Interactive Transaction Processing for In-Memory Database System

Tao Zhu, Donghui Wang, Huiqi Hu(✉), Weining Qian, Xiaoling Wang, and Aoying Zhou

East China Normal University, Shanghai, China
{tzhu,donghuiwang}@stu.ecnu.edu.cn,
{hqhu,wnqian,ayzhou}@dase.ecnu.edu.cn, xlwang@sei.ecnu.edu.cn

Abstract. In-memory transaction processing has gained fast development in recent years. Previous works usually assume the one-shot transaction model, where transactions are run as stored procedures. Though many systems have shown impressive throughputs in handling one-shot transactions, it is hard for developers to debug and maintain stored procedures. According to a recent survey, most applications still prefer to operate the database using the JDBC/ODBC interface. Upon realizing this, the work targets on the problem of interactive transaction processing for in-memory database system. Our key contributions are: (1) we address several important design considerations for supporting interaction transaction processing; (2) a coroutine-based execution engine is proposed to handle different kinds of blocking efficiently and improve the CPU usage; (3) a lightweight and latch-free lock manager is designed to schedule transaction conflicts without introducing many overhead; (4) experiments on both the TPC-C and a micro benchmark show that our method achieves better performance than existing solutions.

Keywords: Transaction · Concurrency control · Network interaction

1 Introduction

In-memory database systems have gained a rapid development in recent years. These systems store the entire database in the main memory and totally removes the performance bottleneck resulted from slow disk I/O. And the major design consideration has evolved into better utilization of the multi-core and multi-socket CPUs [3]. To achieve that, in-memory databases usually **assume** the one-shot transaction model (see Fig. 1), where transactions are run as stored procedures and no client-server interaction was allowed once a transaction got started. One-shot transactions do not worry about I/O-related stall any more, and can keep being processed to the completion if no conflict access is witnessed. Based on that, many lightweight concurrency control schemas are proposed [8,15,16]. All introduce little maintaining overhead in identifying access conflicts. On the other hand, conflicts are resolved by simply aborting and retrying transactions,

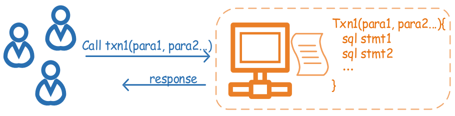


Fig. 1. The one-shot transaction

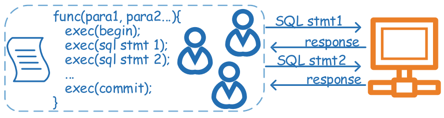


Fig. 2. The interactive transaction

which is actually quite time-consuming [17,18]. Thus, the efficiency of these methods is promised based on the fact that most one-shot transactions finish execution in a short time and conflict with each other seldomly.

Although, one-shot transaction processing achieves impressive throughput, another very important kind of workload is rarely studied, namely the interactive transaction processing (see Fig. 2). In this case, applications operate database system with SQLs and JDBC/ODBC interface. Transactions are invoked by sending SQLs one-by-one to the server. A recent survey [10] shows transaction processing in interactive way is much more common than that in one-shot way. It is reported that 54% responders never or seldom use stored procedure in their DBMSs. Only 16% responders have more than half of transactions ran as stored procedures. Several reasons for the situation are: (1) stored procedures are difficult to maintain and debug; (2) they lack portability, making it hard to deploy applications on different platforms and databases.

Such fact has made us think about how to design in-memory database system for interactive transaction processing. We conclude the major differences between the interactive model and the one-shot model are: (1) a transaction can be stalled by network I/O, which requires the execution engine to handle the network I/O blocking efficiently; (2) a transaction lasts much longer since network latencies are included, which results in more access conflicts. The concurrency control is expected to be efficient in both identifying and resolving conflicts. Our contributions can be summarized as follows: (1) we examine the design space, and conclude several design considerations in implementing interactive transaction processing. (2) A new execution model is designed to interleave transaction execution efficiently. It can fully utilize the CPUs and does not waste much time on handling different kinds of blocking. (3) A lightweight and latch-free lock manager, named as *iLock*, is proposed to schedule conflict operations efficiently. It introduces little overhead regardless of the workload containing lots of contention or not. (4) Experiments on well-known benchmarks show our method achieves the better throughput than existing techniques.

The paper is organized as follows: Sect. 2 analyzes the properties of interactive transaction processing and gives the design consideration. Section 3 presents a new execution engine which efficiently running concurrent transactions on multi-core hardwares. Section 4 designs a new lock manager and analyzes its correctness. Section 5 shows the experiment results. Related works are discussed in Sect. 6 and the work is concluded in Sect. 7.

2 Design Consideration

The section firstly emphasizes two important facts about the interactive transaction workload, and then gives our design considerations.

Frequent Network I/O Blocking. As a transaction is executed by interactively sending SQLs to the server, its processing is frequently blocked by network I/O. As a result, a transaction has itself suspended when a SQL request is finished, and get resumed after the next request arrives. The database system is responsible for handling transaction's suspending and resuming efficiently.

Long Transaction Duration. Transaction duration is the time from the beginning of a transaction to the end. Given a one-shot transaction accessing tens of record, its processing phase only contains memory access and CPU calculation (the commit phase is not included, which requires one disk write). Therefore, it has a very short duration and can usually be finished in less than 100 µs. On the other hand, an interactive transaction requires multiple network communications between the client and the database system. In a cluster, it takes nearly 100 µs to do a simple message round-trip between two servers through Ethernet. If there is 10 client-server interactions, a transaction will last longer than 1000 µs.

Based on these observations, we consider that the in-memory transaction engine should have the following design principles.

CPU-Efficient Execution Model. Consider that a transaction t_x is running on a CPU core c_i. If t_x is blocked by network I/O, c_i becomes idle. In order to make full use of CPU, c_i ought to process another transaction t_y before the next SQL of t_x is received. Hence, a large number of transaction should run concurrently on a relatively small number of CPU cores. To achieve that, existing systems use the Transaction-To-Thread execution model, where a transaction binds its execution with the same thread during its lifetime. If a thread is blocked and a core becomes idle, another thread would be switched in and run on the same core. However, the limitation is that each time a thread is blocked by network I/O, it results in one *context switch*. For instance, on an *Intel E5-2620 CPU*, it takes typically 8–13 µs to finish one context switch. Such overhead makes the execution model less appealing. Hence, we target on designing a new execution model, which handles the network-related blocking more efficiently.

Lightweight Lock-Based Concurrency Control. Since interactive transactions have the long duration, the risk that two transactions access the same data item is greatly increased. As a result, transaction conflicts happen more frequently. (i) Many in-memory systems [8,15] favor the optimistic concurrency control [7]. But it is not suitable for scheduling interactive transactions. Since the protocol identifies conflicts at the end of a transaction with a validation phase, a client has to process a transaction from the scratch once its validation fails. As mentioned in the above, conflicts happen more frequently under the interactive workload, OCC would waste a plenty of CPU time and network resources on retrying failed transactions. (*ii*) With that in mind, we consider the lock-based concurrency control (i.e. 2PL) as a better choice. Under the lock-based protocol, access conflicts are identified by asking each transaction to acquire locks. Conflict can be resolved by blocking a transaction until the required lock is released. The protocol has the balanced performance in scheduling workloads exhibiting either high lock contention or low. In further, in order to work efficiently in the main memory environment, the lock-based protocol requires a lightweight implementation to reduce the CPU overhead. However, We observe existing solutions do not satisfy the requirement. (*a*) Disk-based systems implement the protocol with a centralized lock table. The structure is proven to have considerable maintaining overhead [5,11]. (*b*) Some in-memory systems optimize the protocol by simplifying lock manager (i.e. row locking [8,11]). It is only efficient in identifying access conflicts. When a conflict is witnessed, it forces a transaction to abort and get retried, which consumes much CPU time. Hence, it is not suitable for the case where the workload contains much contention.

In all, the interactive transaction processing calls for (1) an execution engine, which is efficient in handling blocking caused by network I/O or access conflicts; and (2) a lightweight lock manager, which introduces little overhead in identifying and resolving conflicts.

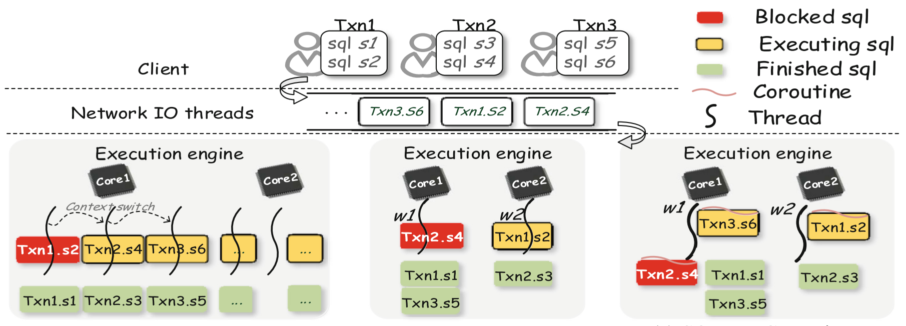
3 Execution Model

This section presents an execution model, which is efficient in handling blocking coming from network I/O and lock conflicts. Overall, our goal is to minimize the number of context switches resulted from blocking. In the following, we start with a simple execution model which is able handle network-related blocking efficiently. Based on that, we take the conflict-related blocking into consideration. And then, a new execution model is given with its features clearly discussed.

3.1 SQL-To-Thread

Here we firstly assume there is no conflict-related blocking. As discuss in the previous section, transaction-to-thread model is not a good design because each time a thread waits for the next SQL request to arrive, it is blocked and generates one context switch. A possible design is to bind a SQL request with a thread. As a client interacts with the server in degree of SQL request, a thread can execute a request to the completion without being blocked. Once a request is processed, the thread continues to process a SQL of another transaction.

In the SQL-To-Thread model, a batch of I/O threads are responsible for communicating with clients. Several working threads are created to handle the transaction execution. The number of working threads **is equivalent to** the number of available CPU cores. Once a request is received, an I/O thread would push it into a task pool. Each working thread keeps pulling a request from the

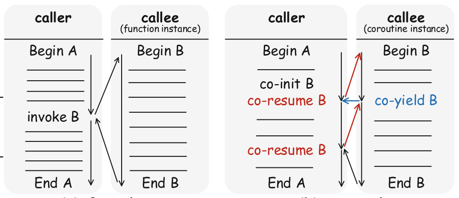


(a) Transaction-To-Thread (b) SQL-To-Thread (c) SQL-To-Coroutine

Fig. 3. Different kinds of execution models.

co-init	allocate resources required by a coroutine instance
co-resume	resume the execution of a coroutine instance
co-yield	exit the current coroutine instance temporarily

Fig. 4. Interface for coroutine usage



(a) function (b) coroutine

Fig. 5. Coroutine mechanism

task pool for processing. After a request is processed, a working thread informs an I/O thread to send results back to the client, and continues to pull a request from the task pool. Figure 3(b) gives an example. Two working threads w_1, w_2 are started on two cores. Here, w_1 processes SQL s1 of transaction Txn1 in the first, and then Txn2.s4. When the next request Txn1.s2 is arrived, w_1 is occupied by Txn2.s4 but w_2 is available. Hence, w_2 begins to process Txn1.s2. In the model, each thread keeps itself busy by servicing requests of different transactions. It eliminates the context switches generated by network I/O.

3.2 SQL-To-Coroutine

Conflict-Related Blocking. In addition to network-related blocking, a transaction can also be blocked by access conflicts. The SQL-To-Thread model cannot sufficiently handle this kind of blocking. Considering a SQL s_1 attempts to acquire a lock l_i and the l_i is already held by another transaction t_y, s_1 can not proceed any more before t_y releases l_i. Facing this situation, s_1 has two choices: (1) abort and retry itself later (only abort s_1, not the whole transaction); (2) wait until the lock is released. The first choice is not acceptable. As analyzed in the previous section, an interactive transaction tends to have a long duration. It is very likely that s_1 will be retried for many times before s_1 acquires the lock and all those failed attempts only waste CPU time without any other return. Hence, we prefer the second choice, which let s_1 wait for t_y to release l_i.

SQL-Wait. The next problem is how to make a SQL wait in a thread. Two common solutions are possible: (1) busy-waiting and (2) condition-waiting. In busy-waiting, the thread keeps checking whether the lock is released. Obviously, it is even worse than the abort-retry strategy, as it not only wastes lots of CPU time on checking the lock condition, but also prevents other requests from being processed. In condition-waiting, the thread gives up the CPU and is awakened later when t_y releases l_i. Before the thread is waked up, its CPU core becomes idle (recall that exact one thread is created for each core). As a result, using condition-waiting will under utilize the CPU resource.

Ideally, we expect the execution engine to process SQLs in the following way. If s_1 is blocked due to a lock conflict, the thread temporarily leaves its execution, and starts to process the next request. After l_i is released, the thread resumes the execution of s_1 again. To achieve that, we propose a SQL-To-Coroutine model. Figures 4 and 5 illustrates the mechanism of coroutine [6] briefly. As we can see in Fig. 5(b), a caller A uses *co-init* to allocate memory space and do some initializations for a coroutine instance B. The caller invokes *co-resume* to begin the execution of B. In the middle of the execution, B can invoke *co-yield* to exit its execution temporarily. In the back-end, *co-yield* saves B's stack into the memory region allocated by *co-init*. Later, when A calls *co-resume*, B would have its stack copied back into the thread's execution context again. This time, B continues its execution from where the last *co-yield* is called. Essentially, the coroutine is similar with the function if *co-yield* is never used. The difference is that a coroutine instance holds all states and variables before it is ended. It can exit in the middle of its execution and later return to the point where it leaves. More information about coroutine can be found in [6].

The SQL-To-Coroutine model processes each SQL request as a coroutine instance. During processing, if there is no lock conflict, the coroutine instance executes the same as a normal function call. Otherwise, if lock conflict does exist, *co-yield* can be called to exit the current execution temporarily. Once the coroutine instance yields, the thread begins to process a new request. In later, if the thread is *informed* that the lock has been released, it gets back the pointer of the coroutine instance, and invoke *co-resume* to resume the execution. The informing mechanism would be left to the next section.

Figure 3(c) gives an example for executing SQL requests as coroutines. In the example, Txn2.s4 is blocked due to conflict data access. The thread w_1 calls *co-yield* to exit the execution of Txn2.s4 temporarily. Then w_1 begins to process the next request Txn3.s6 in the task pool. And Txn2.s4 is resumed when another transaction releases the lock that it is waiting on. As we can see, in the SQL-To-Coroutine model, a thread will not be blocked by transaction conflicts.

3.3 Discussion and Refinement

Running SQL requests as coroutine instances also introduces some inherent overheads, which come from two sources: coroutine initialization and switching.

Initialization. co-init allocates about 1 M bytes for a coroutine instance to save its execution context. If it is always called for each request, lots of CPU time are wasted on memory allocation and deallocation. A simple refinement is to build a resource pool on each thread. To process a request, a thread tries to reuse a coroutine instance in its pool. If the pool is empty, *co-init* is called to create a new instance. After a request is finished, its coroutine instance is put back into the pool. Another concern is whether using coroutines would consume lots of memory resource. Obviously, each thread requires an instance for its in-processing request. In addition, an instance is also required for each blocked transaction that is waiting for a lock. Actually, the number of working threads and blocked requests would not get very big. For example, give a system with 30 CPU cores and 70 transactions waiting for locks, it takes about 100 M bytes memory space, which is relatively small as an in-memory database is deployed with more than hundreds of gigabytes memory.

Coroutine Switching. As discussed in the above, a coroutine instance gives up the thread by using *co-yield*, which saves its stack into the preallocated memory region. The thread uses *co-resume* to copy its stack back and continues its execution. A problem is whether such memory copying takes lots of CPU time. On a *Intel E5-2620 CPU*, a micro-benchmark showed that it usually takes about less than 50 ns to finish one switch. In a comparison, it takes about 10 µs to done one context switch. And it takes about several tens of microseconds (µs) to handle a point-get request in a prototype main memory system. Hence, the overhead introduced by coroutine switching is negligible. In addition, switching only happens when a transaction is blocked. If all requests are processed without any conflicts, coroutine switching never happens.

The section discusses how to handle different kinds of blocking efficiently with a SQL-To-Coroutine execution engine. Based on that, we are required to discuss how to schedule concurrent transactions correctly and efficiently.

4 Lock Manager

This section proposes a lightweight, latch-free lock manager, named as *iLock*. It produces little CPU overhead in both identifying and resolving transaction conflicts. In the following, we firstly present related data structures, and then design its acquisition/releasing algorithms, as well as the deadlock avoidance mechanism. In the last, the correctness of iLock is analyzed.

A **record** stores the following information in its header:

– *lock-state*, a 64-bit variable, representing the state of the lock;
– *write-waiter*, a 64-bit variable, encoding who waits for writing the record;

– *read-waiter*, a 64-bit variable, encoding who waits for reading the record;

The first bit of *lock-state* tells the lock mode of the record, using 1 for write-mode and 0 for read-mode. In write mode, the last 63 bits of *lock-state* stores the transaction identifier of the owner. In read mode, the rest bits tells how many transactions have acquired the read lock. Clearly, the record is not locked if *lock-state* = 0, i.e. all bits of the lock state are zeros.

The *i-th* bit of *write-waiter* represents whether any transaction in the *i-th* working thread is trying to acquire the write lock of the record. Similarly, *read-waiter* tells which thread is waiting for the read lock. Concretely, *write-waiter* uses N bits if there are N working threads. Here N is determined by the number of CPU cores used by transaction processing. Since we leverage the SQL-To-Coroutine model and modern servers are usually equipped with no more than 64 cores, 64 bits are enough for most cases. Besides there is no limitation to use a large-sized *write-waiter* to work with an advanced platform equipped with more CPU cores.

A **transaction** has the following fields in iLock.

– *tid*, a unique identifier for a transaction;
– *co-pointer*, the pointer to the current coroutine instance.

Here, *tid* is allocated as a 63-bit positive integer. Note that 0 is not used as *tid* because the number is used by a dummy write lock, which will be explained in the later. *co-pointer* is only used when a transaction has its SQL execution blocked due to lock conflicting.

A **working thread** maintains some thread-local structures:

– *thd*, a unique index for a thread, numbered starting from 0.
– *wait-map*, a hash map, organizing all suspended transactions.
– *lock-queue* buffers all lock entries received from ended transactions.

The *wait-map* is a hash map permitting multiple entries with the same key. Its key field is a locking request, and the value field is a pointer of a transaction. When a transaction t_x has its lock request l_i blocked due to conflict, it adds an entry $<l_i, t_x>$ into the *wait-map*.

The *lock-queue* buffers all lock entries received from finished transactions. When a transaction releases its lock, it checks the *write-waiter* and *read-waiter* fields of the record, and sends the lock to a proper thread by adding a lock entry into the *lock-queue*. Later, a thread uses the received lock entry to wake up a proper transaction in the *wait-map*.

4.1 Lock Acquisition

Here we discuss how to identify conflict locking requests and to resolve conflicts by suspending a transaction. In the following, we consider the scenario where a transaction t_x is trying to acquire the write/read lock of the record r.

```
   Input: Record $r$, Transaction $t_x$
1  nstate ← $t_x$.tid | mask;
2  if 0 ≠ atomic-cas(r.lock-state, 0, nstate) then
3      changed ← atomic-set(r.write-waiter, thd) ;
4      atomic-synchronize();
5      if 0 = atomic-cas(r.lock-state, 0, $t_x$.tid) then
6          if changed then
7              atomic-unset(r.write-waiter, thd) ;
8          return locked ;
9      $t_x$.co-pointer ← co-self() ;
10     wait-map.put(r.rid, write-mode, $t_x$) ;
11     co-yield($t_x$.co-pointer) ;
12     if timed-out($t_x$) then
13         return aborted ;
14     atomic-cas(r.lock-state, mask, nstate) ;
15 return locked ;
```

Fig. 6. acquire-write-lock

```
   Input: Record $r$, Transaction $t_x$
1  do
2      ostate ← r.lock-state;
3      if write-locked(ostate) then
4          changed ← atomic-set(r.read-waiter, thd) ;
5          atomic-synchronize();
6          if write-locked(r.lock-state) then
7              $t_x$.co-pointer ← co-self() ;
8              wait-map.put(r.rid, read-mode, $t_x$) ;
9              co-yield($t_x$.co-pointer) ;
10             if timed-out($t_x$) then
11                 return aborted ;
12         else if changed then
13             atomic-unset(r.read-waiter, thd) ;
14         ostate ← r.lock-state;
15     nstate ← ostate + 1 ;
16 while ostate ≠ atomic-cas(r.lock-state, ostate, nstate);
17 return locked ;
```

Fig. 7. acquire-read-lock

Identifying Conflicts. To lock a record r down, (1) t_x checks whether its lock request is compatible with the current lock state r.*lock-state*. If the result is yes, (2) t_x updates the field with a new state (the field is updated in different ways for the read lock and the write lock as introduced later). The two steps are done atomically using the compare-and-swap instruction (atomic-cas in figures), which is a hardware-assisted synchronization primitive (See Wikipedia).

Figure 6 gives the pseudo-code for the write lock acquisition. Here, Line 1 creates a new lock state, whose first bit is set as 1 and the rest bits are set as *tid* of t_x, representing a write lock held by t_x. Then Line 2 checks whether r.*lock-state* = 0, i.e. r is not locked by anyone, and try to atomically update r.*lock-state* with the new lock state calculated in Line 1. Figure 7 gives the pseudo-code for the read lock acquisition. (1) Line 2–3 fetches a snapshot of r.*lock-state* and checks whether the record is locked in write mode (i.e. the first bit of r.*lock-state* is 1). (2) If the result is yes, a conflict is identified, otherwise, Line 15 creates a new lock state by increasing the number of readers by 1. (3) Line 16 checks that the latest state r.*lock-state* is equal with the snapshot, and update r.*lock-state* with the new value computed in Line 15. (4) Line 16 may find that r.*lock-state* has been changed as other threads can modify the variable in parallel. In this case, the above steps have to be done again.

Resolving Conflicts. If r.*lock-state* is in an incompatible state, t_x is suspended with the following steps. (1) *Turn the wait bit on.* First of all, the *thd*-th bit of r.*write-waiter* (or r.*read-waiter*) is turned on. It is used to inform the lock holder that a transaction in the *thd*-th thread is waiting for the lock. Then a memory fence (i.e. atomic-synchronize in figures) is added to ensure the wait bit is modified before the following steps are executed. (2) *Try locking again.* t_x must check the locking state again in case that the record is just unlocked after the previous attempt. If t_x can secure the lock, it continues its processing; otherwise, t_x will still be suspended. (3) *Suspend the execution.* t_x saves the pointer of its coroutine instance into r.*co-pointer*, and then add an entry into the *wait-map*, which maps the blocked lock request to t_x itself. After that, *co-yield* is invoked to exit its execution temporarily.

```
Input: Record $r$, Transaction $t_x$
1  ostate ← $r.lock\text{-}state$;
2  nstate ← 0 ;
3  if $r.write\text{-}waiter \neq 0 \lor r.read\text{-}waiter \neq 0$ then
4      nstate ← $mask$;
5  atomic-cas($r.lock\text{-}state$, ostate, nstate) ;
6  atomic-synchronize();
7  if $nstate = 0$ then
8      if $r.write\text{-}waiter \neq 0 \lor r.read\text{-}waiter \neq 0$ then
9          if 0 = atomic-cas($r.lock\text{-}state$, 0, $mask$) then
10             nstate ← $mask$;
11             atomic-synchronize();
12 if $nstate = mask$ then
13     if $r.read\text{-}waiter \neq 0$ then
14         send-read-lock(r) ;
15     else if $r.write\text{-}waiter \neq 0$ then
16         send-write-lock(r) ;
```

Fig. 8. release-write-lock

```
Input: Record $r$, Transaction $t_x$
1  do
2      ostate ← $r.lock\text{-}state$;
3      nstate ← $ostate - 1$ ;
4      if nstate = 0 then
5          if $r.write\text{-}waiter \neq 0 \lor r.read\text{-}waiter \neq 0$ then
6              nstate = $mask$;
7  while ostate ≠ atomic-cas($r.lock\text{-}state$, ostate, nstate);
8  atomic-synchronize();
9  if nstate = 0 then
10     if $r.write\text{-}waiter \neq 0 \lor r.read\text{-}waiter \neq 0$ then
11         if 0 = atomic-cas($r.lock\text{-}state$, 0, $mask$) then
12             nstate ← $mask$;
13             atomic-synchronize();
14 if nstate = $mask$ then
15     if $r.write\text{-}waiter \neq 0$ then
16         send-write-lock(r) ;
17     else if $r.read\text{-}waiter \neq 0$ then
18         send-read-lock(r) ;
```

Fig. 9. release-read-lock

In Fig. 6, Line 3–11 are detail steps used to suspend a write lock request. Line 3–4 turn the *thd*-th bit of *write-waiter* on. Line 5–6 try locking again. If Line 3 does flip the wait bit, Line 7 turns that off if the lock is acquired successfully. It is because no transaction in the thread is waiting for the lock once t_x is granted. If the second lock attempt is failed, Line 9–11 suspend the execution of t_x. In Fig. 7, Line 4–14 suspends a read lock request. Line 4–5 turn the *thd*-th bit of *read-waiter* on. Line 6 checks whether the read lock is acquirable again. If not, Line 7–9 suspend the execution of t_x. Otherwise, Line 12–13 clear the wait bit if it is flipped in the previous. And Line 14–17 try locking again. Here, a write lock request may be starved when new readers are always allowed to acquire the read lock. To fix the issue, we can refine the algorithm in Fig. 7 by blocking any new reader if there is a writer waiting for the lock (i.e. $r.write\text{-}waiter \neq 0$).

4.2 Lock Releasing

Releasing a lock on a record is decomposed into two stages. The first stage reverts the *lock-state* field of the record. The second stage informs some particular threads to wake up their blocked transactions if no one is holding the lock any more. In the following, we consider the scenario where a transaction t_x is releasing its lock on a record r.

Reverting Lock State. If multiple transactions are holding read locks on r, t_x simply decreases $r.lock\text{-}state$ by 1. Otherwise, t_x becomes the last lock holder. In this case, we need check whether any transaction is waiting for accessing r. (A.) *Have a waiter.* If there is a blocked request, $r.lock\text{-}state$ is reverted to a special state $mask = 0x80000000$, whose first bit is 1 and the rest are zeros. It can be viewed as a **dummy write lock** held by a dummy transaction with $tid = 0$. Recall that $tid = 0$ is never used by real transactions. It prevents ongoing transactions from *stealing* the lock, and ensure the lock would be granted to a blocked transaction. (B1.) *Have no waiter.* If no one is blocked, the record is

```
  Input: Record r
1 state ← r.write-waiter;
2 for i ← 1 ; i ≤ thd-num; ++i do
3     j ← (i + thd) mod thd-num;
4     if 0 ≠ (state & (1 ≪j)) then
5         lock-queue[j].push(rid, write-mode) ;
6         break ;
```

Fig. 10. send-write-lock

```
  Input: Record r
1 state ← r.read-waiter;
2 reader-number ← popcount(state) ;
3 atomic-cas(r.lock-state, mask, read-number) ;
4 atomic-synchronize();
5 for i ← 0 ; i < thd-num; ++i do
6     if 0 ≠ (state & (1 ≪i)) then
7         lock-queue[i].push(rid, read-mode) ;
```

Fig. 11. send-read-lock

truly unlocked by set r.*lock-state* ← 0. (B2.) *Have a new waiter.* If the record is unlocked, *r.write-waiter* and *r.read-waiter* should be checked again since a transaction t_y may be just suspended after t_x does the first check. If a new waiter is found, we try to acquire the dummy write lock again and resume t_y.

Figures 8 and 9 give procedures used to release write locks and read ones respectively. In Fig. 8, Line 1–6 reverts *r.lock-state* to *mask* or 0 based on whether a waiter exists or not. Line 7–11 check waiter information again, and try to acquire the dummy write lock if a new waiter is found. In Fig. 9, Line 1–8 reverts the lock state based on the waiter information, while Line 9–13 tries to acquire the dummy write lock if a new waiter arrives. In both algorithms, a memory fence is added after modifying the lock state in order to avoid the execution order is changed by the compiler or CPU.

Resuming Transactions. If r is protected by a dummy write lock, we try to wake up blocked transactions (by Line 12–16 in Fig. 8 and by Line 14–18 in Fig. 9).

Figure 10 is used to wake up a suspended write lock request. Here, (1) *Sending lock entry.* t_x sends a write lock entry to a thread who is waiting. Line 2–4 start from the $(thd + 1)$-th bit, and tries to find a non-zero bit in *write-waiter*. Let j-th bit be the non-zero bit. Then, Line 5 pushes a write lock entry into the *lock-queue* of the j-th thread. (2) *Resuming transaction.* When j-th thread pops the lock entry from its *lock-queue*, it gets a transaction t_y from *wait-map*, who is waiting for the lock entry. Then *co-resume* is invoked on t_y.*co-pointer*, and t_y resumes its execution from Line 12 in Fig. 6. This moment, t_y must succeed in acquiring the lock in Line 14. (3) *Clearing wait bit.* the j-th thread calls atomic-unset(*r.write-waiter*, j) to clear its wait bit, if no transaction is waiting for the write lock on the j-thread any more.

Figure 11 is used to wake up read lock requests, which is a little different from the write one. Before sending read lock entries to any thread, Line 2 calculates the number of threads those require the read lock[1]. Line 3–4 acquires **a dummy read lock** for each of them. It is used to ensure that r is always locked in read mode before each thread has waked all its blocked transactions up. The dummy read lock is released when a thread has finished resuming transactions. After that, Line 5–7 distribute read lock entries to *lock-queue*. Then each thread resumes all transactions blocked by the read lock. And a transac-

[1] popcount is an efficient algorithm for calculating the hamming weight of a bit array.

Lock Release(t_x)	Lock Acquire(t_y)
R3: if r.write-wait ≠ 0	A2: if(0 ≠ cas(r.lock-state, 0, ..)
R4: nstate <- mask	A3: set(r.write-waiter, ..)
R5: cas(r.lock-state, .., nstate)	A5: if(0 = cas(r.lock-state, 0, ..)
R8: if r.write-wait ≠ 0	A7: unset(r.writer-watier, ..)
R9: cas(r.lock-state, .., mask)	

Fig. 12. Critical path of acquiring and releasing a lock

Schedule	Result	
(1) R5 < A2	A2 acquire r.lock	correct
(2) A2 < R5 < A5	(i) if A3 < R3, R4-5 acquire the dummy write lock (ii) if R3 < A3 < R8, R8-9 acquire the dummy write lock or A5 acquire r.lock (iii) if R8 < A3, A5 acquire r.lock for Txn B	correct
(3) A5 < R5 (imply A3 < R8)	(i) if A3 < R3, R4-5 acquire the dummy write lock (ii) if R3 < A3, R8-9 acquire the dummy write lock	correct

Fig. 13. All possible schedules

tion resumes its execution from Line 10 in Fig. 7. In the last, each thread calls atomic-unset(r.*read-waiter*, *thd*) to clear its wait bit and release the dummy read lock.

4.3 Deadlock

Deadlock happens when multiple transactions are waiting for the others to release a lock. Here we adopt a time-out strategy to prevent deadlocks from happening. It allows each transaction to wait for a lock within a limited time. When time runs out, a transaction is aborted. It introduces little maintaining overhead and does not generate many unnecessary aborts. The drawback is that a deadlock can not be identified and solved very fast. We consider that a well-designed application would not generate many deadlocks. Hence, the time-out strategy is chosen by iLock to handle deadlocks. In implementation, each thread periodically checks whether any local blocked transaction is timed-out. A transaction is resumed if it is timed-out. In Figs. 6 and 7, a timed-out transaction aborts itself after being resumed. There are also some other solutions, such as using a wait-graph or a wait-die policy [2]. But, maintaining the wait-graph introduces lots of CPU overhead. It is worthwhile when deadlock happen frequently. On the other hand, the wait-die policy would result in many unnecessary aborts.

4.4 Correctness

The security property requires that no conflict lock requests will be granted at the same time. Detailed speaking, two requests are considered to be conflict if they are for the same record and one is a write lock request. In Fig. 6, a write lock is acquirable only when *lock-state* $= 0$ (Line 1) or *lock-state* $=$ *mask* (Line 14). If r.*lock-state* $= 0$, the record is not locked by any others. If r.*lock-state* $=$ *mask*, the record is just unlocked. In both cases, no one is holding the lock of the record. Hence, iLock does not break the security property.

Another correctness concern is that concurrent acquisition and releasing operations execute correctly. Figure 12 illustrates the scenario where a transaction t_x is about to release its write lock on r. Concurrently, transaction t_y tries to acquire the write lock. We must guarantee that the end of t_x will wake up t_y. Figure 13 gives all schedules resulting from the interleaving of the acquisition and releasing operations.

- *Case 1*, t_x releases its lock **(R5)** before t_y does its first locking attempt **(A2)**, then t_y must succeed in lock acquisition.
- *Case 2*, t_x releases its lock **(R5)** between the two locking attempts of t_y **(A2, A5)**. Here, either t_y directly acquire the lock in the second attempt; or t_x finds that t_y is waiting, acquires the dummy write lock and wake up t_y **(R8–9)**.
- *Case 3*, the lock is released **(R5)** after the second locking attempt from t_y **(A5)**, t_x must find that t_y is in-waiting, and then acquire the dummy write lock for t_y.

  In all, t_y is always able to acquire the write lock after t_x releases that. It is similar to verify the correctness of read lock acquisition/releasing.

5 Experiment

We implement an in-memory database prototype with 12,850 lines of C++ codes to verify the efficiency of our method. It is equipped with an in-memory B$^+$-Tree, a query engine and a concurrency control component. All experiments are conducted on a cluster with 5 servers. Each has *two 2.00 GHz 6-Core E5-2620 processors*, *192 GB* DRAM, connected by a *10 GB switch*. Four servers are used to simulate clients, and one is used to deploy the database.

Three methods are compared in the following experiments.

- *iLock* uses the SQL-To-Coroutine model and the new lock manager proposed.
- *Row-Locking* uses the SQL-To-Thread model and the simplified lock manager (row locking). If a lock conflict happens, the transaction gets its SQL request aborted and retried. The row locking mechanism is proposed and used by VLL [11] and pessimistic transactions in Hekaton [1].
- *Lock-Table* uses the Transaction-To-Thread model and a centralized lock table [5]. Such design is adopted by many disk-based database systems.

Each method uses a time-out mechanism to avoid deadlocks. The TPC-C and a micro benchmark are used to evaluate the performance of different methods. For the TPC-C, we ran a standard transaction mix. On the other hand, the micro benchmark has a table with two columns: key, 64-bit integer and value, 64-bit integer. A transaction would read N records and write M records ($N + M$ SQLs in total). Records are selected with a Zipfian distribution. It is used to generate workloads with varied read/write mixes and access distributions.

5.1 Varying Number of Clients

Figure 14 uses TPC-C to evaluate performances by varying the number of clients. 200 warehouses are populated in the database. Overall, *iLock* has the best performance. Its performance increases with more clients are used, and reaches the

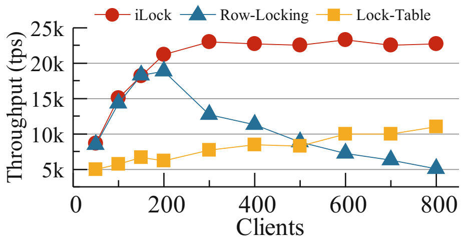


Fig. 14. TPC-C: varying clients

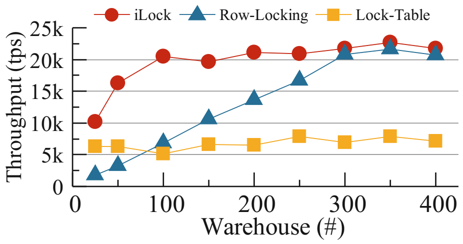


Fig. 15. TPC-C: varying warehouses

peak throughput (about 23k tps) when 300 clients are used. The performance stabilizes because the server has reached the maximal CPU usage. The performance of *Row-Locking* increases when a few clients are used. Its throughput is almost the same as that of *iLock*, because both methods have little CPU overhead if conflicts happen rarely. With the number of clients increased, the performance of *Row-Locking* sees a rapid decrease. It is because access conflicts happen more frequently now, and *Row-Locking* would waste a plenty of CPU time on retrying many failed SQL requests. In contrast, *iLock* is much more efficient. It would suspend a transaction and resume that until the lock is available. In the last, The performance of *Lock-Table* increases slowly. Its peak throughput is about 10k tps. The poor performance is mainly because much CPU time is consumed by context switching and accessing the heavy-weighted lock table. As a result, *Lock-Table* exhausts the CPU easily.

5.2 Varying Number of Warehouses

Next we increase the number of warehouses populated to decrease the workload contention. Figure 15 shows the results. 300 clients are running in default. Overall, *iLock* has the best performance. Its performance increases at the beginning, because when more warehouses are populated, fewer transactions would be blocked by lock conflicts. After the number of warehouses reaches 100, the performance of *iLock* converges as the CPU is fully utilized. The throughput of *Row-Locking* increases almost linearly against the number of warehouses. Under a small number of warehouses, *Row-Locking* is worse than *iLock* because the workload contains many conflicts and the former is less efficient than the latter in handling them. Using more warehouses reduces the probability of lock conflicts, and also reduces the total number of SQL requests retried by *Row-Locking*. As a result, CPU time is saved for processing real work. When 300 warehouses are used, *Row-Locking* reaches its peak throughput, which is similar to that of *iLock*. In the last, *Lock-Table* has a stable but low performance under each case. Decreasing the workload contention has little effect because the performance is always dominated by CPU usage.

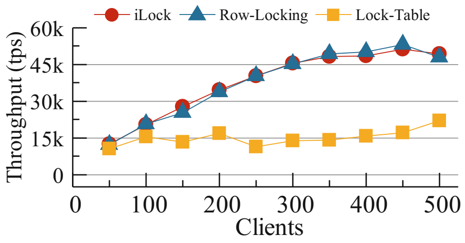


Fig. 16. Micro: 10 reads

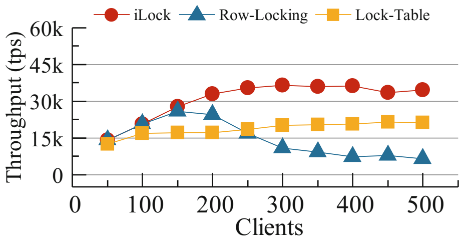


Fig. 17. Micro: 5 reads + 5 writes

5.3 Varying Workload Characteristics

Figures 16 and 17 give micro benchmark performances using different mixes of read/write requests. The Zipfian distribution uses $\theta = 0.6$ to generate requests. Figure 16 uses a read-only workload and each transaction issues 10 read requests. Figure 17 uses a write-intensive workload and each transaction issues 5 read requests and 5 write ones. (1) Under the read-only workload, *iLock* and *Row-Locking* almost have the same performance. It is because there is no conflict in the workload. And both are very efficient in scheduling a conflict-free workload. Their peak throughputs reach 45k tps when the CPU is fully utilized. On the other hand, the throughput of *Lock-Table* is always about 15k tps under each case, which is only one-third the performance of *iLock*. Its poor performance results from spending much valuable CPU time on context switches and maintaining the lock table. (2) Under the write-intensive workload, when 500 clients are used, the performance of *iLock* is about 5.3x that of *Row-Locking*, and 1.6x that of *Lock-Table*. By increasing the number of clients, its performance gets stabilized because more transactions are blocked by lock conflicts. The performance of *Row-Locking* increases at the beginning and get decreased as adding clients results in more lock conflicts. *Lock-Table* shows the similar performance in both the read-only and the write-intensive workloads. The performance gap between *iLock* and *Lock-Table* is smaller than that in Fig. 16. It is because the transaction conflict is the dominating factor for the performance now.

Figure 18 gives throughputs by varying the skewness θ of the Zipfian distribution. Here, we simulate the same write-intensive workload used by Fig. 17. And 200 clients are running concurrently. Figure 19 displays the normalized performances of different methods against that of *iLock*. As we can see, throughputs of all methods are decreasing when the access distribution becomes more skewed. It is because some records become "hot" and being frequently accessed. Transactions are more likely blocked when accessing hot records. When $\theta < 0.6$, there is little contention in the workload, and performances of different methods are limited by the CPU usage. When $\theta > 0.6$, workload contention becomes non-negligible. Here, the throughput of *Row-Locking* drops faster than that of *iLock*. It is because *iLock* is more efficient in scheduling lock conflicts while *Row-Locking* wastes lots of CPU time on retrying failed lock requests. When

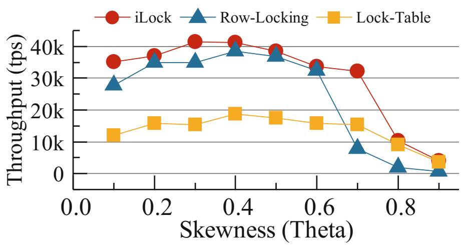


Fig. 18. Micro: skewness (raw)

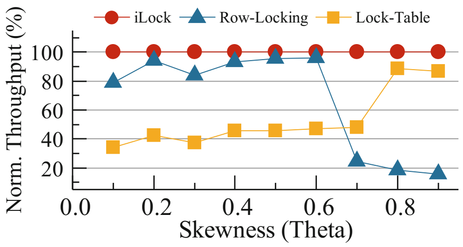


Fig. 19. Micro: skewness (normalized)

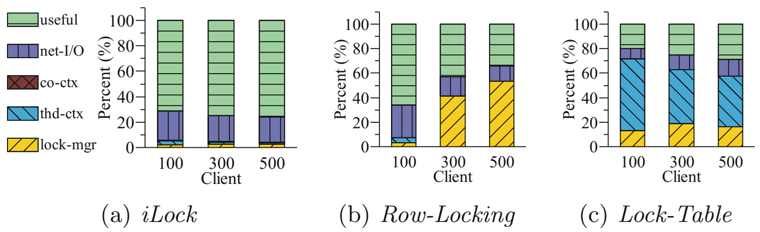


Fig. 20. CPU time breakdown

$\theta \geq 0.8$, *iLock* and *Lock-Table* show very similar performance. At the moment, throughputs are largely limited by the workload contention. Figure 19 shows the performance gaps among different methods clearly. Given a workload containing low contention, a lightweight lock manager gains the better performance. Given a workload containing high contention, the lock manager, which is more efficient in resolving conflicts, works better. Overall, *iLock* shows nice performance under different kinds of workloads.

5.4 Breaking CPU Time down

In Fig. 20, we use VTune, a CPU profiling tool, to track where a working thread has its time gone. Here, we run the TPC-C workload with 200 warehouses. The number of clients is varied to change transaction conflicts in the workload. In Fig. 20(a), *iLock* spends little time on lock management (lock-mgr), context switching (thd-ctx) or coroutine switching (co-ctx). It means that *iLock* handles both network-related and conflict-related blocking efficiently. The network I/O (net-I/O) uses a considerable percent of CPU time. It mainly includes serializing results into a packet and informing I/O threads to respond clients. Most time (about 70%) are spent on real work (e.g. accessing the index, writing records). Figure 20(b) shows the CPU time used by *Row-Locking*. When 100 clients are running concurrently, the workload contains little contention, and *Row-Locking* spends most time on useful work. By increasing the number of clients, there are more conflicts in the workload. *Row-Locking* wastes a large percent of time (41.3%–53.2%) on locking (e.g. retrying failed lock requests, checking the lock condition). Hence, it works poorly when the workload exhibits high contention. Figure 20(c) breaks down the CPU time used by *Lock-Table*. Under different configurations, the most time-consuming part is always context switching (41%–58%) because each request could result in one context switch. On the other hand, lock management also uses considerable time (13%–16%). Its poor performance is due to both context switches and accessing the lock table.

6 Relate Work

Interactive transaction processing is rarely studied under in-memory database system. Existing disk-based systems usually process transactions in interactive way, while in-memory systems are mostly optimized for one-shot transactions. As our best knowledge, the work is a pioneering study on the problem.

As discussed in the above, disk-based database systems usually adopts the Transaction-To-Thread model. But the method could generate lots of CPU overhead. Pandis et al. [9] proposes the data-oriented execution model, where each thread services a disjoint part of database partition. It helps reduce contention because less resources are shared among threads. Transactions should be decomposed into small actions in advance based on the partition, so that each action runs on the proper thread. However, it does not work for interactive transaction since the pre-process is unfeasible.

Disk-based database systems schedule transactions with a centralized lock manager. It is proven to generate much contention on multi-core hardwares. Jung et al. [5] proposes a multi-core scalable lock manager by reducing latch usages. However, acquiring/releasing locks still consumes considerable CPU instructions. Johnson et al. [4] designs a lock inheritance mechanism to reduce the number of lock table access. A transaction can inherit locks directly from an ending one. The mechanism does not work if two transactions have no locks in common. The iLock we proposed here is a latch-free and lightweight structure. It has multi-core scalability and does not generate too much maintaining overhead.

In-memory databases are mostly designed for one-shot transactions. Some new execution models are proposed. Stonebraker [12] proposes a serial execution model, which queues and executes transactions one-by-one. Similarly, Thomson [13,14] proposes the deterministic transaction execution. Concurrency control is done before the real transaction execution. They take full advantage of CPU resources by eliminating all blocking during transaction processing. But, they are limited to handle only one-shot requests.

Many concurrency control schemas are designed for one-shot transactions processing. Ren et al. [11] and Larson et al. [8] design lightweight lock managers, which put the lock information in the header of a record. Tu et al. [15] improves the OCC by eliminating its centralized contention point. Wu et al. [18] adds a healing phase for OCC, which helps reduce operations retried when a transaction has its validation failed. Wang et al. [17] chops a transaction into multiple pieces in an offline phase, and isolates transactions in the degree of pieces. Efficiencies of these methods rely on the one-shot property. Some should analyze transaction logics in advance. And others would like transactions to be short and do not conflict with each other frequently. As a result, they are not suitable for scheduling interactive transactions.

7 Conclusion

This work studies the interactive transaction processing for the in-memory database system. We propose a new execution model and a new lock manager to efficiently process and schedule interactive transactions. Experiments on well-known benchmarks show our method achieves good performance in processing interactive workloads. A future direction is to design a transaction engine for processing the hybrid workload, containing both one-shot requests and interactive ones.

Acknowledgement. This is work is partially supported by National High-tech R&D Program(863 Program) under grant number 2015AA015307, National Science Foundation of China under grant numbers 61702189, 61432006 and 61672232, and Youth Science and Technology - “Yang Fan” Program of Shanghai under grant number 17YF1427800.

References

1. Diaconu, C., Freedman, C., Ismert, E., Larson, P.A., et al.: Hekaton: SQL server’s memory-optimized OLTP engine. In: SIGMOD, pp. 1243–1254 (2013)
2. Gray, J.: Transaction Processing: Concepts and Techniques. Elsevier, Amsterdam (1992)
3. 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)
4. Johnson, R., Pandis, I., Ailamaki, A.: Improving OLTP scalability using speculative lock inheritance. VLDB **2**(1), 479–489 (2009)
5. Jung, H., Han, H., Fekete, A., Heiser, G., Yeom, H.Y.: A scalable lock manager for multicores. TODS **39**(4), 29 (2014)
6. Knuth, D.E.: The Art of Computer Programming: Fundamental Algorithms, vol. 1. Addison Wesley Longman Publishing Co., Inc., Boston (1997)
7. Kung, H.T., Robinson, J.T.: On optimistic methods for concurrency control. TODS **6**(2), 213–226 (1981)
8. Larson, P.Å., Blanas, S., Diaconu, C., Freedman, C., Patel, J.M., Zwilling, M.: High-performance concurrency control mechanisms for main-memory databases. VLDB **5**(4), 298–309 (2011)
9. Pandis, I., Johnson, R., Hardavellas, N., Ailamaki, A.: Data-oriented transaction execution. VLDB **3**(1–2), 928–939 (2010)
10. Pavlo, A.: What are we doing with our lives?: nobody cares about our concurrency control research. In: SIGMOD, p. 3. ACM (2017)
11. Ren, K., Thomson, A., Abadi, D.J.: Lightweight locking for main memory database systems. VLDB **6**, 145–156 (2012)
12. Stonebraker, M., Madden, S., Abadi, D.J., et al.: The end of an architectural era: (it’s time for a complete rewrite). In: VLDB, pp. 1150–1160 (2007)

13. Thomson, A., Abadi, D.J.: The case for determinism in database systems. VLDB **3**(1–2), 70–80 (2010)
14. Thomson, A., Diamond, T., Weng, S.C., Ren, K., et al.: Calvin: fast distributed transactions for partitioned database systems. In: SIGMOD, pp. 1–12 (2012)
15. Tu, S., Zheng, W., Kohler, E., Liskov, B., Madden, S.: Speedy transactions in multicore in-memory databases. In: SOSP, pp. 18–32 (2013)
16. Wang, T., Kimura, H.: Mostly-optimistic concurrency control for highly contended dynamic workloads on a thousand cores. VLDB **10**(2), 49–60 (2016)
17. Wang, Z., Mu, S., Cui, Y., Yi, H., Chen, H., Li, J.: Scaling multicore databases via constrained parallel execution. In: SIGMOD, pp. 1643–1658. ACM (2016)
18. Wu, Y., Chan, C.Y., Tan, K.L.: Transaction healing: scaling optimistic concurrency control on multicores. In: SIGMOD, pp. 1689–1704. ACM (2016)