Unifying Timestamp with Transaction Ordering for MVCC with Decentralized Scalar Timestamp

Xingda Wei, Rong Chen, Haibo Chen, Zhaoguo Wang, Zhenhan Gong, Binyu Zang
Institute of Parallel and Distributed Systems@Shanghai Jiao Tong University
Multi-versioning is important for TXs[1]

Data are stored in **multiple-copies** (aka, versions)

- e.g., updates **not overwrite** (as in single-versioning)

Single-version database

```
| A = 1 | A = 2 |
```

vs.

Multi-version (MV) database

```
| A = 1 | A_v0 = 1 | A_v1 = 2 |
```

[1] Transactions
Multi-versioning is important for TXs\[^1\]

Data are stored in **multiple-copies** (aka, versions)

**Higher concurrency** for readers writers

- Reader reads from the (consistent) past
- Write **concurrently** updates the latest

---

[1] Transactions
Multi-versioning is important for TXs[1]

Data are stored in **multiple-copies** (aka, versions)

**Higher concurrency** for readers writers

- Reader reads from the (consistent) past
- Write **concurrently** updates the latest

How to assign version & choose version to read?

MV database

[1] Transactions
How to assign version & choose version to read?
How to assign version & choose version to read?

Deciding version should be **consistent & efficient**

- **Consistent** = Match TX’s order

<table>
<thead>
<tr>
<th>Time</th>
<th>A_{v0}=0</th>
<th>A_{v1}=1</th>
<th>A_{v2}=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>TX_1</td>
<td>Print(A)</td>
<td>TX_2</td>
<td>TX_3</td>
</tr>
<tr>
<td></td>
<td>A+=1</td>
<td>A+=1</td>
<td>A+=1</td>
</tr>
</tbody>
</table>

\[ V_0 < v_1 < \ldots < v_2 \]

- i.e., versions sorted by TXs
How to assign version & choose version to read?

Deciding version should be consistent & efficient

- **Consistent** = Match TX’s order
- **Efficient**: minimal impact to the TX
A common approach: use timestamp

Yet, ordering timestamps is **challenging in networked systems**

- Recall **the goals: Consistency + Efficiency**
A common approach: use timestamp

Yet, ordering timestamps is **challenging in networked systems**

- Recall **the goals: Consistency + Efficiency**

**#1. Physical clock**

- Nearly **no cost**

- Not consistent: **unsynchronized clocks**
A common approach: use timestamp

Yet, ordering timestamps is **challenging in networked systems**

- Recall **the goals: Consistency + Efficiency**

#2. Global logical clock (GTS)

- Easy to ensure consistency
A common approach: use timestamp

Yet, ordering timestamps is **challenging in networked systems**

- Recall the goals: Consistency + Efficiency

#2. Global logical clock (**GTS**)  
- Easy to ensure consistency
- Non-trivial costs

Extra network requests per TX
A common approach: use timestamp

Yet, ordering timestamps is **challenging in networked systems**

- Recall the goals: Consistency + Efficiency

**#2. Global logical clock (GTS)**

- ✔ Easy to ensure consistency
- ✗ Non-trivial costs
- ✗ Scalability bottleneck

**Handles all the requests**
A common approach: use timestamp

Yet, ordering timestamps is **challenging in networked systems**

- Recall the goals: Consistency + Efficiency

#3. Global **vectorized** clock (VTS)

- Per-worker clock
A common approach: use timestamp

Yet, ordering timestamps is **challenging in networked systems**

- Recall the goals: Consistency + Efficiency

#3. Global **vectorized** clock (VTS)

- Reduce requests to the global server
A common approach: use timestamp

Yet, ordering timestamps is **challenging in networked systems**

- Recall the goals: Consistency + Efficiency

#3. Global **vectorized clock (VTS)**

- ✔ Reduce requests to the global server
- ✗ Still needs the **global server**
A common approach: use timestamp

Yet, ordering timestamps is **challenging in networked systems**

- Recall the goals: Consistency + Efficiency

#3. Global **vectorized** clock (**VTS**)  
- Reduce requests to the global server
- ✗ Still needs the global server
- ✗ **Non-scalable** meta-data

One clock per worker
Can we make the timestamps ordered, with (nearly) zero cost?
Observation: concurrency control (CC)

Recall: consistency = timestamps match TXs’ order

- While CC orders TXs (e.g., 2PL OCC)

Physical clocks are approximately good

- Hybrid clock: using CC in read-write TX to order physical clocks
This work: **Decentralized Scalar Timestamp**

Scalable scalar timestamp mechanism to enable **MVCC**

- Provide **snapshot reads** for read-only TX (i.e., bypasses original CC)

**Consistent & efficient scalar** hybrid physical-logical clock

- Reader uses the physical clock to read snapshots (bypass CC)
- DST is consistent by piggybacking w/ read-write TX’s CC
This work: **Decentralized Scalar Timestamp**

Scalable scalar timestamp mechanism to enable **MVCC**

- Provide **snapshot reads** for read-only TX (i.e., bypasses original CC)
This work: **Decentralized Scalar Timestamp**

Scalable scalar timestamp mechanism to enable **MVCC**

- Provide **snapshot reads** for read-only TX (i.e., bypasses original CC)
This work: **Decentralized Scalar Timestamp**

Use physical clock as an tentative timestamp

- **Minimal cost & Scalable**: scalar + no global timestamp
This work: **Decentralized Scalar Timestamp**

Use the *(un-sync)* physical is not sufficient for read-write TXs

- DST is consistent by piggybacking w/ read-write TX’s CC

\[ \text{Read-write(TX}_1) \]

\[ \begin{align*}
\text{TX start} & \quad v = \quad \text{Write } A_v \\
\text{TX end} & \quad \text{Network} \\
\end{align*} \]

\[ \text{Data server} \]

\[ \begin{align*}
(TX}_2) \quad \text{Read-only} & \quad w/ \text{DST} \\
\text{Network} & \quad v = \quad \text{Piggyback on CC:} \\
\text{Update } v \text{ to be consistent} & \quad \text{CC} \\
\end{align*} \]
This work: **Decentralized Scalar Timestamp**

Use physical clock as an tentative timestamp

- Also **fresh**: snapshot has **bounded staleness** (e.g., ms-scale)
Decentralized (Scalar) Timestamp is not so new

Based on many prior work on decentralized timestamps

- TicToc@SIGMOD’18, Cicada@SIGMOD’17, etc

Differently:

- DST is a general timestamp method for different CC
- DST further targets MVCC in a distributed setting
Outline of the remaining content

How to piggyback on CC for read-write TX? DST rules

How snapshot reads bypass CC? DST reads

How good is DST’s snapshot read? Bounded staleness

Evaluation results
Outline of the remaining content

How to piggyback on CC for read-write TX? **DST rules**

How snapshot reads bypass CC? DST reads

How good is DST’s snapshot read? Bounded-staleness

Evaluations results
DST rules for read-write TX

Inspired from **conflicting relationships** of TXs

Two TXs conflict if and only if:

<table>
<thead>
<tr>
<th>TX access</th>
<th>Read</th>
<th>Read</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td>-</td>
<td>Read-Write</td>
</tr>
<tr>
<td>Write</td>
<td>Write-Read</td>
<td>Write-Write</td>
</tr>
</tbody>
</table>
DST rules for read-write TX

Inspired from **conflicting relationships** of TXs

Two TXs conflict if and only if:

<table>
<thead>
<tr>
<th>TX access</th>
<th>Read</th>
<th>Read</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td>-</td>
<td>Read-Write</td>
</tr>
<tr>
<td>Write</td>
<td>Write-Read</td>
<td>Write-Write</td>
</tr>
</tbody>
</table>

Timestamp should adjust **follow these rules**, too!
Write-Write

If TX₂ has a write-write conflict with TX₁

- $\text{Timestamp}_{tx₂}(Tx₂) \text{ should } > \text{ Timestamp}_{tx₁}(Tx₁)$
Write-Write

If TX$_2$ has a write-write conflict with TX$_1$

- Timestamp$_{tx_2}$(Tx2) should \( > \) Timestamp$_{tx_1}$(Tx1)

Achieved via storing the (last-writer) timestamp with data

Write-Write rule#1:
\[
A.\text{ver} = TX_1
\]

Write-Write rule#2:
\[
TX' = \text{MAX}(TX, A.\text{ver}) + 1
\]
Example: Write-Write in 2PL

Can be simply piggybacked to 2PL in a lightweight way

```plaintext
2PL
A.lock();

A.write(...);
A.unlock();
```
Example: Write-Write in 2PL

Can be simply piggybacked to 2PL in a lightweight way

```
A
```

Server

```
DST
TS = clock();
```

```
2PL+DST
A.lock();
ret A.ver;
```

```
2PL+DST
A.write(...);
A.ver = TS;
A.unlock();
```

```
TX1
Write(A)
```

```
Start
```

```
Lock(A)
```

```
RPC
```

```
RPC
```

```
Write &
Unlock(A)
```

```
TS = MAX(TS, A.ver) + 1
```

```
DST
```
DST rules for read-write TX

Inspired from *conflicting relationships* of TXs

Two TXs conflict if and only if:

<table>
<thead>
<tr>
<th>TX access</th>
<th>Read</th>
<th>Read</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td>-</td>
<td>Read-Write</td>
</tr>
<tr>
<td>Write</td>
<td>Write-Read</td>
<td>Write-Write</td>
</tr>
</tbody>
</table>

Timestamp should adjust *follow these rules*, too!
Outline of the remaining content

How to piggyback on CC for read-write TX? DST rules

How snapshot reads bypass CC? DST reads

How good is DST’s snapshot read? Bounded staleness

Evaluation results
DST reads

Key benefit: Bypass CC for the reader

- e.g., avoid read locks in 2PL
DST reads

Key benefit: Bypass CC for the reader

- e.g., avoid read locks in 2PL

Can we use MVCC w/ DST the same as global clock?
DST reads

Key benefit: Bypass CC for the reader

- e.g., avoid read locks in 2PL

Can we use MVCC w/ DST the same as global clock?

```
Server

DST
TS=clock();

TX1

Read (A)

Start

RPC

MVCC read

Read A w/ version less up to TS;

Time

Start

Read A@TS
```

39
DST reads

Key benefit: Bypass CC for the reader
- e.g., avoid read locks in 2PL

Can we use MVCC w/ DST the same as global clock?
- No, because timestamp of reader does not use CC!
- i.e., not apply DST rules because rules are piggyback to the CC!
DST reads

Key benefit: Bypass CC for the reader

- e.g., avoid read locks in 2PL

DST reads (a lightweight rule for snapshot) to fix this:

\[
\text{DST} \quad \text{TS} = \text{clock}();
\]

\[
\text{MVCC read} + \text{DST}
\]

\[
\begin{align*}
A.\text{ver} &= \max(A.\text{ver}, \text{TS}); \\
\text{Read A w/ version} \quad \text{less up to TS;}
\end{align*}
\]
Outline of the remaining content

How to piggyback on CC for read-write TX? DST rules
How snapshot reads bypass CC? DST reads
How good is DST’s snapshot read? Bounded staleness

Evaluation results
Evaluation results of DST

DST works well various CCs

- 2PL $\iff$ MySQL cluster $\times$ DST
- OCC $\iff$ DrTM+R$^{[Eurosys'16]}$ $\times$ DST
- ROCOCO $\iff$ ROCOCO$^{[OSDI'14]}$ $\times$ DST

Also on various network settings

- Local clusters (TCP/IP), AWS cloud, RDMA
Evaluation results of DST

DST works well various CCs

- 2PL $\rightarrow$ MySQL cluster $\times$ DST
- OCC $\rightarrow$ DrTM+R$^{[Eurosys'16]}$ $\times$ DST
- ROCOCO $\rightarrow$ ROCOCO$^{[OSDI'14]}$ $\times$ DST

Also on various network settings

- Local clusters (TCP/IP), AWS cloud, RDMA
2PL X DST

<table>
<thead>
<tr>
<th>Cluster Name</th>
<th>CPU</th>
<th>Network</th>
<th>Num</th>
</tr>
</thead>
<tbody>
<tr>
<td>Val</td>
<td>2 X 12 cores</td>
<td>10GbE</td>
<td>16</td>
</tr>
</tbody>
</table>

Workloads: TPC-C, Smallbank

- DST ~≈ RC in performance
- Beneficial to write-mostly workloads
- Reduce lock between read/write

**Graphs:**

- TPC-C Thpt (KTxs/sec)
- SmallBank Thpt (MTxs/sec)
OCC X DST

Workloads: TPC-E (read-mostly)

- DST also $\sim=\text{RC}$ in performance

<table>
<thead>
<tr>
<th>Cluster Name</th>
<th>CPU</th>
<th>Network</th>
<th>Num</th>
</tr>
</thead>
<tbody>
<tr>
<td>AWS</td>
<td>r4.2xlarge (8 vcpu)</td>
<td>10GbE</td>
<td>32</td>
</tr>
<tr>
<td>VALR</td>
<td>2 X 12 cores</td>
<td>100Gbp IB (RDMA)</td>
<td>16</td>
</tr>
</tbody>
</table>

Latency (ms) vs Throughput (K reqs/sec)

- With RDMA
### OCC X DST

**Workloads: TPC-E (read-mostly)**

- **Near zero-cost timestamp overhead**

<table>
<thead>
<tr>
<th>Cluster Name</th>
<th>CPU</th>
<th>Network</th>
<th>Num</th>
</tr>
</thead>
<tbody>
<tr>
<td>AWS</td>
<td>r4.2xlarge (8 vcpu)</td>
<td>10GbE</td>
<td>32</td>
</tr>
<tr>
<td>VALR</td>
<td>2 X 12 cores</td>
<td>100Gbp IB (RDMA)</td>
<td>16</td>
</tr>
</tbody>
</table>

![Graph showing latency and throughput](image)

![Graph showing latency with RDMA](image)
Conclusion

DST provides **serializable & fresh MVCC** in a networked setting

- Also generalized to different CCs

**Efficient & Scalable:** Near zero-cost timestamp overhead

Thanks & QA!
Backup