Laser: Load-Adaptive Group Commit in Lock-free Transaction Logging

Huan Zhou[1], Huiqi Hu[1]✉⋆, Tao Zhu[1], Weining Qian[1],
Aoying Zhou[1], and Yukun He[2]

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

Abstract. Log manager is a key component of DBMS and is considered as the most prominent bottleneck in the modern in-memory OLTP system. In this paper, by addressing two existing performance hurdles in the current procedure, we propose a high-performance transaction logging engine Laser and integrate it into OceanBase, an in-memory OLTP system. First, we present a lock-free transaction logging framework to eliminate the lock contention. Then we make theoretical analysis and propose a judicious grouping strategy to determine an optimized group time for different workloads. Experiment results show that it improves 1.4X-2.4X throughput and reduces more than 60% latency compared with current methods.

Keywords: logging, group commit, lock-free, load-adaptive

1 Introduction

Recent years have seen a shift in the design of high throughput OLTP systems: from the conventional transaction engine to the widespread adoption of multi-core memory system. To improve concurrency, many transaction engines focus on eliminating fundamental bottlenecks, such as lock-based shared data structure, concurrency control, centralized log manager etc. Among them, the log manager is considered as the most prominent bottleneck due to centralized design and dependence on I/O [2]. The state of art method[4] integrates three most widely used techniques, i.e. parallel buffering, flush pipelining and group commit, to form an efficient logging procedure and results show that the procedure can significantly achieve better performance than the traditional logging approaches.

In this paper, we propose a high-performance transaction logging engine called Laser. In particular, we observe two defects that limit the performance of the existing procedure: (1)current approach depends on a lock-based method to manage transaction log records, it involves many lock contentions and reduces the CPU utilization as load increases[6]. (2)existing method uses a fixed group

⋆ ✉ is the corresponding author.

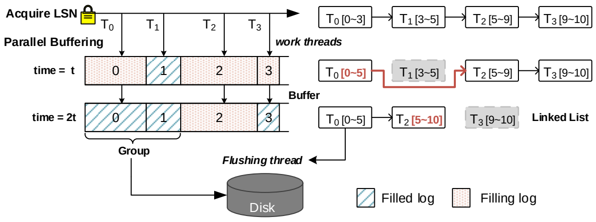


Fig. 1. Transaction commit processing in memory transaction Engine

commit strategy which cannot achieve a good performance when the workload changes. To obtain better latency and throughput, We present a new lock-free transaction logging framework with the help of a well-designed multi-group structure and CAS operation and propose an adaptive group commit where we make theoretical analysis and propose a judicious grouping strategy to determine an optimized grouping time when the workload varies. Then we implement these methods and integrate them into the in-memory OceanBase OLTP system. Results show that it achieves 1.4X-2.4X better performance over the compared methods in throughput as well as reduces more than 60% latency.

2 Related Work

The write-ahead logging(WAL) [5] is widely employed in database systems to provide data durability and recovery. Compared to traditional system, the latest OLTP systems have demonstrated significant performance improvement, however, the log manager is still prone to bottlenecks due to its centralized structure [2].

Many technologies are explored to reduce the overhead of logging. Johnson et al. [4] identify four bottlenecks of the write-ahead logging named I/O-related delay, log-induced lock contention, context switching and log buffer contention. Parallel buffering [4] is used to reduce the log buffer contention. Group commit [1] reduces the I/O-related delay by aggregating multiple log records into one I/O operation. Pat Helland et al. [3] turns out that the database can set group timer to minimize average response time, however it assumes the system load is unchangeable and only examines the effect of grouping time on CPU response time based on the traditional single thread system. Aether [4] utilizes flush pipelining to reduce the overhead of context switching and integrates it with parallel buffering and group commit to form an efficient logging procedure.

3 Preliminary

Transaction logging generally consists of two distinct steps: *pre-logging* and *commit logging*. In the pre-logging step, each transaction fills its log records into an in-memory log buffer. It first acquires a unique log sequence number(LSN) using a lock to indicate its allocated space within the buffer, then copies the log records into the log buffer. The usage of lock is to keep transactions acquiring their LSN in a monotonous serial order. In the second step, log records are physically flushed into disk following the LSN order through some I/O operations.

Three mature techniques such as parallel buffering, flushing pipelining and group commit have been adopted to optimize the procedure as illustrated in Figure 1.It is non-trivial to combine the three techniques together in a logging procedure. As log records are filled in parallel and must be written to disk in

LSN order, log records of large LSN orders cannot be flushed until the front transactions (transactions with small LSN orders) completed buffering. To this end, the flushing thread has to identify a "safe" region (of offset) in the log buffer for group commit. In the example, before notifying its region to flushing thread, the work thread of T_1 must wait until T_0 releases its buffer space. To solve the problem, the state of art [4] forms a linked list to release the buffer region in LSN order. For log records, work threads acquire their LSN and enqueue themselves into the list which protected by a lock. Each node in the list contains a "safe" region which indicates the range of log records starting from it and ending at the first successor that has not yet finish buffering. The "safe" region is figured out in a delegated way. Once a work thread finishes filling log records, it first abandons its node in the linked list and merges its offset into the range of its predecessor. When a flushing thread triggers log flushes using group commit policies, it first figures out buffer region of finished log records from the head of linked list. Log records within the region can be flushed into disks.

4 Lock-free Transaction Logging Framework

Data Structure We use a buffer (denoted by $\mathcal{B}$) with a constant size $|\mathcal{B}|$ to store log records. The buffer is used in a round-robin manner. We rely on a multi-group and a global offset of $\mathcal{B}$ (denoted by o_f) to manage the logging.

The multi-group structure is formed by a sufficiently large array denoted by $\{G_0, G_1 \cdots G_n\}$(n is large, e.g. $n = 10000$) and each group G_i is consisted of a sextuple $\langle$ $\texttt{group\_state}_i$, $\texttt{LSN}_i$, s_i, e_i, n_i, f_i $\rangle$ as shown in Figure **??**, where $\texttt{group\_state}_i$ is the current state of the group, $\texttt{LSN}_i$ is used by work threads to acquire LSN, s_i and e_i are the *logical* start and end offset of the group in $\mathcal{B}$, n_i is used to record the number of active transactions which do not complete filling in the group, f_i is used to indicate whether G_i is frozen. Details of their usages are introduced subsequently. Each group has three possible states `Available`, `Ready` and `Durable`. `Available` means the group is empty and the values($\texttt{LSN}_i$,s_i, e_i, n_i, f_i,) of the group can be set. `Ready` indicates the work threads that join into the group can start to acquire LSN and their log records are allowed to be filled into $\mathcal{B}$. `Durable` means that all the work threads in the group have completed buffering their log records and can flush log records into disk. It is worth notice that the multi-group is also used in a round-robin manner.

We also maintain a logical offset o_f to mark the start offset of transaction log records which will be flushed(i.e. the start offset for the next flush). Notice that we all use logical offset here($o_f/s_i/e_i$) and their physical address can be easily corresponded as $o_f\%|\mathcal{B}|$, $s_i\%|\mathcal{B}|$ and $e_i\%|\mathcal{B}|$ respectively.

Transaction Logging Procedure As shown in Figure 2, we adopt a lock-free mechanism which maintains a global 128 bits structure $\mathcal{Q} =\langle G_i,\ r_i,\ o_i\ \rangle$, where G_i is the group used for acquiring LSN, r_i and o_i are used to record the relative log serial number and offset in the G_i. When a work thread comes to acquire a LSN, it first retrieves $\mathcal{Q}$ and generates a new $\mathcal{Q}$ by increasing $r_i = r_i + 1$, $o_i = o_i + |T|$ (where $|T|$ is the size of log records) with a CAS operation. After acquiring $\mathcal{Q}$, it sets the $n_i = n_i + 1$. When G_i is ready, the work thread detects if its log records can be buffered by comparing $s_i + o_i - o_f <= |\mathcal{B}|$. If the $\mathcal{B}$ has

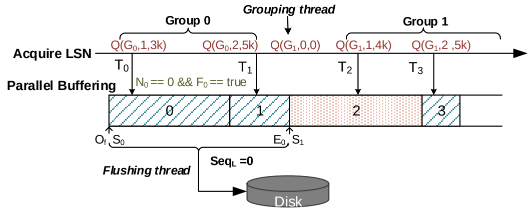


Fig. 2. Lock-Free Transaction Logging Framework

enough space, the work thread assigns the LSN of its transaction as $\texttt{LSN}_i + r_i - 1$ and fills its log records with its offset. After completing buffering, it decreases the value of n_i by one, then it can turn to serve other transactions. When the last work thread completes buffering, it changes the state of group from `Ready` to `Durable`. A work thread confirms itself as the last thread if it finds n_i=0 and its frozen indicator f_i=true assigned by a grouping thread.

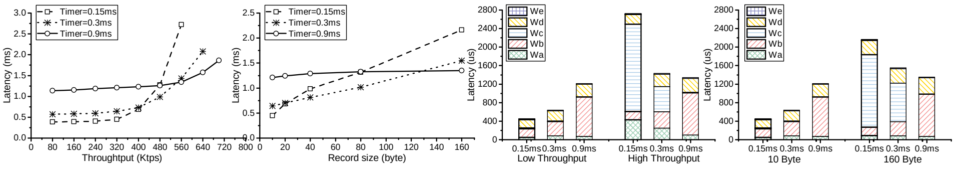
Note that $\texttt{LSN}_i$ and s_i is required for the work thread. Both of them are computed in a *grouping thread* which is used to construct groups. When the condition of group commit is satisfied, the grouping thread generates a new $\mathcal{Q}$ as $\langle G_{i+1},\, r_{i+1} = 0,\, o_{i+1} = 0\rangle$ with a CAS operation. Then it sets the end offset of the previous group G_i as $e_i = s_i + o_i$, $f_i = true$. Next if the state of G_{i+1} is `Available`, it assigns $s_{i+1} = e_i$, $\texttt{LSN}_{i+1} = \texttt{LSN}_i + r_i$ and makes the state into `Ready` so that the next coming transaction can acquire LSN on G_{i+1}.

The flushing thread maintains a position indicator(denoted by seq_L) to determine which group will be written into disk with an I/O operation. When $seq_L = i$, if the state of G_i becomes `Durable`, the flushing thread flushes its contained log records from $(s_i\%|\mathcal{B}|, e_i\%|\mathcal{B}|)$(or $(s_i\%|\mathcal{B}|, |\mathcal{B}|)$ and $(0, e_i\%|\mathcal{B}|)$) into disk, then it increases $o_f = e_i$, $\texttt{group\_state}_i = \texttt{Available}$ and increases seq_L to directing the next group for the next flush.

5 Load-Adaptive Group Committing

Observation In this section, we investigate the influence of different grouping strategy on logging performance. We observe that the best grouping time(group timer) is distinguishing for different loads as shown in Figure 3. We exploit the observation by decomposing the executing time of transaction logging into five main components (1) W_a: the average time that a transaction waits for a group changing to `Ready`. (2) W_b: the average time that a transaction waits for its group becoming `Durable`. (3) W_c: the average time that a transaction waits for its log records to be flushed after its group becomes `Durable`. (4) W_d: the average time that the flushing thread writes the log records into disk. (5) W_e: the rest time spent for logging.

We breakdown the executing time when load throughput and log record size are different in Figure 3. Consider load throughput, given a small one(320k tps), group timer= 0.9ms have the largest latency where W_b takes the most time cost, because the procedure provides a large time window of 0.9ms to make threads acquire their LSNs and buffer their log records. In fact, Figure 3(c) proves 0.15ms is a sufficient gap when the throughput is 320k. On the contrary, 0.15ms is not sufficient to sever when the throughput is high due to the smaller timer generates more groups to be flushed in every seconds. Many transactions



(a) Varying the Load (b) Record Size (c) Different Load (d) Different record Size

Fig. 3. Time Decomposition of Different Loads

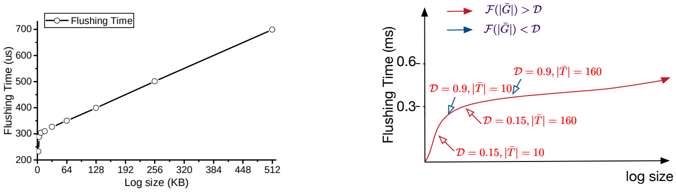
in groups have to wait for the flushing thread, which causes a largest overhead of W_c. Timer= $0.9ms$ significantly reduces the overhead as it lowers the generation rate of group. Similarly, increasing the record size extends the time to flush a group(W_d) in Figure 3(d). Therefore there is a largest overhead of W_c when group timer is $0.15ms$ and record size is 160 byte. On the contrary, the larger timer does not have the overhead of W_c.

In summary, *when the load throughput or the record size grows, it involves increasing overhead of W_c. Expand the grouping time increases cost of transactions waiting for the group becoming durable(W_b), but it can reduce the overhead of transactions in a group waiting for flushing(W_c).*

Theoretical Analysis We further discuss the above conclusion through some theoretical analysis. First, we demonstrate an I/O property between the log size and its respect flushing time. Note that it is a general property for I/O operation that well recognized by many data accessing test. Figure 4(a) demonstrates the property: *the amortized time for writing a larger size of log is less than a smaller one.* For example, when it writes $100kb$ log, it spends about $0.4ms$, however, if it only costs about $0.7ms$ to write $500kb$ (5 times large than $100kb$) log. Based on the property, we analyze the reason why increasing the grouping time can reduce the overhead of W_c. Let λ and $|\bar{T}|$ denote the load throughput and log record size. Let $\mathcal{D}$ denotes the group timer and $\mathcal{F}(|\bar{G}|)$ be the total time to write log records where $|\bar{G}|$ is the *size of grouping log* that contained in one group for flushing. Obviously, if $\mathcal{F}(|\bar{G}|) > \mathcal{D}$, the procedure produces overhead of W_c since the group requires to wait its prior group. $|\bar{G}|$ can be computed by the following equation:

$$|\bar{G}| = \begin{cases} |\bar{T}| * \mathcal{D} * \lambda & \lambda < 1/t_{CAS} \\ |\bar{T}| * \mathcal{D} * 1/t_{CAS} & \lambda \geq 1/t_{CAS}. \end{cases}$$

where t_{CAS} is the constant time for a transaction to acquire LSN with the atomic CAS operation and $\mathcal{D} * 1/t_{CAS}$ is the largest number of contained log records. Therefore, when $|\bar{T}|$ is constant, we adjust the value of $\mathcal{D}$ according to the variation of λ to make sure $\mathcal{F}(|\bar{G}|) \leq \mathcal{D}$. Similarly, we can change the $\mathcal{D}$ based on the variation of $|\bar{T}|$ when λ is fixed. As shown in Figure 4(b), if $\mathcal{D} = 0.15ms$ and $|\bar{T}| = 10$ byte when $\lambda = 320k$ tps, $\mathcal{F}(|\bar{G}|) = 0.158ms$. This is close to its group timer, thus there is almost not overhead of W_c. If the record size increases, e.g. $|\bar{T}| = 160$ byte, the $|\bar{G}|$ enlarges and $\mathcal{F}(|\bar{G}|)$ becomes $0.3ms$ where $\mathcal{F}(|\bar{G}|) > \mathcal{D}$ and causes large overhead of W_c. Now consider using the larger group timer(e.g. $\mathcal{D} = 0.9ms$), though $|\bar{G}|$ increases several times as larger $\mathcal{D}$, but only smaller growth is generated on flushing time due to the I/O property. When $\mathcal{D} = 0.9ms$ and $|\bar{T}| = 160$ bytes, $\mathcal{F}(|\bar{G}|)$ is $0.35ms$ where $\mathcal{F}(|\bar{G}|) << \mathcal{D}$ and avoids the overhead of W_c.



(a) I/O Property (b) Varying the Timer and record Size

Fig. 4. Exploiting the Flushing Time

To conclude, we find *(1) increasing the grouping time will increase the time of group being durable(W_b), but it reduces the overhead of W_c. (2) When the flushing time is less than the group timer($\mathcal{F}(|\bar{G}|) < \mathcal{D}$), it eliminates or significantly reduces the overhead of W_c. (3) When $\mathcal{D}$ is small, $\mathcal{D}$ is smaller than $\mathcal{F}(|\bar{G}|)$, but when $\mathcal{D}$ is large enough, $\mathcal{D}$ will be larger than $\mathcal{F}(|\bar{G}|)$ due to the I/O property.*

Grouping Strategy Based on the above conclusions, we propose our group strategy as *choosing the group timer which generates smallest cost of W_b by eliminating the overhead of W_c*. Formally, it is the minimum group timer $\mathcal{D}^*$ that avoids W_c by satisfying the following equation:

$$\mathcal{D}^* \geq \mathcal{F}(|\bar{G}|_{D^*}). \tag{1}$$

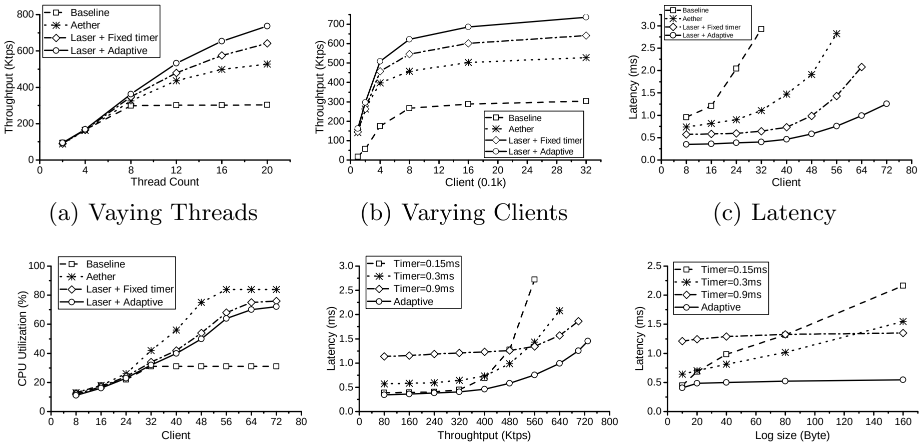
where $|\bar{G}|_{D^*}$ is the size of grouping log by utilizing $\mathcal{D}^*$ as group timer.

To implement the grouping strategy, we monitor the flushing time W_d (where W_d is closed to $\mathcal{F}(|\bar{G}|_{D^*})$). We turn the group timer based on the previous value and the current flushing time in case of it changing too heavily at a time. In particular, it is tuned as $\mathcal{D} = 1/2 * \mathcal{D} + 1/2 * \mathcal{F}(|\bar{G}|_{D^*})$. $\mathcal{D}$ is tuned by the grouping thread and after several times, $\mathcal{D}$ will become close to $\mathcal{F}(|\bar{G}|_D)$. If we observe W_a increase heavily over a threshold τ (e.g. 0.1ms), it means the load throughput has increased and the group is not available, we also increase $\mathcal{D}$ as $\mathcal{D} = \mathcal{D} + \delta$ where δ is a constant time(e.g. $\delta = 2\tau$).

6 Experiment

Experimental Setup The experiments are conducted on a linux server with *268GB* main memory and *two Interl Xeon E5-2630@2.20GHZ processors*, each with *10* physical cores. We use RAID5 with flash-based write cache(FBWC) which has high performance on I/O accesses.We implement `Laser` and the comparing methods into OceanBase [7]. We compare `Laser` with (1) **Baseline**: the origin transaction logging manager that Oceanbase adopts, it uses a single logging thread to acquire LSN order and fills logs in sequence, and flushes the logs with group commit. (2) **Aether** [4]: it utilizes parallel buffering, flush pipeline and group commit to form a logging procedure as described in Section 3.We test the methods on YCSB workload which is popular in evaluating the read/write performance for a database system. We only utilize the update transaction and the record size is from 10B to 160B.

Evaluation of Lock-free Transaction Logging First we compare the *scalability, peak throughput, latency* and *CPU utilization* with Baseline, Aether and Laser+Fixed timer by varying the number of work threads and clients. When we vary one of parameters, the rest parameters are setting by default values where



(a) Vaying Threads (b) Varying Clients (c) Latency

(d) CPU Utilize (e) Varying Load (f) Varying Log Size

Fig. 5. Evaluation of optimized Transaction Logging

the number of threads is 20, the number of client is 3200, group timer= $0.3ms$ and record size is 10 byte. The results are reported in Figure 5. We can see Laser+Fixed timer always achieves the best throughput under all the situations, which improves 1.2X-2X better performance.

Scalability. As shown in Figure 5(a), the performance of Laser+Fixed timer increases nearly linearly when the number of work threads grows. The peak throughput of Aether increases slowly when the number of work threads is bigger than 12 due to acquiring LSN based-on lock which limits the scalability. Baseline quickly becomes saturated when thread count is 8 since the single logging thread becomes the critical bottleneck.

Client-side Throughput. Figure 5(b) shows the performance of Laser+Fixed timer increases when the number of client varies however Aether increases very slowly when client= 1600 and Baseline becomes saturated when client= 800.

Latency. Figure 5(c) shows Laser+Fixed timer always takes the lowest time when the number of client grows, which improves the latency of Aether and Baseline more than 45%.

CPU utilization. Figure 5(d) shows the utilization of Laser+Fixed timer increases gently until the number of client equals 64. However, the cpu utilization of Aether increases fleetly until the client count is 56 due to lock contention appears. The CPU utilization of Baseline does not increase when the number of client equals 32 due to its single thread is saturate.

Evaluation of Adaptive Group Commit Next we compare the adaptive grouping strategy with three fixed group timers in YCSB. We vary the number of client to increase load throughput(each client generates $10k$ transactions per second) and grow the record size where the default value of load is $320k$. Results are shown in Figure 5(e) and 5(f). Adaptive group commit always has lowest latency compared to fixed group timer. For instance, when the record size is 10 bytes, the latency of timer $0.15ms$, $0.3ms$ and $0.9ms$ is roughly $0.44ms$, $0.64ms$ and $1.21ms$ respectively, and the latency of Adaptive is $0.4ms$. Meanwhile, the adaptive group commit improves the peak throughput. Adaptive can serve for

730k load throughput, and the timer 0.15ms, 0.3ms, 0.9ms only serve for 560k, 640k and 700k respectively.

Overall Performance Finally, we integrate the lock-free logging framework and the adaptive group commit as Laser+Adaptive and evaluate the overall performance. Results are reported in Figure 5. We can see (1)Laser+Adaptive offers the **best throughput**, which improves **1.4X-2.4X** better peak throughput than Baseline and Aether. For example, in Figure 5(b), when the number of client equals 3200, the peak throughput of Laser+Adaptive is about 730k, where the peak throughput of Baseline, Aether and Laser+Fixed timer are 300k, 520k and 640k respectively. (2) Laser+Adaptive has the **lowest latency** when varying load throughput which improves the latency by **60%**. For example, in Figure 5(c), when the number of client is 32, the latency of Baseline, Aether and Laser+Fixed timer is 2.9ms, 1.1ms and 0.6ms respectively, and Laser+Adaptive is nearly 0.4ms.

7 Conclusion

In this paper, we propose an optimized transaction logging engine by proposing a new lock-free transaction logging to improve scalability based on a designed multi-group structure and CAS operation, and presenting a judicious grouping strategy which economizes the running time of logging for varied workload through some theoretical analysis. Implementation in Oceanbase and experiment results show the new logging engine can reduce more than 60% latency and achieve 1.4X-2.4X better throughput compared with existing methods.

8 Acknowledgement

This work is partially supported by National Hightech $R\&D$ Pro- gram(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. Hagmann, R.: Reimplementing the Cedar file system using logging and group commit. In: SOSP, pp. 155–162. ACM (1987)
2. Harizopoulos, S., Abadi, D.J., Madden, S., Stonebraker, M.: OLTP through the looking glass, and what we found there. In: SIGMOD, pp. 981–992. ACM (2008)
3. Helland, P., Sammer, J.L., Carr, R., Garrett, P., Reuter, A.: Group commit timers and high volume transaction systems. In: High Performance Transaction Systems, pp. 301–329. Springer (1989)
4. Johnson, R., Pandis, I., Stoica, R., Manos, A.: Aether: a scalable approach to logging. PVLDB, 3(1-2) 681–692 (2010)
5. Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., Schwarz, P.: ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. TODS, 17(1) 94–162 (1992)
6. Wang, T.Z., Johnson, R., Stoica, R., Manos, A.: Scalable logging through emerging non-volatile memory. PVLDB, 7(10) 865–876 (2014)
7. OceanBase website, https://github.com/alibaba/oceanbase/