Optimistic Concurrency Control

Arpit Gagneja, Tulika Thakur
July 20, 2016

University of Waterloo
1. Introduction

2. Optimistic Concurrency Control

3. Idea 1: MV3C

4. Idea 2: Tic Toc OCC

5. Idea 3: Reducing False Aborts

6. Performance on Silo

7. Conclusion and Future Work
Introduction
Starting with low overhead concurrency control

• Remember the idea of "single partition" transaction where the transaction access data on one partition and there is no overhead of coordinating with other transaction.

• This single idea helped in eliminating 42% of the CPU instructions caused by concurrency overheads like locks, latches and undo buffer maintenance.[1]
You might be wondering that is great. But where the problem lies?
Real life

- Many OLTP transactions access multiple partitions.
- This introduces network stalls so as to coordinate distributed transactions.
- In the low overhead concurrency paper, we discussed the idea of allowing the partition to work on other transactions.
Optimistic Concurrency Control
• Read Phase.
• Validation Phase -> aborts transactions to ensure serializability.
• Write Phase.
The idea of Speculative Execution

• Speculative Execution does more than 2PC by allowing other transactions to work on the partition instead of just blocking.[1]

• But the problem is all the parallel transactions going on have to wait for the first transaction to commit.

• And if the first transaction aborts, all the uncommitted transactions are "completely" rolled back from the queue of "uncommitted transactions" and gets back to the queue of "unexecuted transactions".
Main idea of our project

- There are performance bottlenecks while going for an idea to complete rollback all the queued transactions if the first transaction fails. So, in order to improve that, we delve ourselves on integrating other ideas like
  - Only execute those parts of transactions which failed to execute first place i.e. never execute transactions again from the scratch.
  - Adding a timestamp to data.
  - Reduce false aborts.
Now, we will discuss these ideas one by one. We will also discuss how each of this idea increases the performance on silos.
Idea 1: MV3C
Motivating examples behind MV3C

Figure 1: Cases where MV3C is more efficient in repairing the conflicting transactions compared to the “abort and restart” approach.
• Case (a): Observe logically disjoint program paths. MV3C detects that if only block A has some conflict with some parallel transaction. And suppose this parallel transaction commits before the original transaction. In this case, only block A will be re-executed, not block B.

• Case (b): Consider an example where some banking transaction fails at some line of code after doing a bunch of other queries. Discussed on the next slide.

• Case (c): The result set of initial data query is changed by detecting the conflict. For example, the SELECT statement gets one more record after some random concurrent update. In that case, SELECT statement is not executed again.
Case (b) explained and Concept of Predicates and Closure

```c
/* fm = from, acc = account and bal = balance */
TransferMoney(fm_acc, to_acc, amount) {
    START;

    // Predicate 1
    Account WHERE id=:fm_acc => fm_acc_entry
    IF(amount < 100) fee = 1.0;
    ELSE fee = amount * 0.01;

    IF(fm_acc_entry.bal > amount+fee) {
        fm_acc_entry.bal -= (amount + fee);
        fm_acc_entry.persist();

        // Predicate 2
        Account WHERE id=:to_acc => to_acc_entry
        to_acc_entry.bal += amount;
        to_acc_entry.persist();

        // Predicate 3
        Account WHERE id=:FEE_ACC_ID => fee_acc_entry
        fee_acc_entry.bal += fee;
        fee_acc_entry.persist();
        COMMIT;
    }
    ELSE ROLLBACK;
}
```
Idea 2 : Tic Toc OCC
Tic Toc OCC- Main Idea

• Algorithm overview -
  1. Assign read and write timestamps to data items[5].
  2. Compute a transaction’s timestamp lazily at its commit time in a distributed manner based on the tuples it accesses.

• Advantages -
  1. DBMS can exploit more parallelism in the workload, thereby reducing aborts and improving performance.
  2. Its distributed nature avoids all of the bottlenecks inherent in timestamp allocation making the algorithm highly scalable.
Consider the following two transactions-

1. A - read(x)
2. B - write(x)
3. B - commit
4. A write (y)
5. A commit

Figure : An example of two transactions executing using TicToc.
Idea 3: Reducing False Aborts
OCC - False Aborts

**Serializability** - Only transactions forming a cycle in their dependency graph cannot be serialized.

**False Abort** - When a transaction is aborted even though serializability condition is met.

![Transaction History Diagram]

**Serialization Graph:**

Figure: A schedule of the two transactions $T_1$ and $T_2$ (r:read, w:write, a:abort, c:commit) and the corresponding serialization graph. $T_1$ will be unnecessarily aborted by an OCC database.

[6]
• Increase in CPU core count = high data contention.
• High data contention = more aborts, more false aborts.
• More false aborts = reduces system throughput by 3.68x.
OCC - Reduce False Aborts

- Data Dependency- When transactions operate on same tuple and atleast one of the operations is write.
- Essential Dependency Pattern- the necessary condition that a transaction schedule is unserializable.
- Balance Overheads and false aborts- examine only 2 data dependencies.
- Snapshot transactions- all read-only transactions.
- Synchronization Point- Before beginning a new snapshot transaction, finish the active transactions first and take a new snapshot.
1. T3 is the first to commit.
2. T1 commits after T2 starts.
Performance on Silo
Silo has following main features -

- main memory database, works on multi-core machine.
- uses minimal-contention serializable commit protocol.
- track reads and writes in thread local storage.
- keeps track of anti-dependencies as well (write after read).
- uses the Masstree-inspired tree structure for indexes (fast concurrent B-tree).
- achieves roughly 700,000 transactions (OLTP) per second on the standard TPCC benchmark on a single 32-core machine, i.e. about 22,000 transactions per second per core[4].
Conclusion and Future Work
Conclusion

- Idea 4: using undo log as versions[3].
- Check performance improvement by each feature on Silo[4].
- Decide which feature is worth its overhead.
- Compare the results.
References


Questions?