High-Performance Concurrency Control Mechanisms for Main-Memory Databases

Per-Åke Larson[1], Spyros Blanas[2], Cristian Diaconu[1], Craig Freedman[1], Jignesh M. Patel[2], Mike Zwilling[1]

[1]Microsoft Corporation [2]Univ. of Wisconsin-Madison

The problem

Most DBMSs designed for:

- Disk-resident data
- Few CPUs

$50K server in 2012:

- 1TB of RAM
- 40 CPUs

What concurrency control scheme should be used for a high-performance main-memory OLTP system?

Contributions

1. Multi-version optimistic concurrency control
   - Multi-version: readers don't block writers
   - Optimistic: no waiting on database locks
   - Supports all SQL isolation levels
2. Efficient mechanisms for implementing multi-version and single-version locking
3. Experimental evaluation: High performance (millions of TX/sec) and full serializability without workload-specific knowledge

Recent related work

Our approach:

Redesign DBMS storage engine,
make no assumption about workload

- Make existing DBMS storage engine scale:
  - Locking, page latching, B-tree index, logging, …
- Exploit specific workload property:
  - Partitionable workload
  - Deterministic stored procedures

Designing a main memory storage engine

Traditional disk-oriented engine

- Disk-friendly data structures
  - Pages, B-tree index
- Absorbs high disk latency by frequent context switching
- Thread spins for latches
- TX may yield for locks
- Critical sections are thousands of instructions long, and limit scalability

Our main memory prototype

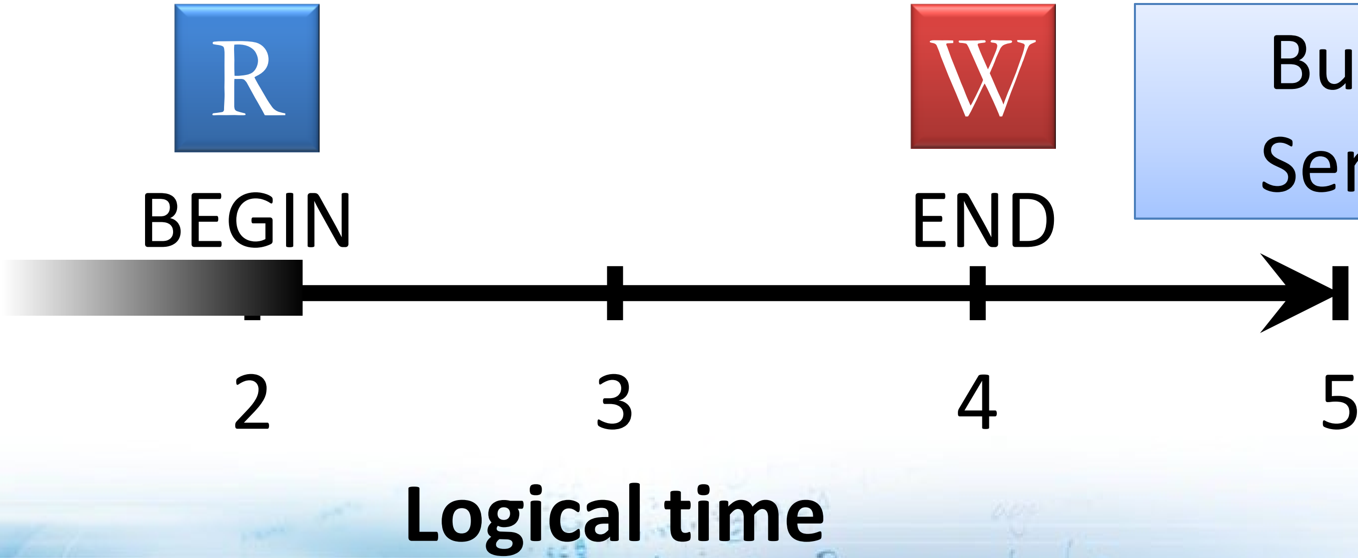
- Latch-free hash table stores individual records
- Minimizes context switching
  - Usually 1, at most 2 per TX
- Eliminates latches
- TX never waits for locks
- No critical sections
  - Many TXs finish in thousands of instructions

Multi-version optimistic scheme
Snapshot Isolation (SI)

- TXs have two unique timestamps: BEGIN, END
- Read as of BEGIN timestamp
- Write as of END timestamp

Sufficient for Read Committed

But not for Serializable



BEGIN
END

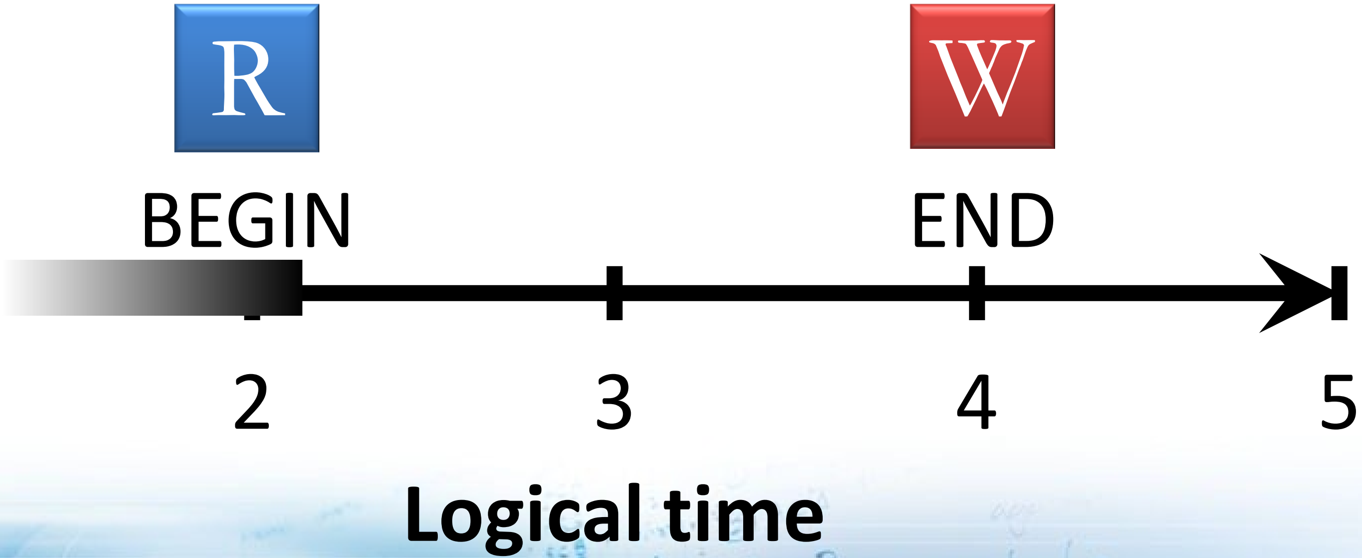
2
3
4
5

Logical time

Making SI serializable

[Bornea et al, ICDE’11]

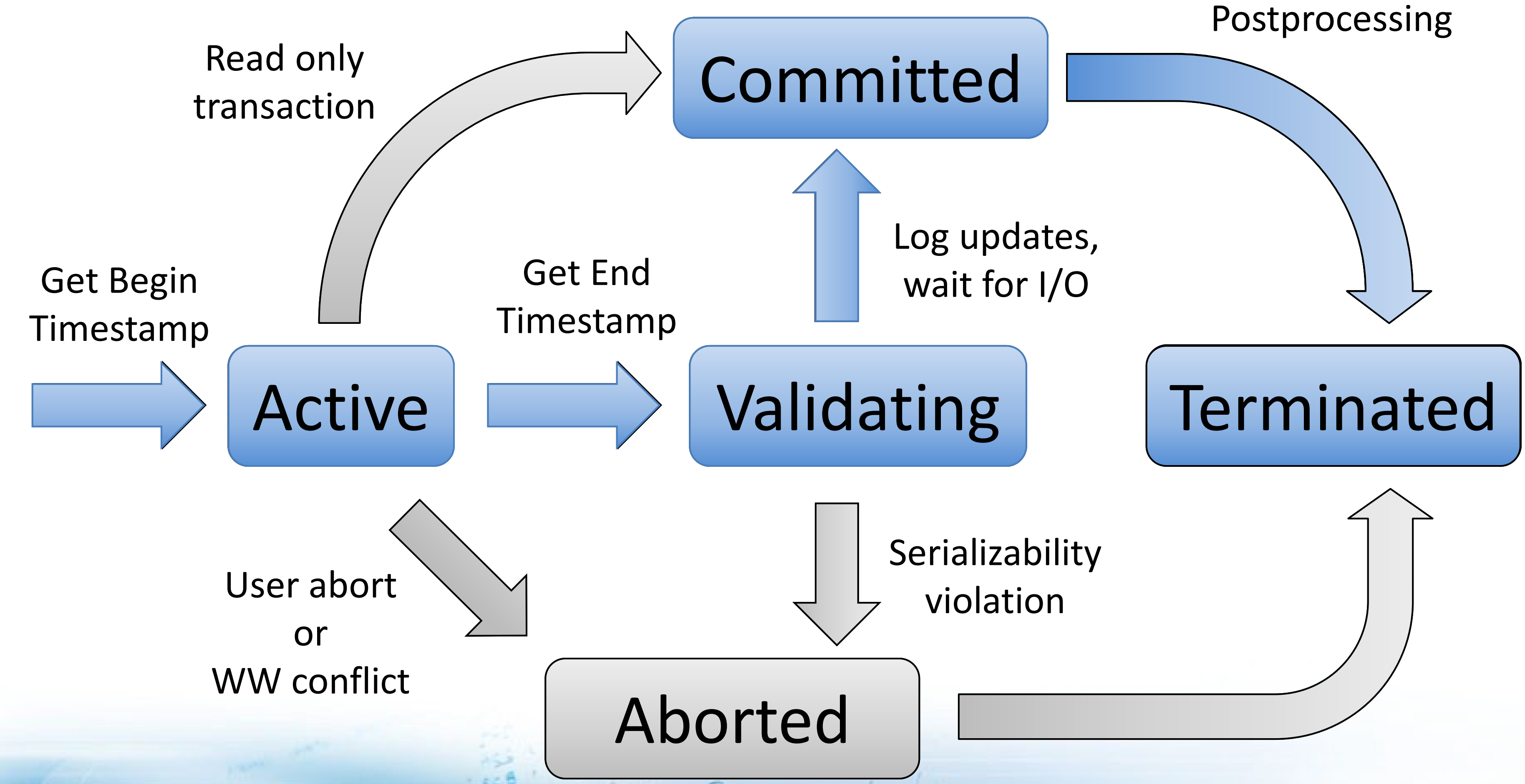
- Read as of BEGIN timestamp
- Repeat Read as of END timestamp, verify no change
- Write as of END timestamp



What needs to be repeated?

- Depends on the isolation level
- Read Committed or SI: No validation needed
  - Versions were committed at BEGIN, will still be committed at END
- Repeatable Read: Read versions again
  - Ensure no versions have disappeared from the view
- Serializable: Repeat scans with same predicate
  - Ensure no phantoms have appeared in the view

Transaction states



Read only transaction

Committed

Postprocessing

Get Begin Timestamp

Active

Get End Timestamp

Validating

Log updates, wait for I/O

Terminated

User abort
or
WW conflict

Serializability violation

Aborted

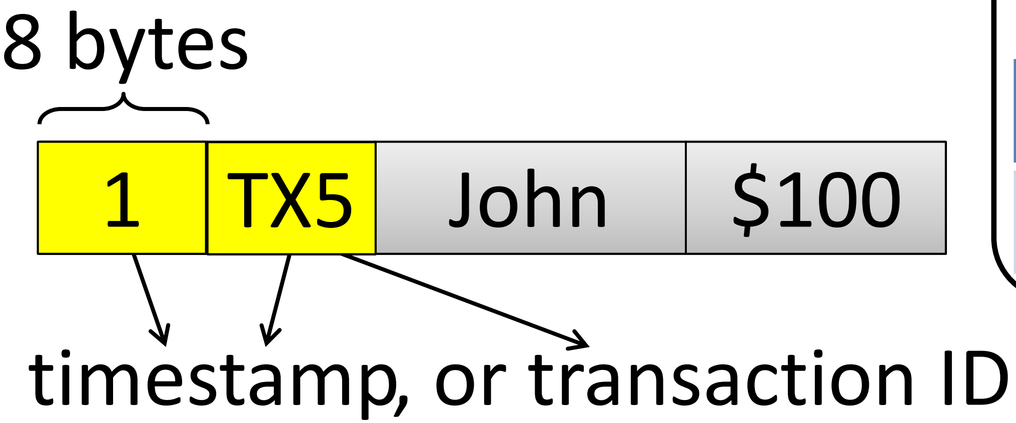
Transaction map

- Stores transaction state, timestamps
- Globally visible

TRANSACTION MAP

TXID	STATE	BEGIN	END
5	ACTIV	2	N/A

Determining version visibility



timestamp, or transaction ID

TRANSACTION MAP

TXID	STATE	BEGIN	END
5	ACTIV	2	N/A

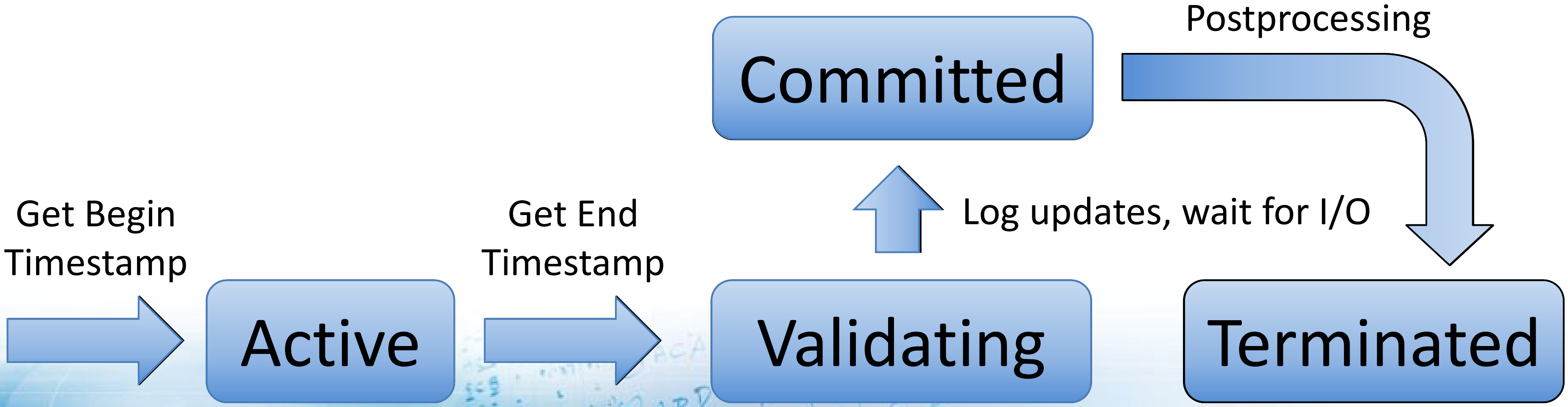
Visibility as of time T is determined by:
version timestamps and TX state

Example: Update to $150

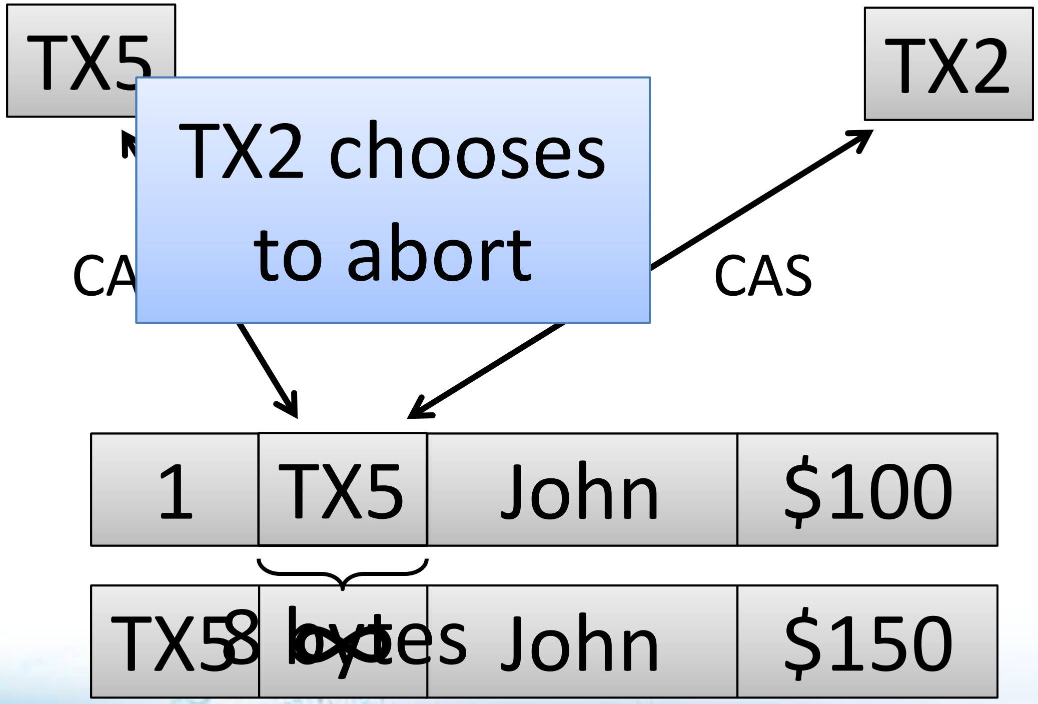
1	4	John	$100
4	∞	John	$150

TRANSACTION MAP

TXID	STATE	BEGIN	END



WW conflicts



First writer wins

TX2 updates $100 to $75

TX5

TX2

TX2 chooses to abort

CA

CAS

1	TX5	John	$100
TX5		John	$150

8 bytes

WR conflicts

TX5	∞	John	$150

Q: When is version visible?

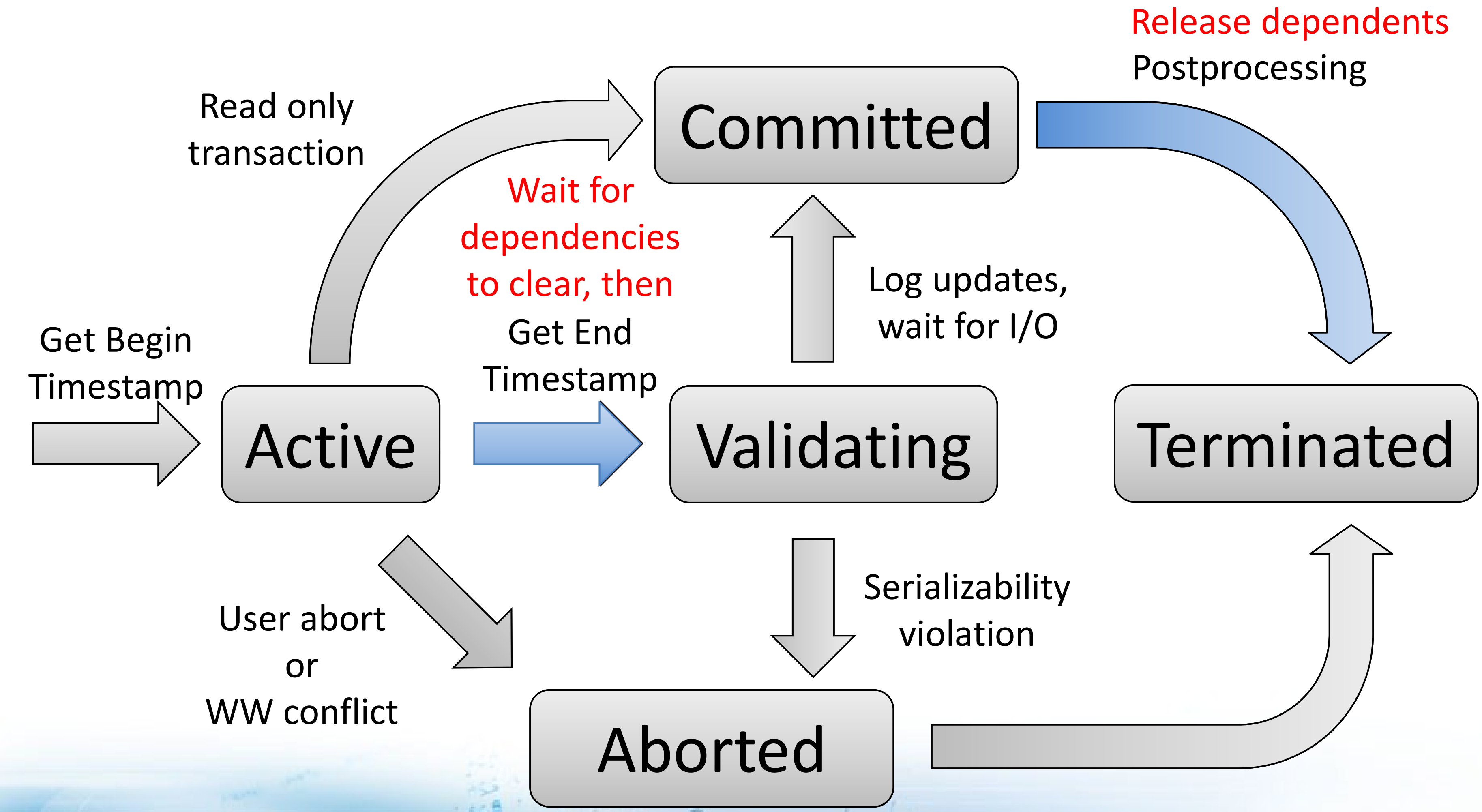
A: Depends on TX state

TX5 State	Visible?
ACTIVE	No, version is uncommitted
VALIDATING	? Speculate YES now, confirm at end
COMMITTED	Maybe, check TX5 END timestamp
ABORTED	No, version is garbage

Commit dependencies

- Impose constraint on serialization order:
  Commit B only if A has committed.
- Implementation: register-and-signal
  - Transform multiple waits on every record access to a single wait at end of TX
  - Dependency wait time “added” to log latency
    - Most common: no wait needed, dependency has cleared
- But: Cascading aborts now possible

Commit dependencies



Release dependents
Postprocessing

Read only
transaction

Committed

Wait for
dependencies
to clear, then
Get End
Timestamp

Log updates,
wait for I/O

Get Begin
Timestamp

Active

Validating

Terminated

User abort
or
WW conflict

Serializability
violation

Aborted

Multi-version optimistic summary

- TXs never wait during the ACTIVE phase
- No deadlock detection is needed
- Lower isolation level = less work
  - Read Committed and SI: No validation at all

Multi-version locking

- Provides lock-like semantics:
  Once a version is read by T, it will remain visible to T until commit.
- No centralized lock table
  - Record lock embedded in version's END timestamp
- Same context switching overhead: At most 2 per TX

But:

- Deadlock detection necessary
- More write traffic, even readers write to memory

Implementation details

- Independent transaction kernel in C++
- Base data structure: latch-free hash table
    - Perfect sizing, perfect hashing
    - Load factor when idle: 1

Single-version two-phase locking

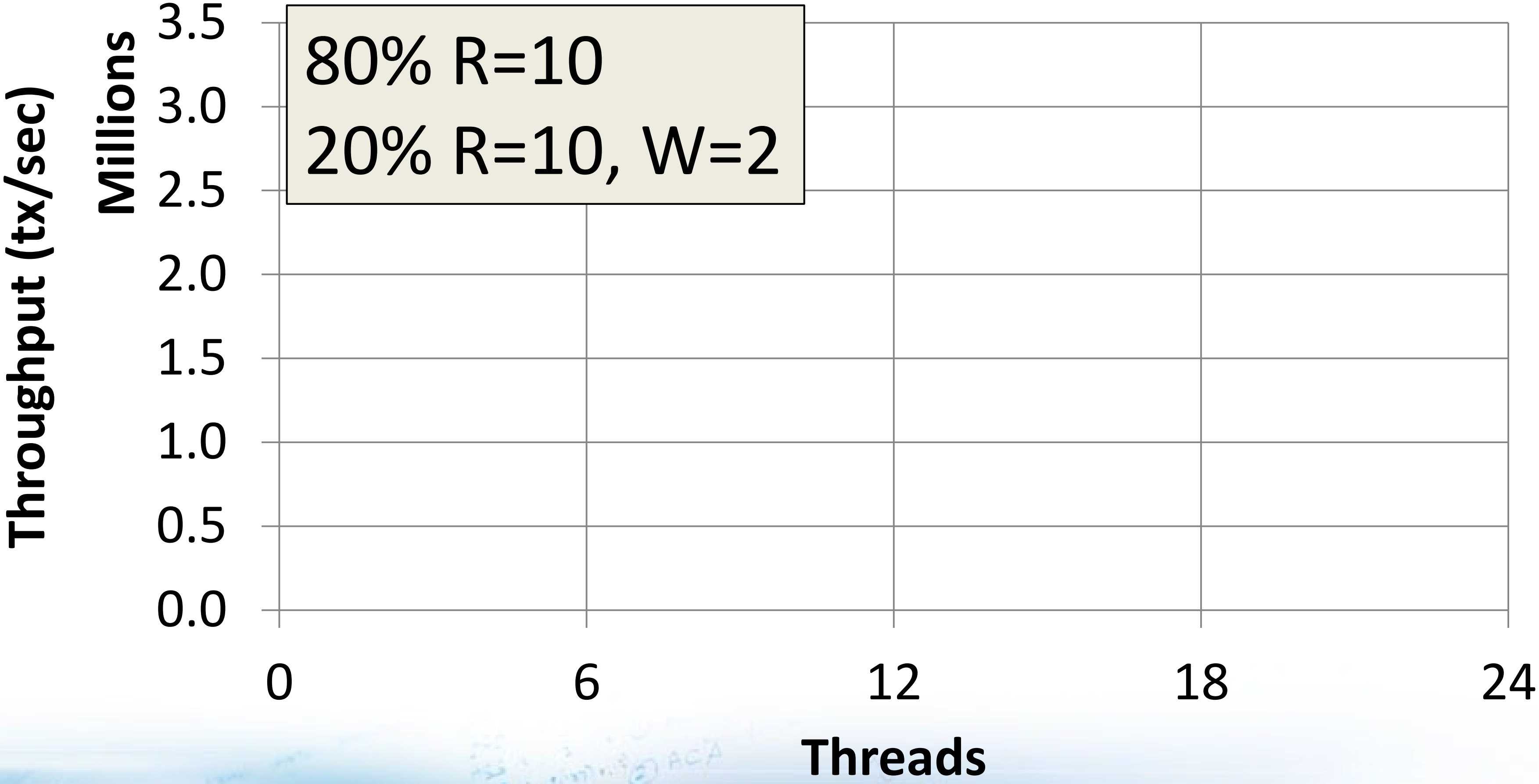
- Traditional 2PL, optimized for main memory
- No central lock manager
- Lock is pre-allocated in hash table bucket
  - Protects hash bucket, prevents phantoms
  - Multiple-reader, single-writer lock
  - For our experiments, also serves as a record lock

Experimental setup

- 2-socket × 6-core Xeon X5650 with 48GB RAM
- TXs don’t wait for the log (lazy commit)
  - Log records <u>are</u> populated and written to disk
- All transactions run under Serializable

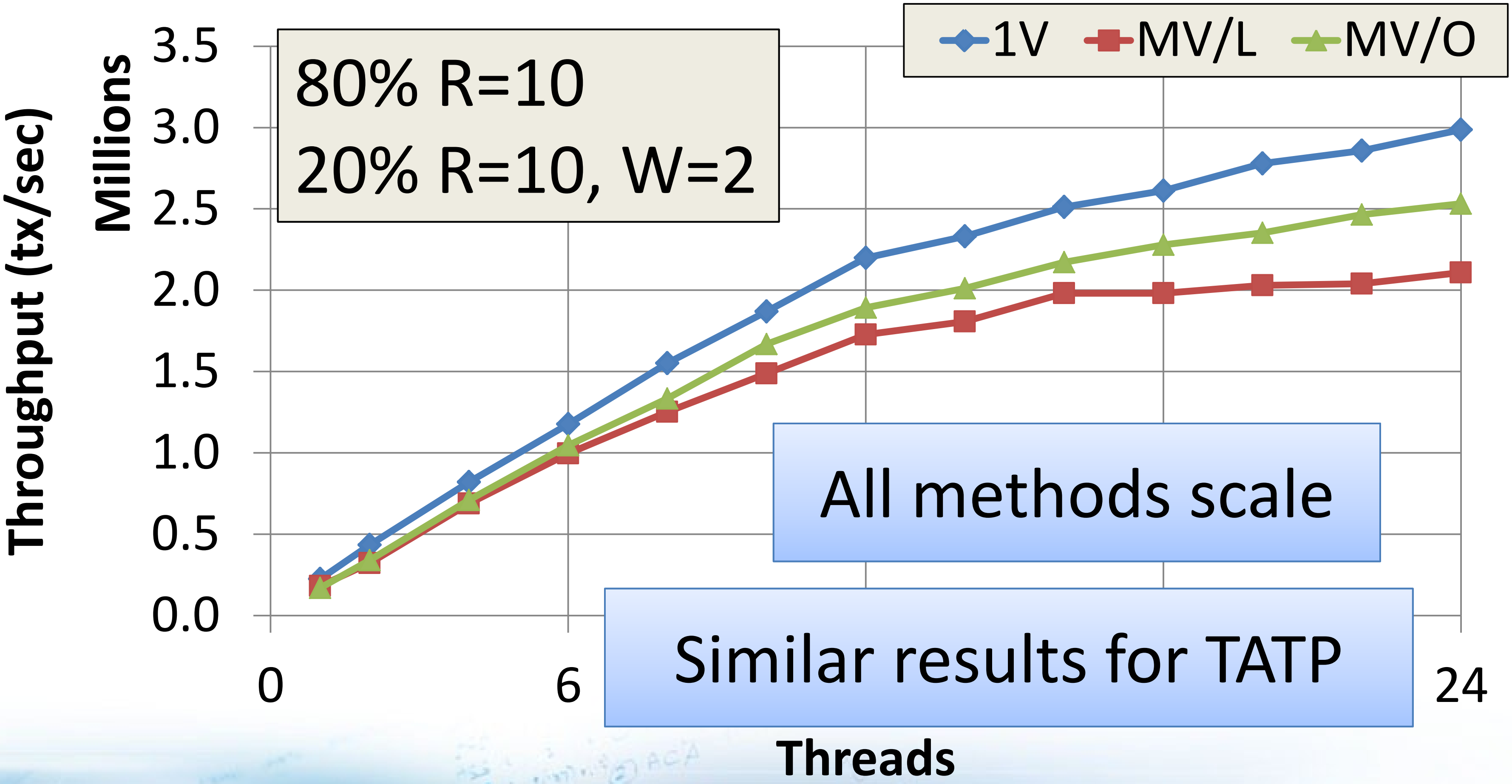
MV/O	Multi-version optimistic
MV/L	Multi-version locking
1V	Single-version two-phase locking

Scalability
No contention (10M row table)



80% R=10
20% R=10, W=2

Scalability
No contention (10M row table)

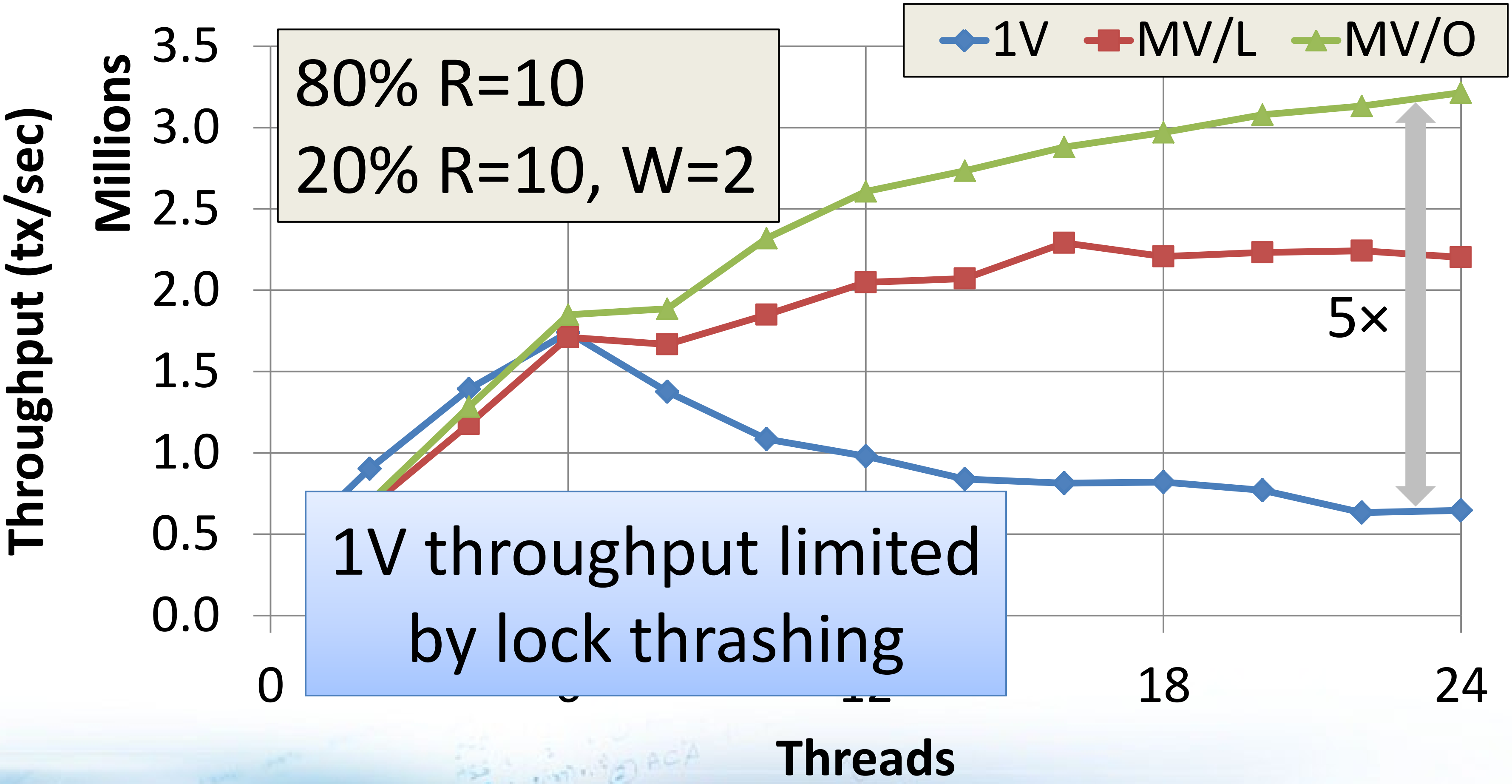


80% R=10
20% R=10, W=2

All methods scale

Similar results for TATP

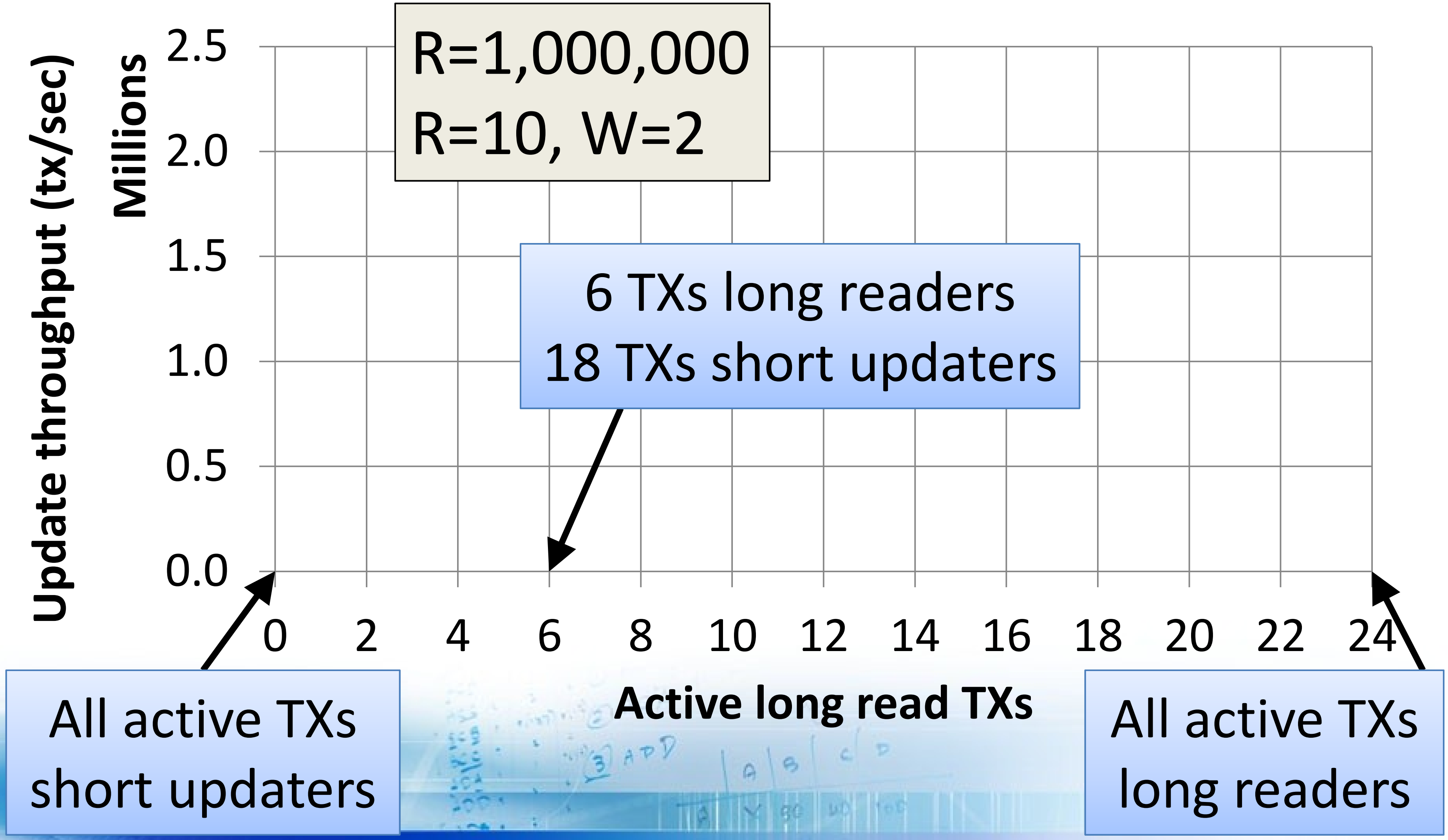
Scalability
Extreme contention (1000 row table)



80% R=10
20% R=10, W=2

1V throughput limited by lock thrashing

Effect of long readers (10M row table)



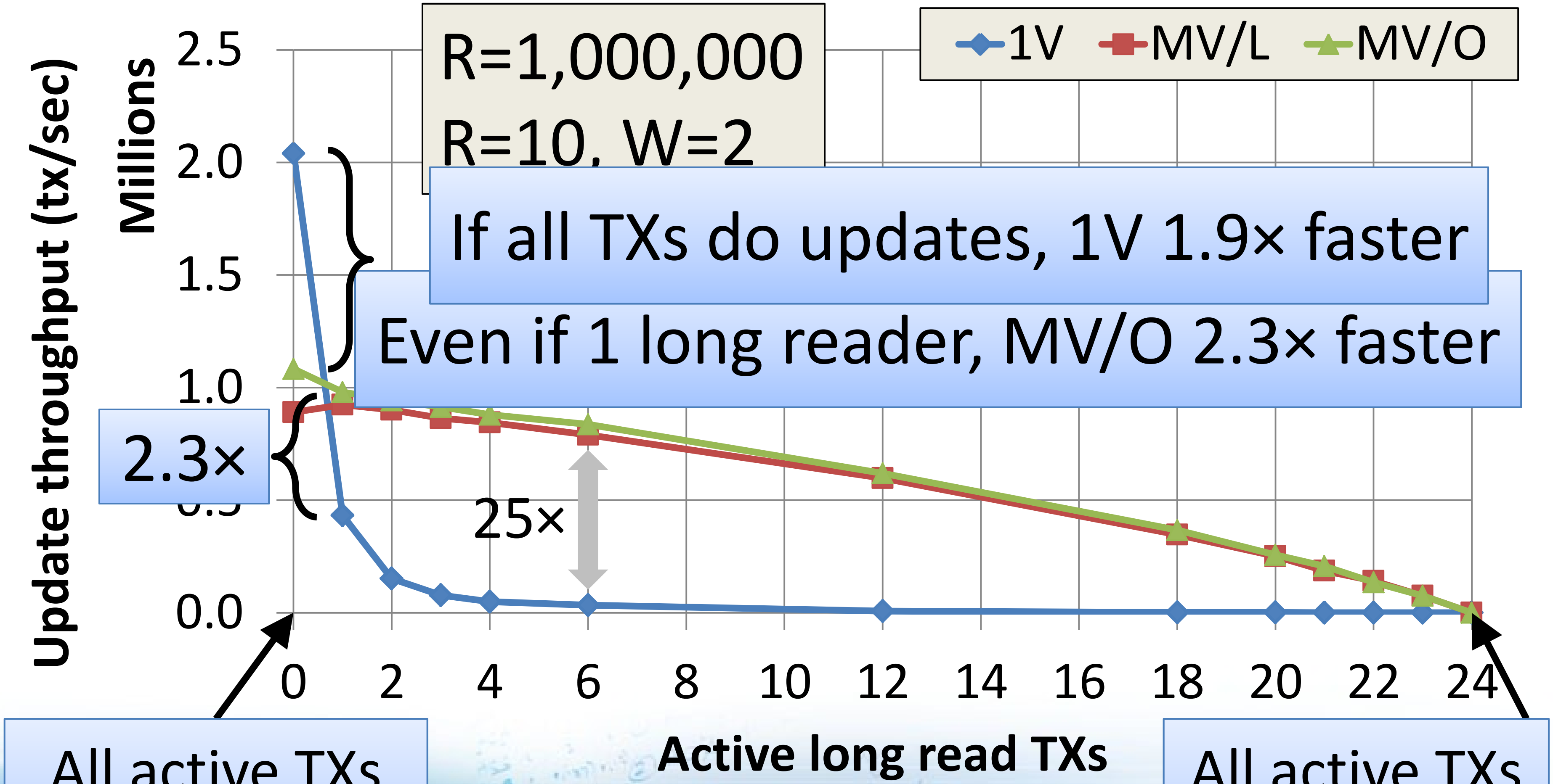
R=1,000,000
R=10, W=2

6 TXs long readers
18 TXs short updaters

All active TXs
short updaters

All active TXs
long readers

Effect of long readers (10M row table)



If all TXs do updates, 1V 1.9× faster

Even if 1 long reader, MV/O 2.3× faster

2.3×

All active TXs short updaters

All active TXs long readers

Conclusions

- Single-version 2PL is fragile
  - – Great for update-heavy workloads, little contention
  - – But: problematic for hotspots, long read TXs
- Multi-version optimistic scheme is robust
  - – Readers don’t block writers, no waiting on locks
- Locking semantics can be offered efficiently
- High performance and full serializability without workload-specific knowledge