The glass is half full

On the application of an optimistic approach to concurrency control schemes

Dominik Moritz

dominik.moritz@student.hpi.uni-potsdam.de

Seminar Paper
Beauty is our Business, Summer term 2012

September 15, 2012 (Revised May 25, 2013)
Supervised by Prof. Dr. Felix Naumann

1 WHY LOCKING CAN BE PROBLEMATIC

Concurrency control (CC) avoids conflicts by coordinating possibly conflicting access to shared data from concurrent transactions. A *transaction* is a sequence of accesses that by itself preserves consistency. Conflicts among concurrent transactions are prevented by providing a single user system view and guaranteeing ACID properties [2]. Consider the example of a Wikipedia[1] article that is edited and read by multiple users. Providing a consistent article to readers can be achieved fairly easy. However, problems may occur when multiple users edit the same article, sentence or even word.

Conventional CC schemes, such as two-phase locking, prevent concurrent access by dynamically acquired locks associated with a *resource* (an article) and a *process* (a user). In the case of an article locks have the apparent disadvantage of *deadlocks*. A user might start editing an article without committing the transaction (saving the article). In general, locking approaches have the following inherent disadvantages:

1. Lock maintenance and deadlock detection require considerable overhead. In System R the overhead adds up to 10% of the total execution time [5].
2. There is no general-purpose deadlock-free protocol that always provides high concurrency.
3. Concurrency is substantially lowered if congested nodes (central and often accessed objects) have to be locked. This problem becomes even more significant if the locking transaction waits for secondary memory access.
4. Locks cannot be released before a transaction commits, because in the event of an abortion locks are required to roll back the changes made by the transaction.
5. Most importantly, locking may be necessary only in the worst case. Locking hunts for too strong preventive actions. Consider the example of equally distributed access from two processes on n objects. In such case the probability for a conflict is $P = 1/n$ and locking is only required every n transactions. The problem of premature locking particularly applies if congested nodes are rarely accessed and only few objects are affected by transactions at a given time.

Optimistic approaches annihilate the mentioned disadvantages by eliminating locks at least in most cases. This seminar paper is based on a paper by H.T. Kung and John T. Robinson from 1982 [7] and discusses the basic ideas of optimistic CC schemes and its applications.

[1] `http://wikipedia.org/`

2 OPTIMISTIC METHODS FOR CONCURRENCY CONTROL

Premature and unnecessary locking of shared resources is the main disadvantage of pessimistic CC schemes that check for conflicts before an operation is allowed to access a resource. In a CC scheme that employs optimistic methods the burden of CC is deferred until the *end of transaction* (EOT, prepared to commit) in contrast to a pessimistic approach where CC normally takes place at the *begin of a transaction* (BOT). Checks for conflict cannot be deferred further, because they have to take place before the transaction is committed. This approach is called optimistic because it explicitly assumes that the probability of data conflict is low.

There are two main ideas behind *optimistic concurrency control* (OCC). First, since reading can never cause a loss of integrity, queries are not restricted. However, the results of a read-only query have to be validated before returned, because the state of the database may have changed during query execution. The second idea is that writes have to be restricted rigorously to preserve integrity since writing transactions may affect other transactions. The execution of transactions under OCC is split into three phases (Figure 1), the *read phase*, the *validation phase*, and the *write phase*. During the read phase, no global writes occur and all writes and successive reads take place on *tentative copies*. Then, the tentative copies are made global during the write phase if during the validation phase it can be ensured that committing the changes cannot cause a loss of integrity. In case a conflict is detected, typically the transaction is aborted and restarted, although other resolution schemes, such as merging, ignoring, or logging, are possible. The problem of starvation in an OCC scheme can be addressed by giving exclusive access to a starving transaction.

Employing OCC increases concurrency, because transactions do not have to wait for locks to be released before changes can be made. Another advantage over locking is that OCC schemes are deadlock free because no locks are used.

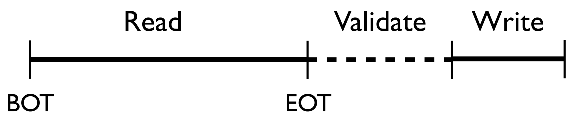


Figure 1: Three phases of a transaction in OCC

3 READ PHASE AND WRITE PHASE

In this section we briefly discuss how the read and write phase can be implemented in a way that they behave as if basic operations are used directly. The basic operations: (*write*, *read*, *edit*, *create*, and *delete*) are assumed to be supported by the underlying DBMS.

3.1 Read phase

For each transaction the DBMS has to maintain a *transaction buffer* that contains tentative copies of created or edited records. A read or write to a record will be first redirected to the transaction's buffer. If the record cannot be found, the global record will be queried. Modified objects have to be stored in the transaction buffer and repeated modifications have to take place on the local objects as well, which requires operations to be rewritten as seen in Listing 1. The transaction buffer supports abortion of partially executed transactions as well as completely executed, but conflicting transactions. It also enables simple commit as described in Section 3.2.

During the read phase, the transaction buffer maintains at least four sets: the *read set*, *write set*, *create set* and the *delete set*. These sets are emptied during the transaction buffer's initialization in `txn_begin`. `txn_end`, as discussed in Section 4.3 and 4.5, is called when the read phase, being in fact the body of the user-written transaction that uses the functions in Listing 1, is finished.

```
# n: object/ record
# v: value

def txn_write(n, v):
    if n in create_set:
        write(n, v)
    elif: n in write_set:
        write(copies[n], v)
    else:
        copies[n] = copy(n)
        write_set.append(n)
        write(copies[n], v)
```

```
def txn_create():
    n = create()
    create_set.append(n)
    return n

def txn_read(n):
    read_set.append(n)
    if n in write_set:
        return read(copies[n])
    else:
        return read(n)

def txn_delete(n):
    delete_set.append(n)
```

Listing 1: Basic transaction operations for the read phase

Note that uniqueness constraints are not supported. Also note that direct modification of global state without having acquired the appropriate locks suggests the idea of snapshot based CC schemes of which one is multi version concurrency control [1].

A query transaction that solely consists of read operations is automatically committed after it passes the validation phase. Writing transactions however, require a write phase as discussed in the next section.

3.2 Write phase

In the write phase, a writing transaction declares its local transaction buffer global. Hence, *writing* does not necessarily mean that records are made persistent but that new records become visible to the global context. Therefore other transactions observe the new values only after the write phase, which requires the modifying transaction to have passed the validation test.

Splitting the transaction into parts, as described in Section 2, results in knowledge about all affected objects at the end of the write phase, which is a valuable advantage for recovery mechanisms. Transactions would flush their REDO information at the end of the write phase [5].

Listing 2 collects the basic operations for the write phase. Note that the clean up, which deletes the transaction buffer, is also necessary if the transaction aborts. However, the clean up is not necessarily part of the write phase and can be executed separately since no other transactions are affected.

```
def commit():
    for n in write_set:
        swap(copies[n], n)
    cleanup()
```

```
def cleanup():
    for n in write_set:
        delete(copies[n])
    for n in delete_set:
        delete(n)
```

Listing 2: Basic operation for the write phase

The commit procedure has to be efficient, because it is often executed serially to avoid conflicts. In fact, it is very efficient, because all objects are only accessed be reference and exchanging the object only requires modifying the pointers. Further optimizations are possible in a way that commit operations that do not affect each other are written in parallel.

4 VALIDATION

Validation is a crucial part of a transaction in an OCC scheme, which guarantees that the database continues to be in a consistent state. It is run after a transaction signals EOT meaning that it has finished making changes in the local transaction buffer and before the local changes are made global. Validation fails if the transaction would cause a loss of integrity or return wrong data when being committed.

Read-only transactions can be easily verified by making sure that the read values are equivalent to the current values. For writing transactions a more sophisticated approach is necessary to verify correctness. A common criterion for verifying the correctness of concurrent transactions is that *serial equivalence* or *serialisability* should not be violated. It may be defined as follows [2]:

Let $T_1, T_2...T_n$ be a set concurrently executed transactions, which can each be described as functions:

$$\begin{aligned} T_i : \quad & D \to D \\ & d_m \to d_{m+1} \end{aligned} \tag{1}$$

D is the set of possible database states, meaning an assignment of values to the variables in the shared data structure. Each transaction has to preserve integrity, meaning that if $d \in D$ satisfies all integrity criteria, then $T(d) \in D$ satisfies all integrity criteria as well.

If a concurrent execution transforms a database from its initial state $d_{initial} \in D$ to a final state d_{final} it is considered serialisable iff (if and only if) there is a permutation $\pi = 1, 2, 3, \ldots n$ that defines an execution order which has the same result as the concurrent execution. Serialisability may be formalized as

$$\exists \pi : d_{final} = T_{\pi(n)} \circ T_{\pi(n-1)} \circ \ldots \circ T_{\pi(1)}(d_{initial}) \tag{2}$$

where $\circ$ denotes functional composition or serial execution. The idea behind this criterion is that $d_{initial}$ is considered to be a consistent database state. By repeated application of the integrity-preserving property of T_i, d_{final} has to satisfy all integrity criteria as well.

Two approaches to validate that serial equivalence is not violated are discussed in Section 4.2.

Serialisability is the weakest criterion to prove that concurrent execution preserves integrity, which means that if an execution is seralizable then it has to preserve integrity [6]. Only complete syntactic information is necessary to verify consistency with the help of the serialisability. However, if additional semantic information is available, other approaches may be more effective [3].

4.1 Transaction numbers

In order to verify serialisability it is necessary to determine the order of transactions. This can be handled by explicitly assigning unique transaction numbers that determine the order of transactions in a serially equivalent schedule. We define that T_i has to be executed before T_j iff $i < j$. Consequently the permutation π in Equation 2 is not an arbitrary permutation but well defined. This definition allows faster validation of serial equivalence since only one order has to be evaluated, instead of $n!$ possible, arbitrary orders.

Transaction numbers are determined by a global counter that is incremented when a number is read. They have to be assigned before the transaction is validated because knowledge of the transaction number of the transaction that is being validated is required in the approaches in Section 4.3 and 4.5. Assigning transaction numbers at BOT is considered pessimistic because of issues with long running transactions. Consider a long running (long read phase) transaction T_0 and a short running transaction T_1. Both start at roughly the same time but T_1 has to wait for the completion of T_0's read phase before it can start its validation phase since the validation of T_1 relies on the knowledge of the write set of T_0 (see Section 4.2). Conceptually, we consider the transaction to execute at transaction number assignment.

Optimistically deferring assignment until EOT allows immediate validation without the necessity of waiting for any transactions. Furthermore validation is simplified because some checks can be omitted. Also note that the second part of Condition 3 is automatically satisfied if transaction numbers are assigned optimistically.

4.2 Validation of serial equivalence

Equation 2 can be directly applied as the validation in concurrency control. Since transaction numbers are explicitly assigned as described in Section 4.1, to guarantee serialisability with $\pi = 1, 2, 3, \ldots$ each pair of transactions T_i and T_j with $i < j$ must observe two rules [5].

First, *no read dependency*. T_j not read any data modified by T_i. Second, *no overwrite*. T_j does not overwrite any data that has been written by T_j. In order to satisfy these rules, one of the following conditions must be true.

Condition 1: T_i completes its write phase before T_j starts its read phase.

Condition 2: The write set of T_i does not intersect the read set of T_j, and T_i completes its write phase before T_j starts its write phase.

Condition 3: The write set of T_i does not intersect the read set or write set of T_j, and T_i completes its read phase before T_j completes its read phase.

Condition 1: no time overlap
Condition 1 as seen in Figure 2 expresses that T_i completes before T_j even starts. This property clearly is the situation in a serial schedule and does not require further verification.

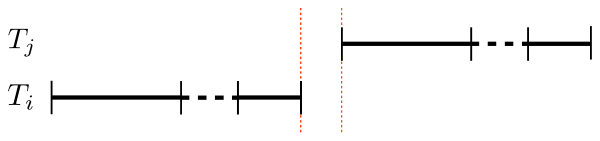
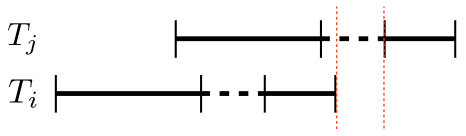


Figure 2: First condition

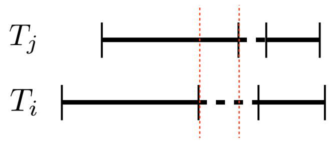
Condition 2: no dirty read
As seen in Figure 3, Condition 2 states that T_i does not overwrite any data modified by T_j (no *lost update*) and T_j does not read any modified data (no *dirty read*).



$$WriteSet(T_i) \cap ReadSet(T_j) = \varnothing \tag{3}$$

Figure 3: Second condition

Condition 3: no data overlap
Condition 3 as seen in Figure 4 can be summarized similar to Condition 2. In contrast, it is not required that T_i finished writing before T_j starts writing. In fact it is only required that T_j operates on unmodified data.



$$WriteSet(T_i) \cap (ReadSet(T_j) \cup WriteSet(T_j)) = \varnothing \tag{4}$$

Figure 4: Third condition

4.3 Serial validation

In this section we look at an implementation of Conditions 1 and 2 in which write phases never overlap. This can be implemented by planting transaction number assignment, validation and the write phase in a critical section which is marked by surrounding $< ... >$. The implementation for such an approach is shown in Listing 3. In `txn_begin`, in addition to clearing the transaction buffer as explained in Section 3.1, a `start_tn` is read from the global counter (`tnc`).

```
def txn_end():
  < finish_tn = tnc; valid = true
    for t in range(start_tn, finish_tn):
        if t.write_set.intersection(read_set):
            valid = false
    if valid:
        commit()
        assign_transaction_number() >
        cleanup()
    else:
        abort()
```

Listing 3: Implementation of serial validation.

Because of Condition 1, committed transactions do not have to be considered. Omitting transactions is done by only taking transactions between `start_tn` and `finish_tn` into account. Condition 2 is verified by confirming that no objects are read that have been written by a previous transaction.

Note that transaction numbers are only assigned if validation has been successful, which has the same effect as assigning it at the beginning of `txn_end`. The only difference is that no transaction numbers are assigned to aborted transactions.

Staging some checks outside the critical section is a second optimization that can be made. The idea behind this optimization is to check intersection with the write sets of transactions until a `mid_tn` and only check the remaining transactions in the critical section as can be seen in Listing 4. Staging can repeated multiple times reducing the number of affected transactions in each step.

```
def txn_end():
    mid_tn = tnc; valid = true
    for t in range(start_tn, mid_tn):  # 1st stage
        if t.write_set.intersection(read_set):
            valid = false
  < finish_tn = tnc
    for t in range(mid_tn, finish_tn):  # 2nd stage
        if t.write_set.intersection(read_set):
            valid = false
    [...] # same as in listing 3
```

Listing 4: Staging as optimization of serial validation

In order to achieve higher concurrency, which becomes necessary if the write phase is long compared to the read phase (which is not necessarily the case), a modified approach is necessary. Such an approach that uses all three conditions is presented in Section 4.5.

4.4 Validating queries

Queries are read only transactions that lack a write phase resulting in no need to be assigned a transaction number. This implies that validation of queries can take place outside a critical section. Without a critical section, the staging optimizations from Section 4.3 or parallel validation as in Section 4.5 are not necessary. Validation in query-dominant systems often consists solely of reading `finish_tn` and comparing it to `start_tn`. Since both numbers would be the same, no further actions are necessary making OOC optimal for such systems.

4.5 Parallel validation

In this section we examine parallel validation which allows greater concurrency, however at higher total costs, compared to serial validation. It takes all three condition from Section 4.2 into account and thus allows interleaved writes. This property is achieved by checking the intersection of the write set with the write and read sets of all transactions that could potentially modify the same objects. These transactions that have completed their read phase but have not yet finished their write phase are called *active transactions*.

The idea behind this implementation is that at least one of two transactions in their validation phase that conflict with one another is invalidated. Note that only modifications to the active and transaction numbers have to be placed in a critical section to maintain the invariant that the active set contains only previous transactions.

Unfortunately the above implementation entails the possibility of invalidating a transaction by a transaction that itself is invalidated eventually. A partial solution is reducing the size of the active set by staging checks (analogous to

```
def txn_end():
  < finish_tn = tnc
    finish_active = get_active_transactions()
    set_active()
    valid = true >
    for t in range(start_tn, finish_tn):
        if t.write_set.intersection(read_set):
            valid = false
    for t in finish_active:
        if t.write_set.intersection(read_set.union(write_set)):
            valid = false
    if valid:
        commit()
      < assign_transaction_number()
        set_inactive() >
        cleanup()
    else:
      < set_inactive() >
        abort()
```

Listing 5: Implementation of parallel validation

Section 4.3) before setting the transaction active. A more sophisticated solution is to wait for causing transactions to finish and only abort if the transaction successfully commits.

5 CONCLUSIONS

OCC can be seen oppositional to locking in terms of conceptual differences as pointed out in Section 2, which leads to the characteristics discussed in Section 5.1. Because of its advantages over locking, OCC has many applications as described in Section 5.2.

5.1 Comparison with pessimistic methods

Transactions in a locking approach are controlled by waiting for locks and ordered by access time. In contrast, in an optimistic CC scheme transactions are controlled by backup and ordered by number assignment. This difference leads to deadlock being the major disadvantage of locking and starvation the major flaw of OCC. In a blocking scheme conflicts with any possible transaction are prevented while in OCC a transaction will be restarted if there is any possibility of having a non-serialisable schedule only with actual transactions. Consequently, an optimistic approach is not inherently worse than a locking scheme validation will only fail if locking would have been necessary in a locking approach.

OCC may well be superior to locking in systems where conflicts are unlikely, such as query dominant systems where the overhead of CC becomes negligible as described in Section 4.4. However, conflicts may not be as rare as suggested, since there are hot spots in realistic access patterns [4]. Therefore restarting is not always preferred over locking.

5.2 Applications and outlook

CC problems arise in operating systems, communication systems, and database systems, among others. Therefore OCC methods are applied in a variety of systems, such as MediaWiki (and Wikipedia), version control systems, and many major database systems. Popular database systems that implement OCC schemes are PostgreSQL, MySQL, ElasticSearch, CouchDB, and DB 2. However, most systems use derived, snapshot based OCC schemes; the most renowned being *Multi Version Concurrency Control* (MVCC) [1].

Overall, OCC is an efficient method for CC, especially in query dominant systems. Parallel validation as presented in Section 4.5 appears to be an efficient and parallelisable implementation to validate that execution preserves integrity. Even though, there are cases with high probability for conflicts in which an optimistic approach is not reasonable, OCC has gained a position as a very effective CC scheme with many practical applications.

REFERENCES

[1] Philip A. Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM Comput. Surv.*, 13(2):185–221, June 1981.

[2] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[3] Abdel Aziz Farrag and M. Tamer Özsu. Using semantic knowledge of transactions to increase concurrency. *ACM Trans. Database Syst.*, 14(4):503–525, December 1989.

[4] Lei Guo, Enhua Tan, Songqing Chen, Zhen Xiao, and Xiaodong Zhang. The stretched exponential distribution of internet media access patterns. In *Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing*, PODC '08, pages 283–294, New York, NY, USA, 2008. ACM.

[5] Theo Härder. Observations on optimistic concurrency control schemes. *Inf. Syst.*, 9(2):111–120, November 1984.

[6] H. T. Kung and C. H. Papadimitriou. An optimality theory of concurrency control for databases. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, SIGMOD '79, pages 116–126, New York, NY, USA, 1979. ACM.

[7] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. In *Proceedings of the fifth international conference on Very Large Data Bases - Volume 5*, VLDB '79, pages 351–351. VLDB Endowment, 1979.