TicToc: Time Traveling Optimistic Concurrency Control

Authors: Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, Srinivas Devadas

Presented By: Shreejit Nair

Background: Optimistic Concurrency Control

- Read Phase: Transaction executes on a private copy of all accessed objects.
- Validation Phase: Check for conflicts between transactions.
- Write Phase: Transaction's changes to updated objects are made public.

Background: Timestamp Ordering Algorithm

- A schedule in which the transactions participate is then serializable, and the equivalent serial schedule has the transactions in order of their timestamp values. This is called **timestamp ordering (TO).**

- The algorithm associates with each database item *X* two timestamp (**TS**) values:

  - **Read_TS(*X*):** The **read timestamp** of item *X;* this is the largest timestamp among all the timestamps of transactions that have successfully read item *X*—that is, read _TS(*X*) = TS(T), where T is the *youngest* transaction that has read *X* successfully.

  - **Write_TS**(***X***)**:** The **write timestamp** of item *X;* this is the largest of all the timestamps of transactions that have successfully written item *X*—that is, write_ TS(*X*) = TS(T), where T is the *youngest* transaction that has written *X* successfully.

Background: Timestamp Ordering Algorithm (Contd)

- Whenever some transaction T tries to issue a read_item(*X*) or a write_item(*X*) operation, the **basic TO** algorithm compares the timestamp of T with read_TS(*X*) and write_TS(*X*) to ensure that the timestamp order of transaction execution is not violated.

- The concurrency control algorithm must check whether conflicting operations violate the timestamp ordering in the following two cases:

1. Transaction T issues a **write_item(*X*)** operation:
   a. If read_TS(*X*) > TS(T) or if write_TS(*X*) > TS(T), then abort and roll back T and reject the operation, else execute **write_item(*X*)** & set write_TS(*X*) to TS(T).

2. Transaction T issues a **read_item(X)** operation:
   a. If write_TS(X) > TS(T), then abort and roll back T and reject the operation, else if write_TS(X) ≤ TS(T), then execute the read_item(X) operation of T and set read_TS(X) to the larger of TS(T) and the current read_TS(X).

Why TicToc?

- Basic T/O (*Timestamp-Ordering*) -based concurrency algorithm involves assigning a unique and monotonically increasing timestamp as serial order for conflict detection.

- This centralized timestamp allocation involves implementing an allocator via a global atomic add operation.

- Actual dependency between two transactions may not agree with the assigned timestamp order causing transactions to unnecessarily abort.

- TicToc computes a transaction's timestamp lazily at commit time based on the data it accesses.

- TicToc timestamp management policy avoids centralized timestamp allocation bottleneck and exploits more parallelism in the workload.

TicToc Timestamp Management Policy

- Consider a sequence of operations

1. A read(x)
2. B write(x)
3. B commits
4. A write(y)

What happens when TS(B) < TS(A) in basic T/O?

TicToc Timestamp Commit Invariant

- Every data version in TicToc has a valid range of timestamps bounded by the write timestamp (*wts*) and read timestamp (*rts*)

Version	Tuple Data	Timestamp Range
V1	Data	$[wts_1, rts_1]$
V2	Data	$[wts_2, rts_2]$

Transaction T writes to the tuple

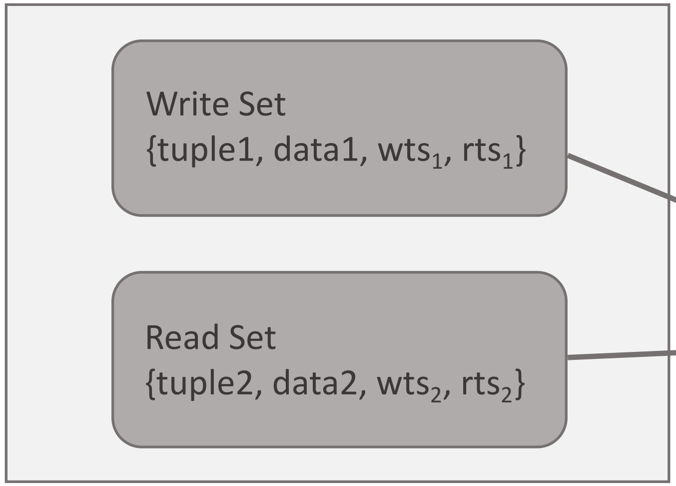
Version	Tuple Data	Timestamp Range
V1	Data	$[wts_1, rts_1]$
V2	Data	$[wts_1, rts_2]$

Transaction T reads from the tuple

- Commit timestamp invariant
  - For all versions read by transaction T, v.wts ≤ commit_ts ≤ v.rts
  - For all versions written by transaction T, v.rts < commit_ts

TicToc Algorithm

- Read phase



Transaction T

Version	Tuple Data	Timestamp Range
V1	data1	$[wts_1, rts_1]$
V1	data2	$[wts_2, rts_2]$

TicToc Algorithm (Contd)

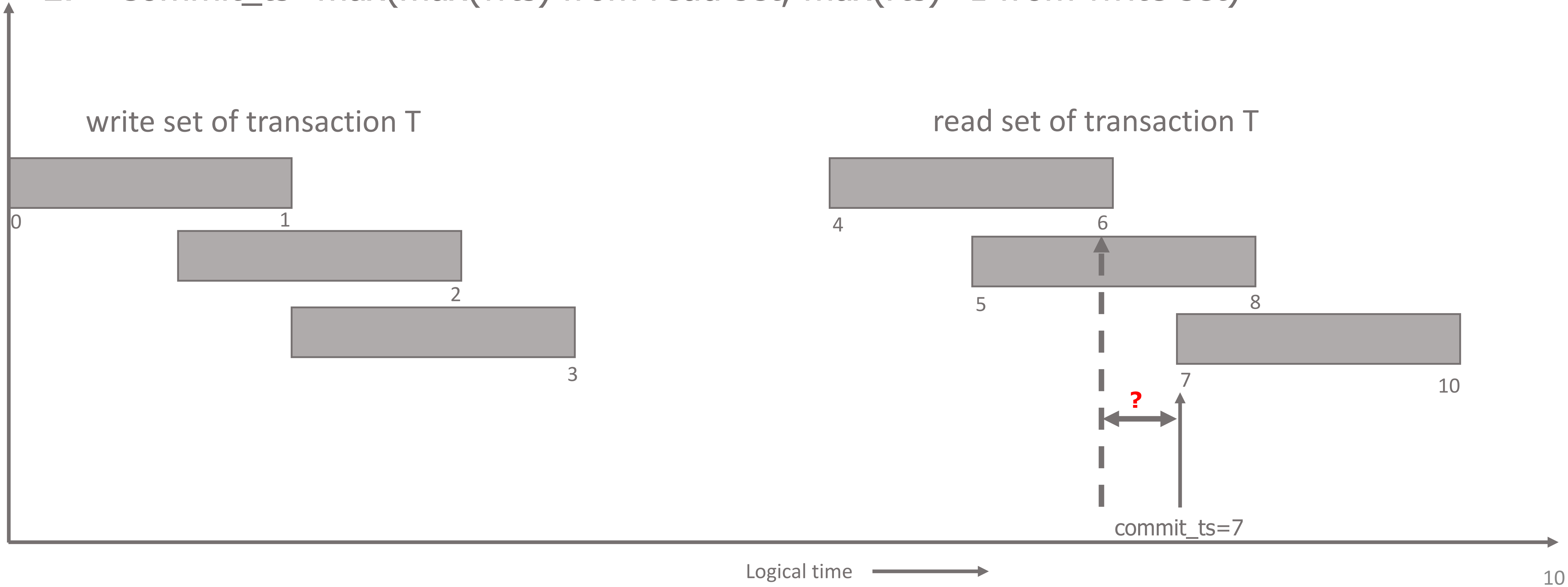
- Validation phase
  1. Lock all tuples in the transaction write set
  2. Commit_ts=max(max(wts) from read set, max(rts)+1 from write set)

Algorithm 2: Validation Phase

```
Data: read set RS, write set WS
# Step 1 – Lock Write Set
for w in sorted(WS) do
    lock(w.tuple)
end
# Step 2 – Compute the Commit Timestamp
commit_ts = 0
for e in WS ∪ RS do
    if e in WS then
        commit_ts = max(commit_ts, e.tuple.rts +1)
    else
        commit_ts = max(commit_ts, e.wts)
    end
end
# Step 3 – Validate the Read Set
for r in RS do
    if r.rts < commit_ts then
        # Begin atomic section
        if r.wts ≠ r.tuple.wts or (r.tuple.rts ≤ commit_ts and
        isLocked(r.tuple) and r.tuple not in W) then
            abort()
        else
            r.tuple.rts = max(commit_ts, r.tuple.rts)
        end
        # End atomic section
    end
end
```

TicToc Algorithm (Contd)

- Validation phase checks
  1. Lock all tuples in the transaction write set
  2. Commit_ts=max(max(wts) from read set, max(rts)+1 from write set)



TicToc Algorithm (Contd)

- Write phase

For all tuples in WS(write set) do:

1. *commit* updated values to database

2. *overwrite* tuple.wts = tuple.rts = commit_ts

3. *unlock(tuple)*

TicToc Working Example

- Step 1: Transaction A reads tuple x

Version	Tuple Data	Timestamp Range
V1	x	[wts=1, rts=3]
V1	y	[wts=1, rts=2]

Read set A = {x,1,3}

- Step 2: Transaction B writes to tuple x and commits at timestamp 4

Version	Tuple Data	Timestamp Range
V2	x	[wts=4, rts=4]
V1	y	[wts=1, rts=2]

Read set A = {x,1,3}

- Step 3: Transaction A writes to tuple y

Version	Tuple Data	Timestamp Range
V2	x	[wts=4, rts=4]
V1	y	[wts=1, rts=2]

Read set A = {x,1,3}
Write set A = {y,1,2}

- Step 4: Transaction A enters validation phase

Version	Tuple Data	Timestamp Range
V2	x	[wts=4, rts=4]
V2	y	[wts=3, rts=3]

Read set A = {x,1,3}
Write set A = {y,1,2}

Tran A commit_ts =3
Tran A COMMITS

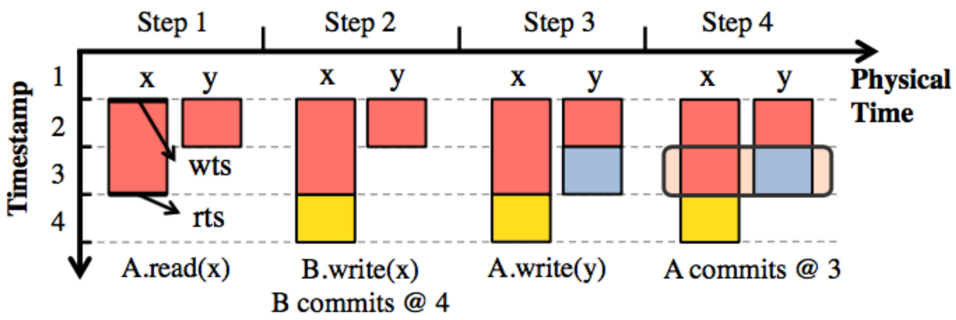


Figure 1: An example of two transactions executing using TicToc.

TicToc Serializability Order

$$A <_s B \triangleq A <_{lts} B \lor (A =_{lts} B \land A \leq_{pts} B)$$

- LEMMA 1 : Transactions writing to the same tuples must have different commit timestamps (lts).
- LEMMA 2: Transactions that commit at the same logical timestamp and physical timestamp do not conflict with each other (e.g. Read-Write or Write-Read operations on the same tuples by different transactions).
- LEMMA 3: A *read* operation from a committed transaction returns the value of the latest *write* to the tuple in the serial schedule.

TicToc Optimizations

- No-Wait locking in validation phase

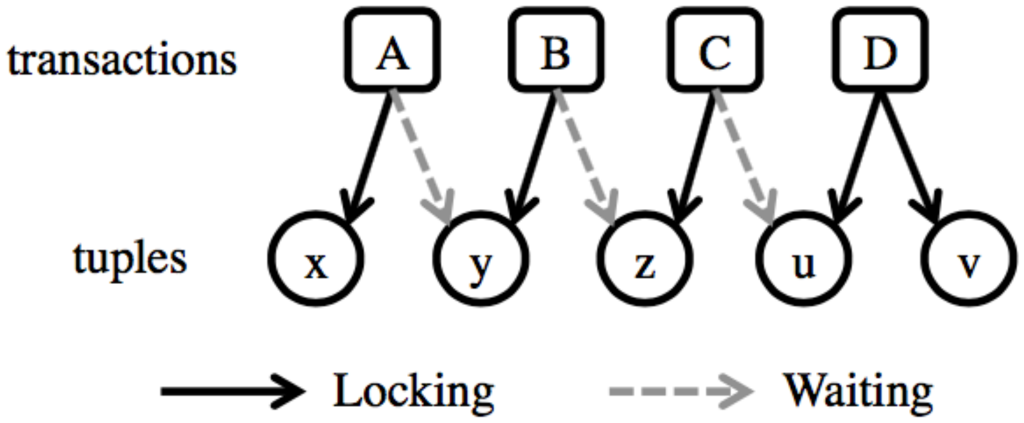


Figure 2: An example of lock thrashing in a 2PL protocol.

TicToc Optimizations (Contd)

- Preemptive Aborts
  - Validation phase causes other transactions to potentially block unnecessarily.
  - Guessing an approximate commit timestamp to observe if transactions would lead to aborts.

Timestamp History Buffer

➢ Step 1: Transaction A reads tuple x

Version	Tuple Data	Timestamp Range
V1	x	[wts=1, rts=2]

Read set A = {x,1,2}

➢ Step 2: Transaction B extends x's rts.

Version	Tuple Data	Timestamp Range
V2	x	[wts=1, rts=3]

Read set A = {x,1,2}

➢ Step 3: Transaction C writes to tuple x

Version	Tuple Data	Timestamp Range
V3	x	[wts=4, rts=4]

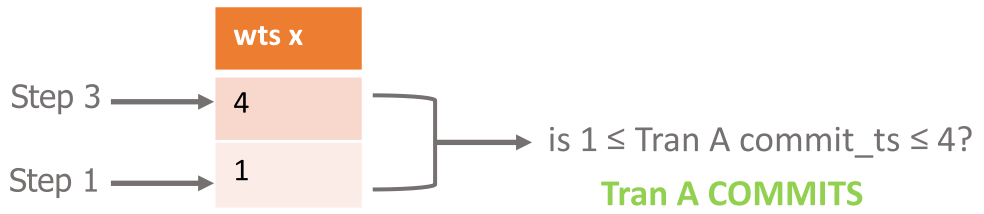
Read set A = {x,1,2}
Tran C commit_ts = 4

➢ Step 4: Transaction A enters validation phase

Version	Tuple Data	Timestamp Range
V3	x	[wts=4, rts=4]

Read set A = {x,1,2}

Tran A commit_ts =3



Timestamp history buffer for tuple x

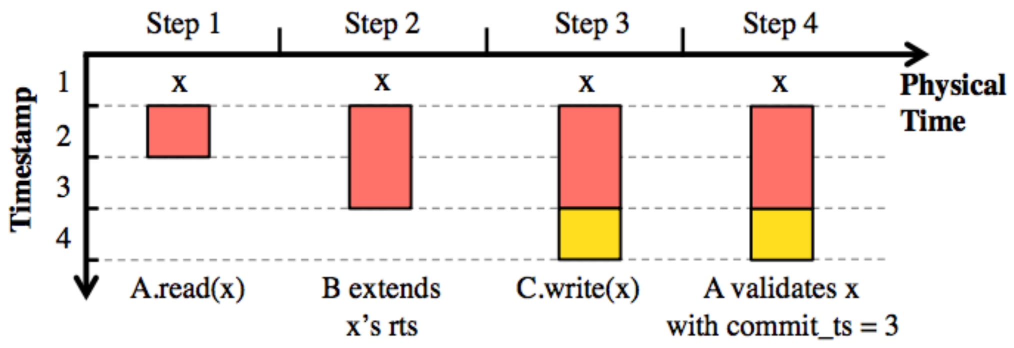


Figure 3: Using a tuple's timestamp history to avoid aborting.

Experimental Evaluation

TICTOC: Time traveling OCC with all optimizations
SILO: Silo OCC
HEKATON: Hekaton MVCC
DL_DETECT: 2PL with deadlock detection
NO_WAIT: 2PL with non-waiting deadlock prevention

System

DBx 1000 40 core machine
4 Intel Xeon E7-4850 128GB RAM
1 Core 2 threads, total 80 threads

Workload

TPC-C
Simulator for warehouse centric order processing application

Experimental Design

Fixed Warehouses
Count 4
(High Contention)

Variable Warehouses
Count [4-80], Threads 80
(Low Contention)

Key Observations

Fixed Warehouses (High Contention):

1. DL_DETECT has worst scalability of all
2. NO_WAIT performs better than DL_DETECT
3. NO_WAIT is worse than TICTOC & SILO due to usage of locks
4. HEKATON is slower than TICTOC due to overhead of multiple versions.
5. TICTOC achieves 1.8x better throughput than SILO & reducing abort rates by 27%.

Variable Warehouses (Low Contention):

1. Advantage of TICTOC over SILO decreases as warehouses increases (contention reduces).
2. TICTOC shows consistently fewer abort rates than SILO due to its timestamp management policy.

Experimental Evaluation (Contd)

System

DBx 1000 40 core machine
4 Intel Xeon E7-4850 128GB RAM
1 Core 2 threads, total 80 threads

Workload

YCSB-C
Standard for large scale online services

Experimental Design

Read-only
2 read queries per transaction

Medium Contention
90% reads, 10% writes
10% hotspot tuples (~60% queries)

High Contention
50% reads, 50% writes 10% hotspot tuples(~75% queries)

Key Observations

Read-only:

1. TICTOC & SILO performs better than other algorithms (no locking overhead).
2. HEKATON concurrency limited by global timestamp counter allocation

Medium Contention:

1. DL_DETECT has worst scalability of all
2. HEKATON performs poorly than SILO, TICTOC due to multi-version overhead
3. TITCTOC has 3.3x lower abort rates than SILO

High Contention:

1. TICTOC & SILO perform almost similarly due to high contention & write intensive workload.

Conclusion

- The paper presented TicToc, a new OCC-based concurrency control algorithm that eliminates the need for centralized timestamp allocation.

- TicToc decouples logical timestamps and physical time by deriving transaction commit timestamps from data items.

- Key features include exploiting more parallelism and reducing transaction abort rates.

- TicToc achieves up to 92% higher throughput while reducing transaction abort rates by up to 3.3x under different workload conditions.

Thoughts…

- TicToc is definitely one of the better performing OCC algorithm.
- Reducing contention within the validation phase?
- Need for write set validation in the validation phase?