Repairing Transaction Conflicts in Optimistic MVCC

Mohammad Dashti, Sachin Basil John, Amir Shaikhha and Christoph Koch

DATA Lab - EPFL

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE
Big Data

3 Vs of Big Data

- Volume
  - Terabytes
  - Records
  - Transactions
  - Tables, files

- Velocity
  - Batch
  - Near time
  - Real time
  - Streams

- Variety
  - Structured
  - Unstructured
  - Semistructured
  - All the above
What is a Transaction?

A *transaction* is a unit of program execution that accesses and possibly updates various data items.

E.g., transaction to transfer 50,000,000 IRR from account A to account B:
1. read\( (A) \)
2. \( A := A - 50000000 \)
3. write\( (A) \)
4. read\( (B) \)
5. \( B := B + 50000000 \)
6. write\( (B) \)

Two main issues to deal with:
- Failures of various kinds, such as hardware failures and system crashes
- Concurrent execution of multiple transactions
Concurrent Executions

- Multiple transactions are allowed to run concurrently in the system. Advantages are:
  - **Increased processor and disk utilization**, leading to better transaction *throughput*
  - E.g. one transaction can be using the CPU while another is reading from or writing to the disk
  - **Reduced average response time** for transactions: short transactions need not wait behind long ones.

What if something goes wrong?
ACID Properties

- When running transactions, to preserve the integrity of data the database system must ensure:

- **Atomicity.** Either all operations of the transaction are properly reflected in the database or none are.

- **Consistency.** Execution of a transaction in isolation preserves the consistency of the database.

- **Isolation.** Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.

- **Durability.** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.
Mechanisms to achieve isolation, atomicity and consistency

That is, to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database

Variants:
- Pessimistic vs. Optimistic
- Single-version vs. Multi-version
Each transaction can only see a snapshot of database
  Based on its start timestamp

All modifications are only visible to the transaction itself, before it commits

Read-only transactions never fail

A validation phase is required before a successful commit for R/W transactions
But …

- What if the validation phase fails?
- We have to abort and restart
  - The entire computation is thrown away
  - That’s really unfortunate

Can we do better?
If we have a partial roll back solution...

- When does it matter?
  - High-contention data objects
  - Long-running transactions
If we have a partial roll back solution…

What can we save?
If we have a partial roll back solution...

What can we save?

```c
/* fm = from, acc = account and bal = balance */

Transaction(acc, amount) {
    START;
    SELECT fm_bal INTO B;
    IF (amount >= 100) THEN {
        fm_bal_final = fm_bal - (amount + fee);
        fm_bal_final INTO C;
    }
    ELSE {
        fm_bal_final = fm_bal + amount;
    }
    UPDATE fm_bal_final WHERE id=:fm_acc;
    UPDATE to_bal_final WHERE id=:to_acc;
    UPDATE bal+fee WHERE id=:FEE_ACC_ID;
    COMMIT;
}

FROM Account WHERE id=:fm_acc;
 .0;
 ;
 {
    FROM Account WHERE id=:to_acc;
    (amount + fee);
    amount;
}
```
If we have a partial roll back solution...

- What are the requirements of such a solution?
  - Have minimal overhead
    - It’s paid for all transactions, even the ones that succeed validation
  - Quickly narrow down the conflicting portions of the transactions and fix them
    - Better be faster than “abort and restart”!
Validation in Optimistic MVCC

- Mainly two approaches:
  - Read-set validation + Phantom avoidance
  - Predicate-based\textsuperscript{[1]}
    - Uses a variation of precision locking\textsuperscript{[2]}


What are predicates?

```plaintext
/* fm = from, acc = account and bal = balance */
TransferMoney(fm_acc, to_acc, amount) {
  START;
  SELECT bal INTO :fm_bal FROM Account WHERE id=:fm_acc;
  IF(amount < 100) fee = 1.0;
  ELSE fee = amount * 0.01;
  IF(fm_bal > amount+fee) {
    SELECT bal INTO :to_bal FROM Account WHERE id=:to_acc;
    fm_bal_final = fm_bal - (amount + fee);
    to_bal_final = to_bal + amount;
    UPDATE Account SET bal=:fm_bal_final WHERE id=:fm_acc;
    UPDATE Account SET bal=:to_bal_final WHERE id=:to_acc;
    COMMIT;
  } ELSE ROLLBACK;
}
```
Validation in Optimistic MVCC – Predicate-based

- How can we use predicates?
  - These are the only pieces of information gathered about read operations during execution
  - The list of versions created by each transaction is also maintained, called “undo buffer”
  - Validation of $T$ consists of:

Predicates have a greater potential.
We can use them for partial rollback and compensate the failure with almost no overhead.
MV3C: MVCC with Closures

- Step 1: Write transaction programs in MV3C DSL to encode dependency info in them (predicates ↔ blocks of code)

- Step 2: Execute a transaction starting from its root predicates and create predicate graph

- Step 3: Validate in topological order

- Step 4: Repair.
  - If validation fails
    - Cleanup failed predicates
    - Retry with a new timestamp

```java
/* fm = from, acc = account and bal = balance */
TransferMoney(fm_acc, to_acc, amount) {
    START;
    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();
    }
    P1 Account WHERE id=:fm_acc=>fm_acc_entry
    P2 Account WHERE id=:to_acc=>to_acc_entry
    to_acc_entry.bal += amount;
    to_acc_entry.persist();
    P3 Account WHERE id=:FEE_ACC_ID=>fee_acc_entry
    fee_acc_entry.bal += fee;
    fee_acc_entry.persist();
    COMMIT;
    } ELSE ROLLBACK;
} 
```
Step 1: Writing MV3C Programs

- Translating T-SQL/PL-SQL into MV3C DSL
  - Identify all read operations as possible failure points
    - Encapsulate data selection criteria into the predicate

- Create dependency graph to identify blocks of code that depend on the result of some operation

- Partition dependency graph into nested sub graphs
  - Each subgraph is the minimal transitive closure of a sub graph with a read operation as root.
Step 2: Executing MV3C Programs

- Read-Write conflicts are detected at validation time.

- How to deal with Write-Write conflicts?
  - Optimistic MVCC has no chance of recovering from this case.
    - Among all concurrent write transactions into the same object, only one might succeed => serial execution
  - MV3C can only perform the conflicted section
    - It is an optional parameter than can be specified system-wide and table-wide, and can be overridden by each update operation
Step 3: Validation

- Validation is done in topological order.
- If a predicate fails validation, all its children predicates are marked as failed without performing validation on them.

/* fm = from, acc = account and bal = balance */
TransferMoney(fm_acc, to_acc, amount) {
   START;
   IF(amount < 100) fee = 1.0;
   ELSE fee = amount * 0.01;
   P1 Account WHERE id=:fm_acc => fm_acc_entry
   IF(fm_acc_entry.bal > amount+fee) {
      fm_acc_entry.bal -= (amount + fee);
      fm_acc_entry.persist();
   }
   P2 Account WHERE id=:to_acc => to_acc_entry
   to_acc_entry.bal += amount;
   to_acc_entry.persist();
   P3 Account WHERE id=:FEE_ACC_ID => fee_acc_entry
   fee_acc_entry.bal += fee;
   fee_acc_entry.persist();
   COMMIT;
} ELSE ROLLBACK;
Step 4: Repair

Tips:
- Removes all the versions created by failed predicates
- Executes the closures bound to top-level failed predicates

```java
/* fm = from, acc = account and bal = balance */
TransferMoney(fm_acc, to_acc, amount) {
    START;
    IF(amount < 100) fee = 1.0;
    ELSE fee = amount * 0.01;
    P1 Account WHERE id=:fm_acc => fm_acc_entry
    IF(fm_acc_entry.bal > amount+fee) {
        fm_acc_entry.bal -= (amount + fee);
        fm_acc_entry.persist();
    }
    P2 Account WHERE id=:to_acc => to_acc_entry
    to_acc_entry.bal += amount;
    to_acc_entry.persist();
    P3 Account WHERE id=:FEE_ACC_ID => fee_acc_entry
    fee_acc_entry.bal += fee;
    fee_acc_entry.persist();
    COMMIT;
} ELSE ROLLBACK;
```
Experimental Results
Banking: Impact of increased concurrency

Throughput (kilo tuples per second) vs # of worker threads for MV3C and OMVCC.
Banking: Impact of amount of conflict

- with 10 concurrent transactions
Banking: The cumulative Effect

![Graph showing cumulative effect over time]

- Time (s) on the y-axis
- Cumulative # of committed transactions on the x-axis
- Two lines representing MV3C and OMVCC
Trading Benchmark

- simulates a simplified trading system

- Tables:
  - Security(s_id, symbol, s_price)
  - Customer(c_id, cipher_key)
  - Trade(t_id, t_encrypted_data)
  - TradeLine(t_id, tl_id, tl_encrypted_data)

- Transactions:
  - TradeOrder
  - PriceUpdate
Trading: Impact of increased concurrency

Fixed $\alpha = 1.4$
Trading: Impact of amount of conflict

- with 10 concurrent transactions
Conclusion

- An efficient conflict resolution mechanism can have a considerable performance improvement for high-contention or long-running transactions.

- MV3C is an algorithm that can make use of compilation and program semantics for building an efficient conflict resolution mechanism for Optimistic MVCC.

- Performs better than Optimistic MVCC under higher contention

- Has almost no overhead compared to Optimistic MVCC