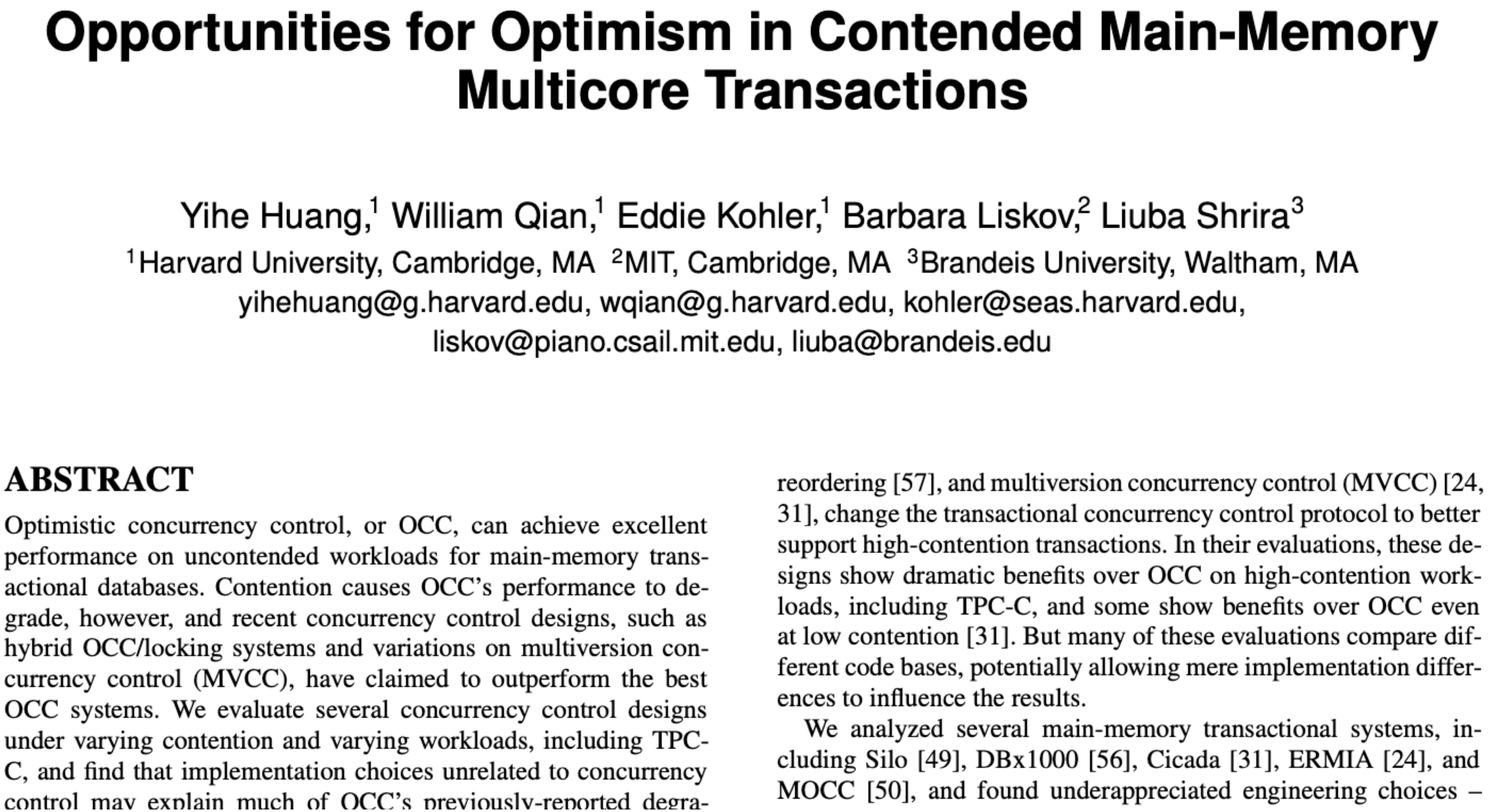
WISCONSIN
UNIVERSITY OF WISCONSIN–MADISON

CS 764: Topics in Database Management Systems
Lecture 24: Modern OCC

Xiangyao Yu
11/25/2020

Today’s Paper: Modern OCC



Opportunities for Optimism in Contended Main-Memory Multicore Transactions

Yihe Huang,[1] William Qian,[1] Eddie Kohler,[1] Barbara Liskov,[2] Liuba Shrira[3]
[1]Harvard University, Cambridge, MA [2]MIT, Cambridge, MA [3]Brandeis University, Waltham, MA
yihehuang@g.harvard.edu, wqian@g.harvard.edu, kohler@seas.harvard.edu,
liskov@piano.csail.mit.edu, liuba@brandeis.edu

ABSTRACT

Optimistic concurrency control, or OCC, can achieve excellent performance on uncontended workloads for main-memory transactional databases. Contention causes OCC’s performance to degrade, however, and recent concurrency control designs, such as hybrid OCC/locking systems and variations on multiversion concurrency control (MVCC), have claimed to outperform the best OCC systems. We evaluate several concurrency control designs under varying contention and varying workloads, including TPC-C, and find that implementation choices unrelated to concurrency control may explain much of OCC’s previously-reported degra-

reordering [57], and multiversion concurrency control (MVCC) [24, 31], change the transactional concurrency control protocol to better support high-contention transactions. In their evaluations, these designs show dramatic benefits over OCC on high-contention workloads, including TPC-C, and some show benefits over OCC even at low contention [31]. But many of these evaluations compare different code bases, potentially allowing mere implementation differences to influence the results.

We analyzed several main-memory transactional systems, including Silo [49], DBx1000 [56], Cicada [31], ERMIA [24], and MOCC [50], and found underappreciated engineering choices –

VLDB 2020 (best paper award)

Outline

Lecture 7 Recap (optimistic concurrency control)

Modern OCC protocols

- Silo
- TicToc

MVCC

Basis factors

Evaluation

OCC, 1981

Goal: eliminating pessimistic locking

Three executing phases:

- Read
- Validation
- Write

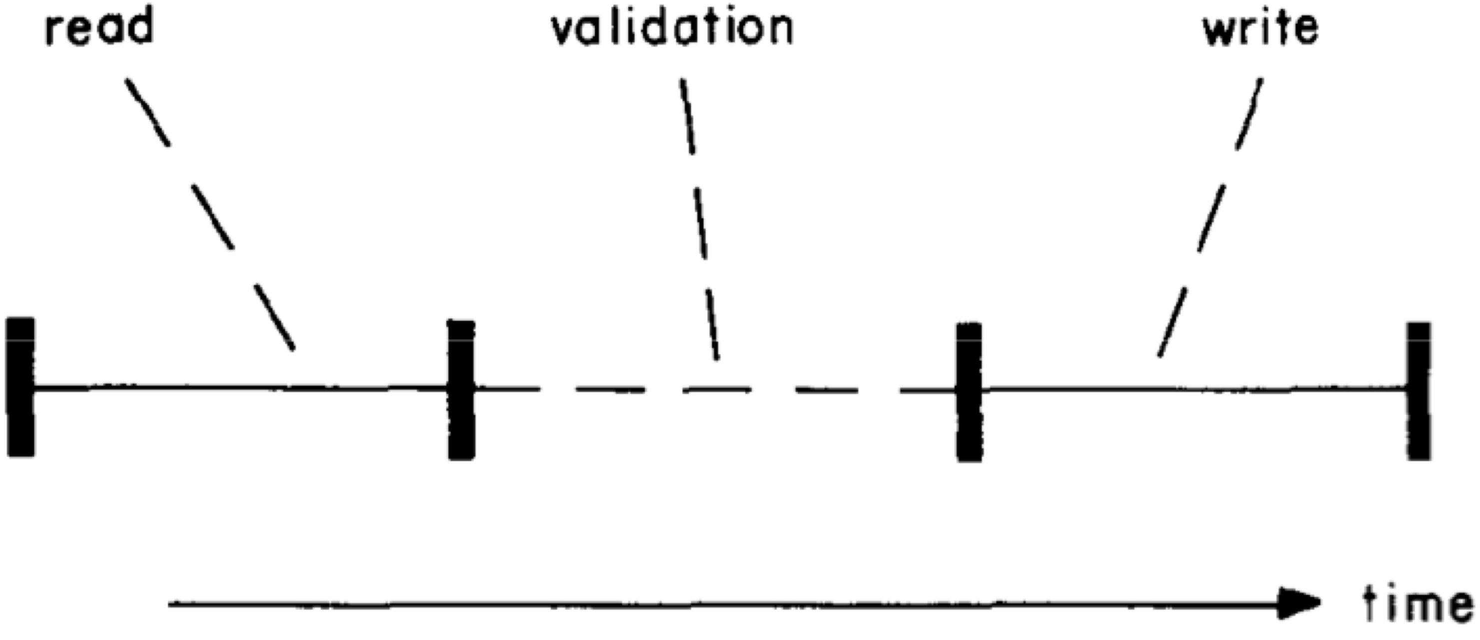
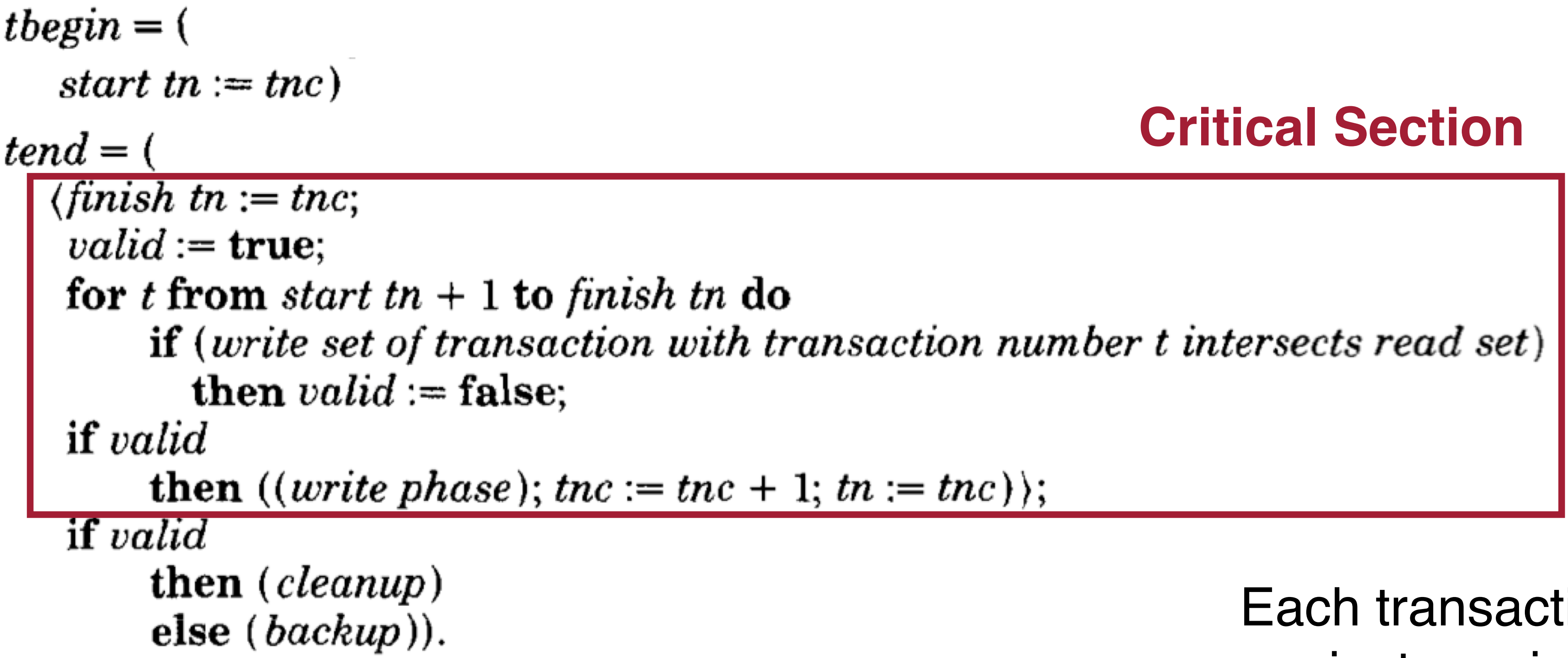


Fig. 1. The three phases of a transaction.

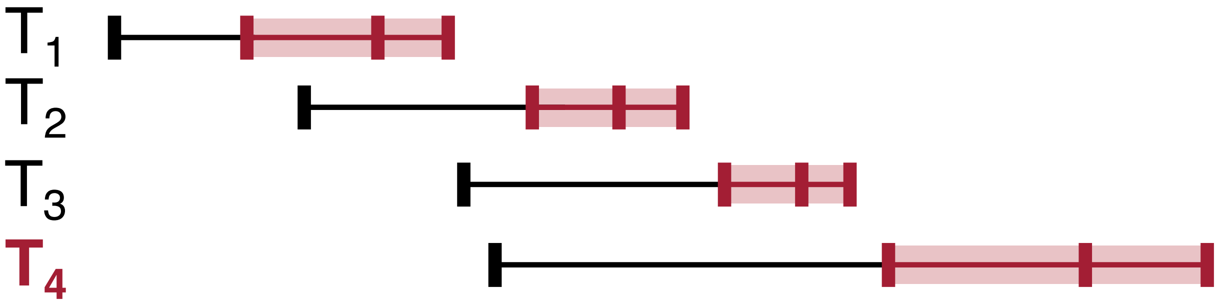
OCC, 1981 — Serial Validation



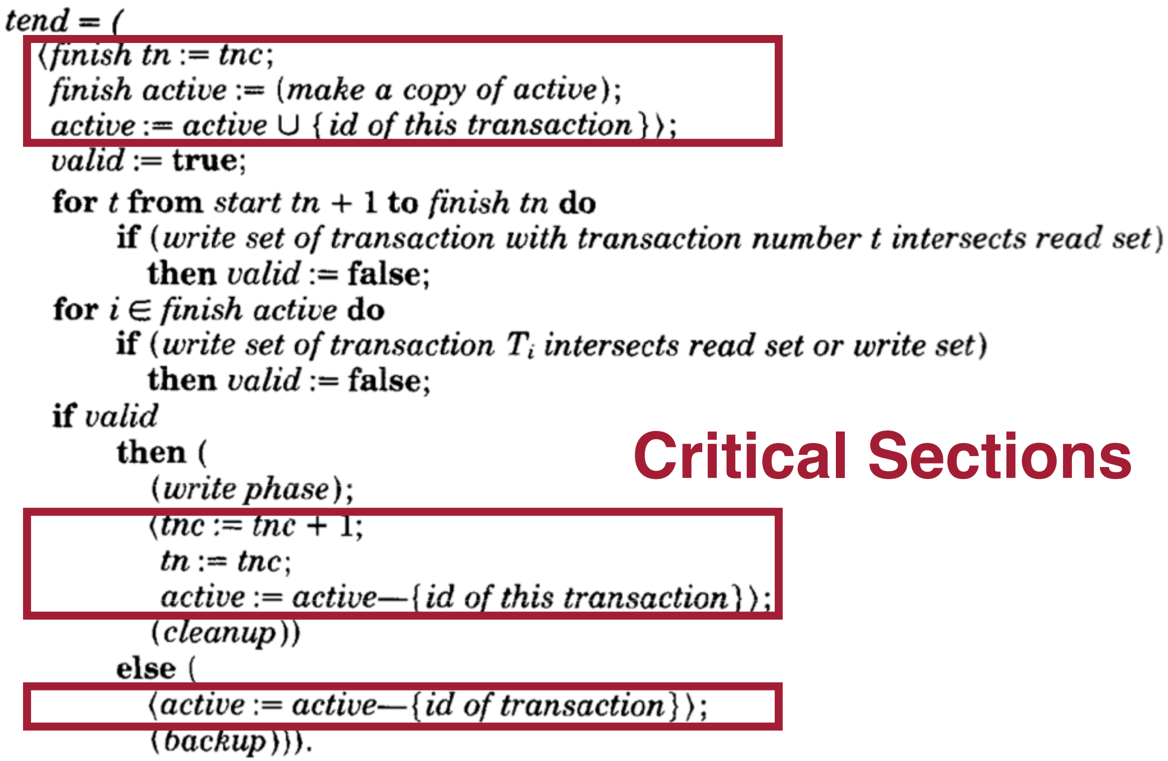
```
tbegin = (
    start tn := tnc)
tend = (
  ⟨finish tn := tnc;
   valid := true;
   for t from start tn + 1 to finish tn do
       if (write set of transaction with transaction number t intersects read set)
          then valid := false;
   if valid
       then ((write phase); tnc := tnc + 1; tn := tnc)⟩;
   if valid
       then (cleanup)
       else (backup)).
```

Critical Section

Each transaction is validated against previous transactions



OCC, 1981 — Parallel Validation



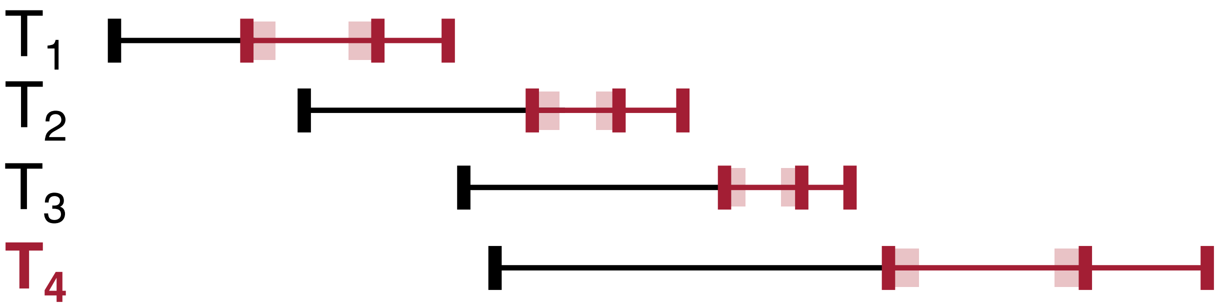
```
tend = (
    ⟨finish tn := tnc;
    finish active := (make a copy of active);
    active := active ∪ {id of this transaction}⟩;
    valid := true;
    for t from start tn + 1 to finish tn do
        if (write set of transaction with transaction number t intersects read set)
            then valid := false;
    for i ∈ finish active do
        if (write set of transaction $T_i$ intersects read set or write set)
            then valid := false;
    if valid
        then (
            (write phase);
            ⟨tnc := tnc + 1;
             tn := tnc;
             active := active—{id of this transaction}⟩;
            (cleanup))
        else (
            ⟨active := active—{id of transaction}⟩;
            (backup))).
```

Critical Sections

Each transaction is validated against previous transactions

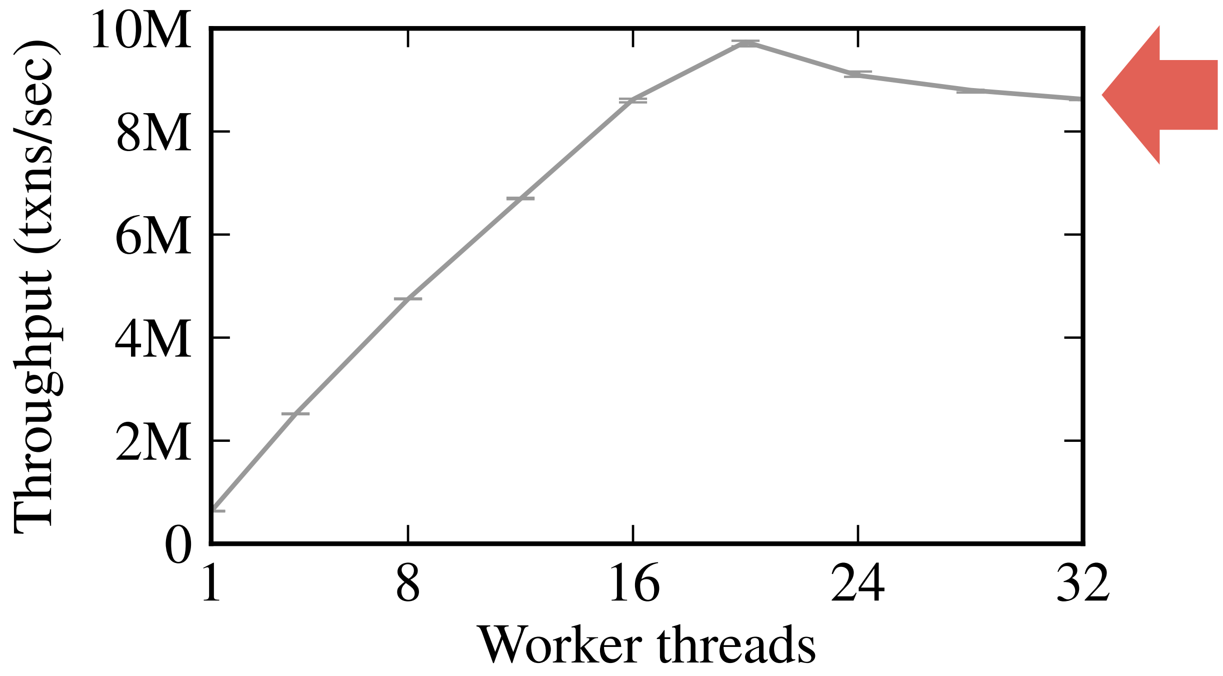
Issue 1: Critical sections become scalability bottlenecks

Issue 2: Need to compare write sets even for non-conflicting transactions



Silo OCC (SOSP 2013)

`atomic_fetch_and_add(&lsn, size);`



Even a single atomic instruction can become a scalability bottleneck

[1] Tu, Stephen, et al. "Speedy transactions in multicore in-memory databases." SOSP 2013

Silo Protocol — Record Layout

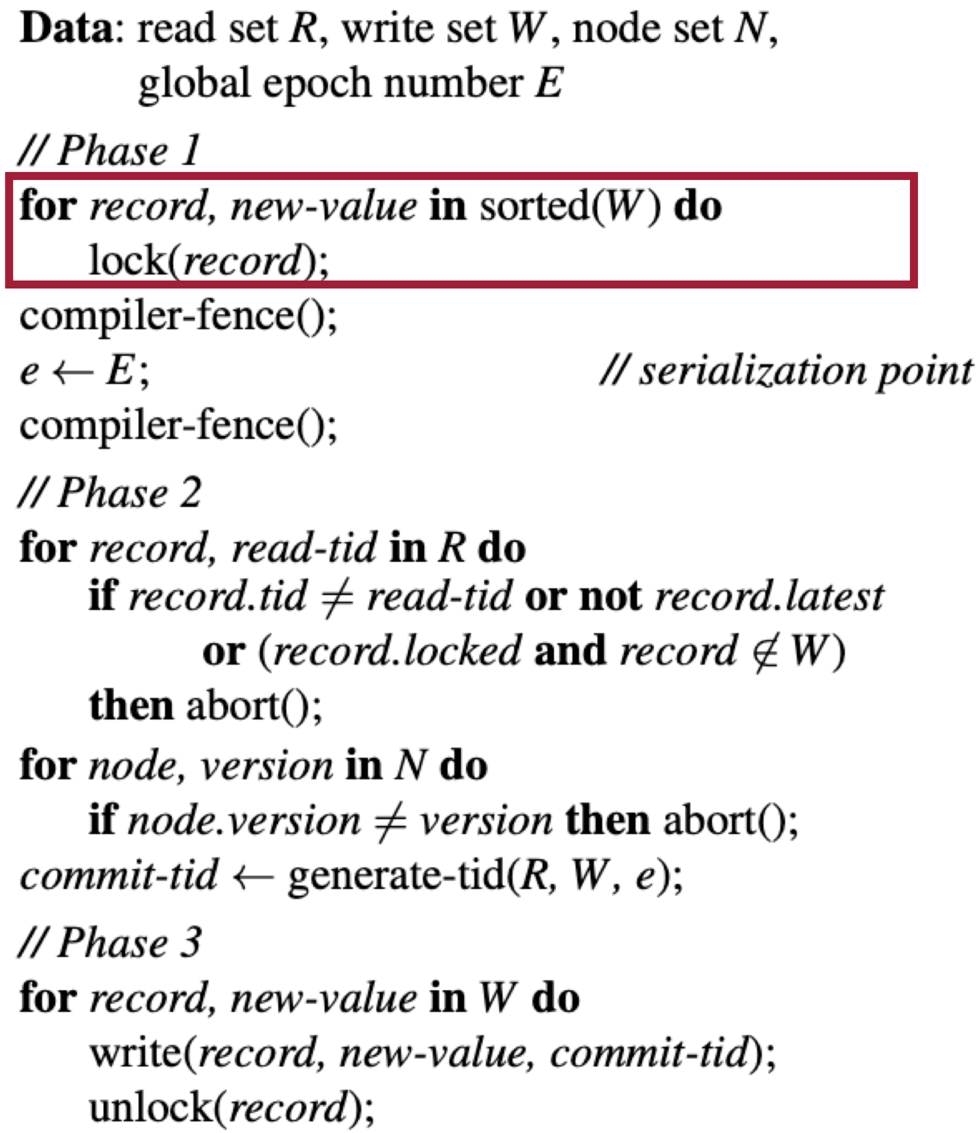
Each tuple contains a TID word which is broken into three pieces:

Sequence number: version number of the tuple

The sequence number is read together with the tuple data

The sequence number is incremented when the tuples is updated

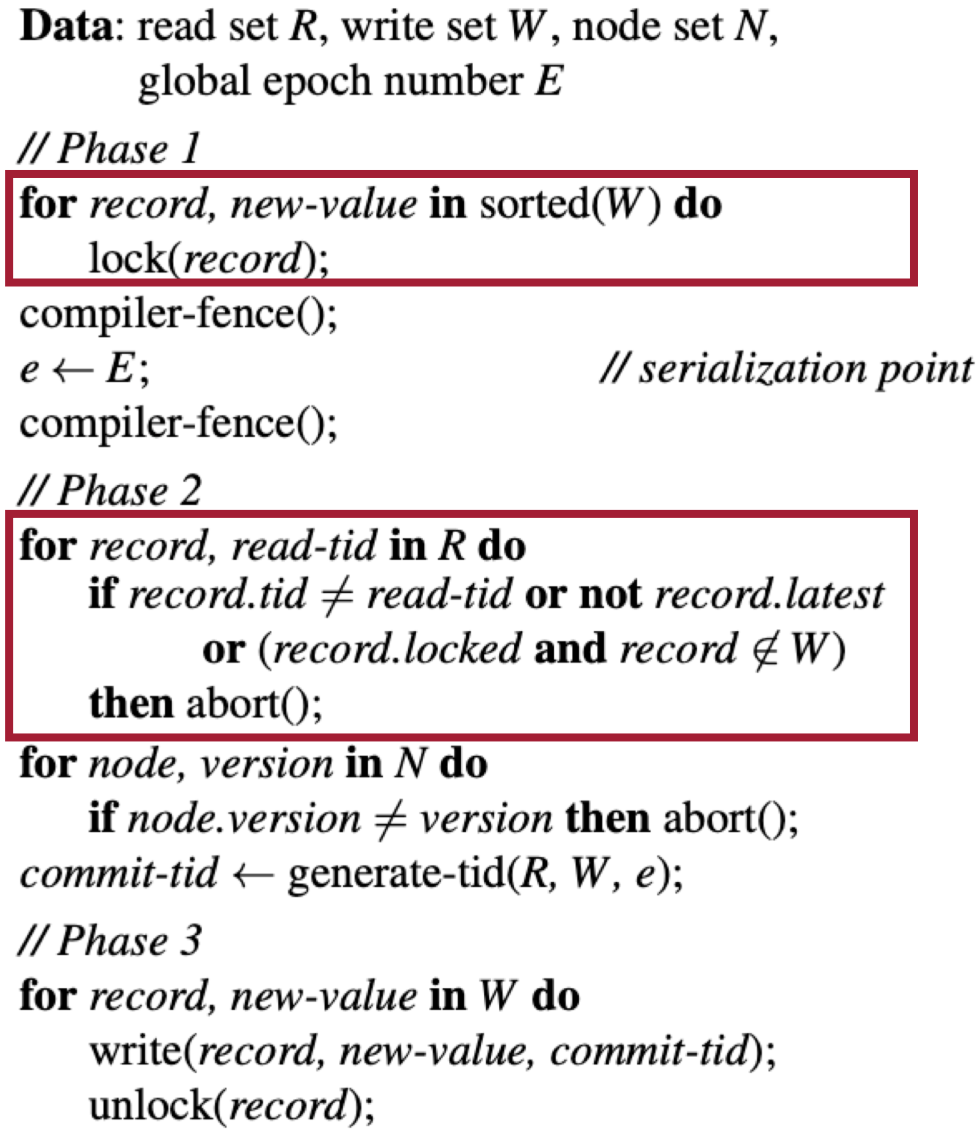
Silo Protocol — Validation and Write Phase



```
Data: read set $R$, write set $W$, node set $N$,
      global epoch number $E$
// Phase 1
for record, new-value in sorted($W$) do
    lock(record);
compiler-fence();
$e \leftarrow E$;                    // serialization point
compiler-fence();
// Phase 2
for record, read-tid in $R$ do
    if record.tid $\neq$ read-tid or not record.latest
            or (record.locked and record $\notin W$)
    then abort();
for node, version in $N$ do
    if node.version $\neq$ version then abort();
commit-tid $\leftarrow$ generate-tid($R$, $W$, $e$);
// Phase 3
for record, new-value in $W$ do
    write(record, new-value, commit-tid);
    unlock(record);
```

Phase 1: Lock the write set

Silo Protocol — Validation and Write Phase



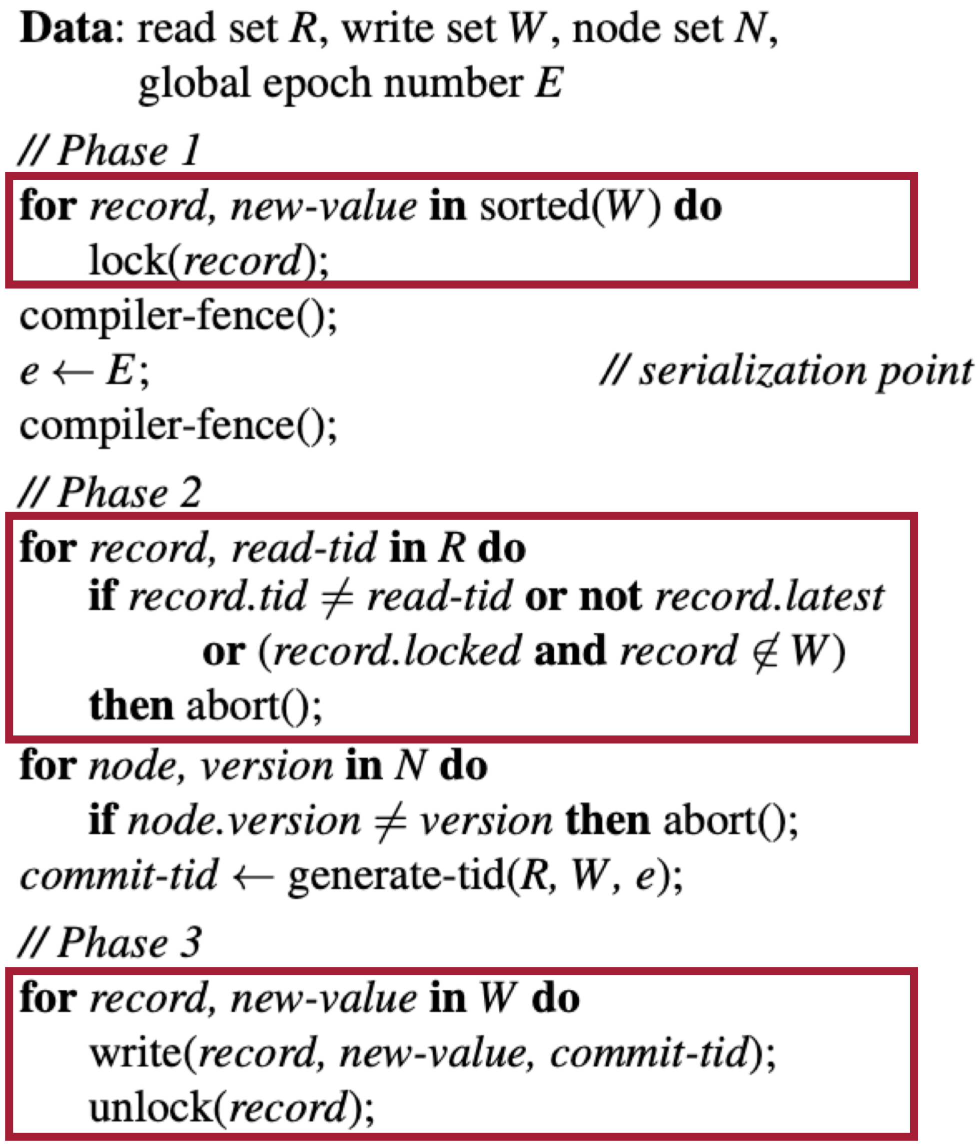
```
Data: read set $R$, write set $W$, node set $N$,
        global epoch number $E$
// Phase 1
for record, new-value in sorted($W$) do
    lock(record);
compiler-fence();
$e \leftarrow E$;                        // serialization point
compiler-fence();
// Phase 2
for record, read-tid in $R$ do
    if record.tid $\neq$ read-tid or not record.latest
            or (record.locked and record $\notin W$)
    then abort();
for node, version in $N$ do
    if node.version $\neq$ version then abort();
commit-tid $\leftarrow$ generate-tid($R$, $W$, $e$);
// Phase 3
for record, new-value in $W$ do
    write(record, new-value, commit-tid);
    unlock(record);
```

Phase 1: Lock the write set

Phase 2: Validate the read set

- Validation fails if (1) the tuple has been modified since the earlier read (TIDs don’t match) or (2) the tuple has been locked

Silo Protocol — Validation and Write Phase



```
Data: read set $R$, write set $W$, node set $N$,
      global epoch number $E$
// Phase 1
for record, new-value in sorted($W$) do
    lock(record);
compiler-fence();
$e \leftarrow E$;                        // serialization point
compiler-fence();
// Phase 2
for record, read-tid in $R$ do
    if $record.tid \neq read\text{-}tid$ or not record.latest
            or (record.locked and $record \notin W$)
    then abort();
for node, version in $N$ do
    if $node.version \neq version$ then abort();
$commit\text{-}tid \leftarrow$ generate-tid($R$, $W$, $e$);
// Phase 3
for record, new-value in $W$ do
    write(record, new-value, commit-tid);
    unlock(record);
```

Phase 1: Lock the write set

Phase 2: Validate the read set

- Validation fails if (1) the tuple has been modified since the earlier read (TIDs don’t match) or (2) the tuple has been locked

Phase 3: Write to database

Silo vs. OCC 1981

Validation against previous transactions vs. tuple versions

Silo vs. OCC 1981

Validation against previous transactions vs. tuple versions

Fault tolerance mechanism (skipped in this lecture)

Silo vs. OCC 1981

Validation against previous transactions vs. tuple versions

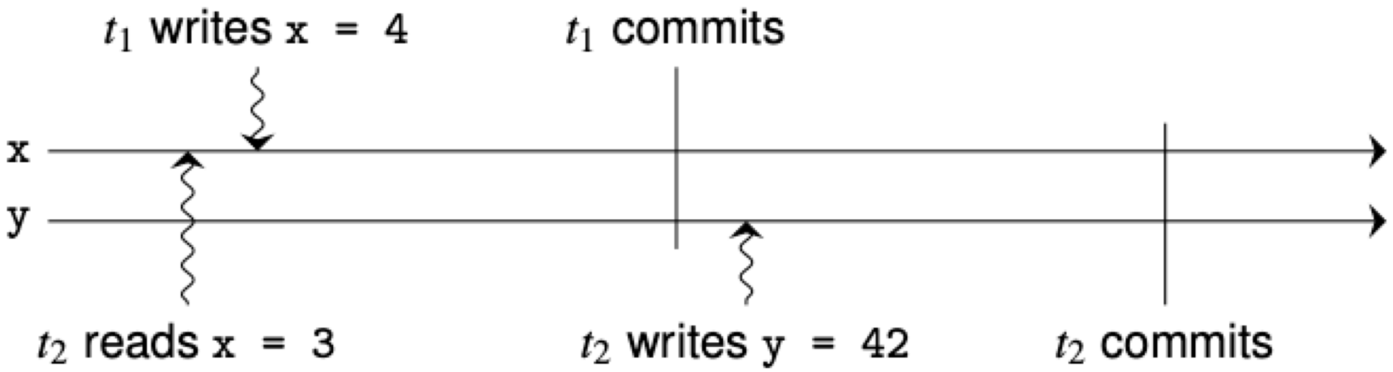
Fault tolerance mechanism (skipped in this lecture)

Low-level optimizations

How to consistently read a record and its TID word without latching?

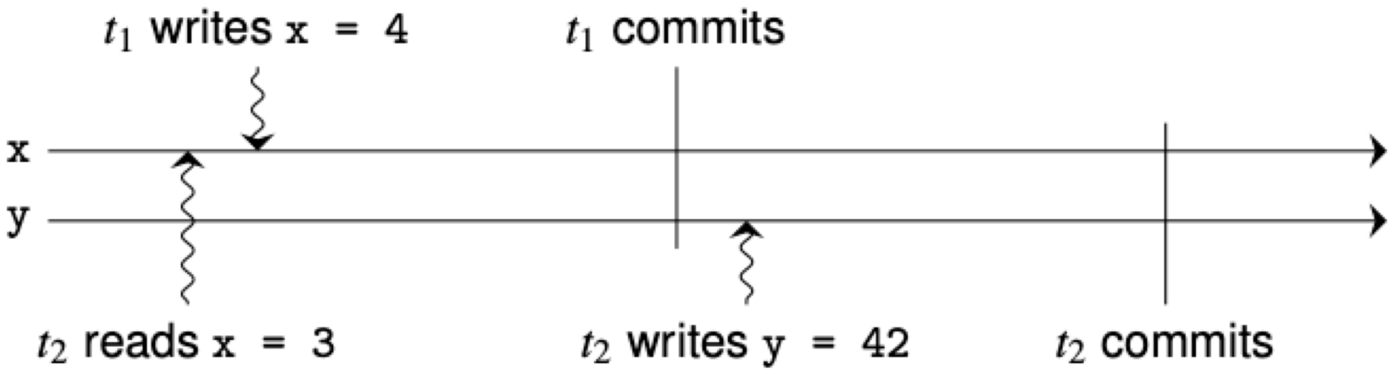
```
// read tuple t
do
    v1 = t.read_TID_word()
    RS[t.key].data = t.data
    v2 = t.read_TID_word()
while (v1 != v2 or v1.lock_bit == 1);
```

TicToc (SIGMOD 2016)



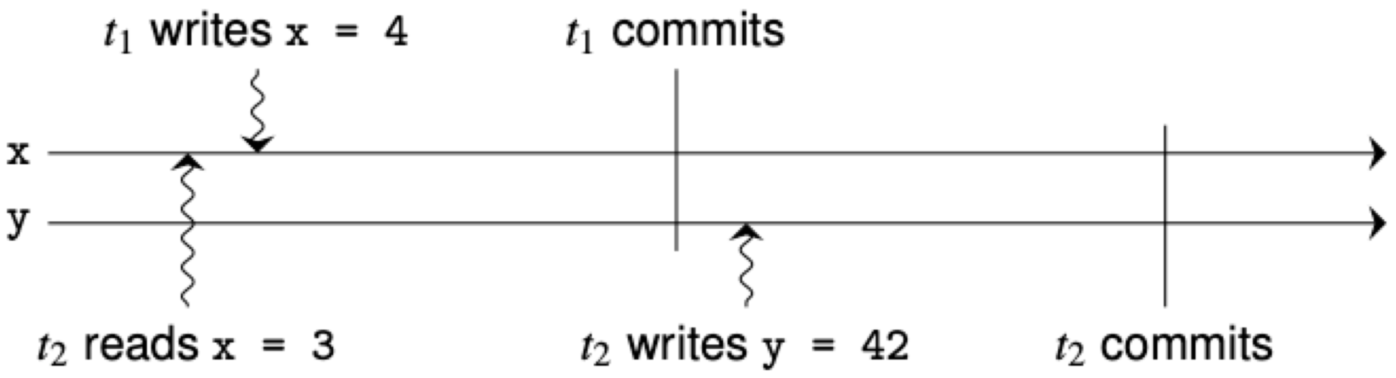
- In the schedule above, existing OCC protocols (including Silo) would abort transaction T2 since its validation of “read x” will fails

TicToc (SIGMOD 2016)



- In the schedule above, existing OCC protocols (including Silo) would abort transaction T2 since its validation of “read x” will fails
- But serializability is not violated if we order T2 before T1

TicToc (SIGMOD 2016)



- In the schedule above, existing OCC protocols (including Silo) would abort transaction T2 since its validation of "read x" will fails
- But serializability is not violated if we order T2 before T1
- **Key idea: dynamically determine the order of transactions based on the data access pattern**
- The determined logical order can be different from the physical time order

TicToc — Record Layout

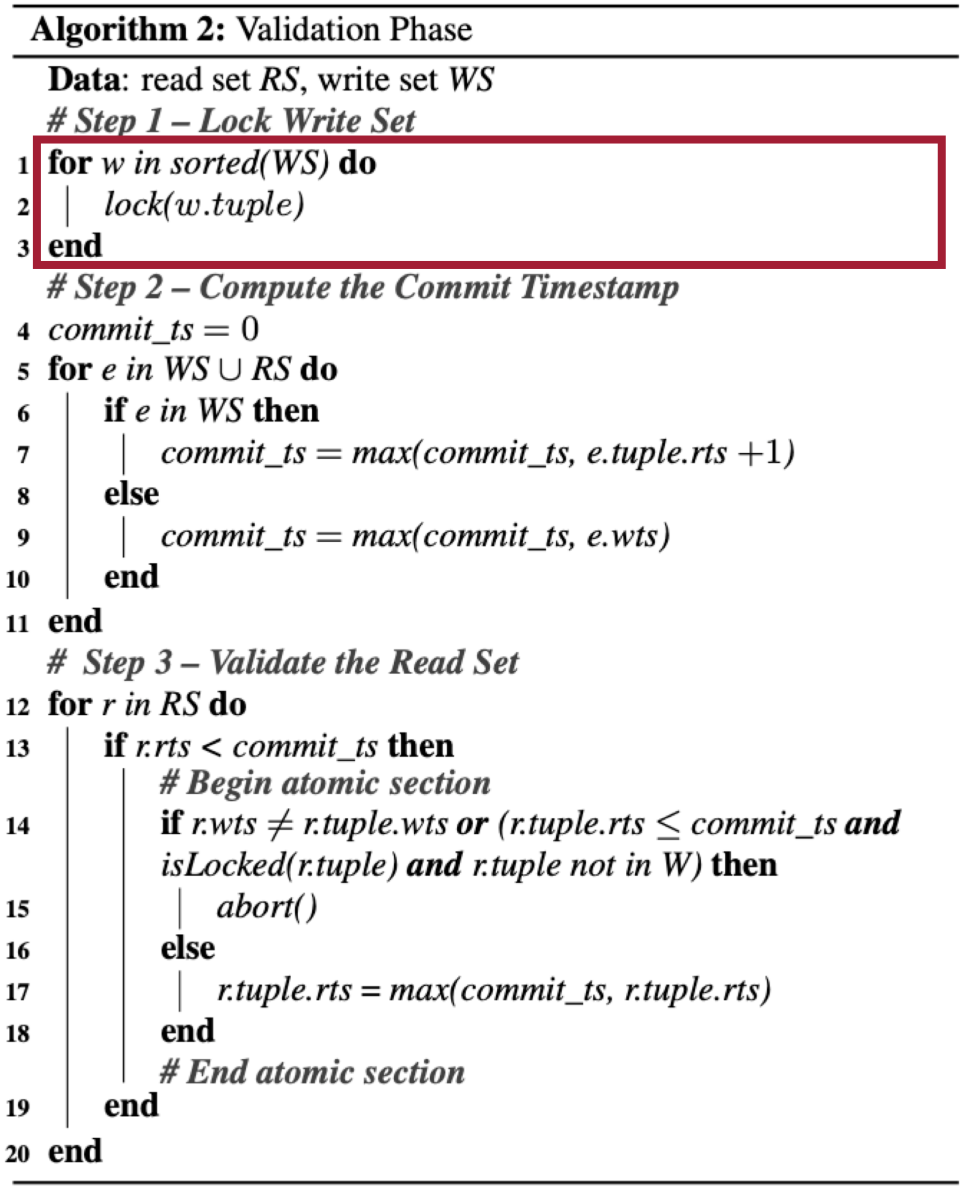
Each tuple contains a locking bit and two timestamps

Locking bits	wts (write timestamp)	rts (read timestamp)

For a read: load the timestamps together with the tuple data

The timestamps are updated during validation and write phases

TicToc — Validation Phase

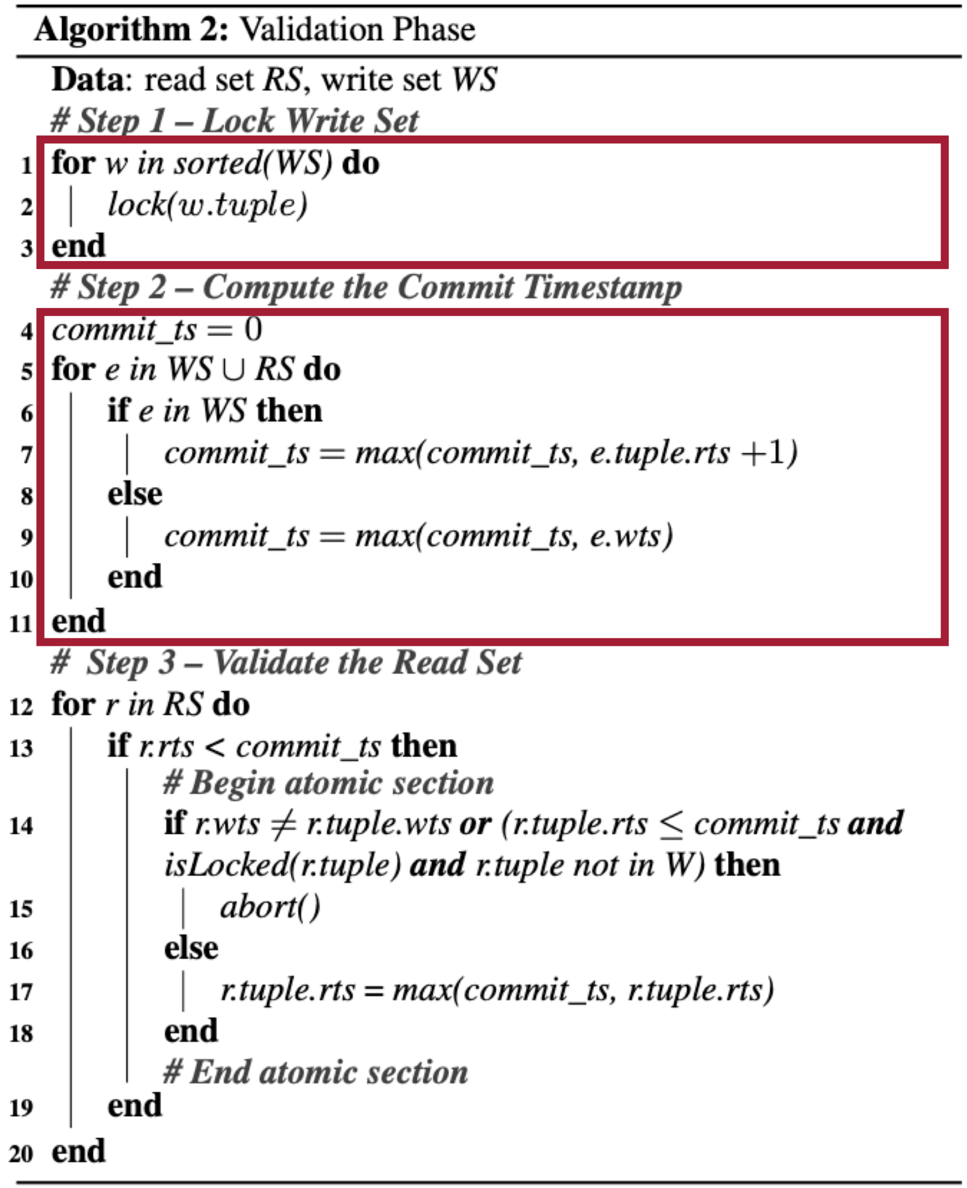


Algorithm 2: Validation Phase

```
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 $WS \cup RS$ 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 $r.wts \neq r.tuple.wts$ or (r.tuple.rts $\leq$ 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
```

Phase 1: Lock the write set

TicToc — Validation Phase



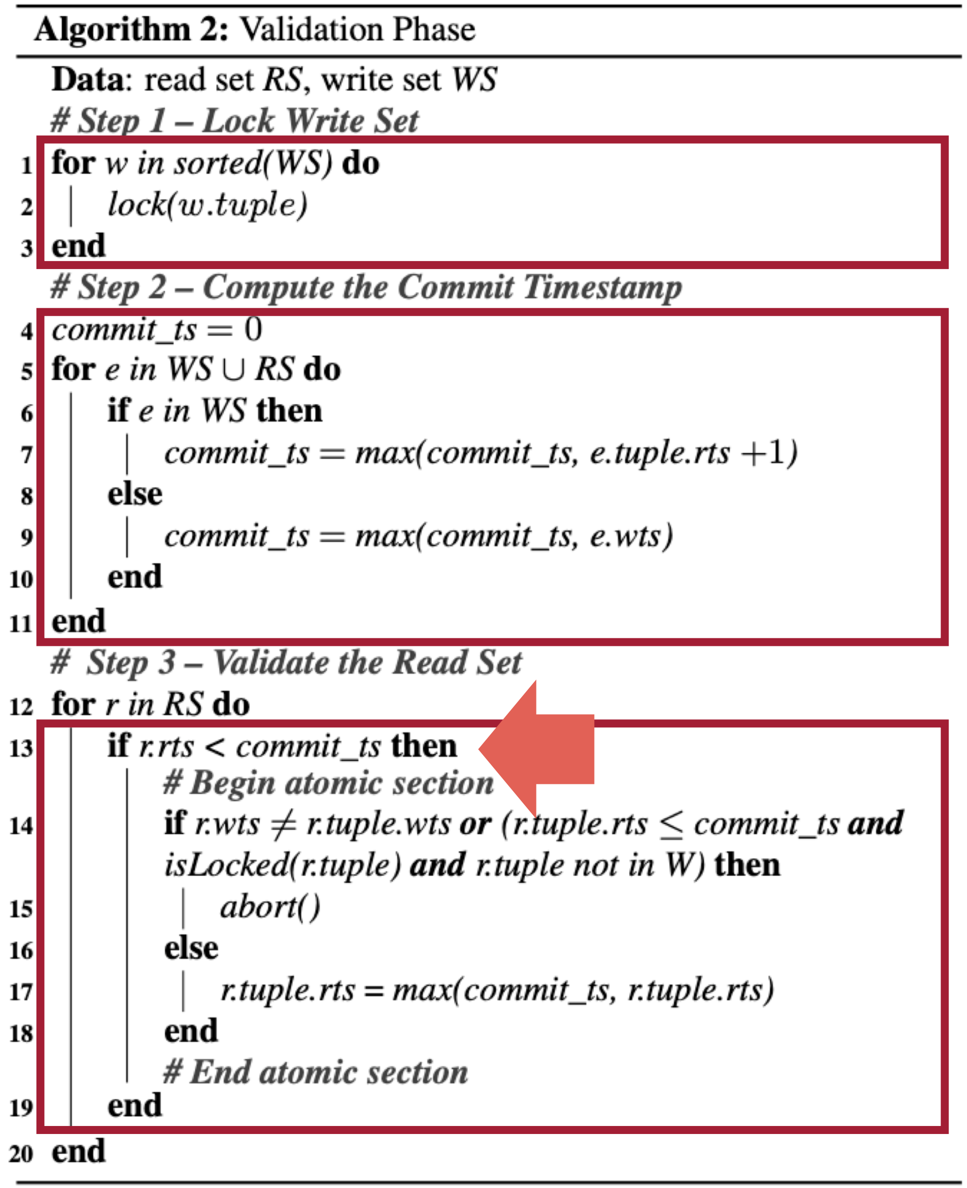
Algorithm 2: Validation Phase

```
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 $WS \cup RS$ 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 $r.wts \neq r.tuple.wts$ or ($r.tuple.rts \leq commit\_ts$ 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
```

Phase 1: Lock the write set

Phase 2: Compute the commit timestamp

TicToc — Validation Phase



Algorithm 2: Validation Phase

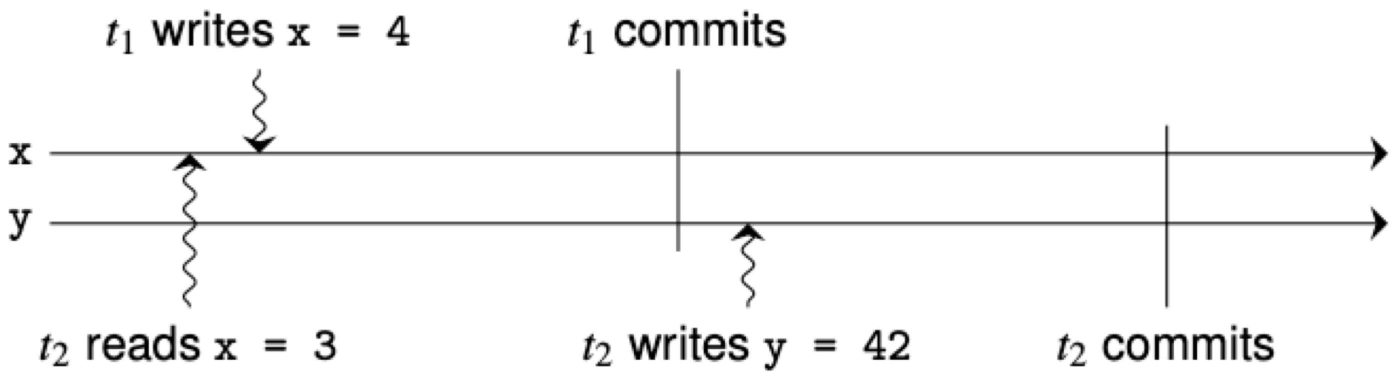
```
Data: read set RS, write set WS
# Step 1 – Lock Write Set
for w in sorted(WS) do
    lock(w.tuple)
end
# Step 2 – Compute the Commit Timestamp
commit_ts = 0
for e in WS ∪ RS do
    if e in WS then
        commit_ts = max(commit_ts, e.tuple.rts +1)
    else
        commit_ts = max(commit_ts, e.wts)
    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
```

Phase 1: Lock the write set

Phase 2: Compute the commit timestamp

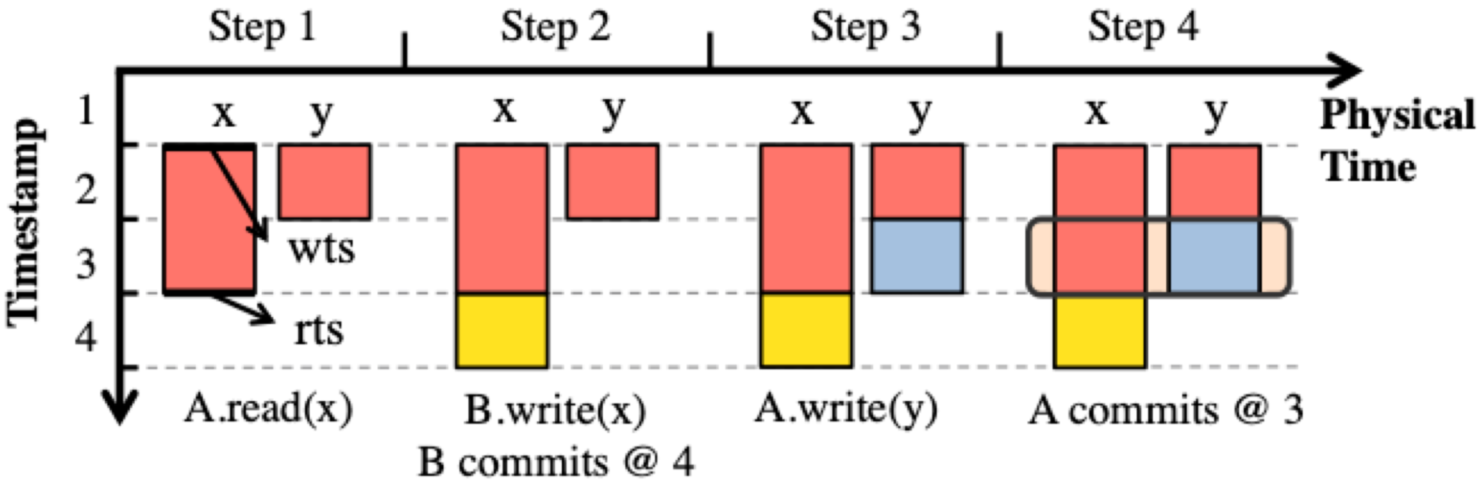
Phase 3: Validate the read set

Silo vs. TicToc

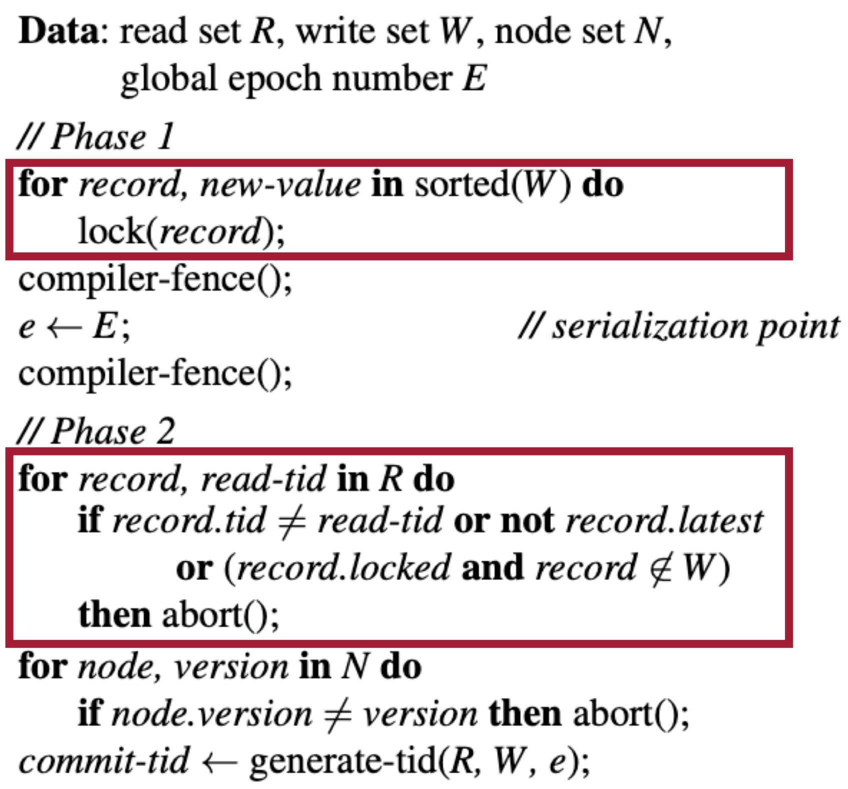


Silo aborts T2

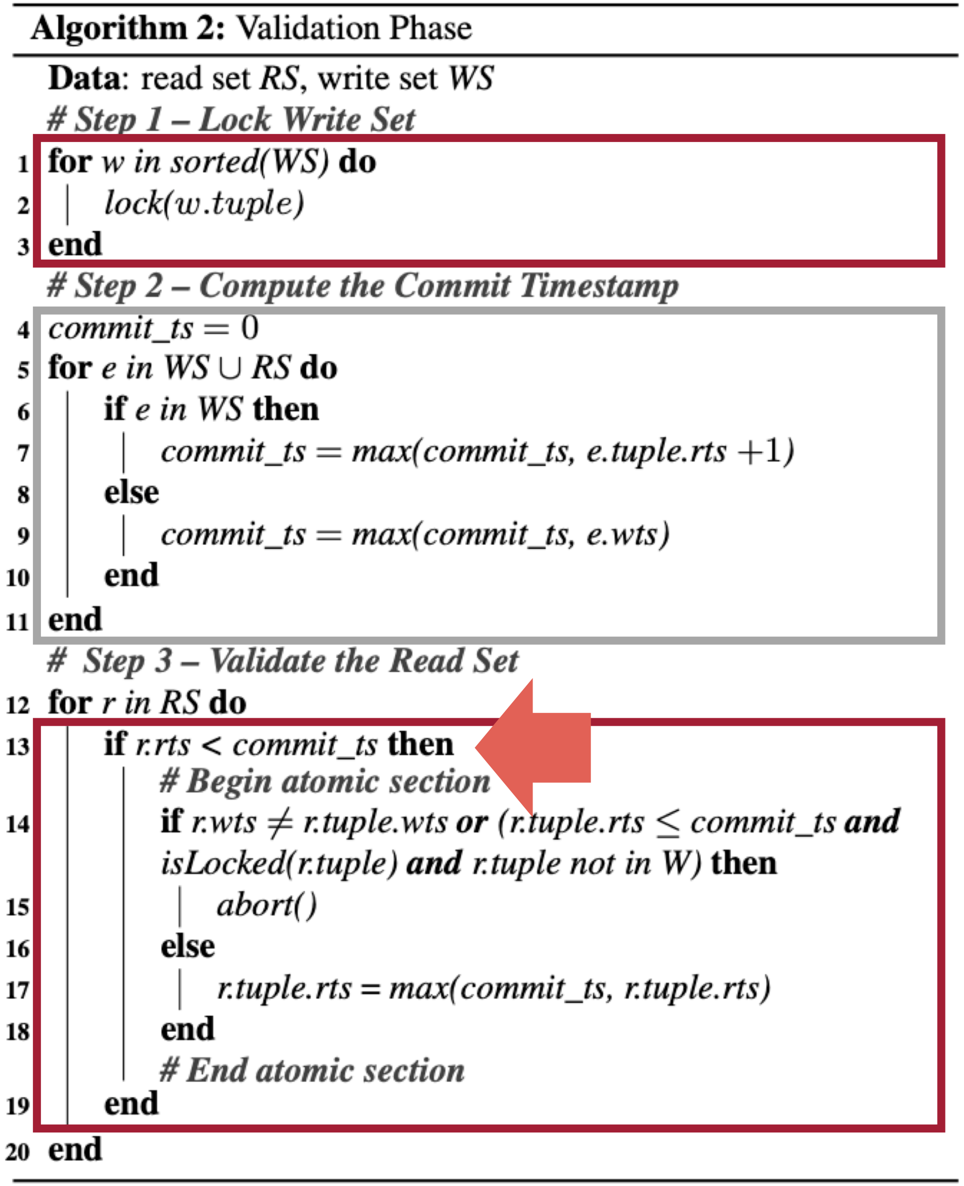
TicToc may commit both transactions



Silo vs. TicToc



```
Data: read set R, write set W, node set N,
      global epoch number E
// Phase 1
for record, new-value in sorted(W) do
    lock(record);
compiler-fence();
$e \leftarrow E$;                          // serialization point
compiler-fence();
// Phase 2
for record, read-tid in R do
    if record.tid $\neq$ read-tid or not record.latest
            or (record.locked and record $\notin W$)
    then abort();
for node, version in N do
    if node.version $\neq$ version then abort();
commit-tid $\leftarrow$ generate-tid(R, W, e);
```



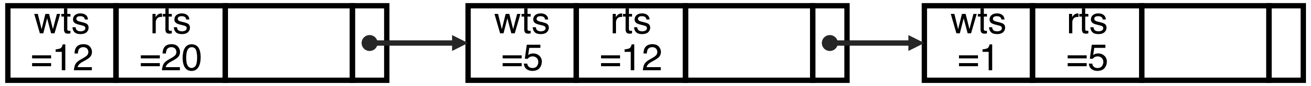
```
Algorithm 2: Validation Phase
Data: read set RS, write set WS
# Step 1 – Lock Write Set
for w in sorted(WS) do
    lock(w.tuple)
end
# Step 2 – Compute the Commit Timestamp
commit_ts = 0
for e in $WS \cup RS$ do
    if e in WS then
        commit_ts = max(commit_ts, e.tuple.rts +1)
    else
        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 $\neq$ r.tuple.wts or (r.tuple.rts $\leq$ 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
```

Phase 1 is identical

Main difference is in the validation phase

Multi-Version Concurrency Control

Version chain

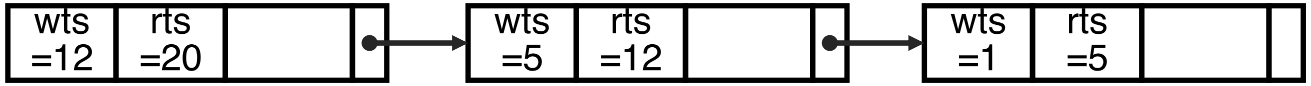


Acquire a timestamp at the beginning of the transaction

Use the allocated timestamp to determine which version to read

Multi-Version Concurrency Control

Version chain



Acquire a timestamp at the beginning of the transaction

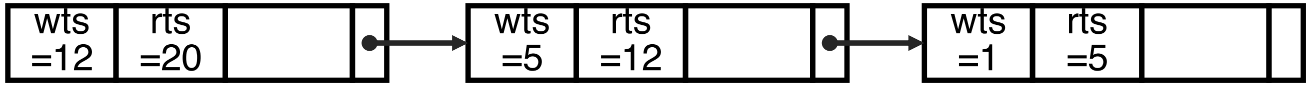
Use the allocated timestamp to determine which version to read

Advantages

- Read-only transaction can read slightly stale data to avoid conflicts
- An early read does not conflict with a later write

Multi-Version Concurrency Control

Version chain



Acquire a timestamp at the beginning of the transaction

Use the allocated timestamp to determine which version to read

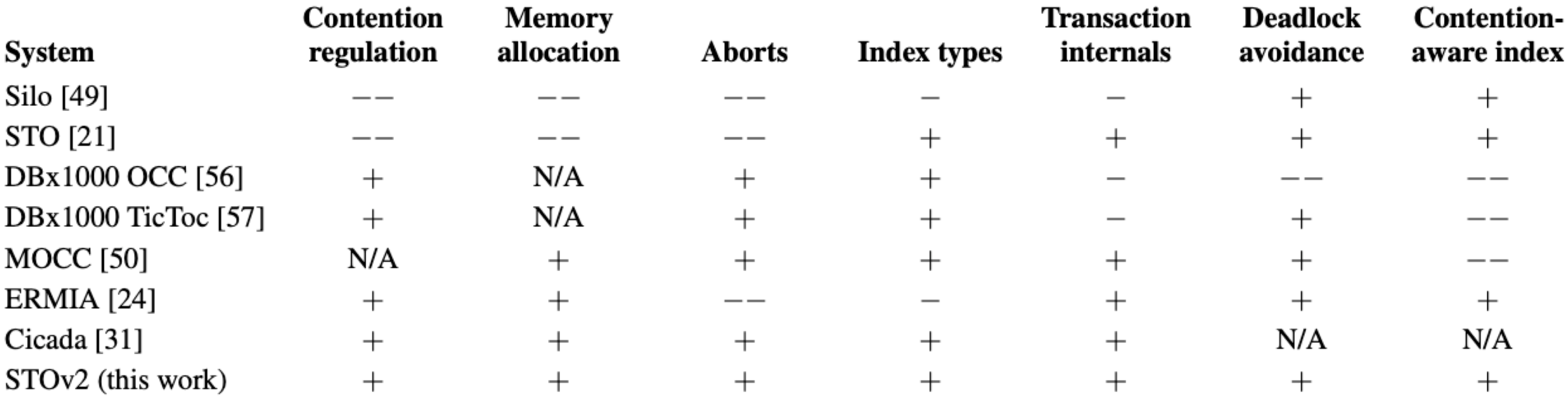
Advantages

- Read-only transaction can read slightly stale data to avoid conflicts
- An early read does not conflict with a later write

Disadvantages

- Overhead of managing multiple versions (e.g., garbage collect)

Basis Factors



System	Contention regulation	Memory allocation	Aborts	Index types	Transaction internals	Deadlock avoidance	Contention-aware index
Silo [49]	−−	−−	−−	−	−	+	+
STO [21]	−−	−−	−−	+	+	+	+
DBx1000 OCC [56]	+	N/A	+	+	−	−−	−−
DBx1000 TicToc [57]	+	N/A	+	+	−	+	−−
MOCC [50]	N/A	+	+	+	+	+	−−
ERMIA [24]	+	+	−−	−	+	+	+
Cicada [31]	+	+	+	+	+	N/A	N/A
STOv2 (this work)	+	+	+	+	+	+	+

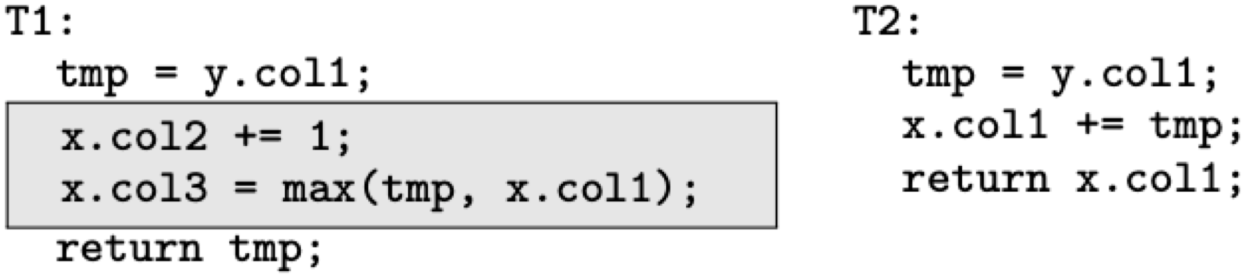
Different choices of basis factors have significant impact on performance

If not picking carefully, the effects of basis factors will hide the effects of concurrency control protocols

High-Contention Optimizations

Optimization 1: Commit-time updates

- Delay blind writes to the end of the transaction



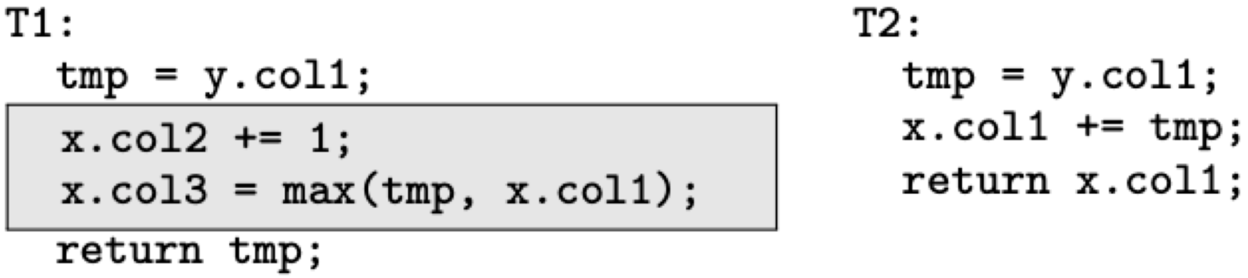
```
T1:
  tmp = y.col1;
  x.col2 += 1;
  x.col3 = max(tmp, x.col1);
  return tmp;
```

```
T2:
  tmp = y.col1;
  x.col1 += tmp;
  return x.col1;
```

High-Contention Optimizations

Optimization 1: Commit-time updates

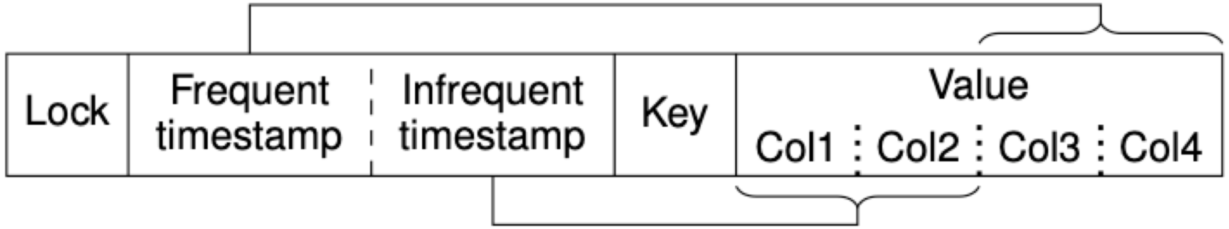
- Delay blind writes to the end of the transaction



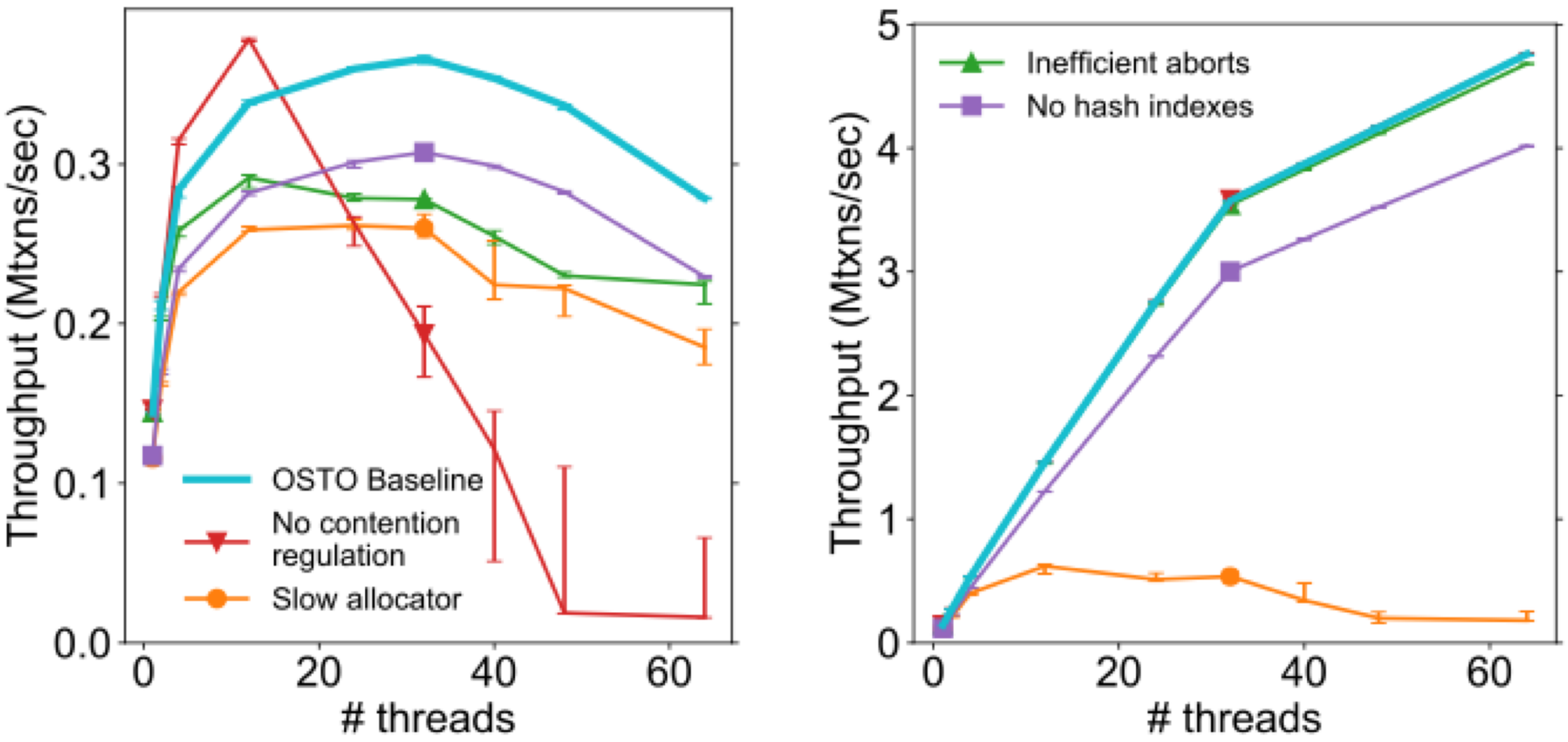
```
T1:
  tmp = y.col1;
  x.col2 += 1;
  x.col3 = max(tmp, x.col1);
  return tmp;

T2:
  tmp = y.col1;
  x.col1 += tmp;
  return x.col1;
```

Optimization 2: Timestamp splitting

- Use different timestamps to manage different attributes (similar to field-level locking)

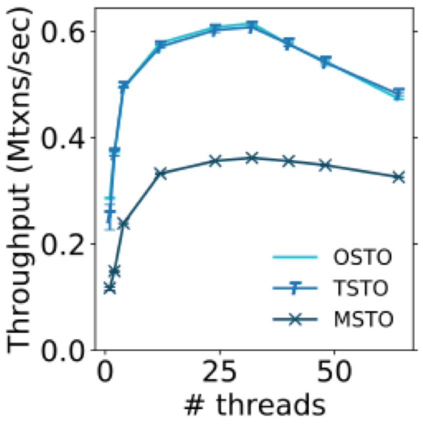


Evaluation – Effects of Basis Factors

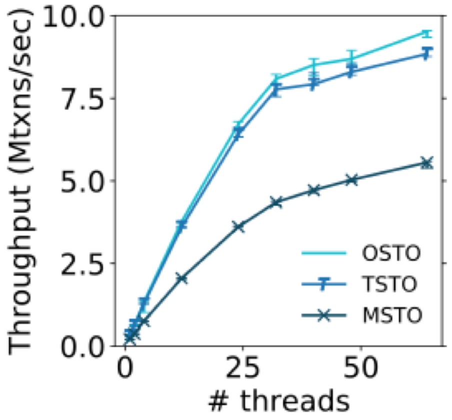


Different choices of basis factors have significant impact on performance

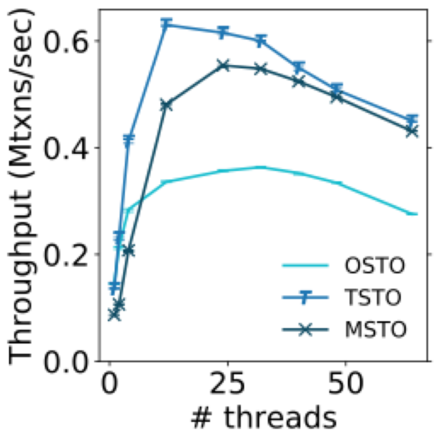
Evaluation – Performance Overview



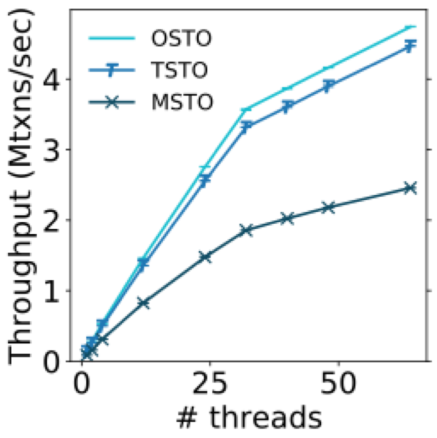
(c) *YCSB-A (high contention: update-intensive, 50% updates, skew 0.99).*



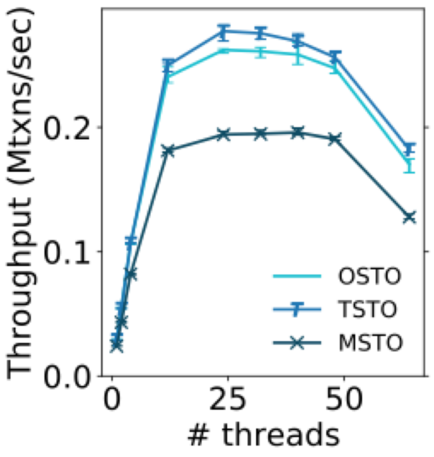
(d) *YCSB-B (lower contention: read-intensive, 5% updates, skew 0.8).*



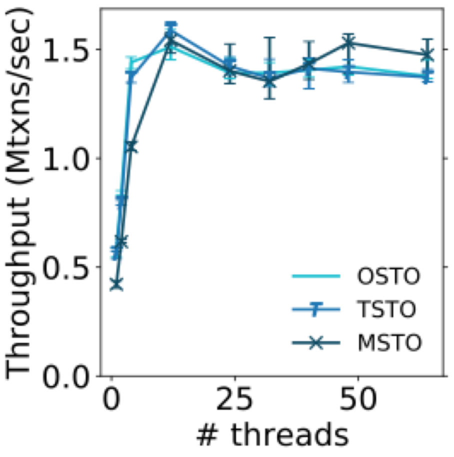
(a) *TPC-C, one warehouse (high contention).*



(b) *TPC-C, one warehouse per worker (low contention).*



(e) *Wikipedia (high contention).*

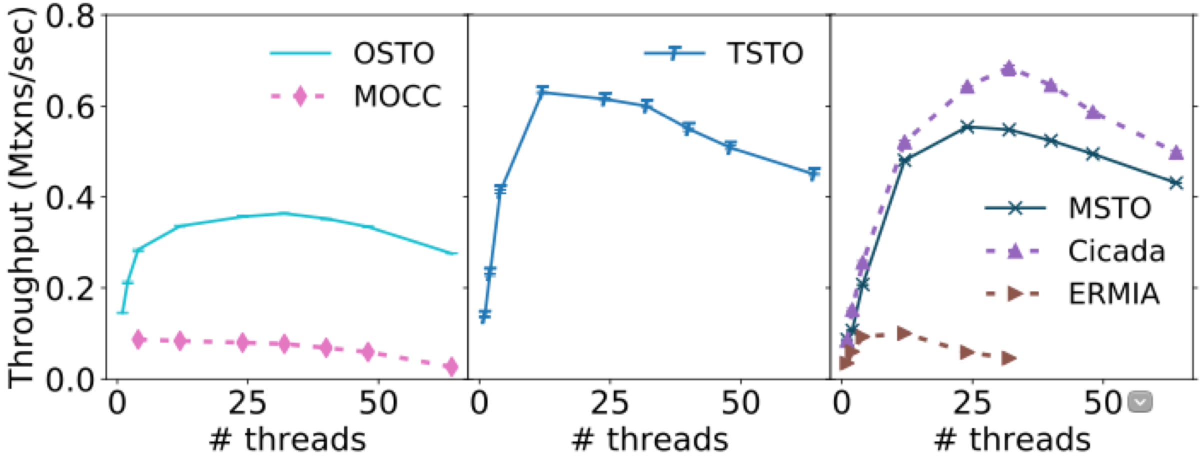


(f) *RUBiS (high contention).*

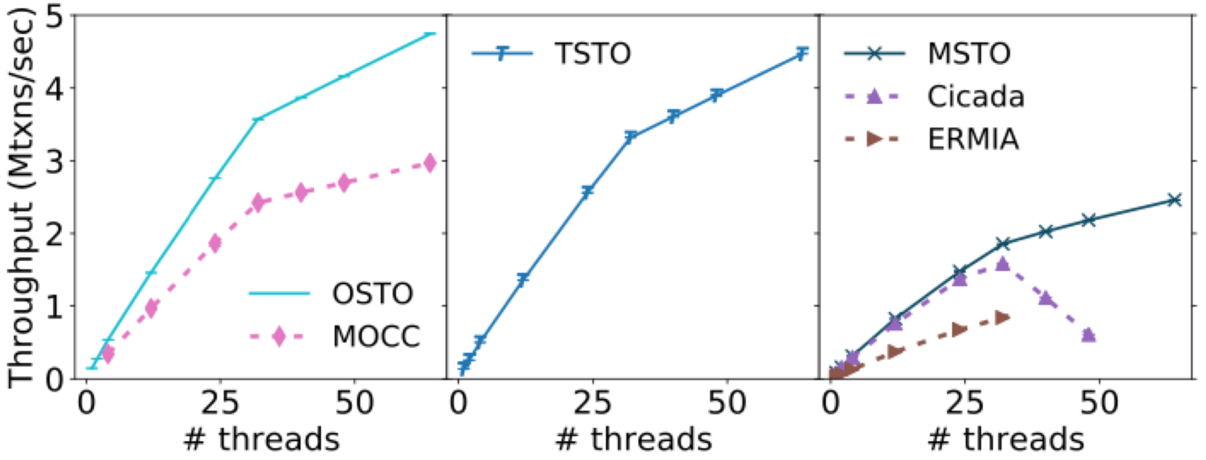
Performance gap between OSTO (Silo) and TSTO (TicToc) is small except for high-contention TPC-C

MVCC has worse performance at low contention due to overhead

Evaluation – Cross-System Comparison

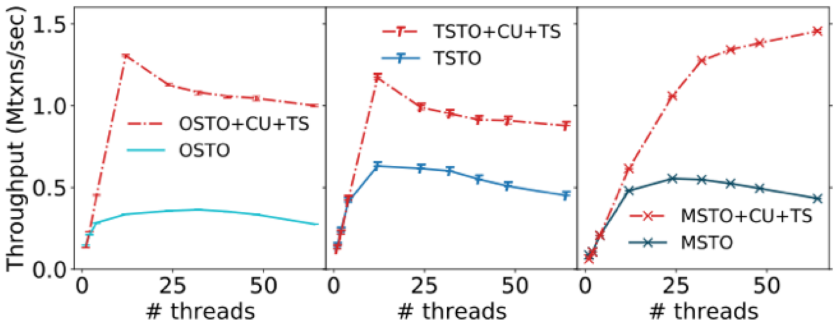


(a) *TPC-C, one warehouse (high contention).*

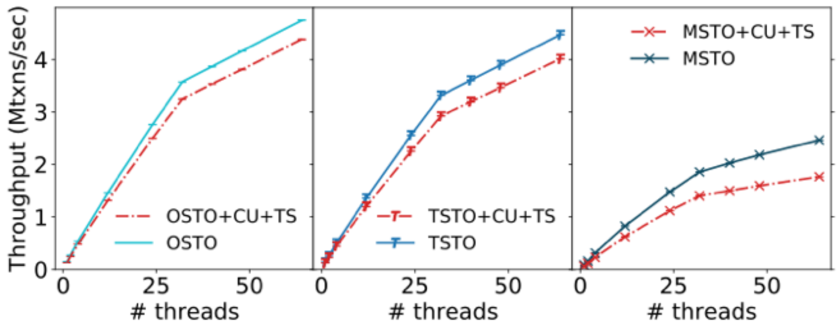


(b) *TPC-C, one warehouse per worker (low contention).*

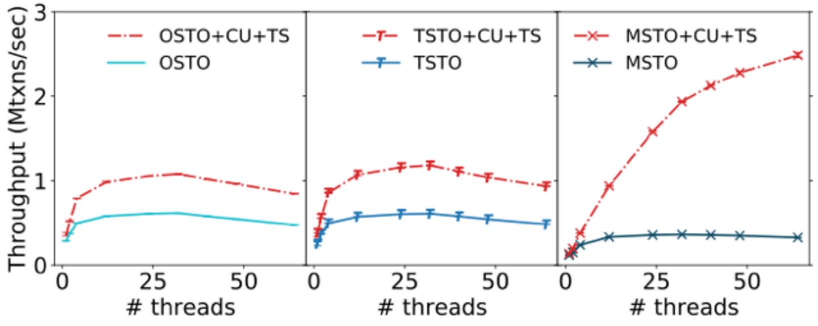
Evaluation – Optimizations



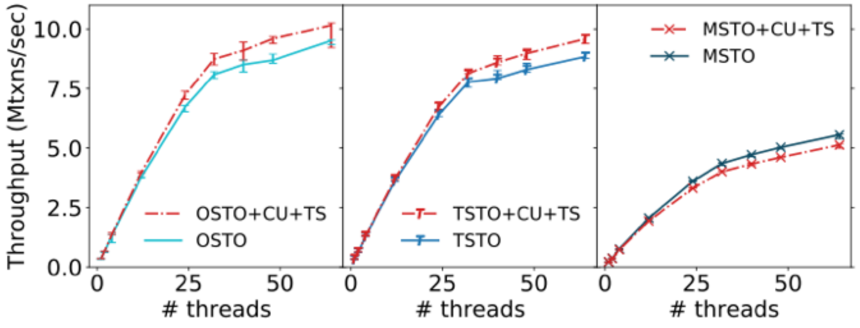
(a) *TPC-C, one warehouse (high contention).*



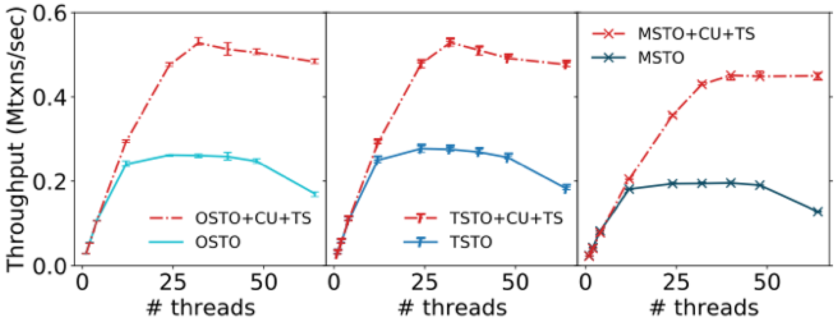
(b) *TPC-C, one warehouse per worker (low contention).*



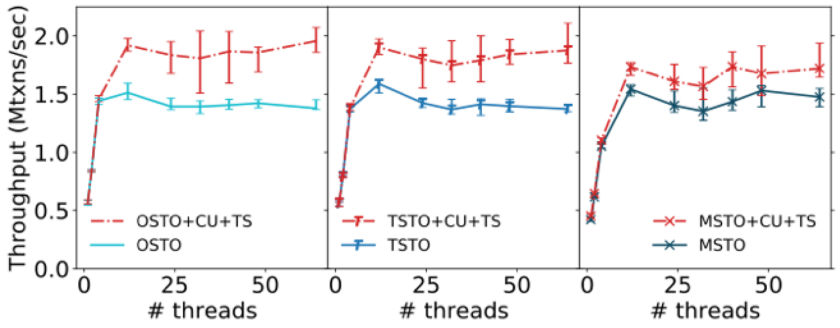
(c) *YCSB-A (high contention: update-intensive, 50% updates, skew 0.99).*



(d) *YCSB-B (lower contention: read-intensive, 5% updates, skew 0.8).*



(e) *Wikipedia (high contention).*



(f) *RUBiS (high contention).*

The optimizations improve performance for all protocols at high contention, especially for MVCC

Q/A – Modern OCC

How do updaters improve the performance of read-modify-write?

What's the intuition behind the following claim?

- "OCC can perform surprisingly well even under high contentions on multi-core main memory systems."

Why need to lock the entire write set?

We focus too much on experimental results nowadays

Overfitting to the studied workloads?

Why these basis factors not found out in previous papers?

How to automate the two optimizations?

Next Lecture

Submit review for

- Clemens Lutz, et al. Pump Up the Volume: Processing Large Data on GPUs with Fast Interconnects, SIGMOD 2020 (**best paper award**)