Team 12, 104060006 關天睿 104060020 蔡念軒

# Database Final Paper Implementation:

# On Optimistic Methods for Concurrency Control

Journal: TODS
Year of Publication: 1981
Authors: H.T. KUNG and JOHN T. ROBINSON
Citation Count: 1678

## 1. Paper Review

This paper first addresses the issues and overheads caused by maintaining the lock protocols for concurrency control. Then proposes a non-locking concurrency control method that performs more efficiently than locking ones in most “optimistic” situations. We briefly describe the motivation and the key ideas of this paper.

### 1.1. Motivation

Most recent approaches to concurrency controls uses locks to guarantee the correctness of data. However, there are a few drawbacks to lock based methods. For example, reading-only transactions that don’t affect the integrity of data still uses locking, or, to allow transactions to abort, we have to hold on to locks until the end of transactions. The authors believe that locking might only be needed in worst case which is relatively rare, so they came up with a concurrency control method that eliminates all need of locking.

### 1.2. Key Idea

Under the assumption that conflicts rarely happen, we remove the usage of locks and optimistically do all transactions. If no conflict happens during the transaction, we commit it, otherwise, retry the transaction. This method consists of three phases: read, validation and write phases (Figure 1.).

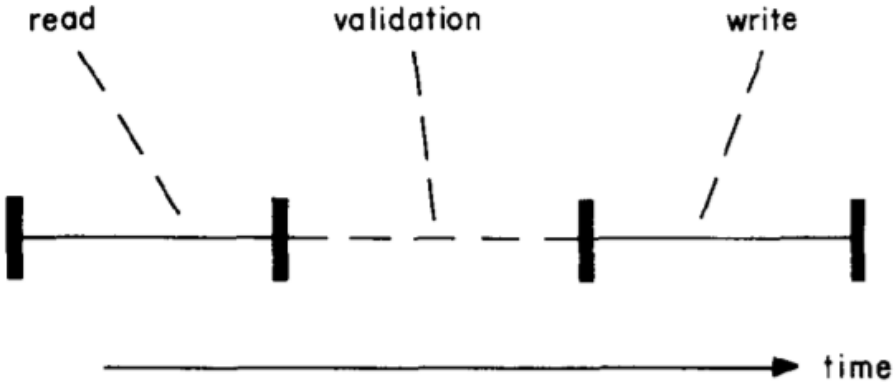



Figure 1. The three phases of optimistic concurrency control.

### 1.2.1. Read Phase

In the read phase, all the actions done by the transactions are done to a local copy of the database held by the transaction.

### 1.2.2. Validation Phase

During the validation phase, the transaction ensures consistency across all active transactions by checking if any write operations from other transactions were done to any data the current transaction is reading. The time span it checks starts from the starting of the current transaction up to the end of the reading phase.

### 1.2.3. Write Phase

Finally, after validation, all modified copies of the local database are written to the global database if it's valid.

# 2. Implementation Details

In this section, we will introduce how we implemented the optimistic concurrency control method. The read and write phases mentioned above is literally the same as shadow read and write in our last assignment with only minor changes, so we'll just go through the differences quickly. Then, we focus on the validation phase where most of the hard work was done and needs more explanation.

## 2.1. Read and Write Phases

After we merged the shadow read/write and in place write of assignment 5, we are already half done with the read and write phases of this method. The only changes that were made were:

(1) We added a HashSet, *readSet*, to the transaction class, so that it can record what instances were read by a transaction.
(2) We added a *putReadVal*() function which gets called whenever the function *getVal*() in *RecordFile*.java gets called, in order to record the read operations.
(3) The *certified*() function is now used when the transaction changes from its validation phase to its write phase.

## 2.2. Validation Phase

In the validation phase, we have to check if transactions conflict with each other and decide to commit or abort a transaction. There are two implemented versions of this phase, the serial and the parallel version. The difference between them is that the whole validation phase of the serial version is in a critical section. In contrast, only part of the validation phase is in the critical section for the parallel version, which increases the level of concurrency. We will explain both of them starting with the serial version which is more simple.

### 2.2.1. Serial Validation

A validation checks for the existence of conflicts so we'll have to learn what a conflict is first. We'll explain it with a small example. Assume that there are two active transactions, transaction A & B, at the same time. First, transaction A reads data block 1 first but hasn't committed yet. Then, transaction B comes to modify data block 1 (it can do so because there are no locks that avoid modification on data block 1) right after. Now, transaction A wants to commit but discovers that data block 1 isn't the same now, this is where a conflict occurs.

To check the occurrences of these conflicts, we first have to record the starting and ending time of the read phase for each transaction. They are recorded in the *startTn* and *finishTn* of each transaction respectively. The starting and ending times are actually represented by the global AtomicInteger tnc in the *OptimisticConcurrencyMgr* which increments every time a transaction is committed.

The *OptimisticConcurrencyMgr* was created to handle most of the work done during the validation phase. Other than the *tnc* used by every transaction, it also records all the committed transactions in the ArrayList *currentTxs*. It is used when we do the checking of other's write operations.

The validation phase is implemented in the *onTxCommit*() function of a transaction (C, for example). We first check if any of the transactions committed between *startTn* and *finishTn* of transaction C modifies data that is read by transaction C. We do so by searching if any element of the *writeSet* of transactions in *currentTxs* intersects with the elements in *readSet* of transaction C.

If the answer is negative, we *certify*() transaction C, which flushes the local writes to the global database, set the committed time of transaction C to *finalTn*, add transaction C to *currentTxs* and increment *tnc*.

On the other hand, if the answer is positive, we clean transaction C and abort it by throwing a *ValidationFaildException*.

Notice that everything done in the validation phase is in a single critical section.

### 2.2.2. Parallel Validation

In order to allow greater concurrency in the validation phase, we do a little more checking in exchange for a smaller critical section. The extra checking that we have to do to maintain the correctness of the database is that additional to the committed transactions, we also have to check if a transaction has a conflict with active transactions or not. Active transactions are transactions that have completed their read phase but not their write phase.

In order to do so, we first create the Set *active* to record the active transactions.

After we do the same checking done in the serial version, we check if any of the *readSet*/*writeSet* of the *active* transactions intersect with the *readSet* of the current transaction being checked. If all is well, we commit the transaction, otherwise, we abort it.

## 2.3. Subtle Details

Here we discuss some subtle implementation details that are worth mentioning.

### 2.3.1. The timing for the Addition of the Concurrency Manager

In a transaction, we need to add the listener of the *OptimisticConcurrencyMgr* first to ensure that other listeners are only called after the validation phase is passed.

### 2.3.2. Using While Loops to Redo Transactions

If a validation fails, a transaction is aborted by throwing a *ValidationFaildException*. So, in order to redo the transaction, we use while loops to redo a transaction if the exception was caught.

## 2.4. Test Cases

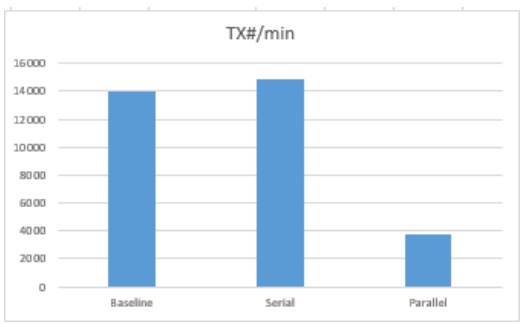
To make the test cases suitable for our scenario, we tweaked some code in the test cases to make them reasonable. The test cases we wanted to focus and test on were the ones related to concurrency control, so we chose the test cases in *fulltestsuite*. Among them were 7 test cases that are tested by causing *LockAbortException*. We removed them because we don't use locks at all, making them meaningless to test. Other than that, we added while loops for the redo of transactions just as we explained above.

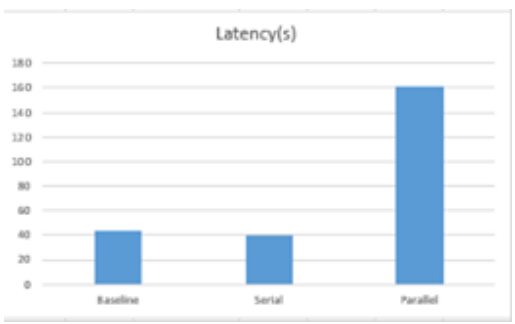
# 3. Experiments and Results

We run the provided benchmark with 30 seconds of warmup and a 1-minute benchmark time under different settings. The parameters that we tweak are RTE number, RW_TX_RATE, TOTAL_READ_COUNT, WRITE_RATIO_IN_RW_TX and HOT_CONFLICT RATE.

## 3.1. Most Optimistic Scenario

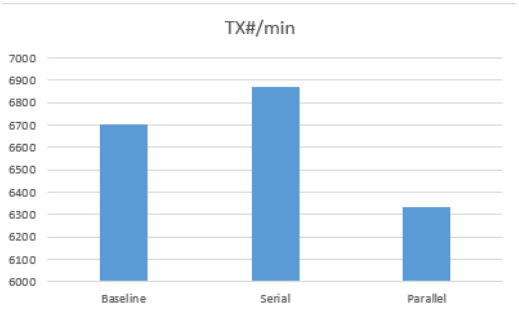
| Parameters | Value |
|---|---|
| RTE | 10 |
| RW_TX_RATE | 0.01 |
| TOTAL_READ_COUNT | 100 |
| WRITE_RATIO_IN_RW_TX | 0.1 |
| HOT_CONFLICT RATE | 0.0001 |

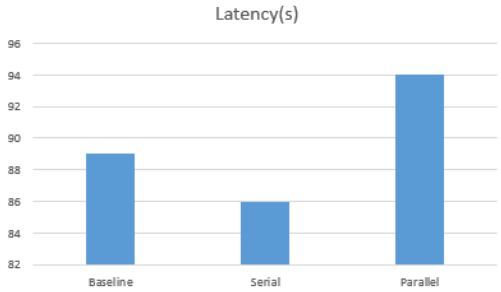


## 3.2. Other Setting

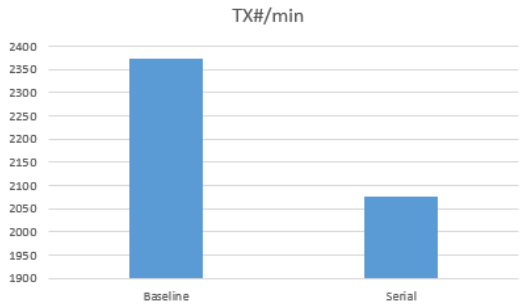
| Parameters | Value |
|---|---|
| RTE | 10 |
| RW_TX_RATE | 0.5 |
| TOTAL_READ_COUNT | 10 |
| WRITE_RATIO_IN_RW_TX | 0.5 |
| HOT_CONFLICT RATE | 0.001 |

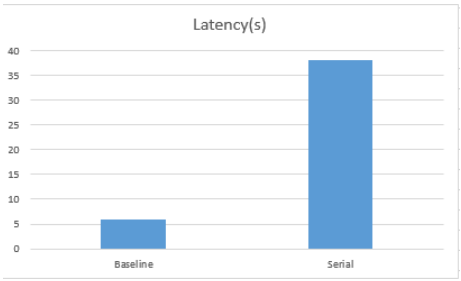



## 3.3. High conflict Rate

| Parameters | Value |
|---|---|
| RTE | 10 |
| RW_TX_RATE | 0.5 |
| TOTAL_READ_COUNT | 10 |
| WRITE_RATIO_IN_RW_TX | 0.5 |
| HOT_CONFLICT RATE | 0.001 |




## 3.4. Analysis

From the experiments shown above we can see that our there is a slight improvement in serial validation but huge decrease in performance in parallel validation. We guess that this happens because the workload of our testbed isn't as optimistic as the authors think. We might be able to get better performance if we mix a little bit of lock protocols into our concurrency control method (recommended by the paper).
On non-optimistic scenarios, optimistic concurrency control methods break completely.

### 3.5. Environment Settings

Intel Core i5-5200U CPU @ 2.20GHz 2.19GHz, 8GB Ram, 1TB HDD, OS Win10

## 4. Conclusion

We implemented a novel non-locking concurrency control method. The paper is easy to read and implement but it lacks details of experiments, In fact, it doesn't have anything about experiments. From our experiments above we can see that, although it might be useful in very very optimistic occasions, but real life scenarios might not be as optimistic as the authors think. Especially when there are regular patterns in access of database in real world situations.