Lab 4 Project Proposal – Optimistic Concurrency Control [Page Level]

Valerie Kwek

1. Project Description

In lab 3, we implemented 2 phase locking for concurrency control, which consists of shared read locks and an exclusive write lock for each page. To uphold the ACID properties, a page may either have multiple transactions reading from it or a single transaction writing to it; a page cannot have transactions writing and reading to it unless the transactions have the same transaction ID because otherwise there will be a race condition depending on who reads or writes first.

As an improvement to lab 3, we now want to implement optimistic concurrency control (OCC) on the page level. This is optimistic because it does not hold locks from the beginning of a transaction to the eventual committing or aborting of the transaction; instead, it will abort if it finds a conflict in the validation phase when trying to commit. This method performs well if concurrent transactions are rare/if there is low contention.

More concretely, OCC consists of 3 phases: the read phase, the validation phase, and the write phase. The read phase consists of a transaction making a copy of the page it needs to read from or write to; if it needs to write to the page, all changes will be made to this copy instead of the original page in the database. When a transaction wants to commit, it will then proceed to the validation phase. The validation phase will check whether any transactions that have been committed since the beginning of the transaction have modified the data that was read or written by the transaction.

From lecture 14, the formal validation conditions to abort transaction T_j (where $T_i < T_j$) are:

1. $W(T_i) \cap R(T_j) \neq \{\ \}$ and T_i does not finish writing before T_j starts
    - This must abort as T_j may not see everything that T_i wrote
2. $W(T_i) \cap (W(T_j) \cup R(T_j)) \neq \{\ \}$ and T_j overlaps with T_i validation or write phase
    - This must abort as all of T_i's writes must appear before any of T_j's writes

This can also be rephrased as a transaction T_j can commit if at least one of the following conditions is true for all $T_i < T_j$:

1. T_i completes its write phase before T_j starts its read phase
2. $W(T_i)$ does not intersect $R(T_j)$, and T_i completes its write phase before T_j starts its write phase
3. $W(T_i)$ does not intersect $R(T_j)$ or $W(T_j)$, and T_i completes its read phase before T_j completes its read phase
4. $W(T_i)$ does not intersect $R(T_j)$ or $W(T_j)$, and $W(T_j)$ does not intersect $R(T_i)$

If there is a conflict, we need to abort the transaction; otherwise, we can proceed to the write phase. The write phase will flush all the dirty page copies that were written to by the transaction to the disk.

2. Implementation Plan

For ease of access, we retained the RWPerm enum for read/write permissions and created a TransactionPhase enum for read/validation/write phase.

We primarily needed to change buffer_pool.go from our lab 3 implementation. To set up for OCC, we first needed to change the BufferPool struct. The BufferPool struct will still have the fields pages (maps page keys to pages), numPages (limit of buffer pool pages), currPage (current page in buffer pool), dirtyPages (maps transaction IDs to dirty page keys), sharedPages (maps transaction IDs to page keys they have read from), and mutex (for buffer pool). We will add on the fields runningTransactions (maps running transaction IDs to transaction phases), transactionPages (maps transaction IDs to a map that correlates page keys to copied pages), and concurrentAccessRecord (maps transaction IDs to a map of other conflicting transaction IDs which each correlate to another map relating page keys that the conflicting transaction IDs used to the page's permissions/whether the conflicting transaction ID read or wrote from the page). The concurrentAccessRecord is important because it is used for validation to see what other transactions were concurrently accessing the same page and what kind of access they had, which is used to determine whether a transaction should abort or not.

We additionally need to change the functions BeginTransaction, GetPage, CommitTransaction, and AbortTransaction in buffer_pool.go.

BeginTransaction checks that the transaction is not already running and then sets the given TID to ReadPhase in runningTransactions. It also initializes the TID's entry in concurrentAccessRecord and transactionPages to empty maps.

GetPage makes a local copy of the requested page if it does not exist in transactionPages for the passed in TID and the permission is a WritePerm (meaning that the page has never been written to before for this TID and we now want to write to it). If the page exists in transactionPages for the TID, we use this copy to edit. If the permission is a ReadPerm and does not exist in transactionPages, we can just return the original page without making a copy. We also update the sharedPages or dirtyPages data structure depending on if it was a read or write permission, respectively.

The GetPage design is based off of Lecture 14’s pseudocode for reads and writes:

```
1. Read:  tread(object):
                read_set = read_set U {object};
                if object in write_set:
                        return read(copies[object]);
                else:
                        return read(object);
2. Write: twrite(object,value):
                if object not in write_set: // never written, make copy
                        m = read(object)
                        copies[object] = m
                        write_set = write_set U {object}
                write(copies[object], value)
```

CommitTransaction checks for conflicts in the validation phase by iterating through the concurrentAccessRecord. The concurrentAccessRecord has a map of potential conflict TIDs with the pages they read and wrote to. For each potentially conflicting TID, we iterate through the page keys they wrote to. If the TID we are trying to validate also read or wrote from the same page key, we need to abort ($W(T_i) \cap R(T_j) \neq \{ \}$, and T_i does not finish writing before T_j starts or $W(T_i) \cap (W(T_j) \cup R(T_j)) \neq \{ \}$, and T_j overlaps with T_i validation or write phase). If we do not abort, we proceed to the write phase. We flush all the pages that were dirtied by the TID by iterating through the pages keys saved in dirtyPages. We also add the TID with its page keys + permissions as a concurrent access/potential conflict to all TIDs in concurrentAccessRecord so that the transaction can be checked against in later validations. Finally, after saving the necessary information in concurrentAccessRecord, we can clear the TID’s entries in transactionPages, concurrentAccessRecord, dirtyPages, sharedPages, and runningTransactions.

AbortTransaction clears the TID’s entries in transactionPages, concurrentAccessRecord, dirtyPages, sharedPages, and runningTransactions. It additionally removes the TID as a potential conflict TID in the entries of concurrentAccessRecord as an aborted transaction can no longer potentially conflict with other concurrent transactions.

3. Testing

As a baseline, the new implementation for concurrency control passed the test cases for lab 3 except for the deadlock tests as we no longer have lock acquisition (we will instead just abort the transaction); these extensively tested that the implementation properly updated the database for single and multithreaded programs. Because we no longer can use the old deadlock tests, we needed to include tests to check that conflicting read + write operations have at least one aborted transaction and that conflicting write + write operations have at least one aborted transaction. To test my implementation, I wrote four different test cases: TestInvalidateWriteWrite, TestInvalidateWriteRead, TestValidateReadWrite, and TestValidateReadRead.

TestInvalidateWriteWrite constructs an invalid situation where tid1 writes t1 to page → tid2 writes t2 to same page → tid1 tries to commit → tid2 tries to commit; the outcome should be that tid1 commits and tid2 aborts. TestInvalidateWriteRead constructs an invalid situation where tid1 writes t1 to page → tid2 reads from same page + writes t2 to different page → tid1 tries to commit → tid2 tries to commit; the outcome should be that tid1 commits and tid2 aborts. TestValidateReadWrite constructs a valid situation where tid1 reads from page + writes t1 to different page → tid2 writes t2 to same page that tid1 read from → tid1 tries to commit → tid2 tries to commit; the outcome should be that tid1 commits and tid2 commits. TestValidateReadRead constructs a valid situation where * tid1 reads from page + writes t1 to different page → tid2 reads from same page tid1 read from + writes t2 to different page than what tid1 read and wrote to → tid1 tries to commit → tid2 tries to commit; the outcome should be that tid1 commits and tid2 commits.

This is sufficient to demonstrate a working implementation because it tests all possible combinations of 2 transactions interleaving. The tests TestInvalidateWriteWrite and TestInvalidateWriteRead specifically test for the abort conditions for validation $(W(T_i) \cap (W(T_j)$ U $R(T_j)) \neq \{\ \}$, and T_j overlaps with T_i validation or write phase or $W(T_i) \cap R(T_j) \neq \{\ \}$, and T_i does not finish writing before T_j starts).

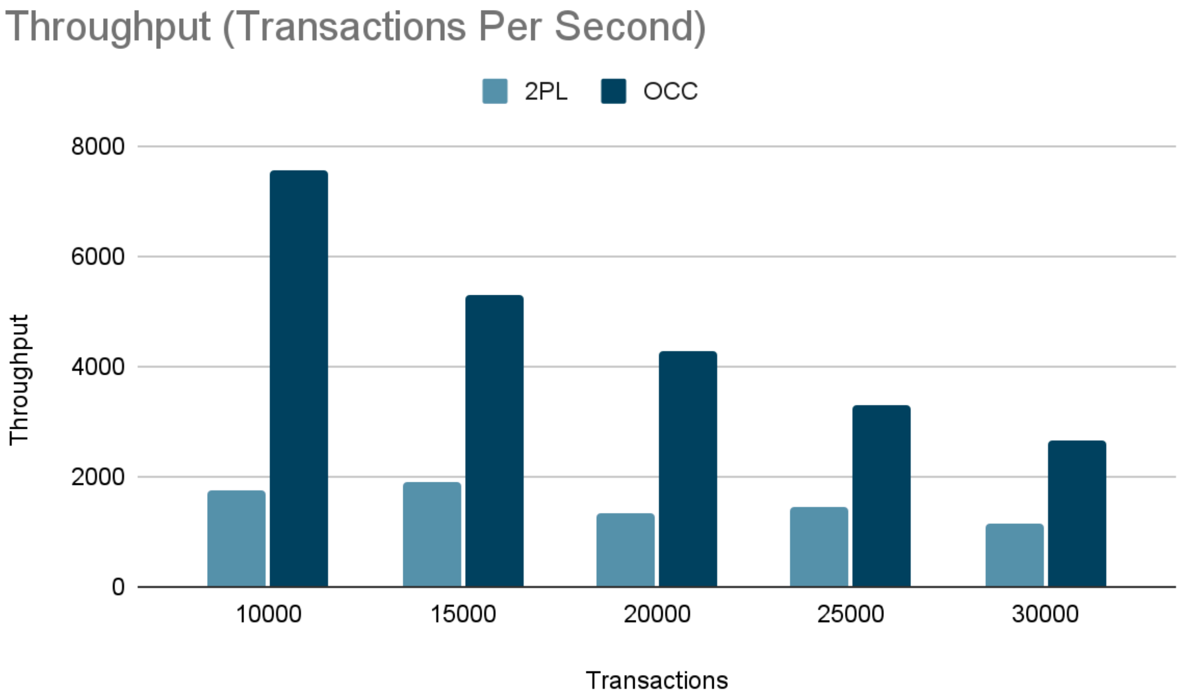
4. Evaluation

For evaluation, we expect that OCC will perform better for tests with low contention compared to the 2PL implementation from lab 3. We tested this on 10000, 15000, 20000, 25000, and 30000 transactions that were run serially (no concurrent transactions) and took an average of the times run over 5 trials.

Here is a table of the result times:

Transactions	**10000**	**15000**	**20000**	**25000**	**30000**
OCC	1.321s	2.825s	4.685s	7.616s	11.300s
2PL	5.695s	7.823s	15.074s	17.451s	26.546s

Here is a graph of the throughput for both implementations:



Overall, we can see that the low contention series of transactions ran a lot faster/had a higher throughput with OCC than 2PL. This makes sense because OCC does not have to wait for locks, resulting in a faster runtime.

5. Challenges + Future Work

The most challenging part of this project was figuring out how to design the layout of OCC. I had a lot of trouble piecing together what concurrentAccessRecord should look like and how it could be used to check when a transaction could abort (in the end, I chose to store more information in concurrentAccessRecord than the recommended approach so that we mainly only needed to depend on this data structure for checking conflicts). It was also a lot of work to write up the validation tests because I had never written tests before in Go as previous labs already provided the test cases.

Everything seems to be working as I expected/explained in previous sections. I did not expect that the throughput for OCC would decrease with an increasing number of transactions (we can also see a little bit of this happening for 2PL as well); this might be due to overhead from running so many transactions. With extra time, it would be interesting to write further tests on more complicated series of interleaving transactions and validate that our OCC implementation is working.

6. Code Repository

Here is the GitHub repo: https://github.com/valkwek/Databases-Final-Project

In particular, this project focused on changing buffer_pool.go. The 2PL implementation from lab 3 is now in buffer_pool_copy.go. The additional tests can be found in validation_test.go. All testing and evaluation were done in locking_test.go, validation_test.go, and transaction_test.go.