Transaction

Xingda Wei

Nov. 10, 2020

Credits:http://news.cs.nyu.edu/~jinyang/fa10/schedule.html

Transaction: key pillar for online trading system

INTRODUCTION

Transaction(TX) is a user specified "program"

Transfer $100 from A to B

```
A_bal = READ(A)
If (A_bal >100)  {
   B_bal = READ(B)
   B_bal += 100
   A_bal  -= 100
   WRITE(A, A_bal)
   WRITE(B, B_bal)
}
```

Report sum of money

```
A_bal = READ(A)
B_bal = READ(B)
Print(A_bal+B_bal)
```

Transaction executions are concurrent

Transfer $100 from A to B

```
A_bal = READ(A)
If (A_bal >100)  {
   B_bal = READ(B)
   B_bal += 100
   A_bal  -= 100
   WRITE(A, A_bal)
   WRITE(B, B_bal)
}
```

Report sum of money

```
A_bal = READ(A)
B_bal = READ(B)
Print(A_bal+B_bal)
```

Storage system

Executed from different threads,
even different machines (distributed).

Why concurrency ? To achieve high performance

- Moore’ s law is dead, single core cannot become faster
- Scaling database to many machines
  - For bigger capacity & better fault tolerance

12306

9.56 million tickets/day

Alibaba.com

$9.3 billion/day

Wildly execute transactions causes inconsistency

Transfer $100 from A to B

```
A_bal = READ(A)
If (A_bal >=100)  {
   B_bal = READ(B)
   B_bal += 100
   A_bal  -= 100
   WRITE(A, A_bal)
   WRITE(B, B_bal)
}
```

Report sum of money

```
A_bal = READ(A)
B_bal = READ(B)
Print(A_bal+B_bal)
```

A = 100
B = 100

Transfer B_bal += 100

Timeline

Report

Wildly execute transactions causes inconsistency

Transfer $100 from A to B

```
A_bal = READ(A)
If (A_bal >=100) {
   B_bal = READ(B)
   B_bal += 100
   A_bal -= 100
   WRITE(A, A_bal)
   WRITE(B, B_bal)
}
```

Report sum of money

```
A_bal = READ(A)
B_bal = READ(B)
Print(A_bal+B_bal)
```

A = 100
B = 100

Transfer B_bal += 100

Timeline

Report B_bal = 200

Wildly execute transactions causes inconsistency

Transfer $100 from A to B

```
A_bal = READ(A)
If (A_bal >=100)  {
   B_bal = READ(B)
   B_bal += 100
   A_bal  -= 100
   WRITE(A, A_bal)
   WRITE(B, B_bal)
}
```

Report sum of money

```
A_bal = READ(A)
B_bal = READ(B)
Print(A_bal+B_bal)
```

A = 100
B = 100

Transfer B_bal += 100

Timeline

Report B_bal = 200 A_bal = 100

Wildly execute transactions causes inconsistency

Transfer $100 from A to B

```
A_bal = READ(A)
If (A_bal >=100)  {
   B_bal = READ(B)
   B_bal += 100
   A_bal  -= 100
   WRITE(A, A_bal)
   WRITE(B, B_bal)
}
```

Report sum of money

```
A_bal = READ(A)
B_bal = READ(B)
Print(A_bal+B_bal)
```

A = 100
B = 100

Transfer B_bal += 100 A_bal -= 100

Timeline

Report B_bal = 200 A_bal = 100

Solutions?

1. **Leave it to application programmers**
   - Application programmers are responsible for locking data items
2. **System performs automatic concurrency control**
   - Concurrent transactions execute as if serially

Solutions?

1. **Leave it to application programmers**
   – Application programmers are responsible for locking data items
2. **System performs automatic concurrency control**
   – Concurrent transactions execute as if seriall
   – Even when the environment is concurrent

System guarantees transaction is ACID

- **A (Atomicity)**
  - All-or-nothing w.r.t. failures
- **C (Consistency)**
  - Transactions maintain any internal storage state invariants
- **I (Isolation)**
  - Concurrently executing transactions do not interfere
- **D (Durability)**
  - Effect of transactions survive failures

What is ACID in details ?

T1: Transfer $100 from A to B

```
A_bal = READ(A)
If (A_bal >100)  {
   B_bal = READ(B)
   B_bal += 100
   A_bal  -= 100
   WRITE(A, A_bal)
   WRITE(B, B_bal)
}
```

T2: Report sum of money

```
A_bal = READ(A)
B_bal = READ(B)
Print(A_bal+B_bal)
```

- **Atomicity: T1 completes or nothing.**
  - E.g., If B += 100 => A -= 100

What is ACID in details ?

T1: Transfer $100 from A to B

```
A_bal = READ(A)
If (A_bal >100)  {
   B_bal = READ(B)
   B_bal += 100
   A_bal  -= 100
   WRITE(A, A_bal)
   WRITE(B, B_bal)
}
```

T2: Report sum of money

```
A_bal = READ(A)
B_bal = READ(B)
Print(A_bal+B_bal)
```

- **Consistency: guarantees application semantics**
  - E.g.., (A_bal + B_bal) is not changed

What is ACID in details ?

T1: Transfer $100 from A to B

```
A_bal = READ(A)
If (A_bal >100)  {
   B_bal = READ(B)
   B_bal += 100
   A_bal  -= 100
   WRITE(A, A_bal)
   WRITE(B, B_bal)
}
```

T2: Report sum of money

```
A_bal = READ(A)
B_bal = READ(B)
Print(A_bal+B_bal)
```

- **Isolation: T1 & T2 isolated from each other**
  - E.g., T2 does not sees T1’ s intermediate result

What is ACID in details ?

T1: Transfer $100 from A to B

```
A_bal = READ(A)
If (A_bal >100)  {
   B_bal = READ(B)
   B_bal += 100
   A_bal  -= 100
   WRITE(A, A_bal)
   WRITE(B, B_bal)
}
```

T2: Report sum of money

```
A_bal = READ(A)
B_bal = READ(B)
Print(A_bal+B_bal)
```

- **Durability: storage survives compute failures**
  - E.g., changes of T1 persists after client/server failures

System guarantees transaction is ACID

- **A (Atomicity)**
  - All-or-nothing w.r.t. failures
- **C (Consistency)**
  - Transactions maintain any internal storage state invariants
- **I (Isolation)**
  - Concurrently executing transactions do not interfere
- **D (Durability)**
  - Effect of transactions survive failures

Ideal isolation semantic: serializability

- **Definition: execution of a set of transactions is equivalent to <u>some</u> serial order**
  - Two executions are *equivalent* if they have the same effect on database and produce same output.

Why serializability?

- **Suppose each TX transfers system from a consistent state to another consistent state**

  CS -> TX -> CS

Then the final state of executing all TXs is consistent

  CS -> TX_1 -> TX_2 … TX_n -> CS

Examples

```
A_bal = READ(A)
B_bal = READ(B)
B_bal += 100
A_bal  -= 100
WRITE(A, A_bal)
WRITE(B, B_bal)
```

```
A_bal = READ(A)
B_bal = READ(B)
Print(A_bal+B_bal)
```

Serializable? R(A),R(B),R(A),R(B),C W(A),W(B),C

Yes, equivalent serial schedule: R(A),R(B),C,R(A),R(B),W(A),W(B),C

Serializable? R(A),R(B), W(A), R(A),R(B),C, W(B),C

Realize serializability

- **Using the standard technique in concurrent programming**
  - Locking-based approach
- **Strawman solution 1:**
  - Grab global lock before transaction starts
  - Release global lock after transaction commits
- **Strawman solution 2:**
  - Grab lock on item X before reading/writing X
  - Release lock on X after reading/writing X

Strawman 2

```
A_bal = READ(A)
B_bal = READ(B)
B_bal += 100
A_bal  -= 100
WRITE(A, A_bal)
WRITE(B, B_bal)
```

```
A_bal = READ(A)
B_bal = READ(B)
Print(A_bal+B_bal)
```

Possible with strawman 2? (short-duration locks)

R(A),R(B), W(A), R(A),R(B),C, W(B),C

Strawman 2

```
A_bal = READ(A)
B_bal = READ(B)
B_bal += 100
A_bal  -= 100
WRITE(A, A_bal)
WRITE(B, B_bal)
```

```
A_bal = READ(A)
B_bal = READ(B)
Print(A_bal+B_bal)
```

Possible with strawman 2? (short-duration locks)

R(A),R(B), W(A), R(A),R(B),C, W(B),C

Read an uncommitted value

Strawman 2

```
A_bal = READ(A)
B_bal = READ(B)
B_bal += 100
A_bal  -= 100
WRITE(A, A_bal)
WRITE(B, B_bal)
```

```
A_bal = READ(A)
B_bal = READ(B)
Print(A_bal+B_bal)
```

Possible with strawman 2? (short-duration locks)

R(A),R(B), W(A), R(A),R(B),C, W(B),C

Locks on writes should be held till end of transaction

Read an uncommitted value

More Strawman

- **Strawman 3**
  - Grab lock on item X before read/writing X
  - Release locks at the end of transaction

Strawman 2

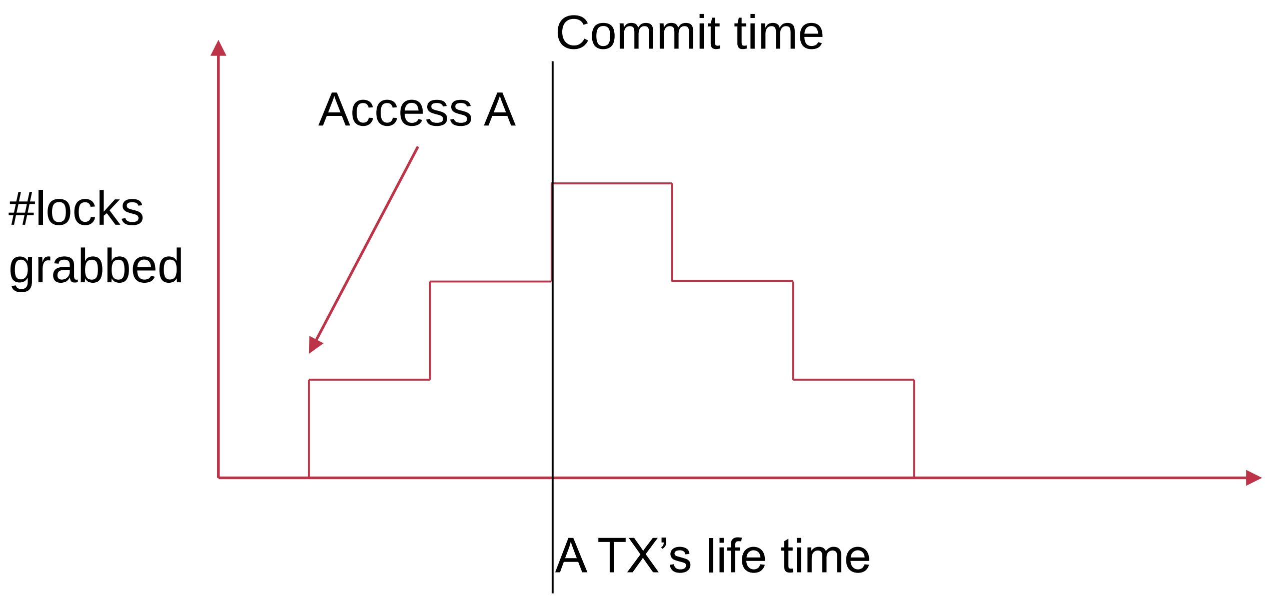
```
lock(A)
A_bal = READ(A)
lock(B)
B_bal = READ(B)
B_bal += 100
A_bal  -= 100
WRITE(A, A_bal)
WRITE(B, B_bal)
unlock(A)
unlock(B)
```

```
lock(A)
A_bal = READ(A)
lock(B)
B_bal = READ(B)
Print(A_bal+B_bal)
unlock(A)
unlock(B)
```

Possible? R(A),R(B), W(A), R(A),R(B),C W(B),C

Strawman 3 is also called 2PL

- **2 phase locking (2PL) guarantees serializability**
  - A growing phase in which the transaction is acquiring locks
  - A shrinking phase in which locks are released



2PL: deadlocks?

```
lock(A)
A_bal = READ(A)
lock(B)
B_bal = READ(B)
B_bal += 100
A_bal  -= 100
WRITE(A, A_bal)
WRITE(B, B_bal)
unlock(A)
unlock(B)
```

```
lock(B)
B_bal = READ(B)
lock(A)
A_bal = READ(A)
Print(A_bal+B_bal)
unlock(A)
unlock(B)
```

2PL: deadlocks?

L(A)

Timeline

```
lock(A)
A_bal = READ(A)
lock(B)
B_bal = READ(B)
B_bal += 100
A_bal  -= 100
WRITE(A, A_bal)
WRITE(B, B_bal)
unlock(A)
unlock(B)
```

```
lock(B)
B_bal = READ(B)
lock(A)
A_bal = READ(A)
Print(A_bal+B_bal)
unlock(A)
unlock(B)
```

2PL: deadlocks?

```
lock(A)
A_bal = READ(A)
lock(B)
B_bal = READ(B)
B_bal += 100
A_bal  -= 100
WRITE(A, A_bal)
WRITE(B, B_bal)
unlock(A)
unlock(B)
```

```
lock(B)
B_bal = READ(B)
lock(A)
A_bal = READ(A)
Print(A_bal+B_bal)
unlock(A)
unlock(B)
```

L(A)

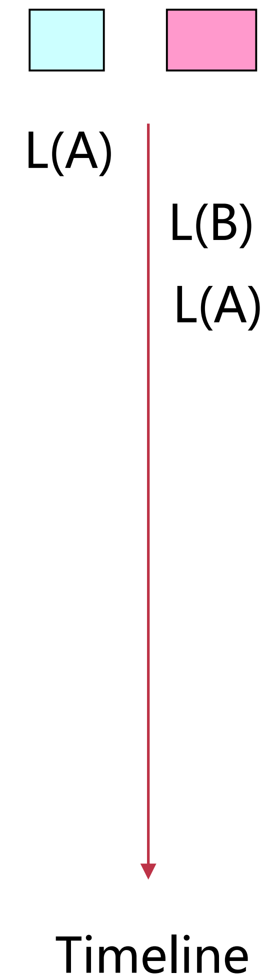
L(B)

Timeline

2PL: deadlocks?

```
lock(A)
A_bal = READ(A)
lock(B)
B_bal = READ(B)
B_bal += 100
A_bal  -= 100
WRITE(A, A_bal)
WRITE(B, B_bal)
unlock(A)
unlock(B)
```

```
lock(B)
B_bal = READ(B)
lock(A)
A_bal = READ(A)
Print(A_bal+B_bal)
unlock(A)
unlock(B)
```



2PL: deadlocks?

```
lock(A)
A_bal = READ(A)
lock(B)
B_bal = READ(B)
B_bal += 100
A_bal  -= 100
WRITE(A, A_bal)
WRITE(B, B_bal)
unlock(A)
unlock(B)
```

```
lock(B)
B_bal = READ(B)
lock(A)
A_bal = READ(A)
Print(A_bal+B_bal)
unlock(A)
unlock(B)
```

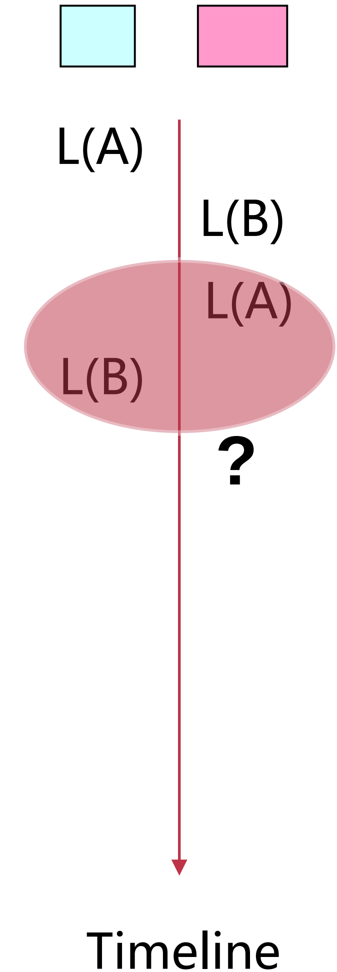
L(A)	
	L(B)
	L(A)
L(B)	

Timeline

2PL: deadlocks?

```
lock(A)
A_bal = READ(A)
lock(B)
B_bal = READ(B)
B_bal += 100
A_bal  -= 100
WRITE(A, A_bal)
WRITE(B, B_bal)
unlock(A)
unlock(B)
```

```
lock(B)
B_bal = READ(B)
lock(A)
A_bal = READ(A)
Print(A_bal+B_bal)
unlock(A)
unlock(B)
```



Disadvantages of locking-based approach

- **Need to detect deadlocks**
  - Distributed implementation needs distributed deadlock detection (bad)
- **Read-only transactions can block update transactions**
  - Big performance hit if there are long running read-only transactions

Optimistic concurrency control

Reads do not acquire locks

- **Use validation-based scheme**

No deadlock

- **Use speculative execution to ensure a deterministic locking order**

Optimistic concurrency control

- **OCC execute transaction speculatively**
  - Execution phase (execute TX with (near no) concurrency control)
  - Validation phase (verify the result of execution phase)
  - Commit phase

- **Only lock the write-set in the validation phase**

OCC: an example

Original transaction

```
A_bal = READ(A)
B_bal = READ(B)
B_bal += 100
A_bal  -= 100
WRITE(A, A_bal)
WRITE(B, B_bal)
```

OCC: an example

Executed using OCC

```
Cache[A_bal] = READ(A)
Cache[B_bal] = READ(B)
Cache[B_bal] += 100
Cache[A_bal]  -= 100
```

Execution phase

Original transaction

```
A_bal = READ(A)
B_bal = READ(B)
B_bal += 100
A_bal  -= 100
WRITE(A, A_bal)
WRITE(B, B_bal)
```

OCC: an example

Original transaction

```
A_bal = READ(A)
B_bal = READ(B)
B_bal += 100
A_bal  -= 100
WRITE(A, A_bal)
WRITE(B, B_bal)
```

Executed using OCC

```
Cache[A_bal] = READ(A)
Cache[B_bal] = READ(B)
Cache[B_bal] += 100
Cache[A_bal]  -= 100
```

Execution phase

```
Lock(A)
Lock(B)
Abort if A_bal or B_bal has changed
```

Validation phase

OCC: an example

Original transaction

```
A_bal = READ(A)
B_bal = READ(B)
B_bal += 100
A_bal  -= 100
WRITE(A, A_bal)
WRITE(B, B_bal)
```

Executed using OCC

```
Cache[A_bal] = READ(A)
Cache[B_bal] = READ(B)
Cache[B_bal] += 100
Cache[A_bal]  -= 100
```

Execution phase

```
Lock(A)
Lock(B)
Abort if A_bal or B_bal has changed
```

Validation phase

```
A_bal = Cache[A_bal]
B_bal = Cache[B_bal]
// unlock A and B
```

Commit phase

OCC: an example

- **Why no deadlock?**
  - after execution phase,
    We explore all the
    Involved data
- **Thus, we can assign**
a deterministic lock order

Executed using OCC

```
Cache[A_bal] = READ(A)
Cache[B_bal] = READ(B)
Cache[B_bal] += 100
Cache[A_bal]  -= 100
```

Execution phase

```
Lock(A)
Lock(B)
Abort if A_bal or B_bal has changed
```

Validation phase

```
A_bal = Cache[A_bal]
B_bal = Cache[B_bal]
// unlock A and B
```

Commit phase

OCC: an example

Original transaction

A_bal = READ(A)
B_bal = READ(B)
Print(A_bal+B_bal)

Executed using OCC

Cache[A_bal] = READ(A)
Cache[B_bal] = READ(B)

Execution phase

Abort if A_bal or B_bal changed

Validation phase

Rrint(cache[A_bal] + cache[B_bal])

Commit phase

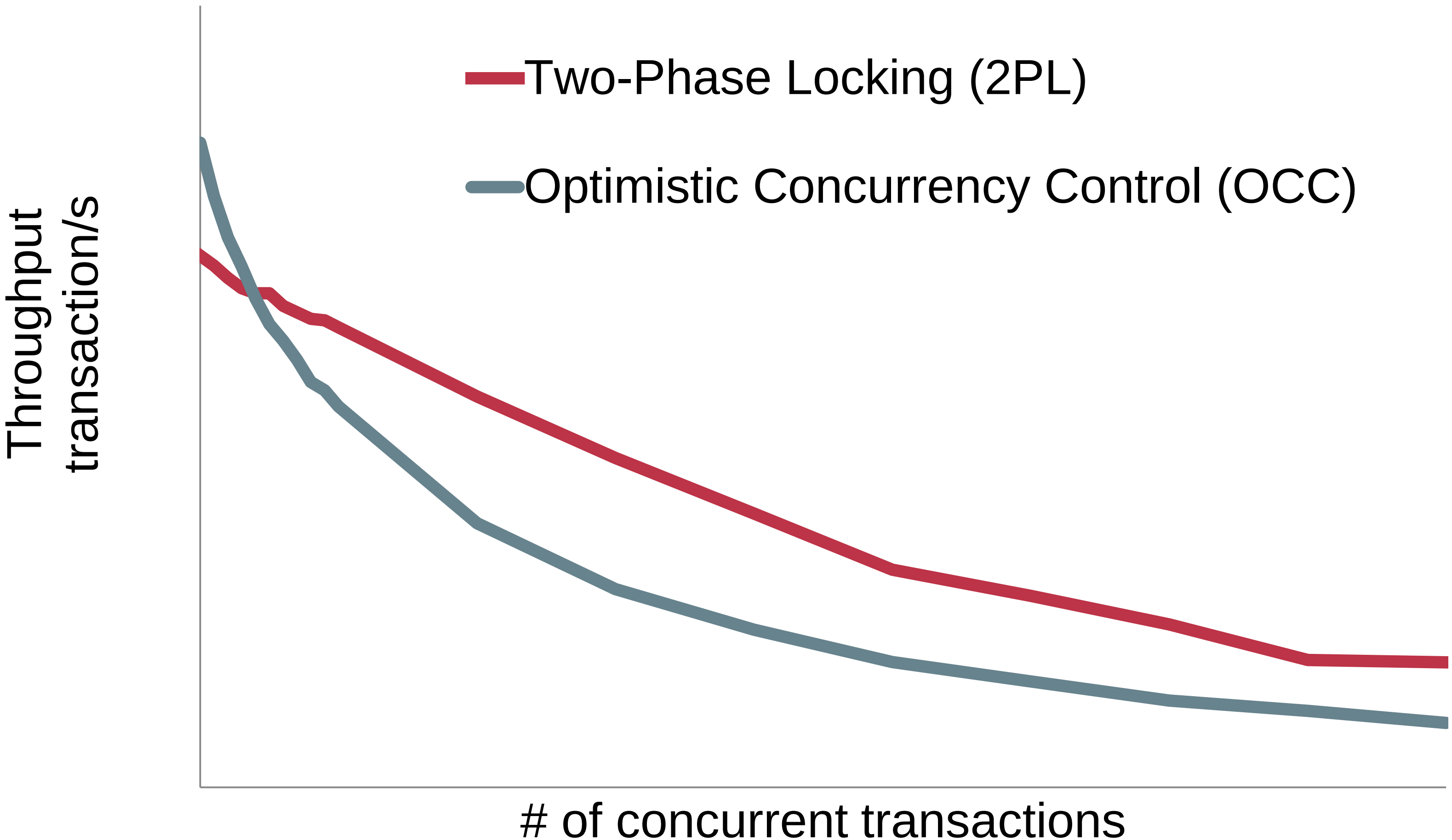
OCC: how to handle aborts?

- **Only retry the execution**
  - Excessive retries cause low performances
- **Cause live lock if there are many contentions !**

Recap

- **We have talked about transaction’s ACID property**
- **We have presented two classic protocols to ensure (ACI) transaction**
- **Most protocols nowadays are those two protocol’s variations !**

However, Serializability is Costly under Contention



Recap: why 2PL & OCC fails under contention ?

- **2PL**
  - Contented transactions block each other
  - Read-only transactions block writers
  - Dead lock (hard in a distributed setting)

- **OCC**
  - Contented transaction frequently retries

Observations

- **Read-only transaction dominates (real) workloads**
  - e.g., 99.8% requests to Facebook TAO are reads[1]
- **Read-only transaction are long-running[2]**
  - Blocking or retry are both unaccepted

[1] TAO [ATC'13]

[2] Challenges to adopting stronger consistency at scale. [HOTOS'15]

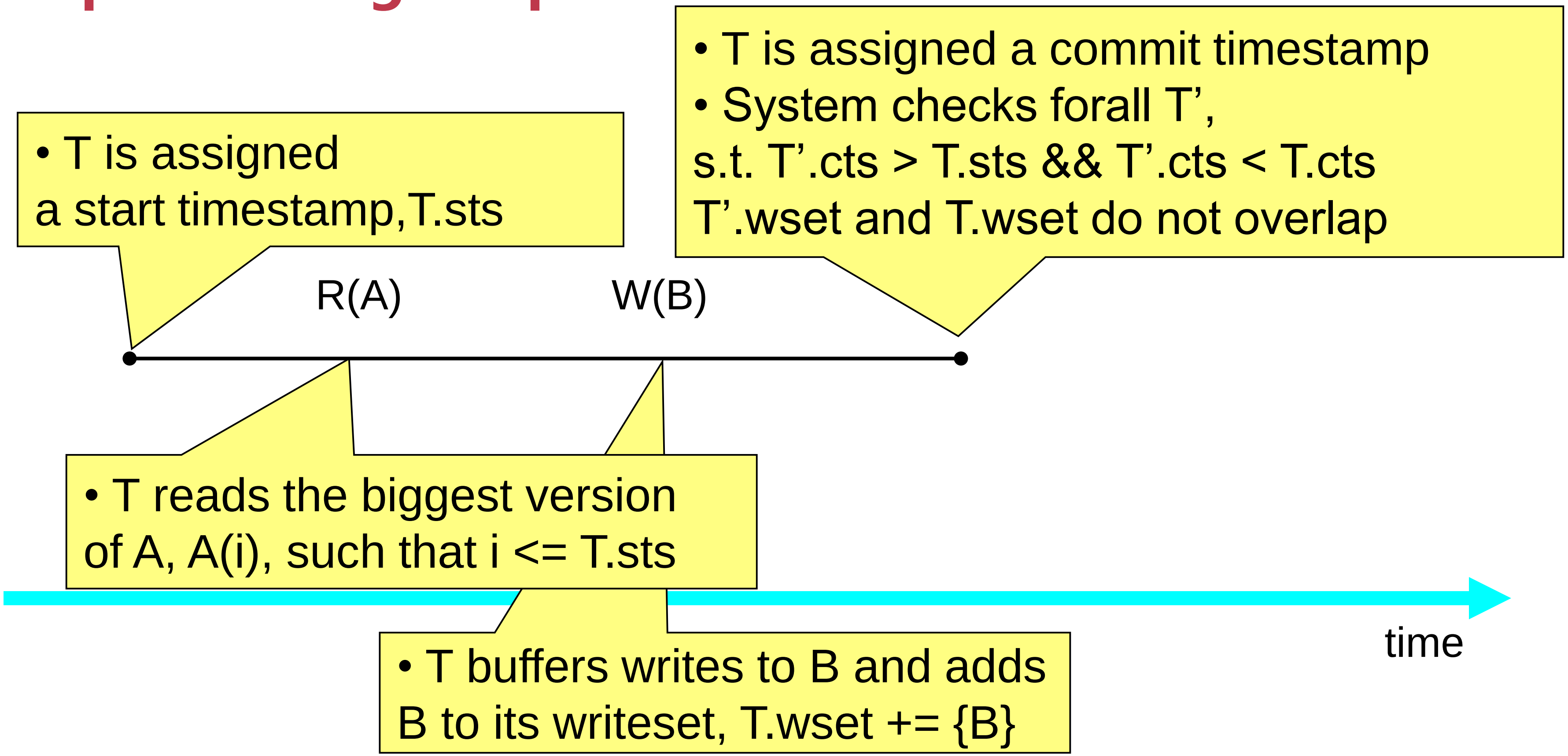
Multi-version protocols

- **Each data item is associated with multiple versions**
- **Multi-version transactions:**
  - Reads choose the appropriate version from a consistent view (no locking and retry!)

Snapshot Isolation (not serializable)

- **A popular multi-version concurrency control scheme**
- **A transaction:**
  - reads a "snapshot" of database image
  - Can commit only if there are no write-write conflict

Implementing Snapshot Isolation



Snapshot Isolation: a concrete example

A_bal = READ(A)
B_bal = READ(B)
B_bal += 100
A_bal -= 100
WRITE(A, A_bal)
WRITE(B, B_bal)

A0 = 100
B0 = 100

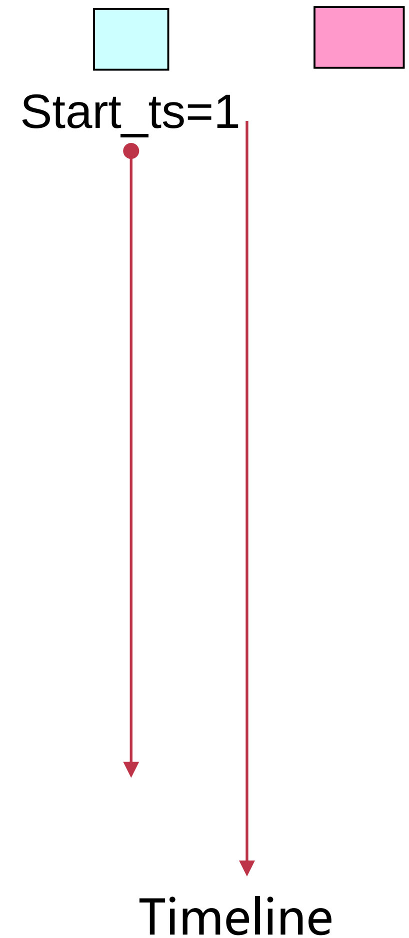
A_bal = READ(A)
B_bal = READ(B)
Print(A_bal+B_bal)

Timeline

Snapshot Isolation: a concrete example

A_bal = READ(A)
B_bal = READ(B)
B_bal += 100
A_bal -= 100
WRITE(A, A_bal)
WRITE(B, B_bal)

A0 = 100
B0 = 100



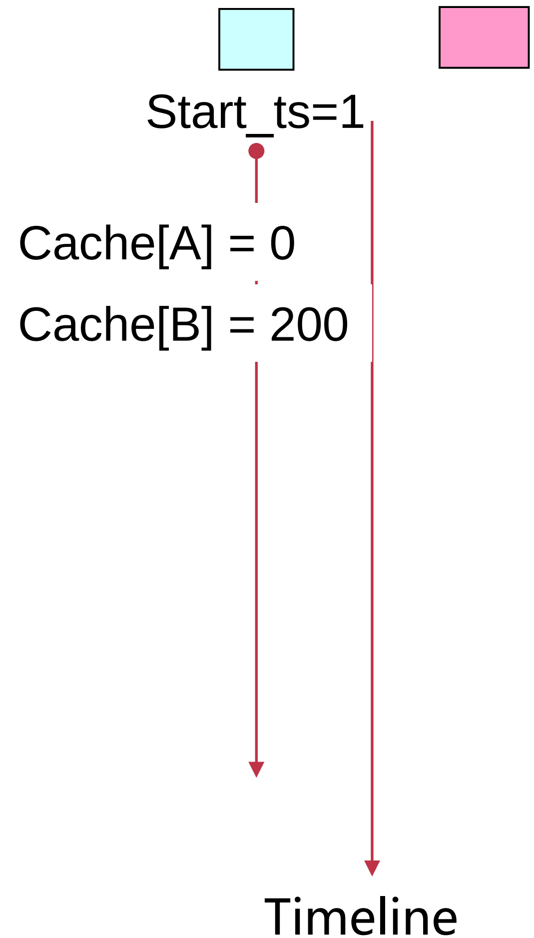
A_bal = READ(A)
B_bal = READ(B)
Print(A_bal+B_bal)

Snapshot Isolation: a concrete example

```
A_bal = READ(A)
B_bal = READ(B)
B_bal += 100
A_bal  -= 100
WRITE(A, A_bal)
WRITE(B, B_bal)
```

```
A_bal = READ(A)
B_bal = READ(B)
Print(A_bal+B_bal)
```

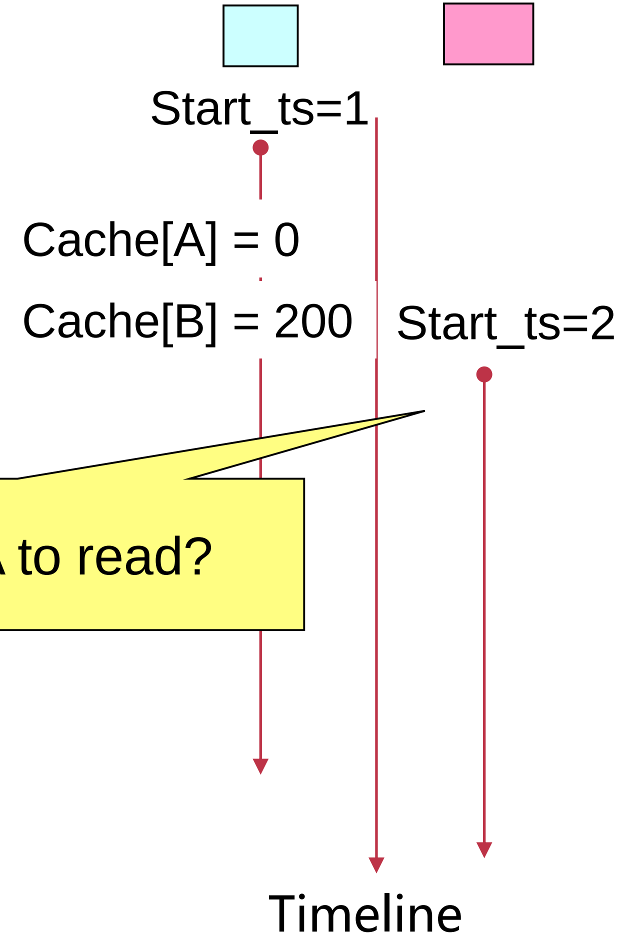
A0 = 100
B0 = 100



Snapshot Isolation: a concrete example

```
A_bal = READ(A)
B_bal = READ(B)
B_bal += 100
A_bal  -= 100
WRITE(A, A_bal)
WRITE(B, B_bal)
```

A0 = 100
B0 = 100



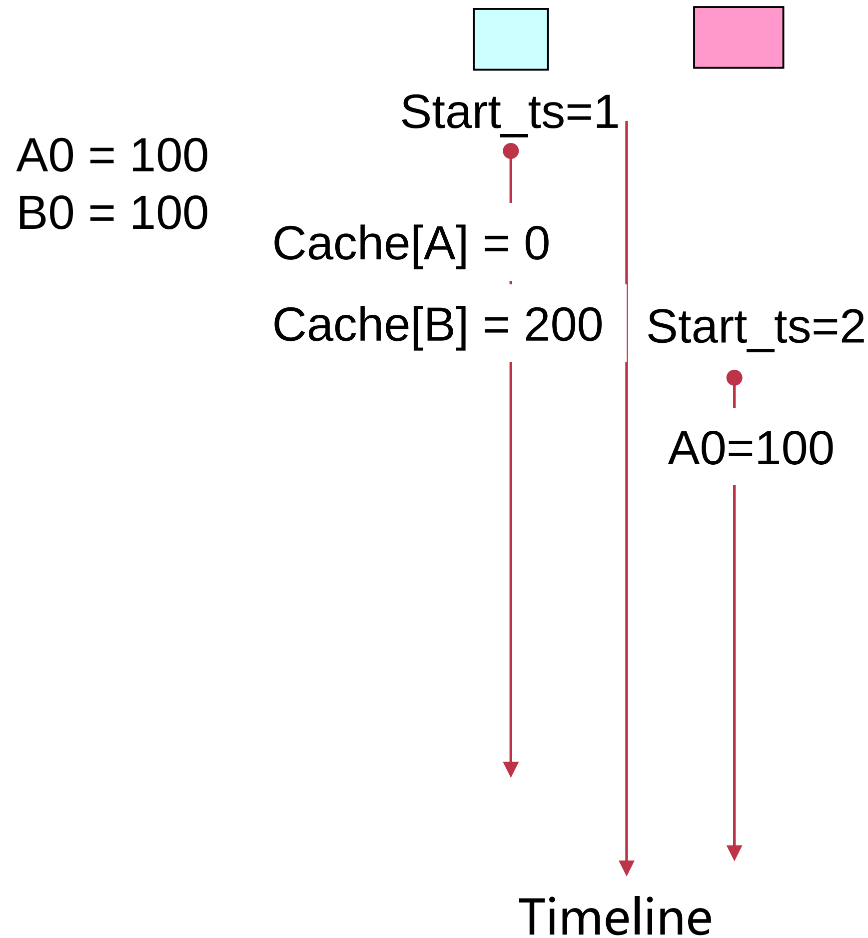
Which time of A to read?

```
A_bal = READ(A)
B_bal = READ(B)
Print(A_bal+B_bal)
```

Snapshot Isolation: a concrete example

```
A_bal = READ(A)
B_bal = READ(B)
B_bal += 100
A_bal  -= 100
WRITE(A, A_bal)
WRITE(B, B_bal)
```

```
A_bal = READ(A)
B_bal = READ(B)
Print(A_bal+B_bal)
```

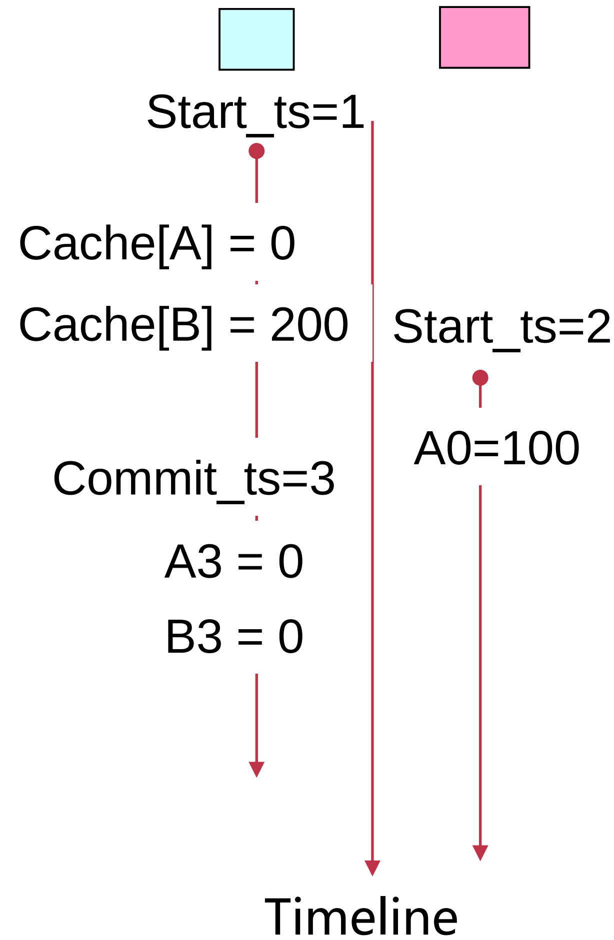


Snapshot Isolation: a concrete example

```
A_bal = READ(A)
B_bal = READ(B)
B_bal += 100
A_bal  -= 100
WRITE(A, A_bal)
WRITE(B, B_bal)
```

A0 = 100
B0 = 100

```
A_bal = READ(A)
B_bal = READ(B)
Print(A_bal+B_bal)
```

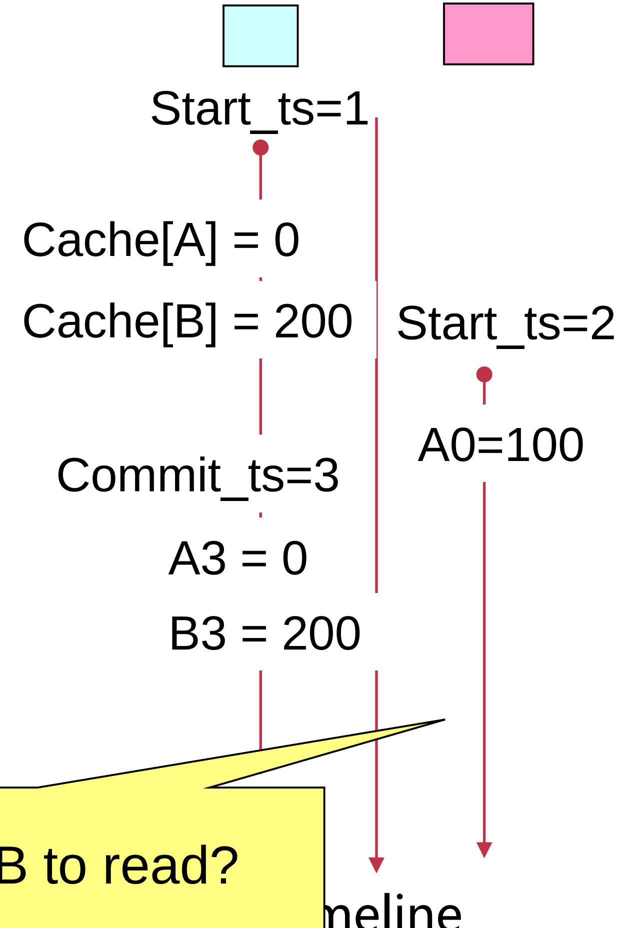


Snapshot Isolation: a concrete example

```
A_bal = READ(A)
B_bal = READ(B)
B_bal += 100
A_bal  -= 100
WRITE(A, A_bal)
WRITE(B, B_bal)
```

```
A_bal = READ(A)
B_bal = READ(B)
Print(A_bal+B_bal)
```

A0 = 100
B0 = 100

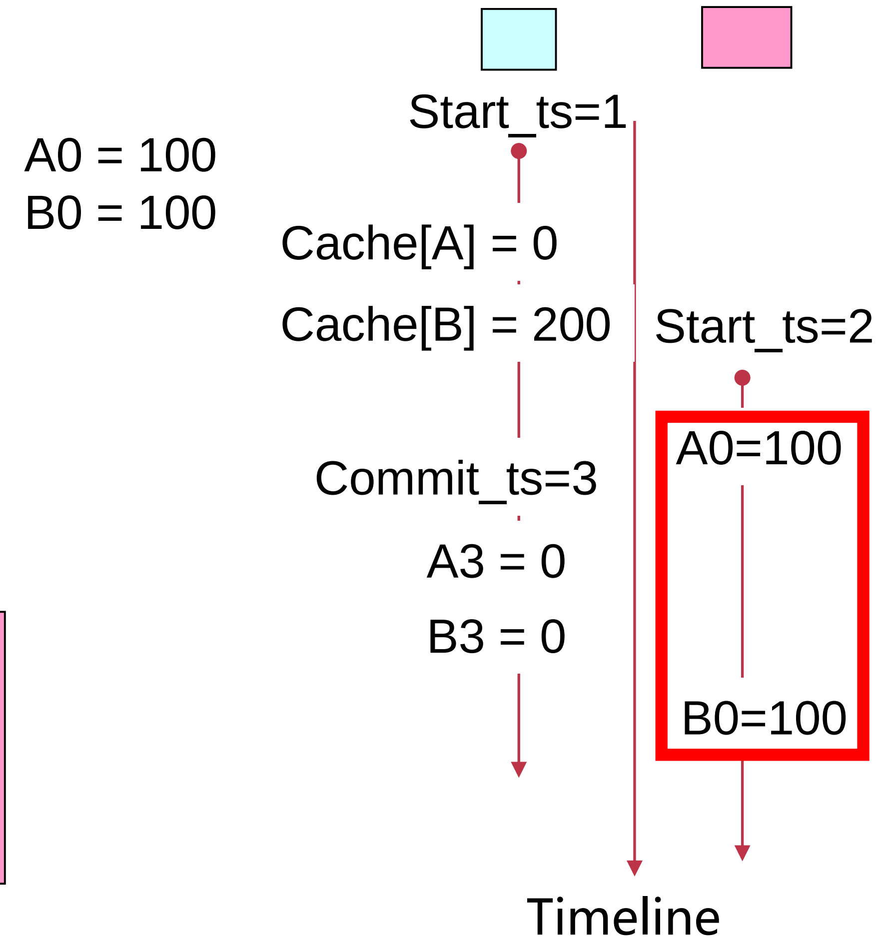


Which time of B to read?

Snapshot Isolation: a concrete example

```
A_bal = READ(A)
B_bal = READ(B)
B_bal += 100
A_bal  -= 100
WRITE(A, A_bal)
WRITE(B, B_bal)
```

```
A_bal = READ(A)
B_bal = READ(B)
Print(A_bal+B_bal)
```



Snapshot isolation < serializability

- The write-skew problem
- Please refer to "A critique of ansi sql isolation levels [SIGMOD' 95" for more detailes

FROM ALGORITHMS TO SYSTEMS

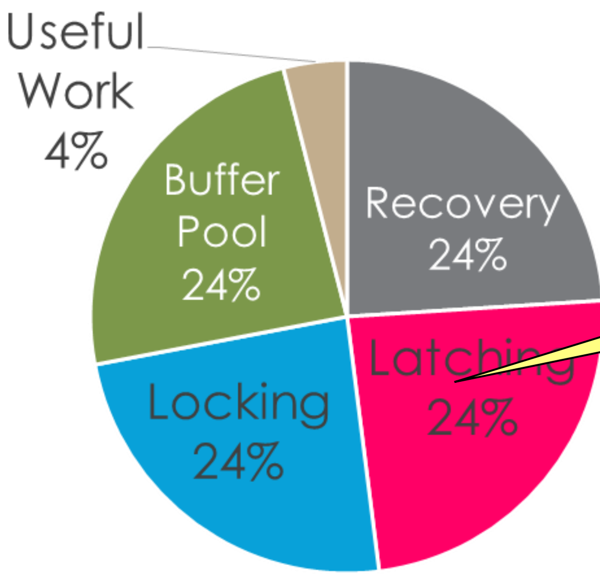
Towards modern transaction systems

- **2PL, OCC and SI are decades old**
  - Yet nowadays they are still the golden standard of concurrency control !
- **How can we continuously improve transaction’s performance?**
  - Through better system designs

Transactions are slow in (traditional) databases

Only **4% of wall-clock time** spent on useful data processing, while the rest is occupied with **buffer pools**, **locking**, **latching**, **recovery**.[1]

-- Michael *Stonebraker*



•50% of time spent on ensuring ACI

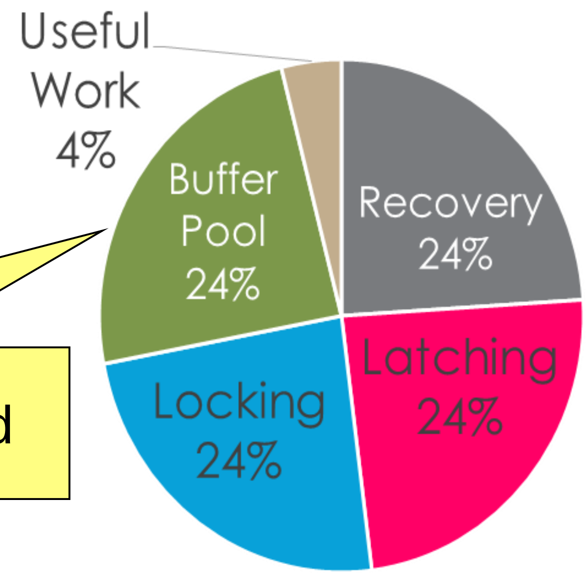
ACI = algorithms + implementations

- **Algorithms (protocols):**
  - 2PL, OCC, etc
- **Implementations**
  - Locking methods (read-write locks), deadlock detection mechanism, etc
  - How to place data (e.g., in-memory, disk, NVM)
  - Different system design choices matters

Case study: Hstore[VLDB' 08]

- **Put all database in memory**
  - Eliminate buffer pool overhead
- **But other parts need redesign**
  - How to do fault tolerance?

•Totally eliminates buffer pool overhead

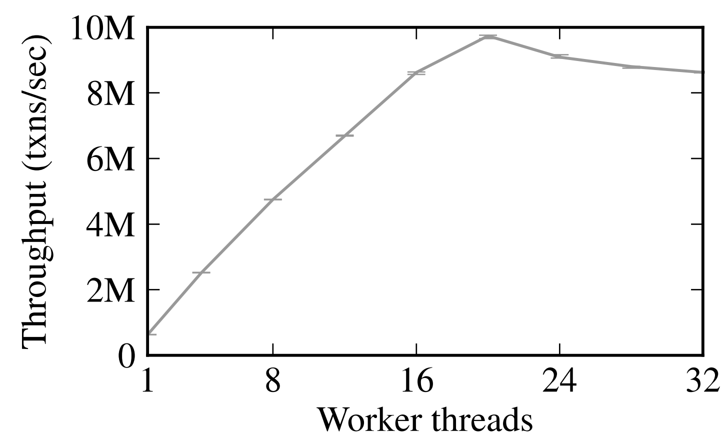


A better algorithm does yield better performance

- **Multi-version incurs better performance**
  - E.g., No abort read-only TX, etc
- **But incurs additional scalability bottleneck**
  - A global counter is used to assign timestamps
  - Global timestamp allocation is slow, even on a single modern multi-core machine !

Scalability bottleneck in multi-core server

```
txn_commit()
{
    // prepare commit
    // […]
    commit_tid =
atomic_fetch_and_add(&global_tid);
    // quickly serialize transactions a la Hekaton
}
```



[Hekaton] concurrency control engine in Microsoft SQL server

Recap: ACI

- **A (Atomicity)**
  - All-or-nothing w.r.t. failures
- **C (Consistency)**
  - Transactions maintain any internal storage state invariants
- **I (Isolation)**
  - Concurrently executing transactions do not interfere
- **D (Durability)**
  - Effect of transactions survive failures

FAULT TOLERANCE

Durability vs. Availability

- **Durability**
  - Storage states survives even if there is failures
  - E.g., TX' modifications flushed to non-volatile storage

- **Availability (for distributed systems)**
  - System survives even if there is failure)

Logging for durability

- **Logging rules**
  - Write log record to disk before modifying persistent state
- **Recovery**
  - After system reboots, drain the logs and recover system states to a consistent state

Challenges of logging for in-memory databases

- **How to recovery logs to a consistent state ?**
  - If T2 reads T1’s modifications, T2’ log must be recovered after T1’s log

- **How to hide disk latency ?**
  - Typically several orders of magnitude longer than TX’s lifetime!

- **Please refer to Silo[SOSP’13]**

Logging + Replication for Availability

- **How to maintain system’s function when there is machine failures ?**
  - Replication: data are replicated to multiple machines; If one machine fails, redirect the workloads to its replica

- **Challenge: How to maintain ACID when there is replications ?**
  - Use logging to sync replica’s states by a consensus protocol, e.g., Paxos

Case study: Spanner[OSDI’10]

- **A distributed (geo-replicated) databases**
- **Use two-phase locking for concurrency control**
- **Uses paxos to sync replication’s state**
  - So that each replica group function as a single machine with no failure

Case study: FaRM[SOSP’15]

- **Drawback of paxos:**
  - At least two network-roundtrips for a single request (e.g., lock, read/write)
- **FaRM uses primary backup replication with vertical paxos**
  - OCC for concurrency control
  - Only the commit phase need to ship log to replicas

Summary

- Protocols to ensure transaction’s ACI property
- How system implementation affect transaction’s performance

Thanks!