Precise Data Access on Distributed Log-Structured Merge-Tree

Tao Zhu[1], Huiqi Hu[1(✉)], Weining Qian[1], Aoying Zhou[1], Mengzhan Liu[2], and Qiong Zhao[2]

[1]School of Data Science and Engineering,
East China Normal University, Shanghai, China
[2]Bank of Communications, Shanghai, China
tzhu@stu.ecnu.edu.cn
{hqhu,wnqian,ayzhou}@dase.ecnu.edu.cn
{liumengzhan,qiongzhao}@bankcomm.com

Abstract. Log-structured merge tree decomposes a large database into multiple parts: an in-writing part and several read-only ones. It achieves high write throughput as well as low read latency. However, read requests have to go through multiple structures to find the required data. In a distributed database system, different parts of the LSM-tree are stored distributedly. Data access issues extra network communications for a server in the query layer to pull entries from the underlying storage layer. This work proposes the precise data access strategy. A Bloom filter-based structure is designed to test whether an element exists in the in-writing part of the LSM-tree. A lease-based synchronization strategy is used to maintain consistent copies of the Bloom filter on remote query servers. Experiments show that the solution has 6x throughput improvement over existing methods.

Keywords: data access, distributed system, consistency

1 Introduction

Log-Structured merge tree [5] organizes all data entries in multiple components: a Memtable and several SSTables, following the notations used in [8]. 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. Data in the Memtable are migrated into a SSTable in batch. It has been widely adopted by distributed storage systems such as BigTable [8], where the Memtable and SSTables are kept in the main memory and distributed file system(e.g. GFS [7]) respectively.

Systems using log-structured storage offer excellent read/write performance but lack some important features. Thus, some database systems (e.g. Megastore[10] and Percolator [9]) choose to 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. A problem

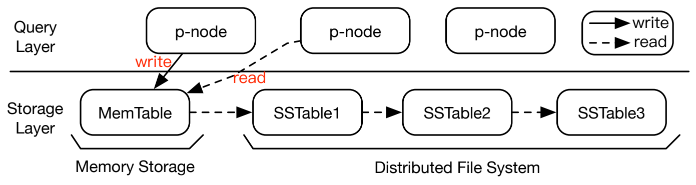

Fig. 1. Data storage and access on distributed LSM-tree

is that data access on a distributed LSM-tree issues many useless communications. A read operation has to iterate over the Memtable and all SSTable until locate the required data item. The dedicated item only exists in one structure and accessing all other structures is useless.

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. In summary, we make the following contributions: 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 maintain a copy of the Bloom filter of Memtable and ensure read consistency when using the copy; 3) A data access algorithm is designed to support a p-node in determining the right structure for data access.

2 Relate Works

[5] proposes the log-structured merge tree. The author exploits a multi-level structure for large database storage. BigTable [8] extends such mechanism in distributed system. It keeps the write-optimized index in main memory while the read-only files in GFS [7]. Percolator [9] and Megastore [10] build their query servers directly on the BigTable. Our work acts as a data access optimizations between the query layer of a database system and the underlying storage layer.

Some other optimizations are also designed for log-structured storage. [8] relies on data compaction to merge multiple SSTables together and reduce the number of SSTable to be visited. Muhammad [12] does performance analysis on the overhead of data compaction and proposes some improvements. bLSM-tree [11] uses the Bloom filter [2] to reduce SSTable access, which is adopted in [8]. In a different, our work is able to filter access to the Memtable. Besides, our technique is designed for distributed system to reduce the network communications between the application servers and the storage layer.

3 Preliminary

Storage Model. A typical structure of a distributed log-structured storage system is illustrated in Figure 1, with a Memtable, several SSTables and p-nodes.

Memtable is the in-memory structure which services for data reads and writes. To ensure durability, redo log entries [4] are forced into durable storage for recovery purpose. Each write operation firstly flushes its redo log entries into the disk, after which, its modifications are applied into the Memtable. Group

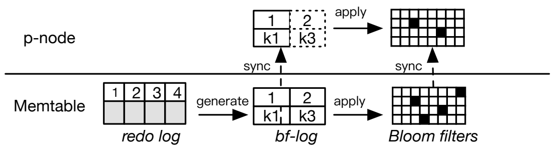

Fig. 2. The Bloom filter maintenance and synchronization

commit [3] is used to improve the disk utilization by combining multiple redo log flushing in a disk write, because existing disk device only offers limited IOPS.

SSTable is the 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 writes. As time goes by, there are several SSTables generated as illustrated in Figure 1, where $SSTable1$ is the latest one and $SStable3$ is the oldest ones.

In Figure 1, to read an entry with key k, a p-node has to go through the 1st Memtable, 2nd SSTable, 3rd SSTable... until seeing the data with k as its key. In addition, Memtable and SSTables are distributedly stored. Hence, a p-node has to issue many remote data access via the network.

Precise Data Access is to let a p-node determine visiting either the Memtable or one SSTable without contacting the underlying storage servers. Let Memtable be m and its owned key set be $\mathcal{K}_m = \{k_1^m, k_2^m, \dots\}$; SSTable be s and its owned key set be $\mathcal{K}_s = \{k_1^s, k_2^s, \dots\}$. Given a query key k, a p-node is required to answer whether $k \in \mathcal{K}_m$ or $k \in \mathcal{K}_s$ stands. 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 2). But, answering whether $k \in \mathcal{K}_m$ is of much more difficulties. Hence, we aim at determining whether Memtable access is necessary for a read operation. There is no essential difference between one SSTable or multiple. We assume there only one SSTable in the following.

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. 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 synchronization. Potential difference between the primary Bloom filter and its copies tends to lead to inconsistency read. Understanding these difficulties, we present solutions in the following sections.

4 Entry Existence

Figure 2 illustrates how to maintain and synchronize a Bloom filter ($\mathcal{B}_m$) for the Memtable. When data writes happen on the Memtable, redo log entries are prepared. Before flushing a group of redo entries into disk, updates on $\mathcal{B}_m$ are generated from redo entries based on the policy in section 4. These updates act as the modification log entries for the $\mathcal{B}_m$ (short for bf-logs). 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.

Maintenance. The Memtable can be probably changed by the following types of operations: *insert*, *update* and *delete*. Considering an entry e with key k:

1. Update operation modifies an existing record entry e. (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 a previous operation has added k into $\mathcal{K}_m$ and handled its update on $\mathcal{B}_m$, the current one does nothing.

2. Delete operation is treated as special *update*, which adds an deleted flag for k. A read operation checks whether the entry is deleted via the flag.

3. Insert operation writes an 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 e can only be found on 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 $\mathcal{B}_m$ should be already processed by one previous *delete* operation. Therefore, we do not need to modify $\mathcal{B}_m$ again.

In summary, $\mathcal{B}_m$ is only modified when an entry is newly created on the Memtable by a *update* or *delete* operation.

Data Access based on $\mathcal{B}_m$. Considering the query $q(k)$ in Definition 1, we denote b_m (b_s) is 1 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. Based on the maintaining policy of $\mathcal{B}_m$, we locate an entry using the following rules.

(1) If $b_m = 0$ and $b_s = 0$, then e is either non-existing or newly inserted into Memtable. A p-node will access Memtable.
(2) If $b_m = 0$ and $b_s = 1$, then e never receives any modification after it is written into the SSTable. A p-node will directly access the SSTable.
(3) If $b_m = 1$ and $b_s = 1$, e is stored on SSTable at first and then get modified. A p-node should visit the Memtable to read the entry.
(4) $b_m = 1$ and $b_s = 0$ is not possible under the maintaining policy for $\mathcal{B}_m$. It only appears when $\mathcal{B}_m$ sees a false positive. In that case, it actually has $b_m = 0$ and $b_s = 0$. Hence, a p-node takes the Memtable as the destination.

As the Bloom filter contains false positives. An entry may not exists in Memtable or SSTable even if $\mathcal{B}_m$ or $\mathcal{B}_s$ confirms its existence respectively. When the entry is not returned by the first access, a p-node access the rest structure to handle any potential false positive.

Lightweight Synchronization. A p-node synchronize the remote $\mathcal{B}_m$ to its local by pulling bf-logs from the Memtable server and replaying these entries. 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 [6].

Each bf-log is indexed by a monotonically increasing serial number. The Memtable server keeps the newest bf-logs in a circular buffer. A p-node pulls bf-logs from the remote by sending the largest serial number N ever received.

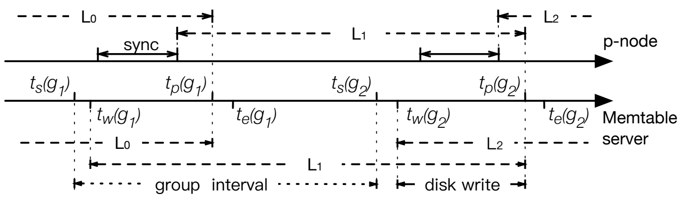

Fig. 3. Group commit and lease management

The Memtable server replies with all bf-logs whose serial number is bigger than N. As the circular buffer has limited memory, new bf-logs may overwrite the oldest ones. The $\mathcal{B}_m$ is sent to a p-node when some required bf-logs are missing.

5 Consistence

A copy of $\mathcal{B}_m$ on a p-node may fall behind the primary one. As a result, a p-node may miss some newly committed entries and suffer from inconsistent read. We present a lease-based solution and use Figure 3 to explain the design.

Group Commit. The Memtable commits write operations with following steps: 1) Generation. Redo entries are buffered in memory. They are flushed into the disk in a fixed period, called the *group interval*, e.g. *from* $t_s(g_1)$ *to* $t_s(g_2)$. 2) Start phase begins at a time $t_s(g_x)$ with a group of redo entries formed. Then, bf-logs are generated and applied into $\mathcal{B}_m$, e.g. *from* $t_s(g_1)$ *to* $t_w(g_1)$. 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* $t_p(g_1)$, which generally several milliseconds to finish under the hard disk driver. 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, e.g. *from* $t_p(g_1)$ *to* $t_e(g_1)$. After that, the group ends at a time $t_e(g_x)$.

Invariance. 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 $t_w(g_1)$ to $t_s(g_2)$ and Memtable is invariant from $t_e(g_1)$ to $t_p(g_2)$. With the temporary invariance of Memtable and $\mathcal{B}_m$, a lease-based mechanism can be designed to ensure the read consistency when a p-node uses a copy of $\mathcal{B}_m$ in data access.

Lease Definition. 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) and a expiration time t_x, and guarantees that for each entry in the Memtable, its bits are correctly set in $\mathcal{B}'_m$ based on the maintaining policy before t_x is reached. It is safe for a p-node to use $\mathcal{B}'_m$ before t_x is reached.

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 $t_w(g_1)$.

Correctness. Between $t_w(g_1)$ and $t_p(g_2)$, the Memtable has two versions while $\mathcal{B}'_m$ includes all bf-logs from g_1 and all previous ended groups. 1) Before $t_p(g_1)$, the Memtable m_0 contains data entries committed by all groups end in prior to g_1. It is safe to use $\mathcal{B}'_m$ because all bf-logs generated by these groups have been applied in $\mathcal{B}'_m$. On the other hand, $\mathcal{B}'_m$ also contains bf-logs from g_1. Though data entries created by g_1 are not included in m_0 at present, a p-node would still be directed to the Memtable when reading them. Such reading can be viewed as a false positive and does not lead to consistency problem. 2) After $t_p(g_1)$, the Memtable m_1 contains data entries committed by g_1 and all previous ended groups. Now $\mathcal{B}'_m$ is the exact structure for m_1.

Two successive leases have overlap in the time-line. A new lease is available for acquisition before the in-using one is going to be expired. In the overlap, both their $\mathcal{B}_m$ work correctly in accessing Memtable. The proof is straightforward.

Lease implementation. 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. The t_x can be any time before the next group publishes, i.e. $t_p(g_{x+1})$. However, the timestamp is not known in advance, but can be inferred by adding the *current time*, the *group interval* and *disk writing time* together (e.g. L_1 in Figure 3). Local processing time, e.g. from $t_s(g_x)$ to $t_w(g_x)$, is ignored as it is very short. Group interval can be given by system configuration. Disk write time can be estimated from the time used for previous groups.

Commit Wait. Since t_x is inferred, it can be smaller or bigger than $t_p(g_{x+1})$. 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*. To avoid *commit wait*, we prefer to use the lower bound of the estimated disk write time in determining the t_x.

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. Typically, a p-node tries to acquire a new lease when the in-using one will be expired in 400 us.

A p-node checks whether the Bloom filter is usable by confirming that the *current time* is smaller than the expiration time of the lease. A problem is the time deviation between servers. PTP [1] can be used to synchronize server clocks, which achieves less than 50 us under a local area network. A p-node infers the time of a remote server by adding its local time with the largest deviation.

6 Experiment

The experiments use 15 servers, equipped with two 2.00 GHz 6-Core processors, 192GB DRAM, connected by 1 Gigabytes switch. The Memtable is stored on a server, the SSTable is shareded over 3 servers. The rest deploy p-nodes. All experiments use the YCSB benchmark with 1 million records in the database. 95% records are stored in the SSTable, and records are accessed in uniform distribution. The workload contains unlimited read requests and 10K writes per second. Three methods are evaluated and compared. (1) *NDA* is the basic access method in LSM-tree. (2) *BDA* maintains Bloom filter of SSTable to prone useless SSTable access. (3) *PDA* is the method presented in this work. Performance are evaluated by read operations processed per second (ops).

Concurrency. Figure 4 shows the performance of different methods by varying the number of clients connected with the system. Overall, PDA has the best performance under all cases. It reaches about 1100k ops when 450 clients are

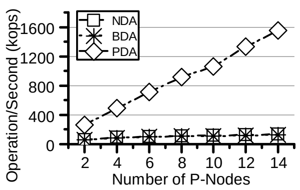

Fig. 4. Scalability.

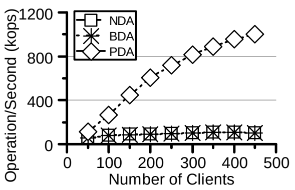

Fig. 5. Concurrency.

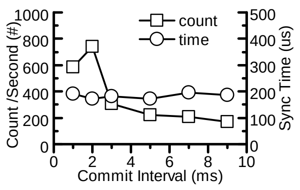

Fig. 6. Synchronization.

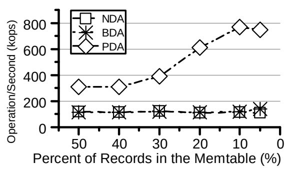

Fig. 7. Storage.

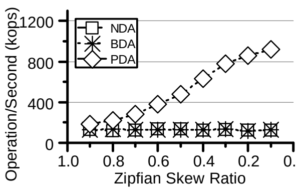

Fig. 8. Skewness.

used, which is about 6 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. Secondly, NDA and BDA share similar performance because the SSTable is well merged and cached on each p-node. Reducing SSTable access does not contributes to performance.

Scalability. Figure 5 evaluates performance by varying the number of p-nodes connected with storage servers. 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 maintenance and 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. NDA and BDA still show similar performance due to the same reason discussed in above.

Synchronization Overhead. Figure 6 shows the synchronization time and frequency by varying the group interval. It always takes about 200us for a p-node to extend a new lease. 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 because a p-node gets a longer lease. An exception is when 1ms group interval is used. When 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.

Storage Distribution. Figure 7 shows the performance by varying the percentage of records stored 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 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 deployment, Memtable does not contain a large percent of records.

Skewed Access Distribution. Figure 8 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. 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. With θ goes down, performance of PDA increases.

7 Conclusion

This work presents the precise data access mechanism for distributed LSM-tree style storage. By maintaining low overhead structures among servers, our design can reduce unnecessary remote Memtable access significantly. Extensive experiments have shown that our solution improves the performance a lot.

Acknowledgements

This is work is partially supported by National Hightech R&D Program(863 Program) under grant number 2015AA015307, National Science Foundation of China under grant numbers 61332006, 61432006 and 61672232, and the Youth Science and Technology- Yang Fan Program of Shanghai (17YF1427800). The corresponding author is Huiqi Hu.

References

1. PTP. https://en.wikipedia.org/wiki/Precision_Time_Protocol
2. Bloom, B.: Space/time trade-offs in hash coding with allowable errors. CACM. 13, 422–426 (1970)
3. DeWitt, D. and Katz, R., et. al: Implementation techniques for main memory database systems. In: SIGMOD, pp. 1–8, (1984)
4. Mohan, C. and Haderle, D., et. al: ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. TODS. 17, 94–162 (1992)
5. O'Neil, P., Cheng, E., Gawlick, D. and O'Neil, E.: The log-structured merge-tree (LSM-tree). Acta Informatica. 33, 351–385 (1996)
6. Gray, J., Helland, P., O'Neil, P., Shasha, D.: The dangers of replication and a solution. In: SIGMOD, pp. 173–182 (1996)
7. Ghemawat, S., Gobioff, H., Leung, ST.: The Google file system. In: SOSP, pp. 29–43 (2003)
8. Chang, F., Dean, J., et. al: Bigtable: A distributed storage system for structured data. In: OSDI, pp. 4:1–4:26 (2008)
9. Peng, D., Dabek, F.: Large-scale Incremental Processing Using Distributed Transactions and Notifications. In: OSDI, pp. 1–15 (2010)
10. Baker, J., Bond, C., et. al: Megastore: Providing scalable, highly available storage for interactive services. In: CIDR, pp. 223–234 (2011)
11. Sears, R., Ramakrishnan, R.: bLSM: a general purpose log structured merge tree. In: SIGMOD, pp. 217–228 (2012)
12. Ahmad, M., Kemme, B.: Compaction management in distributed key-value datastores. In: PVLDB, pp. 850–861 (2015)