VLL: a lock manager redesign for main memory database systems

February 6, 2025

1 Very lightweight locking

The VLL protocol is designed to be as general as possible, with specific optimizations for the following architectures:

- Multiple threads execute transactions on a single-server, shared memory system.
- Data are partitioned across processors (possibly spanning multiple independent servers). At each partition, a single thread executes transactions serially.
- Data are partitioned arbitrarily (e.g., across multiple machines in a cluster); within each partition, multiple worker threads operate on data.

1.1 The VLL algorithm

VLL stores each record's "lock table entry" not as linked list in a separate lock table, but rather as a pair of integer values (C_X, C_S) immediately preceding the record's value in storage, which represents the **number** of transactions requesting exclusive and shared locks on the record, respectively.

In addition, a global queue of transaction requests, called `TxnQueue`, is kept at each partition, tracking all active transactions in the order in which they requested their locks.

When a transaction arrives at a partition, it attempts to request locks on all records at that partition that it will access in its lifetime. Each lock request takes the form of incrementing the corresponding record's C_X or C_S value,

depending whether an exclusive or shared lock is needed. Exclusive locks are considered to be "granted" to the requesting transaction if $C_X = 1$ and $C_S = 0$ after the request, since this means that no other shared or exclusive locks are currently held on the record. Similarly, a transaction is considered to have acquired a shared lock if $C_X = 0$, since that means that no exclusive locks are held on the record.

Once a transaction has requested its locks, it is added to the `TxnQueue`. Both the requesting of the locks and the adding of the transaction to the queue happen inside the same critical section (so that only one transaction at a time within a partition can go through this step). In order to reduce the size of the critical section, the transaction attempts to figure out its entire read set and write set in advance of entering this critical section. This process is not always trivial and may require some exploratory actions. Furthermore, multi-partition transaction lock requests have to be coordinated.

Upon leaving the critical section, VLL decides how to proceed based on two factors:

- Whether or not the transaction is **local** or **distributed**. A local transaction is one whose read and write sets include records that all reside on the same partition; distributed transactions may access a set of records spanning multiple data partitions.

- Whether or not the transaction successfully acquired all of its locks immediately upon requesting them. that acquire all locks immediately are termed free. Those which fail to acquire at least one lock are termed blocked.

VLL handles each transactions differently based on whether they are free or blocked:

- **Free transactions** are immediately executed. Once completed, the transaction releases its locks (i.e., it decrements every C_X or C_S value that it originally incremented) and removes itself from the `TxnQueue`. Note, however, that if the free transaction is distributed, then it may have to wait for remote read results, and therefore may not complete immediately.

- **Blocked transactions** cannot execute fully, since not all locks have been acquired. Instead, these are tagged in the `TxnQueue` as blocked. Blocked transactions are not allowed to begin executing until they are explicitly unblocked by the VLL algorithm.

Since there is no lock management data structure to record which transactions are waiting for data locked by other transactions, there is no way for a transaction to hand over its locks directly to another transaction when it finishes. An alternative mechanism is therefore needed to determine when blocked transactions can be unblocked and executed.

Fortunately, this situation can be resolved by a simple observation: a blocked transaction that reaches the front of the `TxnQueue` will always be able to be unblocked and executed—no matter how large C_X and C_S are for the data items it accesses.

Note that a blocked transaction now has two ways to become unblocked: either it makes it to the front of the queue (meaning that all transactions that requested locks before it have finished completely), or it becomes the only transaction remaining in the queue that requested locks on each of the keys in its read set and write set.

One problem that VLL sometimes faces is that as the `TxnQueue` grows in size, the probability of a new transaction being able to immediately acquire all its locks decreases, since the transaction can only acquire its locks if it does not conflict with any transaction in the entire `TxnQueue`.

We therefore artificially limit the number of transactions that may enter the `TxnQueue`.

```
// Requests exclusive locks on all records in T's
// WriteSet and shared locks on all records in T's ReadSet.
// Tags T as free iff ALL locks requested were
// successfully acquired.
function BeginTransaction(Txn T)
    <begin critical section>
    T.Type = Free;
    // Request read locks for T.
    foreach key in T.ReadSet
        data[key].Cs++;
        // Note whether lock was acquired.
        if (data[key].Cx > 0)
            T.Type = Blocked;
    // Request write locks for T.
    foreach key in T.WriteSet
        data[key].Cx++;
        // Note whether lock was acquired.
        if (data[key].Cx > 1 OR data[key].Cs > 0)
            T.Type = Blocked;
    TxnQueue.Enqueue(T);
    <end critical section>

// Releases T's locks and removes T from TxnQueue.
function FinishTransaction(Txn T)
    <begin critical section>
    foreach key in T.ReadSet
        data[key].Cs−−;
    foreach key in T.WriteSet
        data[key].Cx−−;
    TxnQueue.Remove(T);
    <end critical section>

// Transaction execution thread main loop.
function VLLMainLoop()
    while (true)
        // Select a transaction to run next...
        // First choice: a previously-blocked txn
        // that now does not conflict with older txns.
        if (TxnQueue.front().Type == Blocked)
            Txn T = TxnQueue.front();
            T.Type = Free;
            Execute(T);
            FinishTransaction(T);
        // 2nd choice: Start on a new txn request.
        else if (TxnQueue is not full)
            Txn T = GetNewTxnRequest();
            BeginTransaction(T);
            if (T.Type == Free)
                Execute(T);
                FinishTransaction(T);
```

Figure 1: Pseudocode for the VLL algorithm

1.2 Arrayed VLL

Store all C_X in one vector and all C_S in another vector.

1.3 Single-threaded VLL

Useful in H-Stroe style settings, where data are partitioned across cores within a machine, and there is only oen thread assigned to each partition.

If single-threaded VLL was implemented simply by running only one thread (on each partition) according to the previous specification, the result would be a serial execution of transactions (within each partition). This results in wasted resources, since when the thread needs to sleep, waiting for a message from a distributed node, and no other progress can be made within that partition.

In order to improve concurrency in single-threaded VLL implementations, we allow transactions to enter a third state (in addition to "blocked" and "free"). This third state, "waiting", indicates that a transaction was previously executing but could not complete without the result of an outstanding remote read request.

```
// Requests exclusive locks on all records in T's WriteSet
// and shared locks for T's ReadSet, and sets T's status.
function BeginTransaction(Txn T)
    T.Type = Free;
    foreach key in T.ReadSet // Request T's read locks
        if key is in local storage
            data[key].Cs++;
            if (data[key].Cx > 0)
                T.Type = Blocked;
    foreach key in T.WriteSet // Request T's write locks
        if key is in local storage
            data[key].Cx++;
            if (data[key].Cx > 1 OR data[key].Cs > 0)
                T.Type = Blocked;
    if T is a distributed transaction And T.Type == Free
        T.Type = Waiting

// Releases T's locks and removes T from TxnQueue.
function FinishTransaction(Txn T)
    foreach key in T.ReadSet
        if key is in local storage
            data[key].Cs−−;
    foreach key in T.WriteSet
        if key is in local storage
            data[key].Cx−−;

// Transaction execution thread main loop.
function VLLMainLoop()
    while (true)
        // Select a transaction to run next...
        // First choice: Resume a "waiting" transaction
        if (Received remote message for transaction T)
            HandleReadResult(T, message);
            if ReadyToExecute(T);
                Execute(T);
                FinishTransaction(T);
                TxnQueue.Remove(T);
        // Second choice: a previously-blocked txn
        // that now does not conflict with older txns.
        else if (TxnQueue.front().Type == Blocked)
            Txn T = TxnQueue.front();
            if T is a distributed transaction
                T.Type = Waiting;
                Send local reads to other participating nodes
            else
                T.Type = Free;
                Execute(T);
                FinishTransaction(T);
                TxnQueue.Remove(T);
        // Third choice: Start on a new txn request.
        else if (TxnQueue is not full)
            Txn T = GetNewTxnRequest();
            BeginTransaction(T);
            if (T.Type == Free)
                Execute(T);
                FinishTransaction(T);
            else if (T.Type == Waiting)
                Send local reads to other participating nodes
                TxnQueue.Enqueue(T);
            else if (T.Type == Blocked)
                TxnQueue.Enqueue(T);
```

Figure 2: Psudocode for the single-threaded VLL algorithm

1.4 Impediments to acquiring all locks at once

In order to guarantee that the head of the `TxnQueue` is always eligible to run (which has the added benefit of eliminating deadlocks), VLL requires that all locks for a transaction be acquired together in a critical section. There are two possibilities that make this nontrivial:

- The read and write sets of a transaction may not be known before running the transaction. An example of this is a transaction that updates a tuple that is accessed through a secondary index lookup. Without first doing the lookup, it is hard to predict what records the transaction will access—and therefore what records it must lock.

- Since each partition has its own `TxnQueue` and the critical section in which it is modified is local to a partition, different partitions may not begin processing transactions in the same order. This could lead to distributed deadlock, where one partition gets all its locks and activates a transaction, while that transaction is "blocked" in the `TxnQueue` of another partition. This is hard!

In order to overcome the first problem, before the transaction enters the critical section, we allow the transaction to perform whatever reads it needs to (at no isolation) for it to figure out what data it will access (for example, it performs the secondary index lookups). This can be done in the `GetNewTxnRequest` function. After performing these exploratory reads, it enters the critical section and requests those locks that it discovered it would likely need. Once the transaction gets its locks and is handed off to an execution thread, the transaction runs as normal unless it discovers that it does not have a lock for something it needs to access

There are two possible solutions to the second problem. The first is simply to allow distributed deadlocks to occur and to run a deadlock detection protocol that aborts deadlocked transactions. The second approach is to coordinate across partitions to ensure that multi-partition transactions are added to the TxnQueue in the same order on each partition.

1.5 Trade-offs of VLL

The main disadvantage of VLL is the loss in concurrency. Traditional lock managers use the information contained in lock request queues to figure

out whether a lock can be granted to a particular transaction. Since VLL does not have these lock queues, it can only test more selective predicates on the state: (a) whether this is the only lock in the queue, or (b) whether it is so old that it is impossible for any other transaction to precede it in any lock queue.

As a result, it is common for scenarios to arise under VLL where a transaction cannot run even though it "should" be able to run (and would be able to run under a standard lock manager design). Consider, for example, the sequence of transactions:

txn	Write set
A	x
B	y
C	x,z
D	z

Suppose A and B are both running in executor threads (and therefore still in the `TxnQueue`) when C and D come along. Since transaction C conflicts with A on record x and D conflicts with C on z, both are put on the `TxnQueue` in blocked mode.

VLL			Standard	
Key	C_X	C_s	Key	Request queue
x	2	0	x	A,C
y	1	0	y	B
z	2	0	z	C,D

Next, suppose that A completes and releases its locks. The lock tables would then appear as follows:

VLL			Standard	
Key	C_X	C_S	Key	Request queue
x	1	0	x	C
y	1	0	y	B
z	2	0	z	C,D

Since C appears at the head of all its request queues, a standard implementation would know that C could safely be run, whereas VLL is not able to determine that.

When contention is low, this inability of VLL to immediately determine possible transactions that could potentially be unblocked is not costly. However, under higher contention workloads, and especially when there are distributed transactions in the workload, VLL's resource utilization suffers, and additional optimizations are necessary.

1.6 Selective contention analysis (SCA)

For high-contention and high-percentage multi-partition workloads, VLL spends a growing percentage of CPU cycles in the state where no transaction can be found that is known to be safe to execute—whereas a standard lock manager would have been able to find one. In order to maximize CPU resource utilization, we introduce the idea of SCA.

SCA simulates the standard lock manager's ability to detect which transactions should inherit released locks. It does this by spending work examining contention—but only when CPUs would otherwise be sitting idle (i.e., TxnQueue is full and there are no obviously unblockable transactions). SCA therefore enables VLL to selectively increase its lock management overhead when (and only when) it is beneficial to do so.

Any transaction in the `TxnQueue` that is in the 'blocked' state, conflicted with one of the transactions that preceded it in the queue at the time that it was added. Since then, however, the transaction(s) that caused it to become blocked may have completed and released their locks. As the transaction gets closer and closer to the head of the queue, it therefore becomes much less likely to be "actually" blocked.

In general, the ith transaction in the TxnQueue can only conflict now with up to $i - 1$ prior transactions, whereas it previously had to contend with (up to) the number of `TxnQueueSizeLimit` prior transactions. Therefore, SCA starts at the front of the queue and works its way through the queue looking for a transaction to execute. The whole while, it keeps two-bit arrays, D_X and D_S , each of size 100 kB (so that both will easily fit inside an L2 cache of size 256 kB) and initialized to all 0s. SCA then maintains the invariant that after scanning the first i transactions:

- $D_X[j] = 1$ iff an element of one of the scanned transactions' write sets hashes to j
- $D_S[k] = 1$ iff an element of one of the scanned transactions' read sets hashes to k

Therefore, if at any point the next transaction scanned (let's call it T_{next}) has the properties:

- $D_X[hash(key)] = 0$ for all keys in T_{next}'s read set
- $D_X[hash(key)]$ for all keys in T_{next}'s write set
- $D_S[hash(key)] = 0$ for all keys in T_{next}'s write set

then T_{next} does not conflict with any of the prior scanned transactions and can safely be run.

2 Problems

3 References

References