A Semi-Optimistic Database Scheduler Based on Commit Ordering

William Perrizo and *Igor Tatarinov*

Computer Science Dept,

North Dakota State University

Fargo, ND 58105-5164

Contact author: *Igor Tatarinov*

Phone: (701) 231-9462

Fax: (701) 231-8255

E-mail: itat@acm.org

A Semi-Optimistic Database Scheduler Based on Commit Ordering

Abstract

This paper proposes a semi-optimistic database scheduler that is based on relaxed two-phase locking and commitment ordering. Unlike the traditional strict 2PL scheduler, the proposed scheduler does not block a write request when read locks are set on the requested data item.

We present two implementations of the scheduler: one that does not uses deferred updates and another that does. We demonstrate that in both cases, the proposed scheduler can be easily implemented using the data structures maintained by a typical 2PL scheduler.

1. Introduction

A database scheduler is a part of a database system that is responsible for concurrency control functions. The goal of concurrency control in database systems is to ensure correctness of transaction executions (histories). A trivial way to ensure such correctness is to allow only serial executions but this is too restrictive in most cases. A more efficient algorithm would allow all *serializable* executions. Unfortunately, recoverability and other issues impose additional constraints on acceptable executions. As a result, most existing schedulers allow only *strict* serializable executions.

The most common example of a strict database scheduler is the scheduler based on strict two-phase locking (S2PL). Unfortunately, the S2PL scheduler is more restrictive than a strict scheduler needs to be. For example, the following history is strict and serializable: $r_1[x]\ w_2[x]\ c_1\ c_2$ but the traditional S2PL scheduler would not accept it.

1.1 Research Contributions

The main objective of this study is to modify the S2PL scheduler to make it less restrictive. Our proposed scheduler does not block a write on a data item when another transaction holds a read lock on this data item.

The scheduler, that we have called *semi-optimistic* (SO), uses commitment ordering of transactions [Roaz 92] (similar to ordered shared locking [Agrawal, et al. 1991]) to ensure that the relaxed S2PL scheduler produces only serializable executions. The name *semi-optimistic* reflects the fact that the scheduler is more optimistic than many traditional schedulers: serializability of executions is ensured at commit time. However it is not completely optimistic; some operations are still blocked. Blocking helps the scheduler

abort fewer transactions than pure *optimistic* schedulers would abort. The proposed scheduler strikes a balance between an optimistic (aggressive) and a conservative scheduler.

Unlike many other proposed schedulers, our scheduler does not require any data structures other than those used by a typical S2PL scheduler (excellently described in [Gray, Reuter 93]).

We describe two versions of the SO scheduler: one that does not use deferred updates and another that does. We show that without deferred updates, a handshake between the scheduler and the database manager has to be implemented. This, however, requires only a slight modification to the scheduler data structures. Deferred updates make the scheduler even less restrictive and eliminate the handshake. This makes the second version of the scheduler more attractive when deferred updates are easy to implement.

1.2 Previous Work

A tremendous amount of research has been performed in the area of concurrency control. In this section, we review only some of the work related to two-phase locking and its modifications.

[Agrawal, et al. 91] propose and evaluate ordered shared locking, a concurrency control policy, that is very similar to that presented in this paper. However, the authors did not discuss implementation details of the proposed algorithm. Nor did they elaborate on the scheduler-data manager handshake issues.

[Mizuki, et al 94] propose an efficient implementation of a 2PL scheduler that allows concurrent writes. The second version of our scheduler incorporates the ideas presented in that paper.

Several studies have proposed modifications to S2PL based on the assumptions that transactions know their future actions. For example, altruistic locking [Salem, et al. 94] is less restrictive than the scheduler proposed in this paper provided transactions know if they are going to use a given data item in the future. Not only does this make altruistic locking an unrealistic policy, it also greatly complicates the implementation of the algorithm.

In this paper, we propose a scheduler that is based on the data structures of a typical S2PL scheduler but is less restrictive.

2. Properties of Transaction Executions

2.1 Serializability

Transaction T_2 is in a conflict with transaction T_1 in history[1] H if $p_1[x]$ <(precedes) $q_2[x]$ and p_1 and q_2 are conflicting operations, i.e., at least one of them is a write. (Note that this definition is asymmetric.) Three types of conflicts are possible: write-read (*wr*), write-write (*ww*), and read-write (*rw*). If T_1 is the last unaborted transaction to write a data item before T_2 reads it (*wr* conflict) we will say that T_2 *reads-from* (*rf*) T_1.

A history is *serializable* if its committed projection is equivalent to some serial history [Bernstein, et al. 87]. *Serializability* (SR) is the most important property of histories that needs to be ensured. A non-SR history may be incorrect: some updates may be lost, etc.

Serializability requires that it is possible to compare two histories for equivalence. There are two common definitions of history equivalence. Two histories are called *conflict equivalent* if they are histories over the same set of transactions and the order of all conflicting operations is identical. For example, if T_1 and T_2 are in a *rw (ww,wr)* conflict in history H, they are in the same conflict in any history conflict equivalent to *H*. If the committed projection of a history *H*, C(*H*), is conflict equivalent to some serial history, we say that *H* is conflict serializable (CSR).

Two histories are *view equivalent* if they are over the same set of transactions and all read-from relationships in the histories are identical[2]. If for every prefix *H' of* a history *H*, the committed projection of *H'*, C(*H'*), is view equivalent to some serial history, we say that *H* is view serializable (VSR).

VSR is a less strict requirement than CSR. For example, $w_1[x]\ w_2[y]\ r_2[x]\ w_1[y]\ c_1\ c_2$ is not CSR but it is VSR (the resulting serial history is $w_1[x]\ w_1[y]\ c_1\ w_2[y]\ r_2[x]\ c_1$). However, VSR is too computationally expensive. The problem of checking if a given history is VSR is NP complete [Bernstein, et al. 87]. Because of that, all existing concurrency control algorithms use CSR or a subset of VSR histories to ensure correctness of executions.

The *Serialization Graph (SG)* of history *H* is a directed graph where nodes are committed transactions and edges represent transaction conflicts.

---

[1] For the formal definitions of a transaction and a transaction history, see, for example, [Bernstein, et al. 87].

[2] It is assumed that each history has two additional fictitious transactions: *Tw* and *Tr*. *Tw* writes all data items in the beginning of the history and *Tr* reads all data items in the end of the history. Conflict equivalence does not require existence of such transactions.

The Serializability Theorem: *A history H is CSR iff SG(H) is acyclic.*

To recognize a VSR history a more complex graph is needed. Instead, we will use the definition of VSR to prove that a given history is VSR.

2.2 Recoverability, Cascadelesness, Strictness, and Rigorousness

Recoverability (RC) is an essential property of histories if transactions can abort, i.e., in all real situations. RC requires that $\forall\ T_i,T_j\ c_i \in H \wedge\ T_i\ rf\ T_j \Rightarrow c_j<c_i$ *(RC)*

Note that it is implicitly required that T_j commits. If we used, $\forall\ T_i,T_j\ c_i \in H \wedge c_j \in H \wedge T_i\ rf\ T_j \Rightarrow c_j<c_i$ instead, the following history would be RC (but it is not): $w_1[x]\ r_2[x]\ c_2\ a_1$.

Recoverability also ensures that an SR history has the same semantics as its equivalent serial history. A non-RC serializable schedule may produce different results depending on the presence of aborted transactions. For example, $w_1[x]\ w_2[x]\ r_3[x]\ c_1\ a_2\ c_3$ is CSR but not RC. The result of this history, however, depends on the value written by T_2, an aborted transaction. Therefore, we consider recoverability a necessary property.

When a transaction aborts, RC requires that all transactions that has read from it, also be aborted. This phenomenon is called *cascading aborts*. For example, for the following history to be RC, T_2 also has to be aborted: $w_1[x]\ r_2[x]\ a_1$.

Cascadelessness of histories ensures that an abort never causes cascading aborts. A history *H* is said to avoid cascading aborts (ACA) if a transaction can only read data from an already committed transaction: $\forall\ T_i,T_j\ T_i\ rf\ T_j \Rightarrow c_j<r_i$*(ACA)*

Strictness (ST) implies that: $\forall\ T_i,T_j\ w_j[x]<o_i$*(conflicting)*$\Rightarrow$ *either* $a_j<o_i$ *or* $c_j<o_i$ *(ST)* where *o* is either *r* or *w*.

ST ensures that if a transaction is aborted, all data that it has modified can be restored using the "before-images". It may not be possible to correctly restore the state of the database if a transaction participating in a non-strict execution aborts. For example, the following (non-strict) history $w_1[x]\ w_2[x]\ a_1$ would cause T_1 to restore the initial value of *x* effectively replacing the value written by T_2. A strict scheduler would not admit such a history.

Rigorousness (RG) requires that all possible conflicts be allowed only after the first transaction commits: $\forall\ T_i,T_j\ q_j[x]<p_i[x] \Rightarrow$ *either* $a_j<p_i$ *or* $c_j<p_i$ *(RG)* where *p* and *q* are conflicting operations.

The traditional Strict Two-Phase Locking (S2PL) actually ensures RG not ST, which is not always needed. It turns out that it is not easy to create a correct ST scheduler, see Section 3 for a discussion.

RG is an important issue in heterogeneous distributed environments [Roaz 92]. Such environments are beyond the scope of this paper.

Theorem: *RG* $\subset$ *ST* $\subset$ *ACA* $\subset$ *RC (the containment is strict)*

2.3 Commitment Ordering

Commitment ordering (CO) was introduced in [Roaz 92]. A history is in CO if for any *committed* conflicting transactions the order of conflicting operations agrees with the order of the commit operations:

$$\forall\ T_i,T_j\ c_i,c_j \in H \wedge q_j[x] < p_i[x]\ (conflicting) \Rightarrow c_j < c_i\ (CO)$$

Note that CO does not imply RC. To ensure RC, we need to augment CO with a requirement that if a transaction reads-from another transaction, the second transaction must commit and do so before the first transaction does: $CO + \forall\ T_i,T_j\ c_i \in H \wedge T_i\ rf\ T_j \Rightarrow c_i > c_j$ *(RC CO).*

Along with RC CO, one may define ACA CO and ST CO. RG CO is obviously equivalent to RG.

CO is an interesting property because of the following theorem proved elsewhere [Roaz 92]:

CO Theorem: *CO* $\Rightarrow$ *CSR, that is every CO history is SR.*

2.4 Commit Ordering vs. Ordered Shared Locking

Ordered shared locking (OSL) [Agrawal, et al. 91] is almost identical to CO. In ordered shared locking, transactions are ordered by lock point (the time they release their first lock), rather than commit time. However, S2PL does not allow acquisition of new locks once a lock has been released. Additionally, the moment when a transaction obtains all the locks that it needs, is usually not known beforehand. As a result, transactions release their locks at commit time. This makes CO and OSL identical policies. In fact, the four schedulers studied in [Agrawal, et al. 91] are equivalent to the RC CO, ACA CO, ST CO and RG schedulers that we discussed in Section 2.3.

2.5 Other Types of Transaction Orderings

Besides commit ordering, there exist *start time* (timestamp) and *lock point* orderings. The former is enforced by Basic Timestamp Ordering (BTO), the latter – by Basic Two-Phase Locking (Basic 2PL).

Start time ordering is a less efficient way to ensure SR because relationships between transactions are fixed once transactions start. Because of this, BTO causes too many unnecessary aborts. With CO, we define transaction ordering at commit time when the conflicts between the committing and other transactions have already occurred.

As mentioned in Section 2.4, lock point ordering is hard to achieve. Hence, instead of Basic 2PL, Strict 2PL is used which actually enforces CO.

3. On Feasibility of a Strict Serializable Scheduler

Any database scheduler needs to be SR and RC. Also, cascading aborts should be avoided. Ease of recovery dictates the need for strictness. Hence, the least restrictive database scheduler should allow any history from SR $\cap$ ST.

Suppose we do not have knowledge of the future operations of transactions. To correctly implement a strict serializable scheduler, it is necessary to keep information about committed transactions. To see why that is so, consider the following history: $r_1[x]\ w_2[x]\ c_2\ c_1$. It is strict and serializable. Unfortunately, a correct scheduler cannot commit T_2 unless either it knows somehow that T_1 will not write x later, or it memorizes the fact that another transaction has written x since the moment T_1 read it. Else, the following (ST, but not SR) history could result: $r_1[x]\ w_2[x]\ c_2\ w_1[x]\ c_1$. Both assumptions can be unrealistic. Therefore, in most cases it is not feasible to implement a scheduler that would allow all ST and CSR histories.

In this paper, we are proposing a database scheduler that is less restrictive than the S2PL scheduler and requires no knowledge of transactions future actions. Our scheduler is based on CO. Thus, it will allow all ST CO histories.

4. The Scheduler

In this section, we describe possible implementations of the ST CO scheduler.

4.1 A Naive Implementation

One way to implement a SR scheduler based on CO, is to use an optimistic commit order scheduler [Roaz 92]. To do that, a USG (a SG for all undecided transactions) is maintained. When a transaction T commits, all transactions that precede T in the USG have to be aborted to ensure CO.

The above scheduler can be modified to ensure RC. All that has to be done is aborting all transactions that have read from the transaction being aborted. This procedure should be performed in a recursive fashion.

This scheduler causes to many unnecessary aborts, for example, $r_1[x]\ w_2[x]\ c_2$ would cause T_1 to be aborted. It is possible to add delaying of commits to the scheduler to decrease the number of aborts. This is what we are going to investigate.

4.2 A More Efficient Implementation (SO1)

The first version of our semi-optimistic scheduler (SO1) is a mixed scheduler [Bernstein, et al. 87]. *wr* and *ww* conflicts are resolved using modified two-phase locking. *rw* conflicts are resolved using a Serialization Graph certifier. The certifier also ensures that the combination of the two parts admits only SR histories by making sure that all conflicts agree with commit ordering. This is another reason why the scheduler is semi-optimistic: it integrates a (pessimistic) S2PL scheduler with a pure optimistic SG certifier.

The S2PL sub-scheduler is based on the lock compatibility matrix shown in Table 1. Note that the read-write conflict is asymmetric. When a read lock has been granted on a data item, a write lock can still be acquired on the data item by another transaction. The opposite is not allowed: a write lock prevents both read and write locks from other transactions. Using two-phase locking alone with such a lock compatibility matrix, may result in non-CSR executions, for example, $r_1[x]\ w_2[x]\ c_2\ w_1[x]\ c_1$.

Lock being requested\granted	read	write
read	yes	no
write	yes	no

Table 1: Lock compatibility matrix.

Although the scheduler uses a SG certifier to validate *rw* conflicts and to ensure the S2PL and SG sub-schedulers produce compatible serialization orders, it does not actually need to maintain a serialization graph. All required information is available in the S2PL lock manager data structures. A typical implementation of lock manager data structures is shown in Figure 1 [Gray, Reuter 93]. (We added a read latch on the Lock header. Its use is described in **Procedure 1** below)

The proposed scheduler works as follows:

Procedure 1: When a *r[x]* or *w[x]* operation from transaction *T* arrives, the scheduler places a corresponding lock entry in the *x*'s lock queue and check if there are any write locks in *x*'s lock queue:

- If there is a write lock set by another transaction, *T* is blocked.
- If there are no such locks, the transaction is granted access to *x*.

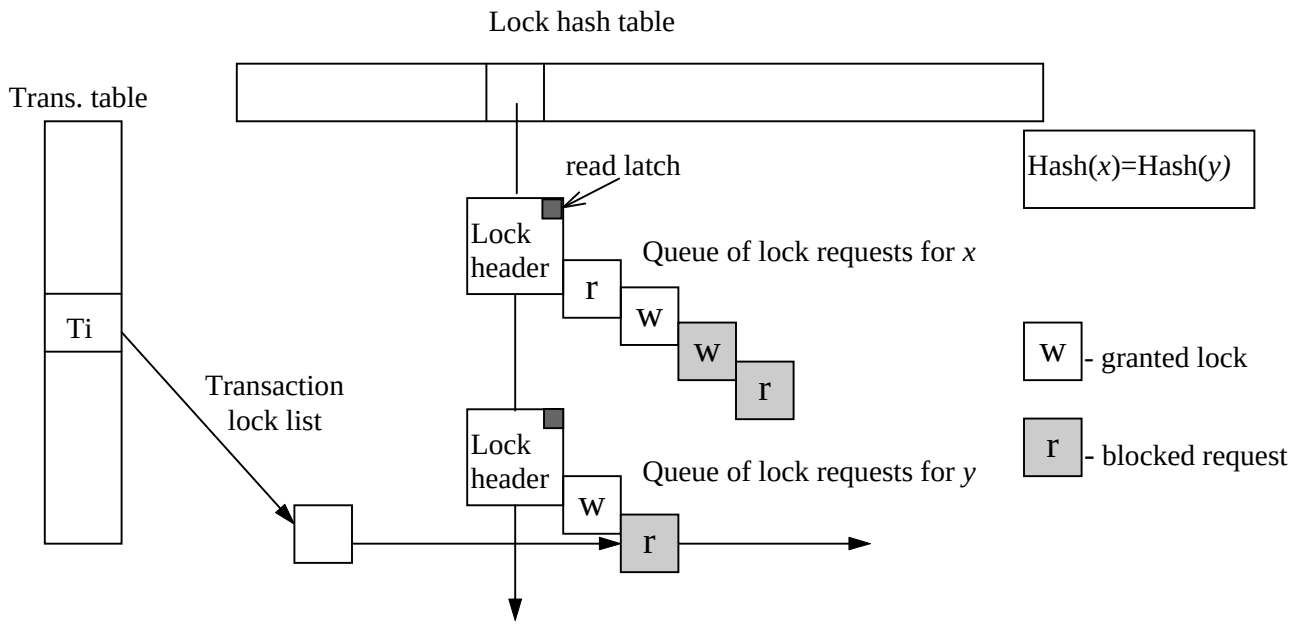

Figure 1: Lock manager data structures, adapted from [Gray, Reuter 93].

If the arriving operation is a read, the read latch in the Lock header (see Figure 1) has to be acquired. When a write operation arrives, the scheduler checks that the latch has been released. The latch is needed to provide a *handshake* between the scheduler and the data manager [Bernstein, et al. 87]. The data manager is free to change the order of operations submitted by the scheduler, e.g., $r_1[x]$ $w_2[x]$ could be performed by the data manager in the opposite order: $w_2[x]$ $r_1[x]$. The latch ensures that the scheduler does not issue a write request until all reads ahead of the write in the lock queue, have been carried out.

The latch should be implemented as a counter. Initially the counter is set to zero. When placed in the queue, each read request increments the latch counter. After the read is performed the counter is decremented. A write request can only proceed when the counter value is zero. For example, the following sequence of requests $r_1[x]$ $r_2[x]$ $r_3[x]$ $w_4[x]$ would cause the write request to wait for the three reads to complete. Note, that this does not mean that T_4 would have to wait for T_1,...,T_3 to commit. (The latch is set for only a short term.).

In Section 4.3, we introduce a better implementation of the scheduler that does not require the scheduler – data manager handshake. □

Similarly to the traditional S2PL scheduler, a deadlock may occur, e.g., $w_1[x]$ $w_2[y]$ $r_1[y]$ $r_2[x]$ would cause a deadlock. Some technique for deadlock detection, for example, the Wait-For Graph analysis or

timeout, has to be used. As a result some transactions may have to be aborted. The algorithm for deadlock detection would be very similar to the one presented in [Gray, Reuter 93].

Procedure 1 guarantees that *wr* and *ww* conflicting operations of any pair of transactions are executed in the same order. At commit time we need to make sure that *wr* conflicts and transaction commits follow the same order.

Procedure 2: When a transaction *T* tries to commits, the scheduler goes through the transaction's lock list. For each write lock set by *T*, the scheduler checks if there are any read locks ahead of it in the lock queue (no other write locks could be set).

- If all *T*'s write lock entries are at the head of the corresponding lock queues, *T* is committed. All transactions blocked by *T* are awaken so that they can resume their execution.

  The lock release operation should be performed more carefully in this scheduler than in S2PL to account for the scheduler-data manager handshake. Else, the awakened transaction may performed their physical operations in an order different from the arrival order. For example, in the following input history $w_1[x]\ r_2[x]\ r_3[x]\ w_4[x]\ c_1$, the release of T_1's write lock at commit time would awaken T_2, T_3 and T_4. This may cause T_4 to perform its write before T_2 and T_3 perform their reads.

  To avoid this problem, the scheduler should update the value of the latch counter when releasing a write lock. It should set the counter to the number of read requests from the head of the queue to the first write request. When awakened, each read request would decrement the counter after the read has been carried out. A write request cannot proceed until the counter is zero (see Procedure 1).

- If there is a read lock ahead of *T*'s write lock, *T* is blocked. Deadlocks are possible at this point. The Wait-For-Graph analysis or timeout could be used to resolve them. □

The abort procedure is similar to that of the S2PL scheduler. Write lock releases should be handled as described in Procedure 2.

Discussion

Unlike the pure optimistic SG tester, the proposed scheduler does not have to check the SG for cycles after each operation. It is also not totally optimistic because some synchronization problems are resolved (transactions are delayed or aborted) before transactions try to commit.

To prove that the scheduler produces only CSR histories, it is enough to note that all histories that the scheduler can produce are CO. Strictness (ST), which we also need, is ensured by the S2PL sub-scheduler.

The scheduler is less restrictive than the S2PL. Our scheduler would allow, for example, the following history: $r_1[x]\ w_2[x]\ c_1\ c_2$ that S2PL would not. In some situations, the larger set of allowable histories may affect response time noticeable: if we modify our example: $r_1[x_1]\ r_1[x_2]$ ... $r_1[x_1000]\ w_2[x_1]$ *(T_2 does some other work, no more conflicts with T_1)*, S2PL would block T_2 at its first write until T_1 commits. The proposed semi-optimistic scheduler would allow $w_2[x_1]$ and T_2 would be able to proceed. Of course, T_2 will still have to be blocked at commit time if it tries to commit earlier than T_1. But its entire work will be done by that time, it will only have to commit. T_2's response time would be much smaller than if S2PL were used.

It may seem that the proposed scheduler will not allow some histories that S2PL does but this is not true. The fact that we made the lock table asymmetric resulted in the possibility that some transactions that would be blocked by S2PL, are restarted by SO1. For example, given the following sequence of operations: $r_1[x]\ w_2[x]\ r_2[y]\ w_1[y]\ c_1\ c_2$, S2PL would block T_2 (when $w_2[x]$ arrives*)* whereas SO1 would restart T_1 (when c_1 arrives). This, however, does not affect the set of allowable histories (that are not delayed or restarted).

The question of which approach is better: restarting (the optimistic approach) or delaying (the pessimistic approach) transaction is still open. Several studies have shown that optimistic concurrency control policies may perform better than pessimistic ones [Bernstein, et al. 87, Salem, et al. 94]. Generally, if resource contention and/or data contention are not high, optimistic policies result in better performance. In most real situations, data contention is low. TPC-C [TPC 97], a well-known OLTP benchmark, for example, assumes that potentially conflicting requests are issued no more often than once in approximately 5 seconds. Most tested systems maintained request response time lower than that, which means that there was absolutely no data contention. This definitely favors more optimistic policies .

The major problem of optimistic policies is frequent restarts. Unless these restarts are fast, an optimistic scheduler may degrade the performance of the system. One easy way to make restarts fast and simple is to use deferred updates. In section 4.3, we modify the algorithm to include deferred updates.

4.3 Adding Deferred Updates (SO2)

With deferred updates, each transaction has a private workspace where it performs all update operations. At commit time updates are atomically propagated into the database (commits are serialized). Of course, some concurrency control method still have to be used to ensure serializability.

If deferred updates are used, aborting and restarting a transactions are very simple. Nothing has to be reversed in the database because transactions do not update data before they are committed. Another

advantage of deferred updates is that they allow for an easy step from conflict serializability to the more general view serializability [Mizuno, et al. 94].

To fully reap the benefits of deferred updates, we will make write locks of S2PL compatible so that two or more transactions could make concurrent deferred updates on the same item.

The second version of our scheduler (SO2) is an integration of two schedulers: *rw* conflicts are synchronized by 2PL; *wr* and *ww* conflicts are resolved by a SG certifier. The certifier is also responsible for integrating the sub-schedulers by checking for identical ordering of conflicts at commit time. The certifier's job is greatly alleviated by the atomicity of deferred updates.

The new lock compatibility matrix looks as follows:

Lock being requested\granted	read	write
read	yes	no
write	yes	yes

Table 2: Lock compatibility matrix when deferred updates are used.

Only read requests for a write-locked data item are blocked immediately in this scheduler. Other types of conflicts are resolved at commit time.

This scheduler works as follows:

Procedure 1: When a *r[x]* or *w[x]* operation from transaction *T* arrives, the scheduler places a corresponding lock entry in *x*'s lock queue. If the arrived operation is *r[x]* the scheduler also checks if there are any write locks in *x*'s lock queue:

- If there is a write lock set by another transaction, *T* is blocked.
- If there are no such locks, *T* is granted access to *x*.

Similarly to S2PL and SO1, deadlock detection may have to be performed here. □

Procedure 2: When a transaction *T* tries to commit, the scheduler goes through the transaction's lock list. For each write lock set by *T*, the scheduler checks if there are any read or write locks ahead of it in the lock queue.

- If there is a read lock ahead of *T*'s write lock in at least one queue, *T* is blocked. Deadlocks may need to be resolved at this point.
- If all *T*'s write lock entries are in the head of the corresponding lock queues, *T* is committed.

- If there are only write locks (or no locks at all) ahead of *T*'s write lock in *all* lock queues, *T* can be committed. This would move *T* ahead of other transactions updating the same data and change the ordering of the *ww* conflict the transactions are involved in. □

The abort procedure is identical to that of the S2PL scheduler.

Theorem. *SO2 ensures view serializability of histories.*

Proof:

To prove that the scheduler produces only VSR histories, we will use the definition of VSR. For each prefix *H'* of a given history *H* produced by the scheduler, we consider a serial history *H"* consisting of the committed transactions in *H'*. The order of transactions in *H"* is to agree with their commit order in *H'*. We are going to prove that *H"* is view equivalent to *H'*. According to the definition of view equivalence, we need to prove that all read-from relationships between transactions in *H'* are identical to those in *H'*.

First, we note that if T_2 is present in *H"*, T_1 also must be there. This is because T_2 is blocked until T_1 commits. Therefore, if T_2 is committed in *H'*, T_1 is committed too.

Suppose that transaction T_2 reads *x* from transaction T_1 in *H'* but it reads *x* from a different transaction (T_3) in *H"*. Two orderings of the transactions in *H"* are possible:

- $T_1<T_3<T_2$

  This ordering would mean that T_3 commits earlier than T_2 in *H'*. In this case, SO2 would ensure that T_2 reads *x* from T_3, the last committed transaction to write *x*. According to *H'*, this did not happen. Thus, such ordering of transactions in *H"* is not possible.

- $T_3<T_2<T_1$

  This ordering would mean that T_2 committed before T_1 in *H'*. This is not possible because T_2 reads from T_1 in *H'* and, therefore, blocked till the latter commits.

  Thus, T_2 does read *x* from T_1 and *H"* is view equivalent to *C(H')*. We have proved that the scheduler allows only view serializable executions. □

Discussion

Unlike the scheduler in Section 4.2, this version of scheduler does not require any handshake between the scheduler and the data manager. Due to deferred updates, a physical write can never happen until the read

ahead of it in the lock queue is performed. This is because transactions do not write before their commit time and they cannot commit until all preceding read locks have been released.

5. Conclusion

We have proposed a semi-optimistic database scheduler that does not block write requests for read-locked data items. The scheduler uses commitment ordering to ensure serializability.

We have presented two implementations of the scheduler: one that does not uses deferred updates and another that does. We have demonstrated that the proposed scheduler can be easily implemented using the data structures of a typical S2PL scheduler.

6. References

[Agrawal, et al. 94] D. Agrawal, A. El Abbadi, A. E. Lang. Performance Characteristics of Protocols with Ordered Shared Locks. In *Proceedings of the Int. Conf. On Data Engineering*, Kobe, Japan, April 1991.

[Bernstein, et al. 87] P. Bernstein, V. Hadzilacos, N. Goodman. *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987.

[Gray, Reuter 93] J. Gray, A. Reuter. *Transaction Processing.* Morgan Kaufmann Publishers, USA, 1993.

[Mizuno, et al. 94] M. Mizuno, J. Zhou, G. Singh. An MWRW Based Strict Two Phase Lock Scheduler. In *Proc. of the Int. Conf. On Computing and Information, ICCI'94.* Peterborough, Canada, May 1994.

[Roaz 92] Y. Roaz, The Principle of Commitment Ordering, or Guaranteeing Serializability in a Heterogeneous Environment of Multiple Autonomous Resource Managers Using Atomic Commitment. *In Proc. Of the 18th VLDB Conference,* Vancouver, Canada, 1992.

[Salem, et al. 94] K. Salem, H. Garcia-Molina, J. Shands. Altruistic Locking. *ACM Trans. on Database Systems*, 19(1), March 1994.

[TPC 97] Transaction Processing Performance Council. *TPC Benchmark C, Standard Specification, Revision 3.3*. April 1997.