Rethinking serializable multiversion concurrency control

Jose M. Falerio & Daniel J. Abadi
Yale University

Presenter: Suyash Gupta, Purdue University

January 24, 2017

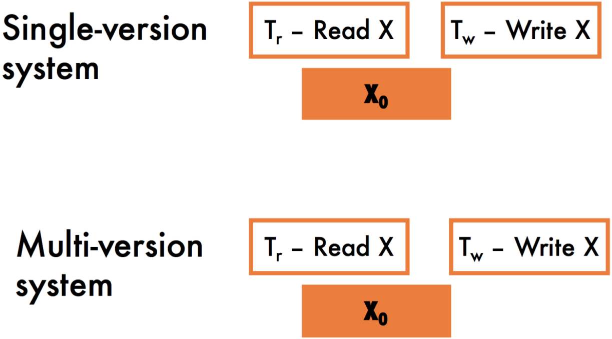
Food For Thought

Can we design a concurrency control protocol for multi-version database systems that is scalable?

Lets Refresh !!!

- Serializability
- Multiversion
- Timestamps
- Anti-dependency

Basics I: Single vs. Multi-version Systems



Basics I: Single vs. Multi-version Systems

Single-version system

T_r – Read X

T_w – Write X

X_0

T_r and T_w cannot simultaneously execute

Multi-version system

T_r – Read X

T_w – Write X

X_0

X_1

T_r and T_w can both simultaneously make progress

Basics I: Single vs. Multi-version Systems

Multi-versioning buys more concurrency! Right?

Basics II: Multi-versioning Example

```
T0:
if savings + checking >= 100
     savings -= 100
```

```
T1:
if savings + checking >= 75
     checking -= 75
```

Savings: 100

Checking: 50

Constraint:
savings + checking >= 0

Basics II: Multi-versioning Example

T_0 commits

T_0:

```
if savings + checking >= 100
     savings -= 100
```

T_1 commits

T_1:

```
if savings + checking >= 75
     checking -= 75
```

Savings: 100 → **Savings: 0**

Checking: 50 → **Checking: -25**

Constraint:
savings + checking >= 0

Basics II: Multi-versioning Example

Use of conservative isolation techniques, similar to single-version systems can avoid such inconsistencies.

Basics III: Multi-versioning Example II

T_0:

savings += 100

begin: | **end:**

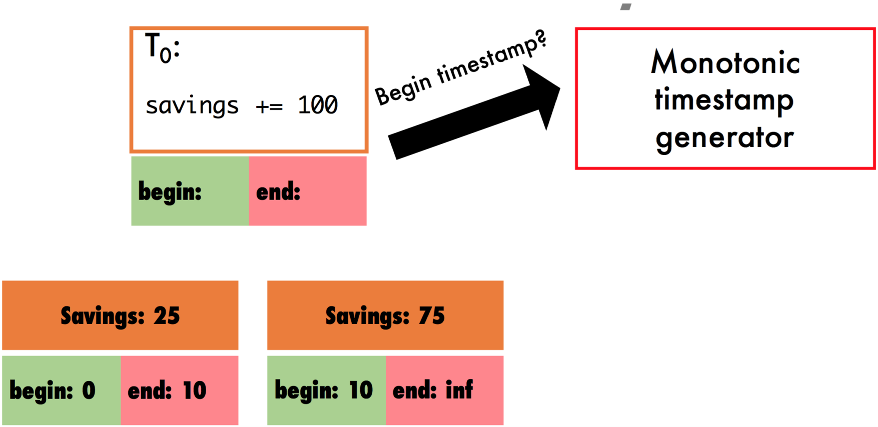
Monotonic timestamp generator

Savings: 25	
begin: 0	end: 10

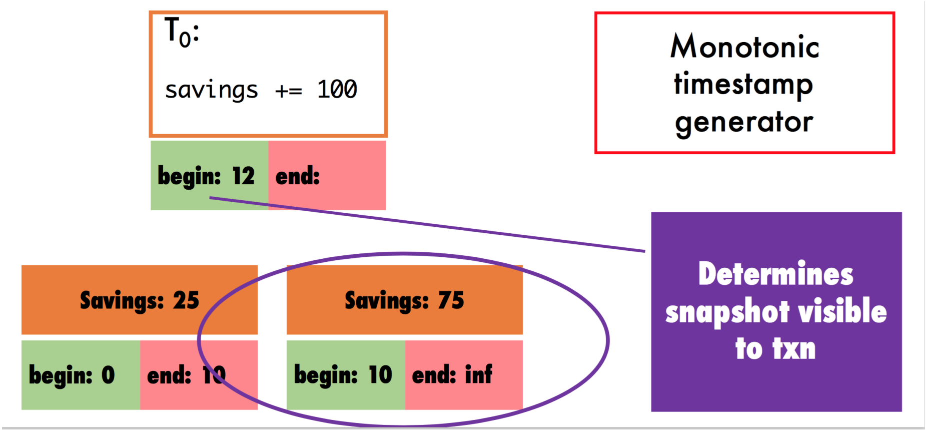
Savings: 75	
begin: 10	end: inf

PURDUE UNIVERSITY

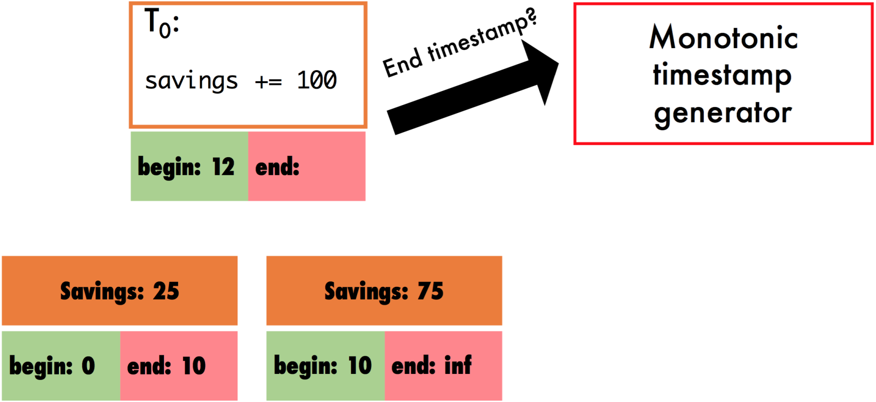
Basics III: Multi-versioning Example II



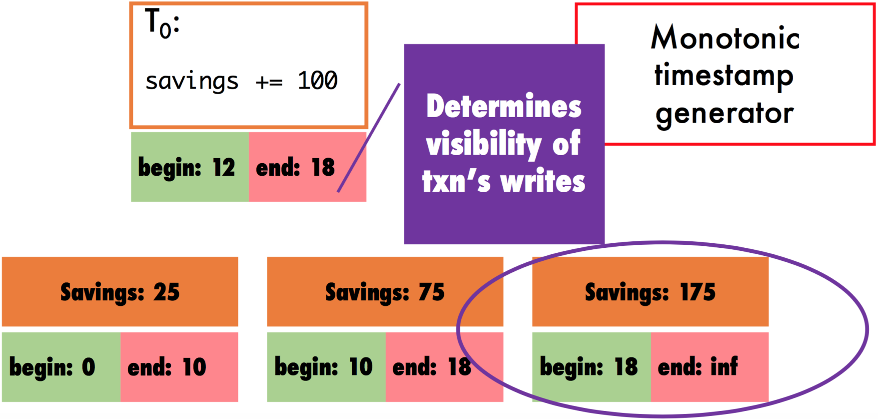
Basics III: Multi-versioning Example II



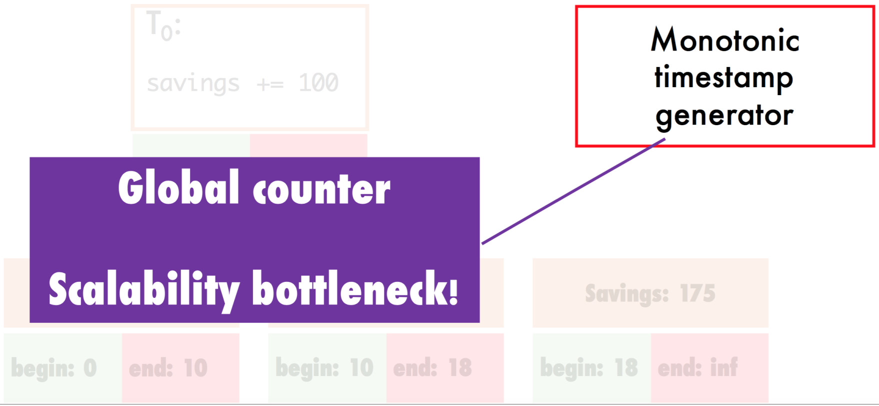
Basics III: Multi-versioning Example II



Basics III: Multi-versioning Example II



Motivation I – Non-Scalability



Motivation II – Management issues

Version management overhead

Savings: 25		Savings: 75		Savings: 175	
begin: 0	end: 10	begin: 10	end: 18	begin: 18	end: inf

Motivation III – Guaranteeing Serializability

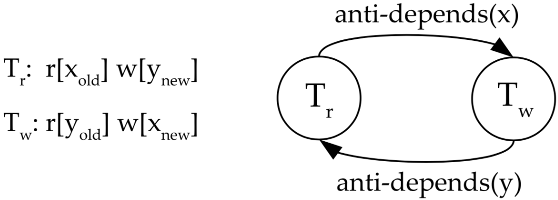


Figure 1: Non-serializable interleaving, and corresponding serialization graph of T_r and T_w. $r[x_1]$ denotes to a *read* of version 1 of record x, correspondingly, $w[x_1]$ denotes a *write* to record x, which produces version 1. A record's subscript corresponds to the version read or written by the transaction.

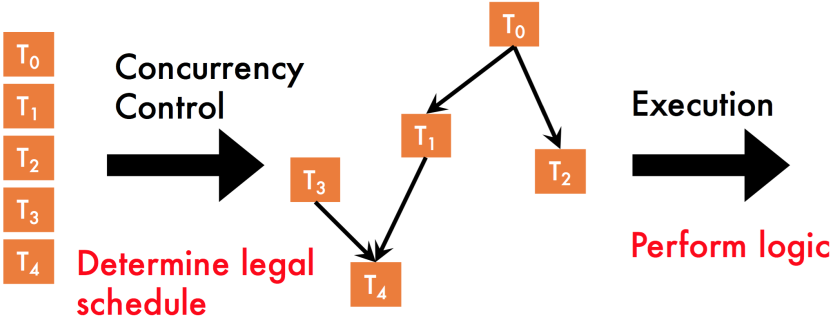
Bohm

- Separate concurrency control from transaction execution.
- Concurrency control determines transaction order and version visibility.
- Execution performs logic given concurrency control ordering

System at a Glance

- Transactions handed over to a single thread that maintains a log.
- Position of a transaction in the log is its timestamp.
- *m* concurrency threads, each owning a logical partition, analyze each transaction in the log, and create a space "placeholder" if the transaction writes to any record in its partition.
- *n* separate execution threads, each execute a batch of transactions, and fill the pre-allocated spaces (write operation).

BOHM Diagrammatically



Timestamp Assignment

- Prior Multi-version CC systems assign each transaction, two timestamps, t_{begin} and t_{end}.
  - t_{begin} – determines which versions of pre-existing records are visible.
  - t_{end} – determines the time at which writes become visible to other transactions.
- BOHM assigns each transaction a single timestamp, t_s.
- Each transaction appears to execute atomically at time ts.

Inserting Placeholders

- Several threads contribute to the processing of a single transaction's write-set.
- BOHM assigns the responsibility of each record to one concurrency control thread.
- When concurrency control layer receives a transaction, every concurrency control thread examines its write-set to determine whether any records belong to their partitions.

Version Field's

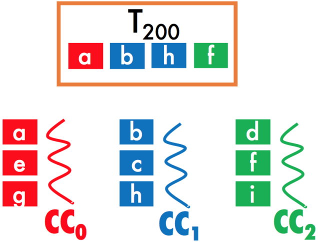
- Start timestamp set to the timestamp of the transaction that creates the version.
- End timestamp is set to infinity,
- Txn pointer is set to the transaction that creates the version.
- Data is left uninitialized.
- Prev pointer is set to the preceding version of the record.

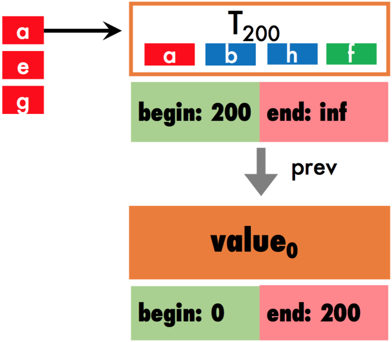
Batching

- Once a transaction has been processed by all threads it can be handed off to the transaction execution layer.
- One way – use synchronization barriers.
- BOHM avoids global coordination.
- Amortizes the cost of coordination across large batches of transactions.
- Each concurrency control thread receives an ordered batch of transactions

Concurrency Control Layer

- Partition data across multiple threads
- For every write, create a new version





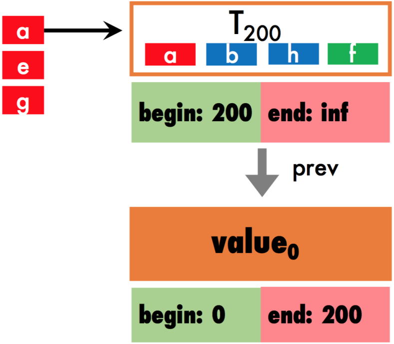
Evaluating Transaction Logic

- Computation of version data occurs in this phase.
- Read dependencies may cause a transaction to wait as data may not be available.
- Write dependencies allow parallel execution
- Read-Modify-Write may require similar wait as Read.
- Versions can be Garbage Collected, as transactions are ordered in batches.

Execution Layer

- Begins executing a batch after concurrency control completes

- Perform txn logic, write out data

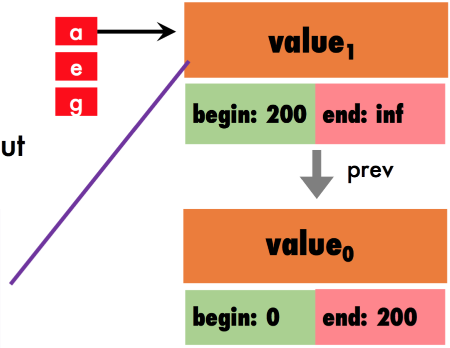


Execution Layer

- Begins executing a batch after concurrency control completes

- Perform txn logic, write out data

Replace txn reference with actual data



Future Thoughts

- BOHM's biggest disadvantage is its need to pre-determine the write-sets of the transaction, prior to its execution.
- Interesting thought can be to design an approach on similar lines for on-line or real-time systems, with obvious tradeoffs.
- Batching transactions entering at same instant.


THANK YOU!