Plover: Parallel In-Memory Database Logging on Scalable Storage Devices

Huan Zhou[1], Jinwei Guo[1], Ouya Pei[2], Weining Qian[1](✉), Xuan Zhou[1], and Aoying Zhou[1]

[1] School of Data Science and Engineering, East China Normal University, Shanghai 200062, China
{zhouhuan,guojinwei}@stu.ecnu.edu.cn,
{wnqian,xzhou,ayzhou}@dase.ecnu.edu.cn

[2] Northwestern Polytechnical University, Xian 710072, China
oypei@mail.nwpu.edu.cn

Abstract. Despite the prevalence of multi-core processors and large main memories, most in-memory databases still universally adopt a centralized ARIES-logging with a single I/O channel, which can be a serious bottleneck. In this paper, we propose a parallel logging mechanism, named `Plover` for in-memory databases, which utilizes the partial order property of transactions' dependencies and allows for concurrent logging in scalable storage devices. To further alleviate the performance overheads caused by log partitioning, we present a workload-aware log partitioning scheme to minimize the number of cross-partition transactions, while maintaining load balance. As such, `Plover` can scale well with the increasing number of storage devices and extensive experiments show that `Plover` with workload-aware partitioning can achieve 2× speedup over a centralized logging scheme and more than 42% over `Plover` with random partitioning.

Keywords: In-memory database · Parallel logging · Scalability

1 Introduction

The advent of multi-core processors makes low-speed disk a major performance bottleneck. Owing to the increasing size of main memory, many databases can host the entire data set in main memory to reduce disk I/Os. Unfortunately, to ensure the durability of transactions, in-memory systems have to flush logs to permanent storage regularly. Using a single disk as the permanent storage is not performant, due to its limited I/O bandwidth. Meanwhile, these systems still rely heavily on a centralized ARIES-style [1] logging mechanism to guarantee the global order of log entries. Since the total order property of logging implies the dependencies among transactions, databases can be reconstructed correctly in accordance of the order of log entries after failure recovery. However, contentions for the centralized log buffer and limited synchronous I/Os still exist, which may become a major overhead as system load increases.

Y. Cai et al. (Eds.): APWeb-WAIM 2018, LNCS 10988, pp. 35–43, 2018.
https://doi.org/10.1007/978-3-319-96893-3_3

In this paper, we propose a parallel logging mechanism for the in-memory database called `Plover`, which utilizes partial order of transactions' dependencies. The key idea is to employ distributed logging instead of centralized logging to mitigate the contention on the centralized data structure, and to use scalable storage devices to increase the I/O bandwidth. Implementing such a distributed logging is not trivial, due to two main challenges: (1) how to preserve the temporal order among log entries; (2) how to distribute the log entries across the storage devices. To address the first challenge, we use a global sequence number to identify the partial order of log entries, and a persistent group commit method to ensure all log entries of a transaction are persistent before committing. To simplify the implementation and accelerate the recovery process, we adopt tuple-level distributed logging, which partitions log entries by tuples. However, this leads to the second challenge: cross-partition transactions and workload skew, which may significantly deteriorate the performance. To resolve the potential defects, we propose a workload-aware log partitioning scheme, which applies a graph partitioning algorithm to find workload balanced partitions, while minimizing the number of distributed transactions. Finally, we demonstrate that `Plover` can achieve linear scalability with an increasing number of storage devices. In TATP and TPC-C, `Plover` with workload-aware log partitioning outperformed centralized logging by a factor of $2\times$ and `Plover` using random partitioning by a factor of $1.42\times$ on two storage devices.

2 Background and Related Work

Centralized Logging. To recover data from failures, a database system needs to leverage logging mechanism to guarantee atomicity and durability for transactions. For a in-memory database, the ARIES logging ensures that all REDO log entries are organized in a global order and a transaction can be committed only if all of its log entries have been persisted. The log sequence number (LSN)—which is unique and monotonically increasing— can be used to guarantee the global order of log entries. More specifically, the procedure of logging is described as follows:

(1) Log entry insertion. Before copying the log entry to the centralized log buffer, the transaction must acquire an LSN and claim the buffer space it will eventually fill with the intended log entry by a lock or a mutex. The lock or mutex will be released once the transaction finishes copying the log entry.

(2) Log entry persistence. The logging subsystem appends the log entries cached in log buffer to the log file in a single storage device. This can ensure that the entries are consecutive in the log file.

(3) Transaction committing. The transaction can commit safely after the log entries whose LSNs are less than or equal to those of its own log entries are persisted in the storage device.

However, with the CPU cores increases in a single machine, centralized logging is becoming a main bottleneck, especially in main-memory database systems, where logging is the only source of synchronous I/Os. Traditional centralized logging faces the following challenges: (1) upper limit of generating LSNs; (2) log buffer contention; and (3) limited synchronous I/Os.

Related Work. To improve the scalability of centralized logging, there have been active researches on above bottlenecks to develop new logging protocols. To alleviate the contention of allocation of LSNs, Kim et al. [3] presented a latch-free approach and Jung et al. [10] designed a concurrent data structure to ensure the global order of log entries; to improve the performance of log insertion, Johnson et al. [2] proposed a scalable logging with decoupling log inserts method so that log entries of different transactions can be copied into the log buffer in parallel; to eliminate the cost of synchronous log writes, most databases provided asynchronous commit strategy but at expense of durability. And there have been active researches to develop new logging protocols [7,9] based on the arrival of non-volatile memory (NVM) technology; to eliminate the limited I/O bandwidth of single storage device, Zheng et al. [6] implemented a transaction-level distributed logging mechanism with multiple storage devices and Wang et al. [8] proposed a universal distributed logging mechanism on multiple NVMs.

To the best of our knowledge, there are not works that can address all the issues we proposed. Therefore, we design a novel parallel logging mechanism, which utilizes partial order property of transactions' dependencies and adopts multiple log buffers and storage devices.

3 Parallel Logging

Overview. `Plover` aims at providing excellent performance and scalability for transaction logging, by leveraging distributed logging and multiple permanent storage devices. In our approach, the distributed logging is partitioned under tuple level, each log partition is processed by a dedicated logger thread and all of the log partitions can be accessed by all the worker threads. As modifications from a transaction may be written into many log partitions, there are two main challenges: (1) how to identify transaction dependencies for log entries over multiple log partitions; (2) how to protect committed work for a transaction. To tackle the two challenges, we prefer to employ a *global sequence number* (GSN), and propose a variant of group commit method, *persistent group commit*. The `GSN` provides a partial order based on logical clock [4] and guarantees the transaction dependencies among log entries over multiple log buffers. And a transaction can not safely commit until all of its log entries, along with all the log entries that logically precede them, have become persistent. Therefore, the persistent group commit starts a daemon thread to periodically monitor the submission of all logger threads and ensures that transactions can correctly commit.

Normal Processing. Next, we detailedly describe the logging processing of transactions ($\mathrm{t} \times 6$, $\mathrm{t} \times 7$, $\mathrm{t} \times 8$) in `Plover` with two log buffers (partitions) *partition A, B*, as illustrated in Fig. 1.

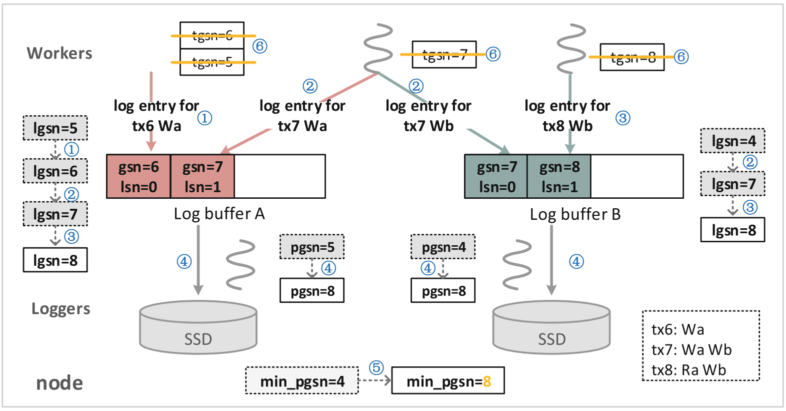


Fig. 1. Overview of parallel logging for main-memory database.

(1) Log entry generation. When a transaction is ready to commit, a worker thread generates a corresponding number of log entries based on the data partitions modified by the transaction. For the running transactions, $t \times 6$ and $t \times 8$ severally have a log entry, but $t \times 7$ produces two log entries.

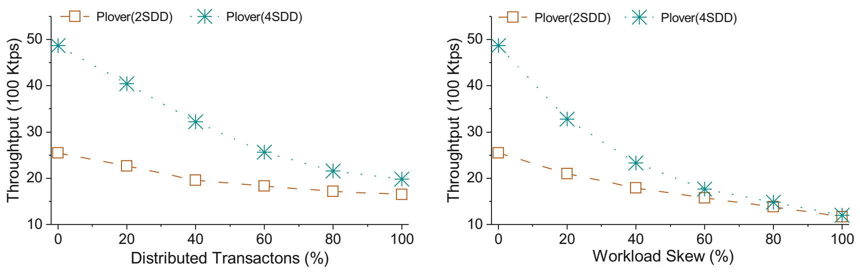
(2) Log entry insertion. Before writing the generated log entries into matched log partitions, the worker thread needs to assign a GSN for all the log entries. The value of GSN is also maintained in each transaction (`t_GSN`) and each log partition (`l_GSN`). Computing a GSN should get the `l_GSN`s of corresponding partitions and set the value as $max(\texttt{l\_GSN}_i) + 1$, where i is the serial number of corresponding partitions. For tx6 which updates tuple a, it only acquires the `l_GSN` of partition A ($\texttt{l\_GSN}_a = 5$) and assigns its GSN as $\texttt{l\_GSN}_a + 1 = 6$, as step ①. For $t \times 7$ which modifies tuple a and b, it must get the `l_GSN` of partition A and B and computes its GSN as $max(\texttt{l\_GSN}_a = 6, \texttt{l\_GSN}_b = 4) + 1 = 7$, as step ②. To guarantee the true-dependency (RAW) and anti-dependency (WAR) among transactions, we also consider the case that read and write operation of a transaction across over multiple partitions. For $t \times 8$, although it only modifies tuple b, it also needs to acquire $\texttt{l\_GSN}_a = 7$, $\texttt{l\_GSN}_b = 7$ and sets its GSN as 8, as step ③. In addition to the GSN, each log entry also stores a LSN, which is used to indicate the space of an individual log buffer. Moreover, to further improve performance, we release the buffer latch once a transaction have obtained the GSN so that many worker threads can copy log entries in parallel.

(3) Log entry persistence. When many log entries are accumulated in log buffers, each logger thread triggers group commit to force them into disk within a single I/O, and then updates its thread-local variable (`pgsn`) as the GSN of the last log entry that have been persistent, as step ④. Subsequently, the persistent group commit daemon examines the `pgsn` of all logger threads and computes the smallest `pgsn` as `min_pgsn`, as step ⑤. The `min_pgsn` represents the upper bound of persistent log entries and transactions whose $\texttt{t\_GSN} \leq \texttt{min\_pgsn}$ can be allowed to commit, as step ⑥. If a logger thread takes too long to update its pgsn (perhaps because of the corresponding partition accessed by a long read-only transaction), the persistent group commit daemon updates the logger thread's pgsn as the maximum value among all the pgsn of logger threads.

4 Recovery

Checkpoint. To accelerate data recovery from a failure, the in-memory database mandates a periodic checkpoint of its state during normal processing. In our Plover, the checkpoint is also partitioned according to tuples. Each checkpoint partition relates to a log partition and is processed by a dedicated checkpointer thread. When launching a new checkpoint, a checkpoint manager records the current min_pgsn as c_GSN which indicates the timestamp for a consistent snapshot, and then starts up n checkpointer threads, where n is the number of storage devices. Each checkpointer thread stores the consistent snapshot into m checkpoint files and reports to the checkpoint manager. At last, the manager writes the c_GSN and checkpoint metadata into a special file.

Failure Recovery. Plover masks outages by loading the most recent checkpoints (checkpoints recovery) and then repaying the log entries in log files (log recovery). In checkpoints recovery phase, a recovery manager thread acquires the newest metadata and c_GSN, where c_GSN denotes the starting point for log recovery, and then initiates m * n threads to recovery all the checkpoint files in parallel. In log recovery phase, all the recovery threads are used to replay the log entries whose GSNs are larger than c_GSN and less than r_GSN. The r_GSN is the latest min_pgsn at the database crash, which written into a storage device by the persistent group commit daemon during transaction processing.



(a) across-partition transaction (b) workload skew

Fig. 2. Impact of distributed transactions and workload skew on throughput.

5 Workload-Aware Log Partitioning

Performance Issues. Recall that the normal processing of our parallel logging, we find that the execution of a transaction is closely related to log partitioning

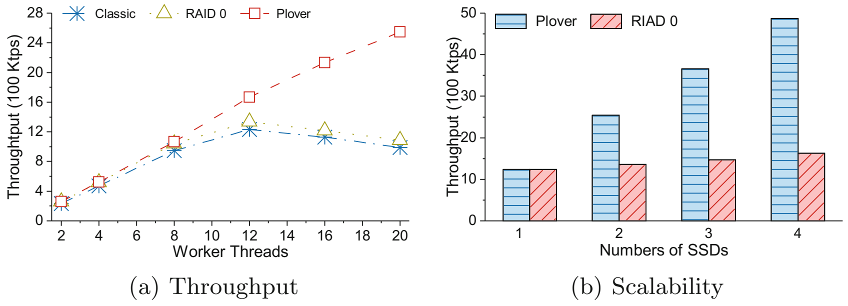


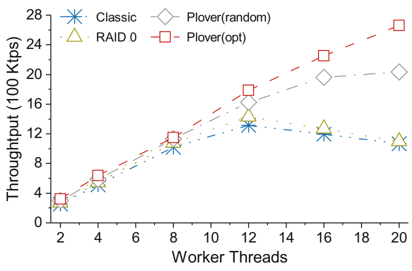
Fig. 3. Performance of parallel logging when running the microbenchmark.

and application workloads. Therefore, there are two subtle performance pitfalls: distributed transactions and workload skew. For the distributed transactions, as their `GSN` generation involves multiple log buffers, it increases computing overhead and reduces parallelism for logging processing. For the workload skew, it causes a log partition to suffer significant contention and excessive I/O overhead. As shown in Fig. 2, we explore the impact of distributed transaction and workload skew on throughput. We perform `Plover` with 2 and 4 log partitions (referred as `2SDD` and `4SDD`) respectively in microbenchmark and the experimental setup is shown in Sect. 6.

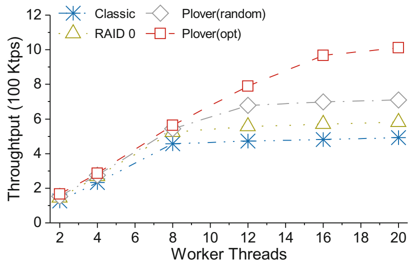
Partitioning Design. To solve the problems mentioned above, we implement a workload-aware log partitioning in our distributed logging. Firstly, we model the workload as a graph, $G = (V, E)$, where each vertex $v \in V$ represents a tuple, and the edge $e_{ij} \in E$ between v_i and v_j represents the connected tuples accessed by a same transaction. Each edge is associated with an edge weight w_e which accounts for the frequency of the transactions. After establishing the graph, we use a *k-way balanced min-cut partitioning* [5] to split the graph into k non-overlapping partitions such that the number of distributed transactions is minimized, while keeping the partitions within a constant factor perfectly balanced. To achieve the workload evenly across partitions, we consolidate the tuple size and access frequencies as a *factor* and assign the factor to each vertex.

6 Evaluation

Experimental Setup. All of our experiments are run on a single machine with two Intel Xeon E5-2630 (a total of 20 physical cores). The machine is equipped with 268GB DRAM and 4 pieces of SATA SSDs. We implemented a transactional logging prototype `Plover` in Java and each thread combines a database worker thread with a workload generator in our implementation. We compare the performance of our parallel logging equipped with multiple SSDs (referred to as *plover*) with two approaches: centralized logging with a single SSD (*classic*) and centralized logging equipped with RAID 0 (*raid0*). And



(a) TATP



(b) TPC-C

Fig. 4. Performance of parallel logging with workload-aware log partitioning.

Table 1. Recovery performance.

Variant	Checkpoint recovery time (seconds)	Log recovery time (seconds)	Total time (seconds)
Classic	67.7	163.5	231.2
RAID0	37.9	87.9	125.8
Plover	34.8	82.6	117.4

then we conduct experiment on the logging with the proposed workload-aware partitioning (*plover(opt)*) and with random partitioning (*plover(random)*). We run a microbenchmark which models a single write transaction with a 100 bytes log entry, TATP (Insert Call Forwarding) and TPC-C (New Order) on all system variants. For each benchmark and variant, each point reported in all graphs is the average throughput of three consecutive 120 s runs.

Effectiveness of Parallel Logging. We first compare the throughput and scalability with `Plover`, `Classic` and `RAID0` in microbenchmark. Figure 3 illustrates the experimental results.

Throughput. In this experiment, `Plover` and `RAID0` are equipped with two SSDs. In Fig. 3(a), as we increase the number of worker threads, the throughput of both `Classic` and `RAID0` rises steadily at first, but dramatically decreases when the number is larger than 12. However, `Plover` achieves linear scalability up to 20 threads. Owing to two logging simultaneously, `Plover` avoids the intensive contention of centralized logging and improves near 2× better performance in terms of peak throughput than `Classic` and `RAID0`.

Scalability. As shown in Fig. 3(b), `Plover` scales effectively as we increase the number of SSD drivers. The performance of `Plover` is proportional to the number of SSDs, but for `RAID0`, the non-linear speed-up is due to contention on the centralized log buffer.

Overall Performance. Next, we evaluate the performance of our parallel logging with the workload-aware log partitioning scheme in diverse workloads.

For TATP, both random partitioning and our approach can perfectly avert distributed transactions. But the random partitioning may suffers workload-skew. Hence, as shown in Fig. 4(a), `Plover` with the workload-ware partitioning `plover(opt)` has the best performance as increasing the number of worker threads, which improves 2× better peak throughput than `Classic` and `RAID0`, and increases performance more than 30 % compared with `plover(random)`.

For TPC-C, "New Order" produces a variable-sized log entry, about from 800 byte to 2250 byte. The larger size per log entry makes the peak throughput of `Classic` and `RAID0` quickly become saturated as growing the number of worker threads, as shown in Fig. 4(b). And the throughput of `plover(random)` does not further increase when the number of worker threads is larger than 12. That is because there are distributed transactions and workload skew in the random log partitioning. But our proposed scheme, `plover(opt)` achieves the best performance, which improves the peak throughput by factor of 2× over `Classic` and `RAID0`, and more than 42% over `plover(random)`.

Recovery. To investigate the effectiveness of our logging for recovery, we use the microbenchmark without distributed transactions and workload skew. When the system fails, we acquire 28 GB checkpoints and 54 GB log files. In this experiment, `Plover` and `RAID0` are equipped with two SSDs. As shown in Table 1, `Classic` has the largest total recovery time. This is because all of the checkpoint files and log files are stored in a single storage device and the limited I/O bandwidth seriously reduces its recovery performance. Owing to the parallel load, `RAID0` and `Plover` can respectively improve the recovery time by a factor of 1.83× and 1.97× speedup over `Classic`.

7 Conclusion

In this paper, we introduce a parallel logging in the main memory database named `Plover`, which replaces the centralized log buffer with multiple tuple-level distributed log buffers and allows log entries to be simultaneously forced into multiple storage devices. Our distributed logging relies on a logical global sequence number to identify the uniqueness of log entries and a persistent group commit method to ensure a transaction can be safely committed. We also analyze the impacts of distributed transactions and workload skew on performance and present a workload-aware log partitioning scheme based on a graph-partitioning algorithm to produce high-quality partitions. Our experimental evaluations demonstrate that `Plover` can provide linear scalability with the growing number of storage devices and the increasing number of worker threads. Due to the parallel design, our approach significantly alleviates the contention of centralized logging and the limitation of single I/O bandwidth.

Acknowledgement. This work is partially supported by National High-tech R&D Program (863 Program) under grant number 2015AA015307, National Key Research & Development Program of China (No. 2018YFB1003400), National Science Foundation of China under grant numbers 61702189, 61432006 and 61672232.

References

1. 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)
2. Johnson, R., Pandis, I., Stoica, R., Manos, A.: Aether: a scalable approach to logging. VLDB **3**(1–2), 681–692 (2010)
3. Kim, K., Wang, T.Z., Johnson, R., et al.: Ermia: fast memory-optimized database system for heterogeneous workloads. In: SIGMOD, pp. 1675–1687 (2016)
4. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7), 558–565 (1978)
5. Andreev, K., Racke, H.: Balanced graph partitioning. Theor. Comput. Syst. **39**(6), 929–939 (2006)
6. Zheng, W.T., Tu, S.: Fast databases with fast durability and recovery through multicore parallelism. In: OSDI, pp. 465–477 (2014)
7. Huang, J., Schwan, K., Qureshi, M.K.: NVRAM-aware logging in transaction systems. PVLDB **8**(4), 389–400 (2014)
8. Wang, T.Z., Johnson, R., Stoica, R., Manos, A.: Scalable logging through emerging non-volatile memory. PVLDB **7**(10), 865–876 (2014)
9. Arulraj, J., Perron, M., Pavlo, A.: Write-behind logging. PVLDB **10**(4), 337–348 (2016)
10. Jung, H., Han, H., Kang, S.: Scalable database logging for multicores. PVLDB **11**(2), 135–148 (2017)