Morty

Scaling Concurrency Control with Re-Execution

Rahul Kushwaha, May 9, 2024

Morty: Introduction

- Storage system utilizing transaction re-execution to increase throughput of Serializable and Interactive Transactions.

Concurrency Control: Introduction

- Two categories
  - Optimistic Concurrency Control
    - Ex: TAPIR
  - Pessimistic Concurrency Control
    - Ex: Spanner

Optimistic Concurrency Control

- 3 Phases of a transaction: Read, Validation, and Write.

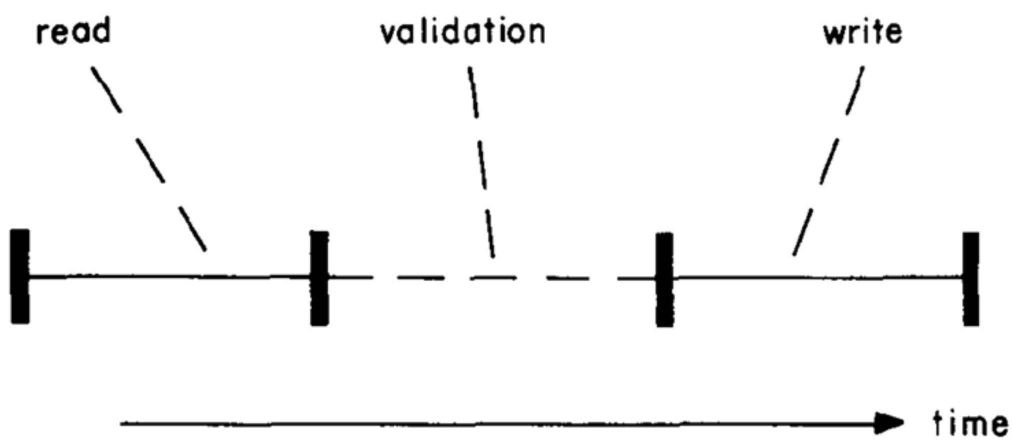


Fig. 1. The three phases of a transaction.

Img Src: [1]

Optimistic Concurrency Control

- Read: Concurrent, multiple transactions can be executing in parallel in this phase.
  - Any mutation to the data is kept within the context of txn.
  - Read your own writes.
- Validation: Serial, global critical section.
- Write: Serial, global critical section.

Optimistic Concurrency Control

- Suffers from high abort rates under contention.

Pessimistic Concurrency Control

- Utilizes locking schemes to prevent transactions reading or writing each others data.
- 2 Phase Locking
  - Growing Phase: Locks are acquired. No lock can be released in this phase.
  - Shrinking Phase: Locks are release. No lock can be acquired in this phase.
  - Needs methods to prevent or resolve deadlocks.

Pessimistic Concurrency Control

- Suffers from deadlocks and lock thrashing under contention.

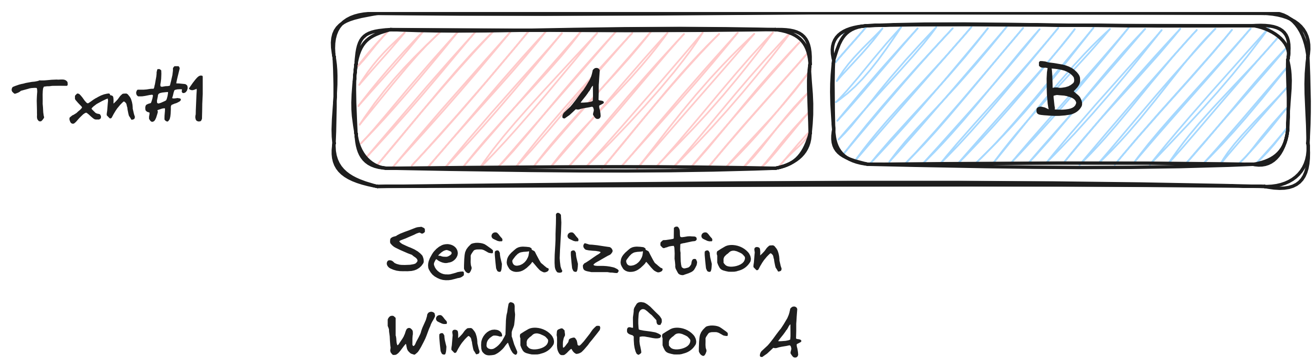
How to make progress?

- Retry the transaction with exponential backoff when faced with deadlock or abort.
  - Blind guessing how to space transactions.
    - Conservative guess: less progress due to contention.
    - Liberal guess: higher latencies.

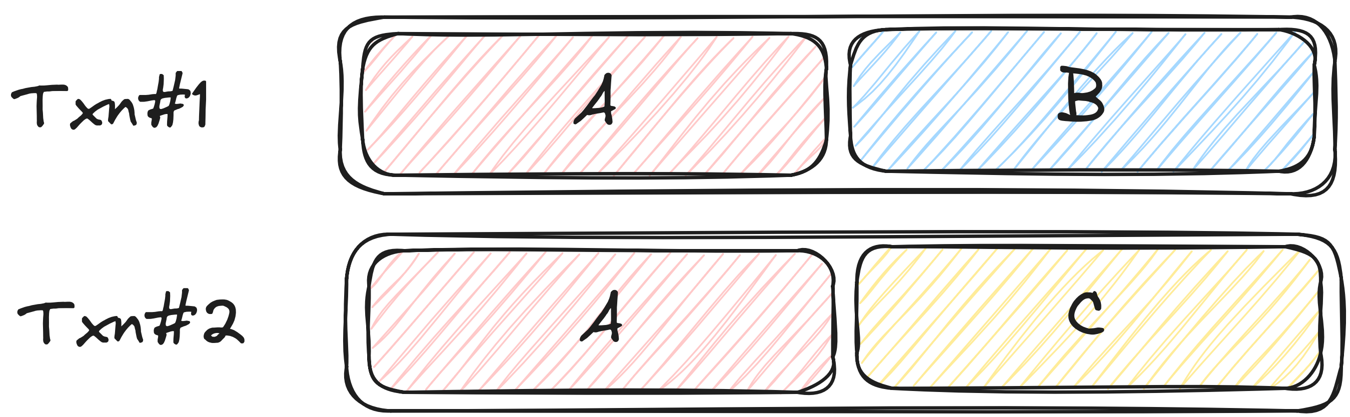
Morty: Serialization Windows

- Transactions reading and writing objects create Serialization Windows.
- Serialization Window for an object
  - Starts at the write of x whose value is being observed.
  - Ends when the transaction's write becomes visible.

Morty: Serialization Windows Example



Morty: Conflicting Serialization Windows



Morty: Idea

- Avoid conflicting serialization windows by re-arranging transactions.
- When such a re-arrangement takes place, some part/s of the txn being re-arranged need to be re-executed.
- During re-execution, Morty knows what needs to be re-executed rather than blindly restarting the txn.
- Claim: Re-Execution is better.

Morty: Transaction Re-Execution

- Imagine there are two transactions, T1 & T2.
- Serialization window of T1 & T2 overlaps.
- Resolve the overlap by:
  - Change the read-set of T2 using the write-set of T1.
  - Order becomes: T1 -> T2

Morty: Transaction Re-Execution

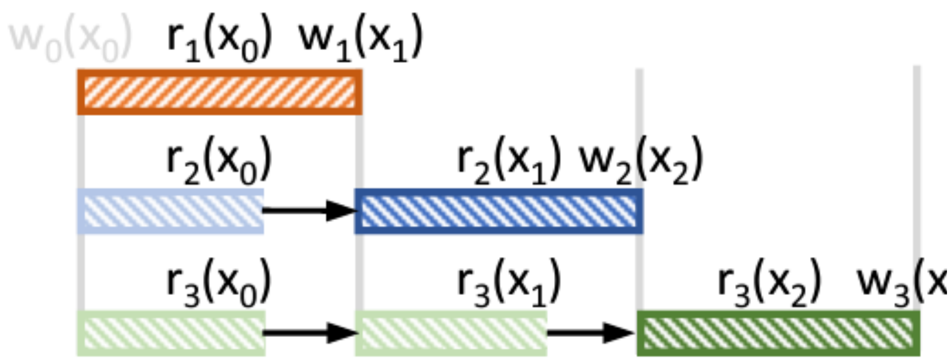


Figure 3. Transaction re-execution.

From the paper

Morty Design: Implementing Re-Execution

- Read Unrolling
  - Transaction Re-Execution moves transactions forward in time by invalidating read-set.
  - Need application side logic to undo the effects of previous reads.

Morty Design: Implementing Re-Execution

- Continuation-based API
  - Control flow is specified using function calls.

```
void ProcessPayment(uint w_id, uint amt,
    continuation_t cont) {
 auto ctx = make_ptr<PaymentCtx>();
 client.Begin(ctx);
 client.Get(move(ctx), "warehouses", w_id,
   [&client, &cont](ptr<PaymentCtx> ctx,
    string val){
    auto wh = ParseWarehouse(val);
    wh.SetCol("ytd", wh.GetCol("ytd") + amt);
    client.Put(ctx, "warehouses", w_id, wh);
    client.Commit(move(ctx), cont);
 });
}
```

(b) CPS: explicit continuations define control flow dependencies.

Morty Design: Transaction Execution

- Uses MVTSO. Timestamp determines the transactions’s position in total order.
- Integrates Concurrency Control with Replication.

Morty Design: Transaction Execution

- Begin(ctx)
  - Coordinator starts a transaction by assigning a unique verion = (ts, id). ts = local clock, id = coordinator id.
  - Ver defines the expected position in total order.

Morty Design: Transaction Execution

- GET(ctx, key, cont)
  - Coordinator sends the get, Get(ver, key), request to a nearby replica.
  - Replica replies with the key-value with the largest version smaller than ver.

Morty Design: Transaction Execution

- PUT(ctx, key, cont)
  - Coordinator adds (key, val) to write set.
  - Boardcast a Put(ver, key, val) to all replicas.
  - Replica checks for read-misses.
    - Replica would have replied with current (key, value) to a read already completed.
    - In such cases, Replica replies to Coordinator with GetReply(ver, val) to fix things.

Morty Design: Transaction Execution

- Re-Execution
  - GetReply triggers a re-execution of the transaction.

Morty Design: Transaction Execution

- COMMIT(ctx, cont)
  - Morty integrates concurrency control with Replication.

Morty Design: Transaction Execution

- Commit Result
  - Commit protocol has 2 outcomes: Commit or Abort
  - Morty can re-execute transactions. Therefore, each transaction has multiple executions.
  - Morty outcomes: Commit or Abandon
    - Commit: If at least one execution is successful.
    - Abort: If all executions are abandoned.

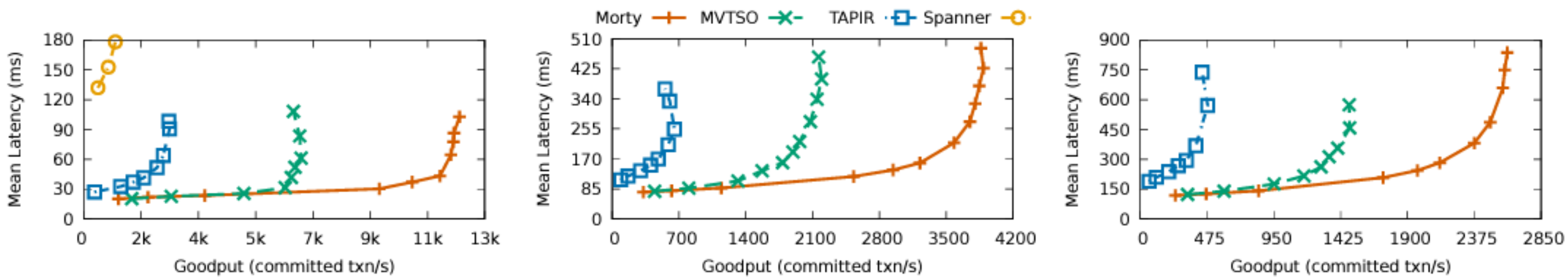
Morty Evaluation

- 3 Systems
  - TAPIR (OCC)
  - Spanner (PCC)(in house implementation)
  - Morty

Morty Evaluation

- 3 Setups: Simulated using Linux Traffic Control
  - REG: Replicas located in different availability zones of the same region.
  - CON: Replicas loacted in different regions.
  - GLO: Replicas in US and Europe.
  - RTT simulation numbers are measured using AWS.

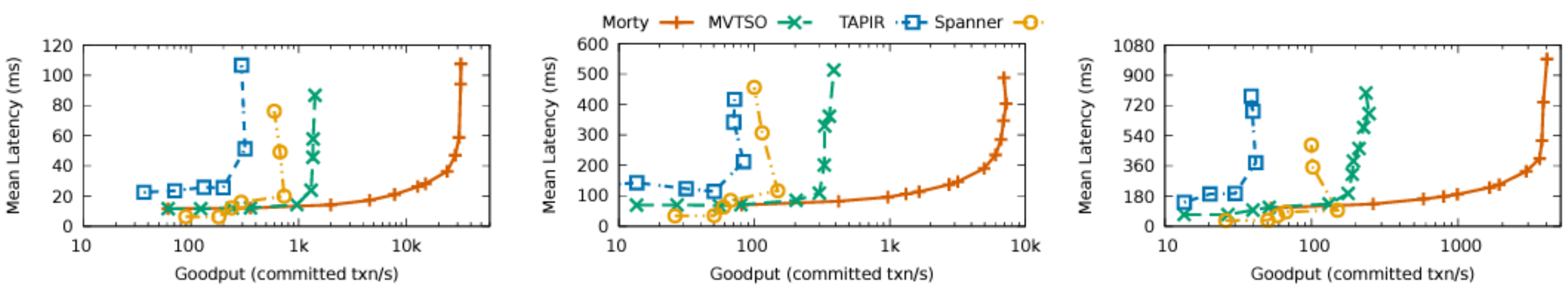
Morty Evaluation



(a) REG. **(b)** CON. **(c)** GLO.

Figure 6. Morty achieves higher goodput at saturation on TPC-C with 100 warehouses.

Morty Evaluation



(a) REG. **(b)** CON. **(c)** GLO.

Figure 7. Morty achieves higher throughput at saturation on Retwis with 10M keys and Zipf parameter 0.9.

Thank You!!