s0;
5  else
6  |  read e from s0;
7  |  alter = {m1, m2};
8  if e is null then
   |  /* re-access in case of false positives */
9  |  read e from alter;
10 return e;
```

In addition to minor compaction, the LSM-Tree also adopts the *major compaction* to merge multiple SSTable files into a single one. When it happens, the SSTable and its $\mathcal{B}_s$ are updated. A p-node only needs to cache a new $\mathcal{B}_s$ for the new SSTable generated by major compaction.

6 Fault tolerance

Node failure is common in distributed environment. In case of node failures, this section introduces how to recover a failed node and re-construct the structures used by precise data access. In all, there are three types of node failures: 1) a server that stores the Memtable is failed, noted as *Memtable server failure*; 2) a server that stores the SSTable is failed, noted as *SSTable server failure*; 3) a p-node that handles query processing is failed, noted as *processing unit failure*.

Obviously, A SSTable server does not lose data when a node failure happens. It is because the SSTable is disk-resident structure. When a SSTable server is restarted, it is only required to re-construct its in-memory bloom filter $\mathcal{B}_s$ by scanning its local SSTable data. In the following, we mainly discuss the recovery of Memtable server and p-node.

6.1 Memtable server failure

Both the Memtable and the primary $\mathcal{B}_m$ is kept in the main memory. When a server is restarted after node failure, the recovery procedure is responsible for restoring both Memtable and $\mathcal{B}_m$, which is achieved by replaying redo log entries.

A redo entry can be abstracted as a quadruple: $\langle LogId, DML, Key, Value\rangle$, where **LogId** field is the unique identifier for each log entry. It is assigned in monotonically increasing order, **DML** represents the type of the operation, i.e., *insert*, *update* or *delete*, **Key** and **Value** fields correspond to *key* and *value* parts of a record.

The recovery procedure replays redo log entries in the order of **LogId**, i.e., the log entry with smaller LogIds are replayed before those with larger LogIds. In the first, the replaying procedure read the *replay-position* file, which keeps the compaction log entry (CLE) position of the last finished minor compaction. Then, the procedure replays all log entries whose LogIds are bigger than that of the CLE.

Given each redo entry, the (Key, Value) pair is firstly written into the Memtable. In the next, $\mathcal{B}_m$ and the bf-log buffer are restored based on the (DML, Key) fields of the redo entry using the same policy described in Section 3.

If a CLE is encountered, the replaying procedure creates a new Memtable, a new $\mathcal{B}_m$, and writes a *bf-change* entry into the bf-log buffer. Next redo entries are replayed into the new Memtable and new $\mathcal{B}_m$. When the recovery procedure is completed, data will be written into the newest Memtable. The old immutable Memtable continues to be converted into SSTable format and be stored into the distributed file system.

Next we proof the correctness of the above procedure that Memtable (or Memtables) are correctly recovered without losing data.

Proof The recovery procedure correctly restores the Memtable as well as the $\mathcal{B}_m$. 1) No data is lost. For redo entries whose LogIds are smaller than the CLE, their data has been already stored as SSTable in the durable storage. Thus, there is no necessity to replay these redo entries. For redo entries with LogIds being bigger than the CLE, they are replayed by the recovery procedure. Hence, their data does not get lost either. 2) Multiple Memtables and their associated $\mathcal{B}_m$ are correctly recovered. The compaction log entry (CLE) acts as the boundary for redo log entries belonging to different Memtables. For those in front of the CLE, they are recovered into an old Memtable; for those in behind of the CLE, they are recovered into a new Memtable. Hence, multiple Memtables can be correctly recovered. On the other hand, each $\mathcal{B}_m$ is re-created using redo entries belonging to the corresponding Memtable. Hence, each $\mathcal{B}_m$ can be recovered correctly.

6.2 Processing unit failure

A p-node is required to recover the data structures ($\mathcal{B}_s$ and $\mathcal{B}_m$) used to precise data access. 1) For SSTable, a p-node

communicates with the SSTable server to fetch the $\mathcal{B}_s$. 2) For Memtable, a p-node acquires the latest copy of $\mathcal{B}_m$ using the policy described in Sections 3 and 4.

When a p-node is restarted, it has not received any bf-log from the Memtable server. The largest serial number N for bf-log is initialized to 0 (see Section 3). A p-node sends the number $N = 0$ to the Memtable server. The Memtable server tries to reply with the p-node with all bf-logs whose serial numbers are bigger than 0. If any bf-log has been overwritten in the circular buffer, then the server directly sends the whole $\mathcal{B}_m$ back to the p-node. If multiple $\mathcal{B}_m$ exist when the minor compaction is started and there are a frozen Memtable and an active one, all of them are sent to the p-node. By pulling $\mathcal{B}_s$ from SSTable server, acquiring a copy of $\mathcal{B}_m$, the p-node can continue to use the precise data access algorithm.

6.3 Replication

In the above, we discuss how to recover a failed server. But, before a failed server comes back, it can not process any user requests. In order to provide continuous service, replication can be adopted to improve system availability [15]. Here we discuss how to maintain replicas for each kind of server.

SSTable server Since a SSTable is read-only structure, it can be easily replicated over multiple servers without worrying about data consistency. In default, a SSTable has three replicas on three distinct servers. Each can process read requests of the SSTable correctly. Whenever one is down, the rest two can still provide service. Hence, a SSTable is available when at least one replica is alive.

Memtable server To maintain replicas for a Memtable, we use the distributed consensus protocol Raft [16]. In default, the Memtable is replicated over three distinct servers. Using the Raft protocol, one of them is elected as the leader while the others become followers. All read/write requests of the Memtable are processed by the leader. For each write request, the leader is required to synchronize its redo entry to followers. A write request is considered to be successfully committed only when its redo entry is durable on more than half of all replicas.

By replicating the Memtable over three servers, the system can tolerate the failure of one server. If a follower is down, the leader processes read requests of the Memtable as usual. For write requests, the leader replicates their redo entries to the other living follower. Therefore, write requests get committed because their redo entries become durable on more than half of all replicas. If the leader is down, a follower will be elected as the new leader and continue to service user requests. Hence, with only one server being down, the system can still process read/write requests of the Memtable. More details of the Raft-based replication mechanism can be found in [16, 17].

When the leader is down, a follower is elected as the new leader. In precise data access, a p-node begins to read the Memtable, pull the $\mathcal{B}_m$ and extend leases from the new leader. Hence, it is necessary to guarantee that the Memtable on the new leader contains all previous committed data and the corresponding $\mathcal{B}_m$ is well constructed. In implementation, a write request is successfully committed only when the leader has synchronized its redo entry to half of all followers. When a follower has received consecutive redo entries, it flushes them into its local disk in the first. Then, a follower replays redo entries to restore Memtable and $\mathcal{B}_m$ using the algorithm discussed in Section 6.1. Once a follower is elected as the new leader, the Raft protocol guarantees that the server has all previous committed redo log entries [16]. Hence, the Memtable on the new leader contains all committed data and the corresponding $\mathcal{B}_m$ is well constructed.

P-node A p-node is only responsible for receiving user requests, reading records from storage servers, processing user-defined relational operators and writing records into the Memtable server. Since a p-node does not store data, it does not require any replica. Whenever a p-node is down, user requests can be processed by any other living node.

Summary Table 3 summaries how to recover different kinds of servers from node failures, and how to provide fault tolerant service using replication. Firstly, a SSTable server does not need to recover its data because all records are stored in the disk. It re-constructs $\mathcal{B}_s$ by scanning the SSTable and add all record keys into $\mathcal{B}_s$. If a SSTable has k replicas (in default $k = 3$), the system can tolerate at most $k-1$ replicas to be lost. Secondly, a Memtable server recovers its data by replaying the redo log. Its $\mathcal{B}_m$ is recovered by replaying bf-logs, which can be generated from the redo log. When $2k + 1$ replicas are configured for the Memtable (in default $k = 1$), the system can tolerate at most k Memtable servers to be lost. Lastly, a p-node does not require data recovery since it does not store any data. When a p-node is restarted, it pulls $\mathcal{B}_s$ and $\mathcal{B}_m$ from storage servers. And there should be at least one p-node left so that the system can service user requests.

Network partition In addition to the node failure, another kind of failure is the link failure, which leads to the network partition among servers. Link failures split a distributed system into several subsets. For servers in the same subset, they can communicate with each other. For those from different subsets, there are communication errors among them. If network partition happens, the system is available only

Table 3 Recovery and replication of different servers

Server type	Data recovery	Bloom filter recovery	Replicas	Minimum requirement
SSTable server	Not necessary	Re-build $\mathcal{B}_s$ by scanning SSTable	k	1
Memtable server	Replay the redo log	Re-generate and replay bf-logs	$2k+1$	$k+1$
p-node	Not necessary	Pull $\mathcal{B}_s$ or $\mathcal{B}_m$ from remote servers	k	1

when there is a subset, which contains more than half of Memtable replicas, at least one SSTable replica and one p-node. In such a subset, a Memtable server can be elected as the new leader by the Raft protocol. The leader is able to handle read/write requests of the Memtable. The living SSTable server can process read requests of the SSTable. And the p-node can pull a copy of $\mathcal{B}_m$ and extend lease from the leader Memtable server. Then it can use precise data access to fetch records from the Memtable or the SSTable.

7 Experiment

We have implemented our method into *Oceanbase*, a distributed relational database developed by *Alibaba*. It has two layers: a query processing layer and a storage layer. In the storage layer, the Memtable is kept by a server named *Updateserver* and the SSTable is stored on *Chunkserver*.

All the experiments are conducted on a cluster with 20 servers. Each has *two 2.00 GHz 6-Core E5-2620 processors*, *192GB* DRAM and *1 TB* HDD, connected by a *1 Gigabytes switch*. Three servers is used to host Memtable and three are used to host the SSTable. Both Memtable and SSTable are replicated over three servers. Among three Memtable servers, one acts as the leader and the others are followers. The rest 14 servers are used to deploy p-nodes.

All experiments use the YCSB benchmark, which is popular in evaluating the read/write performance for a database system. We populate *1 million records* in the database. In default, 95% records are initially stored in the SSTable and 5% are stored in the Memtable. We further adjust the storage distribution in Section 2. The workload contains unlimited read requests and a fixed rate of write requests (50K writes/sec). In default, records are accessed in uniform distribution. Skewed access distribution is considered in Section 2.

In the following experiments, three methods are evaluated.

1) In *NDA*, we use the naive data access method. A p-node firstly pull records from the Memtable, if no record is returned, it accesses the SSTable in further.

2) In *BDA*, a p-node examines the Bloom filter of SSTable to avoid unnecessary access as discussed in Section 1.

3) In *PDA*, we use the techniques designed in this work.

In these methods, a p-node would try to cache data from SSTable. Since a SSTable is immutable, such cache mechanism can reduce duplicate remote access efficiently. A Bloom filter uses 2 million bits and 4 hash functions. Performance of different methods are evaluated by read operations processed per second (ops).

7.1 Concurrency & scalability

Concurrency Figures 4–6 shows the performance of different methods by varying the number of clients connected with the system. 9 p-nodes are deployed to service requests. Figure 4 shows the performance under a read-only workload. Overall, PDA has the best performance under all cases. It reaches about 1,100k ops when 450 clients are used, which is about six times that of the NDA or BDA. The performance of NDA and BDA increases with more clients are simulated, but stabilizes once the Memtable server is overloaded. They easily make the Memtable server be performance bottleneck since they have to access the Memtable for every request. On the other hand, performance of PDA improves all the time and does not witness bottleneck from Memtable access. Figures 5 and 6 add 50k and 100k writes/second into the workload respectively. The peak performance of NDA and BDA decreases when more writes requests are added into the workload. They have performance bottleneck on the Memtable access but writes requests are also heavily processed by the Memtable server. Performance of PDA also slightly decreases because more records are brought into Memtable by ongoing writes operations and less Memtable access is filtered by PDA.

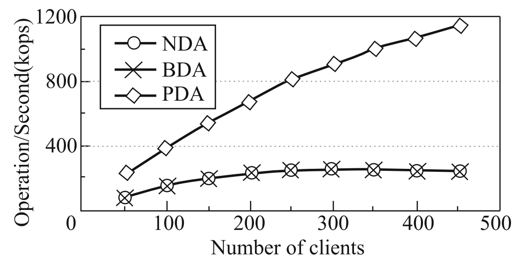

Fig. 4 Varied clients, read-only workload

An important observation is that *NDA and BDA share similar performance in all cases, as well as in the following experiments.* It is because the SSTable is well merged and

cached on each p-node. Reducing SSTable access does not contribute to performance at all. Actually, BDA shows its benefits only when there are many small SSTable files in the system and no SSTable cache is maintained on each p-node. Hence, reading an entry has to issue many SSTable access. And SSTable access is expensive since no local SSTable cache is enabled on a p-node.

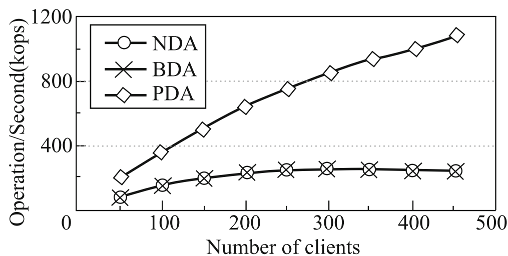

Fig. 5 Varied clients, 50k writes/second

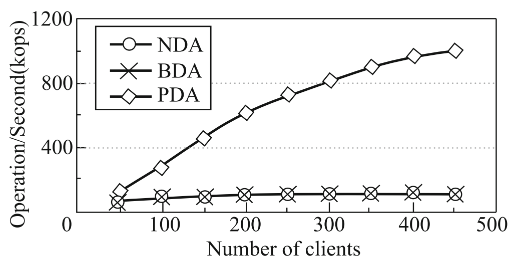

Fig. 6 Varied clients, 100k writes/second

Scalability Figures 7–9 evaluates performance by varying the number of p-nodes connected with storage servers. In each case, we adjust the number of clients used to achieve the best performance. By deploying more p-nodes, the synchronization overhead of PDA is increased. But PDA still shows linear scalability with respect to the number of p-nodes used. The overhead introduced by Bloom filter synchronization is negligible compared with those unnecessary Memtable access eliminated by PDA. On the other hand, BDA and NDA achieve their peak performance when about 10 p-nodes are deployed. They are severely influenced by the mass useless Memtable access. By adding write requests into the workload, NDA and BDA still witness a performance decrease due to the overloaded Memtable server. In all, it is worthwhile to adopt PDA in processing a read-intensive workload.

7.2 Storage & access distribution

Storage distribution In the previous experiments, 5% records have their latest versions stored in Memtable. Figure 10 shows the performance by varying the percentage of records in the Memtable. When about 50% records should be read from Memtable, PDA achieves about 300k ops. With the percentage goes down, the performance keeps increasing. In comparison, both NDA and BDA are not sensitive to the parameter. Given a record who has its lasted version in the Memtable, PDA process in the same with the others. Thus, when the percentage of these records increases, the performance of PDA would get closer to that of NDA/BDA. But it still shows about 200% improvement even when 50% records should be read from Memtable. In real workload, Memtable would not contain a large percent of records. For utilizing hardware resources better, a large Memtable would be frozen and transformed into SSTable.

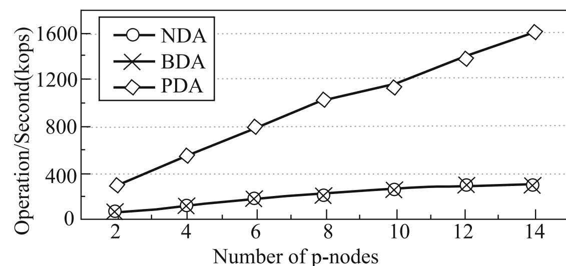

Fig. 7 Varied p-nodes, read-only workload

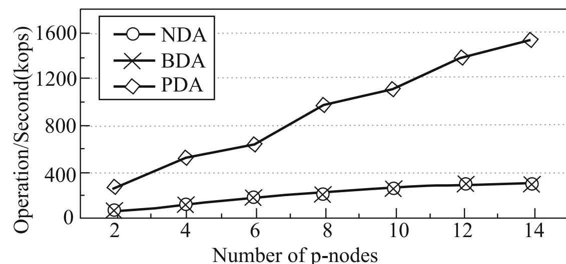

Fig. 8 Varied p-nodes, 50k writes/second

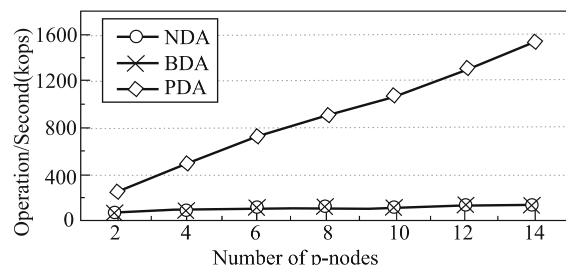

Fig. 9 Varied p-nodes, 100k writes/second

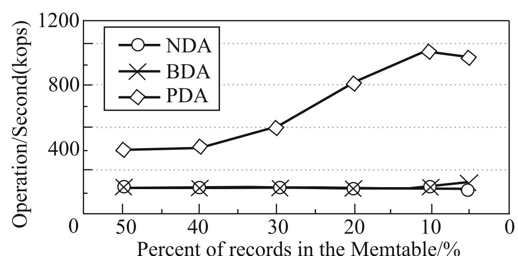

Fig. 10 Varied data storage distribution

Skewed access distribution Figure 11 shows the performance under a skewed access distribution. In YCSB, request parameters are generated under a Zipfian distribution, which uses θ to adjust the skewness. If the θ is larger, the distribution is skewer. Under a skew access distribution, some records are "hot" being frequently updated and read, and some are "cold" being seldom accessed. When $\theta = 0.9$, PDA achieves about 187k ops while NDA/BDA is about 128k ops. PDA has about 1.46x improvements. It is because most records read are also get updated under a very skewed workload. When θ goes down, performance of PDA increases. Under $\theta = 0.1$, it achieves 921k ops and has about 4.93x improvements over that under $\theta = 0.9$.

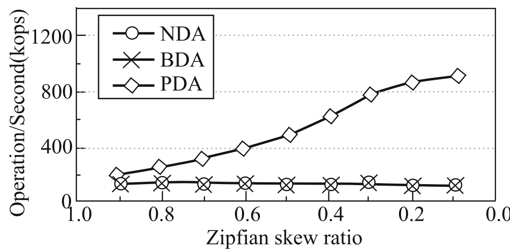

Fig. 11 Skewed access distribution

7.3 Synchronization & clock inaccuracy

Synchronization overhead Figure 12 shows the overhead of the Bloom filter synchronization. A p-node refreshes its local Bloom filter when the current lease is going to be expired. The Memtable server compute a lease's expiration time based the group interval and disk write time. Figure 12 shows the synchronization time and frequency by varying the group interval. Firstly, it always takes about 200 μs for a p-node from sending a synchronization request to receiving the response. The time used is relatively very short compared with the group interval. Secondly, when Memtable flushes one group of redo entires per 2ms, each p-nodes issues about 700 synchronizations per second. Synchronization frequency decreases when a longer group interval is used and a p-node is granted with a longer lease. An exception is the case where 1ms group interval is used. Its synchronization frequency is also smaller than the case where 2ms group interval is used. By using a short group interval, many small groups are formed. Writing small groups increases the average disk write time because HDD favors large sequential writes. As a result, the disk write time is increased, making each p-node receive a longer lease again.

Clock inaccuracy In precise data access, all servers should have their system clocks synchronized. Here, we use the precision time protocol (PTP) to synchronize their clocks. On a local area network, it achieves clock accuracy in the degree of microseconds. As discussed in Section 4, the inaccuracy of clock has impact on the usability of a $\mathcal{B}'_m$ on each p-node. If the clock is very inaccurate, a p-node may overestimate the time of Memtable server and consider the current lease to be expired. As a result, it is not possible for the p-node to use PDA. Instead, it is only allowed to use BDA since $\mathcal{B}_s$ can be used to filter some SSTable access.

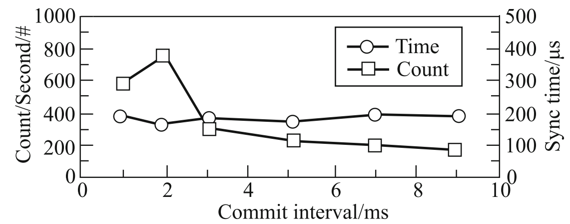

Fig. 12 Synchronization overhead

Figure 13 studies the impact of clock inaccuracy on the performance. Here we simulate the maximal amount of clock inaccuracy between two servers. To achieve that, we assume all server clocks are exact accurate under the PTP. When a p-node fetches its system time, its clock returns the time by pulsing current time with a random number. The random number is chosen between $[-DELETEta, DELETEta]$ with a truncated normal distribution. Hence, the maximal amount of inaccuracy between a p-node and the Memtable server is *DELETEta*. In the experiment, we gradually increase *DELETEta* and evaluate its influence on the performance. When *DELETEta* increases from 1ms to 3ms, the throughput changes a little. It is because the group commit happens in every 5ms and the generated lease lasts longer than 5ms. The lease is long enough to ensure that a new one can be extended before the current one is considered to be expired. When the amount of inaccuracy continues to increase, the throughput begins to drop because a p-node may consider its current lease to be expired due to overestimating the time of the Memtable server. As a result, a p-node can only uses the BDA instead of the PDA. When the inaccuracy reaches 20ms, the throughput of PDA (about 160 k) is still slightly bigger than that of BDA (about 120k) or NDA (about 110k).

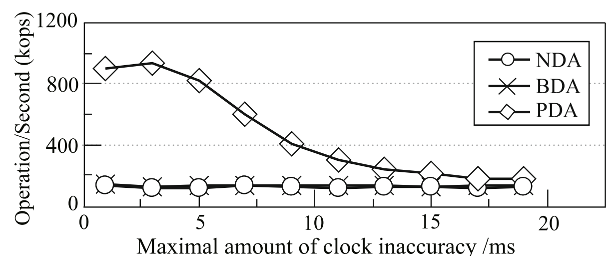

Fig. 13 Impact of clock inaccuracy

7.4 Compaction & fault tolerance

Compaction Figure 14 shows the impact of data compaction. We keep tracks of the throughput of the system by running it with the default workload for a period and conducting data compaction. 1) A minor compaction operation starts at 0′15″. The operation has little impact on the performance since it only freezes the active Memtable and replace it with a new one on the Memtable server. 2) At 0′45″, the major compaction is started to merge the old SSTable with the one created by minor compaction. The major compaction operation also notifies the p-node to read the new SSTable from SSTable servers. Though the new SSTable is not physically created yet, a SSTable server would access multiple SSTable files, and return the latest version when a p-node tries to read the new SSTable. Since each p-node never caches data items for the new SSTable, it has to issue more network communications with SSTable servers. As a result, the performance drops. With more and more data items of the new SSTable are cached on p-nodes, the performance keeps increasing and returns to the normal level at about 2′30″.

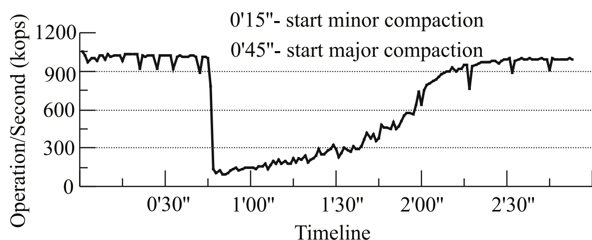

Fig. 14 Impact of compaction

Fault tolerance In Figs. 15–18, we simulate the impact of node failures by running the default workload. Different types of servers are killed at a given time.

In Fig. 15, one of ten p-nodes is kill at 0′30″. The performance drops a bit because some clients can not send queries any more since they lose their connections with the failed p-node. The p-node is started at 1′00″ and clients are able to re-connect into the system again. Then the performance returns again.

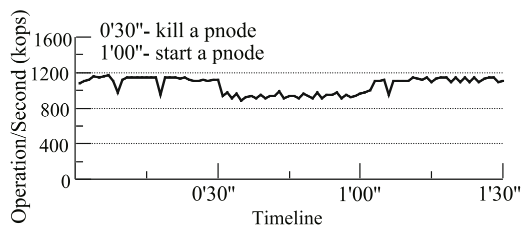

Fig. 15 Failure of p-node

In Fig. 16, one of three SSTable servers is killed at 0′30″, which has little impact on the performance. It is because the SSTable is replicated over three servers and is also cached on p-nodes. When a server is lost, SSTable access can be serviced by the local cache of the p-node or replicas on the other SSTable servers. The killed SSTable server is restarted at 1′00″ and it still has little impact on the performance.

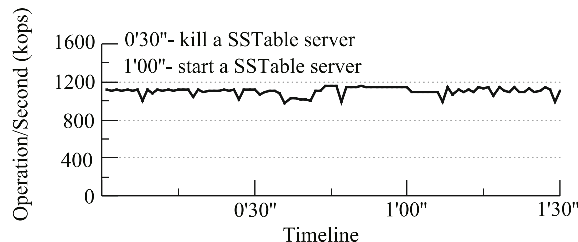

Fig. 16 Failure of SSTable Server

In Fig. 17, the leader of three Memtable servers is killed at 0′30″. Since all p-nodes can not communicate with the leader any more, they can not synchronize their local $\mathcal{B}_m$, extend leases or access the Memtable. As a result, the throughput drops to zero. It takes about 8 seconds to elect a new leader among the rest living Memtable servers. After that, each p-node establishes connections with the new leader. Both read operations and write ones can be serviced again. At 1′00″, we restart the killed Memtable server. It re-joins into the cluster and acts as a follower now. In Fig. 18, a follower of three Memtable servers is killed at 0′30″. Its failure has little impact on the performance, because the follower is not responsible for processing any user requests directly. As long as both the leader and the other follower are still living, the system is able to service both read and write operations as usual. The failed server re-joins into the cluster at 1′00″ and still acts as a follower.

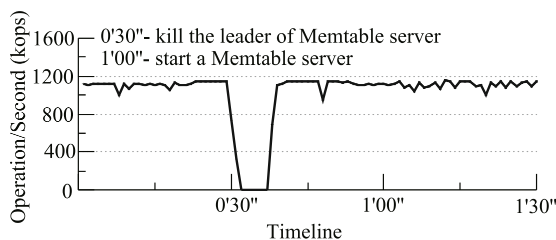

Fig. 17 Failure of Memtable server (leader)

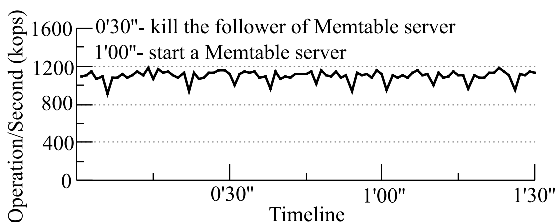

Fig. 18 Failure of Memtable server (follower)

8 Relate works

The design of the log-structured storage can be found in some early works which utilize a write-optimized structure and some read-optimized structure(s) for database storage. Severance and Lohman [18] proposes the *differential files* for storing a large database. It separates a large database into two parts: a differential file and a main file. A consistent and read-only database snapshot is stored in the main file, while all data modifications on the snapshot are consolidated into a relatively small storage area, called differential file. Later, O′Neil et al. [2] proposes the log-structured merge tree. The author exploits a multi-level structure for large database storage. The LSM-tree allows multiple read-only structures used and re-organized multiple structures better than [18]. Both [2] and [18] are used to index records on a single node. And O′Neil et al. [2] is implemented in LevelDB and RocksDB. Storing all data in a single node avoids the performance overhead brought by extra remote data access. However, a single node only provides limited storage capacity. When the database get larger, the performance can be greatly limited by the disk I/O as fewer records could be cached in the memory. In comparison, the precise data access is built upon the distributed LSM-Tree. When the database gets larger, a distributed LSM-Tree is able to scale out its storage capacity by deploying more SSTable servers.

BigTable [3] implements the LSM-Tree in the distributed fashion. In BigTable, the writing part of the LSM-Tree in stored in the memory and the read-only part is stored in the Google file system [5]. Its design is inherited by other distributed NoSQL systems, such as Apache HBase. Some other database systems are built upon BigTable. Percolator [7] and Megastore [6] build their application servers directly on the BigTable. The major drawback of a distributed LSM-Tree is that processing each read operation requires several network communications, which increases the latency. Our work acts as a data access optimization between the query layer and the storage layer of a database system. It helps avoid most unnecessary remote data access.

Some other optimizations are also designed for LSM-Tree. bLSM-tree [8] uses the Bloom filter [9] to reduce disk access on a SSTable, which is adopted in [3]. The method only works well in filtering unnecessary SSTable access. It does not work for the Memtable. Since the SSTable is a read-only structure, its Bloom filter is immutable after being constructed. Hence, there is no consistency issue when caching the Bloom filter on a p-node. In a different, the problem becomes much more difficult considering that the Memtable changes over time. This work proposes methods to synchronize the Bloom filter efficiently when its Memtable receives new updates. Besides, our data access algorithm guarantee the linearizability consistency when using a Bloom filter in filtering Memtable access. Muhammad [19] improves the performance of major compaction so that multiple SSTable files can be merged more quickly. It helps reduce the number of SSTable so that a read operation is required to access fewer SSTables. However, all read operations are still required to access the Memtable even if all SSTables are merged into a single one using the method proposed by [19]. Diff-Index [20] designs secondary index for distributed LSM-tree. The work offers four index update schemas with different consistency guarantee and analyze their performance. It allows a read operation to read a record using its non-primary-key column without scanning the whole table. The proposed method reduces the maintenance overhead of the secondary index at the cost of weakening the consistency guarantee. In a different, our work is mainly designed for querying a record using its primary key. We improve the performance as well as ensuring the linearizability consistency.

In comparison, the precise data access is able to filter most unnecessary Memtable access without weakening the consistence. However, it also has the following disadvantage. It is limited to be used on a local area network, because the lease management requires all server clocks to be well synchronized. Under a wide area network, the clock inaccuracy is too large to make our method work. Besides, it also takes much longer for a p-node to extend a lease from the Memtable server if they are deployed in different regions.

This work is an extended version of [21], which firstly presents the precise data access algorithm for distributed LSM-Tree. However, Zhu et al. [21] does not take data compaction and component failures into consideration when designing the algorithm. In this work, we make the algorithm fault-tolerant. The distributed LSM-Tree can correctly recover the data structures used by the algorithm after node failures happens. In addition, the minor compaction of the LSM-Tree is carefully designed, so that our algorithm work correctly and efficiently even if the LSM-Tree is re-organizing its data storage in the backend.

9 Conclusion

This work presents the precise data access for the distributed LSM-tree. Primarily, it is designed for applications whose

database is more than terabytes in size. Such database is better to be stored in the distributed LSM-tree, and the precise data access help improve its performance in servicing read heavy workloads. For example, some online shopping websites stores TB-sized or PB-sized data and is required to service massive user queries at the peak time. By maintaining low overhead structures among servers, the precise data access help reduce most unnecessary remote Memtable access. Extensive experiments have shown that our solution improves the performance a lot. At the present, the Bloom filter-based mechanism only handles point query efficiently. A future direction is to consider how to reduce unnecessary remote range scan on both Memtable and SSTable.

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

References

1. Chen J C, Chen Y G, Du X Y, Li C P, Lu J H, Zhao S Y, Zhou X. Big data challenge: a data management perspective. Frontiers of Computer Science, 2013, 7(2): 157–164
2. O'Neil P E, Cheng E, Gawlick D, O'Neil E J. The log-structured merge-tree. Acta Informatica, 1996, 33(4): 351–385
3. 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, 2008, 26(2): 4
4. Lakshman A, Malik P. Cassandra: a decentralized structured storage system. ACM SIGOPS Operating Systems Review, 2010, 44(2): 35–40
5. Ghemawat S, Gobioff H, Leung S T. The Google file system. In: Proceedings of ACM Symposium on Operating Systems Principles. 2003, 29–43
6. Baker J, Bond C, Corbett J C, Furman J J, Khorlin A, Larson J, Leon J M, Li Y W, Lloyd A, Yushprakh V. Megastore: providing scalable, highly available storage for interactive services. In: Proceedings of the 5th Biennial Conference on Innovative Data System Research. 2011, 223–234
7. Peng D, Dabek F. Large-scale incremental processing using distributed transactions and notifications. In: Proceedings of USENIX Symposium on Operating Systems Design and Implementation. 2010, 1–15
8. 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
9. Bloom B H. Space/time trade-offs in hash coding with allowable errors. Communications of the ACM, 1970, 13(7): 422–426
10. Herlihy M, Wing J M. Linearizability: a correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems, 1990, 12(3): 463–492
11. Levandoski J J, Lomet D B, Sengupta S. The Bw-Tree: a B-tree for new hardware platforms. In: Proceedings of the 29th IEEE International Conference on Data Engineering. 2013, 302–313
12. Mohan C, Haderle D J, Lindsay B G, Pirahesh H, Schwarz P M. Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. ACM Transactions on Database Systems, 1992, 17(1): 94–162
13. DeWitt D J, Katz R H, Olken F, Shapiro L D, Stonebraker M, Wood D A. Implementation techniques for main memory database systems. In: Proceedings of ACM SIGMOD International Conference on Management of Data. 1984, 1–8
14. Gray J, Helland P, O'Neil P E, Shasha D E. The dangers of replication and a solution. In: Proceedings of ACM SIGMOD International Conference on Management of Data. 1996, 173–182
15. Tang Y, Sun H L, Wang X, Liu X D. An efficient and highly available framework of data recency enhancement for eventually consistent data stores. Frontiers of Computer Science, 2017, 11(1): 88–104
16. Ongaro D, Ousterhout J. In search of an understandable consensus algorithm. In: Proceedings of USENIX Annual Technical Conference. 2014, 305–319
17. Wang D H, Cai P, Qian W N, Zhou A Y, Pang T Z, Jiang J. Fast log replication in highly available data store. In: Proceedings of Asia-Pacific Web and Web-Age Information Managemet Joint Conference on Web and Big Data. 2017, 245–259
18. Severance D G, Lohman G M. Differential files: their application to the maintenance of large databases. ACM Transactions on Database Systems, 1976, 1(3): 256–267
19. Ahmad M Y, Kemme B. Compaction management in distributed key-value datastores. Proceedings of the VLDB Endowment, 2015, 8(8): 850–861
20. Tan W, Tata S, Tang Y Z, Fong L L. Diff-index: differentiated index in distributed log-structured data stores. In: Proceedings of International Conference on Extending Database Technology. 2014, 700–711
21. 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 and Web-Age Information Managemet Joint Conference on Web and Big Data. 2017, 210–218

Tao Zhu 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, transaction processing and distributed system.

Huiqi Hu is currently a lecturer 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.

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.

Huan Zhou is a PhD candidate in the School of Data Science and Engineering, East China Normal University, China. Her research interests include in-memory database system implementation, parallel computing and transaction processing.

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.