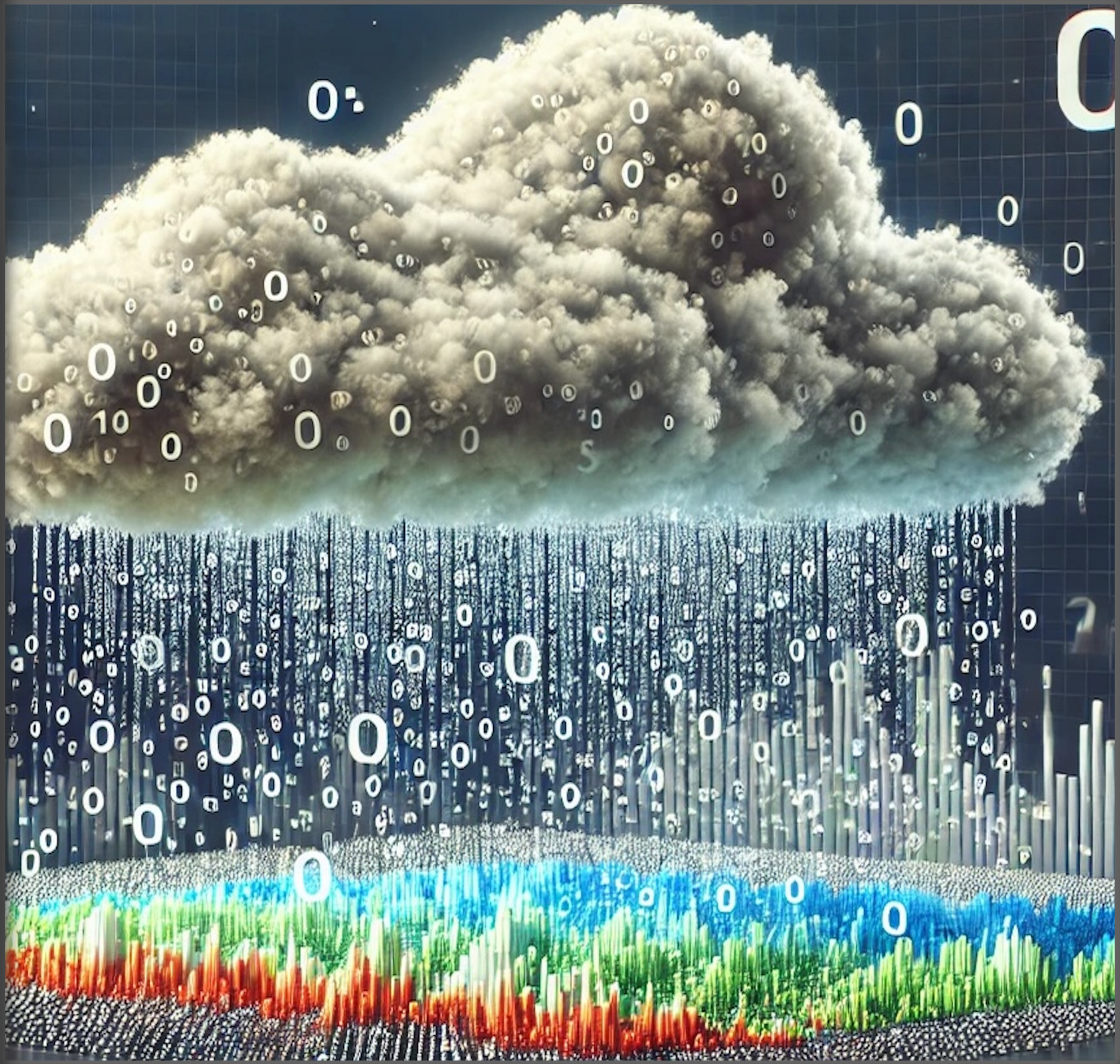
Carnegie Mellon University Advanced Database Systems (15-721)

Lecture #10

Memory-Optimized OLTP

Fall 2024 ❯❯ Prof. Jignesh Patel



ANNOUNCEMENTS

- Building blocks seminar (today) on Monday, September 30 @ 4:30pm
  - Accelerating Apache Spark workloads with Apache DataFusion Comet
  - https://db.cs.cmu.edu/events/building-blocks-apache-datafusion-comet-andy-grove/
- Talk from Oracle on (tomorrow) Tuesday, October 1, @ noon in 6501 GHC.
  - Unifying relational and document/JSON management.
  - https://cmu.zoom.us/my/jignesh
- Initial project meeting. You should have scheduled a 15-minute meeting slot. If not do that ASAP @ https://calendly.com/pateljm/initial-discussion-for-class-project
- Exam: Oct 9th in GHC 8102 between 1-4 pm. Open book.
  - Start anytime. Stop 90 minutes later.

BACKGROUND: SQL SERVER (BACK THEN) AND OLTP

- Many OLTP databases fit in memory. Now memory accesses can become the new bottleneck. Needs to rethink design choices.
- Analysis of transactional workloads: Where does time go in SQL Server?
  - CPI: Cycles per instructions
  - IR: Instructions Required:
  - SF: Scalability factor
- CPI: Influenced by code (e.g., fewer branches is better), and hardware.
  - Was 1.6 already in SQL Server – not much room to improve.
- SF: Property of the CC method and implementation
  - 1.89 (Ideal is 2). So not too far.
- Big gains will come from IR reduction: reduce instructions/txn.

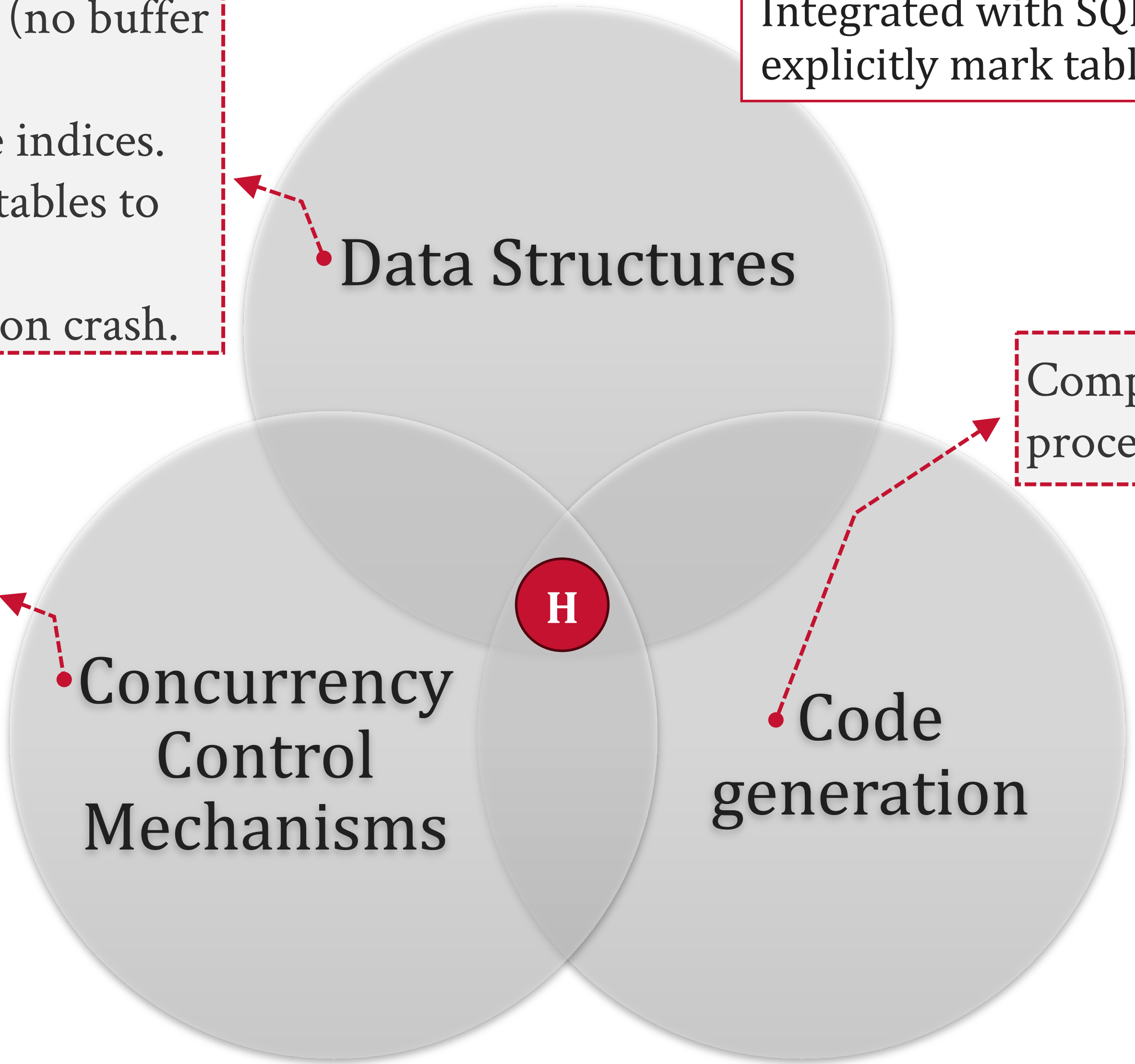
IN-MEMORY OLTP: RETHINK

Reduce instruction count / txn.

No need to partition the database across cores.

Integrated with SQL server, but need to explicitly mark tables as in-memory.

- In-memory data structures (no buffer pool overhead).
- In-memory hash and range indices.
- Checkpoint and Log main tables to disk (needed for recovery).
- Don't log indices – rebuild on crash.

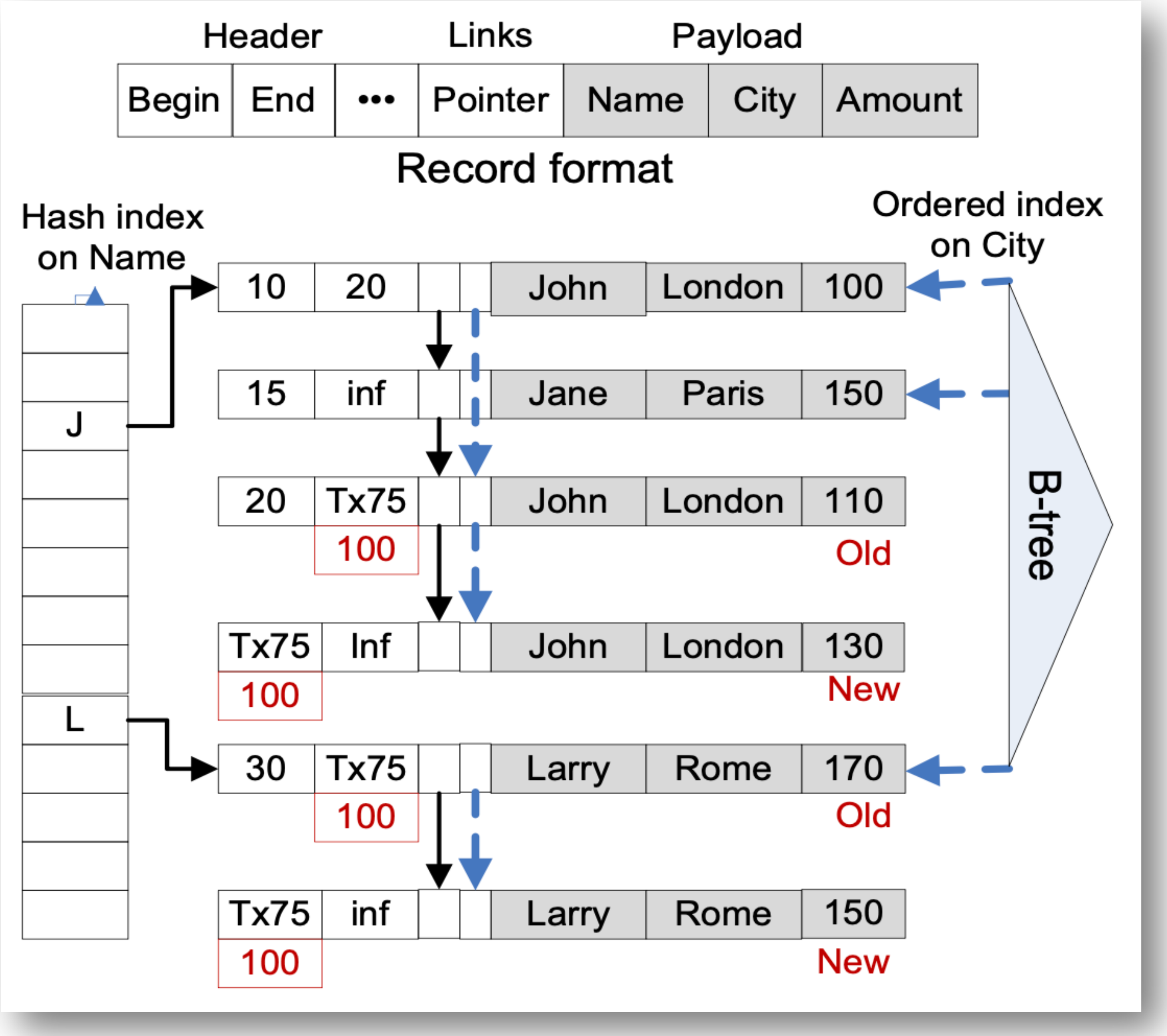


Compile statement and stored procedures to native code.

- Latch-free data structures (no locks).
- Optimistic MVCC Protocol.

Indices

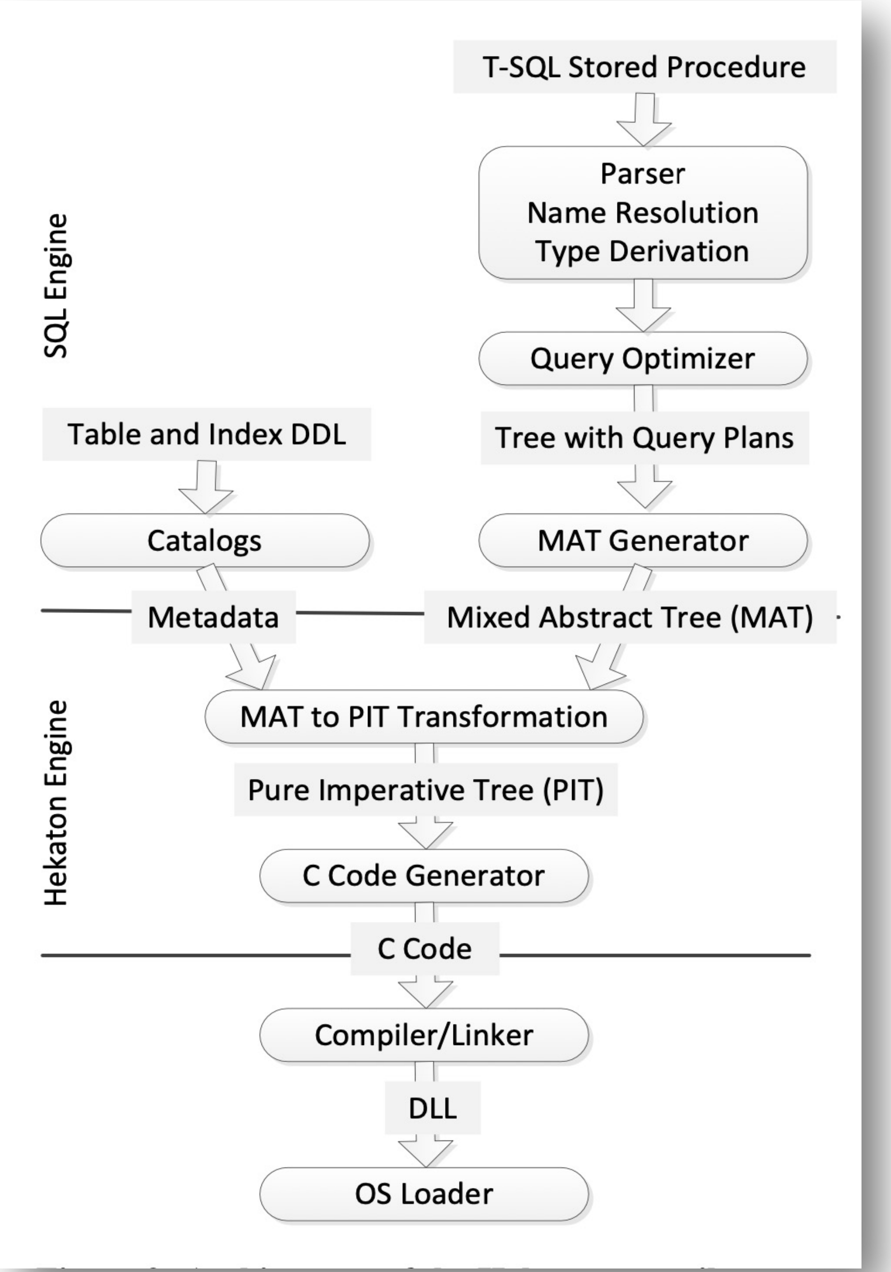
- In-memory, latch-free, hash and B-tree indices.
- The B-tree version is called Bw-tree.
  - Node changes stored as delta records.
  - Compare-and-swap (CAS) for atomic updates.
  - Log-structured page storage.
- Key points: need a fast concurrent index structure, so that each transaction can read/update indices quickly.



Levandoski et al. : The Bw-Tree: A B-tree for new hardware platforms. ICDE 2013
Wang et al.: Building a Bw-Tree Takes More Than Just Buzz Words. SIGMOD 2018

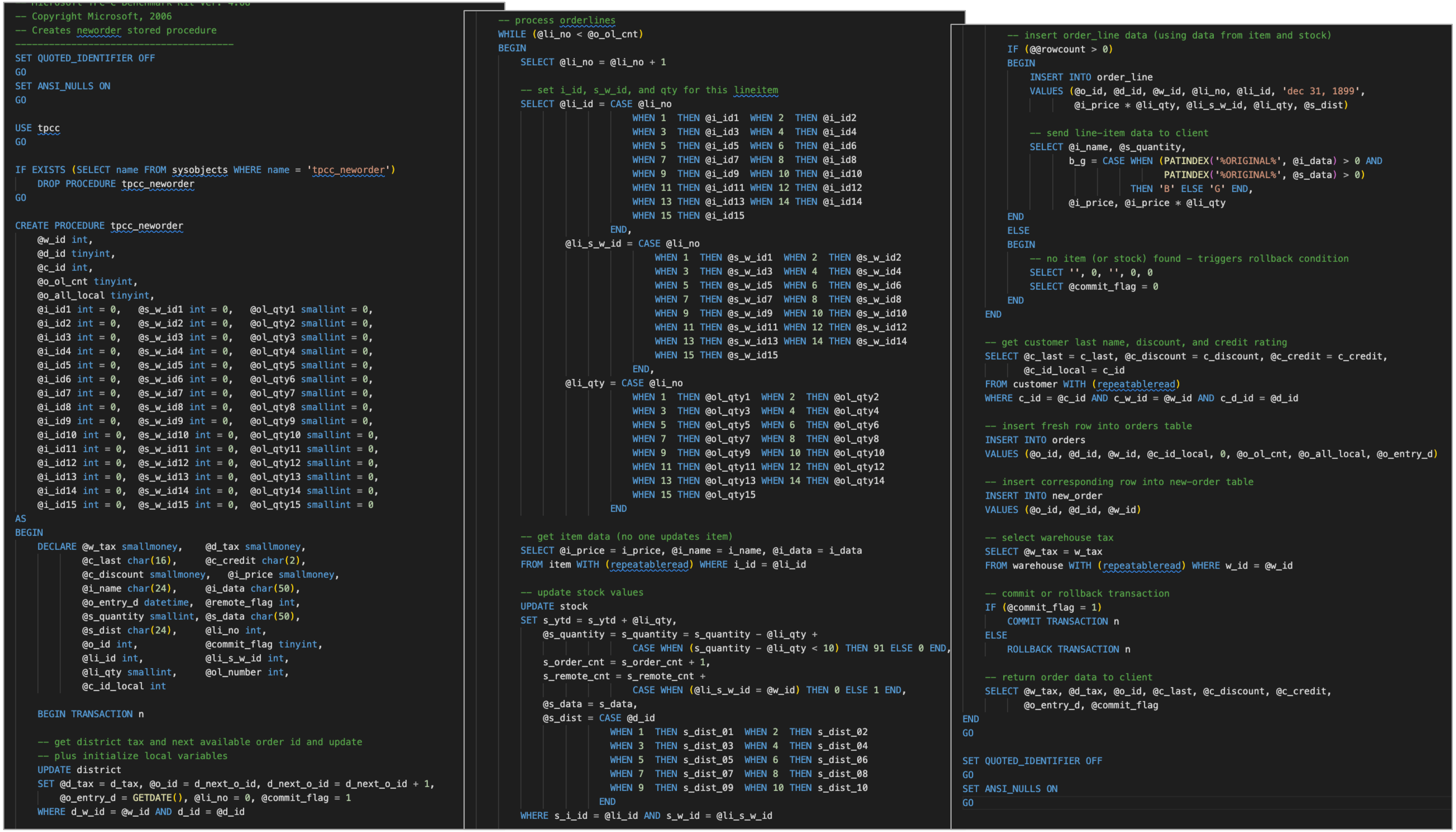
HEKATON: CODE COMPILER

- Codegen: T-SQL to C to machine code.
- Table creation also requires codegen.
  - To the compiler, records are opaque.
  - Functions like compareRecords() needs to be generated as schema changes.
- Type mismatch between T-SQL and C-types.
  - Stored Procedure -> MAT -> PIT -> Code.
- Other differences between C and SQL
  - NULLs: Special handling for operations like outerjoins.
  - Semantics of exceptions (e.g. divide by zero) differs in T-SQL and C.
- Note: Can't access regular (non in-memory) tables from a compiled stored procedure.



T-SQL is SQL Server's procedural programming language, similar to Oracle's PL/SQL and PostgreSQL's PL/pgSQL.

TPC-C New-Order Transaction as a T-SQL Stored Procedure



```sql
-- Microsoft TPC-C Benchmark Kit Ver. 4.68
-- Copyright Microsoft, 2006
-- Creates neworder stored procedure
----------------------------------------
SET QUOTED_IDENTIFIER OFF
GO
SET ANSI_NULLS ON
GO

USE tpcc
GO

IF EXISTS (SELECT name FROM sysobjects WHERE name = 'tpcc_neworder')
    DROP PROCEDURE tpcc_neworder
GO

CREATE PROCEDURE tpcc_neworder
    @w_id int,
    @d_id tinyint,
    @c_id int,
    @o_ol_cnt tinyint,
    @o_all_local tinyint,
    @i_id1 int = 0,   @s_w_id1 int = 0,   @ol_qty1 smallint = 0,
    @i_id2 int = 0,   @s_w_id2 int = 0,   @ol_qty2 smallint = 0,
    @i_id3 int = 0,   @s_w_id3 int = 0,   @ol_qty3 smallint = 0,
    @i_id4 int = 0,   @s_w_id4 int = 0,   @ol_qty4 smallint = 0,
    @i_id5 int = 0,   @s_w_id5 int = 0,   @ol_qty5 smallint = 0,
    @i_id6 int = 0,   @s_w_id6 int = 0,   @ol_qty6 smallint = 0,
    @i_id7 int = 0,   @s_w_id7 int = 0,   @ol_qty7 smallint = 0,
    @i_id8 int = 0,   @s_w_id8 int = 0,   @ol_qty8 smallint = 0,
    @i_id9 int = 0,   @s_w_id9 int = 0,   @ol_qty9 smallint = 0,
    @i_id10 int = 0,  @s_w_id10 int = 0,  @ol_qty10 smallint = 0,
    @i_id11 int = 0,  @s_w_id11 int = 0,  @ol_qty11 smallint = 0,
    @i_id12 int = 0,  @s_w_id12 int = 0,  @ol_qty12 smallint = 0,
    @i_id13 int = 0,  @s_w_id13 int = 0,  @ol_qty13 smallint = 0,
    @i_id14 int = 0,  @s_w_id14 int = 0,  @ol_qty14 smallint = 0,
    @i_id15 int = 0,  @s_w_id15 int = 0,  @ol_qty15 smallint = 0
AS
BEGIN
    DECLARE @w_tax smallmoney,    @d_tax smallmoney,
            @c_last char(16),     @c_credit char(2),
            @c_discount smallmoney,   @i_price smallmoney,
            @i_name char(24),     @i_data char(50),
            @o_entry_d datetime,  @remote_flag int,
            @s_quantity smallint, @s_data char(50),
            @s_dist char(24),     @li_no int,
            @o_id int,            @commit_flag tinyint,
            @li_id int,           @li_s_w_id int,
            @li_qty smallint,     @ol_number int,
            @c_id_local int

    BEGIN TRANSACTION n

    -- get district tax and next available order id and update
    -- plus initialize local variables
    UPDATE district
    SET @d_tax = d_tax, @o_id = d_next_o_id, d_next_o_id = d_next_o_id + 1,
        @o_entry_d = GETDATE(), @li_no = 0, @commit_flag = 1
    WHERE d_w_id = @w_id AND d_id = @d_id

    -- process orderlines
    WHILE (@li_no < @o_ol_cnt)
    BEGIN
        SELECT @li_no = @li_no + 1

        -- set i_id, s_w_id, and qty for this lineitem
        SELECT @li_id = CASE @li_no
                            WHEN 1  THEN @i_id1  WHEN 2  THEN @i_id2
                            WHEN 3  THEN @i_id3  WHEN 4  THEN @i_id4
                            WHEN 5  THEN @i_id5  WHEN 6  THEN @i_id6
                            WHEN 7  THEN @i_id7  WHEN 8  THEN @i_id8
                            WHEN 9  THEN @i_id9  WHEN 10 THEN @i_id10
                            WHEN 11 THEN @i_id11 WHEN 12 THEN @i_id12
                            WHEN 13 THEN @i_id13 WHEN 14 THEN @i_id14
                            WHEN 15 THEN @i_id15
                        END,
                @li_s_w_id = CASE @li_no
                                WHEN 1  THEN @s_w_id1  WHEN 2  THEN @s_w_id2
                                WHEN 3  THEN @s_w_id3  WHEN 4  THEN @s_w_id4
                                WHEN 5  THEN @s_w_id5  WHEN 6  THEN @s_w_id6
                                WHEN 7  THEN @s_w_id7  WHEN 8  THEN @s_w_id8
                                WHEN 9  THEN @s_w_id9  WHEN 10 THEN @s_w_id10
                                WHEN 11 THEN @s_w_id11 WHEN 12 THEN @s_w_id12
                                WHEN 13 THEN @s_w_id13 WHEN 14 THEN @s_w_id14
                                WHEN 15 THEN @s_w_id15
                            END,
                @li_qty = CASE @li_no
                            WHEN 1  THEN @ol_qty1  WHEN 2  THEN @ol_qty2
                            WHEN 3  THEN @ol_qty3  WHEN 4  THEN @ol_qty4
                            WHEN 5  THEN @ol_qty5  WHEN 6  THEN @ol_qty6
                            WHEN 7  THEN @ol_qty7  WHEN 8  THEN @ol_qty8
                            WHEN 9  THEN @ol_qty9  WHEN 10 THEN @ol_qty10
                            WHEN 11 THEN @ol_qty11 WHEN 12 THEN @ol_qty12
                            WHEN 13 THEN @ol_qty13 WHEN 14 THEN @ol_qty14
                            WHEN 15 THEN @ol_qty15
                        END

        -- get item data (no one updates item)
        SELECT @i_price = i_price, @i_name = i_name, @i_data = i_data
        FROM item WITH (repeatableread) WHERE i_id = @li_id

        -- update stock values
        UPDATE stock
        SET s_ytd = s_ytd + @li_qty,
            @s_quantity = s_quantity = s_quantity - @li_qty +
                            CASE WHEN (s_quantity - @li_qty < 10) THEN 91 ELSE 0 END,
            s_order_cnt = s_order_cnt + 1,
            s_remote_cnt = s_remote_cnt +
                            CASE WHEN (@li_s_w_id = @w_id) THEN 0 ELSE 1 END,
            @s_data = s_data,
            @s_dist = CASE @d_id
                        WHEN 1  THEN s_dist_01  WHEN 2  THEN s_dist_02
                        WHEN 3  THEN s_dist_03  WHEN 4  THEN s_dist_04
                        WHEN 5  THEN s_dist_05  WHEN 6  THEN s_dist_06
                        WHEN 7  THEN s_dist_07  WHEN 8  THEN s_dist_08
                        WHEN 9  THEN s_dist_09  WHEN 10 THEN s_dist_10
                      END
        WHERE s_i_id = @li_id AND s_w_id = @li_s_w_id

        -- insert order_line data (using data from item and stock)
        IF (@@rowcount > 0)
        BEGIN
            INSERT INTO order_line
            VALUES (@o_id, @d_id, @w_id, @li_no, @li_id, 'dec 31, 1899',
                    @i_price * @li_qty, @li_s_w_id, @li_qty, @s_dist)

            -- send line-item data to client
            SELECT @i_name, @s_quantity,
                   b_g = CASE WHEN (PATINDEX('%ORIGINAL%', @i_data) > 0 AND
                                    PATINDEX('%ORIGINAL%', @s_data) > 0)
                              THEN 'B' ELSE 'G' END,
                   @i_price, @i_price * @li_qty
        END
        ELSE
        BEGIN
            -- no item (or stock) found - triggers rollback condition
            SELECT '', 0, '', 0, 0
            SELECT @commit_flag = 0
        END
    END

    -- get customer last name, discount, and credit rating
    SELECT @c_last = c_last, @c_discount = c_discount, @c_credit = c_credit,
           @c_id_local = c_id
    FROM customer WITH (repeatableread)
    WHERE c_id = @c_id AND c_w_id = @w_id AND c_d_id = @d_id

    -- insert fresh row into orders table
    INSERT INTO orders
    VALUES (@o_id, @d_id, @w_id, @c_id_local, 0, @o_ol_cnt, @o_all_local, @o_entry_d)

    -- insert corresponding row into new-order table
    INSERT INTO new_order
    VALUES (@o_id, @d_id, @w_id)

    -- select warehouse tax
    SELECT @w_tax = w_tax
    FROM warehouse WITH (repeatableread) WHERE w_id = @w_id

    -- commit or rollback transaction
    IF (@commit_flag = 1)
        COMMIT TRANSACTION n
    ELSE
        ROLLBACK TRANSACTION n

    -- return order data to client
    SELECT @w_tax, @d_tax, @o_id, @c_last, @c_discount, @c_credit,
           @o_entry_d, @commit_flag
END
GO

SET QUOTED_IDENTIFIER OFF
GO
SET ANSI_NULLS ON
GO
```

HEKATON'S APPROACH

- MVCC + Optimistic
- Supports multiple isolation levels without locking, including snapshot isolation.
  - Recall Snapshot Isolation is a weaker form of isolation than Weaker than Serializable.
  - Reads are as of the start of the txn.
  - Writes as of the end of the txn.

TRANSACTION = UNIT OF WORK

Example: A bank rewards old customers with a high balance

TRANSACTION

1. Look up George's account balance
2. Look up Alice's account balance
3. Look up Bob's account balance
4. Add $5 to account with highest balance

Atomicity

Isolation

Concurrency control ensures these properties

DESIGNING FOR IN-MEMORY OLTP

Traditional disk-oriented engine	In-memory engine
Disk-friendly data structures: Pages, B-tree index.	Latch-free hash table / B-trees stores individual records.

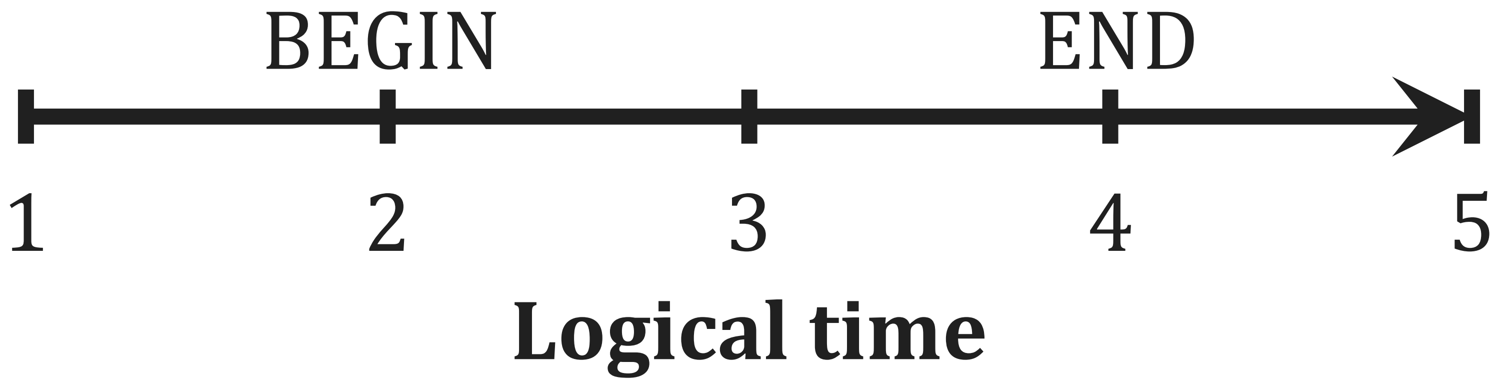
Compare Hekaton To other Approaches (H-Store)

H-Store	Hekaton
Scales **out**.	Scales **up**.
Communication across partitions is **expensive**.	Main memory is **shared** and **coherent**.
One CPU can access a given record.	**Any** CPU can access a given record.
TXs that span partitions participate in **2PC**.	TXs **validate** their reads to enforce isolation.
Perfect for **partitionable** workloads.	**Generic**, no need to specify partitions.

H-Store is an example of an approach that partitions the data and optimizes for txns that touch a single partition. The motivation for that approach is that many txns can be made to work in a single partition, and can we make those go as fast as we can.

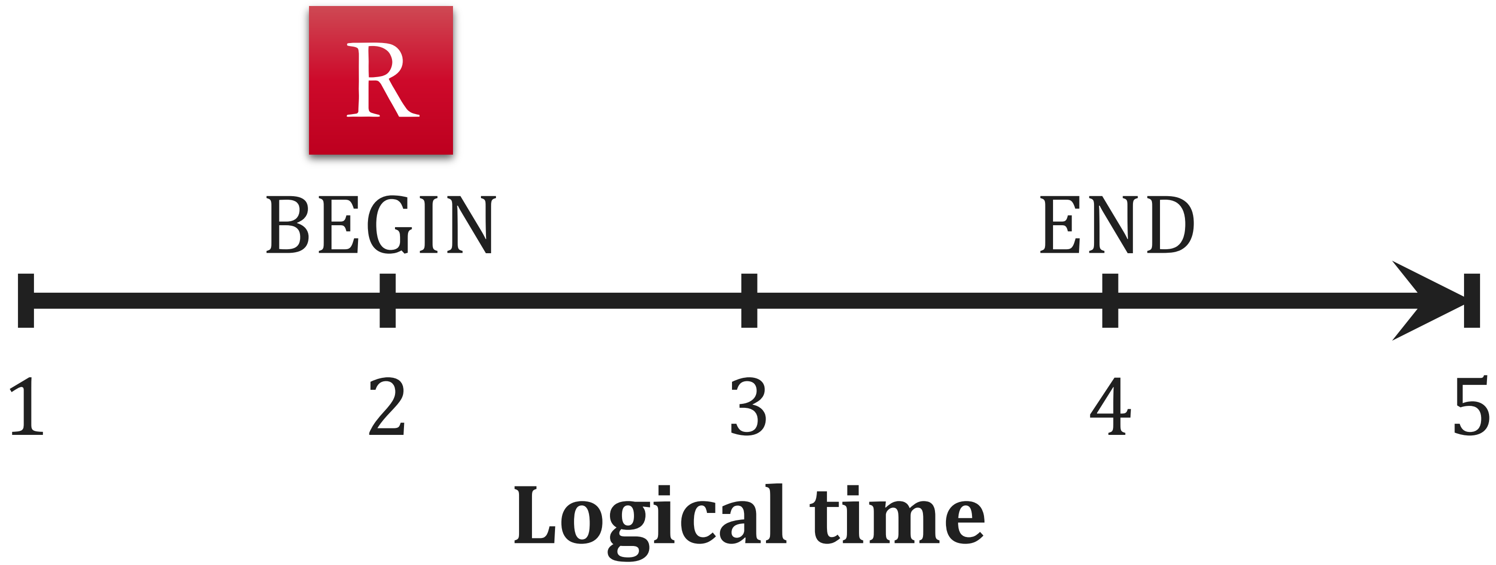
Multi-version optimistic scheme: Snapshot Isolation

- TXs have two unique timestamps: BEGIN, END.



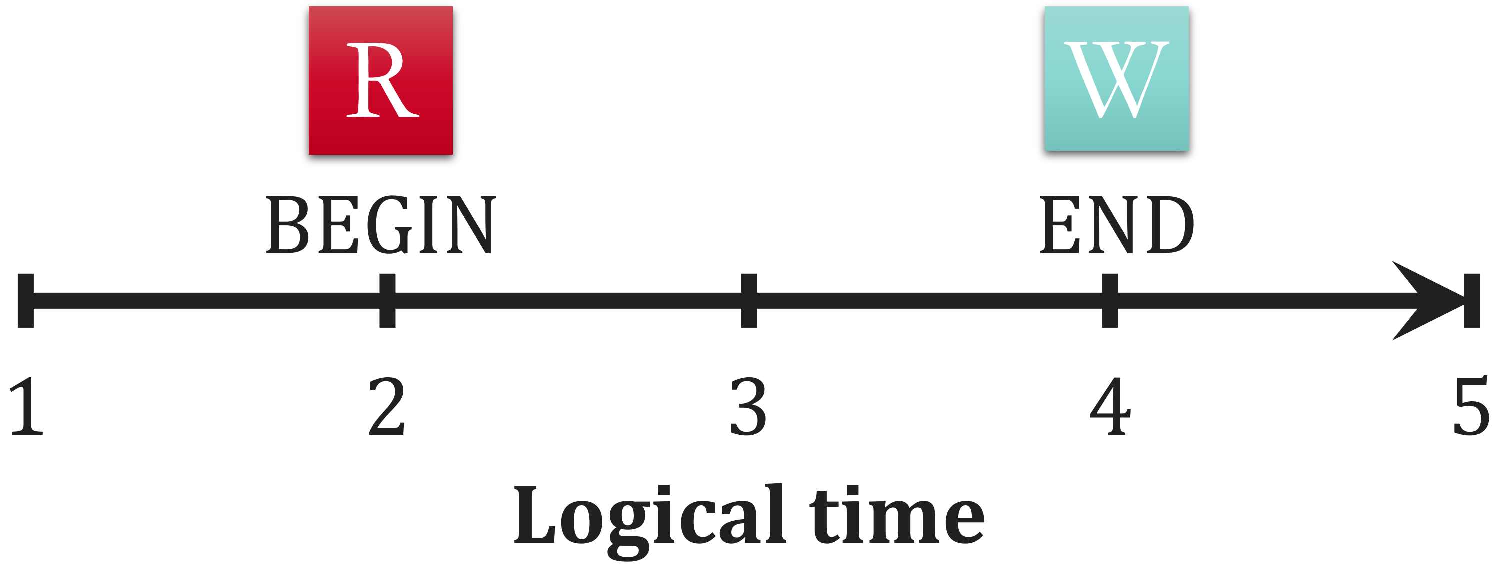
Multi-version optimistic scheme: Snapshot Isolation

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



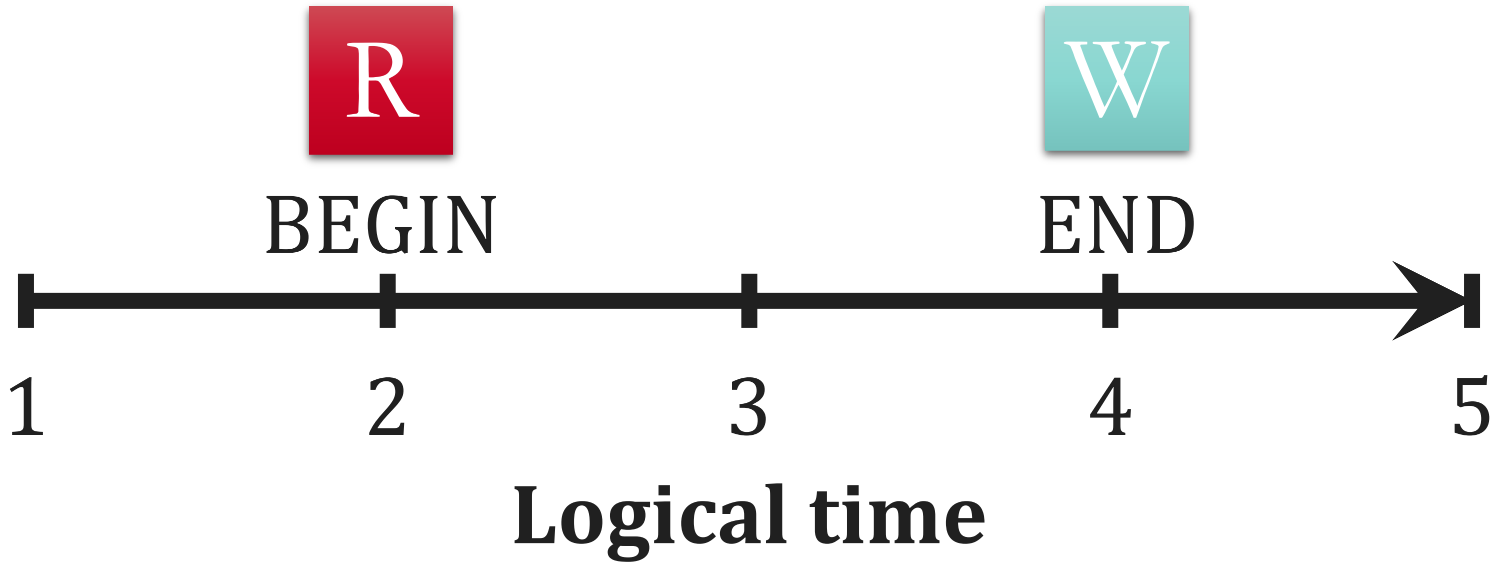
Multi-version optimistic scheme: Snapshot Isolation

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



Multi-version optimistic scheme: Snapshot Isolation

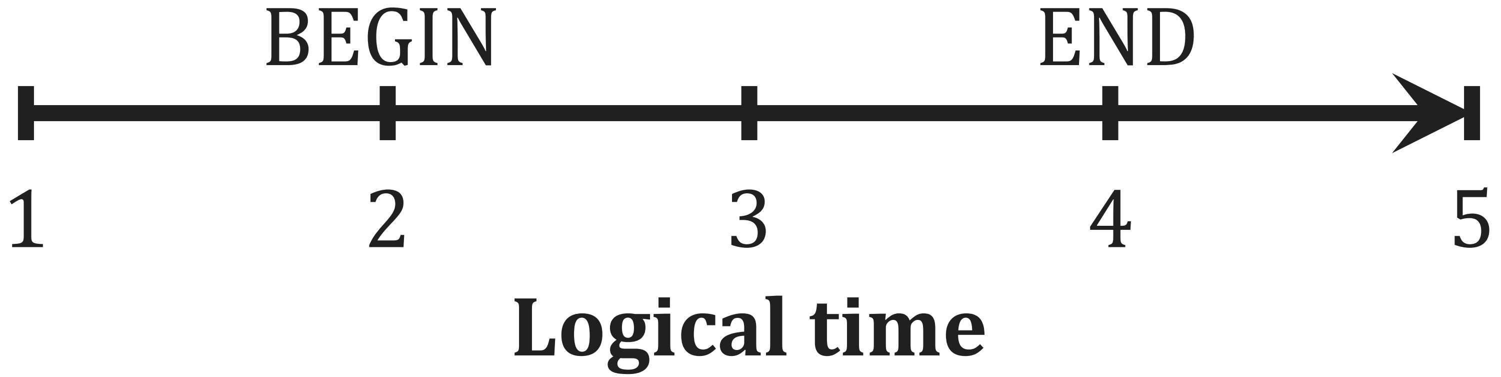
- TXs have two unique timestamps: BEGIN, END.
- Read as of BEGIN timestamp.
- Write as of END timestamp.



Sufficient for Read Committed.

But not for Serializable.

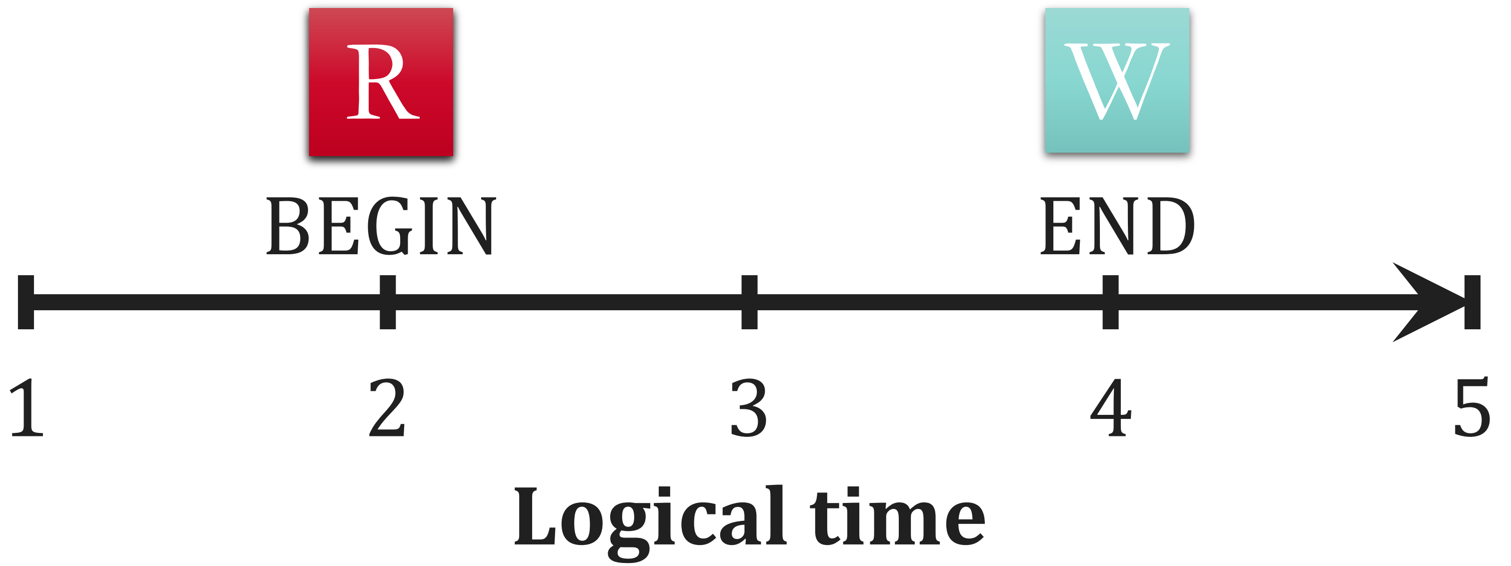
Making Snapshot Isolation (SI) Serializable



Mihaela A. Bornea, Orion Hodson, Sameh Elnikety, Alan D. Fekete: One-copy serializability with snapshot isolation under the hood. ICDE 2011

Making Snapshot Isolation (SI) Serializable

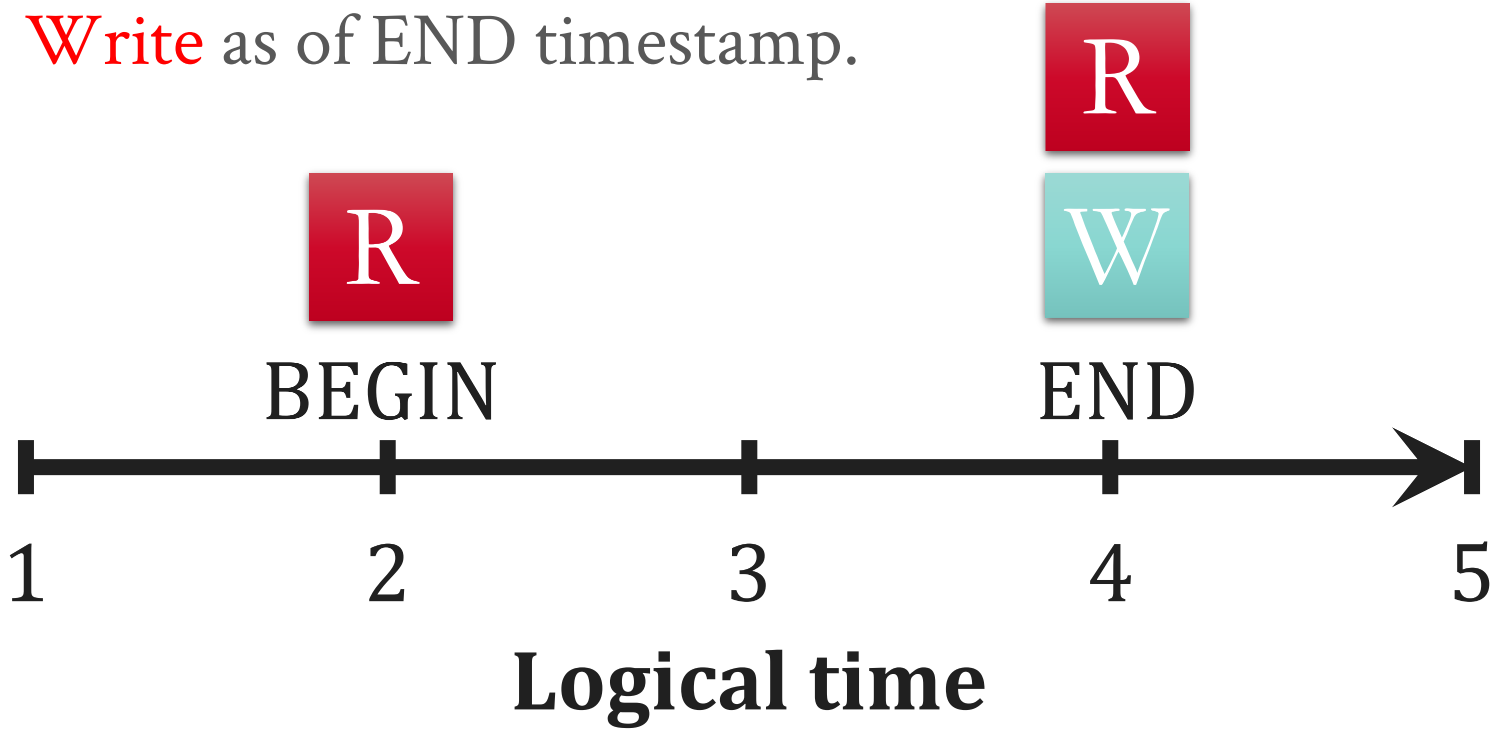
- Read as of BEGIN timestamp.



Mihaela A. Bornea, Orion Hodson, Sameh Elnikety, Alan D. Fekete: One-copy serializability with snapshot isolation under the hood. ICDE 2011

Making Snapshot Isolation (SI) Serializable

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



Mihaela A. Bornea, Orion Hodson, Sameh Elnikety, Alan D. Fekete: One-copy serializability with snapshot isolation under the hood. ICDE 2011

Support Multiple Isolation Levels

- SQL has multiple isolation levels, and we want to support that.
  - These trade isolation for performance.
  - Want to allow concurrent transaction with different isolation levels.
- Can a multi-version optimistic CC protocol support these isolation levels?

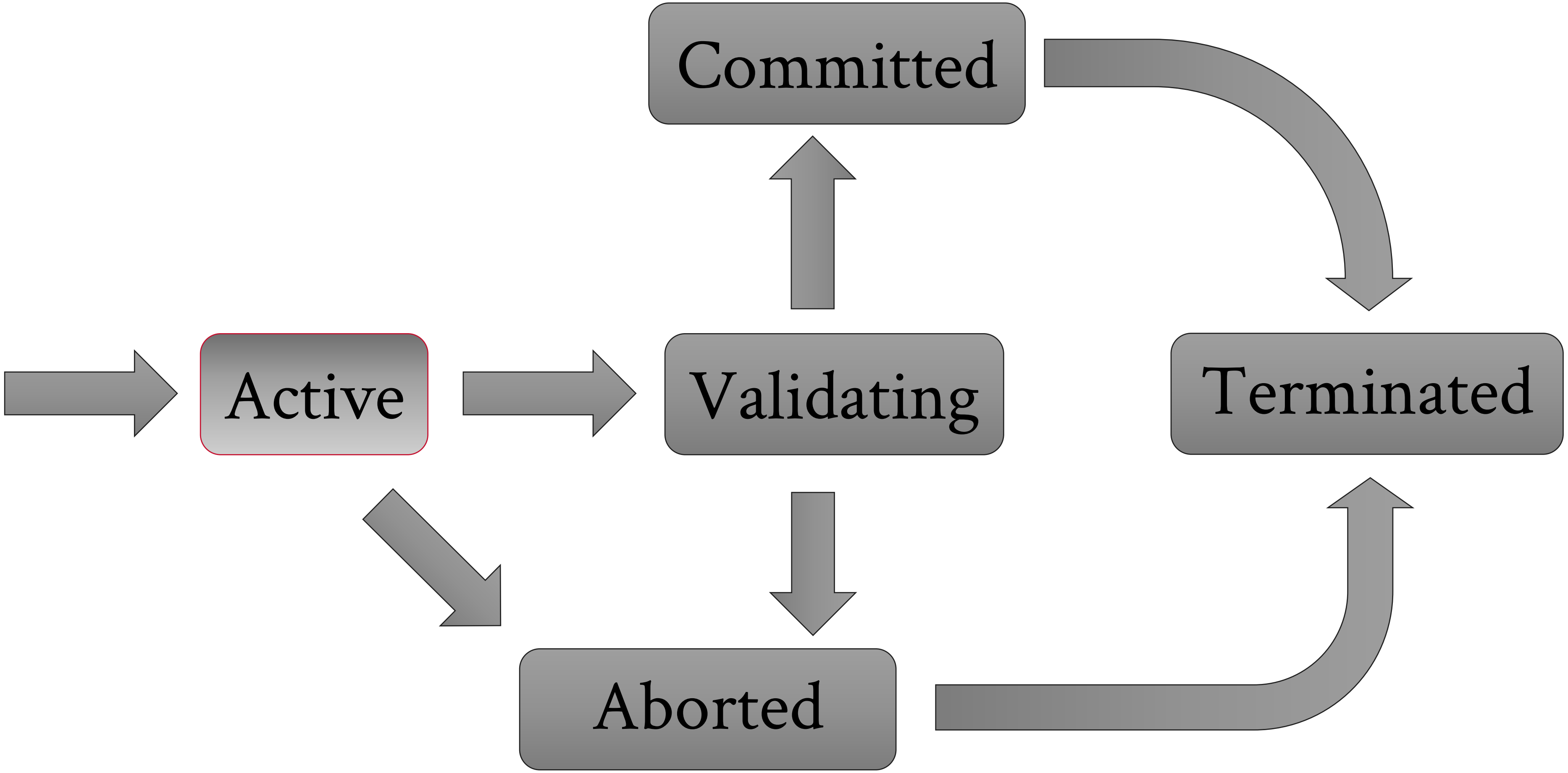
SQL level
Serializable
Repeatable Read
Read Committed
Read Uncommitted

MV/O can offers this choice too!

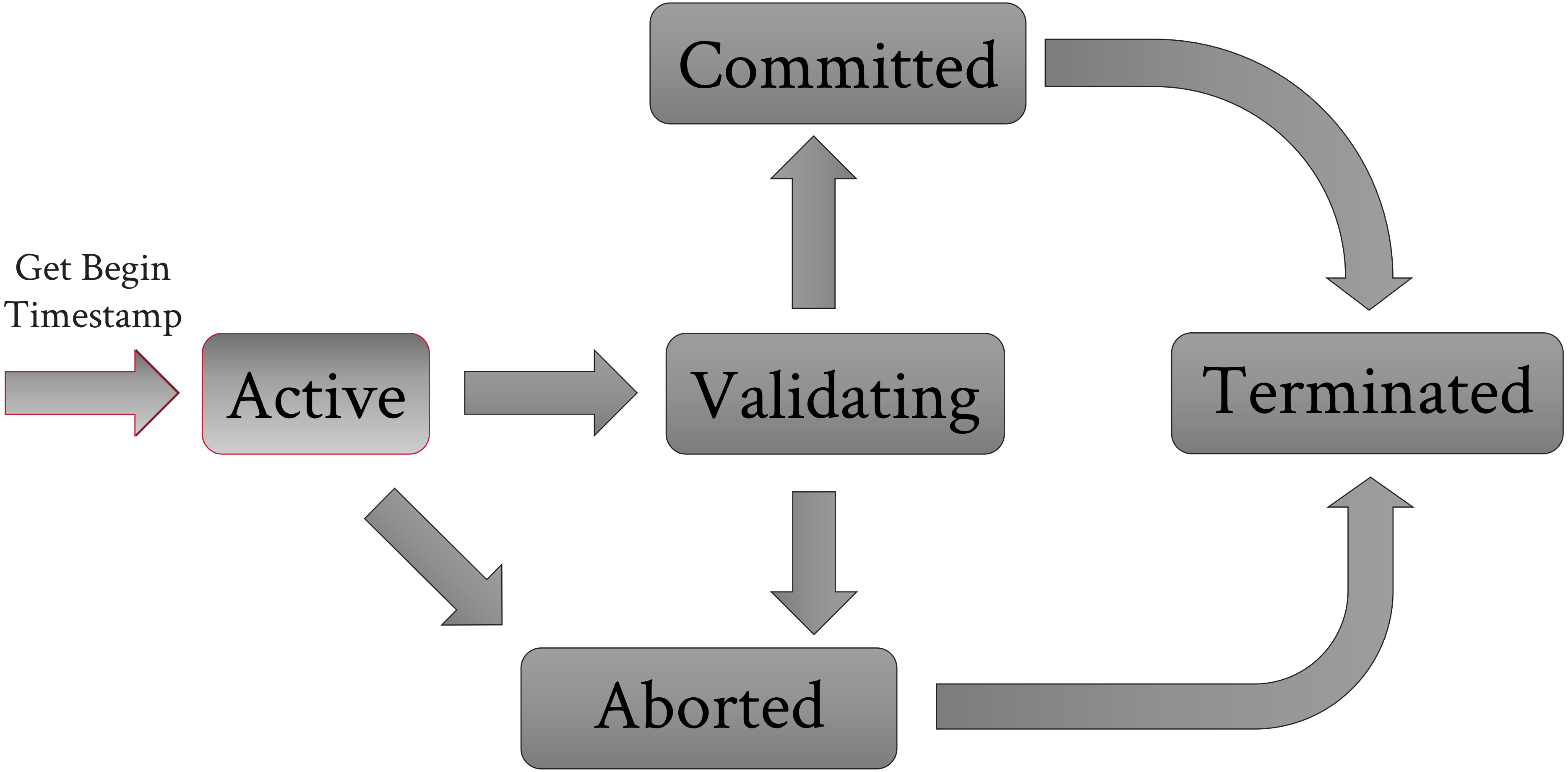
MV/O: What needs to be validated?

- Depends on the isolation level.
- Read Committed: 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.

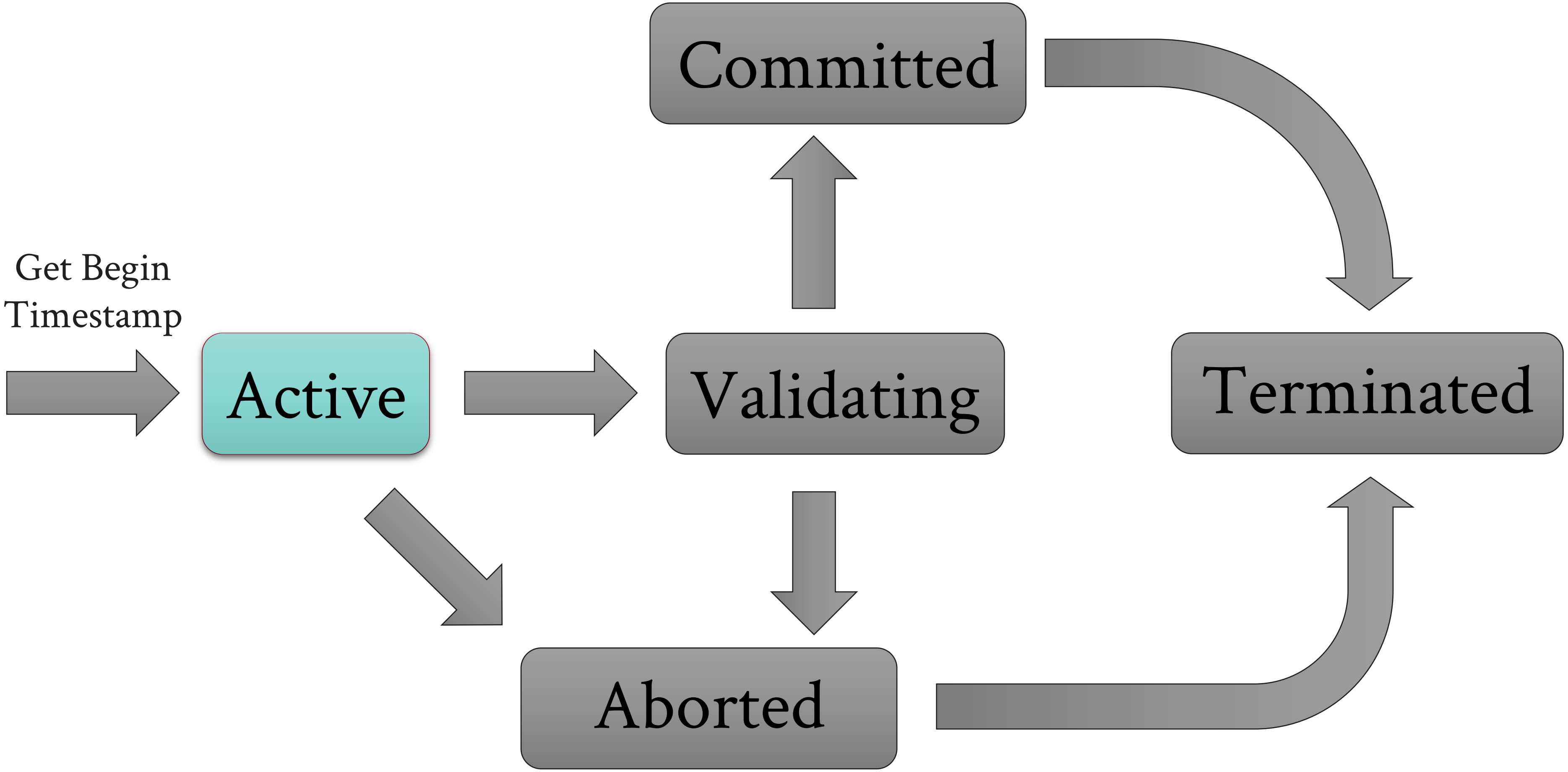
TRANSACTION STATES



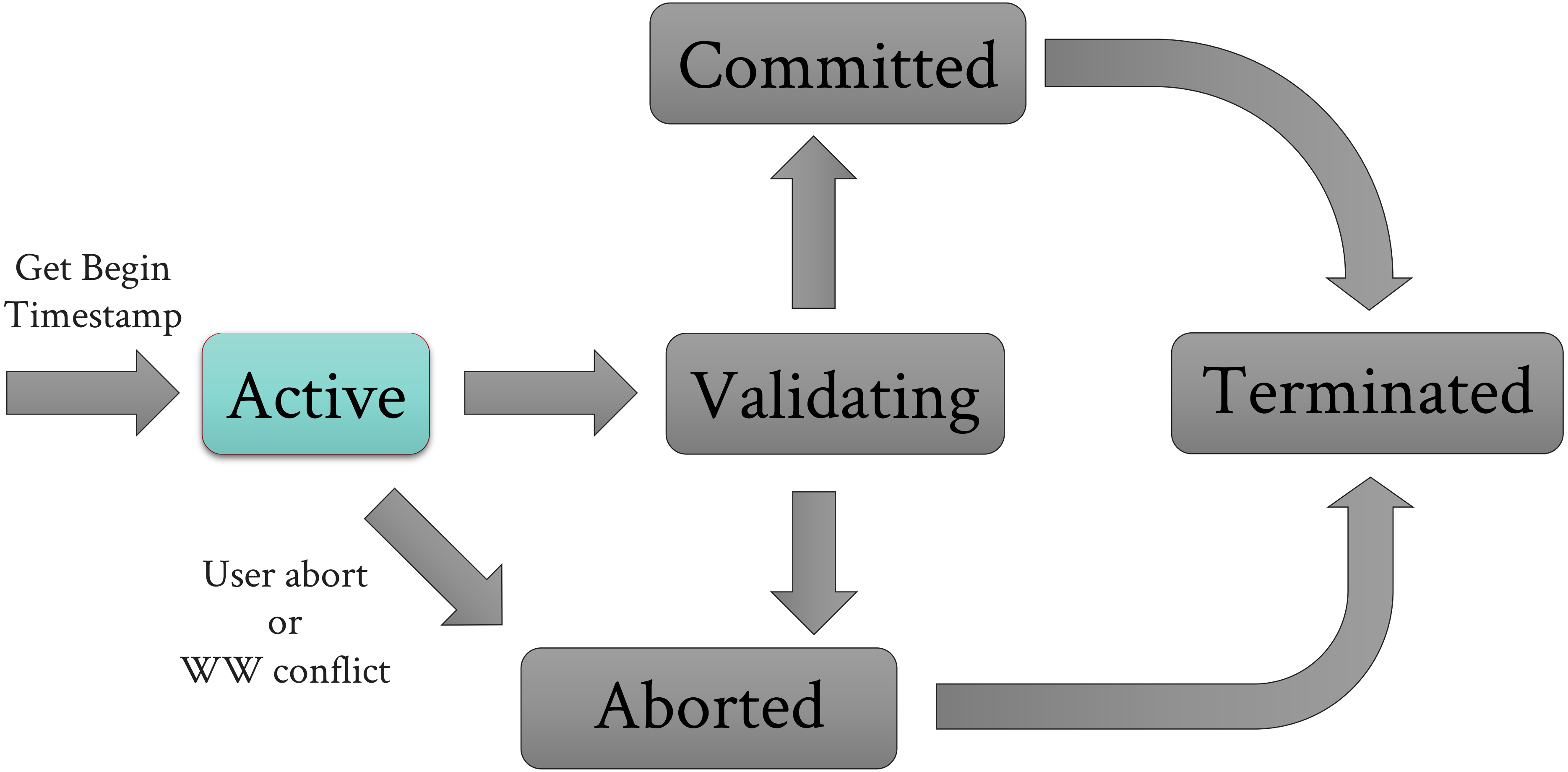
TRANSACTION STATES



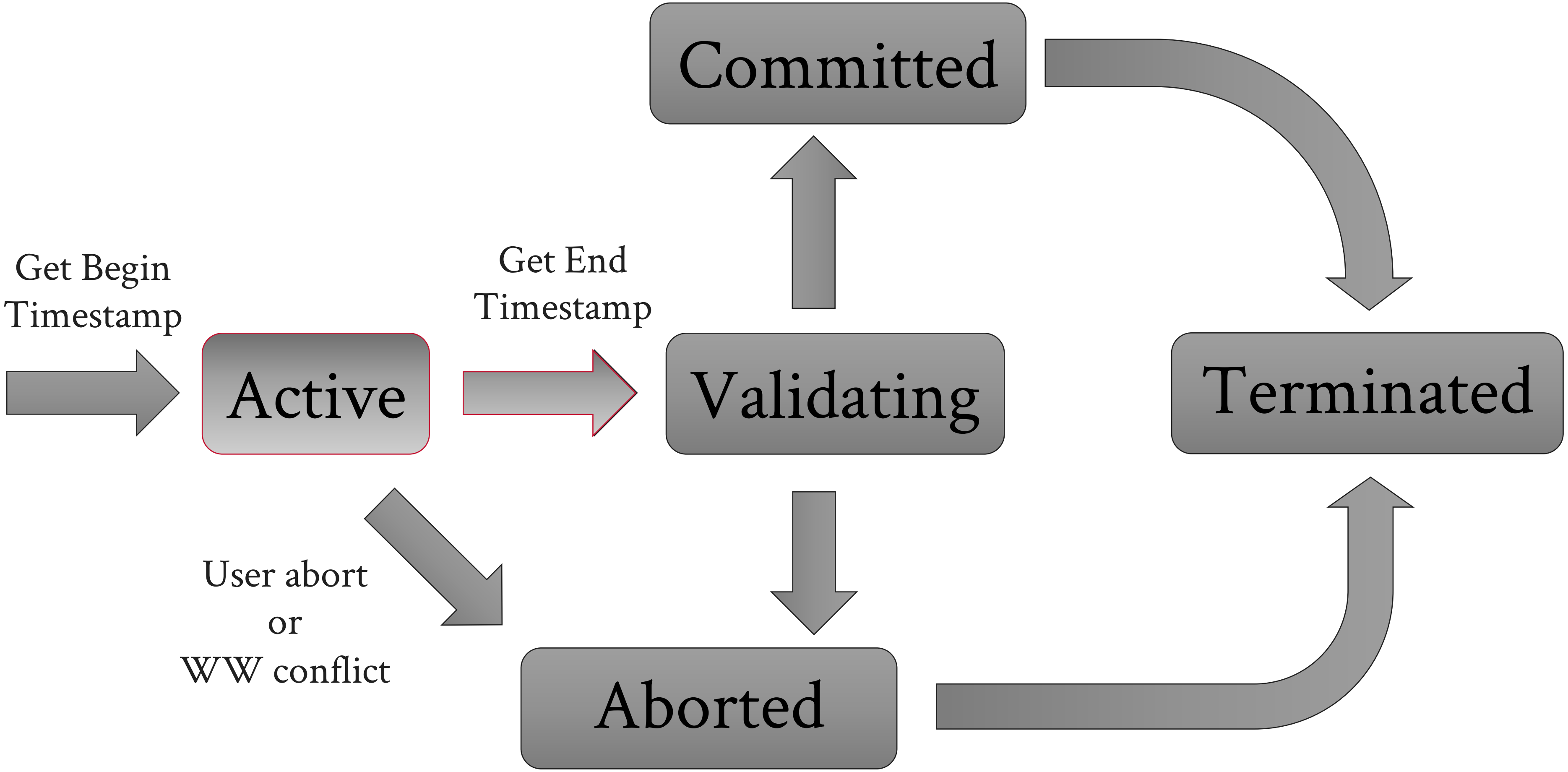
TRANSACTION STATES



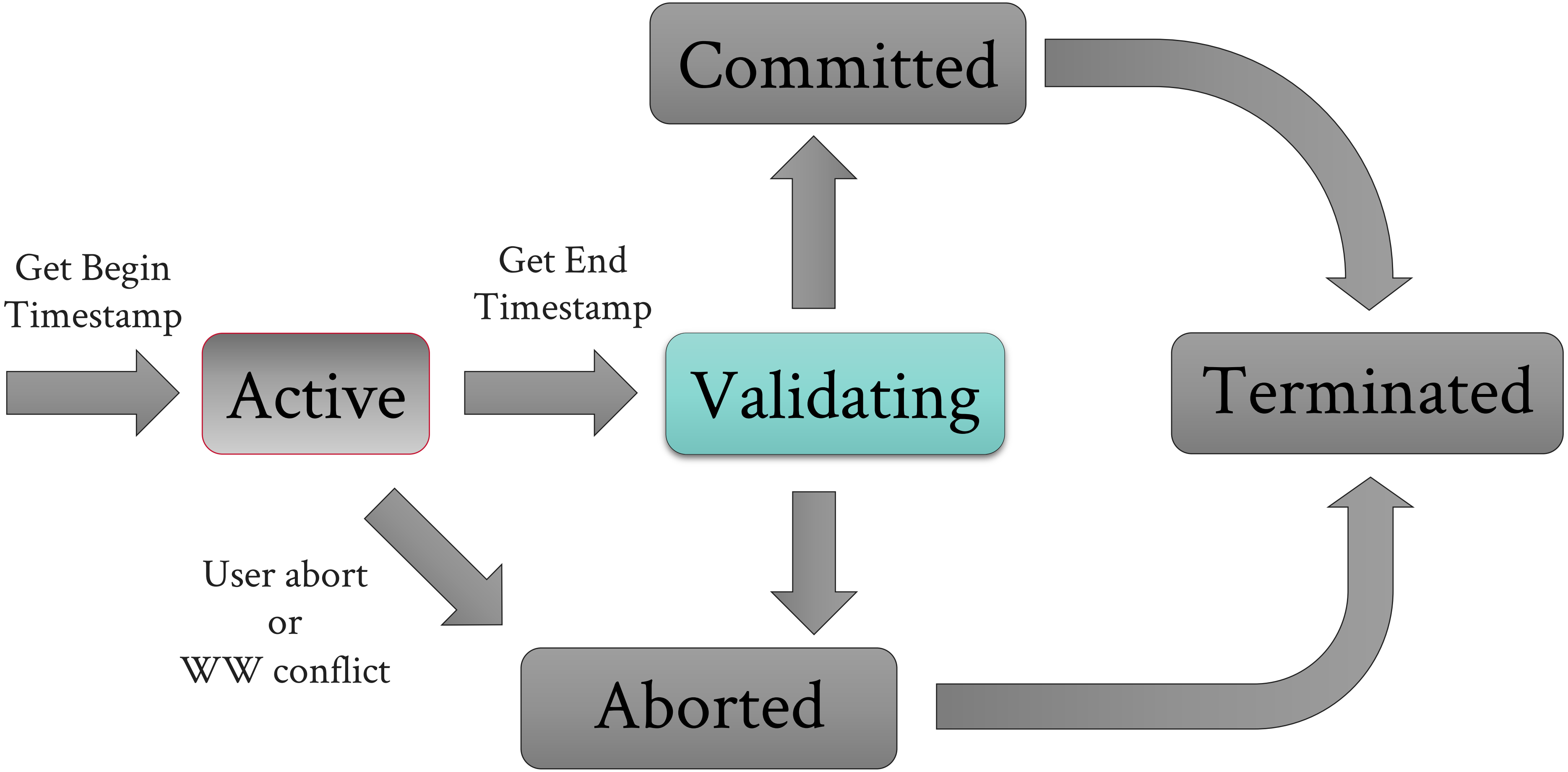
TRANSACTION STATES



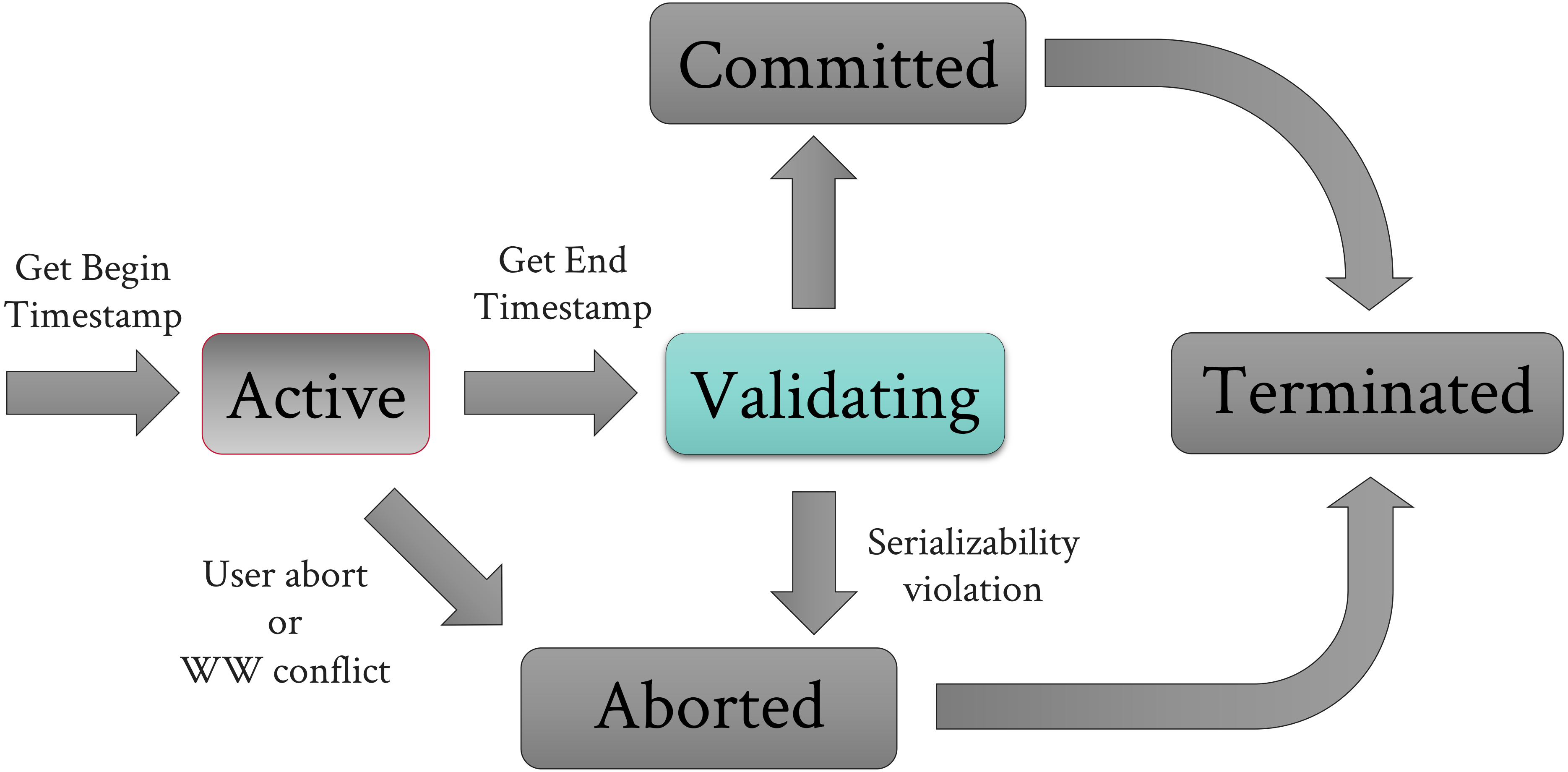
TRANSACTION STATES



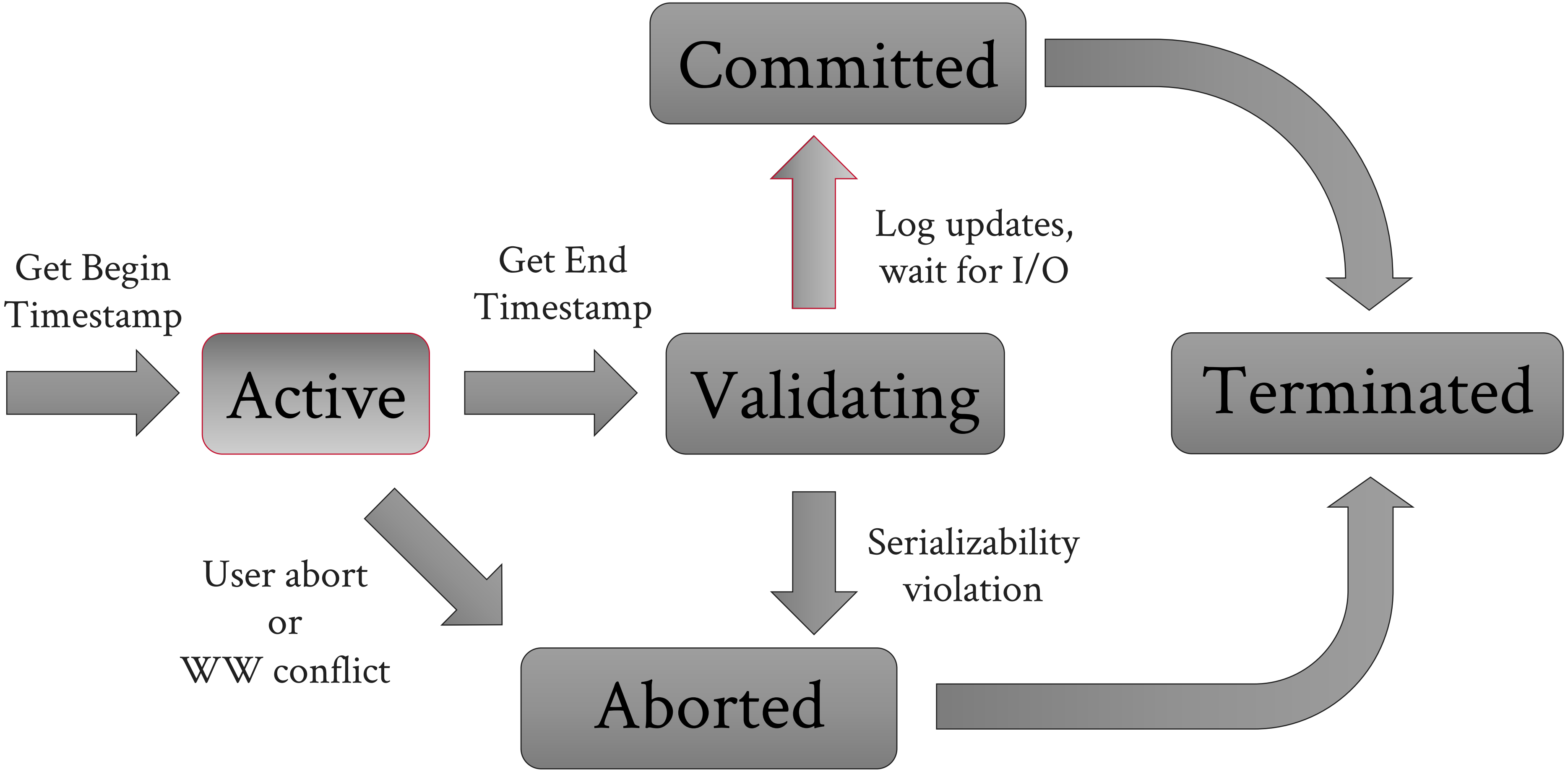
Transaction States



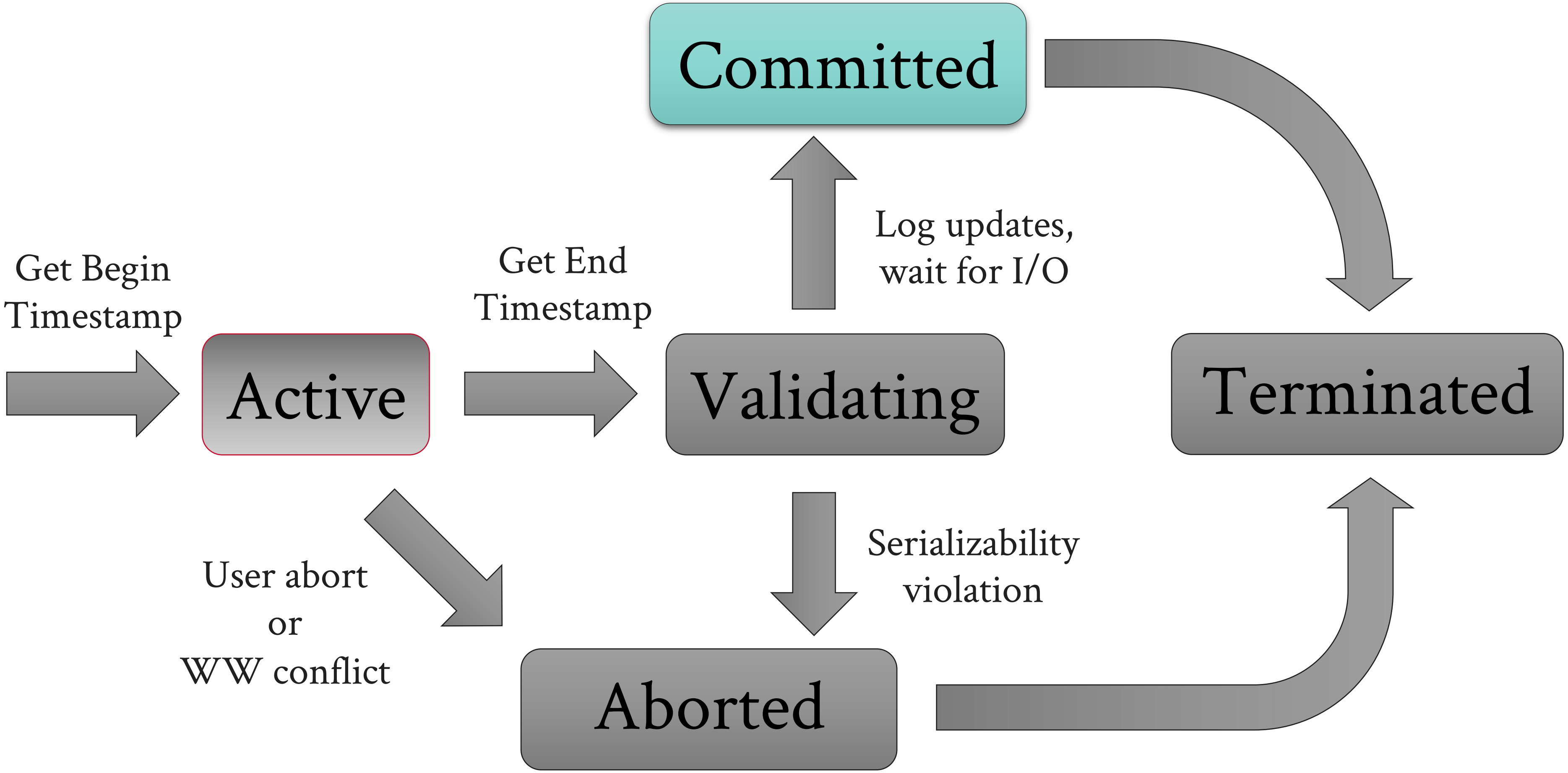
TRANSACTION STATES



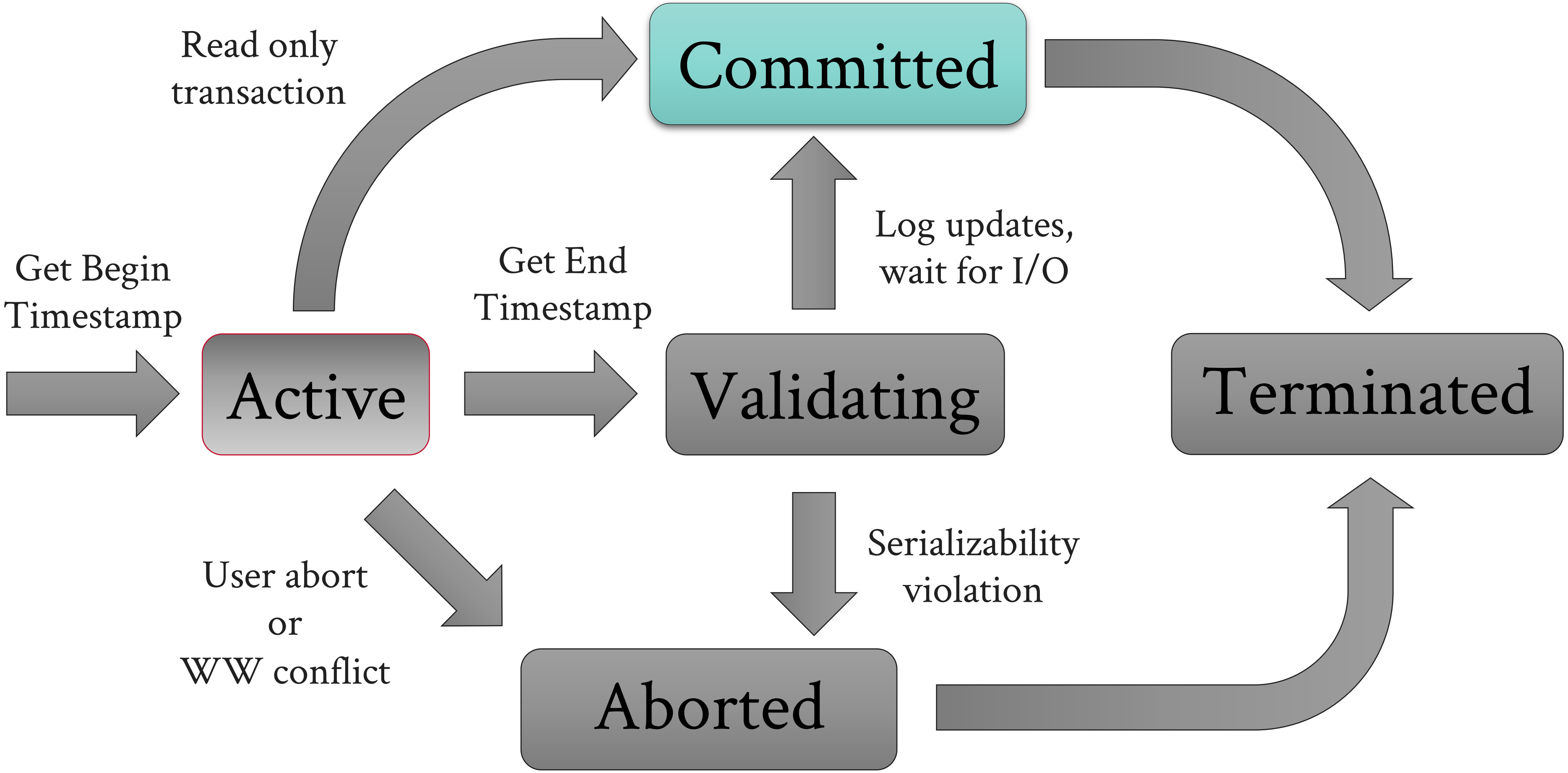
TRANSACTION STATES



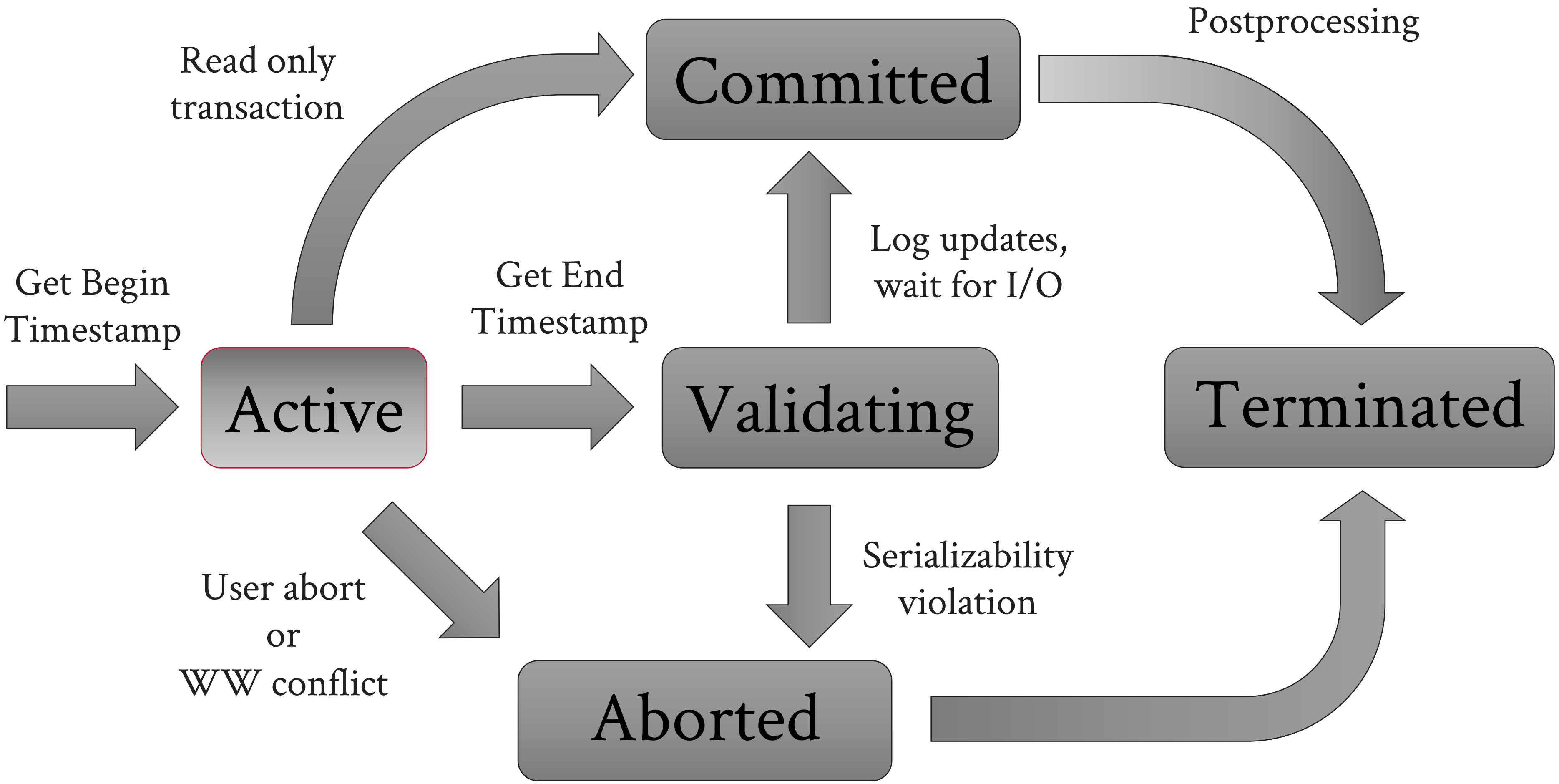
TRANSACTION STATES



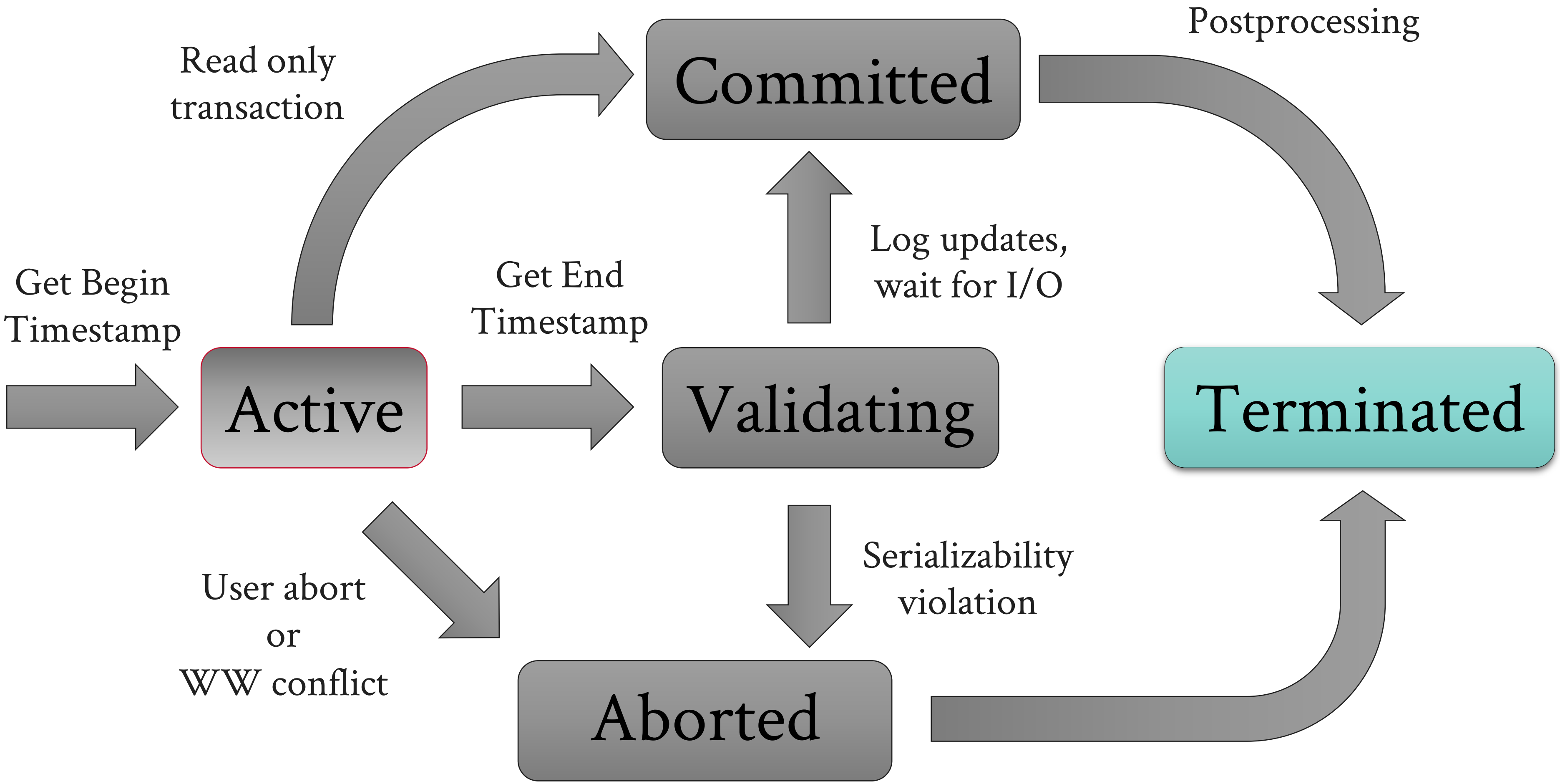
TRANSACTION STATES



Transaction States



Transaction States



Example

- Bank stores (customer, account balance).
- Bank wants to reward good customers.
- Transaction:
  1. Lookup balance for George, Alice, Bob.
  2. Add $5 to the account with the highest balance.

Alice	$75
Bob	$92
David	$106
Frank	$31
George	$98

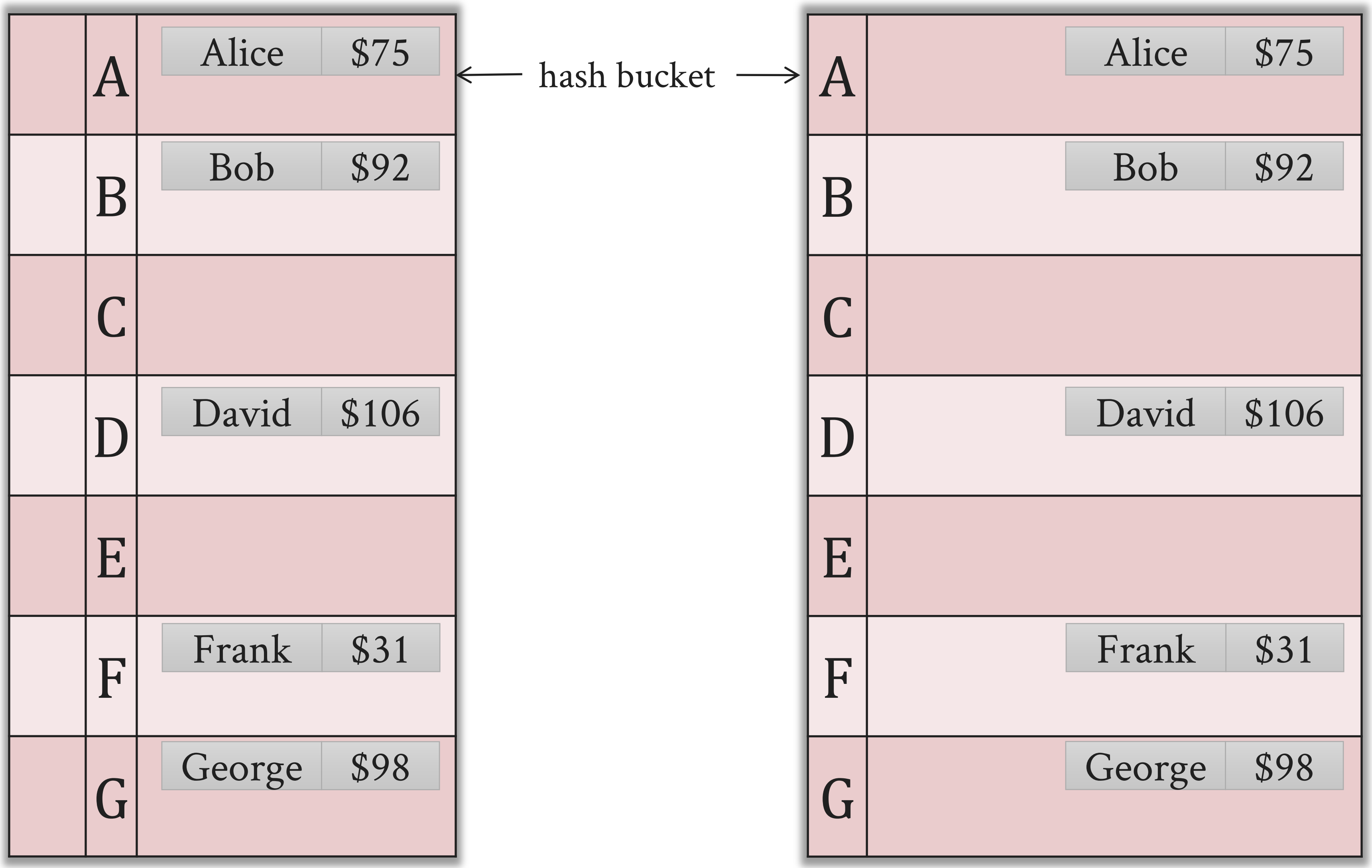
Compare MV with Single Version

1V

- Traditional algorithm, optimized for memory-resident data.
- Keeps a single version.
- Synchronization via locks:
  - Acquired on access.
  - Released after commit.

MV/O

- New concurrency control algorithm.
- Keeps multiple versions.
- Identifies correct version to read from timestamp information.
- Needs garbage collection.



1V

	A	Alice	$75
	B	Bob	$92
	C		
	D	David	$106
	E		
	F	Frank	$31
	G	George	$98

MV/O

A	Alice	$75
B	Bob	$92
C		
D	David	$106
E		
F	Frank	$31
G	George	$98

1V

	A	Alice	$75
	B	Bob	$92
	C		
	D	David	$106
	E		
	F	Frank	$31
	G	George	$98

MV/O

A	Alice	$75
B	Bob	$92
C		
D	David	$106
E		
F	Frank	$31
G	George	$98

1V

🔒	A	Alice	$75
🔒	B	Bob	$92
🔒	C		
🔒	D	David	$106
🔒	E		
🔒	F	Frank	$31
🔒	G	George	$98

MV/O

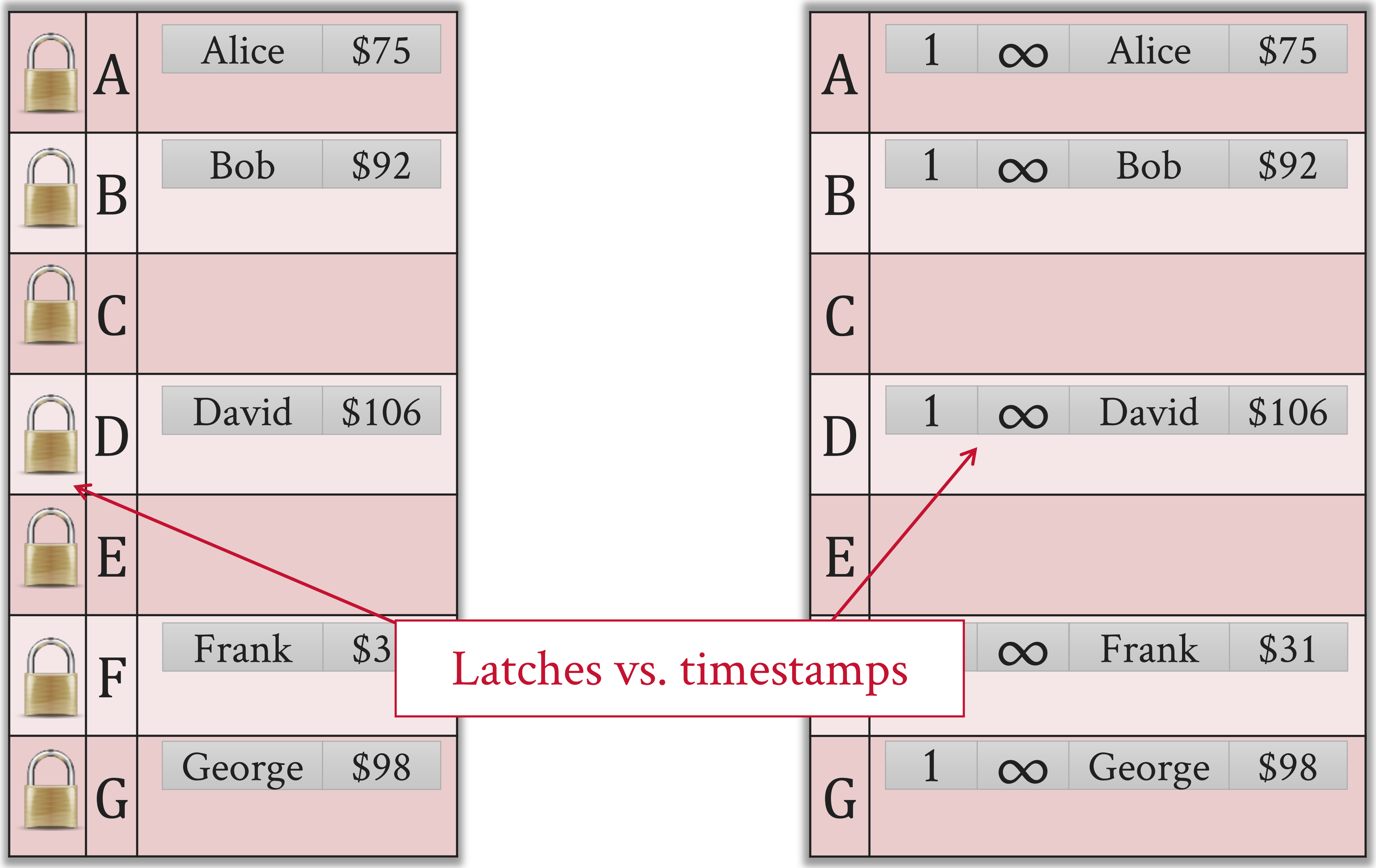
A	Alice	$75
B	Bob	$92
C		
D	David	$106
E		
F	Frank	$31
G	George	$98

1V

🔒	A	Alice	$75
🔒	B	Bob	$92
🔒	C		
🔒	D	David	$106
🔒	E		
🔒	F	Frank	$31
🔒	G	George	$98

MV/O

A	1	∞	Alice	$75
B	1	∞	Bob	$92
C				
D	1	∞	David	$106
E				
F	1	∞	Frank	$31
G	1	∞	George	$98



1V

🔒	A	Alice	$75
🔒	B	Bob	$92
🔒	C		
🔒	D	David	$106
🔒	E		
🔒	F	Frank	$31
🔒	G	George	$98

MV/O

A	1	∞	Alice	$75
B	1	∞	Bob	$92
C				
D	1	∞	David	$106
E				
F	1	∞	Frank	$31
G	1	∞	George	$98

1V

	A	Alice	$75
	B	Bob	$92
	C		
	D	David	$106
	E		
	F	Frank	$31
	G	George	$98

TX5
Read George
Read Alice
Read Bob
Update George
Commit
Postprocessing

MV/O

A	1	∞	Alice	$75
B	1	∞	Bob	$92
C				
D	1	∞	David	$106
E				
F	1	∞	Frank	$31
G	1	∞	George	$98

1V

	A	Alice	$75
	B	Bob	$92
	C		
	D	David	$106
	E		
	F	Frank	$31
🔒	G	George	$98

TX5
Read George
Read Alice
Read Bob
Update George
Commit
Postprocessing

MV/O

A	1	∞	Alice	$75
B	1	∞	Bob	$92
C				
D	1	∞	David	$106
E				
F	1	∞	Frank	$31
G	1	∞	George	$98

1V

🔒	A	Alice	$75
	B	Bob	$92
	C		
	D	David	$106
	E		
	F	Frank	$31
🔒	G	George	$98

TX5

- Read George
- Read Alice
- Read Bob
- Update George
- Commit
- Postprocessing

MV/O

A	1	∞	Alice	$75
B	1	∞	Bob	$92
C				
D	1	∞	David	$106
E				
F	1	∞	Frank	$31
G	1	∞	George	$98

1V

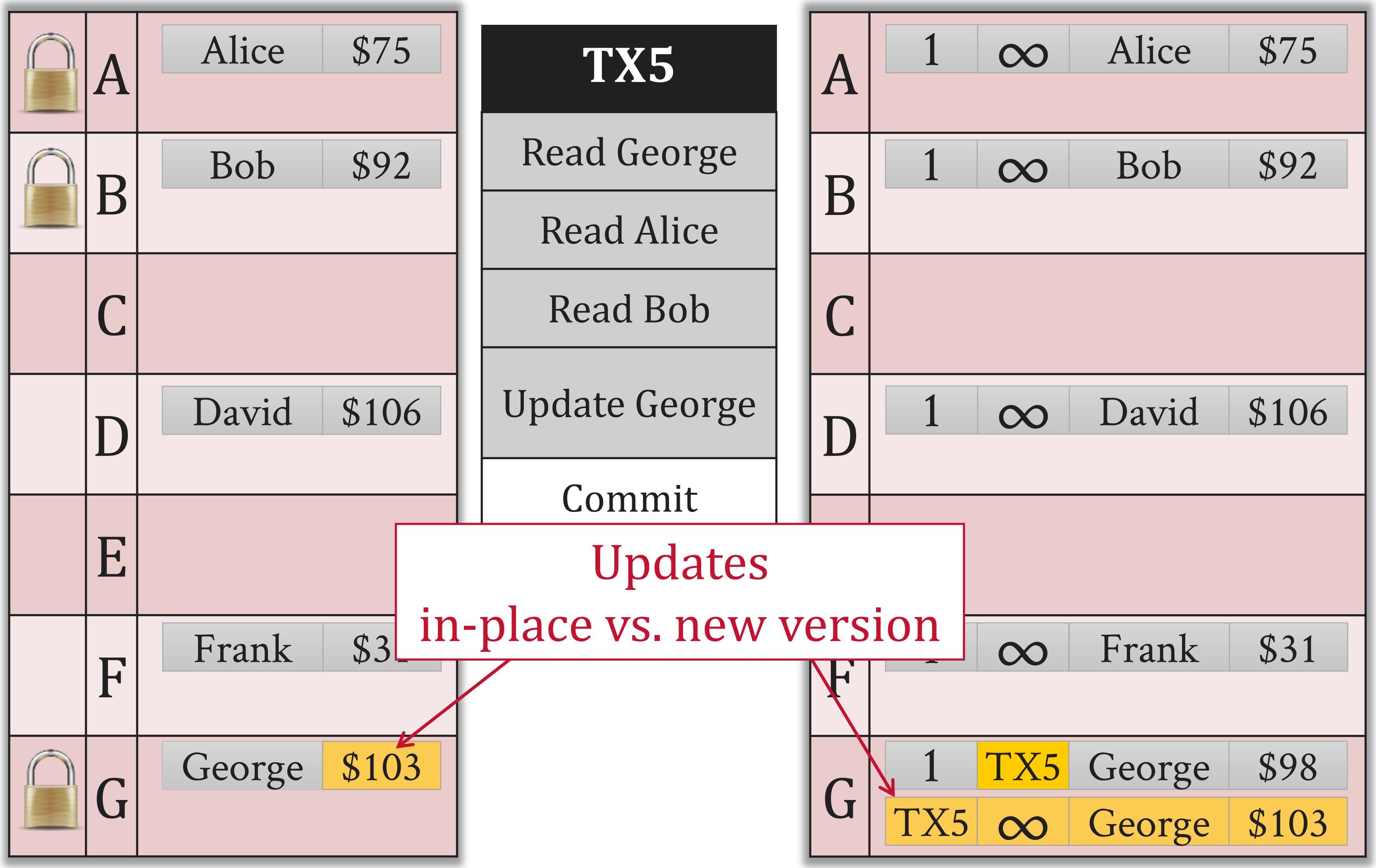
🔒	A	Alice	$75
🔒	B	Bob	$92
	C		
	D	David	$106
	E		
	F	Frank	$31
🔒	G	George	$98

TX5

- Read George
- Read Alice
- Read Bob
- Update George
- Commit
- Postprocessing

MV/O

A	1	∞	Alice	$75
B	1	∞	Bob	$92
C				
D	1	∞	David	$106
E				
F	1	∞	Frank	$31
G	1	∞	George	$98



1V

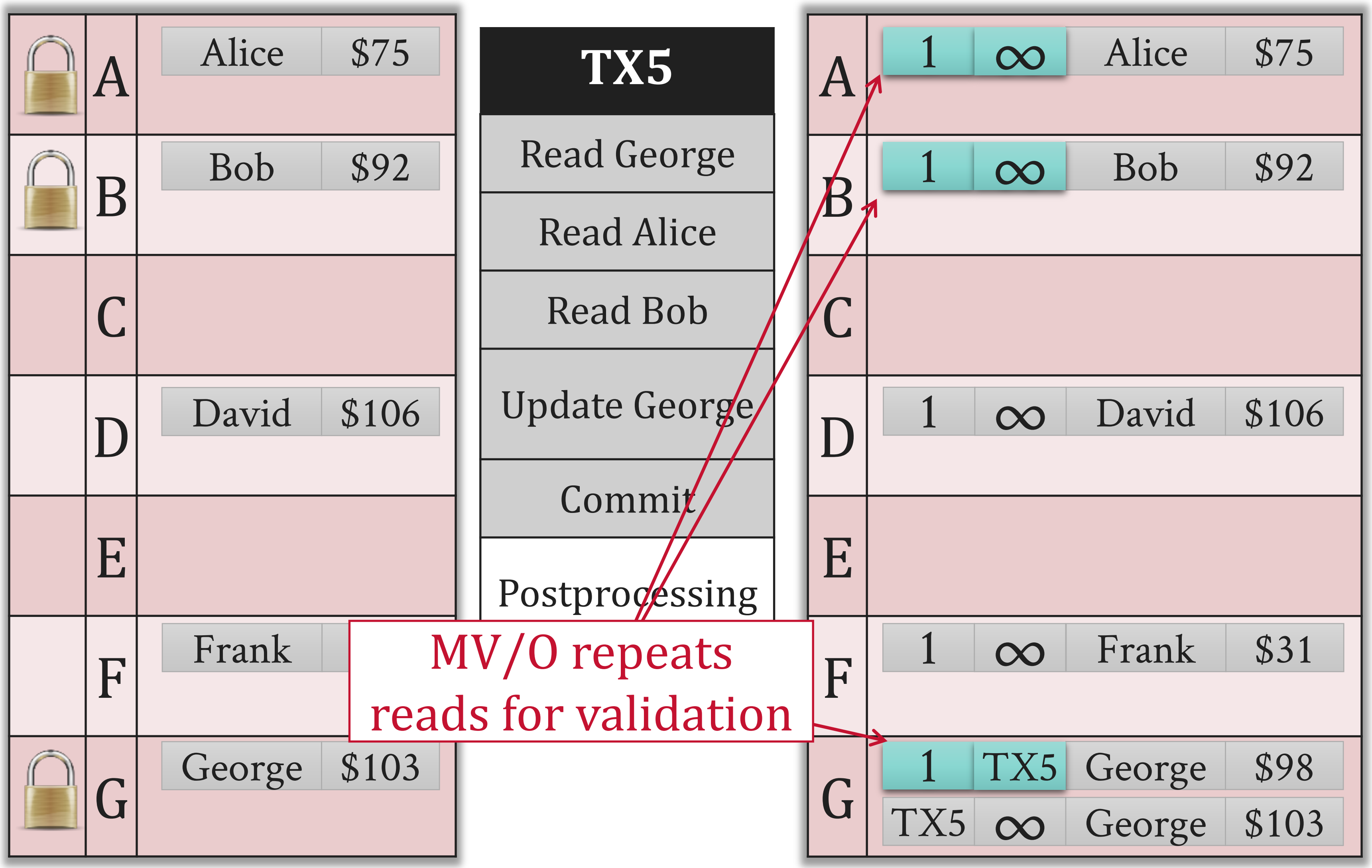
🔒	A	Alice	$75
🔒	B	Bob	$92
	C		
	D	David	$106
	E		
	F	Frank	$31
🔒	G	George	$103

TX5
Read George
Read Alice
Read Bob
Update George
Commit

MV/O

A	1	∞	Alice	$75
B	1	∞	Bob	$92
C				
D	1	∞	David	$106
E				
F	1	∞	Frank	$31
G	1	TX5	George	$98
	TX5	∞	George	$103

Updates
in-place vs. new version



1V

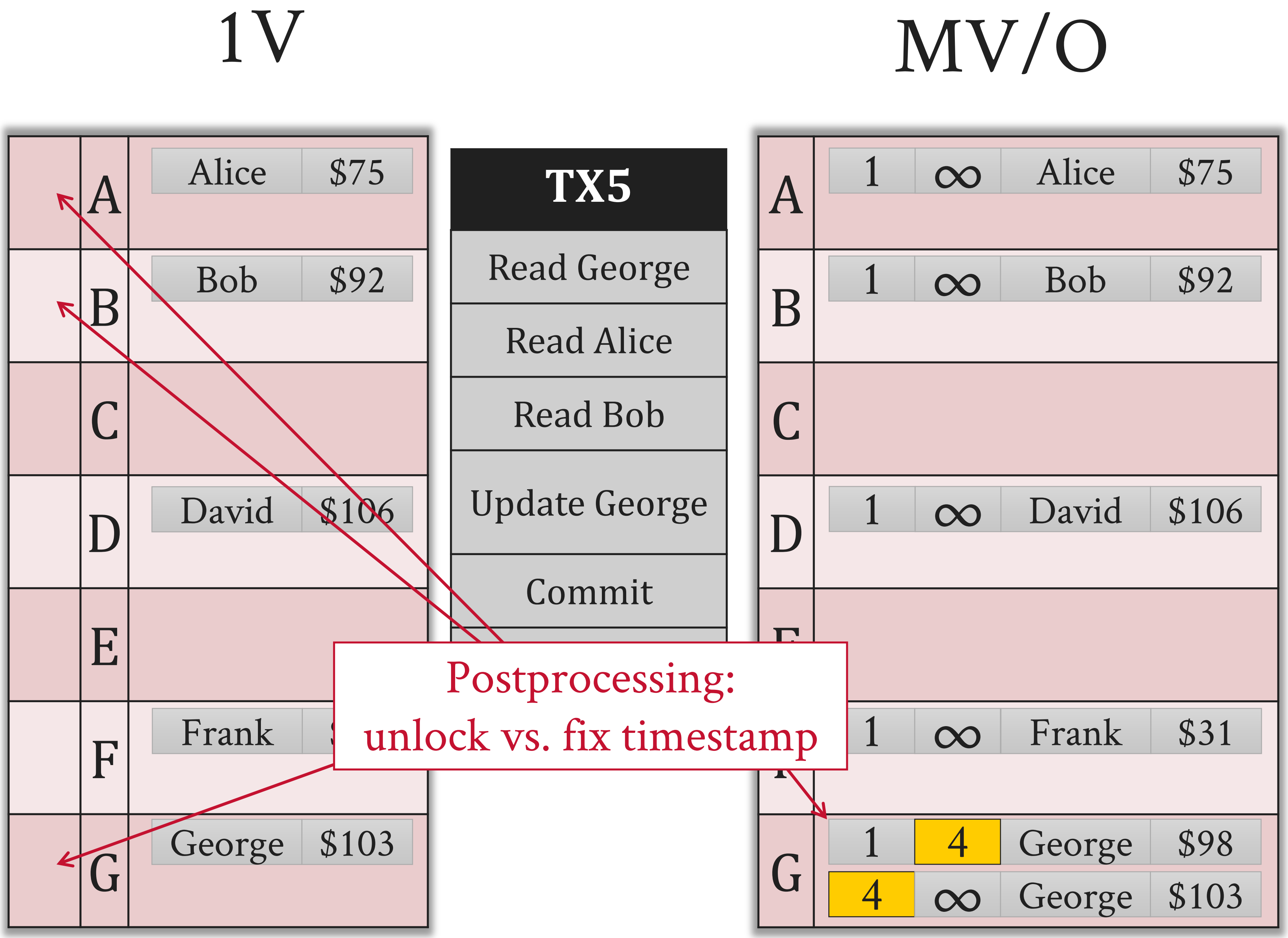
🔒	A	Alice	$75
🔒	B	Bob	$92
	C		
	D	David	$106
	E		
	F	Frank	
🔒	G	George	$103

TX5
Read George
Read Alice
Read Bob
Update George
Commit
Postprocessing

MV/O

A	1	∞	Alice	$75
B	1	∞	Bob	$92
C				
D	1	∞	David	$106
E				
F	1	∞	Frank	$31
G	1	TX5	George	$98
	TX5	∞	George	$103

MV/O repeats reads for validation



1V

	A	Alice	$75
	B	Bob	$92
	C		
	D	David	$106
	E		
	F	Frank	$
	G	George	$103

TX5
Read George
Read Alice
Read Bob
Update George
Commit

MV/O

A	1	∞	Alice	$75
B	1	∞	Bob	$92
C				
D	1	∞	David	$106
E				
F	1	∞	Frank	$31
G	1	4	George	$98
	4	∞	George	$103

Postprocessing:
unlock vs. fix timestamp

MEMORY ACCESSES ON CRITICAL PATH

1V

- Read operation:

  1 mem read to record.

  1 mem write to lock.

- Update operation:

  1 mem write to record.

In 1V, readers write to memory!

MV/O

- Read operation:

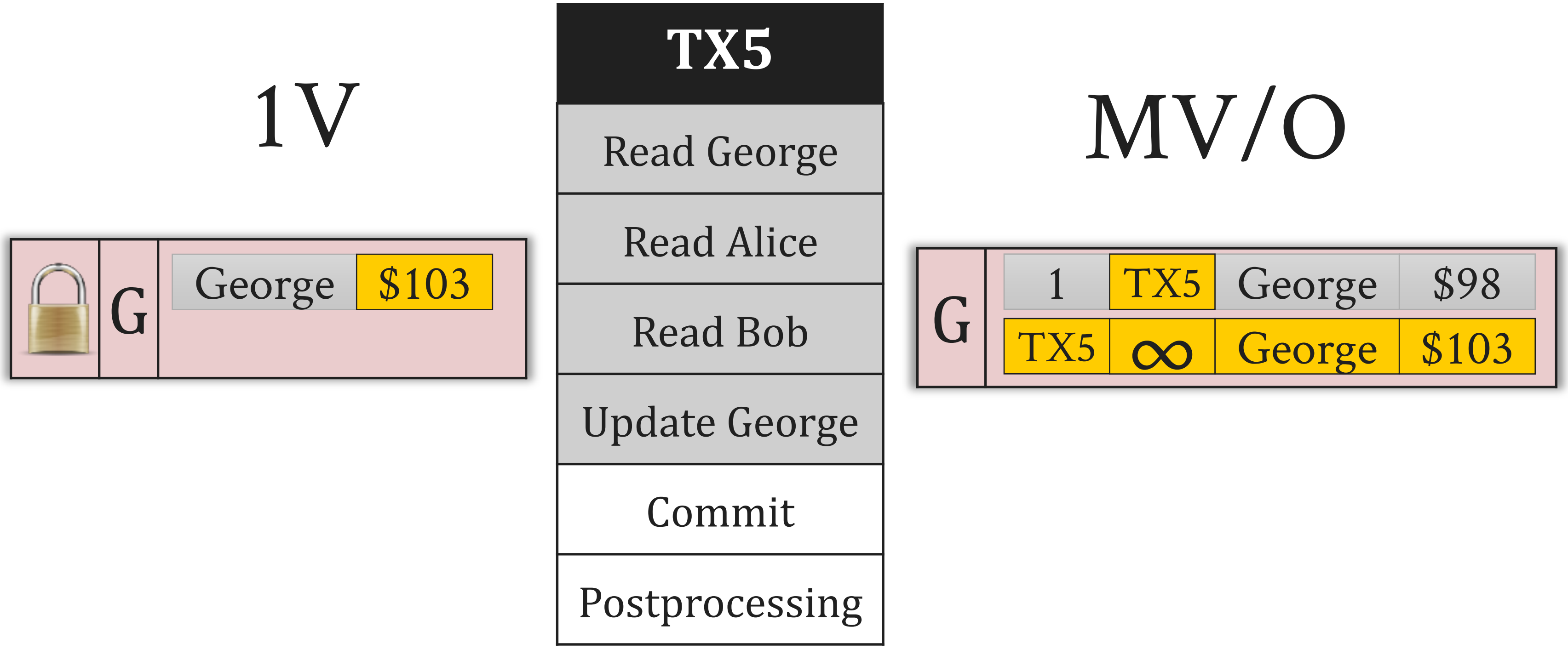
  1 mem read to version.

- Update operation:

  1 mem write to new version.

  1 mem write to old version.

RW Conflicts



1V

🔒	G	George	$103

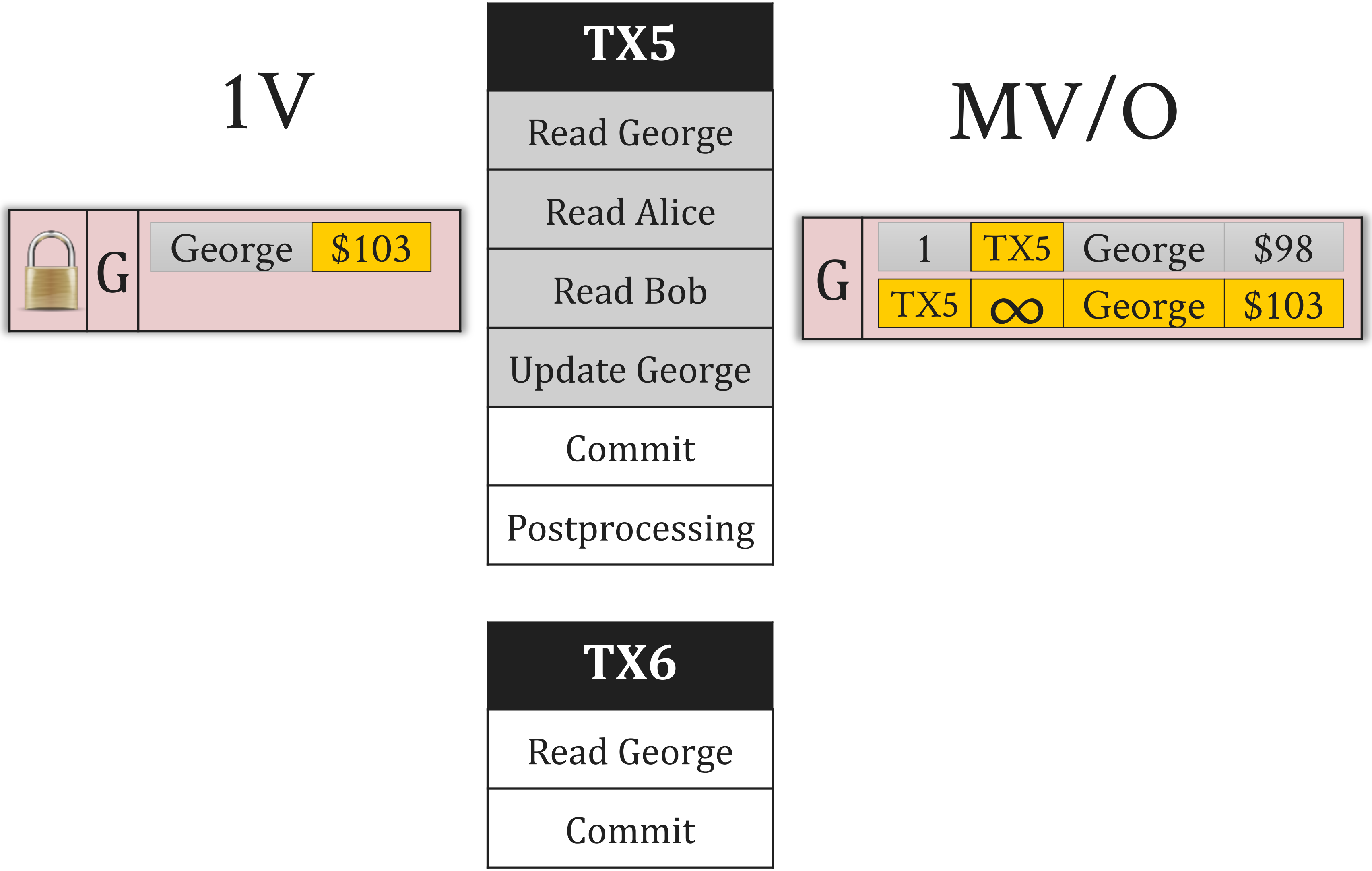
TX5

- Read George
- Read Alice
- Read Bob
- Update George
- Commit
- Postprocessing

MV/O

G	1	TX5	George	$98
	TX5	∞	George	$103

RW Conflicts



1V

🔒	G	George	$103

TX5

- Read George
- Read Alice
- Read Bob
- Update George
- Commit
- Postprocessing

MV/O

G	1	TX5	George	$98
	TX5	∞	George	$103

TX6

- Read George
- Commit

RW Conflicts

1V

🔒	G	George	$103

TX6 waits for lock

TX5

- Read George
- Read Alice
- Read Bob
- Update George
- Commit
- Postprocessing

MV/O

G	1	TX5	George	$98
	TX5	∞	George	$103

TX6 reads old version and commits

TX6

- Read George
- Commit

RW Conflicts

1V

🔒	G	George	$103

TX6 waits for lock

TX5
Read George
Read Alice
Read Bob
Update George
Commit
Postprocessing

MV/O

G	1	TX5	George	$98
	TX5	∞	George	$103

TX6 reads old version and commits

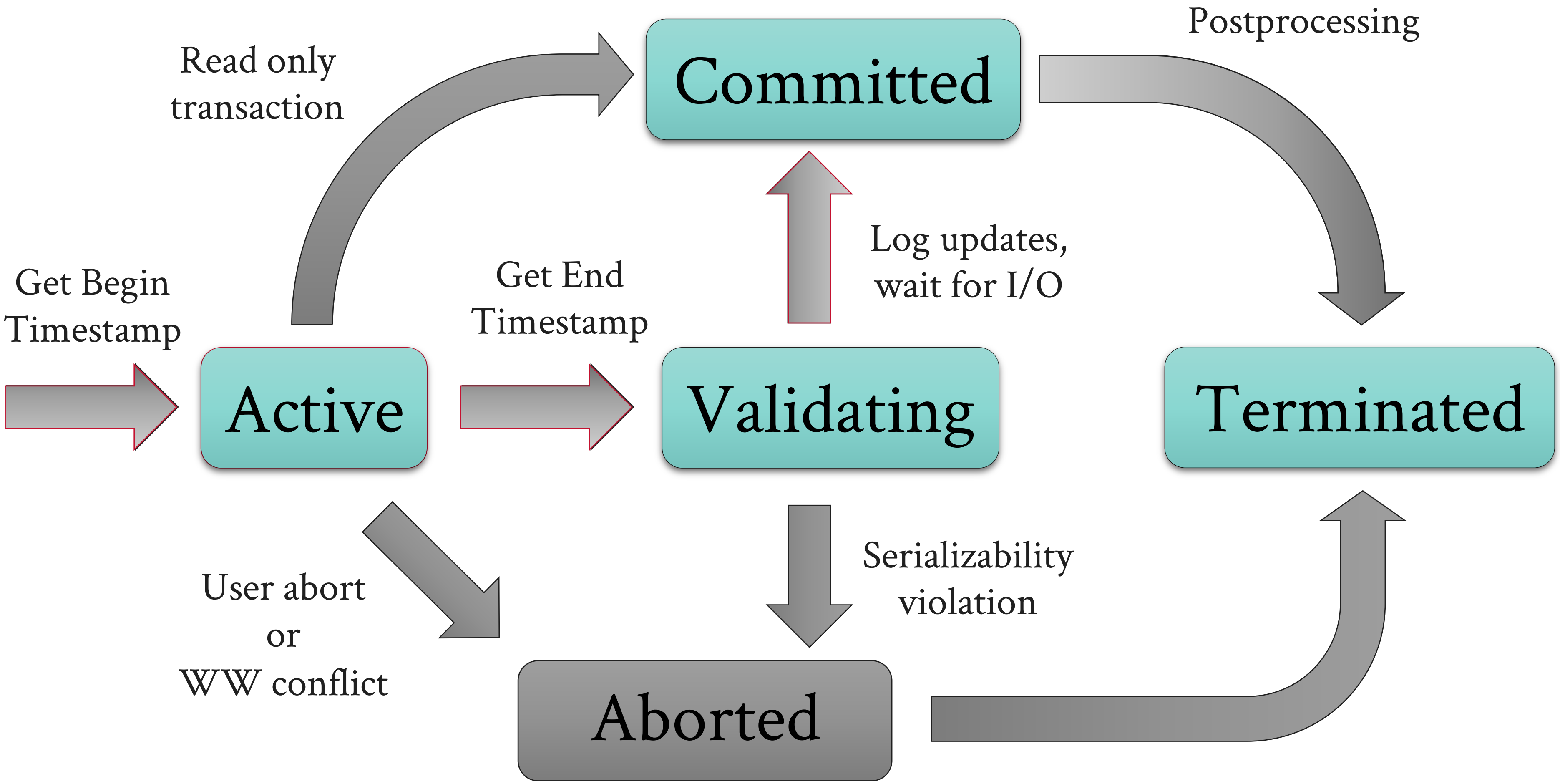
TX6
Read George
Commit

MV/O isolates readers from writers

MULTI-VERSION OPTIMISTIC SUMMARY

- There are no latches or locks:
  - Txn reads don't cause memory writes.
  - Txns will never wait during the ACTIVE phase.
- Isolates readers from writers.
- Supports all isolation levels.
  - Lower isolation level = less work.
- No deadlock detection is needed.

TRANSACTION STATES



Transaction Map

- Stores transaction state, timestamps.
- Globally visible.

Transaction Map

TXID	STATE	BEGIN	END
5	N/A	N/A	N/A

Transaction Map

- Stores transaction state, timestamps.
- Globally visible.

Transaction Map

TXID	STATE	BEGIN	END
5	N/A	2	N/A

TRANSACTION MAP

- Stores transaction state, timestamps.
- Globally visible.

Transaction Map

TXID	STATE	BEGIN	END
5	ACTIV	2	N/A

Transaction Map

- Stores transaction state, timestamps.
- Globally visible.

Transaction Map

TXID	STATE	BEGIN	END
5	ACTIV	2	N/A

Transaction Map

- Stores transaction state, timestamps.
- Globally visible.

Transaction Map

TXID	STATE	BEGIN	END
5	ACTIV	2	4

Transaction Map

- Stores transaction state, timestamps.
- Globally visible.

Transaction Map

TXID	STATE	BEGIN	END
5	VALID	2	4

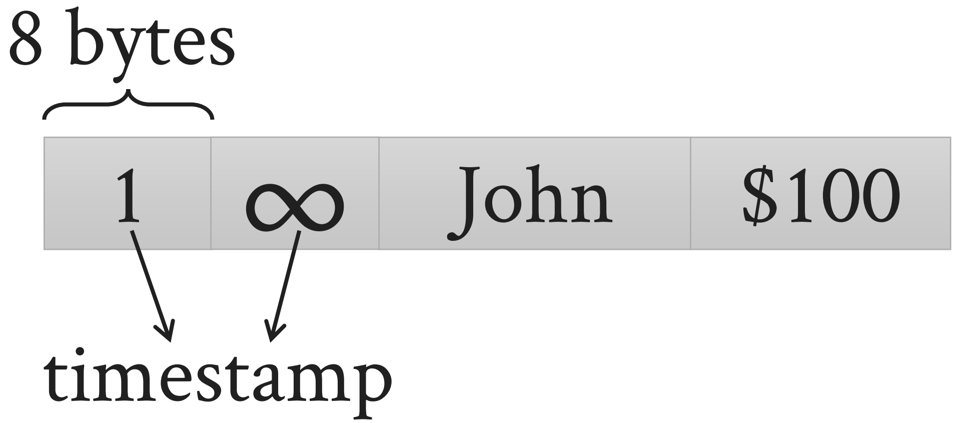
Transaction Map

- Stores transaction state, timestamps.
- Globally visible.

Transaction Map

TXID	STATE	BEGIN	END
5	COM	2	4

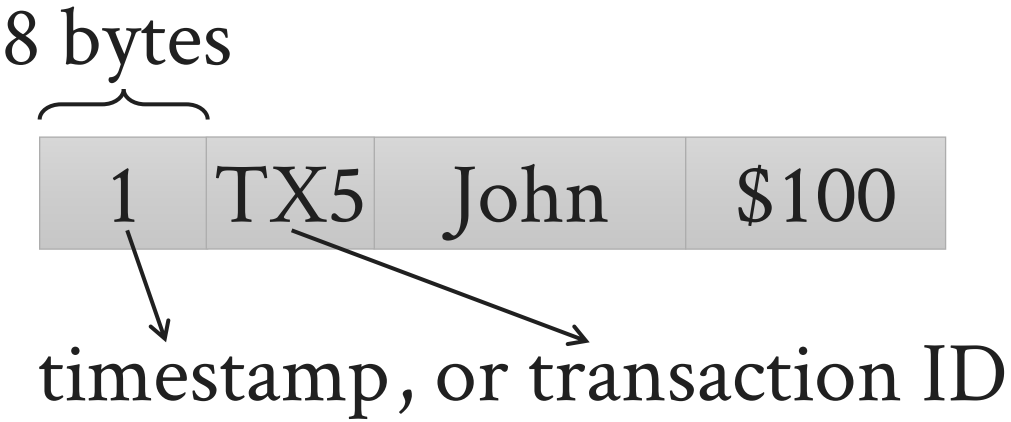
Determining Version Visibility



Transaction Map

TXID	STATE	BEGIN	END
5	ACTIV	2	N/A

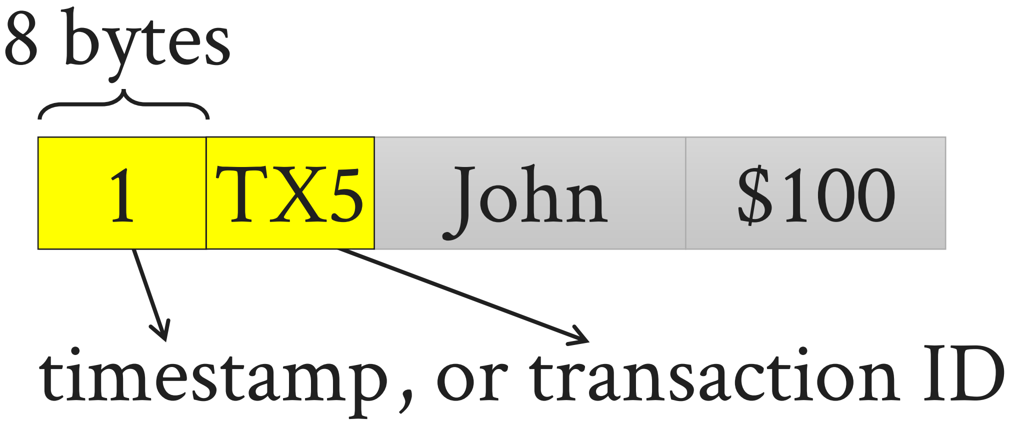
DETERMINING VERSION VISIBILITY



Transaction Map

TXID	STATE	BEGIN	END
5	ACTIV	2	N/A

DETERMINING VERSION VISIBILITY

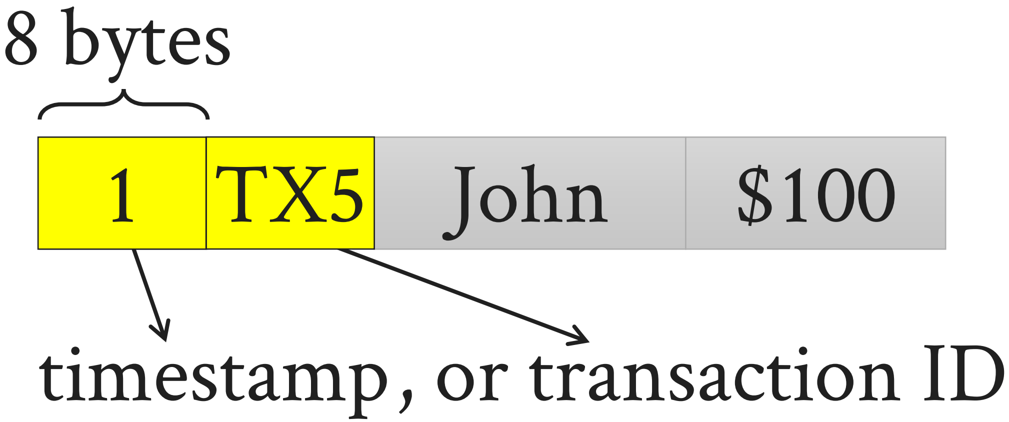


Transaction Map

TXID	STATE	BEGIN	END
5	ACTIV	2	N/A

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

DETERMINING VERSION VISIBILITY



Transaction Map

TXID	STATE	BEGIN	END
5	ACTIV	2	N/A

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

Generate timestamps efficiently using Atomic Addition (CAS).

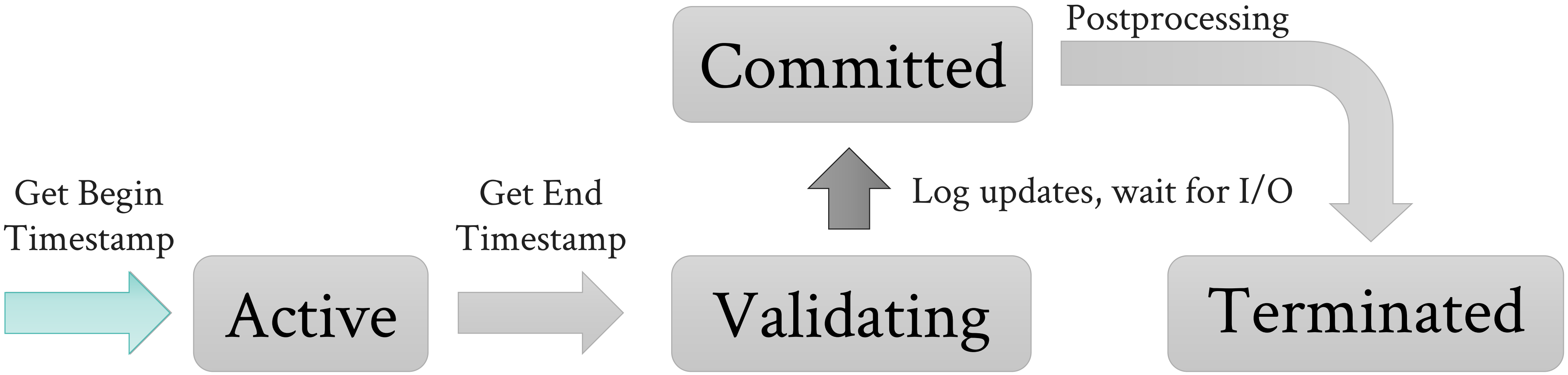
Can also use a hardware clock (see previous lectures).

EXAMPLE: UPDATE TO $150

1	∞	John	$100

Transaction Map

TXID	STATE	BEGIN	END
5	N/A	N/A	N/A

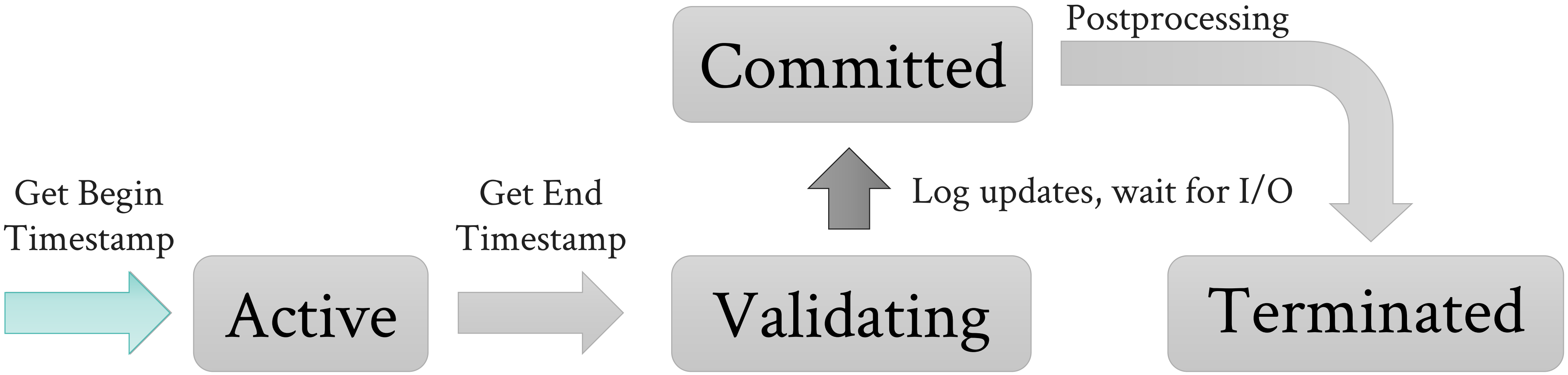


EXAMPLE: UPDATE TO $150

1	∞	John	$100

Transaction Map

TXID	STATE	BEGIN	END
5	N/A	2	N/A

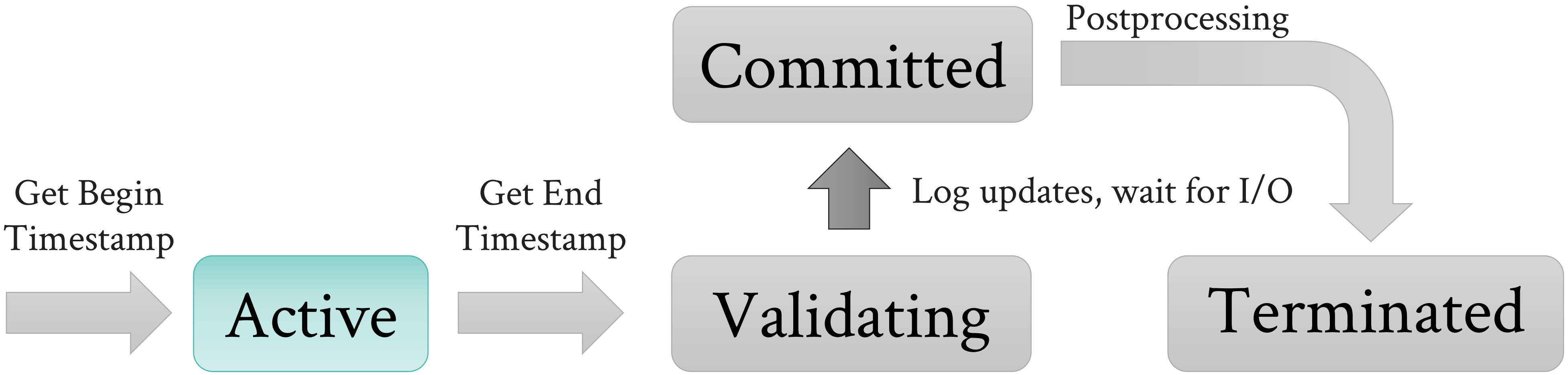


EXAMPLE: UPDATE TO $150

1	∞	John	$100

Transaction Map

TXID	STATE	BEGIN	END
5	N/A	2	N/A



Get Begin Timestamp

Active

Get End Timestamp

Validating

Log updates, wait for I/O

Committed

Postprocessing

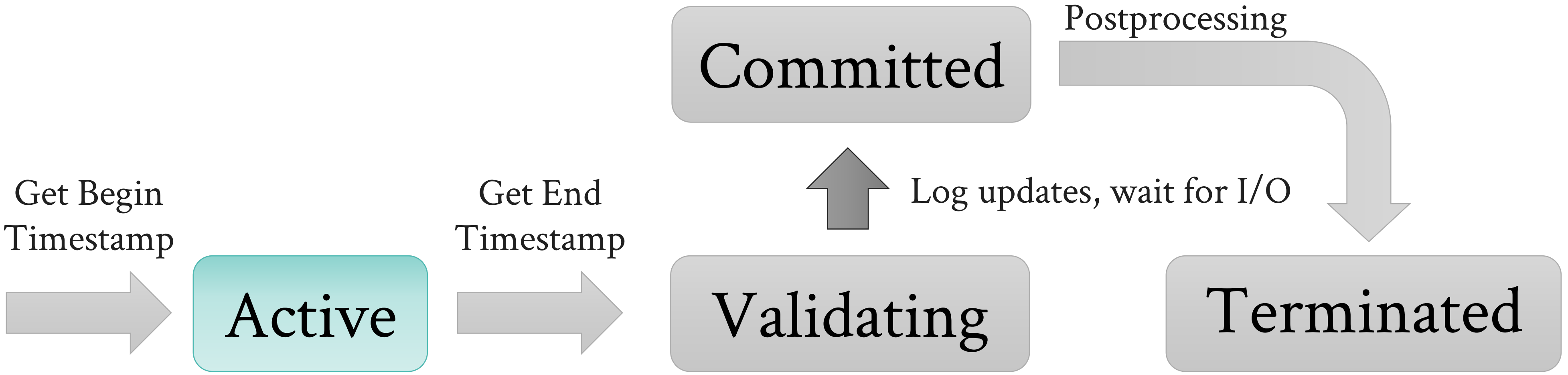
Terminated

Example: Update to $150

1	∞	John	$100

Transaction Map

TXID	STATE	BEGIN	END
5	ACTIV	2	N/A



Get Begin Timestamp

Active

Get End Timestamp

Validating

Log updates, wait for I/O

Committed

Postprocessing

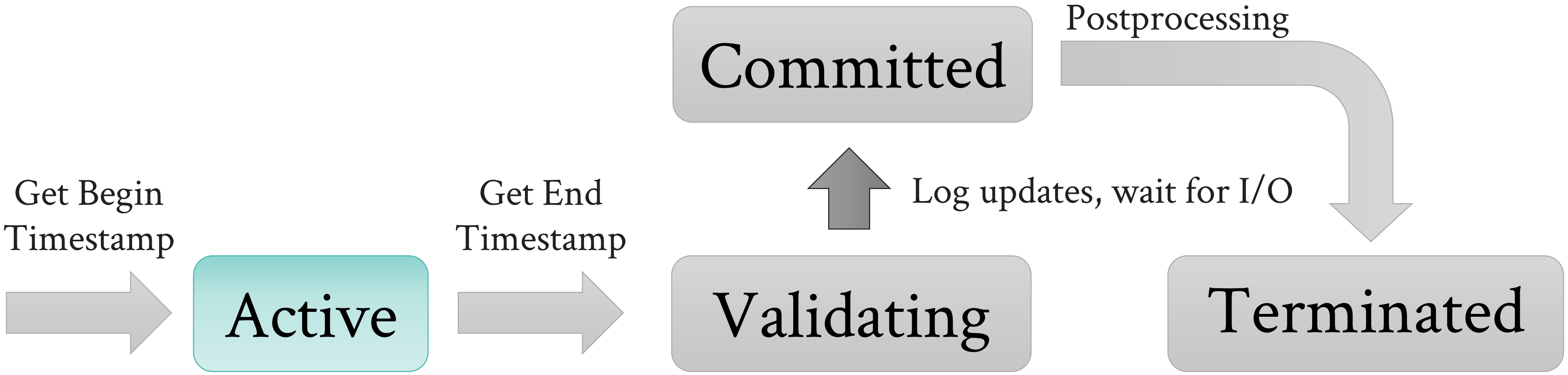
Terminated

Example: Update to $150

1	∞	John	$100

Transaction Map

TXID	STATE	BEGIN	END
5	ACTIV	2	N/A

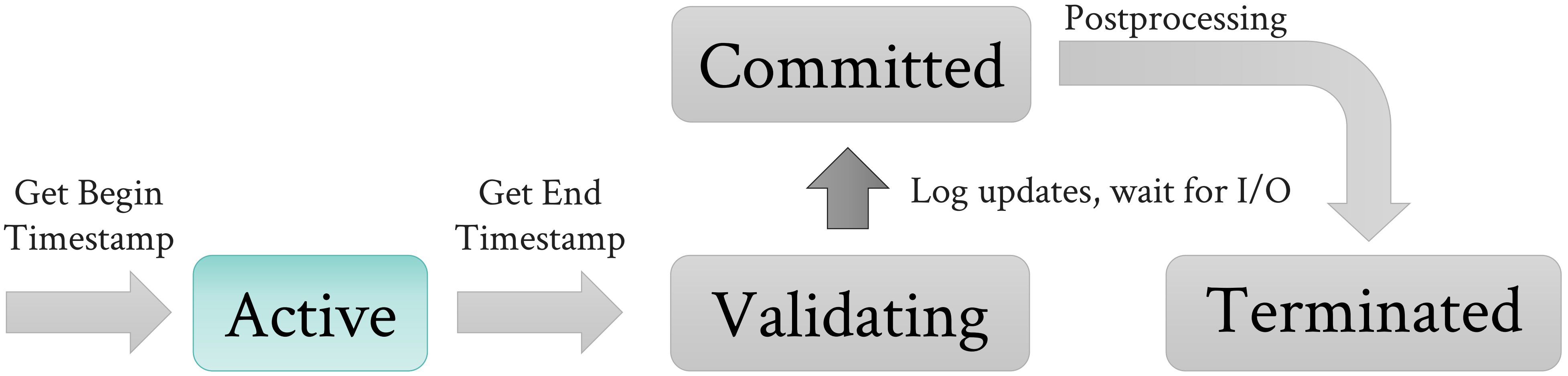


EXAMPLE: UPDATE TO $150

1	TX5	John	$100

Transaction Map

TXID	STATE	BEGIN	END
5	ACTIV	2	N/A



Get Begin Timestamp

Active

Get End Timestamp

Validating

Log updates, wait for I/O

Committed

Postprocessing

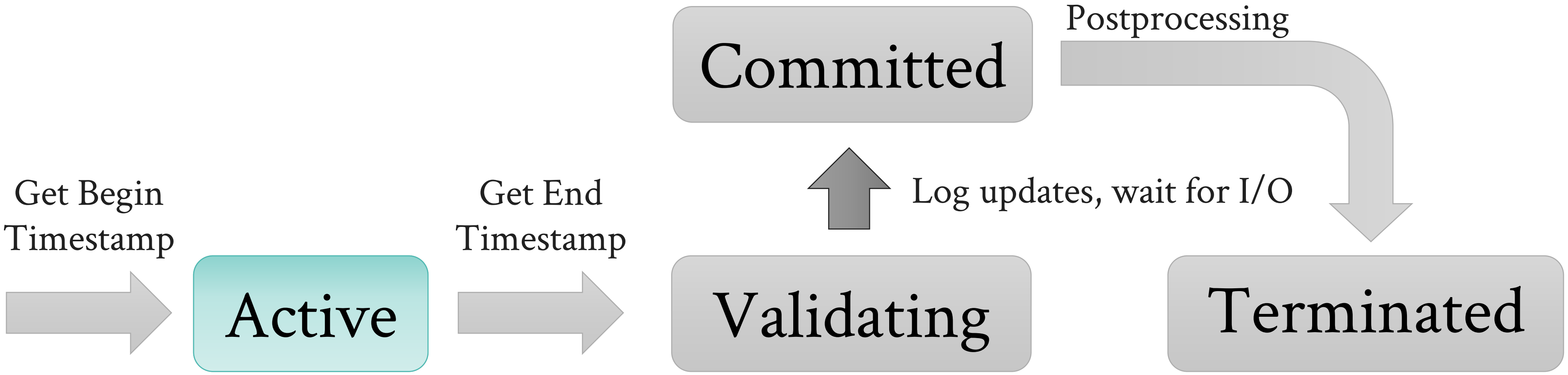
Terminated

Example: Update to $150

1	TX5	John	$100
TX5	∞	John	$150

Transaction Map

TXID	STATE	BEGIN	END
5	ACTIV	2	N/A

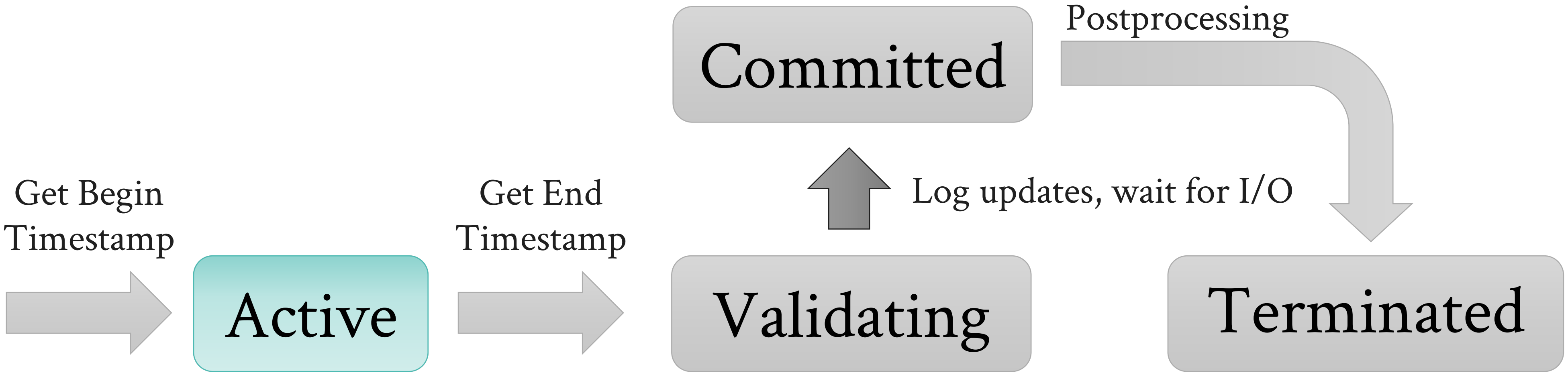


EXAMPLE: UPDATE TO $150

1	TX5	John	$100
TX5	∞	John	$150

Transaction Map

TXID	STATE	BEGIN	END
5	ACTIV	2	N/A

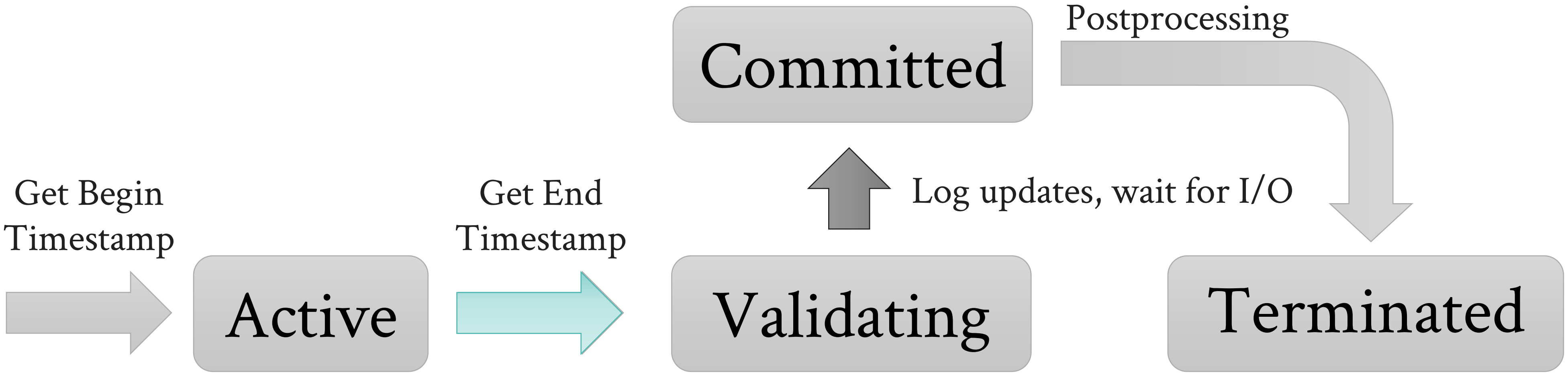


EXAMPLE: UPDATE TO $150

1	TX5	John	$100
TX5	∞	John	$150

Transaction Map

TXID	STATE	BEGIN	END
5	ACTIV	2	N/A

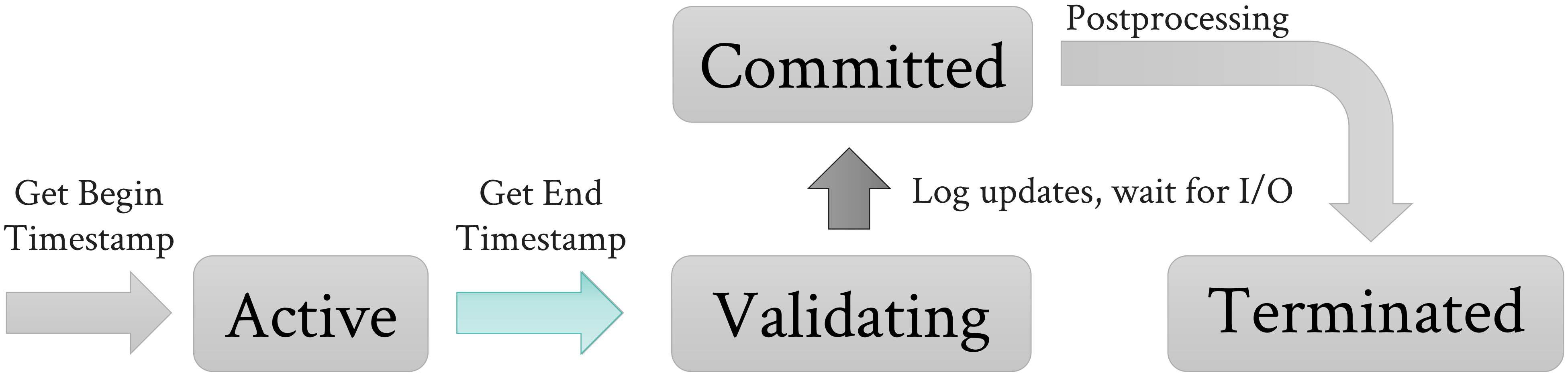


EXAMPLE: UPDATE TO $150

1	TX5	John	$100
TX5	∞	John	$150

Transaction Map

TXID	STATE	BEGIN	END
5	ACTIV	2	4

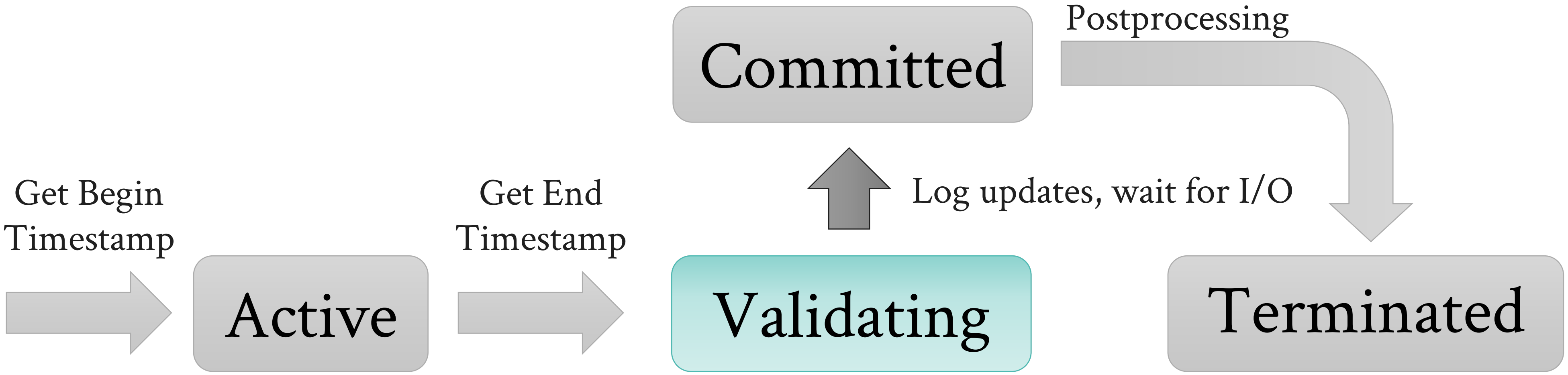


EXAMPLE: UPDATE TO $150

1	TX5	John	$100
TX5	∞	John	$150

Transaction Map

TXID	STATE	BEGIN	END
5	ACTIV	2	4

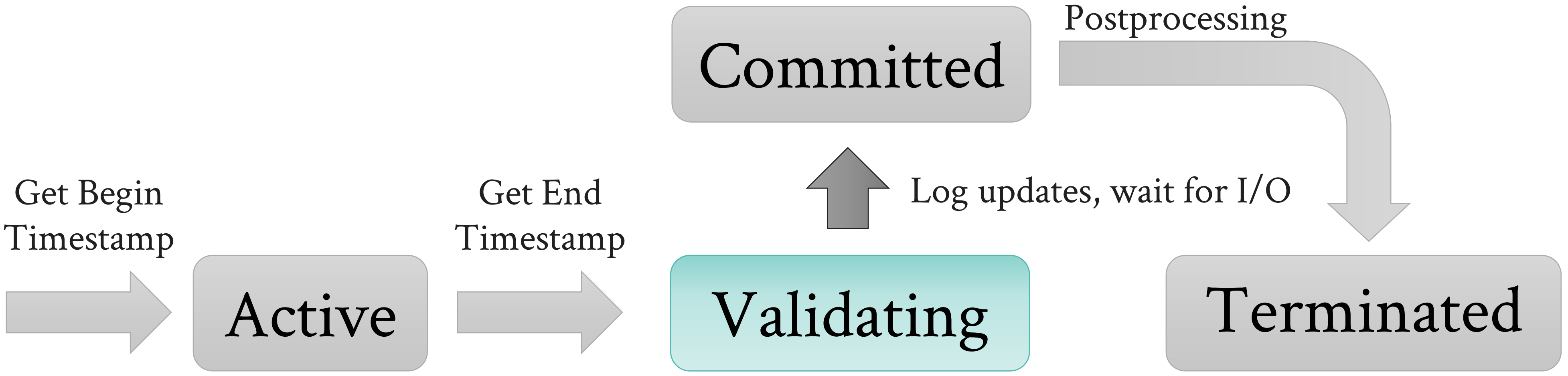


Example: Update to $150

1	TX5	John	$100
TX5	∞	John	$150

Transaction Map

TXID	STATE	BEGIN	END
5	VALID	2	4

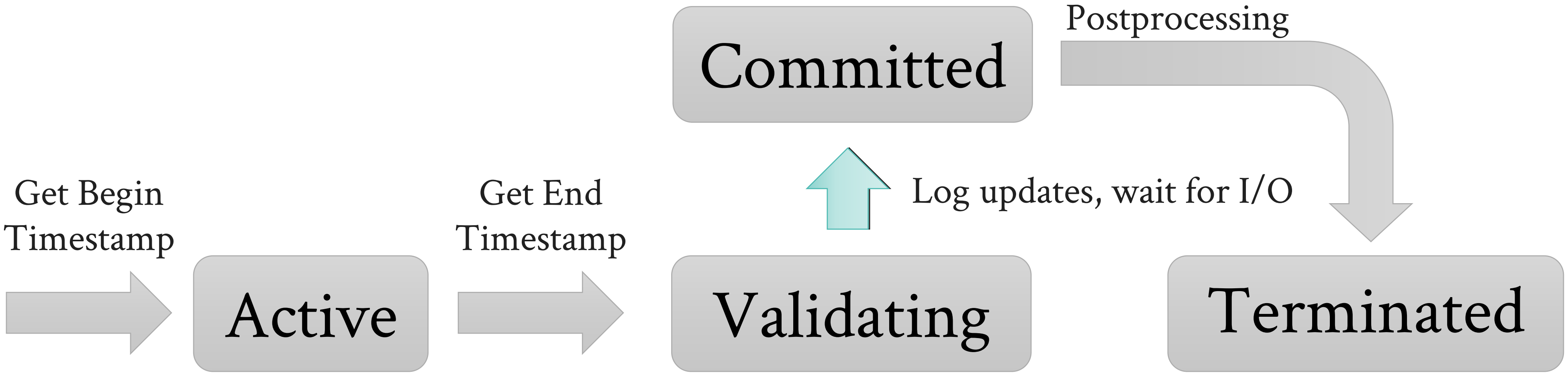


EXAMPLE: UPDATE TO $150

1	TX5	John	$100
TX5	∞	John	$150

Transaction Map

TXID	STATE	BEGIN	END
5	VALID	2	4



Get Begin Timestamp

Active

Get End Timestamp

Validating

Log updates, wait for I/O

Committed

Postprocessing

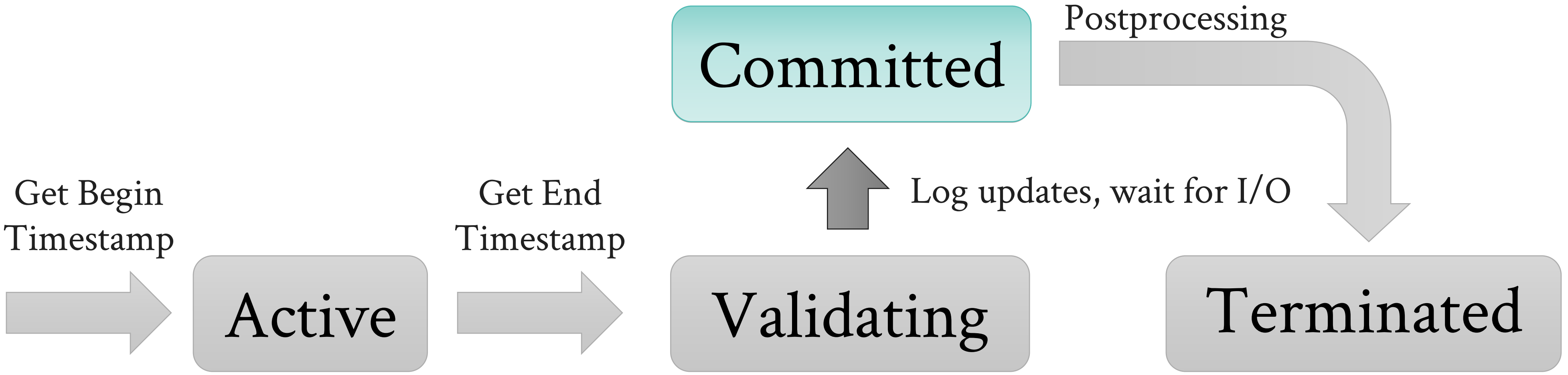
Terminated

Example: Update to $150

1	TX5	John	$100
TX5	∞	John	$150

Transaction Map

TXID	STATE	BEGIN	END
5	VALID	2	4

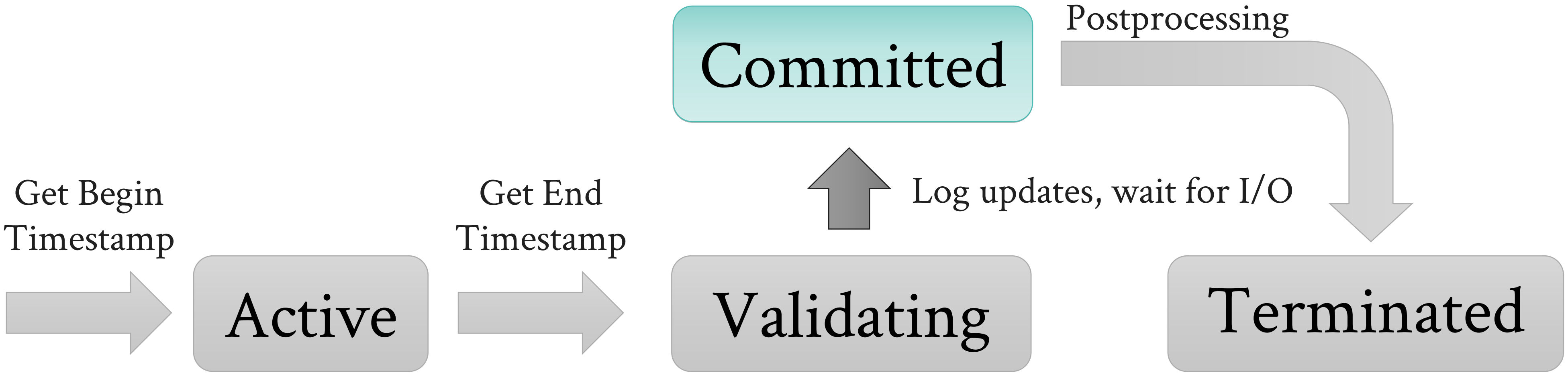


EXAMPLE: UPDATE TO $150

1	TX5	John	$100
TX5	∞	John	$150

Transaction Map

TXID	STATE	BEGIN	END
5	COM	2	4

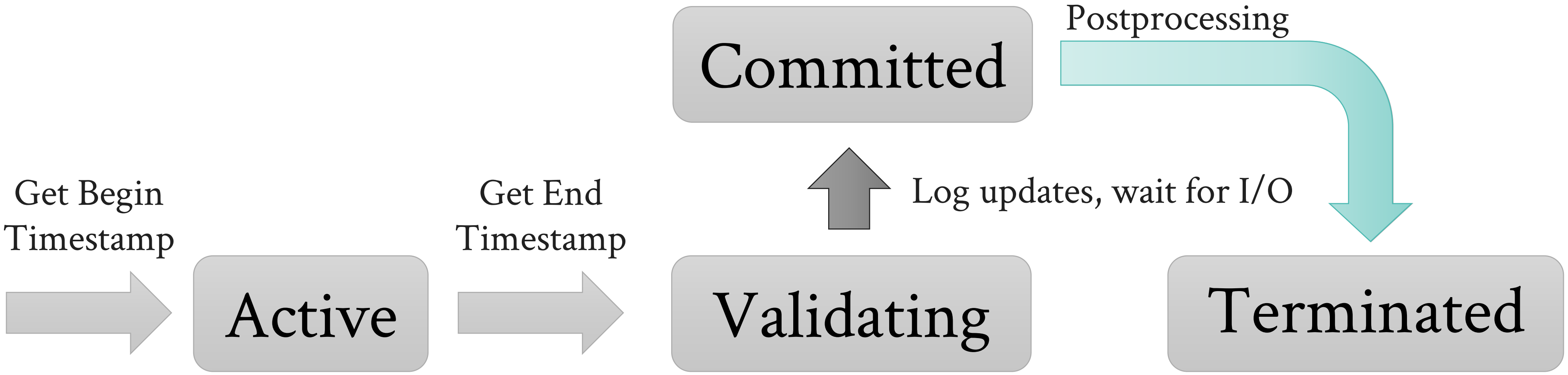


EXAMPLE: UPDATE TO $150

1	TX5	John	$100
TX5	∞	John	$150

Transaction Map

TXID	STATE	BEGIN	END
5	COM	2	4

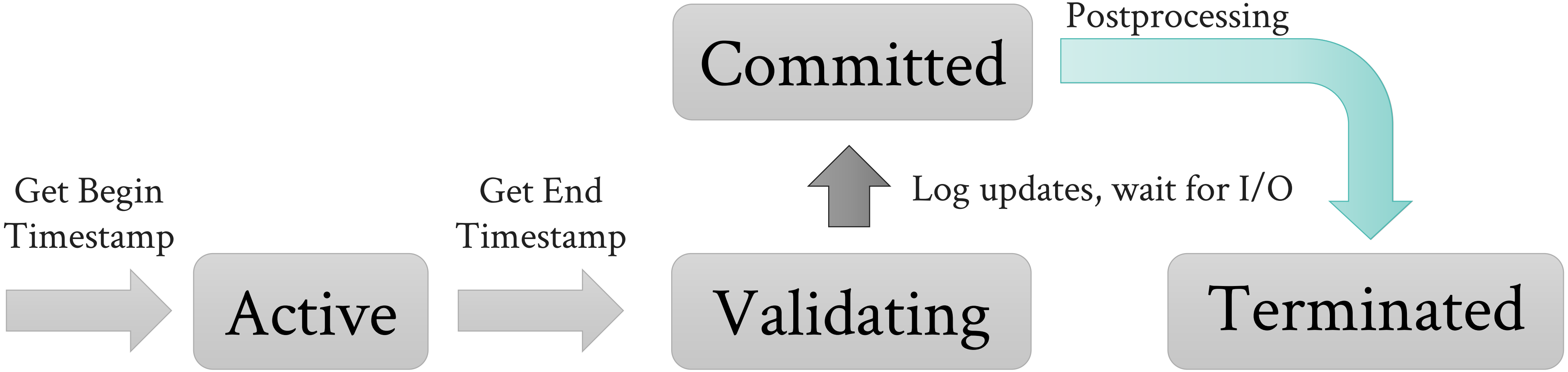


EXAMPLE: UPDATE TO $150

1	TX5	John	$100
TX5	∞	John	$150

Transaction Map

TXID	STATE	BEGIN	END
5	COM	2	4

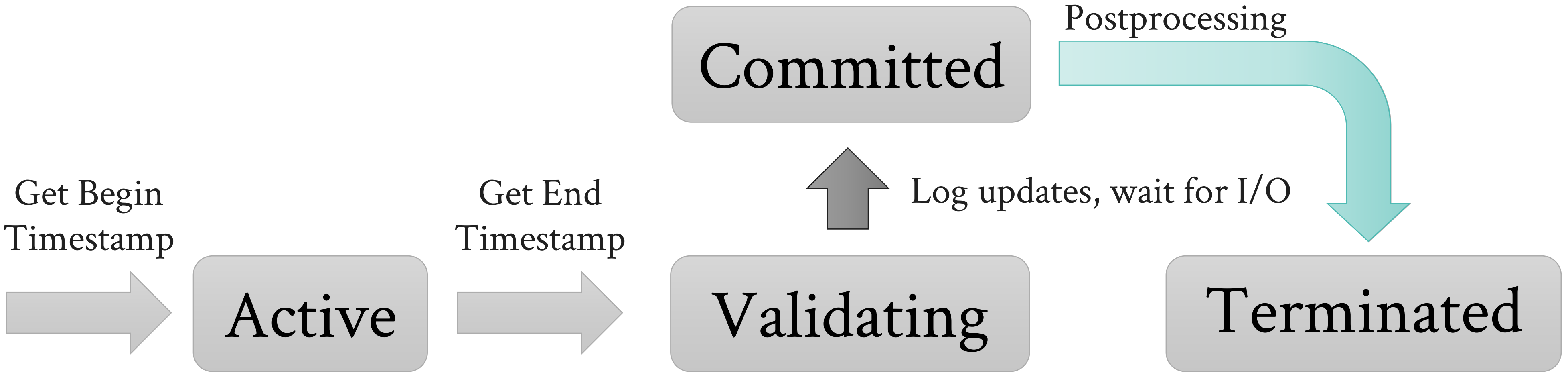


Example: Update to $150

1	4	John	$100
TX5	∞	John	$150

Transaction Map

TXID	STATE	BEGIN	END
5	COM	2	4

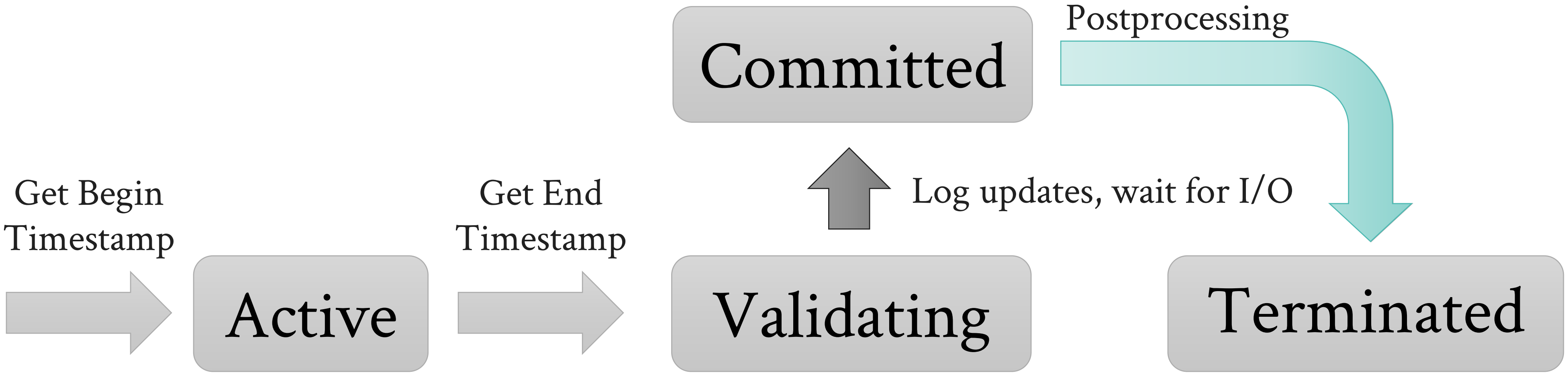


EXAMPLE: UPDATE TO $150

1	4	John	$100
4	∞	John	$150

Transaction Map

TXID	STATE	BEGIN	END
5	COM	2	4

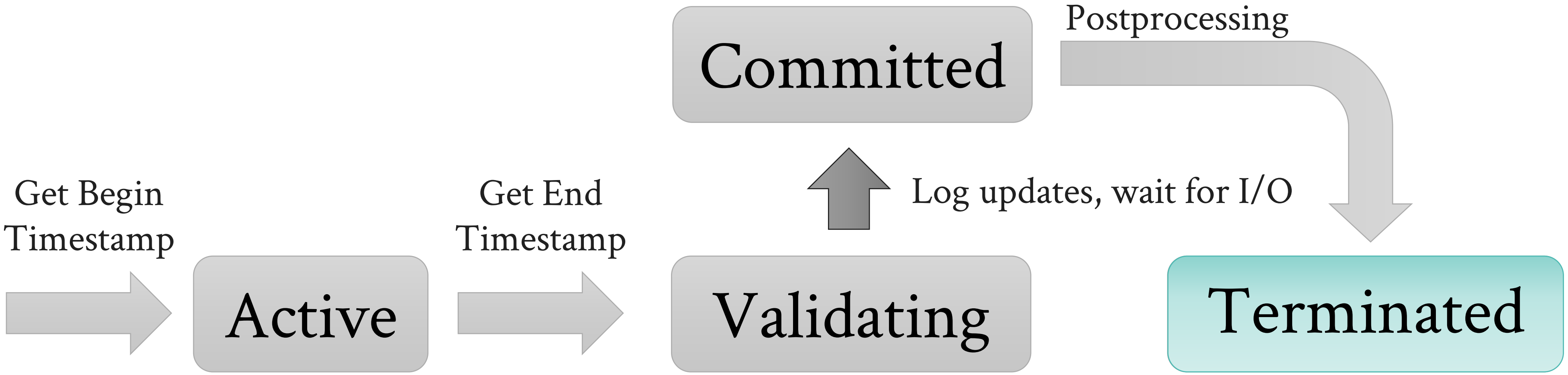


EXAMPLE: UPDATE TO $150

1	4	John	$100
4	∞	John	$150

Transaction Map

TXID	STATE	BEGIN	END
5	COM	2	4



Get Begin Timestamp

Active

Get End Timestamp

Validating

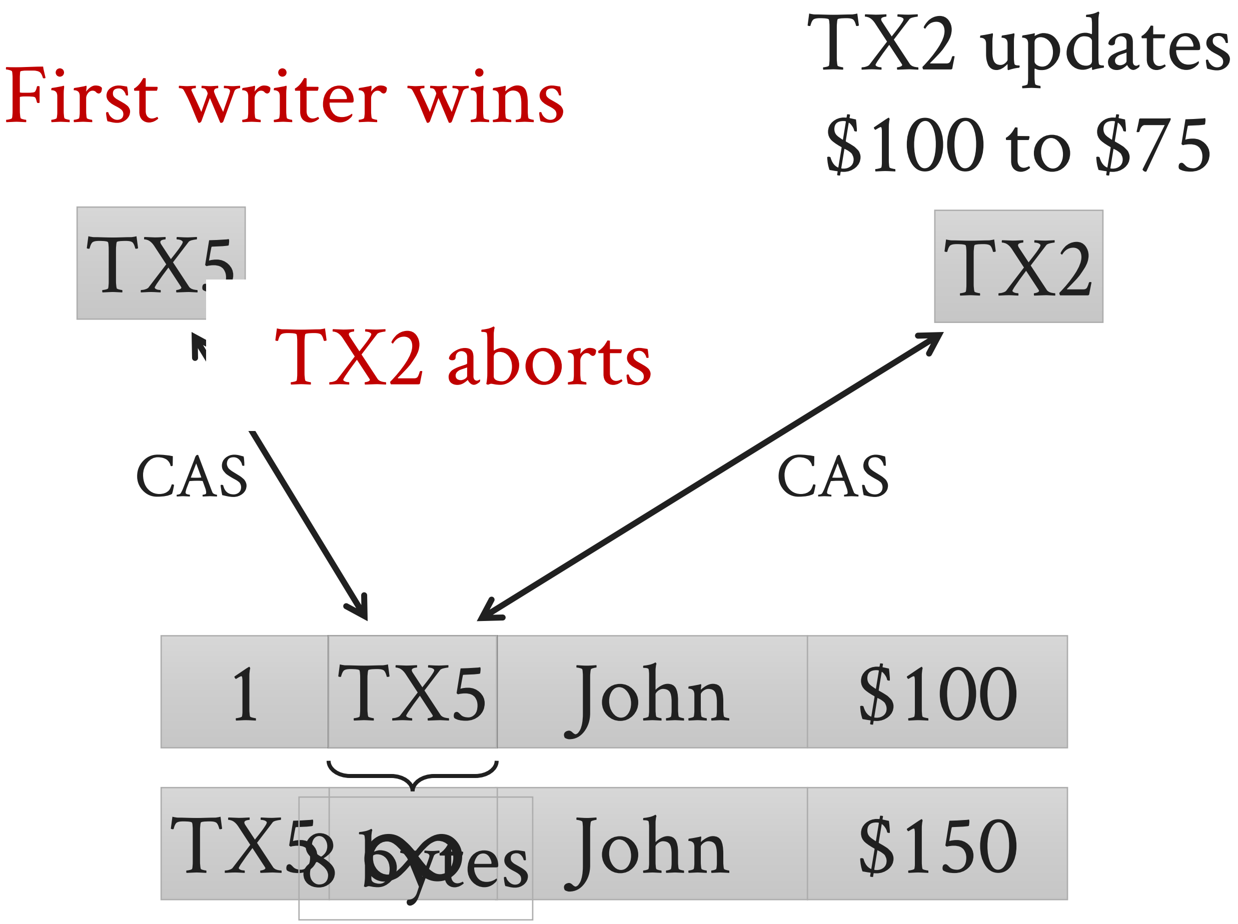
Log updates, wait for I/O

Committed

Postprocessing

Terminated

WW Conflicts



First writer wins

TX2 updates
$100 to $75

TX5

TX2

TX2 aborts

CAS

CAS

1	TX5	John	$100
TX5	∞	John	$150

8 bytes

WR Conflicts

TX5	∞	John	$150

Q: When is a version visible?

A: Depends on the txn state.

WR Conflicts

TX5	∞	John	$150

Q: When is a version visible?

A: Depends on the txn state.

TX5 State	Visible?
ACTIVE	No, the version is uncommitted.
VALIDATING	?
COMMITTED	Maybe, check TX5 END timestamp.
ABORTED	No, this version is garbage.

WR Conflicts

TX5	∞	John	$150

Q: When is a version visible?

A: Depends on the txn state.

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

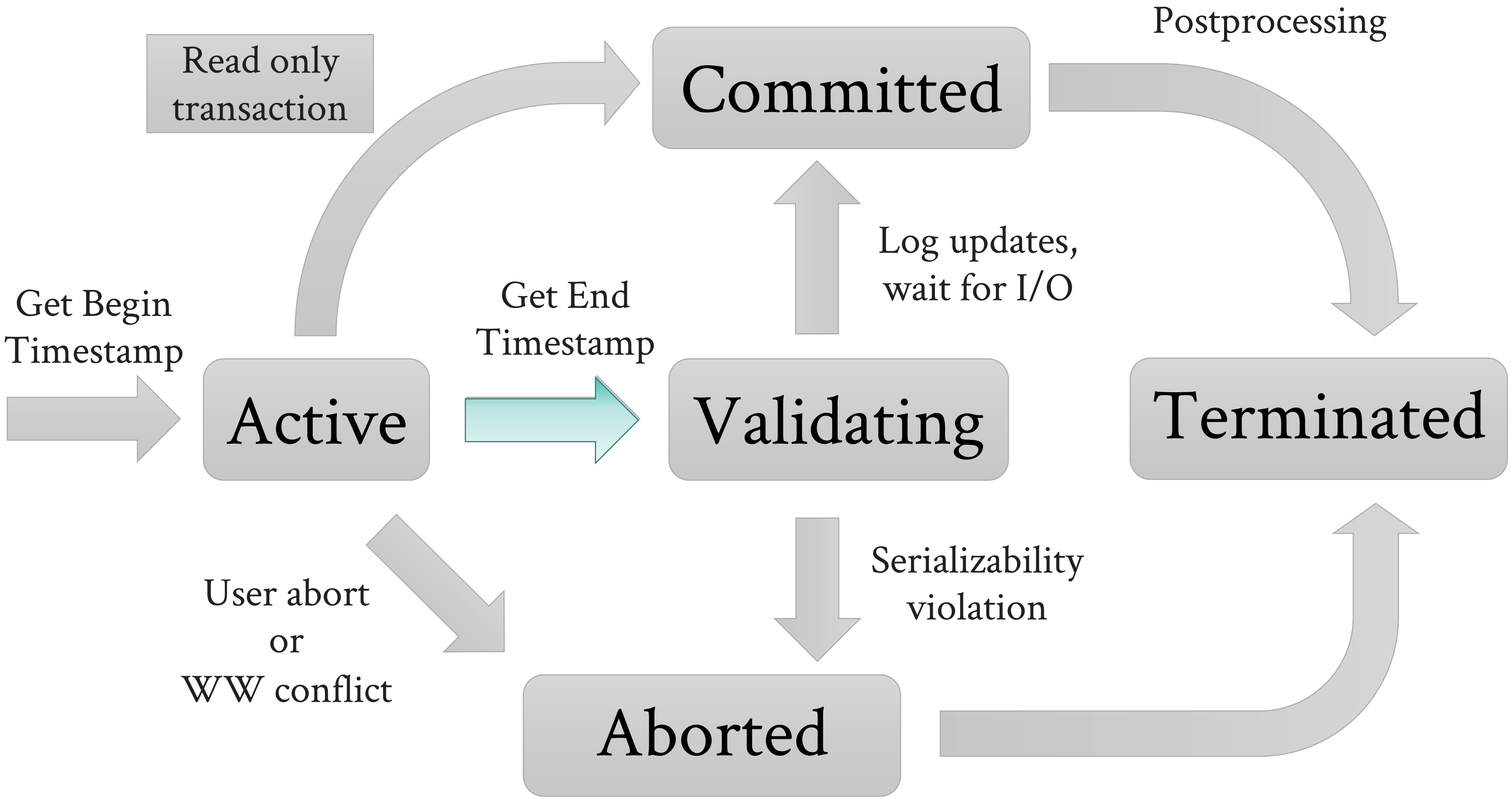
Commit Dependencies

- 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 the end of the txn.
  - Dependency wait time "added" to log latency.
- But: Cascading aborts are now possible.

COMMIT DEPENDENCIES



Get Begin Timestamp

Active

Read only transaction

Get End Timestamp

Validating

Committed

Log updates, wait for I/O

Postprocessing

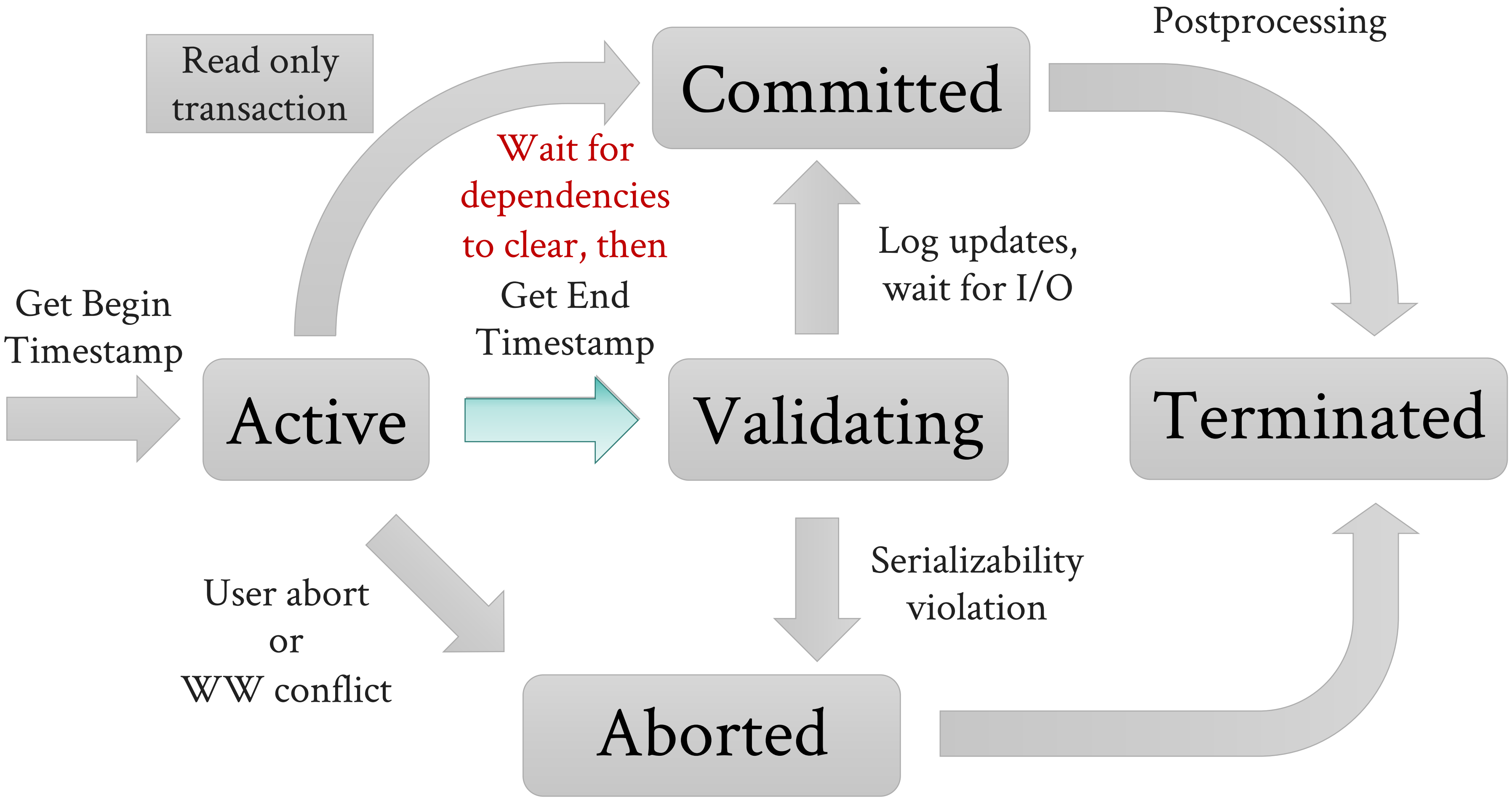
Terminated

User abort or WW conflict

Serializability violation

Aborted

COMMIT DEPENDENCIES



Read only transaction

Committed

Postprocessing

Wait for dependencies to clear, then

Get End Timestamp

Log updates, wait for I/O

Get Begin Timestamp

Active

Validating

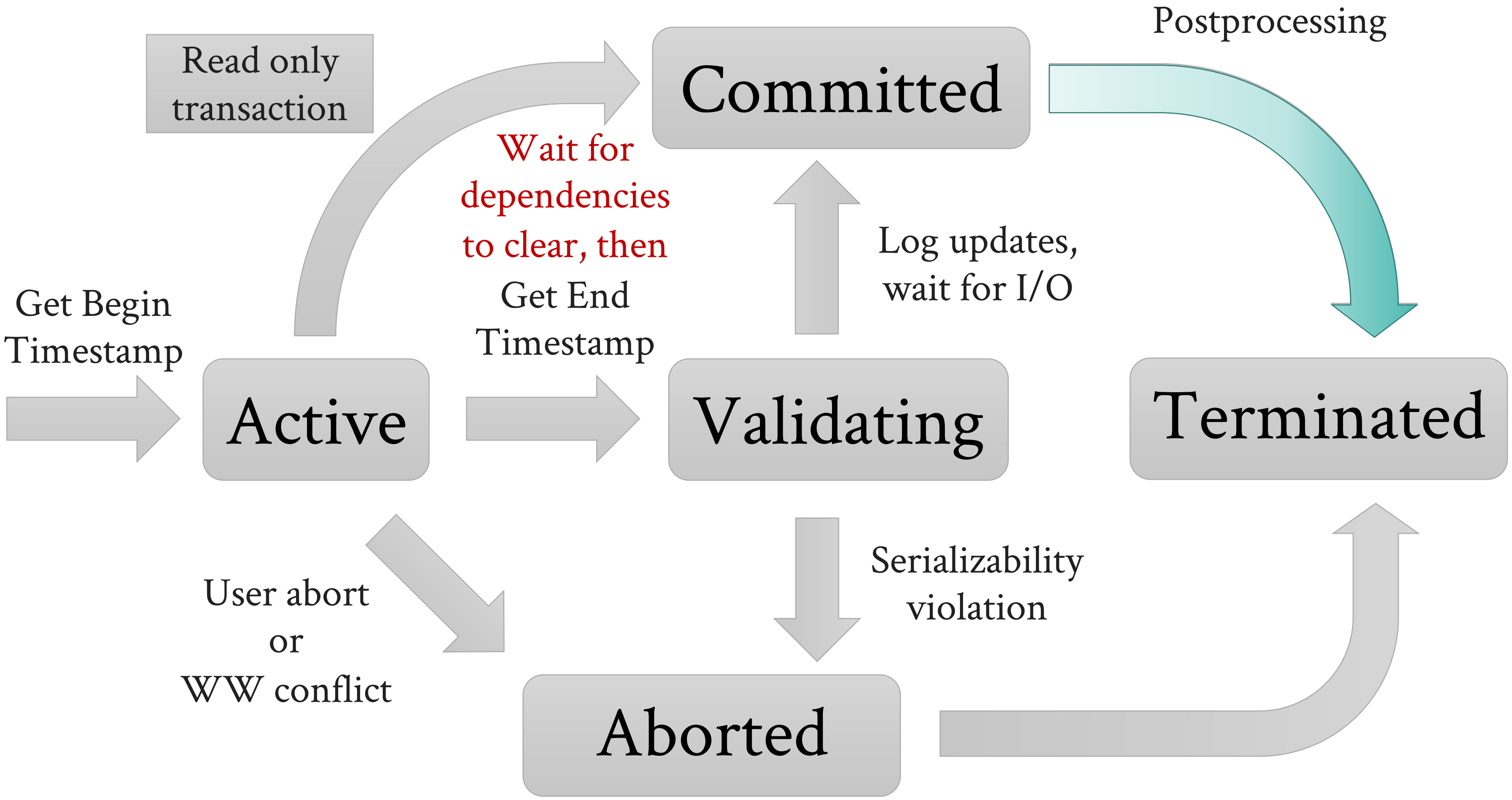
Terminated

User abort or WW conflict

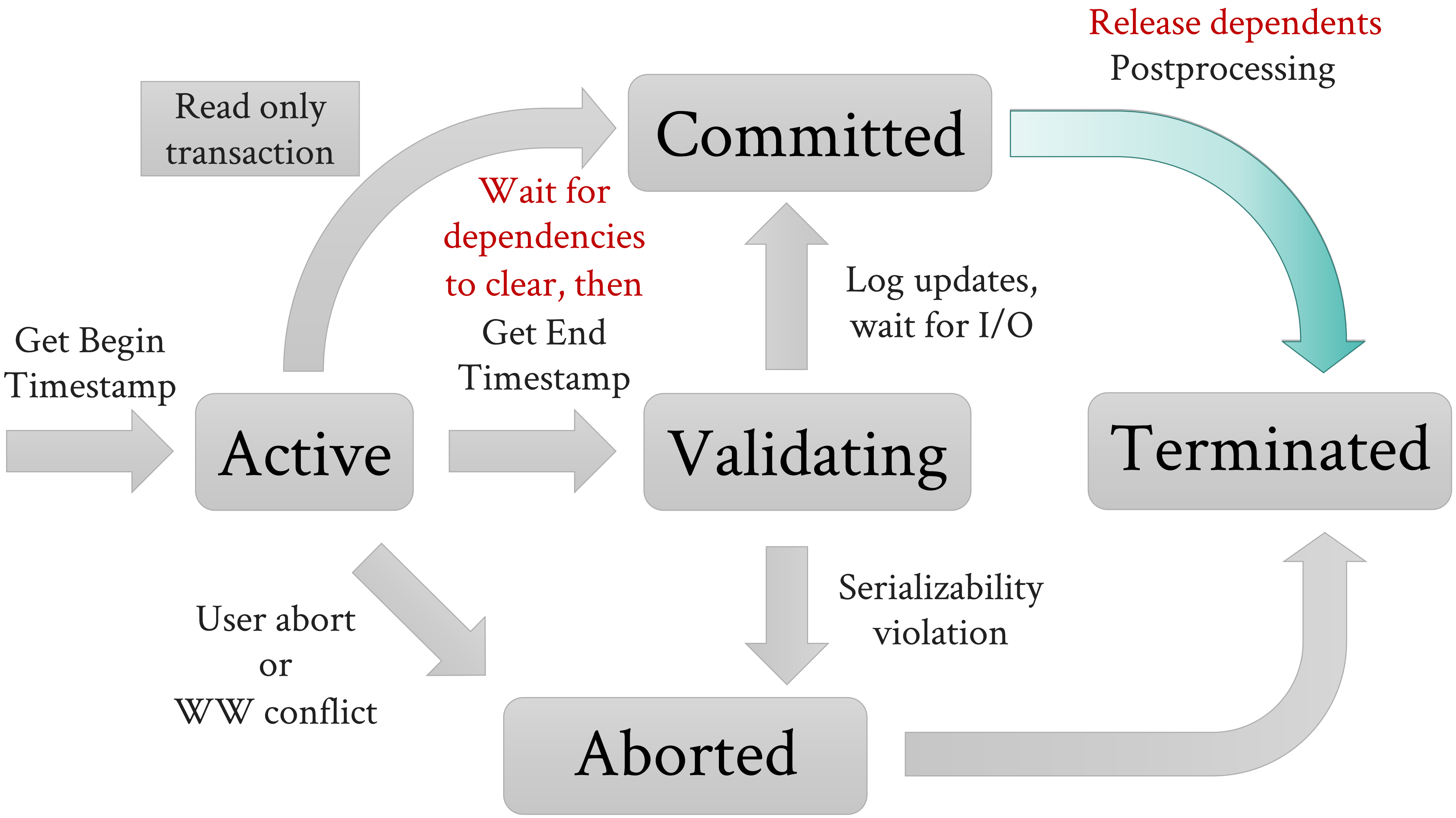
Serializability violation

Aborted

COMMIT DEPENDENCIES



Commit Dependencies



Evaluation

- 2-socket × 6-core Xeon X5650 with 48GB RAM.
- All transactions run under Serializable isolation.

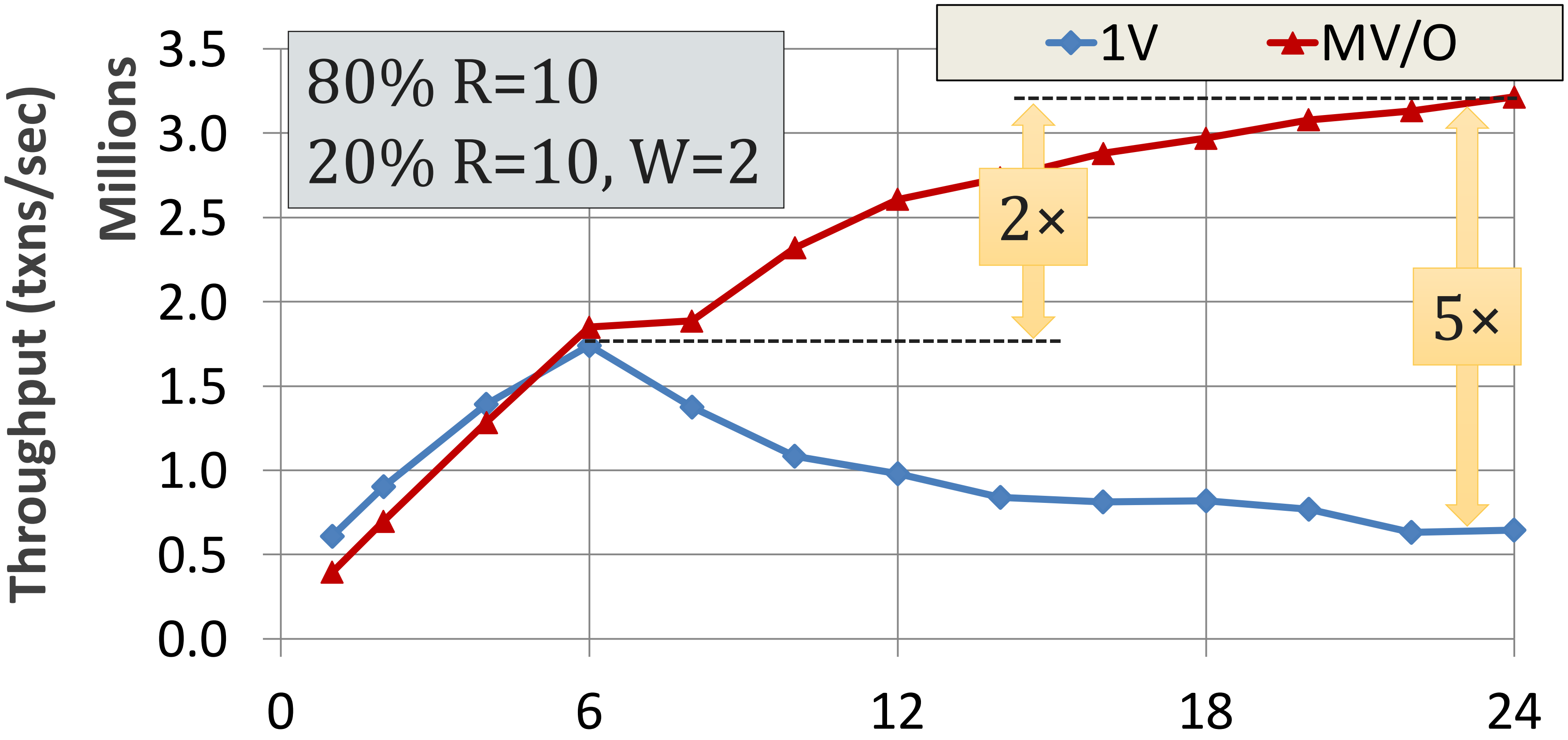
MV/O	Multi-version optimistic
1V	Single-version two-phase locking

Evaluation: TATP Benchmark

- Simulates a telecommunications application.
  - 4 tables, 7 different transactions, sized for 20M subscribers.
- Very short transactions: Less than 5 ops/txn on average.
- Very little contention.

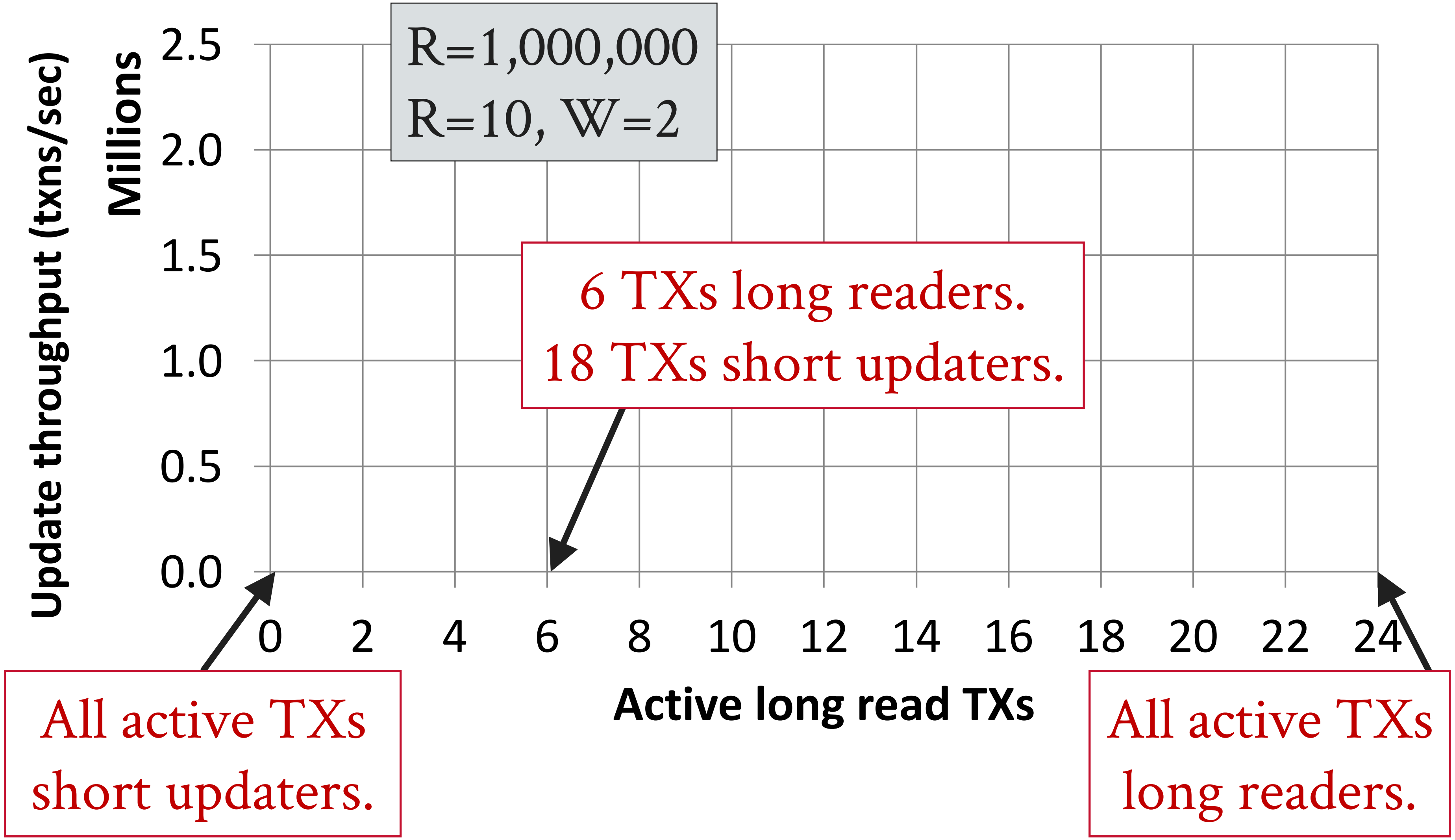
Scheme	Throughput (txn/sec)
MV/O	3,121,494
1V	4,220,119

SCALABILITY: EXTREME CONTENTION (1000 ROWS SYNTHETIC DATABASE)



MV/O does not break under contention.
MV/O does not need throttling for max perf.

EFFECT OF LONG READERS (10M ROW TABLE SYNTHETIC DATABASE)



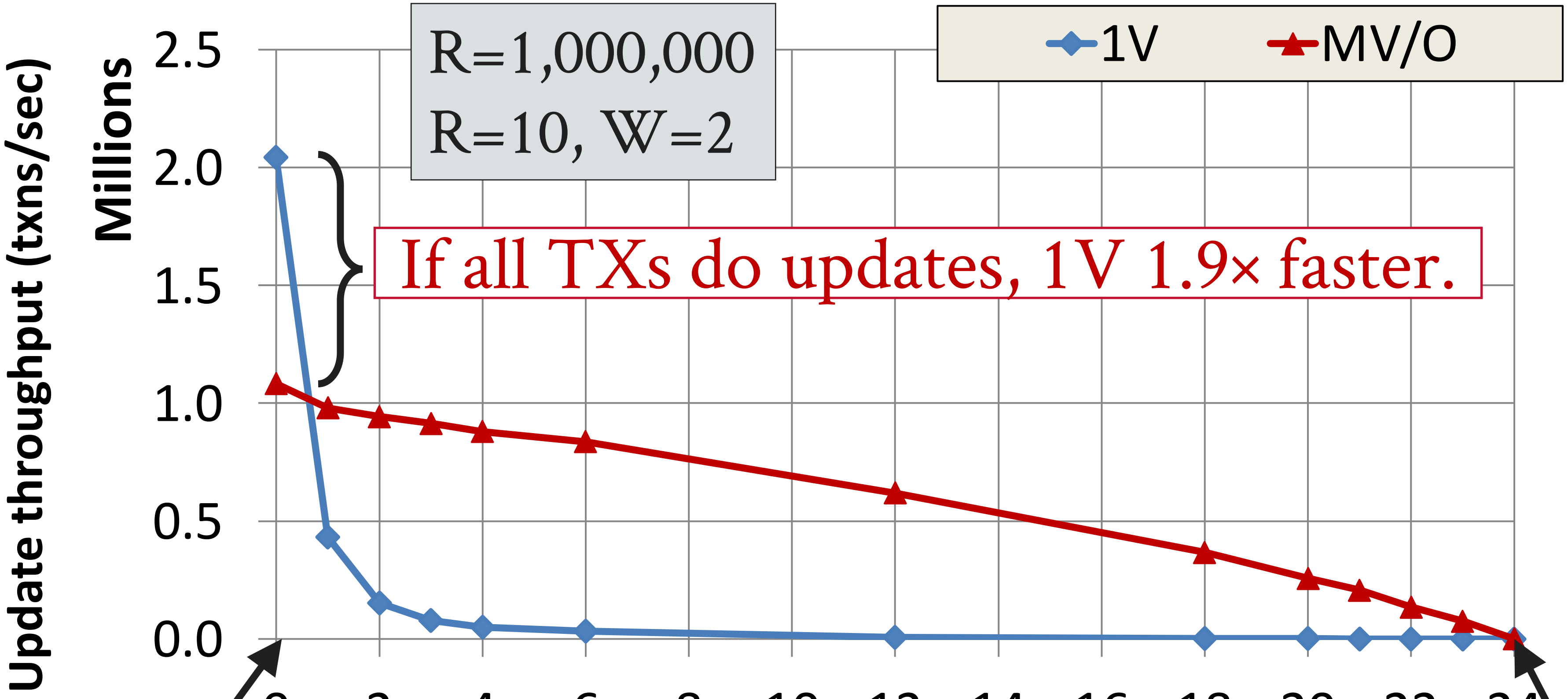
R=1,000,000
R=10, W=2

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 SYNTHETIC DATABASE)

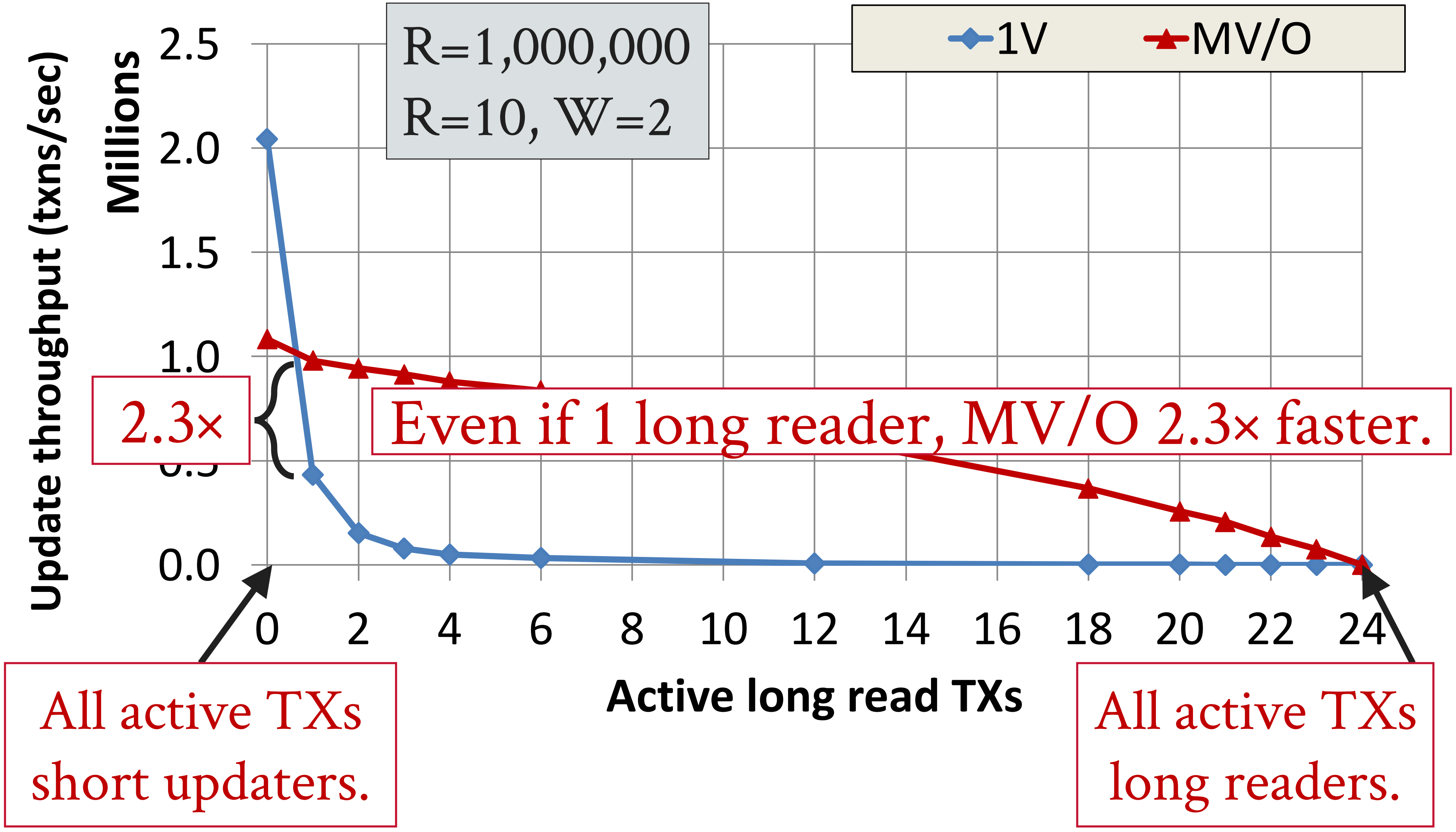


If all TXs do updates, 1V 1.9× faster.

All active TXs short updaters.

All active TXs long readers.

Effect of Long Readers (10M row table Synthetic database)



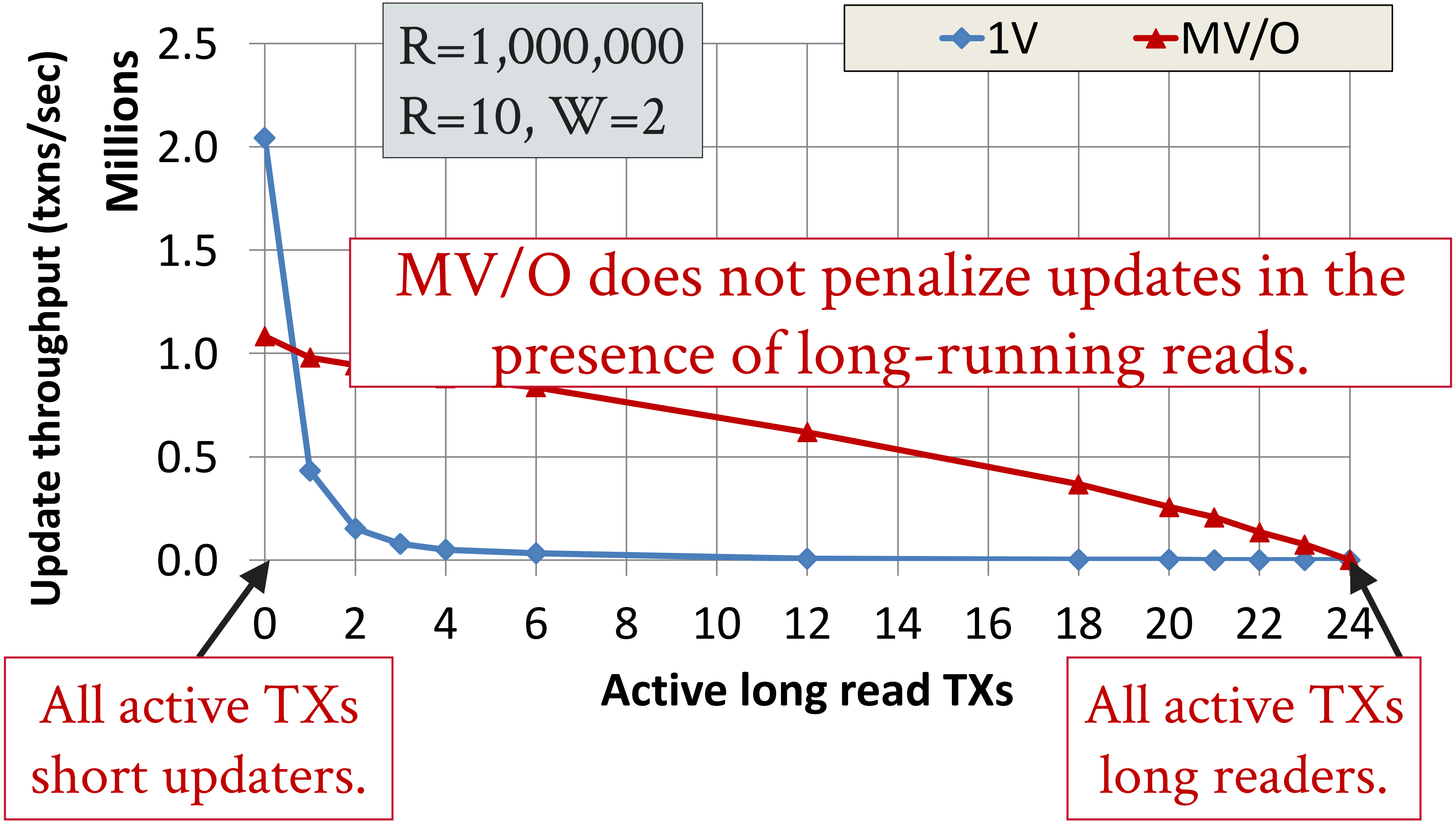
2.3×

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

All active TXs short updaters.

All active TXs long readers.

EFFECT OF LONG READERS (10M ROW TABLE SYNTHETIC DATABASE)



MV/O does not penalize updates in the presence of long-running reads.

All active TXs short updaters.

All active TXs long readers.

OTHER NOTES

- Other aspects like checkpointing and recovery still have be performed. Can optimize these for the in-memory case.
  - Create “data” files, and “delta” files.
  - Data files: inserts and updates covering a specific time range.
  - Delta files: which version in the data files have been deleted.
  - Rebuild indices from these files.
  - To reduce the size of these files, periodically merge the data files, and apply delta (sort of like the compaction in LSM trees).
- Garbage collection is now critical.
- Hekaton creates new version (the chains are oldest-to-newest). Can do the reverse too, and can be more efficient for accesses to the new values.

HTAP

- Huge interest in Hybrid OLTP + OLAP systems.
- Storage formats clash: OLTP wants a row-store, and OLAP wants a column-store.
  - Can support both storage formats in the same engine.
  - Can be further optimized so that the row-store part is in-memory (as we just saw in Hekaton).
- Often a notion of "delta" is used, where the changed/uncommitted values are stored.
  - We saw these in the version chains in Hekaton.
- The re-scan cost in the MVCC can be expensive. A clever ideas it to use "Precison Locks" (see the Hyper paper)
  - Remember the predicate in the WHERE clause of the SQL query.
  - Run that predicate against the deltas (new versions) of records created by transactions that committed after the current txn started.
  - This delta set is much smaller, so the rescan can be significantly faster.

Alfons Kemper, Thomas Neumann: HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. ICDE 2011

SUMMARY AND OUTLOOK

- Multi-version schemes are necessary for high OLTP performance.
  - Readers don't block writers.
- MV schemes + OCC is a nice combination for in-memory OLTP.
  - No waiting on locks, and latch-free data structures.
  - Also can use codegen.
  - Want a low instruction count / txn for high performance.
- Orthogonally need a disaster recovery method.
- OLTP on clusters bring new challenges. Need to run a commit protocol like 2PC. Need to have a replication method like RAFT.
- HTAP systems need to find a way to do both row and column store in the same engine.
- Building OLTP systems in a disaggregated cloud ecosystem bring additional challenges, including rethinking the storage layer.