Hindawi
Mathematical Problems in Engineering
Volume 2021, Article ID 6654461, 12 pages
https://doi.org/10.1155/2021/6654461

Research Article

Dependence-Cognizant Locking Improvement for the Main Memory Database Systems

Ouya Pei,[1,2] **Zhanhuai Li**,[1,2] **Hongtao Du**,[1,2] **Wenjie Liu**,[1,2] **and Jintao Gao**[1,2]

[1]*School of Computer Science, Northwestern Polytechnical University, Xi'an 710072, China*
[2]*Key Laboratory of Big Data Storage and Management, Northwestern Polytechnical University, Xi'an 710072, China*

Correspondence should be addressed to Ouya Pei; peiouya2013@mail.nwpu.edu.cn and Hongtao Du; duhongtao@nwpu.edu.cn

Received 29 October 2020; Revised 9 December 2020; Accepted 15 January 2021; Published 22 February 2021

Academic Editor: Shianghau Wu

The traditional lock manager (LM) seriously limits the transaction throughput of the main memory database systems (MMDB). In this paper, we introduce dependence-cognizant locking (DCLP), an efficient improvement to the traditional LM, which dramatically reduces the locking space while offering efficiency. With DCLP, one transaction and its direct successors are collocated in its context. Whenever a transaction is committed, it wakes up its direct successors immediately avoiding the expensive operations, such as lock detection and latch contention. We also propose virtual transaction which has better time and space complexity by compressing continuous read-only transactions/operations. We implement DCLP in Calvin and carry out experiments in both multicore and shared-nothing distributed databases. Experiments demonstrate that, in contrast with existing algorithms, DCLP can achieve better performance in many workloads, especially high-contention workloads.

1. Introduction

Although have arisen since the 1980s, but until recent decade, with the emergence of larger capacity and cheaper memory in which most or even all of data can be resident, MMDB began to achieve some success in commercial fields.

However, for MMDB, many of their workloads only consist of short transactions that access only a few records each. Therefore, the traditional locking mechanism has become the primary performance bottleneck. Based on the stand-alone single-core architecture, Harizopoulos et al. reported that 16% to 25% of the transaction overhead is spent on the lock manager [1]. As for the multicore processor architecture, some research studies similarly show that concurrently accessing the lock manager needs larger amounts of overhead [2–4].

As described in [5], the most common way to implement LM is as a hash table that maps each record's primary key to a linked list of lock requests. For lock acquiring, LM (1) probes the internal hash table to find the corresponding entry and then latches it, (2) appends a new request object to the entry list and blocks if it is incompatible with current holders, and (3) unlatches the entry and returns the (granted) request to the transaction. For lock releasing, it has to go through the steps of the lock request, and the only difference is removing the request from the requests' list.

These operations are still expensive. Firstly, row locking consumes too many computations and memory resources. Secondly, too many latches are applied to ensure correctness. Thirdly, list operations impose high overhead as the number of active transactions increases.

For reducing these overheads, some studies have reduced the number of lock acquisitions by redesigning the processing logic of LM [4, 6]. Others focus on optimizing the design and implementation of LM for improving LM scalability by reducing the latch and critical sections [7, 8], but these research studies still use the basic two-phase locking design.

With the vigorous development of deterministic transaction execution strategy, very lightweight locking (VLL) and selective contention analysis (SCA) were proposed by Kun Ren et al. [5, 9].

VLL's core idea is that use two simple semaphores containing the number of outstanding requests for that lock

(C_x for write requests and C_s for read requests). In this way, linked list operations are completely eliminated. However, tracking less information about contention results in poor throughput under high contention. SCA simulates the standard lock manager's ability to discover unblocked transactions by rescan TxnQueue that stores all active transactions in the order of arrival.

However, there are still some drawbacks which are as follows:

Unable to timely detect unblocked transactions. SCA only just makes VLL have the ability to detect unblocked transactions.

Easily blocked. For efficiency, low threshold is set for the number of blocked transactions. Thus, encountering long chain dependent or high contention workloads, the low threshold is easily exceeded so that subsequent transactions make no progress even if they are conflict-free.

1.1. Brief Introduction about DCLP. On the basis of the above analysis, it is necessary to revisit LM's design in MMDB. In this study, DCLP is introduced to reduce the expensive overhead of LM and ensure the ability to detect unblocked transactions without delay. Its core idea is tracking contention through co-locating one transaction with its direct successors. These following strategies are adopted to ensure efficiency:

Fragmented storage of lock requests means the lock request lists are partially removed, and the direct conflict information between transactions is collocated with the corresponding direct predecessors

Virtual transaction: using virtual transaction compresses continuous read requests by reference counting, and list operations are almost eliminated

With these two strategies, DCLP almost eliminates latch acquisitions and list operations and has the ability to timely detect which transactions should inherit released locks. Comparative experiments show that compared with VLL\SCA, DCLP achieves better (maximum 50% higher in some workloads) performance in high-contention workloads, long-dependent workloads, and workloads with high multipartition transactions, while it has approximately equal or slightly better performance in low-contention workloads.

1.2. Contributions and Paper Organization. This paper makes the following contributions:

We propose fragmented storage of lock requests. It completely eliminates the expensive overhead of list operations in terms of write-only workloads.

We propose the virtual transaction strategy. It effectively reduces the overhead of list operations by compressing continuous read requests.

We design a large number of experiments to evaluate DCLP overall. Four locking mechanisms are implemented in Calvin, and two benchmarks are used. Experiments show that DCLP is effective and efficient.

2. Principles and Algorithms

It is known that the major advantage of hash or hash-based data structures is they are efficient to search efficiency and more apparent when processing massive data. Especially, if the hash buckets are relatively large enough, a hash-based data structure that is a low collision or even collision-free can be easily designed and provides O(1) time complexity.

Hence, hash lock, a coarse-grained lock, which locks the primary keys with the same hash value at once, is the best choice. An ingenious hash lock is presented, and its format is as follows (for details, refer to Table 1):

$$(T_{lw}, V_{\text{ref}}, V_{\text{wait}}, V)$$

2.1. Fragmented Storage of Lock Requests. As the above analysis, lock acquire and release have to traverse the list. As the number of active transactions, the traversing list imposes higher overhead.

For reducing this overhead, fragmented storage of lock requests (fragmented storage for short) is proposed. For ease of interpretation, direct predecessors and direct successors are introduced, and their formal definitions are as follows.

Direct successor: here is from the perspective of the read-write lock conflict. Especially, if $T_i \prec T_k \prec T_j$ and $R_i(a) \prec R_k(a) \prec W_j(a)$, T_j is both a direct successor of T_i and a direct successor of T_k**.** Assume that T_i and T_j are any two transactions that appear in the same lock request list at the same time. If T_i precedes T_j, then $T_i \prec T_j$. For any two transactions T_i and T_j, T_j is the direct successor of T_i if and only if any one of the following three conditions is met:

(1) If key $\in T_i$.readSet $\cap$ T_j.writeSet, $T_i \prec T_j$, then (Tex translation failed), and $T_i \prec T_k \prec T_j$
(2) If key $\in T_i$.readSet $\cap$ T_j.writeSet, $T_i \prec T_j$, then (Tex translation failed), and $T_i \prec T_k \prec T_j$
(3) If key $\in T_i$.readSet $\cap$ T_j.writeSet, $T_i \prec T_j$, then (Tex translation failed), and $T_i \prec T_k \prec T_j$

Direct predecessor: for any two transactions T_i and T_j, T_i is the direct predecessor of T_j if T_j is the direct successor of T_i.

The core idea of fragmented storage is that a transaction is collocated with its direct successors. Each transaction has a list named successor List (preallocation and variable-length) to store its direct successors. Therefore, the lock request list is simplified, and its regular form is $W\{0, 1\}R^*$ (W for the last write request and R for the read request following W without other write transactions between them).

Hence, operations of granting to lock requests are significantly simplified. A write request can be granted if its corresponding request list is empty. A read request

Table 1: Notations.

Symbol	Meaning
T_{lw}	The last write transaction pointer
V_{ref}	The number of read transactions following one write transaction
V_{wait}	The number of transactions waiting to be granted
V	The virtual transaction pointer

can be granted if the head of its corresponding request list is not a write request. Releasing a write request is to remove the head of its request list if the head is itself. Releasing a read request (suppose R_{release}) is slightly complex, and the following two situations need to be considered:

(1) If the head of the request list is write, there is nothing to do because of compressing by the new write request
(2) If the head of the request list is read, traverse the request list, and then remove R_{release} if found

Write compression is essential for the efficiency of fragmented storage. The higher the write ratio, the better the efficiency of fragmented storage. For write-only workloads, the length of the request list is strictly no more than 1, and thereby, fragmented storage gains the best efficiency. For read and write mixed workloads, fragmented storage gains the second-highest efficiency. Why? The reason is that the length of the request list under such workloads is just shortened but generally not less than 2 (possibly larger). As the read ratio increases, the efficiency of fragmented storage gets worse and worse.

For more detailed processing logic, refer to Algorithms 1 and 2. These two algorithms are relatively simple, and so, we ignore additional supplementary instructions.

2.2. Virtual Transaction. For performing well in a continuous reading scenario, the virtual transaction is proposed to effectively reduce the overhead of list operations by compressing continuous read requests.

Virtual transaction: a special transaction which has only the right to lock and unlock primary keys and only operates shared read lock.

Continuous read compression: it is used to compress continuous reads per hash(key) to one which is represented by a virtual transaction. A simple semaphore containing the number of continuous outstanding read requests for that lock with no other write requests interrupted (actually, V_{ref} for continuous read requests) is used, and each successive read operation corresponds to a virtual transaction. From the above analysis, the regular form of the lock request list per lock under the combination of hash lock and fragmented storage is $W\{0,1\}R^*$, and the subpattern R^* is the essential reason for poor effectiveness of fragmented storage in both high-ratio-read workloads and read-only workloads. With read compression, the regular form of the lock request list per lock is further strictly limited to $W\{0,1\}R\{0,1\}$.

```
Input: T: a transaction pointer
(1) foreach key ∈ T ⟶ writeSet do
(2)    L = data[key].lockRequestList;
(3)    if T ≡ L ⟶ front.ownTrans then
(4)       L ⟶ popFront;
(5)    end
(6) end
(7) foreach key ∈ T ⟶ readSet do
(8)    L = data[key].lockRequestList;
(9)    if W ≡ L ⟶ front.lockMode then
(10)      L ⟶ removeIfExist(T);
(11)   end
(12) end
(13) foreach E ∈ T ⟶ successorList do
(14)   E ⟶ prednum − −;
(15)   if 0 ≡ E ⟶ prednum then
(16)      ReadyQueue←E;
(17)   end
(18) end
```

Algorithm 1: Lock release with fragmented storage.

This brings the following two benefits:

The memory cost and CPU cost of the lock request list are constant and minimized. $W\{0,1\}R\{0,1\}$ means that the length of the lock request list is no greater than 2, and thereby, the fixed-length list can be preallocated to reduce the high cost of operations (i.e., create and destroy). Two transaction pointers are placed in each hash lock, respectively, pointing to the last write transaction and the virtual transaction.

The cost of write lock request processing is significantly reduced. Processing a write lock request (assume W) may traverse the corresponding lock request list (assume L), and the transaction that issues W is added to successorList of each element of L. The longer L, the more times the additions. However, after introducing reference counting, W will be used as a direct successor to only two transactions (the last write transaction and the virtual transaction).

How does it work? Consider, for example, the sequence of transactions A, B, C, and D. Among them, A and D only write x, while B and C only read x. At first, x.T_{lw}, x.V, $x.V_{\text{ref}}$, and $x.V_{\text{wait}}$ are initialized to NULL, NULL, 0, and 0. Transaction A arrives. Because of conflict free, $x.T_{lw}$ is set to A and then grants A. Transaction B follows. Because of read-write collision, the following three steps are executed sequentially. (1) Add B to successorList of A. (2) Create a new virtual transaction T_{virtual}. (3) Increase prednum of T_{virtual} by one ($x.V_{\text{ref}}$: 0 ⟶ 1). When C arrives, steps 1 and 3 are selected to execute sequentially. Now, x.T_{lw}, x.V, $x.V_{\text{ref}}$, and $x.V_{\text{wait}}$ are A, T_{virtual}, 2, and 0, respectively. Finally, transaction D comes. Because of read-write collision, D is only added to successorList of T_{virtual} rather than to that of B and C.

Unique virtual transaction for each hash key: one virtual transaction was used to represent all reads for each key. In other words, there cannot be several virtual transactions on the same primary key. Any continuous read operations are

```
    Input: T: a transaction pointer
    Output: Granted: 1 if granted, otherwise 0
(1) Granted:  = 1;
(2) foreach key ∈ T ⟶ writeSet do
(3)   L = data[key].lockRequestList;
(4)   if L ⟶ Empty then
(5)     L ⟵ ⟨W, T⟩;
(6)   else
(7)     if T ≡ L ⟶ back.own Trans then
(8)       foreach E ∈ L do
(9)         l = E.ownTrans ⟶ successorList;
(10)        if T ≡ l ⟶ back then
(11)          Granted = 0;
(12)          l ⟵ T;
(13)          T ⟶ prednum + +;
(14)        end
(15)      end
(16)      L ⟶ clear;
(17)      L ⟵ ⟨W, T⟩;
(18)    end
(19)  end
(20) end
(21) foreach key ∈ T ⟶ readSet do
(22)  L = data[key].lockRequestList;
(23)  if L ⟶ Empty then
(24)    L ⟵ ⟨R, T⟩;
(25)  else
(26)    if T ≡ L ⟶ back.own Trans then
(27)      E = L ⟶ front;
(28)      if W ≡ E.lockMode then
(29)        l = E.ownTrans ⟶ successor List;
(30)        if T ≡ l ⟶ back then
(31)          Granted = 0;
(32)          l ⟵ T;
(33)          T ⟶ prednum + +;
(34)        end
(35)      end
(36)      L ⟵ ⟨R, T⟩;
(37)    end
(38)  end
(39) end
```

Algorithm 2: Lock acquire with fragmented storage.

passed to the write operation immediately following them by reference counting.

How to implement it? The only to do is to save multiple successor transactions for the virtual transaction rather than only to save its direct successor transactions. For saving multiple successor transactions, new $V_{\text{successorList}}$ is attached to virtual transactions each (for details, refer to Algorithms 3 and 4).

Assume that transaction is T and its corresponding hash lock is Lock.

For write lock acquire, see Algorithm 3, 2 to 27 lines. The basic processing logic is as follows:

(1) Check whether Lock.V_{ref} is greater than zero:

If true, prednum of T is increased by LockInfo.V_{ref}. And then, tid of T is added to successorList of Lock.V.

If false, T is added to successorList of Lock.T_{lw} if Lock.T_{lw} is not NULL.

(2) Lock.T_{lw} is assigned T.

For write lock release, see Algorithm 4, 1 to 6 lines. The basic processing logic is as follows:

(1) If Lock.T_{lw} and T are the same, Lock.T_{lw} is assigned NULL.

For read lock release, see Algorithm 4, 7 to 38 lines. The basic processing logic is as follows:

If Lock.T_{lw} is NULL, Lock.V_{ref} is decreased by one, and Lock.V_{wait} is assigned zero.

If Lock.T_{lw} is not NULL, successorList of $L.V$ is traversed sequentially to find the first transaction (assume T_{found}) whose prednum is greater than zero. If T_{found} is found, prednum of T_{found} is decreased by one. Finally, T_{found} is added to ReadyQ ueue immediately if its prednum is zero.

For read lock acquire, see Algorithm 3, 28 to 41 lines. The basic processing logic is as follows:

(1) If Lock.T_{lw} is not NULL, prednum of T is increased by one, and T is added to successorList of Lock.T_{lw}
(2) Lock.V_{ref} is increased by one

Why does successor List of virtual transaction store tid? When one transaction is completed, its related resources are released, and thereby, its pointer is invalid. When a virtual transaction (assume V_1) wakes the first successor transaction (assume T_1) whose prednum is greater than 0, T_1 may not be removed from V successor List of V_1 if prednum is still greater than 0 after decreased. When V_1 performs the wake-up operations again, V_1 will acquire the first successor transaction whose prednum is greater than 0 again. So, if V successor List stores pointers to transactions, V_1 needs to operate these pointers (of course, includes T_1), while the pointer of T_1 may be invalid because of other transaction's waking operations. In addition, TransMap is used to track all unfinished normal transactions, and its format is $\langle$tid, $p_t\rangle$ (tid for transaction ID and p_t for transaction pointer). When a virtual transaction performs wake operations, it will use TransMap to detect whether its successor transaction is active. Hence, the virtual transaction mechanism is running correctly and smoothly.

In a word, these techniques incur far less overhead than maintaining a traditional lock manager and timely detect which transactions should inherit released locks.

3. Dependence-Cognizant Locking

In DCLP, lock is a four-tuple: $(T_{lw}, V_{\text{ref}}, V_{\text{wait}}, V)$. A global map of transaction requests (called TransMap) is maintained, tracking all active transactions with unorder. Because of only detecting whether one transaction is still active, std: : unorder_map is adopted for the possibly best performance.

In order to facilitate the explanation of DCLP, this paper assumes that each partition has one lock thread to run DCLP.

```
     Input: T: a transaction pointer
     Output: Granted: 1 if granted, otherwise 0
 (1) Granted: = 1;
 (2) foreach key ∈ T ⟶ writeSet do
 (3)   LockInfo = data[hash(key)];
 (4)   if T ≡ LockInfo.T_lw then
 (5)     if 0 < LockInfo.V_ref then
 (6)       l: = LockInfo.V ⟶ successor List;
 (7)       if 0 == LockInfo.V_wait then
 (8)         l ⟶ reuse;
 (9)       end
(10)       l←T ⟶ tid;
(11)       LockInfo.V_wait + +;
(12)       T ⟶ prednum+ = LockInfo.V_ref;
(13)       LockInfo.V_ref = 0;
(14)       Granted = 0;
(15)     else
(16)       if NULL ≡ LockInfo.T_lw then
(17)       l: = LockInfo.T_lw ⟶ successorList;
(18)       if l ⟶ empty‖T ≡ l ⟶ back then
(19)         Granted = 0;
(20)         l←T;
(21)         T ⟶ prednum + +;
(22)       end
(23)     end
(24)   end
(25)   LockInfo.T_lw = T;
(26)   end
(27) end
(28) foreach key ∈ T ⟶ readSet do
(29)     LockInfo = data[hash(key)];
(30)     if T ≡ LockInfo.T_lw then
(31)       if NULL ≡ LockInfo.T_lw then
(32)         l: = LockInfo.T_lw ⟶ successorList;
(33)         if l ⟶ empty‖T ≡ l ⟶ back then
(34)           T ⟶ prednum + +;
(35)           l←T;
(36)           Granted = 0;
(37)         end
(38)       end
(39)     end
(40)     LockInfo.V_ref + +;
(41)   end
(42) TransMap←⟨T ⟶ tid, T⟩;
```

Algorithm 3: Lock acquire with DCLP.

```
     Input: T: a transaction pointer
 (1) foreach key ∈ T ⟶ writeSet do
 (2)   LockInfo = data[hash(key)];
 (3)   if T ≡ LockInfo.T_lw then
 (4)     LockInfo.T_lw: = NULL;
 (5)   end
 (6) end
 (7) foreach key ∈ T ⟶ readSet do
 (8)     LockInfo = data[hash(key)];
 (9)     if NULL ≡ LockInfo.T_lw then
(10)       LockInfo.V_ref − −;
(11)       LockInfo.V_wait = 0;
(12)     else
(13)       L: = LockInfo.V ⟶ successorList;
(14)       grantedLock: = 0;
(15)       while !L ⟶ empty do
(16)         got: = TransMap.find(L.front);
(17)         if got ≡ TransMap.end then
(18)           E: = got ⟶ second;
(19)           if 0 ≡ E ⟶ prednum then
(20)             L ⟶ popFront;
(21)           else
(22)         E ⟶ prednum − −;
(23)         grantedLock = 1;
(24)         if 0 ≡ E ⟶ prednum then
(25)           ReadyQueue←E;
(26)         end
(27)         break;
(28)       end
(29)     else
(30)       L.popFront;
(31)     end
(32)   end
(33)   if 0 ≡ granted then
(34)   LockInfo.V_ref − −;
(35)   end
(36)     LockInfo.V_wait = L ⟶ size;
(37)   end
(38) end
(39) foreach E ∈ T ⟶ successorList do
(40)   E ⟶ prednum − −;
(41)   if 0 ≡ E ⟶ prednum then
(42)     ReadyQueue←E;
(43)   end
(44) end
(45) TransMap.erase(T ⟶ tid);
```

Algorithm 4: Lock release with DCLP.

When a transaction arrives at a partition, it requests locks on records belonging to the partition. According to the type of lock request, the processing logic of lock requests is different. In Algorithm 3, 2 to 27 lines correspond to the write lock processing, while 28 to 41 lines correspond to the read lock processing. Write locks are granted to the requesting transaction if $V_{\text{ref}} = 0$ and $T_{lw} = \text{NULL}$ (or is itself). Similarly, read locks are granted if $T_{lw} = \text{NULL}$ (or is itself).

For correctness, one transaction must be added to TransMap after requesting all lock requests. Because of the sequential processing mode, the requesting of the locks and the adding of the transaction to the map can be treated as an atomic operation. As for predetermining the read set and write set of transactions in advance, Thomson [9] had proposed a solution method by allowing a transaction to prefetch whatever reads it needs to (at no isolation) for it to figure out what data it will access before it enters the critical section.

Similar to VLL, when one transaction completes all lock requests, it can be tagged with only one of the three states: free, blocked, and waiting. The specific meanings of these three states are as follows:

> Free means that a transaction has acquired all locks whether it is a single-partition or multi-partition transaction

Blocked means that a transaction has not acquired all of its lock requests immediately upon requesting them

Waiting means that a transaction could not be completely executed without the result of an outstanding remote read request

For a single-partition transaction, its state must be either free or blocked, and thereby, only one state transition is from blocked to free. For a multipartition transaction, its state can be any one of the three states. So, there are two state transition paths. One is from waiting to free if all remote readings are successfully returned. The other is first from blocked to waiting if all ungranted locks are granted and then to free. Only free transactions can begin execution immediately.

DCLP records the dependencies between transactions and thereby has the ability to timely detect unblocked transactions. When one transaction is completed, it is easy to hand over its locks directly to its direct successors by decreasing prednum of its direct successors (see Algorithm 4 for more details).

For VLL, with an increase of active transactions, the probability of a new transaction being tagged free decreases. For good performance, the number of active transactions should be limited. However, it is difficult to tune the number of active transactions since different levels of contention demand different thresholds. So, a threshold is set by the number of active but blocked transactions since high-contention workloads will reach this threshold sooner than low-contention workloads.

However, the fixed threshold brings two defects, and they are as follows:

(1) For most of the real workloads, the contention is not fixed, but is changing from time. Therefore, it is difficult to select a fixed value of this threshold.

(2) When encountering the workloads that include numerous fragmented continuous conflict transactions, the performance is greatly affected. It is because that once the threshold is exceeded, new transactions make no progress even if they are free.

For DCLP, the above two defects have been completely overcome. First, there is no need to set a threshold for active but blocked transactions. DCLP tracks the direct conflict between transactions, and thus, blocked transactions can be unblocked and executed immediately. A preallocated and variable-length list attaches each transaction to store all its direct successors. Hence, compared with VLL, DCLP can nonblocking handle more transactions without being restricted by the threshold of blocked transactions.

However, in consideration of memory and performance, the number of active transactions should be set a threshold, especially when encountering workloads with long transaction execution time.

Figure 1 depicts an example execution trace for a sequence of transactions. Free transactions (here A) are pushed directly into ReadyQueue. Transactions B and A have RAW dependencies on key x, and thus, B is termed blocked. Meanwhile, B is added to successorList of A. The end of A will unblock B and then push B to ReadyQueue. The processing logic of C is similar to that of transaction B and will not be expanded in detail.

4. Evaluation

To evaluate DCLP's performance, a large number of experiments are designed to compare DCLP against deadlock-free 2PL (Calvin's deterministic locking protocol), VLL, and VLL\SCA in a number of contexts. These experiments are divided into two groups: single-machine multicore experiments and experiments in which data are partitioned across many commodity machines in a shared-nothing cluster. The single-machine multicore experiments were conducted on a Linux (2.6.32–220.el6) of two 2.3 GHz six-core Intel Xeon E5-2630 machine with 24 CPU threads and 64 GB RAM. The experiments in a distributed database were conducted on the same server configuration as the single-machine multicore experiments used, connected by a single 10 gigabit Ethernet switch.

As a comparison point, DCLP is implemented inside Calvin. This allows for an apples-to-apples comparison in which the only difference is the locking strategy.

The same configuration as [5] was used: devote 3 out of 8 cores on every machine to those components that are completely independent of the locking scheme, and devote the remaining 5 cores to worker threads and lock management threads. For all experiments, we stationary use fixed 4 work threads and one lock thread.

Although OCC (or MVOCC) shows greater advantages than PCC in short-transaction workloads for MMDB, PCC is still widely adopted by many databases. This paper focuses on PCC and single data version (not MVCC). Therefore, OCC and MVCC are not within the scope of this paper to discuss.

4.1. Multicore, Single-Server Experiments

4.1.1. Standard Microbenchmark Experiments. Two experiments are presented in this section:

The first experiment is called short microbenchmark. Each microbenchmark transaction reads 10 records and updates a value at each record. Of the 10 records accessed, one is chosen from a small set of "hot" records, and the rest are chosen from a larger set of 'cold' records. The contention levels are tuned by varying the size of the set of hot records. The set of cold records is always large enough so that transactions are extremely unlikely to conflict. In addition, the term contention index is used to represent contention levels. For example, if the number of hot records is 1000, the contention index would be 0.001.

The second experiment is called long microbenchmark. The only difference between long microbenchmark and short microbenchmark is that the former consumes a certain amount time to calculate after reading each record (default provided by Calvin).

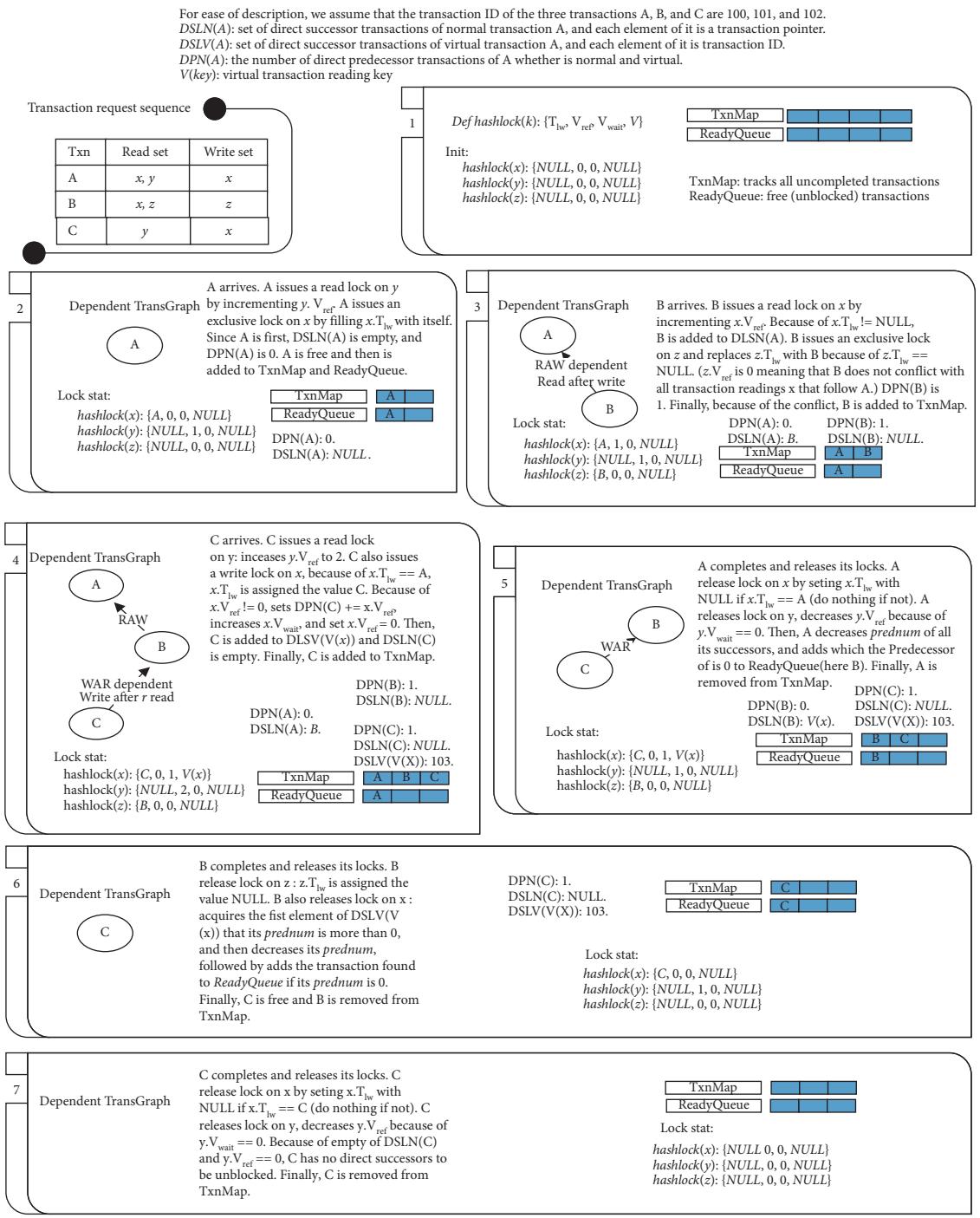

Figure 1: Example execution of a sequence of transactions A, B, and C using DCLP. Each transaction's read and write set is shown in the top left box.

As shown in Figure 2, when contention is low, DCLP, VLL, and VLL\SCA yield near-optimal transaction throughput, while only almost 50% of their transaction throughput can be provided by deadlock-free 2PL. The reason is that VLL and VLL\SCA almost completely eliminate the overhead of latch acquisitions and linked list operations.

Why does DCLP gain such high transaction throughput in low contention? There are three reasons. First, DCLP also adopts acquiring all locks at once for reducing the overhead of latch acquisitions. Second, the hash lock almost completely eliminates the overhead of primary key storage. Third, fragmented storage and virtual transaction effectively eliminate the overhead of linked list operations. Low contention means that the overhead to maintain the conflict information between transactions is negligible.

As contention increases, all decrease, while VLL presents the trend of the first decline and then is almost stable.

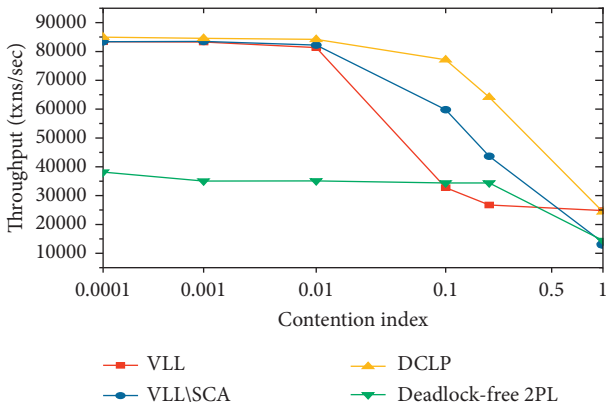

Figure 2: Transaction throughput with short microbenchmark vs. contention under a deadlock-free workload.

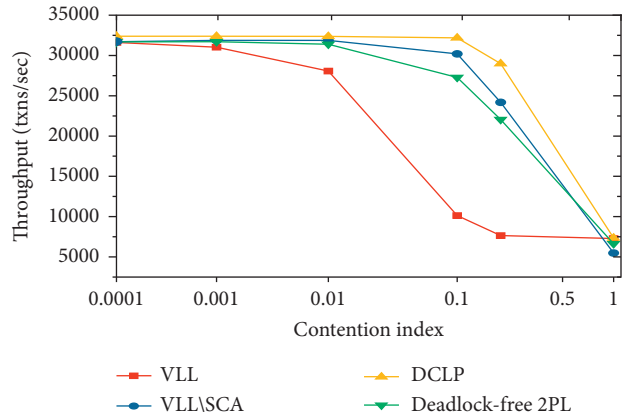

Figure 3: Transaction throughput with long microbenchmark vs. contention under a deadlock-free workload.

Without tracking conflict information, blocked transactions can only be executed serially, and then VLL's performance falls quickly. With tracking conflict information, the higher the contention, the more useful the conflict information. So, the performance of the remaining three mechanisms falls slowly.

Figure 3 shows the trend of the transaction throughput of the four locking mechanisms is very similar to the short microbenchmark results, and DCLP still performs well. However, the transaction throughput of all the four locking mechanisms receives a significant impact and is only about 40% of their throughput in short microbenchmark. The reason is that the executing time of long transactions becomes the primary performance bottleneck rather than the lock overhead.

4.1.2. Modified Microbenchmark Experiments. The standard microbenchmark is write-only, and therefore, these four locking mechanisms cannot be fully evaluated. For a comprehensive and complete evaluation, some modified microbenchmarks by combining different ratios of read to write and different degrees of conflict were designed.

The first set of modified microbenchmarks only just replaces write-only with read-only, with the purpose to prove that DCLP is efficient and has the same high performance as VLL and VLL\SCA on read-only workloads.

As expected, Figure 4 confirms that DCLP has almost the same high performance as VLL and VLL\SCA with respect to read-only workloads no matter how high or low the contention is.

The second set of modified microbenchmarks varies contentions with a fixed ratio of read to write. As we all know, for some real transaction processing systems, reading more and writing less is a very important feature. Considering some of the default workloads of YCSB [10], a special workload with 95% read and 5% write is chosen for this set of experiments.

As shown in Figure 5, when contention is low (actually only 5% write means that factual contention is lower), DCLP, VLL\SCA, and VLL have almost the same high performance. Low contention does not fully demonstrate the advantage of these three locking mechanisms in detecting unblocked transactions. As contention increases, DCLP

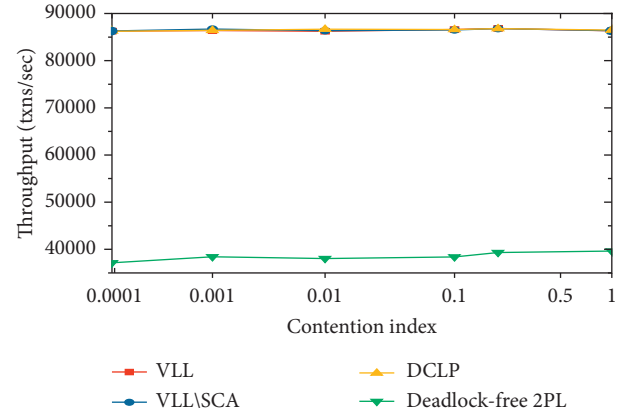

Figure 4: Transaction throughput with read-only short microbenchmark vs. contention under a deadlock-free workload.

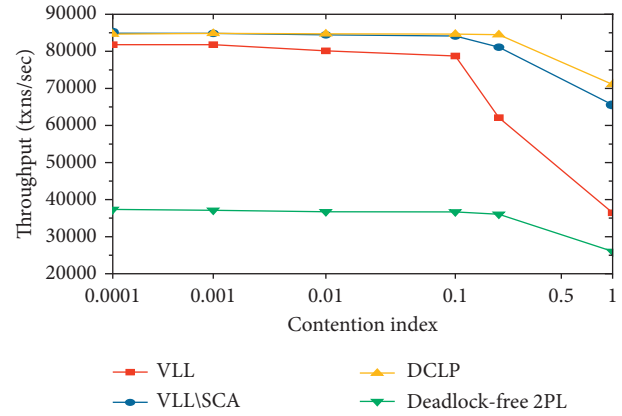

Figure 5: Transaction throughput with modified short microbenchmark (5% write and 95% read) vs. contention under a deadlock-free workload.

gradually demonstrates this advantage, and thereby, its performance is the highest. Due to the loss of the ability to detect unblocked transactions, VLL is lower than VLL\SCA. As for deadlock-free 2PL, its performance is still the lowest because of its high overhead of locking processing.

The third set of modified microbenchmarks varies both contention and write percent. Two representative contention points are selected: 0.1 for the high-contention and 0.0001 for the low-contention point (as [5]). Due to similarity and simplification, short microbenchmark with low contention was executed, but no longer presented.

For short transactions, the high overhead of locking processing is the primary bottleneck that affects transaction throughput. Therefore, DCLP and VLL\SCA overtake deadlock-free 2PL. As for VLL, without the ability to detect unblocked transactions, its performance dramatically decreases. Figure 6 presents these conclusions.

In real transaction processing systems, long chain dependence between transactions is also an important feature, for example, exchange rate transactions in which the exchange rate is being read continuously along with regular update with a small update interval. So, we design the last set of experiments to test such workloads. Similar to the standard short microbenchmark, each transaction has one hot record and nine cold records, and any of the cold records of each transaction is read and updated. The difference is that the size of the hot dataset is fixed to 1. And then, we tune the ratio of read to write to simulate the depth of dependence between transactions.

Table 2 shows the results of the last set of experiments. Every transaction operates the only one hot record meaning that one transaction blocks all its subsequent transactions. So, for the four locking mechanisms, the efficiency in detecting unblocking transactions can improve their transaction throughput. The conclusion is that DCLP is the most efficient in detecting unblocked transactions, and the second is VLL\SCA. Without the ability in detecting unblocked transactions, VLL achieves poor transaction throughput. In addition, although it has the ability in detecting unblocked transactions, deadlock's performance is even lower than that of VLL. It also verifies from another perspective that deadlock-free 2PL is expensive.

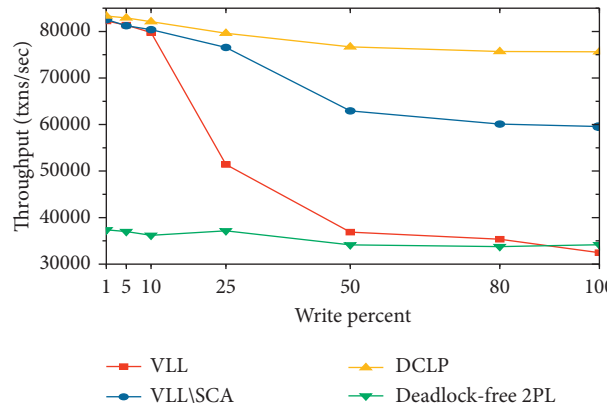

Figure 6: Transaction throughput with the modified short microbenchmark (high contention) under a deadlock-free workload, varying the write percent.

Table 2: Transaction throughput with the modified microbenchmark under a deadlock-free workload, varying read and write interdependent depth.

Throughput (txns/sec)	9 depth	19 depth	99 depth
VLL	18315.4	23381.7	56825.4
VLL\SCA	20016.2	33874.2	65905.2
DCLP	27632.5	51659.9	95499.4
Deadlock-free 2PL	15261.1	21644.7	38026.6

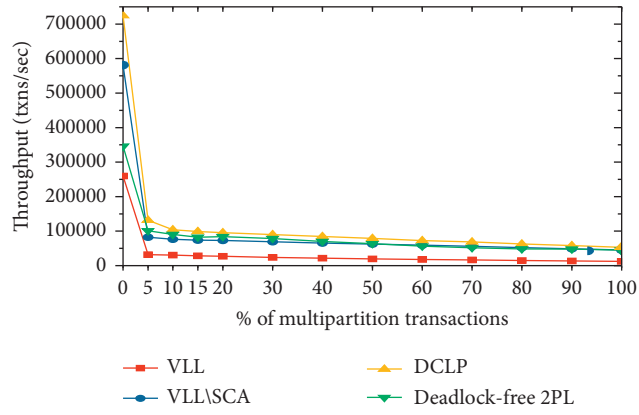

Figure 7: Microbenchmark throughput with high contention under a deadlock-free workload, varying how many transactions span multiple partitions.

4.2. Distributed Database Experiments

4.2.1. Standard Microbenchmark Experiments. The same standard short microbenchmark as in Section 4.1.1 is used. Both contention and percentage of multipartition transactions are varied. Two representative contention points are also selected: 0.1 for the high-contention and 0.0001 for the low-contention point (same as [5]). In addition, these experiments were run on 8 machines.

Although the data partition strategy is different from that of [5], there is no doubt that SCA is extremely important no matter contention is high or low. Under high contention, SCA improves the performance of VLL by at least 55% (maximum of up to 73%). Under low contention, SCA improves the performance of VLL by at least 40% (maximum of up to 67%) when the percentage of multipartition transactions is greater than 20%.

Figure 7 shows that DCLP is the best one and has about 15% performance advantage over VLL\SCA. It is easy to understand that DCLP is more efficient in detecting unblocked transactions. Because of the long executing time, the performance of all four locking mechanisms presents a downward cliff from non-multipartition transactions to 5%. When the percentage is greater than 5%, all transactions almost can only be executed serially so that the performance is almost stable.

Figure 8 also shows that DCLP is the best one. Compared with VLL\SCA, DCLP has about 2% to 21% performance advantage. Although contention is low, with the increase of multipartition transactions and the only one lock thread, more and more transactions are blocked. The performance gap between DCLP, VLL, and VLL\SCA is mainly due to the efficiency of detecting unblocked transactions.

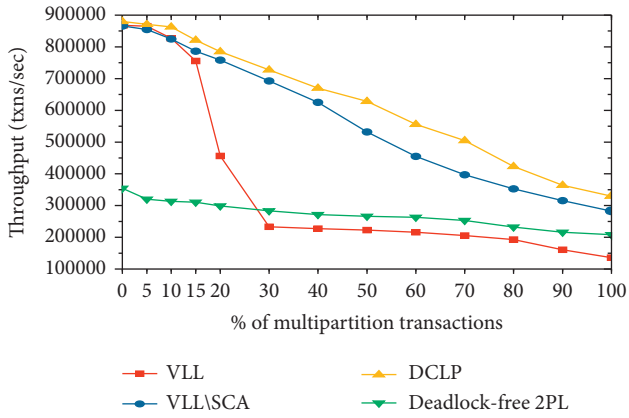

Figure 8: Microbenchmark throughput with low contention under a deadlock-free workload, varying how many transactions span multiple partitions.

The scalability of three of the four locking mechanisms is also tested at low contention when there are 10% and 20% multipartition transactions. These locking mechanisms are deadlock-free 2PL, VLL\SCA, and DCLP. We scale from 2 to 8 machines in the cluster. Figure 9 shows that DCLP achieves the same linear scalability as VLL\SCA and deadlock-free 2PL and still has the best performance at scale.

4.2.2. TPC-C Experiments. The same TPC-C benchmark as in [5] is used. In order to vary the percentage of multipartition transactions in TPC-C, the percentage of new order transactions that access a remote warehouse is varied. 96 TPC-C warehouses were divided across the same 8-machine cluster described in the previous sections, but there is a subtle difference that all the four lock mechanisms partitioned the TPC-C data across 8 twelve-warehouse partitions (one per machine and one partition has twelve warehouses). Each partition corresponds to one machine. In the end, we would expect to achieve similar performance if we were to run the complete TPC-C benchmark.

Figure 10 shows the transaction throughput results of the four locking mechanisms. Overall, the performance of different locking mechanisms is very similar to the result of the high-contention microbenchmark with multipartition transactions. DCLP still has the best performance and has maximum of about 5% performance advantage over VLL\SCA.

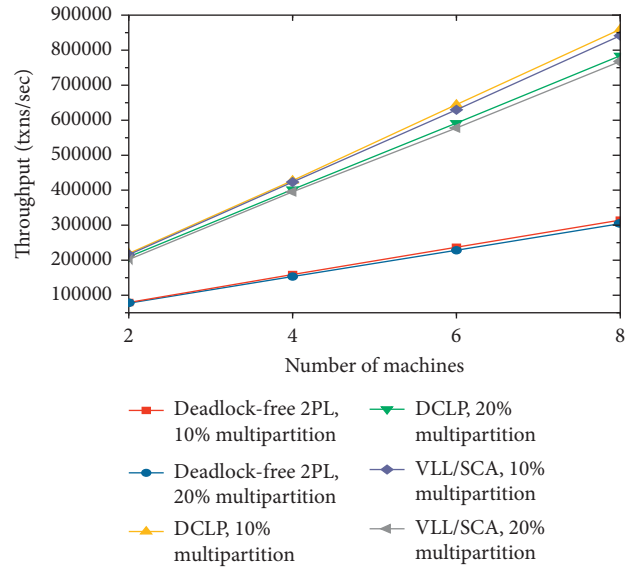

Figure 9: Scalability under a deadlock-free workload.

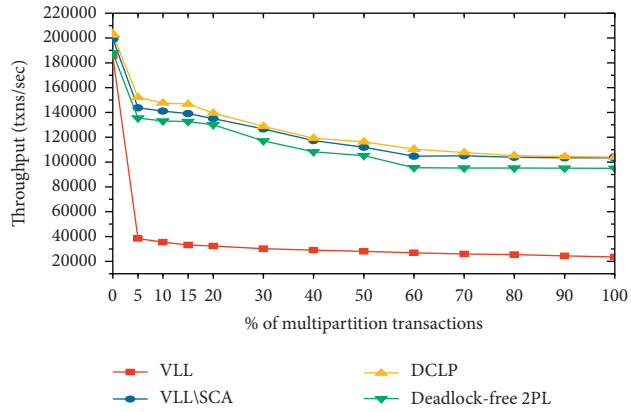

Figure 10: Transaction throughput with TPC-C under a deadlock-free workload, varying how many transactions span multiple partitions.

5. Related Work

5.1. Reducing Lock Acquisitions. These research works mainly reduce the number of lock acquisitions by redesigning the processing logic of LM, such as [4, 6, 7, 11]. Speculative lock inheritance (SLI) [4] allows a completing transaction to pass on some locks (hot locks, frequently acquired in a short time) which it acquired to transactions which follow. This successfully avoids a pair of release and acquire calls to the lock manager for each such lock. However, SLI only performs well on hot locks because SLI does not optimize LM itself and only just reduces the number of calls to LM. RHS [7] adopts the RAW pattern and barrier synchronization to greatly reduce the use of latches and thus generally improves the performance. However, it is still in the category of traditional lock manager (optimized).

5.2. Job Scheduling. In other communities except for the database community, there has been extensive research on scheduling problems in general. These research works are to minimize time costs from four different perspectives and have achieved better performance. These four aspects are, respectively, to minimize the sum of completion time [12], the latest completion time [13], the completion time variance [14, 15], and the waiting time variance [16]. However, the assumption of these research works that each processor/worker can be used by only one job at a time is not applicable to the database community, while locks can be held in shared and exclusive modes in the database.

5.3. Dependency-Based Scheduling. Scheduling transactions with dependencies among them has been studied for many years. Tian et al. [17] proposed a contention-aware transaction scheduling algorithm, which captures the contention and the dependencies among concurrent transactions. It is different from most existing systems which rely on a FIFO (first in, first out) strategy to decide which transactions to grant the lock to. Although it achieves better performance than FIFO, it may cause some problems to the applications based on FIFO because [17] granted the lock to the transaction with the largest dependency set, rather than to the first requested transaction. However, in real applications, in some situations, applications want databases to execute transactions as their sending order.

5.4. Optimistic Concurrency Control. Optimistic concurrency control (OCC) is a popular concurrency control due to its low overhead in low-contention settings [18–26]. Huang et al. [18] presented two optimization techniques, commit-time updates and timestamp splitting, which can dramatically improve the high-contention performance of OCC. TicToc [25] used a technique called data-driven timestamp management to completely eliminate the timestamp allocation bottleneck. AOCC [19] adaptively chose an appropriate tracking mechanism and validation method to reduce the validation cost according to the number of records read by a query and the size of write sets from concurrent update transactions.

5.5. Multiversion Concurrency Control. The main advantage of MVCC is that it potentially allows for greater concurrency by permitting parallel accesses on different versions. Many MMDBs are in favor of MVCC (e.g., Hekaton and MemSQL). They utilize mechanisms commonly known as MVOCC or MV2PL [27, 28]. Wu et al. [27] made a comprehensive empirical evaluation and identified the limitations of different designs and implementation choices.

5.6. Lightweight Locking. Lightweight locking also attracts many researchers. In some situations, [5, 9] almost eliminate all the expensive overhead of traditional lock managers. In VLL, LM is extremely simplified by replacing the lock request list with two simple semaphores containing the number of outstanding requests for that lock (C_x for write requests and C_s for read requests). However, VLL performs badly on high-contention workloads, while VLL\SCA does not have the ability to schedule unblocked transactions without delay.

From previous literature about the optimization and redesign of LM, we can conclude that designing optimal LM for databases has remained an open problem. Considering that almost all existing systems rely on a FIFO (first in, first out) strategy to decide which transactions to grant the lock to, we decide to follow the FIFO strategy. In addition, acquiring all locks at once can avoid deadlock and then achieve good performance. In this paper, we blend the advantages of dependency-based scheduling and lightweight locking to redesign LM for reducing the locking space and providing higher performance.

6. Conclusion

In this paper, we presented dependence-cognizant locking (DCLP) combined with refined scheduling with lightweight locking that provides a fast and efficient concurrency control for database systems. For MMDB, the lock and latch cost of the traditional lock mechanism is expensive. In DCLP, we manage transactions by dependency chains. It eliminates most of the cost of latch, and more importantly, transactions can be awakened immediately when their required data are unlocked. Furthermore, for better lock transfer performance, we proposed two optimization techniques. One is named fragment storage of lock mechanism which ensures transactions getting successors in place. The other is the virtual transaction mechanism (VT for short), compressing continuous read requests into a special read request as VT, which reduces the scheduling complexity significantly, especially in heavy workloads. Experiments show that DCLP achieves better performance than deadlock-free 2PL and VLL\SCA, without inhibiting scalability. In the future, we will intend to integrate hierarchical locking approaches, column locking, and row/column hybrid locking mechanism into DCLP and investigate multiversion variants of the DCLP strategy.

Data Availability

All experiments are based on Calvin, which is an open-sourced transactional database, and its URL is https://github.com/yaledb/calvin. Except DCLP, the benchmarks and the other three lock mechanisms used in this paper have been implemented in Calvin. This paper has given the pseudocode of the algorithm DCLP. Further help is available from the first author upon request.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

Acknowledgments

This work was supported by the National Natural Science Foundation of China under Grant nos. 61672434, 61472321, and 61732014.

References

[1] S. Harizopoulos, D. J. Abadi, S. Madden et al., "Oltp through the looking glass, and what we found there making databases work: the pragmatic wisdom of michael stonebraker," 2018.

[2] P. Larson, "High-performance concurrency control mechanisms for main-memory databases," *Proceedings of Vldb Endowment*, vol. 24, 2011.

[3] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki, "Data-oriented transaction execution," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 928–939, 2010.

[4] R. Johnson, I. Pandis, A. Ailamaki et al., "Improving OLTP scalability using speculative lock inheritance," *Proceedings of the VLDB Endowment*, vol. 2, no. 1, pp. 479–489, 2009.

[5] K. Ren, A. Thomson, and D. J. Abadi, "Lightweight locking for main memory database systems," *Proceedings of the Vldb Endowment*, vol. 6, no. 2, pp. 145–156, 2012.

[6] A. M. Joshi, G. M. Lohman, A. Sernadas et al., "Adaptive locking strategies in a multi-node data sharing environment," in *Proceedings of the International Conference on Very Large Data Bases*, Catalonia, Spain, February 1991.

[7] H. Jung, H. Han, A. Fekete, G. Heiser, and H. Y. Yeom, "A scalable lock manager for multicores," *Acm Transactions on Database Systems*, vol. 39, no. 4, pp. 1–29, 2014.

[8] T. Horikawa, "Latch-free data structures for DBMS: design, implementation, and evaluation," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data. ACM*, New York, NY, USA, June 2013.

[9] A. Thomson, "VLL: a lock manager redesign for main memory database systems," *Vldb Journal the International Journal of Very Large Data Bases*, 2015.

[10] B. F. Cooper, A. Silberstein, E. Tam et al., "Benchmarking cloud serving systems with YCSB," in *Proceedings of the Symposium on Cloud Computing*, pp. 143–154, Indianapolis, IN, USA, June 2010.

[11] X. Yu, G. Bezerra, A. Pavlo et al., "Staring into the abyss: an evaluation of concurrency control with one thousand cores," *Proceedings of the VLDB Endowment*, vol. 7, 2014.

[12] C. He, J. Y.-T. Leung, K. Lee, and M. L. Pinedo, "Improved algorithms for single machine scheduling with release dates and rejections," *4OR-Q J Operational Research*, vol. 14, no. 1, pp. 41–55, 2016.

[13] B.-C. Choi, S.-H. Yoon, S.-J. Chung et al., "Minimizing maximum completion time in a proportionate flow shop with one machine of different speed," *European Journal of Operational Research*, vol. 176, no. 2, pp. 964–974, 2007.

[14] A. M. Krieger and M. Raghavachari, "V-shape property for optimal schedules with monotone penalty functions," *Computers & Operations Research*, vol. 19, no. 6, pp. 533-534, 1992.

[15] X. Cai, "V-shape property for job sequences that minimize the expected completion time variance," *European Journal of Operational Research*, vol. 91, no. 1, pp. 118–123, 1996.

[16] S. Eilon and I. G. Chowdhury, "Minimising waiting time variance in the single machine problem," *Management Science*, vol. 23, no. 6, pp. 567–575, 1977.

[17] B. Tian, J. Huang, B. Mozafari et al., "Contention-aware lock scheduling for transactional databases," *Proceedings of the VLDB Endowment*, vol. 11, no. 5, 2018.

[18] Y. Huang, W. Qian, E. W. Kohler et al., "Opportunities for optimism in contended main-memory multicore transactions," *Proceedings of the VLDB Endowment*, vol. 12, no. 4, 2020.

[19] J. Guo, P. Cai, J. Wang, W. Qian, and A. Zhou, "Adaptive optimistic concurrency control for heterogeneous workloads," *Proceedings of the VLDB Endowment*, vol. 12, no. 5, pp. 584–596, 2019.

[20] M. E. Schule, L. Karnowski, J. Schmeißer et al., "Versioning in main-memory database systems: from musaeusdb to tardisdb," in *Proceedings of the 31st International Conference on Scientific and Statistical Database Management*, pp. 169–180, Santa Cruz, CA, USA, July 2019.

[21] T. Zhu, Z. Zhao, F. Li et al., "SolarDB," *ACM Transactions on Storage*, vol. 15, no. 2, pp. 1–26, 2019.

[22] B. Ding, L. Kot, and J. Gehrke, "Improving optimistic concurrency control through transaction batching and operation reordering," *Proceedings of the VLDB Endowment*, vol. 12, no. 2, pp. 169–182, 2018.

[23] H. Lim, M. Kaminsky, and G. David, "Andersen. cicada: dependably fast multi-core in-memory transactions," in *Proceedings of the ACM international conference on management of data. ACM*, Chicago IL, USA, May 2017.

[24] T. Wang and H. Kimura, "Mostly-optimistic concurrency control for highly contended dynamic workloads on a thousand cores," *Proceedings of the VLDB Endowment*, vol. 10, no. 2, pp. 49–60, 2016.

[25] X. Yu, A. Pavlo, D. Sanchez et al., "Tictoc: time traveling optimistic concurrency control," in *Proceedings of the 2016 International Conference on Management of Data*, pp. 1629–1642, San Francisco, CA, USA, June 2016.

[26] H. T. Kung and J. T. Robinson, "On optimistic methods for concurrency control," *ACM Transactions on Database Systems*, vol. 6, no. 2, pp. 213–226, 1981.

[27] Y. Wu, J. Arulraj, J. Lin, R. Xian, and A. Pavlo, "An empirical evaluation of in-memory multi-version concurrency control," *Proceedings of the VLDB Endowment*, vol. 10, no. 7, pp. 781–792, 2017.

[28] G. Weikum and Gottfried Vossen, *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*, ACM, New York NY, USA, 2001.