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.
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:

1. **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, $\text{read}_\text{TS}(X) = \text{TS}(T)$, where $T$ is the youngest transaction that has read $X$ successfully.

2. **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, $\text{write}_\text{TS}(X) = \text{TS}(T)$, where $T$ is the youngest transaction that has written $X$ successfully.
Whenever some transaction $T$ tries to issue a $\text{read\_item}(X)$ or a $\text{write\_item}(X)$ operation, the **basic TO algorithm** compares the timestamp of $T$ with $\text{read\_TS}(X)$ and $\text{write\_TS}(X)$ 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 $\text{write\_item}(X)$ operation:
   a. If $\text{read\_TS}(X) > \text{TS}(T)$ or if $\text{write\_TS}(X) > \text{TS}(T)$, then abort and roll back $T$ and reject the operation, else execute $\text{write\_item}(X)$ & set $\text{write\_TS}(X)$ to $\text{TS}(T)$.

2. Transaction $T$ issues a $\text{read\_item}(X)$ operation:
   a. If $\text{write\_TS}(X) > \text{TS}(T)$, then abort and roll back $T$ and reject the operation, else if $\text{write\_TS}(X) \leq \text{TS}(T)$, then execute the $\text{read\_item}(X)$ operation of $T$ and set $\text{read\_TS}(X)$ to the larger of $\text{TS}(T)$ and the current $\text{read\_TS}(X)$. 
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.
Consider a sequence of operations

1. $\mathcal{A}$ read($x$)
2. $\mathcal{B}$ write($x$)
3. $\mathcal{B}$ commits
4. $\mathcal{A}$ write($y$)

What happens when $\text{TS}(\mathcal{B}) < \text{TS}(\mathcal{A})$ 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 ($rts$)

<table>
<thead>
<tr>
<th>Version</th>
<th>Tuple Data</th>
<th>Timestamp Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1</td>
<td>Data</td>
<td>$[wts_1, rts_1]$</td>
</tr>
<tr>
<td>V2</td>
<td>Data</td>
<td>$[wts_2, rts_2]$</td>
</tr>
</tbody>
</table>

**Transaction T writes to the tuple**

<table>
<thead>
<tr>
<th>Version</th>
<th>Tuple Data</th>
<th>Timestamp Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1</td>
<td>Data</td>
<td>$[wts_1, rts_1]$</td>
</tr>
<tr>
<td>V2</td>
<td>Data</td>
<td>$[wts_1, rts_2]$</td>
</tr>
</tbody>
</table>

**Transaction T reads from the tuple**

- Commit timestamp invariant
  - For all versions read by transaction T, $v.wts \leq commit\_ts \leq v.rts$
  - For all versions written by transaction T, $v.rts < commit\_ts$
TicToc Algorithm

Read phase

Write Set
{tuple1, data1, wts₁, rts₁}

Read Set
{tuple2, data2, wts₂, rts₂}

Transaction T

<table>
<thead>
<tr>
<th>Version</th>
<th>Tuple Data</th>
<th>Timestamp Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1</td>
<td>data1</td>
<td>[wts₁, rts₁]</td>
</tr>
<tr>
<td>V1</td>
<td>data2</td>
<td>[wts₂, rts₂]</td>
</tr>
</tbody>
</table>
TicToc Algorithm (Contd)

Validation phase

1. Lock all tuples in the transaction write set
2. Commit_ts = \text{max(max(wts) from read set, max(rts) + 1 from write set)}

\begin{algorithm}
\caption{Validation Phase}
\textbf{Data:} read set RS, write set WS
\begin{algorithmic}
  \STATE \textbf{Step 1} – Lock Write Set
  \FOR {w in \text{sorted(WS)}}
  \STATE lock(w.tuple)
  \ENDFOR
  \STATE \textbf{Step 2} – Compute the Commit Timestamp
  \STATE commit_ts = 0
  \FOR {e in \text{WS} \cup RS}
  \IF {e in WS}
  \STATE commit_ts = \text{max(commit_ts, e.tuple.rts + 1)}
  \ELSE
  \STATE commit_ts = \text{max(commit_ts, e.wts)}
  \ENDIF
  \ENDFOR
  \STATE \textbf{Step 3} – Validate the Read Set
  \FOR {r in RS}
  \IF {r.rts < commit_ts}
  \STATE \begin{algorithmic}
    \STATE \text{Begin atomic section}
    \IF {r.wts \neq r.tuple.wts \text{ or } (r.tuple.rts \leq \text{commit_ts and isLocked(r.tuple)} \text{ and r.tuple not in } W)}
    \STATE abort()
    \ELSE
    \STATE r.tuple.rts = \text{max(commit_ts, r.tuple.rts)}
    \ENDIF
  \STATE \end{algorithmic}
  \ENDIF
  \ENDFOR
\end{algorithmic}
\end{algorithm}
Validation phase checks

1. Lock all tuples in the transaction write set

2. Commit_ts=\max(\max(\text{wts})\text{ from read set}, \max(\text{rts})+1\text{ 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 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 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 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.
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.
Preemptive Aborts

- Validation phase causes other transactions to potentially block unnecessarily.
- Guessing an approximate commit timestamp to observe if transactions would lead to aborts.
Step 1: Transaction A reads tuple x

<table>
<thead>
<tr>
<th>Version</th>
<th>Tuple Data</th>
<th>Timestamp Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1</td>
<td>x</td>
<td>[wts=1, rts=2]</td>
</tr>
</tbody>
</table>

Read set A = \{x,1,2\}

Step 2: Transaction B extends x’s rts.

<table>
<thead>
<tr>
<th>Version</th>
<th>Tuple Data</th>
<th>Timestamp Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>V2</td>
<td>x</td>
<td>[wts=1, rts=3]</td>
</tr>
</tbody>
</table>

Read set A = \{x,1,2\}

Step 3: Transaction C writes to tuple x

<table>
<thead>
<tr>
<th>Version</th>
<th>Tuple Data</th>
<th>Timestamp Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>V3</td>
<td>x</td>
<td>[wts=4, rts=4]</td>
</tr>
</tbody>
</table>

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

Step 4: Transaction A enters validation phase

<table>
<thead>
<tr>
<th>Version</th>
<th>Tuple Data</th>
<th>Timestamp Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>V3</td>
<td>x</td>
<td>[wts=4, rts=4]</td>
</tr>
</tbody>
</table>

Read set A = \{x,1,2\}  
Ttran A commit_ts =3

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

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

**TPC-C**: Simulator for warehouse centric order processing application

**System**

**Workload**

**Experimental Design**

**Key Observations**

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%.

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

Workload

Experimental Design

Key Observations

1. TICTOC & SILO perform almost similarly due to high contention & write intensive workload.
2. DL_DETECT has worst scalability of all
3. TICTOC has 3.3x lower abort rates than SILO
4. TICTOC & SILO performs better than other algorithms (no locking overhead).
5. HEKATON concurrency limited by global timestamp counter allocation
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?