NIOSIT: Efficient Data Access for Log-Structured Merge-Tree Style Storage Systems

Bing Xiao
Institute for Data Science and Engineering
East China Normal University
bingxiao@stu.ecnu.edu.cn

Jinwei Guo
Institute for Data Science and Engineering
East China Normal University
guojinwei@stu.ecnu.edu.cn

Weining Qian
Institute for Data Science and Engineering
East China Normal University
wnqian@sei.ecnu.edu.cn

Huiqi Hu✉
Institute for Data Science and Engineering
East China Normal University
hqhu@sei.ecnu.edu.cn

Aoying Zhou
Institute for Data Science and Engineering
East China Normal University
ayzhou@sei.ecnu.edu.cn

ABSTRACT

Recent years, the log-structured merge-tree(LSM-tree) style storage has been widely adopted in distributed data storage systems(e.g. Bigtable and HBase) and commercial database systems(e.g. OceanBase, Cassandra, SQLite, etc.) to provide both large-volume storage capacity and high-performance data updates. Write operations become easier as the LSM-tree style storage avoids writing in place by updating a data copy in memory. However, read operations are affected as it requires an additional step during a data compaction to check if there exists the newest update of data record in memory, which brings many of costly empty reads in real usage since the volume of immutable data is pervasively far more massive than incremental delta data. To address this issue, we design a new network request processing mechanism to allow data access being processed in an auxiliary lightweight network communication IO thread. And a Bloom filter is incorporated with the network IO thread to effectively filter out the empty reads. We also analyze the efficiency advantage of the mechanism and introduce its detailed implementation based on the well-known OceanBase system. Experimental study using the YCSB benchmark demonstrates the proposed mechanism can significantly achieve 20 percent to 30 percent better performance than existing method.

CCS CONCEPTS

•**Information systems** →**Data access methods;** *Point lookups;*

KEYWORDS

Data Access, Log-Structured Merge-Tree, Network IO Processing, Bloom Filter

ACM TUR-C '17, May 12-14, 2017, Shanghai, China
© 2017 ACM. ISBN 978-1-4503-4873-7/17/05…$15.00
DOI: http://dx.doi.org/10.1145/3063955.3063977

1 INTRODUCTION

Data access in memory has demonstrated significant advantage over disk by providing millions of times higher performance. As a result, database systems [9, 12, 18] that place all data in memory become more and more prevalent. However, with the growing amounts of data, data cannot be entirely fit into memory for many applications. For instance, some distributed data or file storage systems like Bigtable [6] and HBase [1] came out to provide pegabytes of data storage ability as well as the high performance lightweight transactional data access. Some commercial application also requires to manage billions of data records and provide low-latency ACID transaction services. One typical example is OceanBase [2], developed by Alibaba to provide transaction processing on the Chinese largest e-commerce platform.

The mechanism behind Bigtable, HBase and OceanBase is their LSM-tree style storage(for ease of expression, we simply call the LSM-tree style storage as log-structured storage in this paper), which adopts distinct hardware and data structure to store data. Many other systems such as Cassandra [10], SQLite [3] and Dynamo [8] also use this structure. The strategy is regarded as a compromise choice to make balance of providing unlimited storage capacity and high performance data updates. Typically, all data updates as well as deletes within a period are not written in place but maintained together in memory as *delta updates*. The data remain unchanged are called *static data*(usually stored in disk). At a certain time point, all the static data will be replaced with their newest delta updates(if exist) through a uniform compact operation. The efficient performance of data writes is the highlight of log-structured systems as it only operates an in-memory data structure.

On the other hand, it is an inherent limit to cost an additional step for read operation in LSM styled storage. It first checks the delta data in memory, if the object is not detected as in memory, then a second step is required to check the static data in disk. It becomes even costly in real-world scenario, where concurrent data access requests are processed through certain thread pool [4, 15]. Once multiple read requests are sent from clients, the sever which contains the delta data receives and push them into a queue. Then the thread pool will provide available work threads to process the requests popped from the queue. As the number of work thread is limited, it is common to have the additional queueing time if the number of concurrent requests is relatively large.

In this paper, we study a faster request processing method for the LSM-tree style storage system. We target at the storage structure that delta data and static data are separately stored and requests of data access are organized by thread pools. Typically, we observe that most read requests only access static data since delta data only occupy a very small percentage of the whole dataset. By pushing those read requests into the queue, it causes obviously extravagant *empty reads* on delta data. Therefore, we design a new network request processing mechanism to allow requests being processed in an auxiliary lightweight network communication IO thread. A Bloom filter that describes the allocation of delta data is incorporated with the network IO thread to filter out those empty reads before they are pushed into the queue. Such design facilitates all the read requests: First, for the requests that only access static data, it could be processed in an extremely short time; Second, it shortens the read request accumulated in the queue, and significantly reduces the overhead of enqueue, dequeue, and context switching between the work thread. Notice that some systems directly use Bloom Filter in client to avoid empty read [6, 19]. It is an orthogonal method as (i) in some real world use cases, the storage systems have to offer complete support for data access, where client application logic is totally separate with storage(i.e. unaware of data allocation); (ii) most existing systems simply use asynchronized Bloom Filter between clients and storage as read access for non-newest data is allowed. In our study, the Bloom filter and delta data are on the same node, they are synchronized all the time to provide consistent read access on the newest data. An optimized variant also runs in experimental anlysis. In the variant, a Bloom filter for specific version of delta data can be transmitted periodically to other nodes containing static data, which avoids some read requests to the single delta data node.

Specifically, we summarize the following contributions proposed in the paper:

- We proposed a network IO solved-in-time(*NIOSIT*) method to allow requests to be directly processed in IO thread;
- We combined *NIOSIT* with Bloom Filter to check empty reads to transaction node for delta data;
- We implemented the *NIOSIT* method in OceanBase, a commercial database that employs log structure storage.
- Extensive experiments on microbenchmark demonstrated 20 percent to 30 percent improvement that *NIOSIT* offers.

The rest of the paper is organized as follows. In section 2, we revisit the LSM style storage and outline the motivation and considerations behind the design of our method. In section 3, we explain the framework of *NIOSIT*. Therotical analysis and technical characteristics are introduced in Section 4. Section 5 covers the detailed implementation of our method in OceanBase system. Experimental results are analyzed in section 6. Section 7 reviews the related works. We conclude our work in section 8.

2 PRELIMINARY

2.1 Log-Structured Storage

The log-structure was came up with the effort to support higher writing throughput over very large size of web data, besides scalability. It changes the traditional in-place update. In this architecture, recent delta data are written to memory supporting single-row transactions while large amount of static data are stored in disk. Since there are many kinds of online websites having more data remaining unchaged than updating data during a certain time period, the storage for static data is usually far more than delta data. When these two parts are further seperated to different physical server machines, as delineated in Figure 1, cross-row and cross-table transactions could be handled on the transaction node gathering all updates.

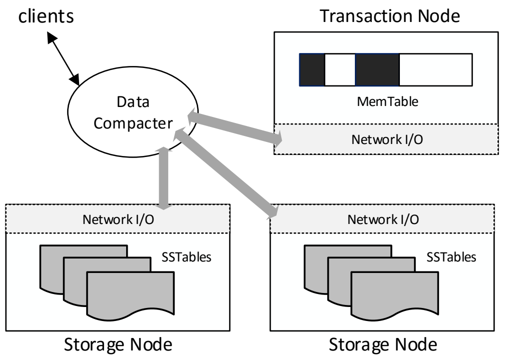

Figure 1: Physically seperated log-structured architecture

In this kind of system architecture, the static immutable data which we call static data are stored in multiple static data servers(SNode, Storage Node) in a mode similar to distributed file system's, and the updated data which we call delta data are stored in the transaction processing severs(TNode, Transaction Node). Each query compacts target static data with delta data in the single transaction node through data compactor and then return to the user. It is practically welcomed by online shopping website, since the consistency is guaranteed without complicated distributed transanction processing, e.g. two-phase commit protocol.

Generally, log-structured architecture provides a number of interesting behaviors: writes are always fast done in memory regardless of the size of dataset (append-only), and random reads are either served from memory or require a quick disk seek. Once the MemTable reaches a certain size, it is flushed to disk as an immutable SSTable. However, we will maintain all the SSTable indexes in memory, which means that for any read we can check the MemTable first, and then walk the sequence of SSTable indexes to find our data.

2.2 Data Access in TNode

Nevertheless in the physically seperated log-structured architecture described in Figure 1, MemTables are all maintained together in transaction node but seperated with SSTables. When large amount of concurrent user queries and writes are put on the transaction node at the same time, the pressure may be too heavy to tolerate. At this moment, the node could be referred to a queueing system. Under this situation, we focus on the data access inside the transaction node, which is illustrated in Figure 2.

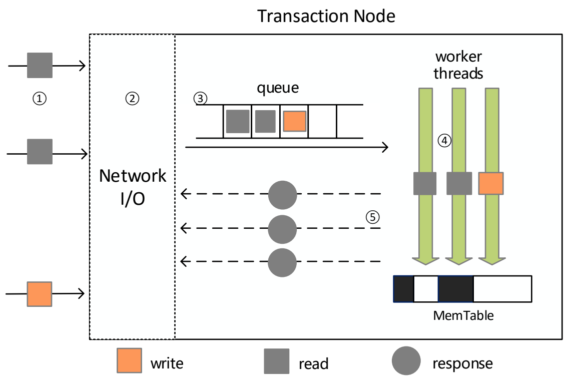

Figure 2: Data access in the transaction node

Generally in our system model, all the requests including queries from other nodes come through network IO and wait in queue for the worker thread to pop and handle, where server nodes could be recognized as queueing systems. Only when the worker thread finish reading or writing MemTable, which corresponds step 4 in Figure 2, response with result and data could be returned in step 5.

Since we found that in most application scenarios like those in Web, there is usually not much data changed in a short time, while online transaction data could be up to ten folds. That is to say, there are a lot of reads being sent to the delta data node but carrying nothing back. At the same time, this kind of reads can spend much more time on waiting in queue than being processed by worker thread. In order to reduce the context switching from network IO thread to worker thread and the waiting time in queue of these requests, we originally desire to support them directly being processed in network IO thread and response immediately.

However in the existing mechanism, whether a request is such an empty read is not decided until it really get nothing. We choose Bloom Filter, a space-and-time efficient data structure to determine the existence of a certain read key in network IO thread. Since Bloom Filter has false positive but no false negative, the correctness can be guaranteed.

2.3 Bloom Filter

A standard Bloom filter [5] is used to represent a series of elements of a subset S in a big universe U. The number of elements in S(called members), n=| S |, is usually much smaller than the size of the universe, N=| U |.

$$S = \{e_1, e_2, ..., e_n\} \tag{1}$$

A Bloom filter itself is an m–bit array cooperating with a hash family H of k independent uniformly random hash functions. These hash functions map to range {1, 2, ..., m}. For each element e in S, the $h_i(e)$-th bits in the Bloom filter after hash family's mapping are set to '1'. For a membership query about whether an element x is in the set S, the answer is 'yes' if all the mapped bits of x are '1' and 'no' otherwise.

$$H = \{h_i \mid i = 1, 2, ..., k\} \rightarrow \{1, 2, ..., m\} \tag{2}$$

The query of a standard Bloom filter has no false negative but false positive. That is to say, if an element is a member, the query always returns 'yes', while if not, there is some possibility that it also returns 'yes'. Assuming that the hash functions are perfectly random, the probability of the false positive for a non-member element can be calculated. After all the n members are hashed to the Bloom filter, the probability that a specific bit remains '0' is simply:

$$p = \left(1 - \frac{1}{m}\right)^{kn} \approx e^{\frac{-kn}{m}} \tag{3}$$

Then the probablity of a false positive is:

$$P_{FP} = (1-p)^k = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - e^{\frac{-kn}{m}}\right)^k \tag{4}$$

To obtain the best performance of the Bloom filter, we would like to choose k that minimize the false positive probability. Taking the derivative of P_{PF} to be zero shows the optimal solution:

$$\begin{aligned} k &= \frac{m}{n} \ln 2 \\ P_{FP} &= \left(\frac{1}{2}\right)^k = 2^{-\frac{m}{n}\ln 2} \approx 0.6185^{\frac{m}{n}} \end{aligned} \tag{5}$$

The space usage of a standard Bloom filter is $O(m)$.

3 THE NIOSIT METHOD

In this section, we describe the mechanism of our work–*NIOSIT*, which we have been implemented in OceanBase [2]. Compared to the traditional approach, *NIOSIT* allows empty read requests filtered out in network IO threads, which saves the expense of context switching and blocking time in queue. More precisely, we employ Bloom Filter to recognize empty reads from read type, rather than searching the MemTable to take the risk of occupying network resources and cause thread suspended animation.

3.1 Framework

In our system model, a task like a read request in the transaction processing server node with nothing back is short in running time. The short task can often be processed quickly and reply to its clients (e.g., Compacter) as early as possible. For those long running tasks like long read-write transactions, they need to wait for the lock on shared resources so they can be added into a queue and then passed to specific long-task processing threads.

As shown in Figure 3 , each IO thread has an unique event loop registering a listening watcher and a thread watcher to listen a socket. The IO thread pool illustrated at the left part of the sever mainly takes charge of the use of communication memory. The read requests from clients are divided into two types: $P_{r\_nodelta}$ which represents empty reads, and P_{r_delta} for matter reads. The empty reads represent those requests with searching key that has

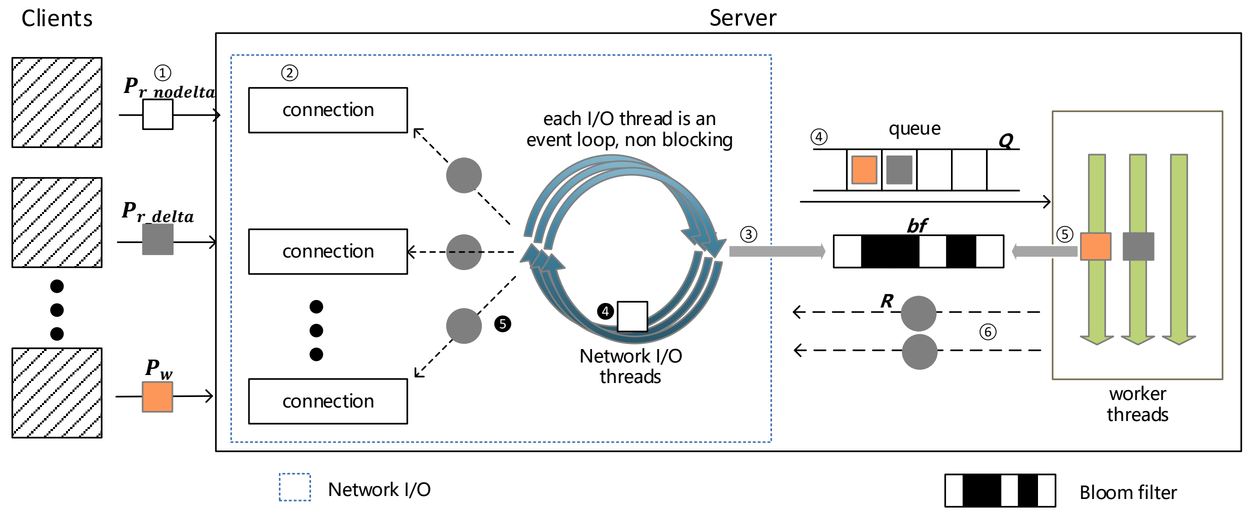

Figure 3: Framework of *NIOSIT*

no delta data, while matter reads represent the opposite. We make empty read filtration be processed in IO thread by using Bloom filter *bf*, and push others to the waiting queue *Q*. We will depict relative details in the Section 5.

Our multi-IO-thread design with event loop underneath could solve the problem of context switching when receiving and forwarding small requests. The concurrent processing pattern of requests enhances the network ability. When starting, the server or client creates network IO, configures the thread pool's capacity and allocates other resources.

3.2 Interfaces

Interfaces of our log-structured system architecture are similar to but not just those in BigTable:

- **start(clientID, transactionID)**
  A client is corresponding to a session. A transaction is started in its's host session to do request and guarantee ACID.
- **write(key, columnName, timestamp, newValue, operation, transactionID)**
  Insertion and deletion are input as an operation tag, both of which are operated on a certain column with a certain key that determines a certain cell in MemTable.
- **read(key, columnName, timestamp, transactionID)**
  A read is manipulated on a merged view of the sequence of SSTables and the MemTable.
- **commit(clientID, transactionID)**
  After a trasanction commits, the modification is supposed to have been written to commit log and MemTable. Then periodically compaction will happen automatically.

3.3 Request Processing

We have introduced application program interfaces (API) above, which correspond different requests in our database system. We use P_s, P_c, P_w and P_r to represent the type of request of the interface **start**, **commit**, **write** and **read** respectively. In this subsection, we will describe the process flow of these requests on TNode using *NIOSIT*.

When TNode is started, it maintains a Bloom filter bf, which is configured to represent the set of data in the active MemTable. This data structure is globally shared in TNode, i.e., the network IO threads and worker threads can access the same bf. The bf is attached with a timestamp $bf.ts$, which represents the **commit-timestamp** of the last committed transaction impacting the bf. When the size of MemTable increases and reaches a threshold, the MemTable is frozen and converted to an SSTable, and a new MemTable is set up. At the same time, the bf is also frozen, and a new bf is created for the new MemTable.

In *NIOSIT*, the key point is the inside process of network IO thread. As illustrated in Algorithm 1, when the TNode receives a request packet, its network IO thread gets the header from the packet first. Since the type of the request and the object ID—which is the key of the record required by the client—is included in the header, the IO thread can handle the package in accordance with the request type. If it is not a read, the package will be pushed into the waiting queue; otherwise, the request will be pre-processed in the IO thread. More precisely (if the request is a read), the IO thread checks whether the **start-timestamp** $P.ts$ of the transaction, included in the package, is valid. If the timestamp is invalid (e.g., 0), the IO thread sets $P.ts$ to $bf.ts$, which may be used in the worker thread. Then, the network IO employs bf to query membership of read key in line 8. If the query returns false, it means that there is no delta data in MemTable for the search key and a *none* result is

Algorithm 1: Process in network IO thread

```
Function IOProcess(P) /* algorithm as an IO process function */
    Data: Network Request Packet, P
    Result: Network Response Result, R
    /* decoding the packet header are omitted here */
    if $P.type == P_r$ then /* each packet has a code to tag the type */
        get the object key $O_{id}$ from the header;
        if $P.ts == 0$ then
            $P.ts = bf.ts$;
        /* according to the filtering result to decide whether to response directly or wait in the queue */
        if $bf.query(O_{id})$ then
            push the packet to Q;
        else
            add none to R;
            return R;
    else
        push the packet to Q;
```

Algorithm 2: Process in worker thread

```
Function WorkerProcess(Q) /* algorithm as a worker process function */
    Data: Request Queue of the Worker Thread, Q
    Result: Process Result of Request in the Queue, R
    while Q has P do
        get P from Q;
        if $P.type == P_s$ then
            create a new session s for new transaction t;
            use last_ts as the start timestamp $t.ts$;
            add transaction ID $t.id$ and $t.ts$ to R;
        else
            get the session s in accordance with $P.tid$;
            if $P.type == P_w$ then
                lock the request object;
                add the object to the write set $t.ws$;
                add SUCCESS to R;
            else if $P.type == P_r$ then
                use $P.ts$ to read data from MemTable;
                add version object to read set $t.rs$;
                add the data to R;
            else
                if validate(t.ws) then
                    generate and flush a commit log;
                    add $t.ws$ to $bf$;
                    unlock the correponding objects;
                    apply write set $t.ws$ to MemTable;
                    update last_ts and $bf.ts$;
                    add SUCCESS to R;
                else
                    unlock the correponding objects;
                    add ABORT to R;
                free the session s;
    return R;
```

returned; otherwise, the read object may exist in the MemTable, thus the packet is pushed into the waiting queue and the worker threads are triggered to process the packet from the queue.

The mechanism of worker thread—which is illustrated in Algorithm 2—is similar to a conventional database system. When a worker thread gets a package from the waiting queue, it processes the request in accordance with the package type. When the worker processes a **start**, it creates a new session for the new transaction t using *last_ts*, the timestamp of last committed transaction as the t's **start-timestamp** $t.ts$, and assigns a unique transaction ID $t.id$. The **read** and **write** are processed conventionally, e.g., the worker need to maintain the read set $t.rs$ and write set $t.ws$ of the specified transaction t. Note that if a transaction t is committed successfully, the worker needs to insert the object from the $t.ws$ into the bf, and to update the $bf.ts$ with the t's **commit-timestamp** later.

3.4 Correctness

Bloom Filter is a space-and-time-efficient and hash-coding data structure, which is considered as a proper structure to take charge of read key existence check in network IO thread. Because network IO is a kind of relatively precious resources in distributed systems, time-assuming tasks are not expected to be processed in network IO causing blocking even suspended animation of IO thread.

Transaction correctness is typically guaranteed by concurrency control in different isolation levels. Nevertheless the wirte and read procedure themselves are supposed to sustain validity. As depicted in the process in worker thread in Algorithm 2, Bloom filter of delta data is updated immediately after the new data are appended to MemTable in line 21. When MemTable is flushed to disk as SSTable, a new Bloom filter for new MemTable takes up the post. In addition, Bloom filter has no false negative, which ensures that when a read is tagged 'not exist', there is no possibility that it has delta data in MemTable.

3.5 Optimization

What's more, there is also an opposite data accessing direction in log-structured architecture, where an user query come to SNodes for static data first. Then to synchronize the Bloom filter from TNode to SNodes will further reduce the stress of the former. Since SNodes could maintain this Bloom filter of MemTable with a commit timestamp. A read request of which the timestamp is larger than the commit timestamp of MemTable Bloom filter will be directly sent to TNode. While the others will check existence in MemTable through the Bloom filter first.

4 EFFICIENCY ANALYSIS

We have mentioned that the TNode could be recognized as a queueing system, so we are going to apply queueing theory to analyze the efficiency of our work. Queueing theory is an abstract mathematical model of waiting lines to predicate queue lengths and waiting time in a queue system, which is widely used in numerous fields like telecommunications, transporting, computer communication networks, etc. According to the queueing theory [16], a database system configured by managers of CPU, transaction, concurrency control, disk and buffer can be treated as a queueing network model consisting of several M/M/1 systems. Since we focus on the request response performance, we model a request's response time with queueing theory as below:

$$T_R = \frac{1}{\mu - \lambda}, (\lambda < \mu) \tag{6}$$

The model M/M/1, which while assuming Poisson arrivals of customers(requests), allows the service distribution to be exponential in one server. In Formula 6, λ is the average arriving rate of requests and μ is the average service rate for requests.

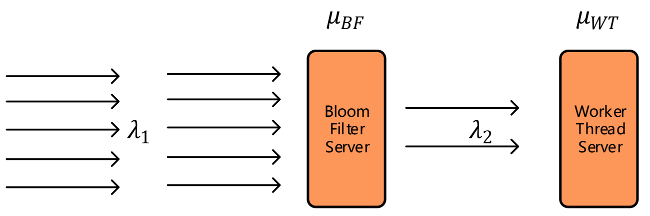

Figure 4: Queueing system view of our efficient data access

In our strategy, the Bloom Filter is considered as an additional server in front of the worker thread server. These two servers work in series, which is illustrated in Figure 4. Ignoring the context switching before worker thread, we can get the native response time T_{origR} without Bloom Filter server as below:

$$T_{origR} = \frac{1}{\mu_{WT} - \lambda_1}, (\lambda_1 < \mu_{WT}) \tag{7}$$

Assuming that Bloom Filter could exactly check out certain α percent empty reads of those reading requests and response the client immediately, we could reason about the average arriving rate λ_2 of worker thread server based on the original one λ_1 :

$$\lambda_2 = (1-\alpha)\frac{1}{T_{R1}} = (1-\alpha)(\mu_{BF} - \lambda_1) \tag{8}$$

and the new average response time T_{newR} of the queueing system could be:

$$T_{newR} = T_{R1} + (1-\alpha)T_{R2} = \frac{1}{\mu_{BF} - \lambda_1} + (1-\alpha)\frac{1}{\mu_{WT} - \lambda_2} \tag{9}$$

Recall that the worker thread serve multiple kinds of requests including read and write, so we have a hypothesis that $\mu_{BF} \gg \mu_{WT}$. Further more, comparing Formula 7 and Formula 9 hoping that T_{newR} is therotically smaller than T_{origR}, we can figure up the relationship between μ_{BF} and λ_1.

For instance, when we assume that $\alpha = 0.8$ and $\mu_{BF} = 10\mu_{WT}$, we will have $\mu_{BF} \in (\lambda_1, 3\lambda_1]$, which is actually satisfied in the real environment.

In a nutshell, when empty read takes a great proportion, we consider it imperative to serve requests discriminatingly for empty reads after Bloom Filter check in network IO thread. Since most of the web business on database systems have a large proportion of read operations, we will focus on improving performance of data access on delta data by our ideas of directly processing in multiple IO threads. All of above shows the feasibility of our work.

5 IMPLEMENTATION

The best way to understand how the method works is to understand how a TCP flow is processed in it. This is described in the following subsections. Since our Bloom filter is implemented through MurmurHash and is configurable to database administrator, related details are omitted here.

5.1 Data Structures

We bring the memory pool strategy of dynamic expansion of memory to manage network IO, connection, message, request and other resources' allocation and release. Resources at all levels are maintained in the form of linked list. This is illustrated in Figure 5. The memory is managed in connections, that is to say, the memory resource is released by connections and resources among connections do not interfere with each other. Each connection has a reference counter. When the reference counter has the value of 0, the connection releases its memory pool. In one more granularity, the release of message is also based on reference counter. When creating a new connection, N+1 memory pools are allocated where N is the number of messages in the connection and the other one is for storing the metadata of the connection itself. Then what matters most is when will the reference counter be changed.

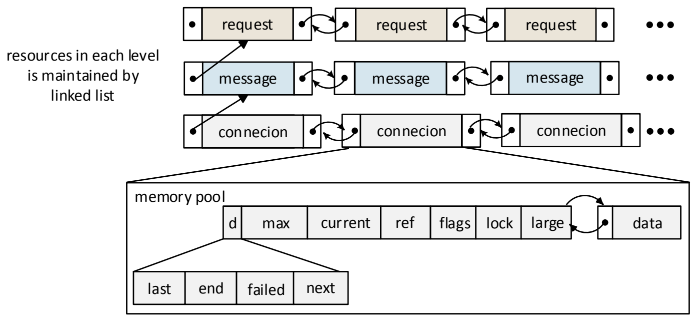

Figure 5: Resources in levels

Then we discuss all the situations that cause the change of reference counter of messages and connections.

For message to be increased by one, first when the response packet is pushed into output buffer of a connection, its reference counter will be increased by one because the result is not returned and the memory pool should not be released. Second, in asynchronous request processing, it's obvious the memory pool could not be destroyed since requests are allocated from message's memory pool. The same reason and result are in the third situation when a worker thread is waked up by a network IO thread. worker thread correspondingly, after the response packet is sent to the client, the cleanup function of message will decrease the reference counter of the message by one. Also when a network IO thread is waked up, the server processes a 'finished' request from the worker thread, so the reference counter will decrease one. However, after IO thread finishes processing a request, it should be checked that whether requests on the message are all finished or not. When it's true that all requests has been finished, the reference counter will be decreased to 0, then the memory pool will be released.

Then for connection, the situation may be simpler. The reference counter of a connection is increased by one when it is processed either in user's processing method in network IO thread or in server's worker thread. And at each time a network IO thread sends response for a finished request, the connection reference counter will be decreased to be 0 and the memory pool is destroyed.

5.2 Network Communication

It is directly perceived that when a SNode sends a request to a TNode, a connection is created then the network processing happens. Therefore, the network processing of our mechanism is going to be introduced by two parts based on node's role, as a client and as a server.

At the beginning of a primary TCP connection, every thread in the thread pool opens its listening port and sets the event loop to listen to the socket accept event. Once an accept event comes, the thread constructs connection and initiates read-write watcher and timeout watcher. At this moment, the event loop on the connection begins to listen to read or write event. When there is readable data, the loop triggers appropriate read callback function to read data and response to the request.

A thread opens its listening port by adding timer event to the listen watcher. If the timer is set to trigger its timeout watcher every 100 milliseconds, its corresponding callback function will use the timer to grab listening lock. Once the lock is grabbed, it converts current listening thread to itself and adds EV_READ. When the thread switches, the watcher will be reset so the triggering frequency of listen watcher can be changed.

After the thread triggers accept operation on the listening file descriptor and accept the socket wiring successfully, it will create a connection. When the file descriptor is readable, read callback function is triggered. At the same time, whether current end is client or server will be judged for different processing methods are needed for these two.

For the server end to process request, firstly it invokes decode function from users to decode the request, i.e., parse data from the buffer of request transforming binary data to network packet transported in the model. Each parsed request will be processed in a callback function in IO thread, where all write requests are pushed into the input queue, which will be sent to worker thread of the server. As for read requests, a check by Bloom filter will be finished first. Then the response will be returned immediately for empty reads. While if not, the request will continue to queue. The processing callback functions of server involve the mechanism of IO thread processing or queueing. The inside event loop determines when to trigger the callback.

For the client end to process response when the output queue is not empty, it begins with invoking encode function from users to write response data to buffer, i.e., transforming network packets into binary data.

6 EXPERIMENTAL EVALUATION

In this section, we conducted an experimental study to evaluate the performance of the database system CEDAR, which is based on OceanBase, using *NIOSIT*. We use the term `NIOSIT` to denote the TNode employing original version of *NIOSIT*, which is described in Section 3.1. Let `NIOSIT-opt` to represent the system using optimized version of *NIOSIT*, which is described in Section 3.5. We compare the performance of `NIOSIT` and `NIOSIT-opt` to two others, described in the followings:

- `PLAIN`: The `PLAIN` system does not adopt *NIOSIT*. In other words, all requests received by TNode are pushed to the waiting queue.
- `WEAK`: All the methods described above can provide strong consistency. The `WEAK` system offers a weaker level of consistency, e.g., a request is not forwarded to the TNode, therefore the static data can only be acquired.

6.1 Experimental Setup

Database configuration: The experiments were conducted on a cluster of commodity machines, each was equipped with the same configuration. Our database system was configured with one TNode and six SNodes, each was deployed on a unique server. Multiple clients concurrency was simulated by threads on the client end. Each server was running an operating system of CentOS release 6.5 and equipped with 2-socket Intel(R) Xeon(R) CPU E5606@2.13GHz CPU, Gigabit Ethernet card, and 96GB RAM.

Bloom filter: The Bloom filter assumes that the outputs of hash functions are perfectly random. Then we had to trade off the randomness and computing cost of the hash functions. Therefore we chose MurmurHash which is widely used in the industry. The Bloom filter used in our experiments was configured with $m = 10,000,000$ and $k = 5$. Note that if the number of unique objects in MemTable is less than $1,000,000$, the false positive probability P_{FP} of the filter will be less than 0.01.

Benchmark: We adopted YCSB [7]—a popular key-value benchmark from Yahoo—to evaluate our implementation. Our main workloads are the YCSB A and B (abbr workload-A and workload-B), which have a read/write ratio of 50/50 and 95/5 respectively. The clients, which request reads and writes to a middleware node with keys chosen randomly using Zipfian distribution, are deployed on the SNodes. The database is preloaded with 10 million records, where about 90 percent are persist on SNodes. In order to avoid the bottleneck of network, the size of each record is limited to 100 bytes.

6.2 Experimental Results

We conclude three groups of experiment results here. In order to show our motivation through data, the first group ran on the original plain network communication model treating read requests

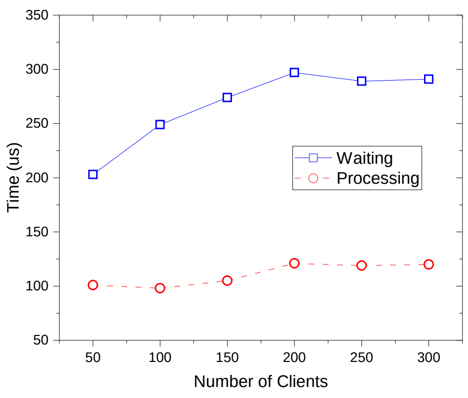

Figure 6: Average time consumption of P_{r_delta}(Waiting, Processing) on TNode in PLAIN

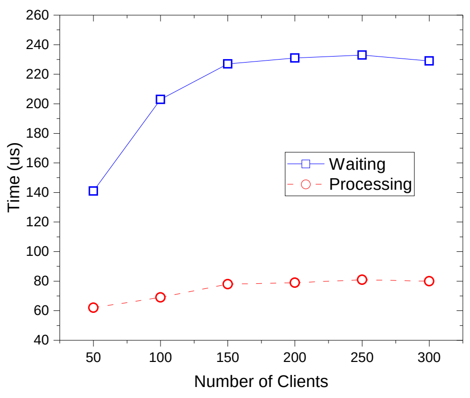

Figure 7: Average time consumption of $P_{r\_nodelta}$(Waiting, Processing) on TNode in PLAIN

equally without discrimination. With respect to show the impact of our work, the following two groups ran on our *NIOSIT* for TNode and for the whole system are given. Each group of these two compares processing mechanisms of plain network communication model and *NIOSIT*.

6.2.1 Breakdown of Read Request Processing. In order to show the data foundation of our problem, the Figure 6 gives the time consumption when the read requests get the delta data in MemTable. We note that, while the number of clients increases, the avarage processing time of this kind of request almost stands around 120 microsecond, while the average waiting time in network I/O reachs a peak value on a certain level. Similiarly, in Figure 7, the experimental data of carry-with-nothing read request to TNode shows the same trend. Nevertheless, the average processing time maintains nearly the half of that of read requests which could fetch delta data. What is noteworthy is that the waiting time of read-none-delta-data requests in network I/O thread stands on a high level which is almost triple the processing time in worker thread.

How to decrease the waiting time in network I/O thread of read requests those carry nothing back from TNode to improve the read performance of a scalable database system under this system model becomes a challenge. We implemented *NIOSIT* that introduced above to solve this problem. The following part will show the effect we make.

6.2.2 Impact of NIOSIT on TNode. Our experiment consists of a microbenchmark on YCSB where each client applies a connection and continually sends write requests 20 times and then sends read requests random times to the database system. This setup guarantees enough delta data in MemTable of a reading request and forms the scenario of a nearly-full request queue in network I/O thread. In that if a slow request of blocking write-type requests is also directly processed in I/O thread, it could result in fake blocking of thread and affect the performance, we don't make comparative analysis on complete write-type workload. The experiments are such that the read requests are divided into two classes in our system model,

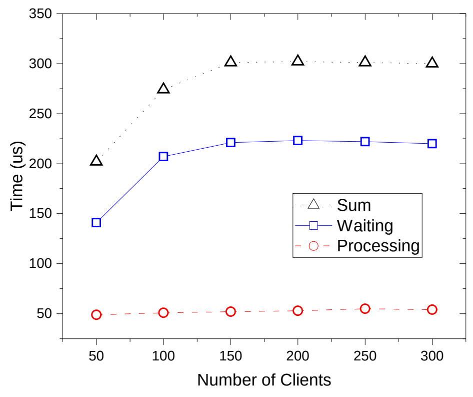

Figure 8: Average time consumption comparison between PLAIN(Sum, Waiting) and NIOSIT(Processing) on TNode

which are reading expected delta updates in MemTable and reading with nothing from the TNode.

Both Figure 8 and Figure 9 illustrate the contrasts of the plain network communication model and *NIOSIT* on TNode through waiting and processing time as well as throughput. Since there is a situation that a request comes to TNode carrying with nothing back but spends much more time on waiting than being processed, *NIOSIT* is implemented allowing this kind of requests to be directly processed in network IO thread, so that the waiting time could be avoided. It is easy to understand that the plain communication network model costs more time than direct processing in IO thread for each request, since waiting time is an extra pay besides effective processing time. Intuitively we can see that the directly processing time in network IO thread is nearly equal to the pure processing time in worker thread that we extract before.

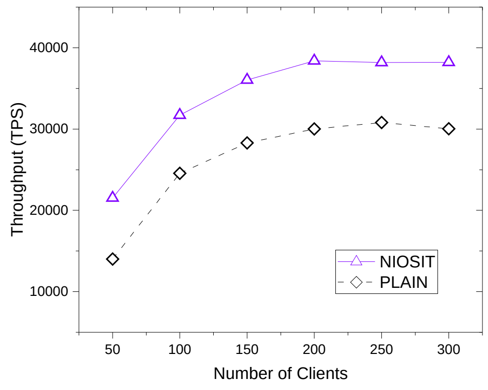

Figure 9: Throughput comparison under YCSB 50/50 workload on TNode

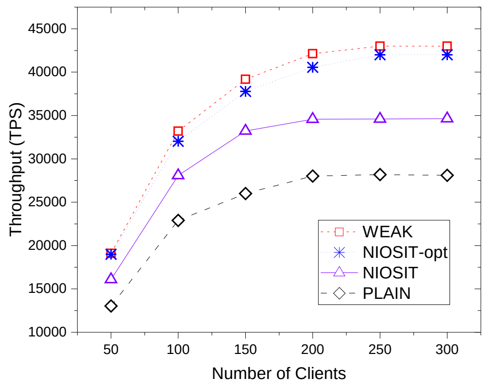

Figure 11: Throughput comparison under YCSB 50/50 workload on system

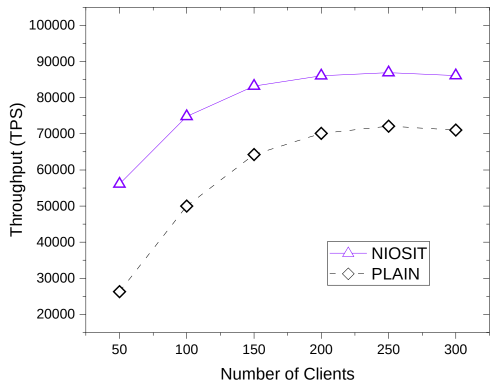

Figure 10: Throughput comparison under YCSB 95/5 workload on TNode

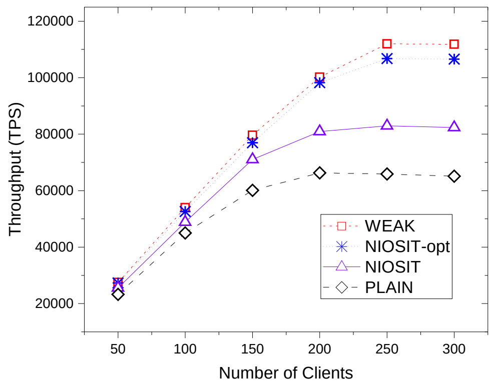

Figure 12: Throughput comparison under YCSB 95/5 workload on system

In addition, when a request queue needs a specific worker thread to handle each request, context switching is an another extra contribution for latency. Figure 10 shows us the improvement on throughput the new network communication makes under more read workload, which is based on the elimination of pure waiting time and the decreases of context switch time. The new throughput achieves nearly 20 percent promotion.

6.2.3 Impact of NIOSIT on System. In order to compare different methods including our optimization proposal, we used an consistency configuration `WEAK` to show the ideal throughput of the whole system. Since clients can only access data from SNodes, if the data from the request are uniform, WEAK can provide a good performance in terms of linear scalability.

Figure 11 and Figure 12 illustrate all of these methods' effect when the number of clients varies. In YCSB 95/5 workload, although the `NIOSIT` outperformed the `PLAIN` by almost 20 percent when number of clients was 300, since the performance of TNode was limited to the capacity of single node, `NIOSIT` had a more than 20 percent performance degradation relative to `WEAK`, which had the highest throughput under workload of 300 clients. Along with our optimization on *NIOSIT*, the throughput of `NIOSIT-opt` could nearly reach the ideal effect. Note that the Bloom filter in compacter in SNodes, which was synchronized with the one in TNode, could filter out almost all the empty reads. Figure 11 shows the same trend illustrated in Figure 12. Since the write operations can be only processed by TNode, the performance in YCSB 50/50 was less than results in Figure 11.

In the two cases, we can conclude that `NIOSIT-opt` has also a good performance in terms of scalability, especially in read intensive workload.

7 RELATED WORK

Bigtable [6], a distributed storage system for managing structured data, is designed to reliably scale to petabytes of data and thousands of machines. However, the distributing way of slicing tables solves the problem of scalability but does not support transactions over tables and rows. LSM Tree(Log-Srtuctured Merge Tree) [13] is the behind manner in which Bigtable uses MemTables and SSTables to store updates to tablets. It gives a way of not-update-in-place strategy but append. Sorted data are buffered in memory before being written to disk, and reads must merge data from memory and disk. Cassandra [10] also absorbs the idea of LSM Tree to achieve storage scalability.

Some optimizations are designed for log-structured storage data access besides compaction. bLSM-tree [17] uses the Bloom filter [5] to reduce disk access on the read-only structures, which is adopted in [6]. Our work stands on a totally different ground of update structrue.

Deuteronomy [11] decomposes functions of a database storage engine kernel into transaction component(TC) and data component(DC), supporting efficient and scalable ACID transactions for cloud data. The TC and DC is transparent to each other. A salient feature of Deuteronomy is the ability for transactions to span multiple DCs. Whereas the transaction processing node does not have any data inside providing concurrency control and recovery, which is different from the LSM architecture we discussed.

RAMCloud shifts the primary locus of online data from disk to DRAM, with disk relegated to a backup/archival role. It emphasizes the insight that scaled storage access rate should match the effectively scaled compute power and storage capacity [14]. Complicated concurrency control is not a part of it. Systems like Dynamo [8] use eventual consistency to provide high availability and partition tolerance for cross-datacenter replication.

8 CONCLUSION

In this paper, we introduced *NIOSIT* method to improve data access efficiency for log-structured merge-tree storage systems. *NIOSIT* allows network request to be processed directly aiming at response in time. The method also incorporate Bloom filter to filter out empty reads for delta data in transaction node which is responsible for all the user reads and writes. Traditional plain network communication model utilizes worker thread with request queue is promoted by this method. Therefore, the waiting time in queue and context switching time is pruned to improve the data access performance.

In our method, efficient memory pool management for network IO is implemented. Besides, event-loop model with multiple network IO threads makes proper advantage of multi-core systems. Experiments show at least 20 percent throughput performance improvement of IO thread processing than queueing mechanism on YCSB benchmark.

9 ACKNOWLEDGEMENT

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

REFERENCES

[1] 2017. HBase website. http://hbase.apache.org/. (2017).
[2] 2017. OceanBase website. https://github.com/alibaba/oceanbase. (2017).
[3] 2017. SQLite website. https://sqlite.org/. (2017).
[4] Philip Bernstein and Eric Newcomer. 1997. *Principles of Transaction Processing: For the Systems Professional.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
[5] Burton H. Bloom. 1970. *Commun. ACM* 13, 7 (July 1970), 422–426.
[6] Fay Chang, Jeffrey Dean, and others. 2006. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7.* 15–15.
[7] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing.* 143–154.
[8] Decandia and others. 2007. Dynamo: amazon's highly available key-value store. In *SOSP.*
[9] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Åke Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL Server's Memory-optimized OLTP Engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data.* 1243–1254.
[10] Avinash Lakshman and Prashant Malik. 2010. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.* 44, 2 (April 2010), 35–40. DOI: https://doi.org/10.1145/1773912.1773922
[11] Lomet D. B. Mokbel M. F. et al. Levandoski, J. J. 2011. Deuteronomy: Transaction Support for Cloud Data. In *CIDR.* 123–133.
[12] Simon Loesing, Markus Pilman, Thomas Etter, and Donald Kossmann. 2015. On the Design and Scalability of Distributed Shared-Data Databases. (2015), 663–676.
[13] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The Log-structured Merge-tree (LSM-tree). *Acta Inf.* 33, 4 (June 1996), 351–385. DOI: https://doi.org/10.1007/s002360050048
[14] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Diego Ongaro, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. 2011. The Case for RAMCloud. *Commun. ACM* 54, 7 (July 2011), 121–130. DOI: https://doi.org/10.1145/1965724.1965751
[15] Ippokratis Pandis, Ryan Johnson, Nikos Hardavellas, and Anastasia Ailamaki. 2010. Data-oriented Transaction Execution. *Proc. VLDB Endow.* 3, 1-2 (September 2010), 928–939. DOI: https://doi.org/10.14778/1920841.1920959
[16] Sheldon M. Ross. 2006. *Introduction to Probability Models, Ninth Edition.* Academic Press, Inc., Orlando, FL, USA.
[17] Russell Sears and Raghu Ramakrishnan. 2012. bLSM: A General Purpose Log Structured Merge Tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data.* 217–228.
[18] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. 2007. The End of an Architectural Era (It's Time for a Complete Rewrite). In *Proceedings of the 33rd International Conference on Very Large Data Bases.* 1150–1160.
[19] Huanchen Zhang, David G. Andersen, Andrew Pavlo, Michael Kaminsky, Lin Ma, and Rui Shen. 2016. Reducing the Storage Overhead of Main-Memory OLTP Databases with Hybrid Indexes. In *Proceedings of the 2016 International Conference on Management of Data.* 1567–1581.