15-721 DATABASE SYSTEMS

Lecture #03 – Concurrency Control Part I

Andy Pavlo // Carnegie Mellon University // Spring 2016

TODAY'S AGENDA

Transaction Models
Concurrency Control Overview
Many-Core Evaluation

TRANSACTION DEFINITION

TRANSACTION DEFINITION

A txn is a sequence of actions that are executed on a shared database to perform some higher-level function.

Txns are the basic unit of change in the DBMS. No partial txns are allowed.

ACTION CLASSIFICATION

Unprotected Actions

→ These lack all of the ACID properties except for consistency. Their effects cannot be depended upon.

Protected Actions

→ These do not externalize their results before they are completely done. Fully ACID.

Real Actions

→ These affect the physical world in a way that is hard or impossible to reverse.

TRANSACTION MODELS

Flat Txns

Flat Txns + Savepoints

Chained Txns

Nested Txns

Saga Txns

Compensating Txns

FLAT TRANSACTIONS

Standard txn model that starts with **BEGIN**, followed by one or more actions, and then completed with either **COMMIT** or **ROLLBACK**.

Txn #1	*Txn #2*
BEGIN	BEGIN
READ(A)	READ(A)
WRITE(B)	WRITE(B)
COMMIT	ROLLBACK

LIMITATIONS OF FLAT TRANSACTIONS

The application can only rollback the entire txn (i.e., no partial rollbacks).

All of a txn's work is lost is the DBMS fails before that txn finishes.

Each txn takes place at a single point in time

LIMITATIONS OF FLAT TRANSACTIONS

Multi-Stage Planning

→ An application needs to make multiple reservations.
→ All the reservations need to occur or none of them.

Bulk Updates

→ An application needs to update one billion records.
→ This txn could take hours to complete and therefore the DBMS is exposed to losing all of its work for any failure or conflict.

TRANSACTION SAVEPOINTS

Save the current state of processing for the txn and provide a handle for the application to refer to that savepoint.

The application can control the state of the txn through these checkpoints:

→ **ROLLBACK** – Revert all changes back to the state of the DB at the savepoint.

→ **RELEASE** – Destroys a savepoint previously defined in the txn.

TRANSACTION SAVEPOINTS

Txn #1

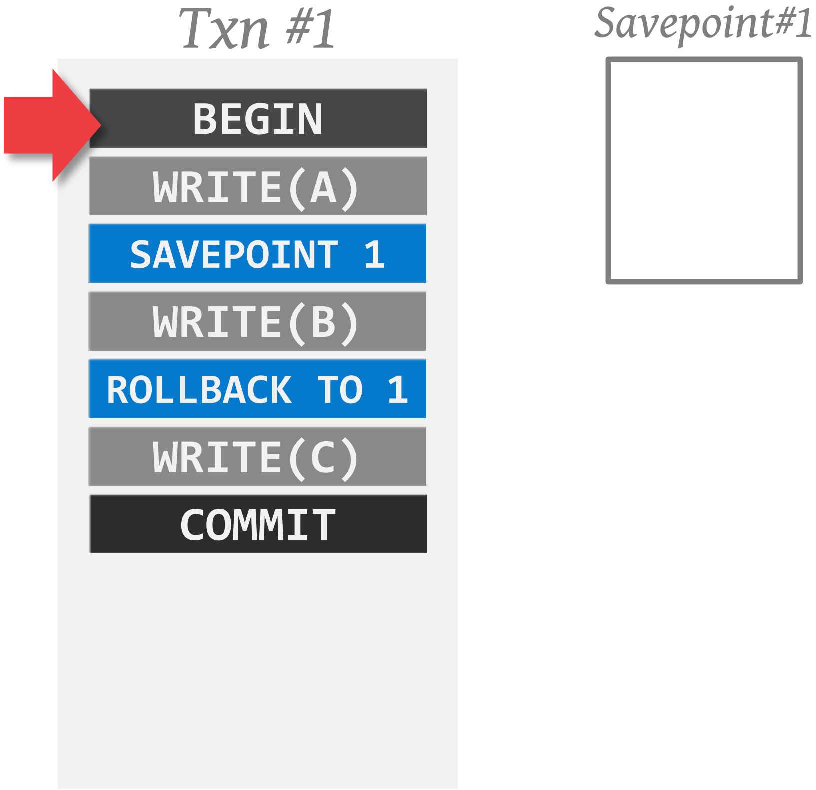
```
BEGIN
WRITE(A)
SAVEPOINT 1
WRITE(B)
ROLLBACK TO 1
WRITE(C)
COMMIT
```

TRANSACTION SAVEPOINTS

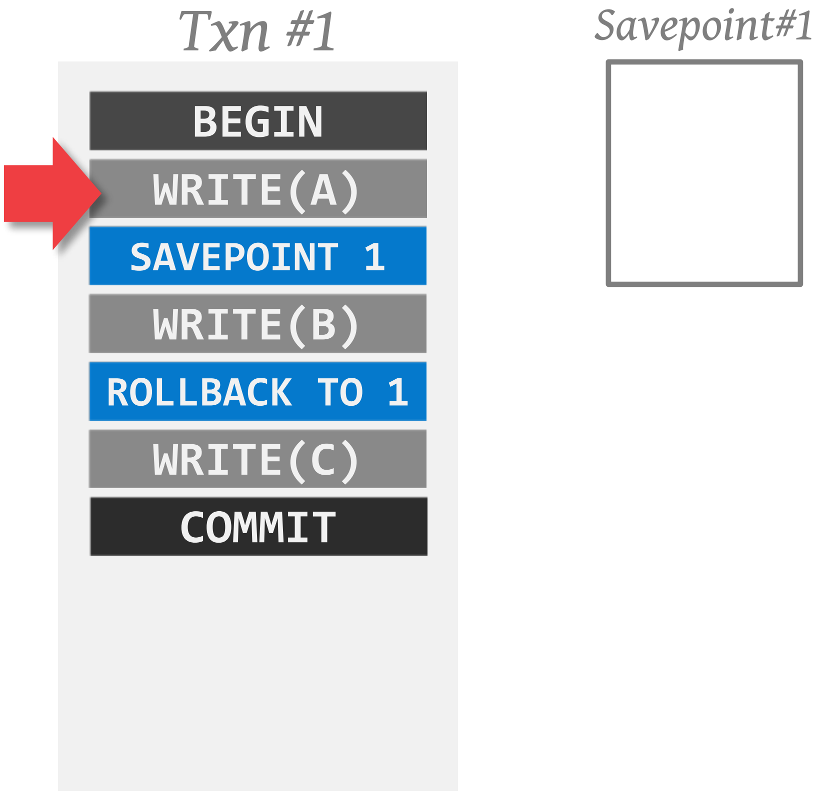
Txn #1

```
BEGIN
WRITE(A)
SAVEPOINT 1
WRITE(B)
ROLLBACK TO 1
WRITE(C)
COMMIT
```

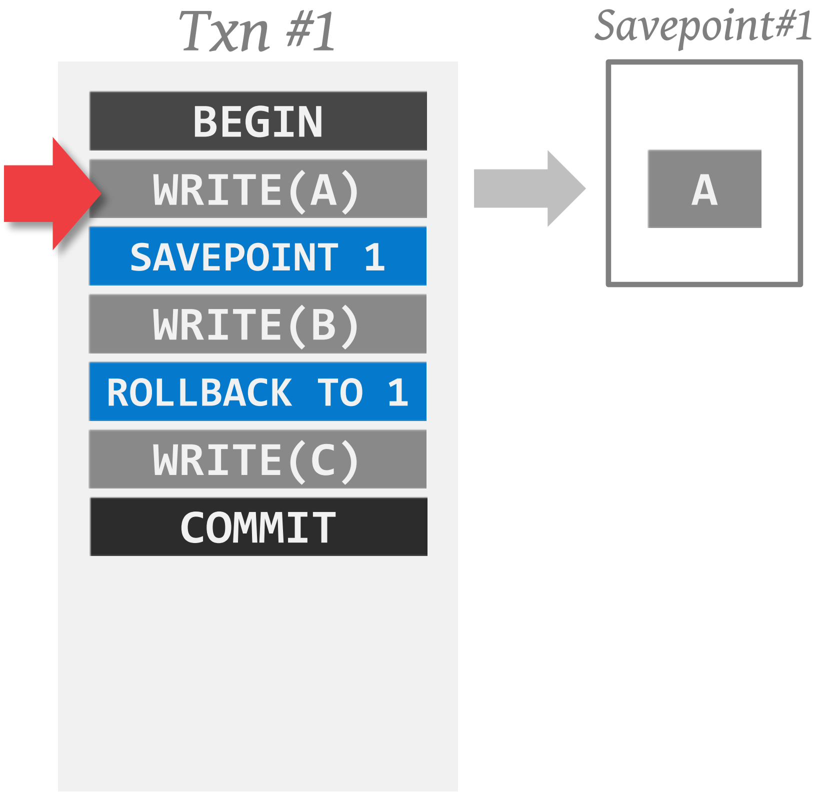
TRANSACTION SAVEPOINTS



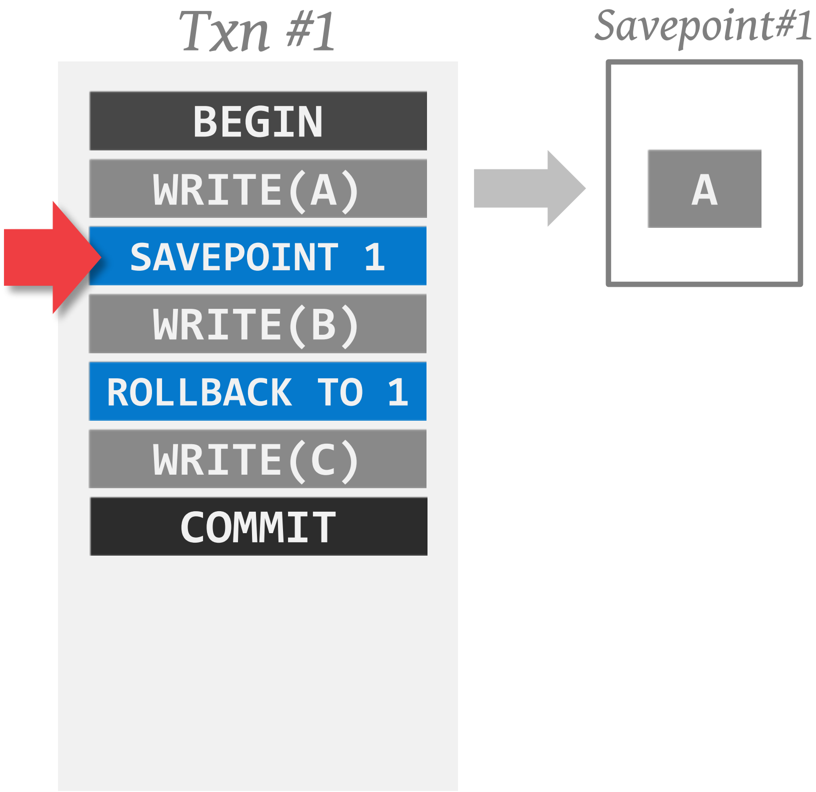
TRANSACTION SAVEPOINTS



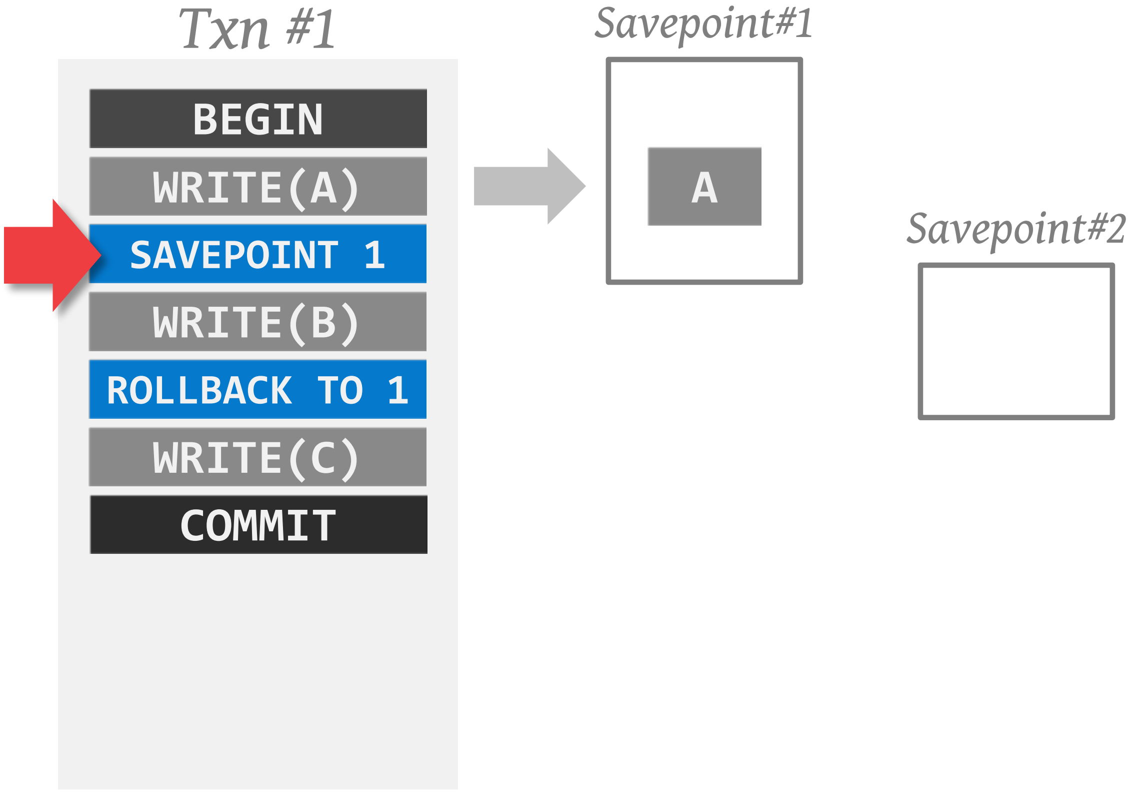
TRANSACTION SAVEPOINTS



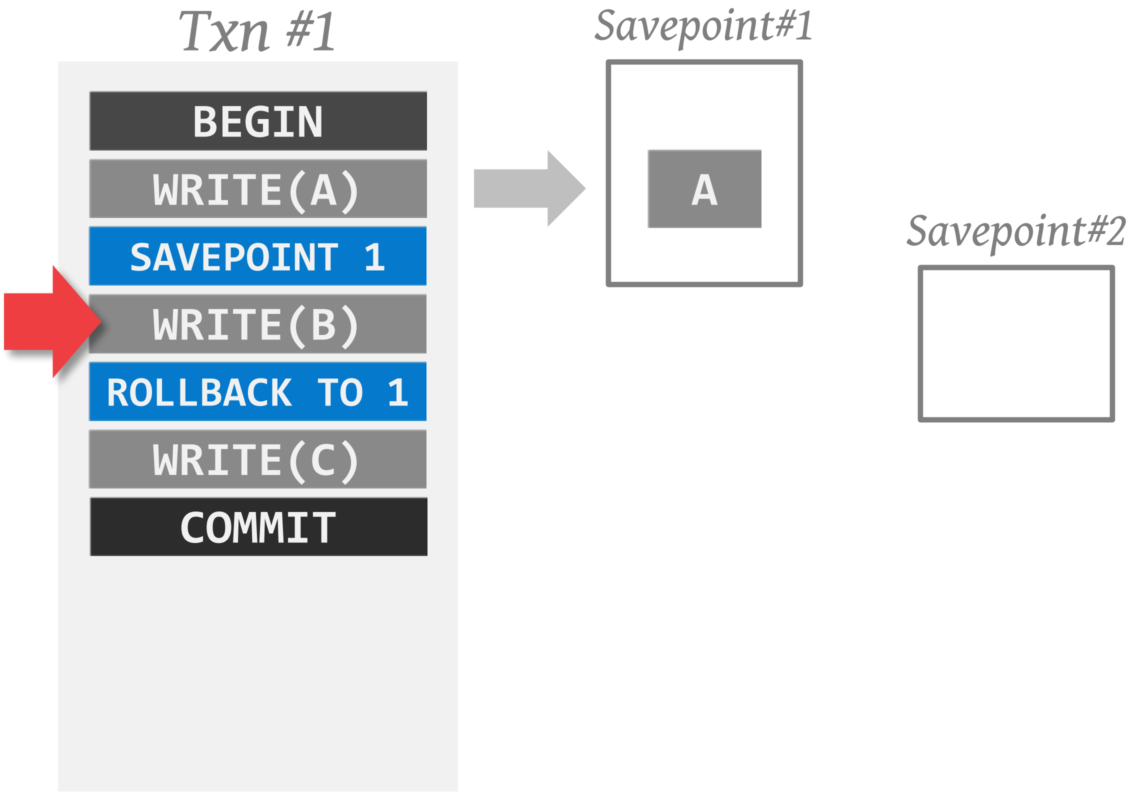
TRANSACTION SAVEPOINTS



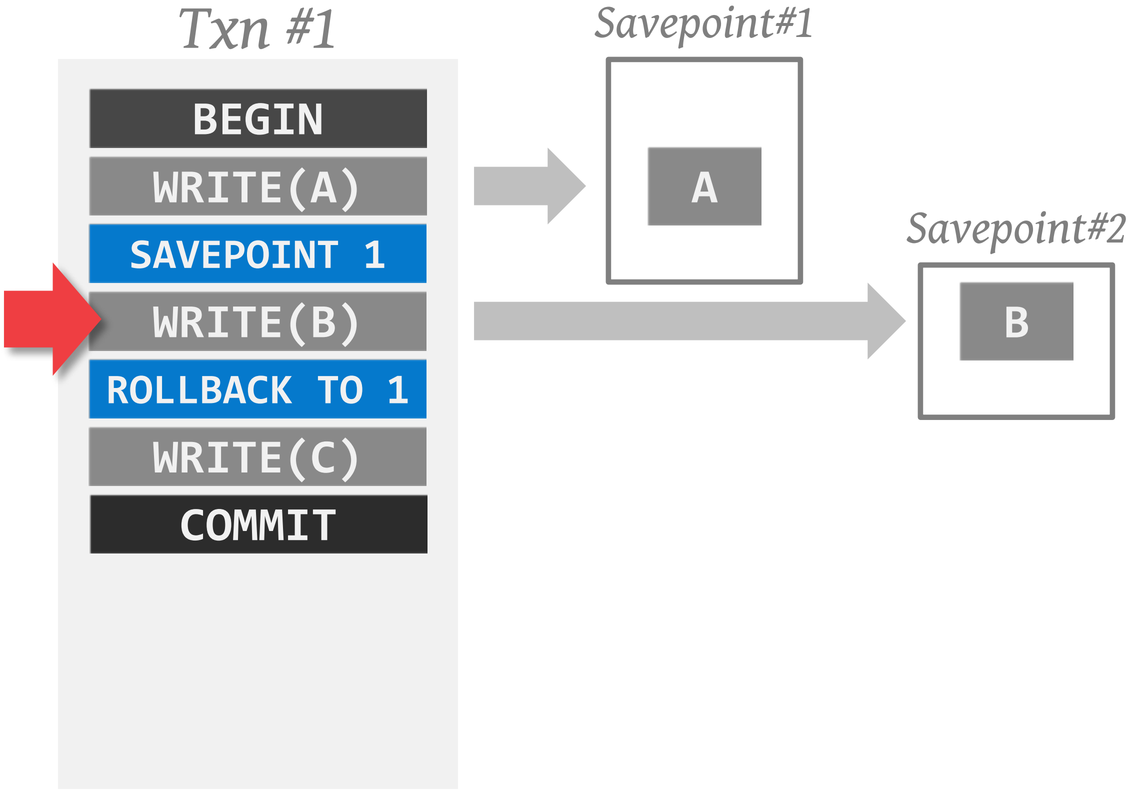
TRANSACTION SAVEPOINTS



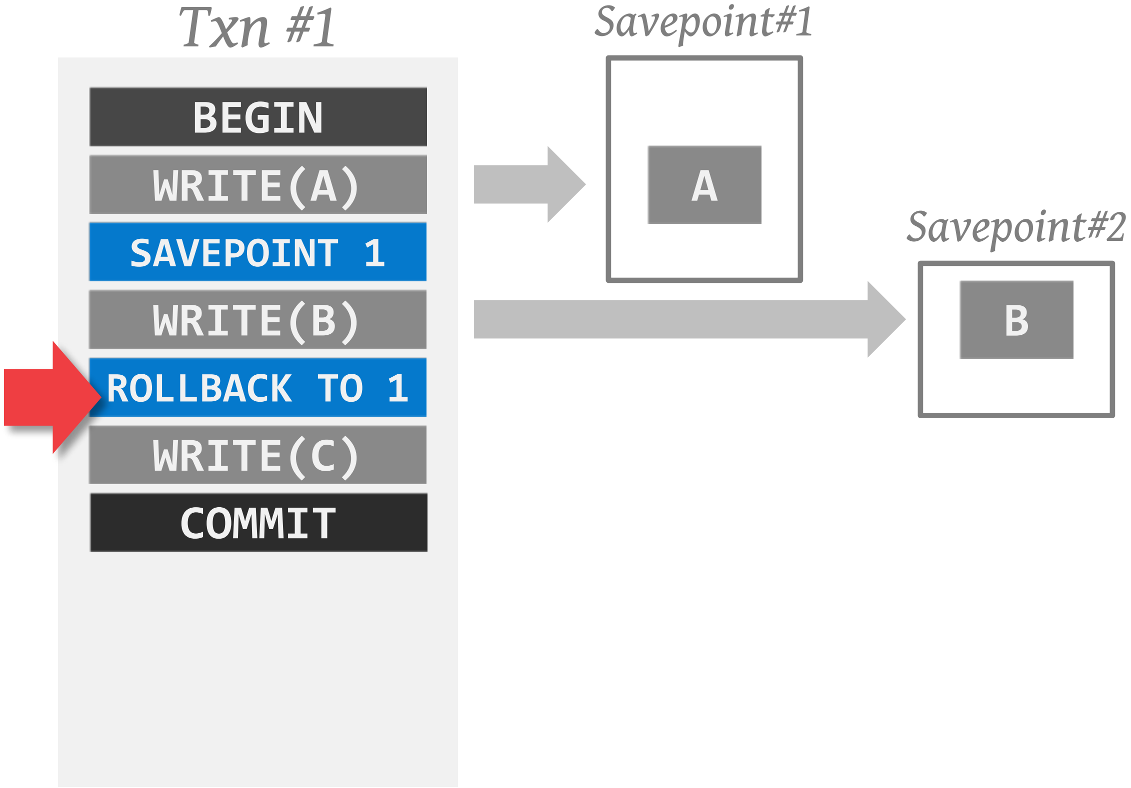
TRANSACTION SAVEPOINTS



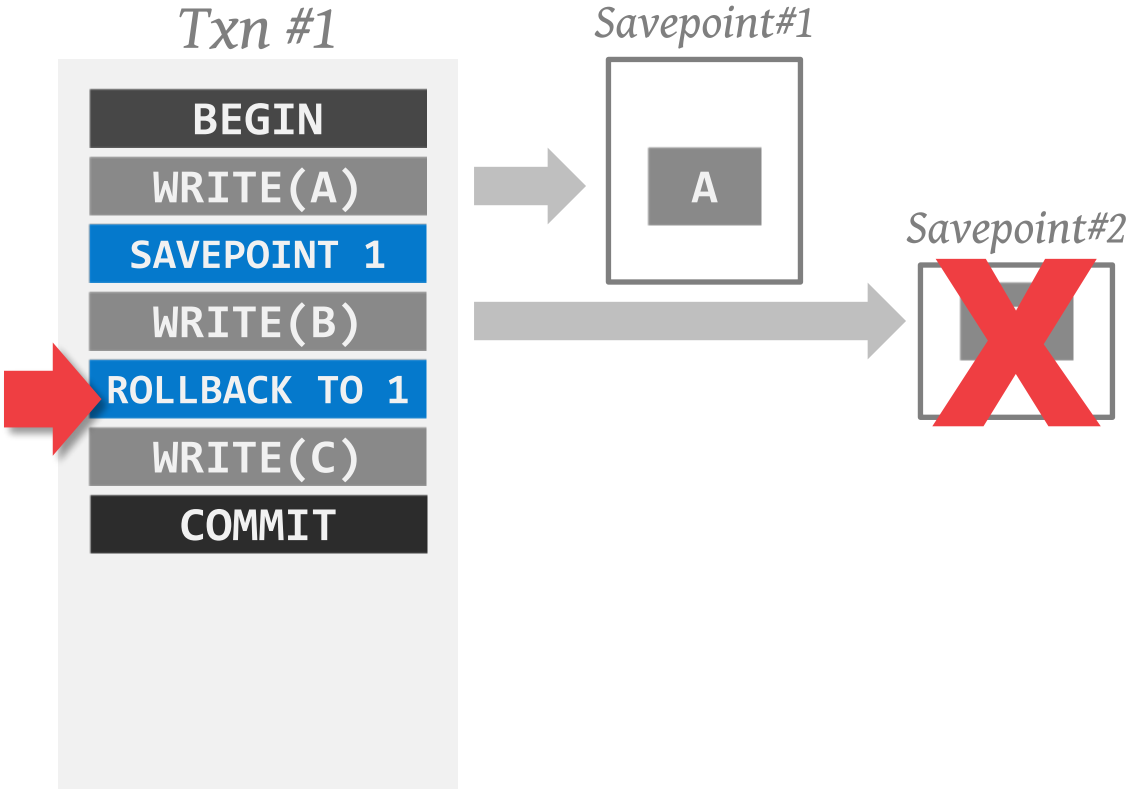
TRANSACTION SAVEPOINTS



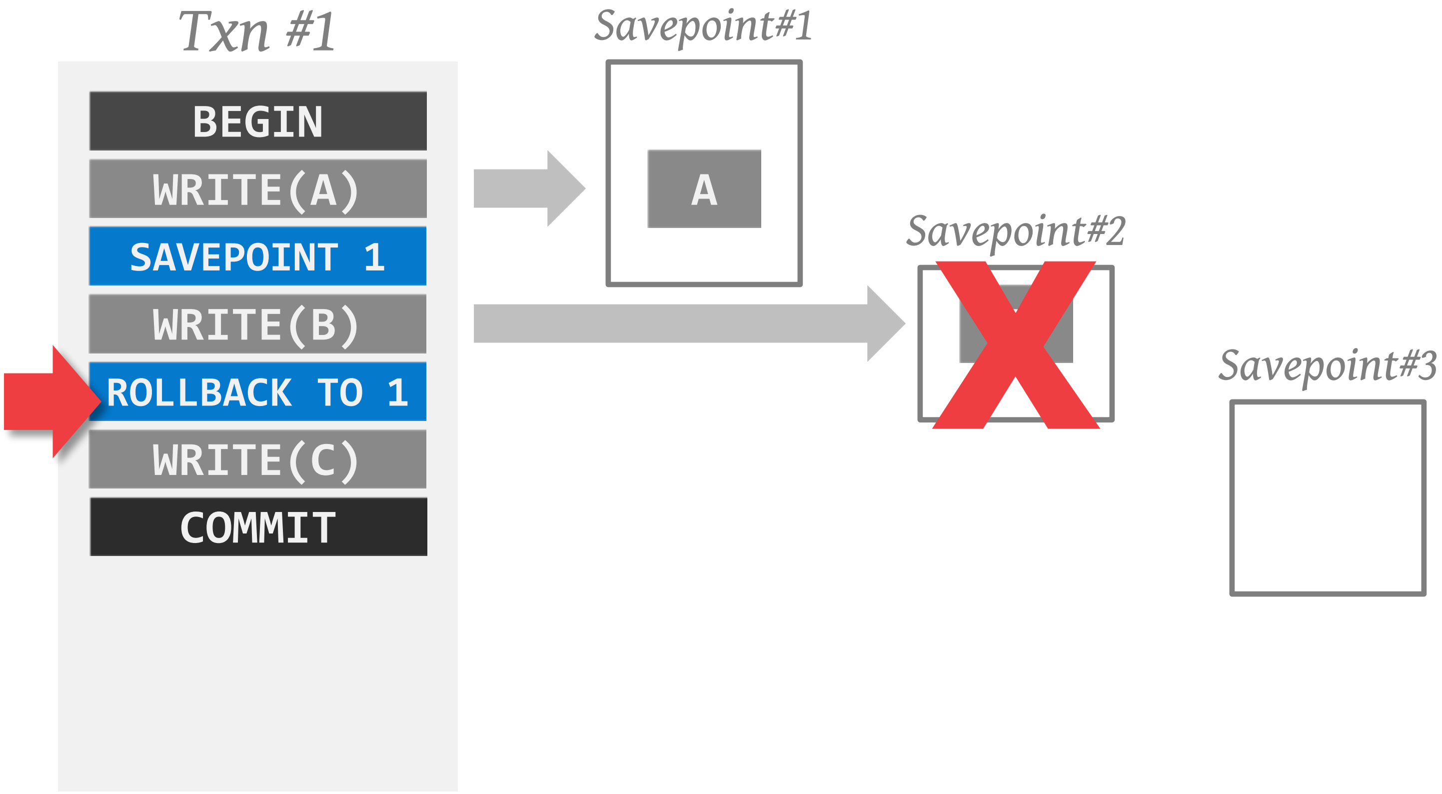
TRANSACTION SAVEPOINTS



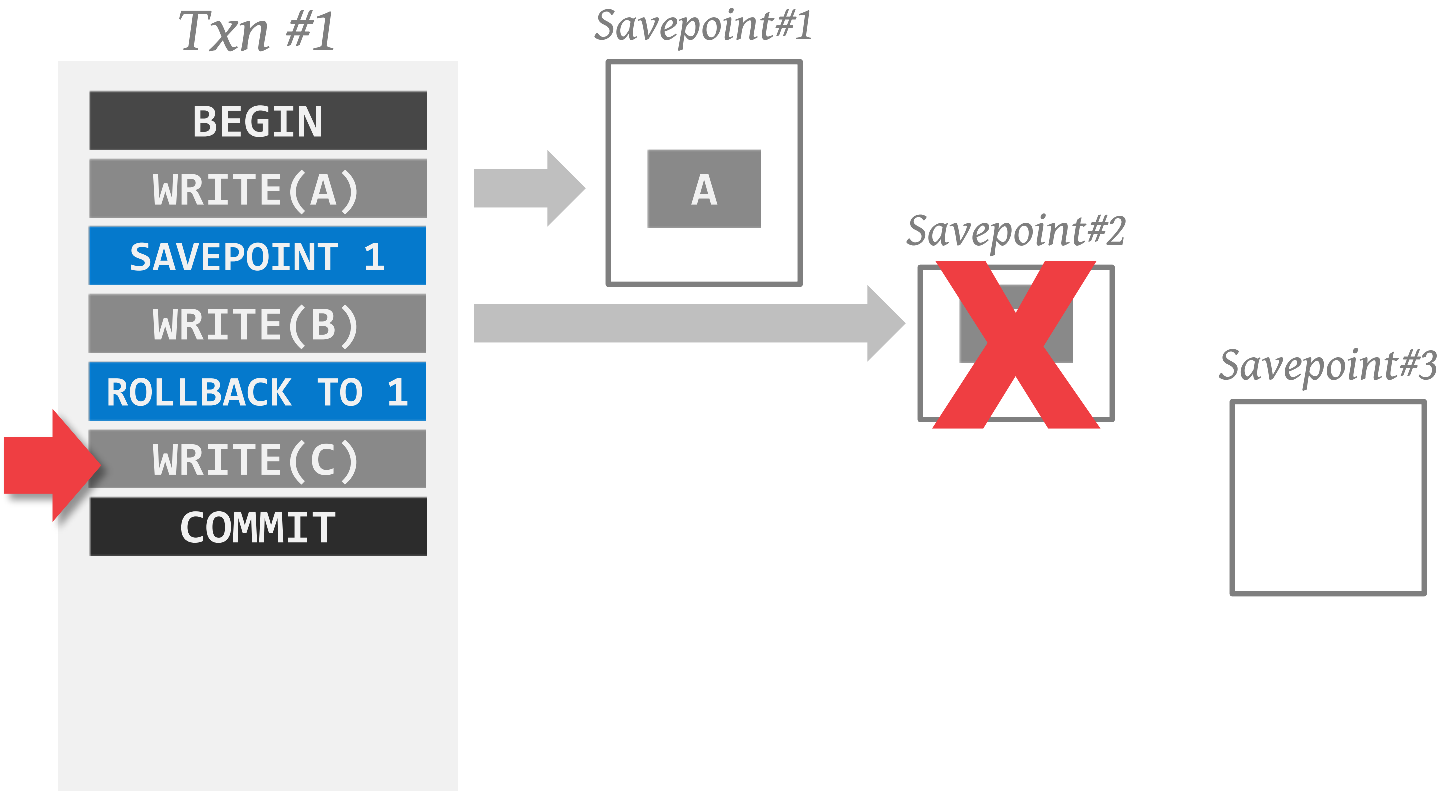
TRANSACTION SAVEPOINTS



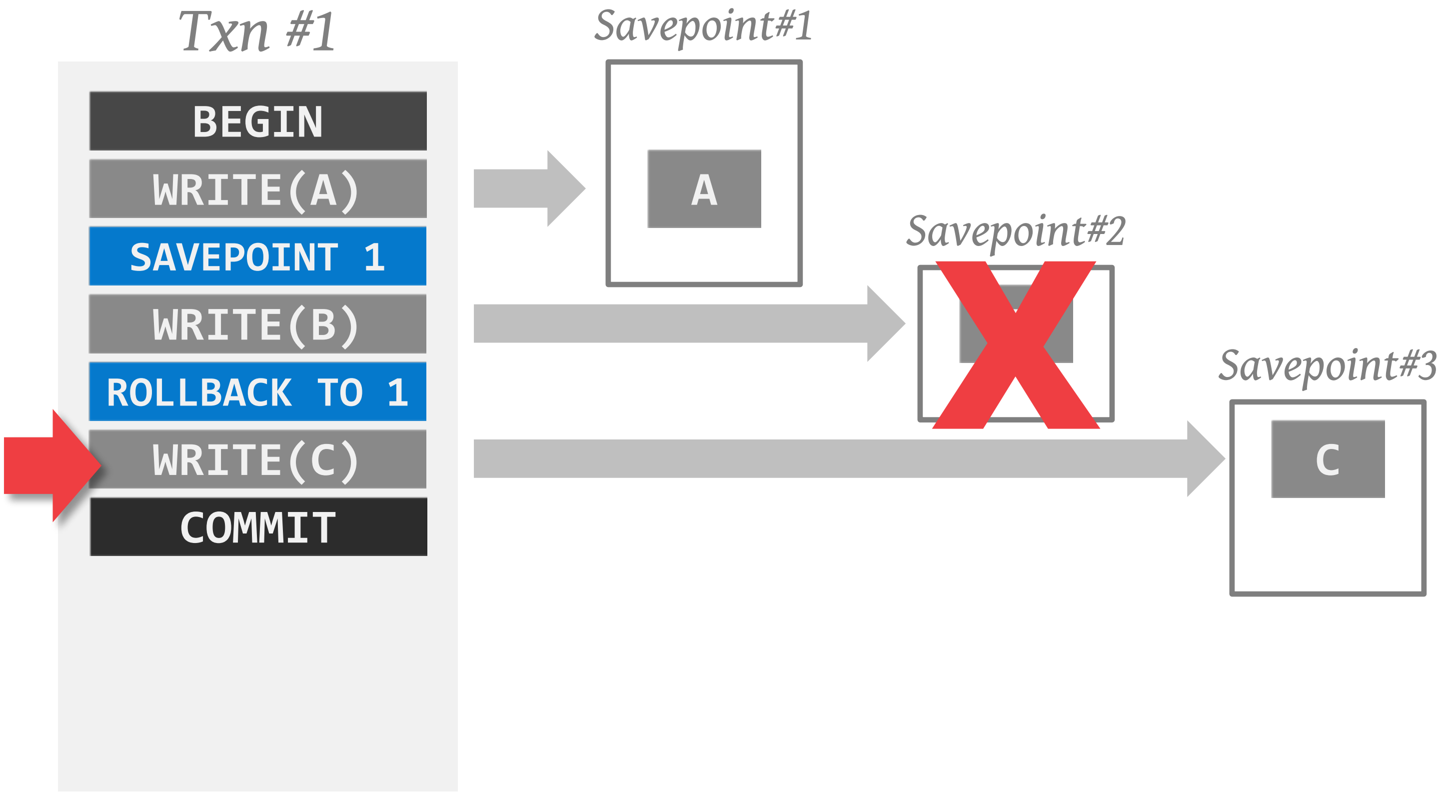
TRANSACTION SAVEPOINTS



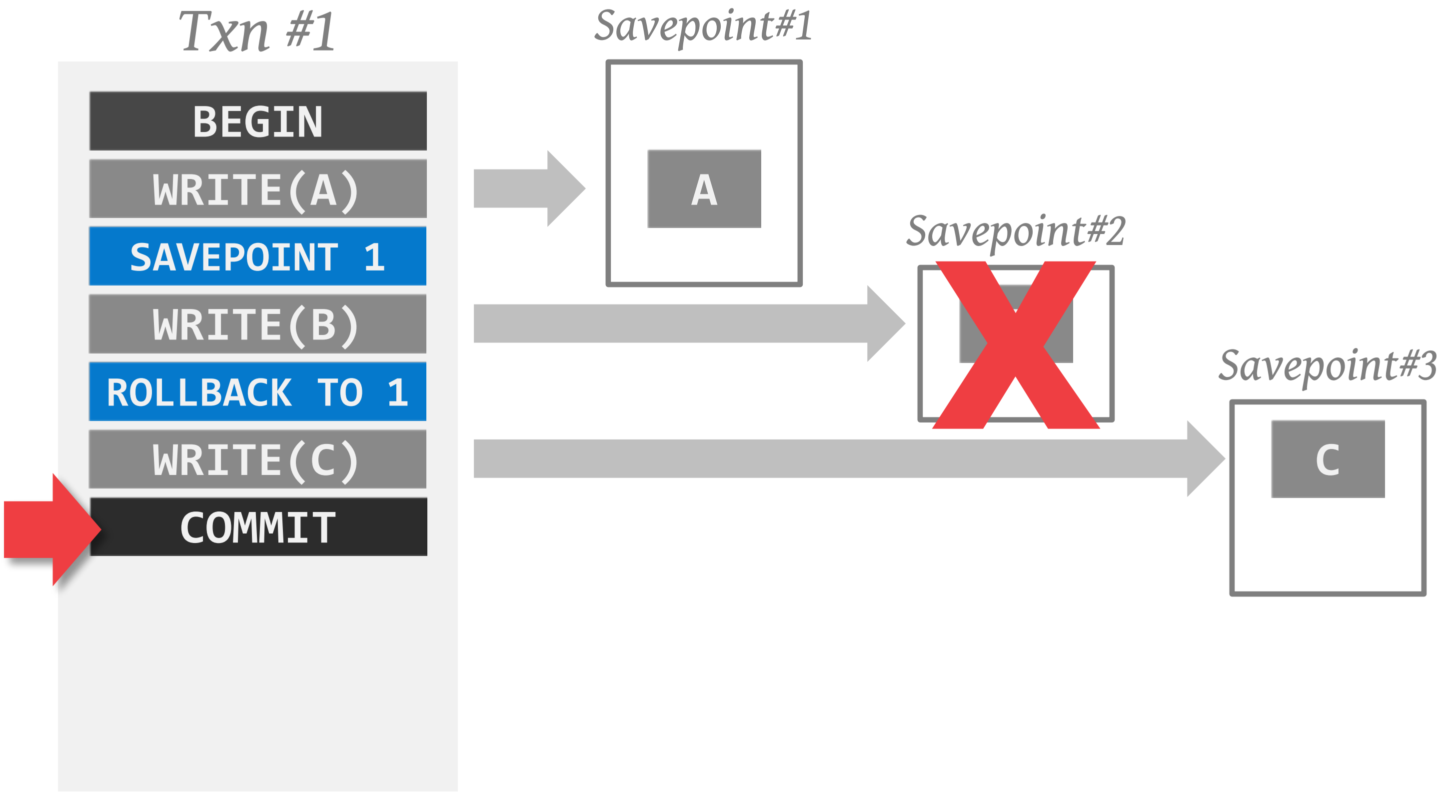
TRANSACTION SAVEPOINTS



TRANSACTION SAVEPOINTS



TRANSACTION SAVEPOINTS



TRANSACTION SAVEPOINTS

Txn #1

```
BEGIN
WRITE(A)
SAVEPOINT 1
WRITE(B)
SAVEPOINT 2
WRITE(C)
SAVEPOINT 3
RELEASE 2
WRITE(D)
ROLLBACK TO 3
```

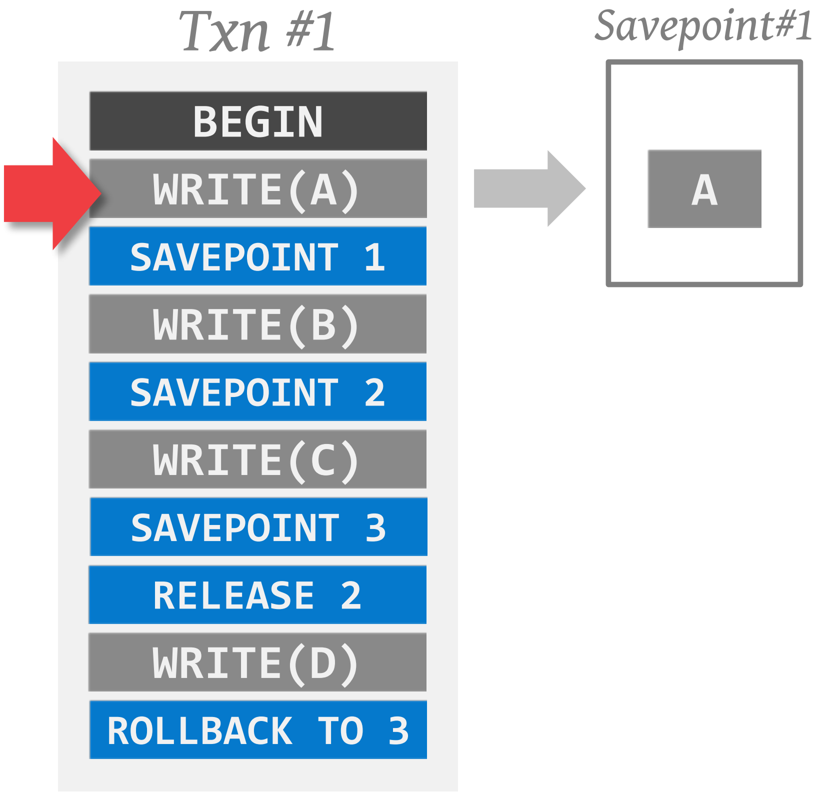
TRANSACTION SAVEPOINTS

Txn #1

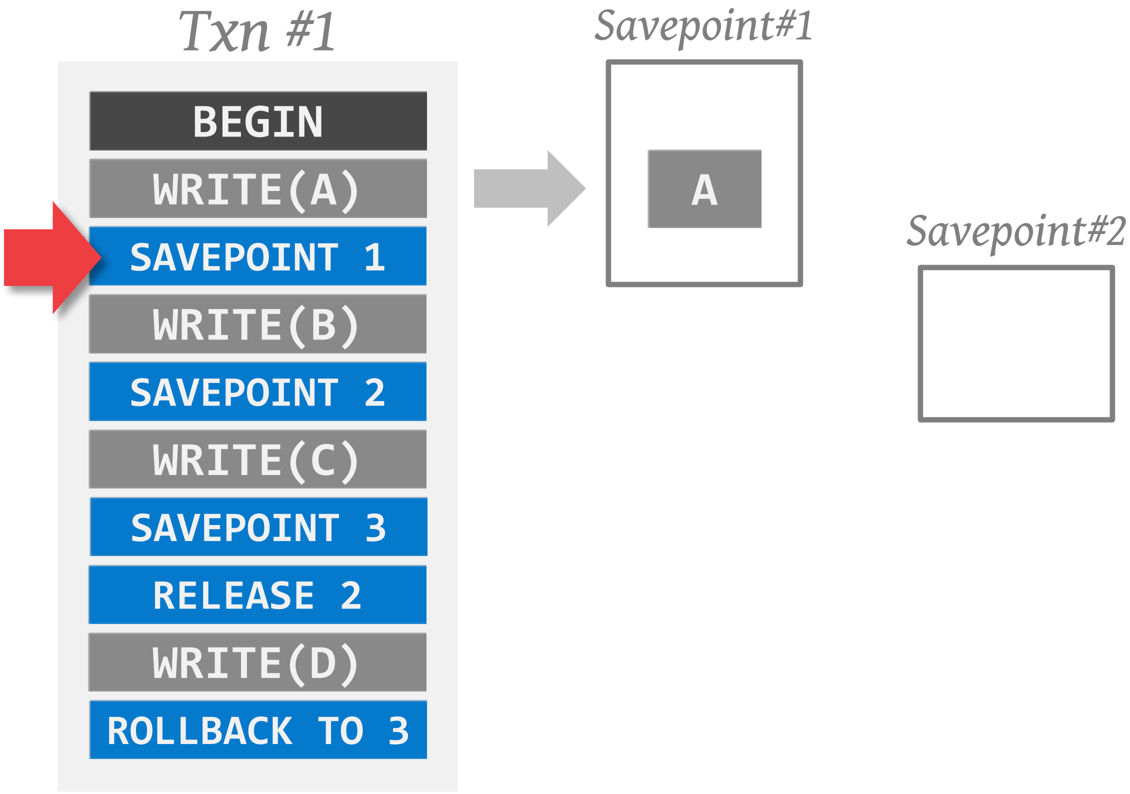
- BEGIN
- WRITE(A)
- SAVEPOINT 1
- WRITE(B)
- SAVEPOINT 2
- WRITE(C)
- SAVEPOINT 3
- RELEASE 2
- WRITE(D)
- ROLLBACK TO 3

Savepoint#1

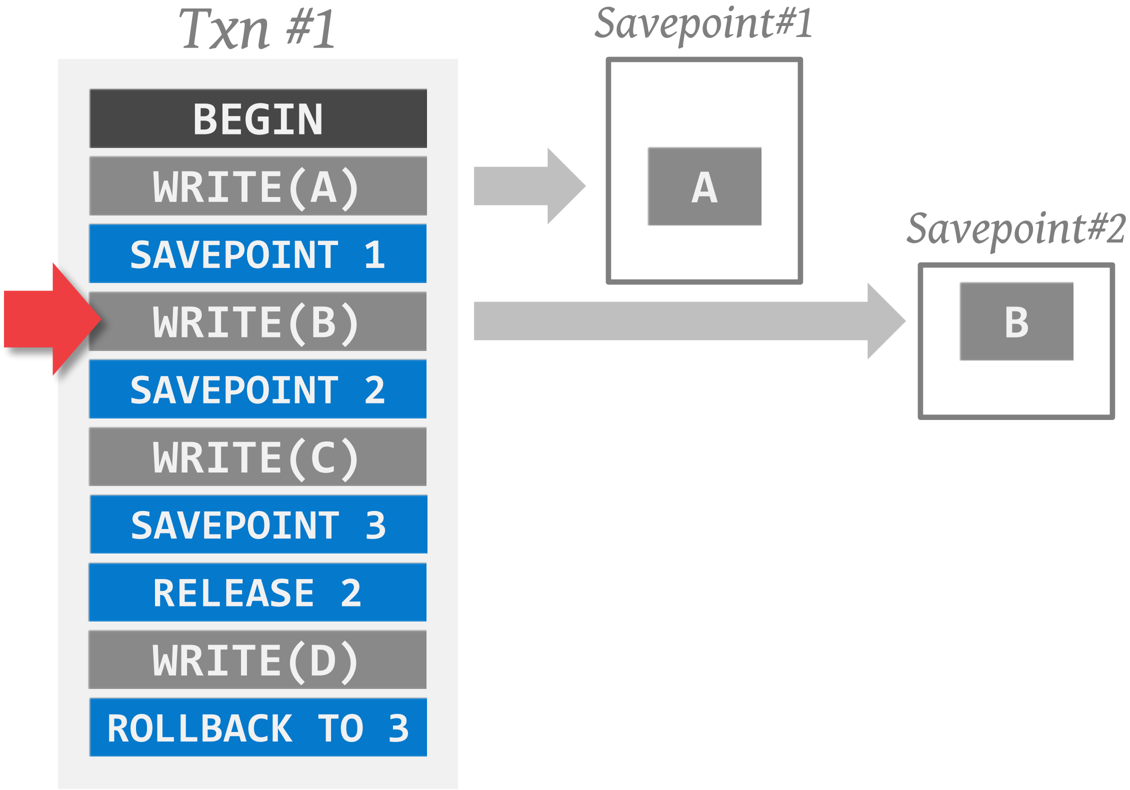
TRANSACTION SAVEPOINTS



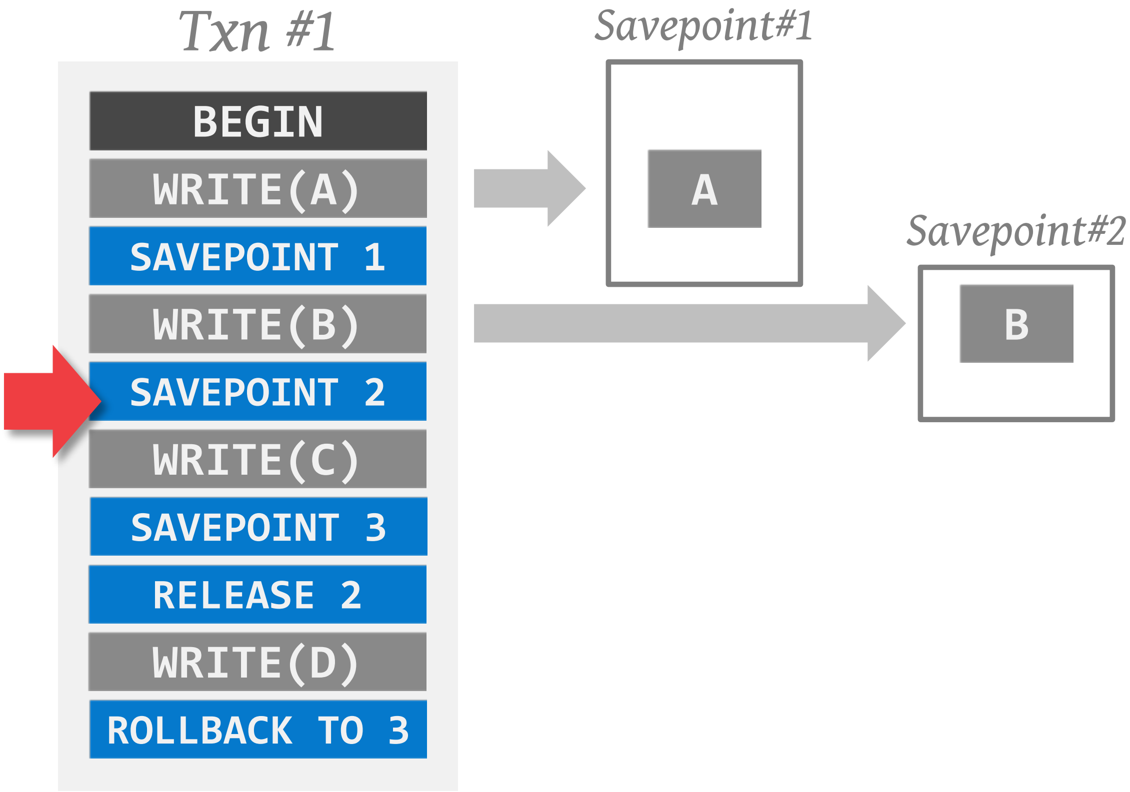
TRANSACTION SAVEPOINTS



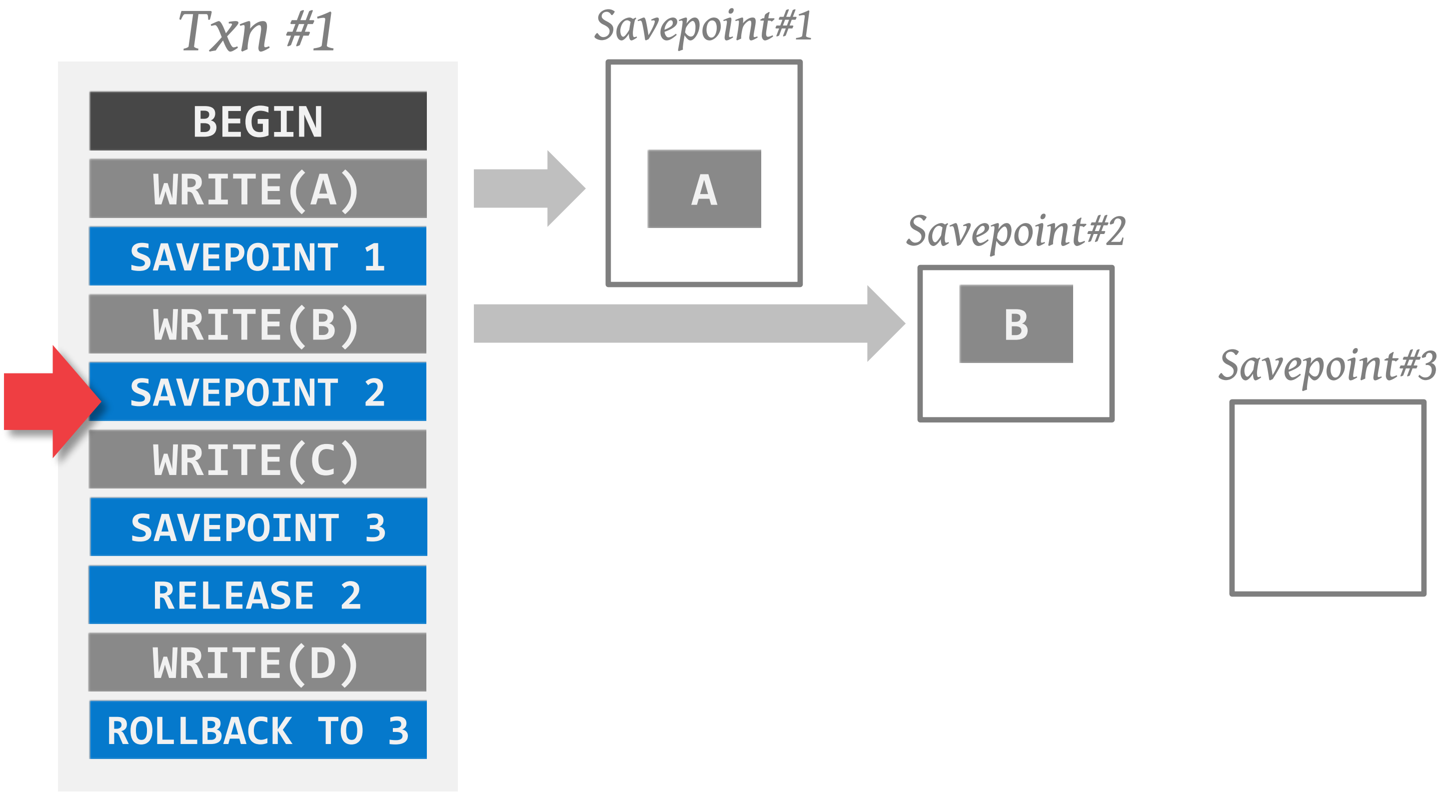
TRANSACTION SAVEPOINTS



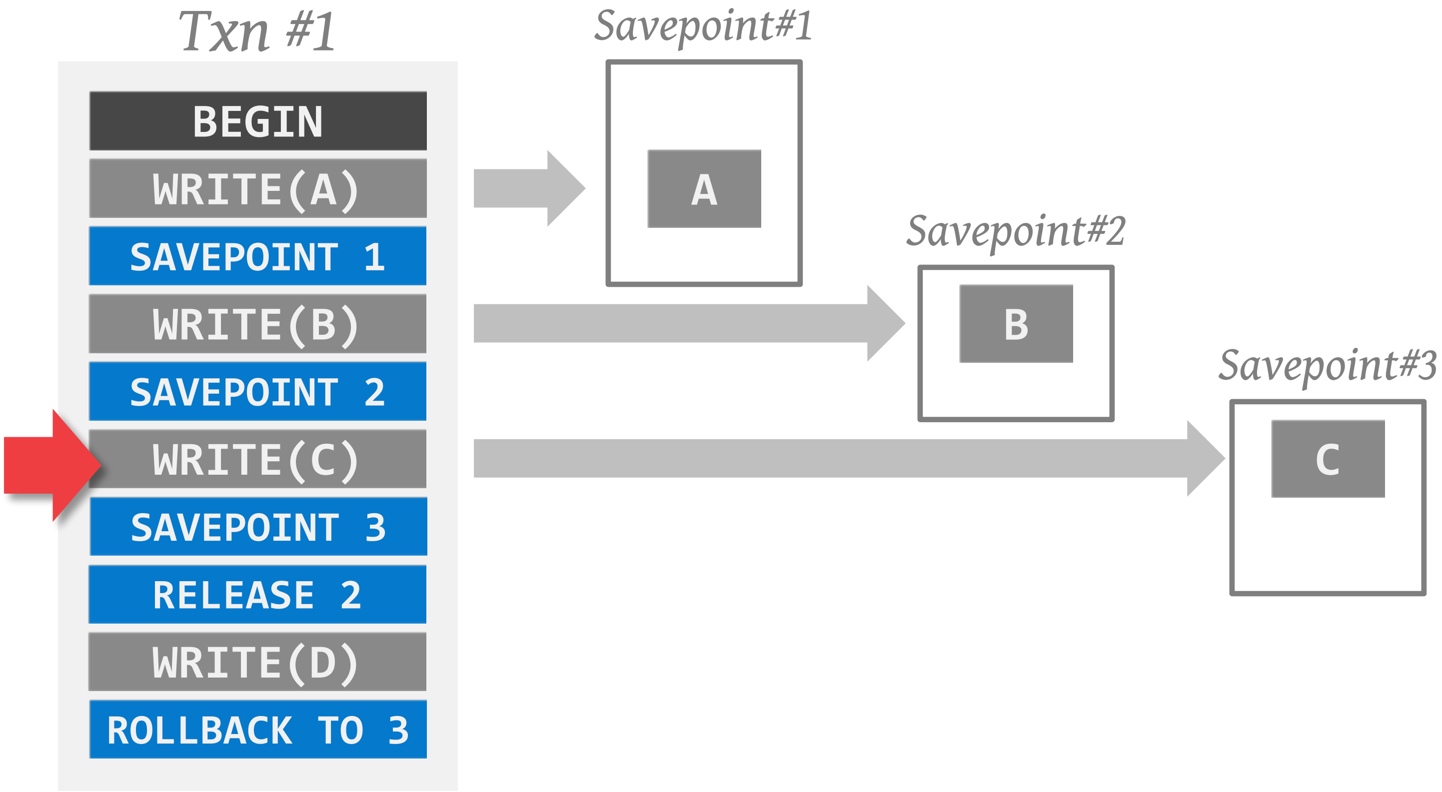
TRANSACTION SAVEPOINTS



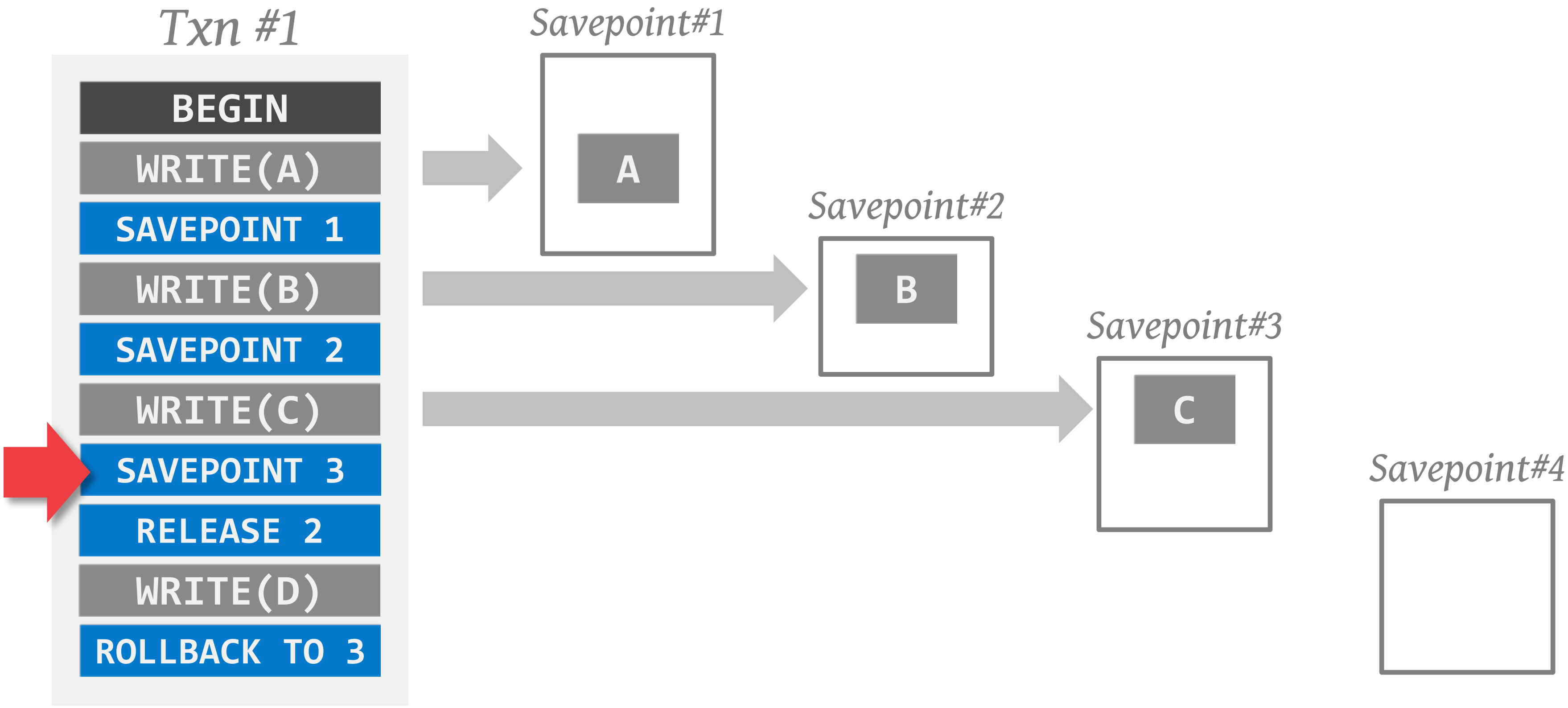
TRANSACTION SAVEPOINTS



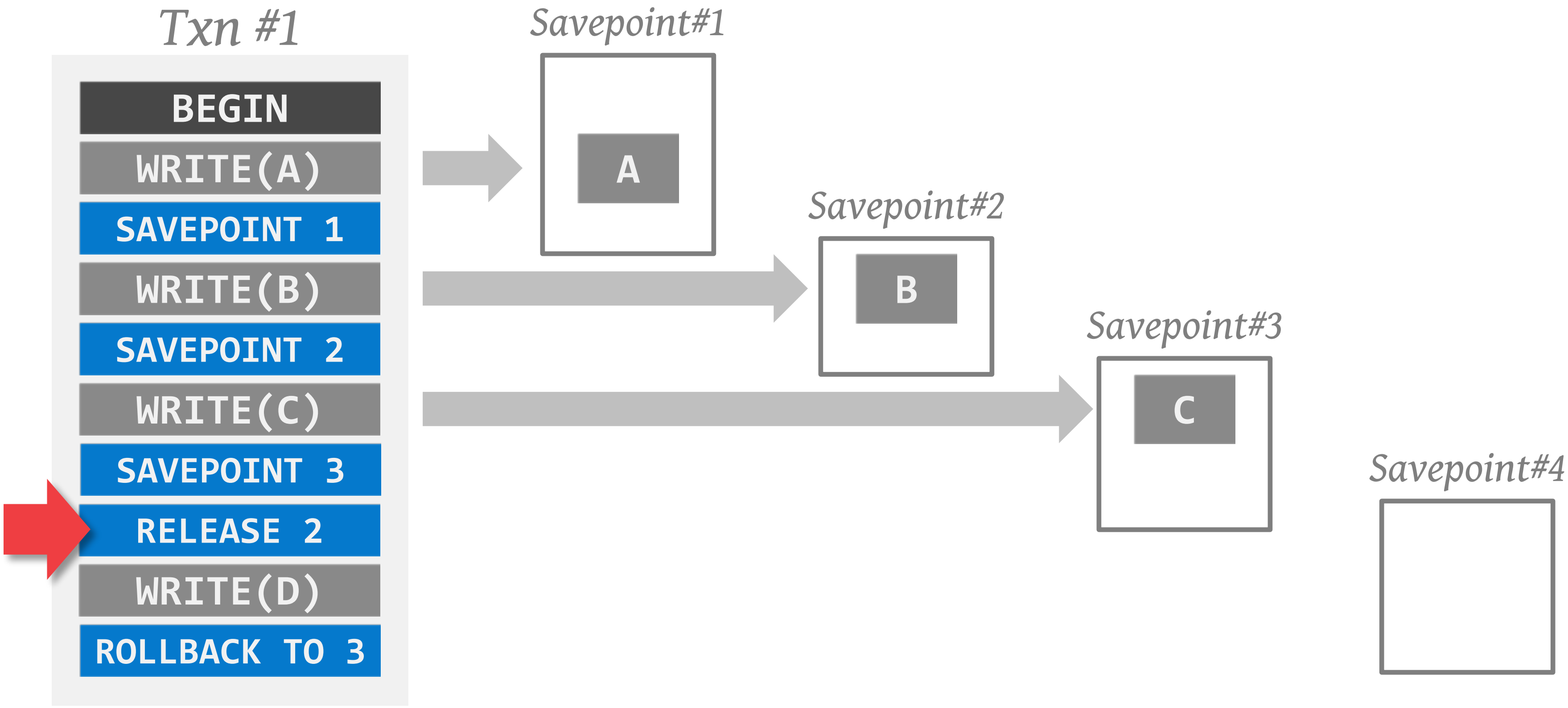
TRANSACTION SAVEPOINTS



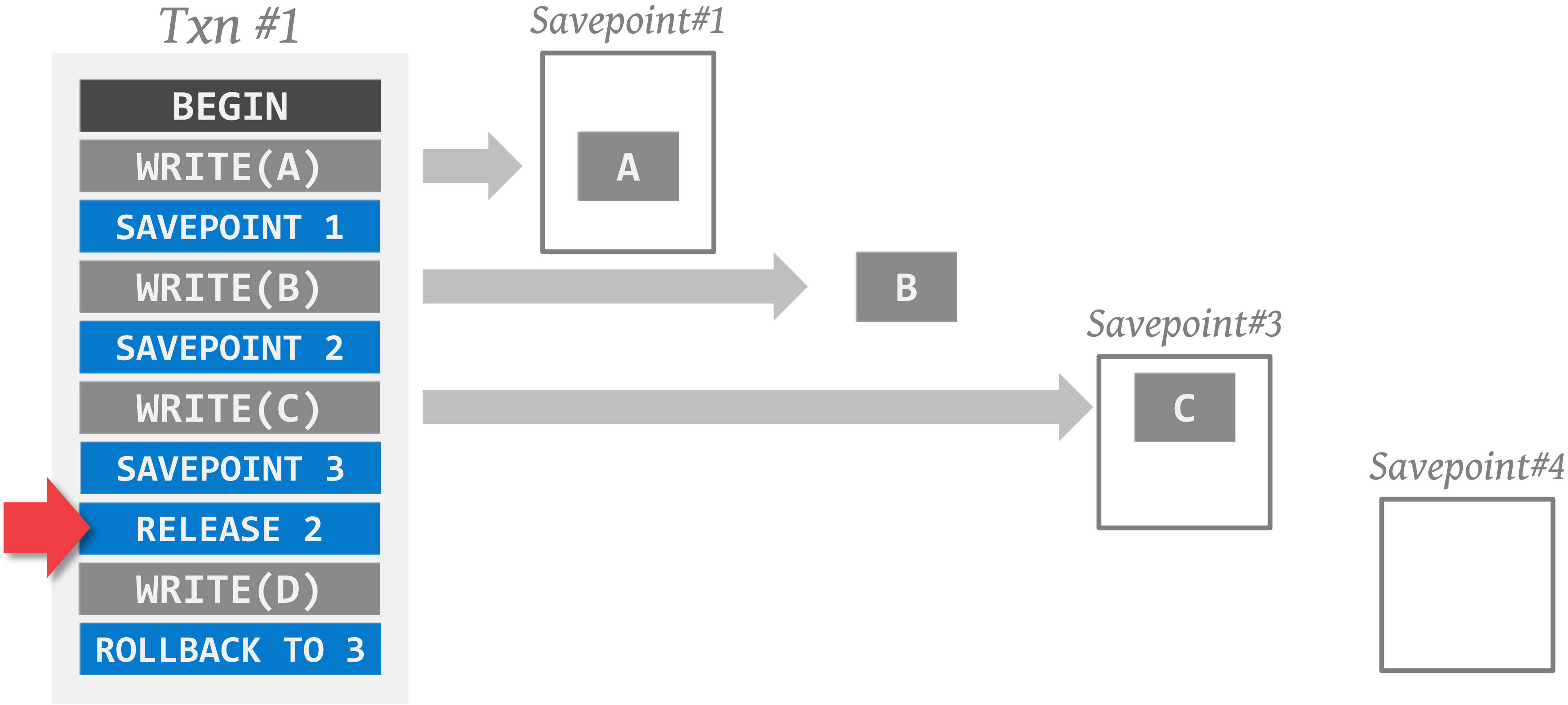
TRANSACTION SAVEPOINTS



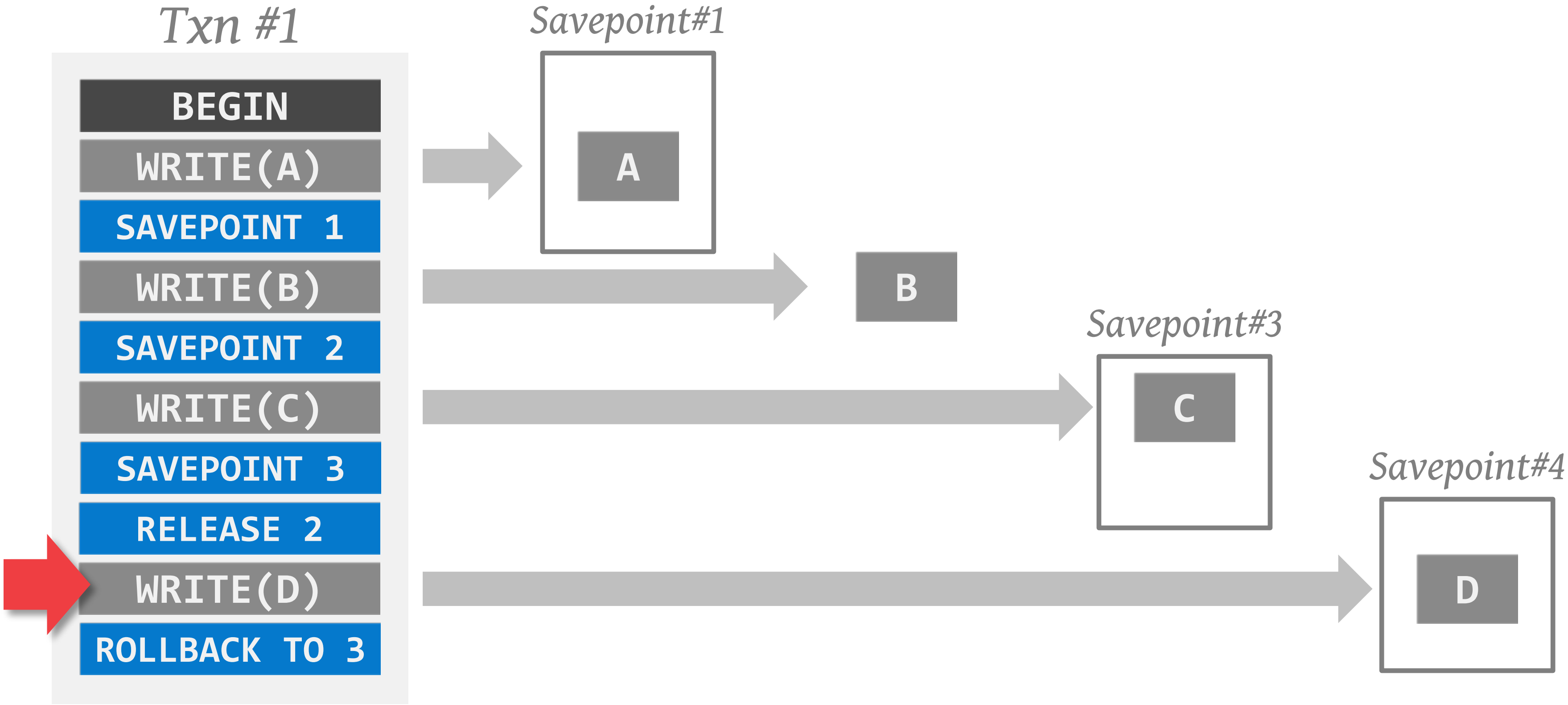
TRANSACTION SAVEPOINTS



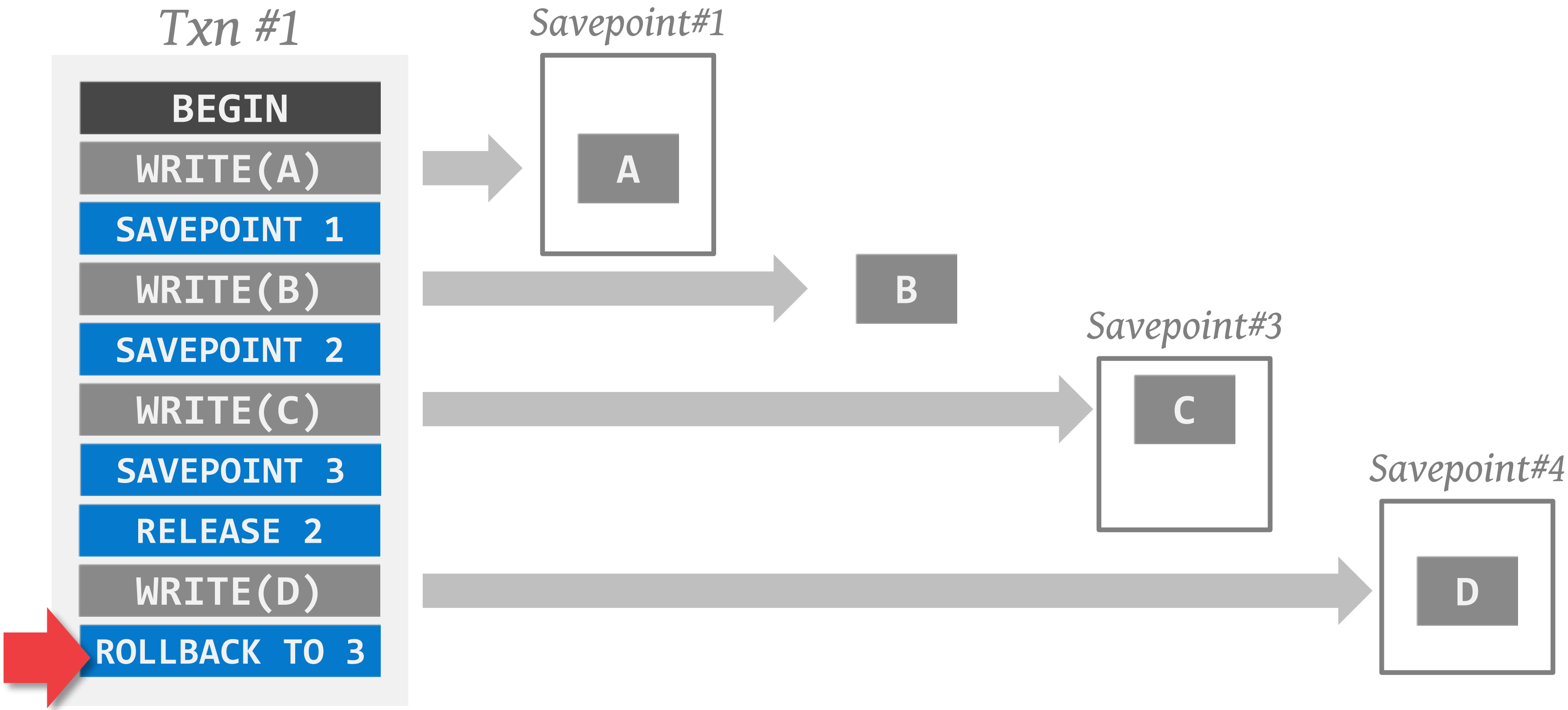
TRANSACTION SAVEPOINTS



TRANSACTION SAVEPOINTS



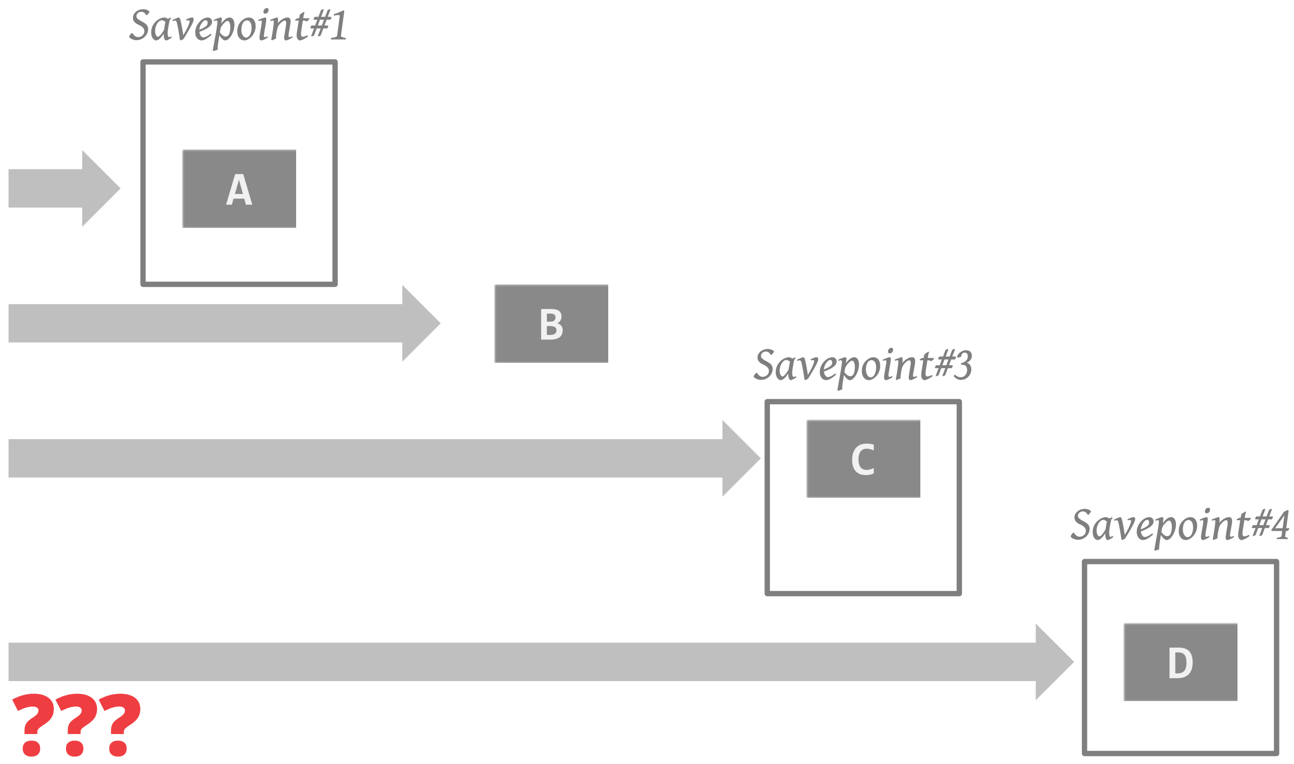
TRANSACTION SAVEPOINTS



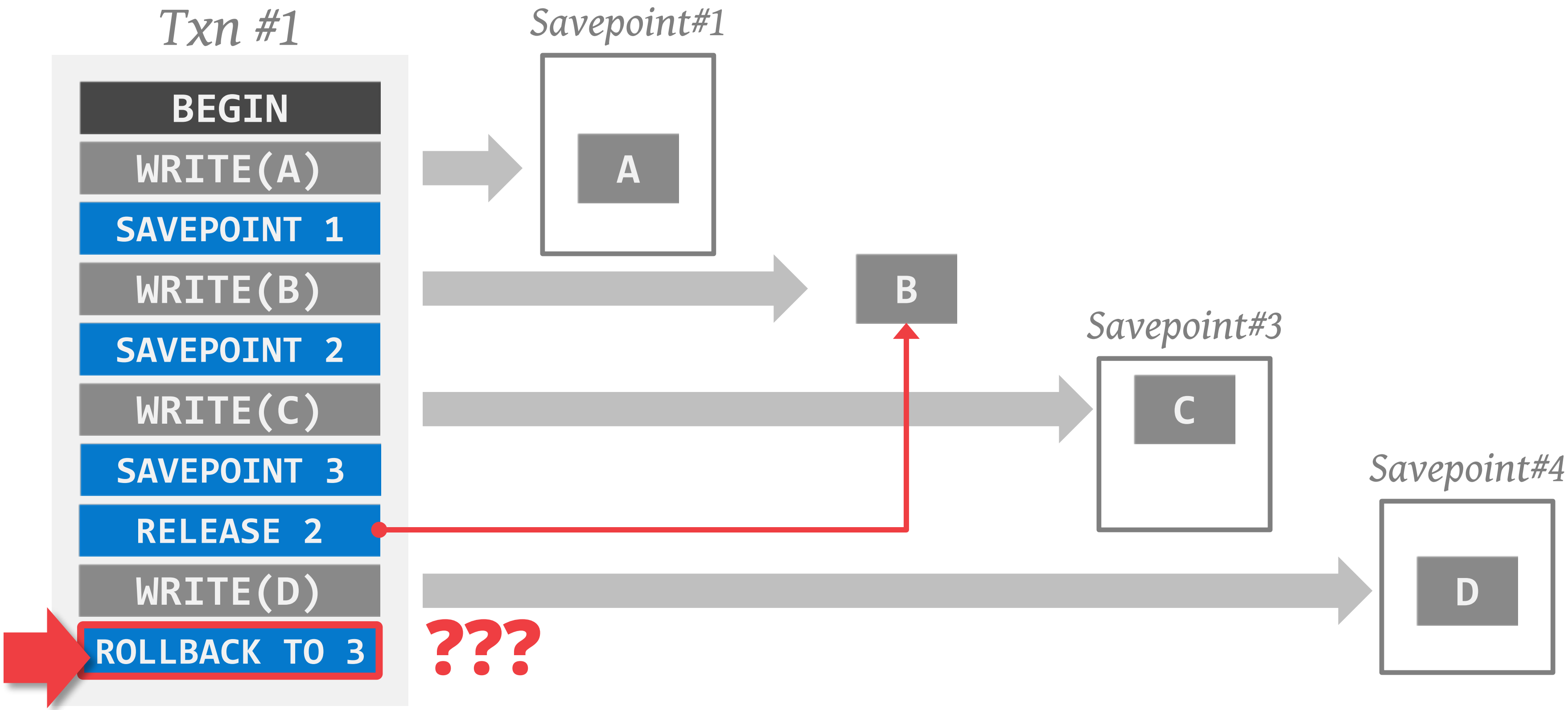
TRANSACTION SAVEPOINTS

Txn #1

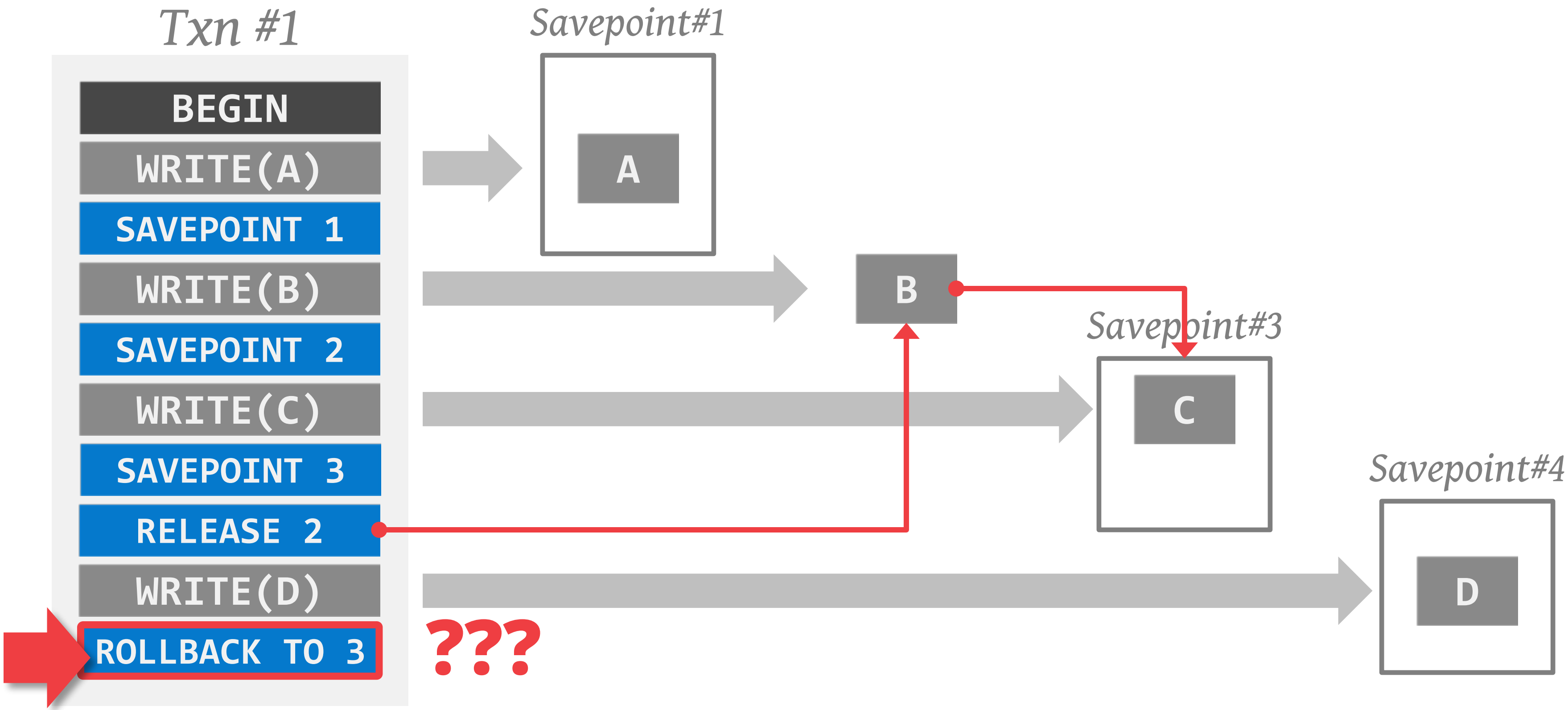
```
BEGIN
WRITE(A)
SAVEPOINT 1
WRITE(B)
SAVEPOINT 2
WRITE(C)
SAVEPOINT 3
RELEASE 2
WRITE(D)
ROLLBACK TO 3
```



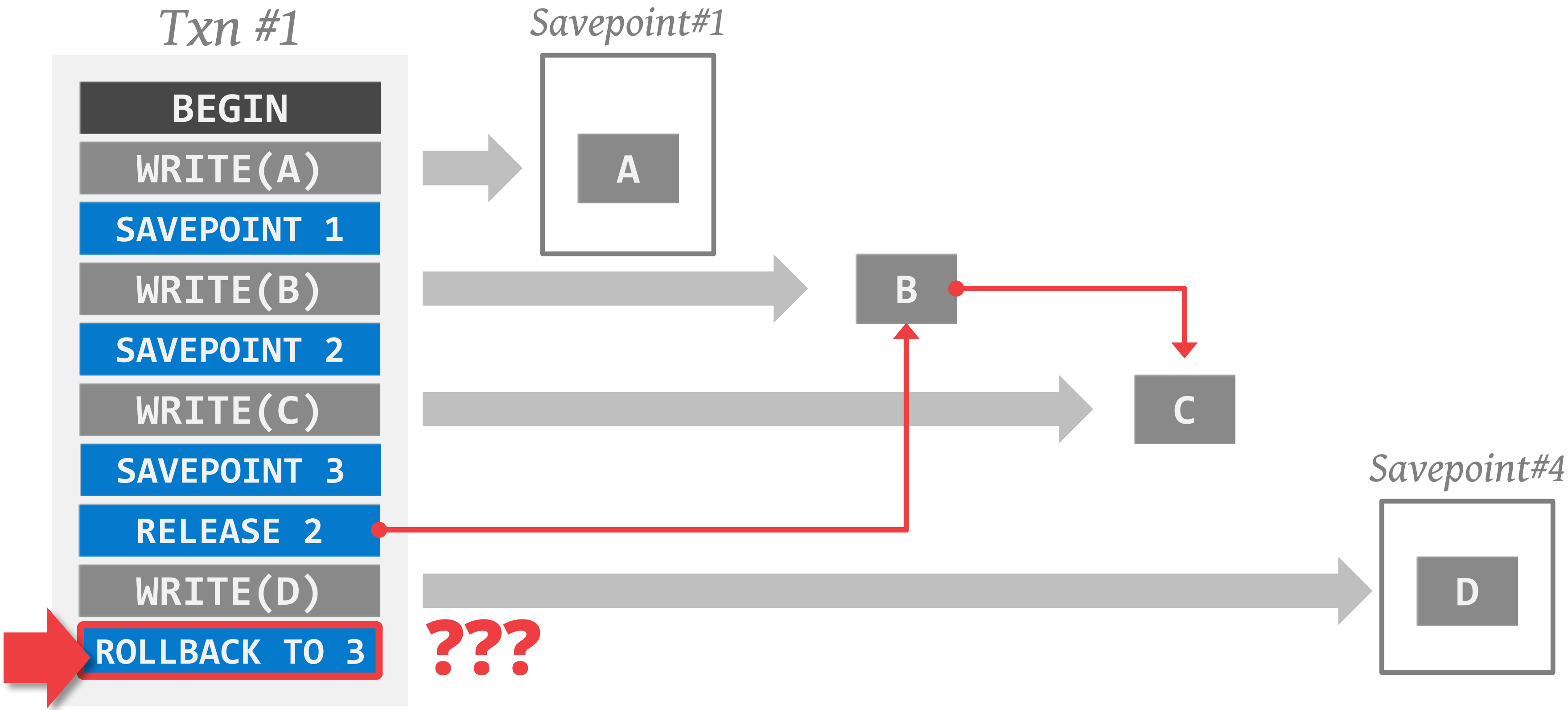
TRANSACTION SAVEPOINTS



TRANSACTION SAVEPOINTS



TRANSACTION SAVEPOINTS



TRANSACTION CHAINS

Multiple txns executed one after another.

Combined **COMMIT** / **BEGIN** operation is atomic.

→ No other txn can change the state of the database as seen by the second txn from the time that the first txn commits and the second txn begins.

Differences with savepoints:

- **COMMIT** allows the DBMS to free locks.
- Cannot rollback previous txns in chain.

TRANSACTION CHAINS

Txn #1

BEGIN
WRITE(A)
COMMIT

Txn #2

BEGIN
WRITE(B)
COMMIT

Txn #3

BEGIN
WRITE(C)
ROLLBACK

TRANSACTION CHAINS

Txn #1

BEGIN
WRITE(A)
COMMIT

Txn #2

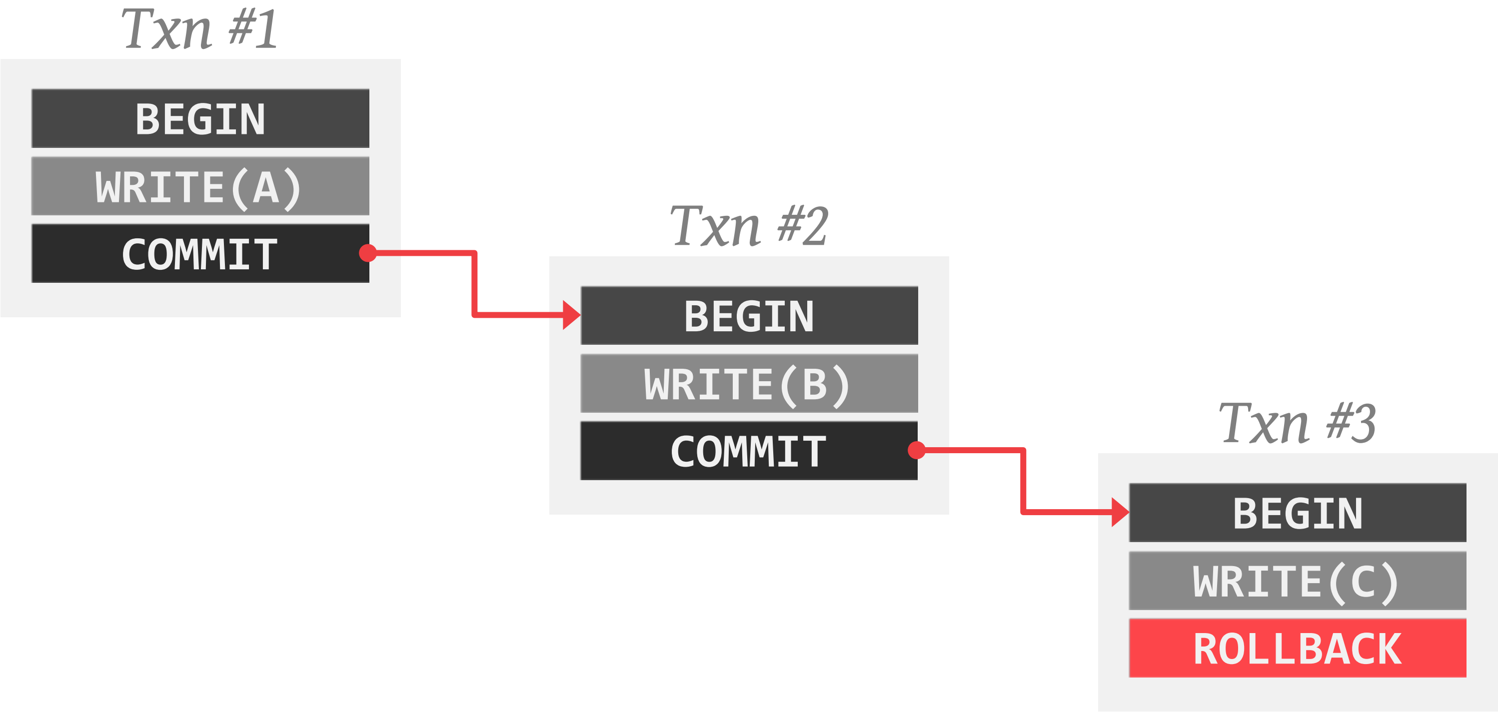
BEGIN
WRITE(B)
COMMIT

Txn #3

BEGIN
WRITE(C)
ROLLBACK

TRANSACTION CHAINS

TRANSACTION CHAINS



NESTED TRANSACTIONS

Savepoints organize a transaction as a **sequence** of actions that can be rolled back individually.

Nested txns form a **hierarchy** of work.

→ The outcome of a child txn depends on the outcome of its parent txn.

NESTED TRANSACTIONS

Txn #1

```
BEGIN
WRITE(A)
BEGIN
WRITE(B)
BEGIN
WRITE(C)
COMMIT
WRITE(D)
ROLLBACK
COMMIT
```

NESTED TRANSACTIONS

Txn #1

```
BEGIN
WRITE(A)
BEGIN
WRITE(B)
BEGIN
WRITE(C)
COMMIT
WRITE(D)
ROLLBACK
COMMIT
```

NESTED TRANSACTIONS

Txn #1

```
BEGIN
WRITE(A)
BEGIN
WRITE(B)
BEGIN
WRITE(C)
COMMIT
WRITE(D)
ROLLBACK
COMMIT
```

NESTED TRANSACTIONS

Txn #1

```
BEGIN
WRITE(A)
BEGIN
WRITE(B)
BEGIN
WRITE(C)
COMMIT
WRITE(D)
ROLLBACK
COMMIT
```

NESTED TRANSACTIONS

Txn #1

BEGIN
WRITE(A)
BEGIN
WRITE(B)
BEGIN
WRITE(C)
COMMIT
WRITE(D)
ROLLBACK
COMMIT

NESTED TRANSACTIONS

Txn #1

```
BEGIN
WRITE(A)
BEGIN
WRITE(B)
BEGIN
WRITE(C)
COMMIT
WRITE(D)
ROLLBACK
COMMIT
```

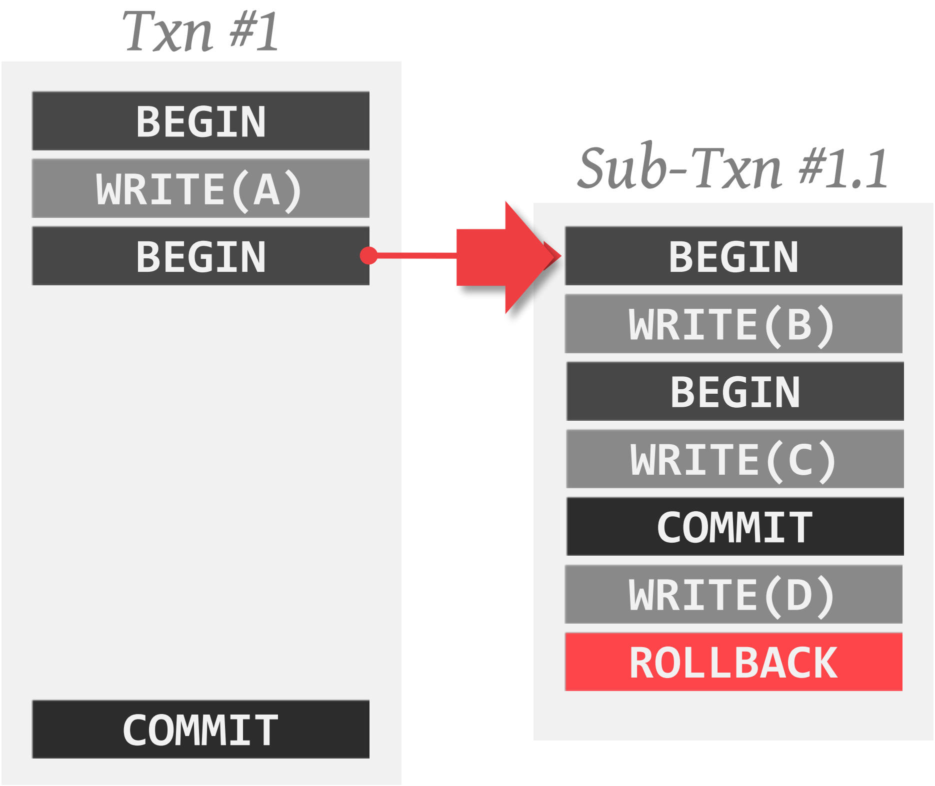
NESTED TRANSACTIONS

Txn #1

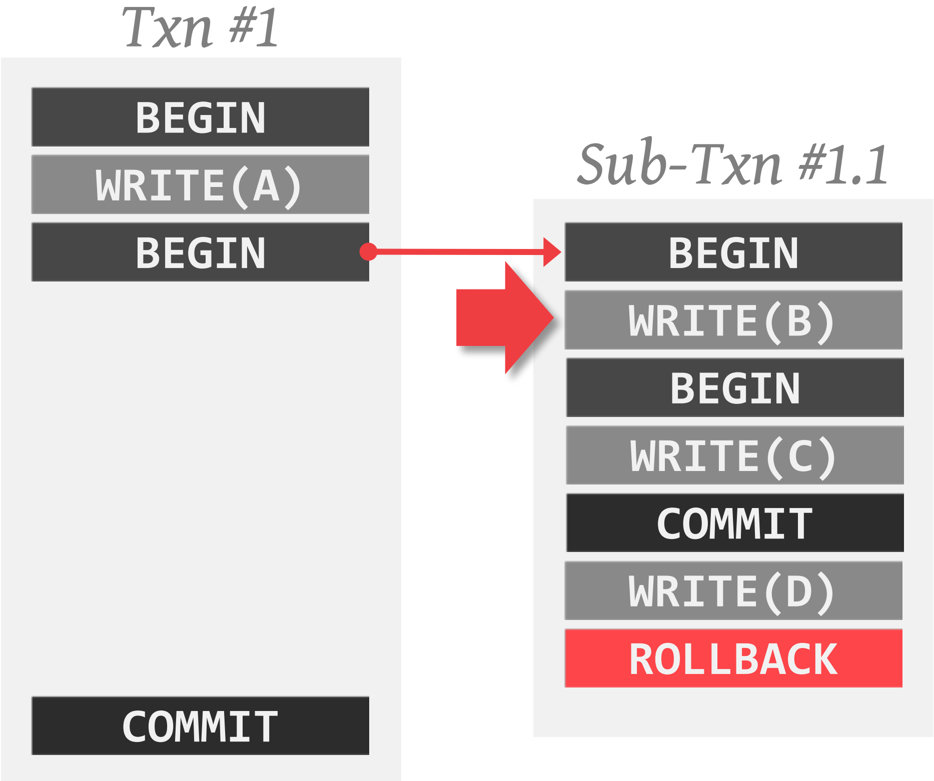
```
BEGIN
WRITE(A)
BEGIN
WRITE(B)
BEGIN
WRITE(C)
COMMIT
WRITE(D)
ROLLBACK
COMMIT
```

Sub-Txn #1.1

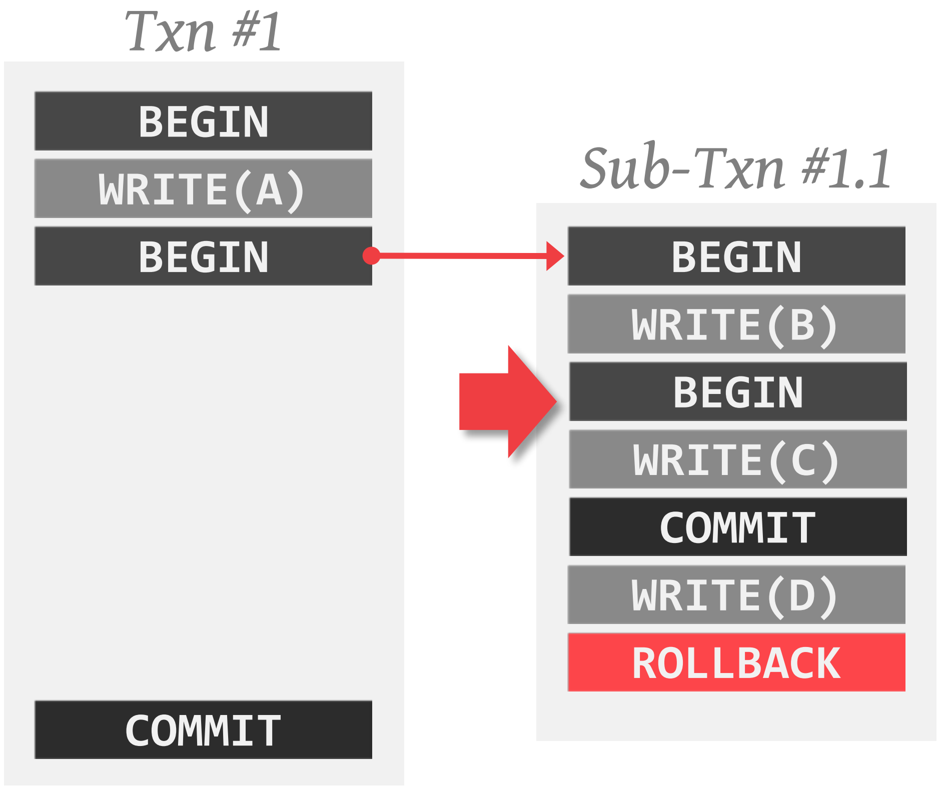
NESTED TRANSACTIONS



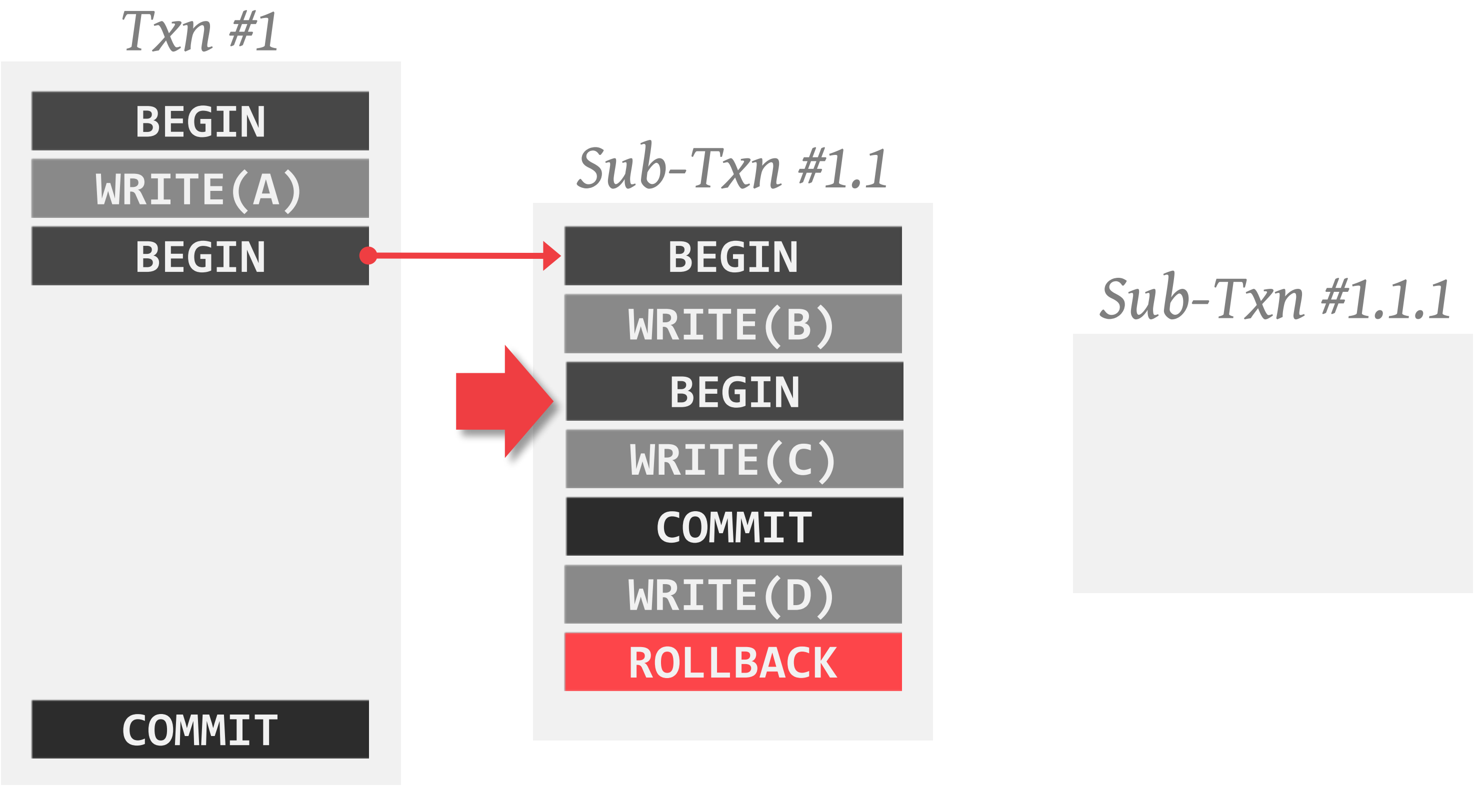
NESTED TRANSACTIONS



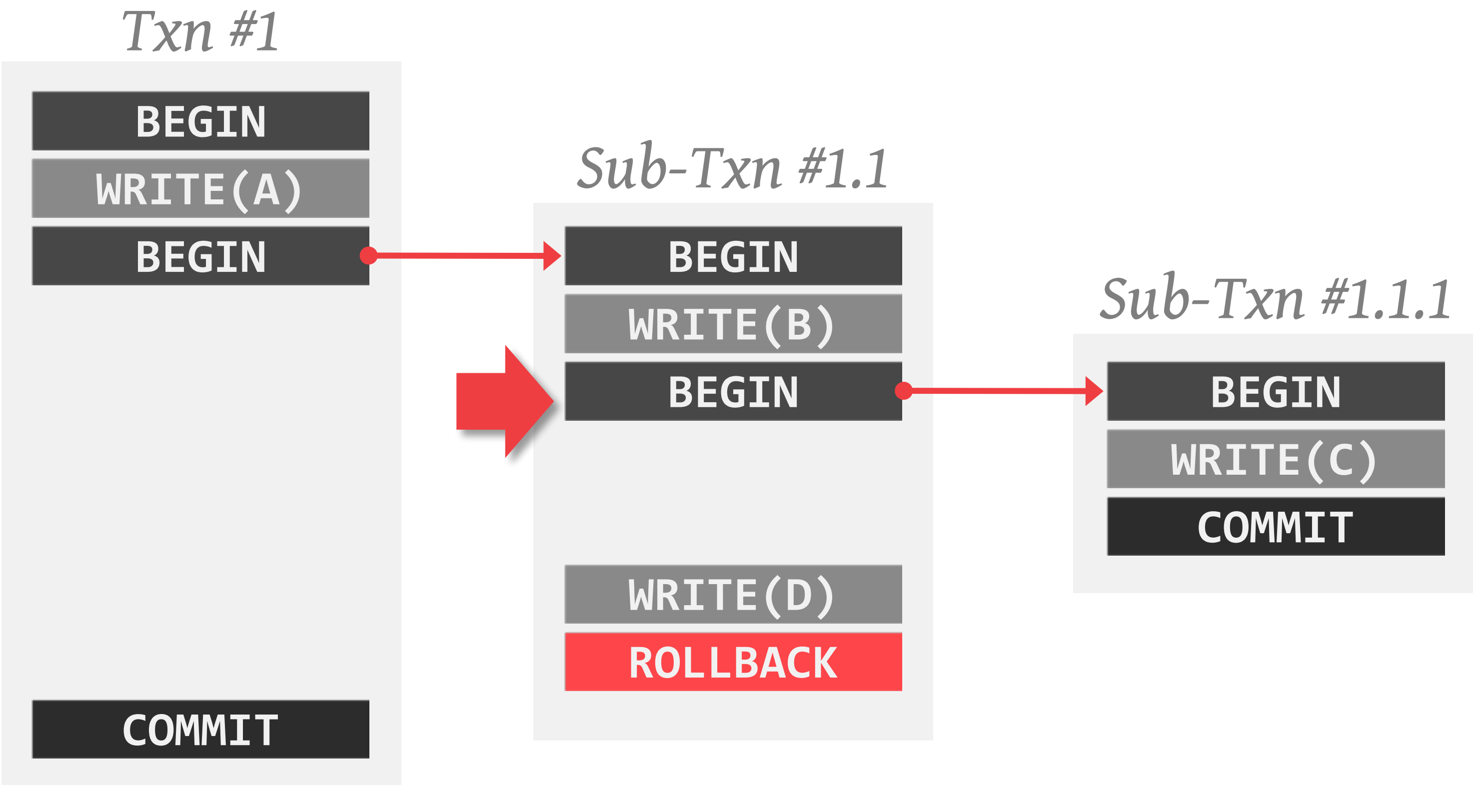
NESTED TRANSACTIONS



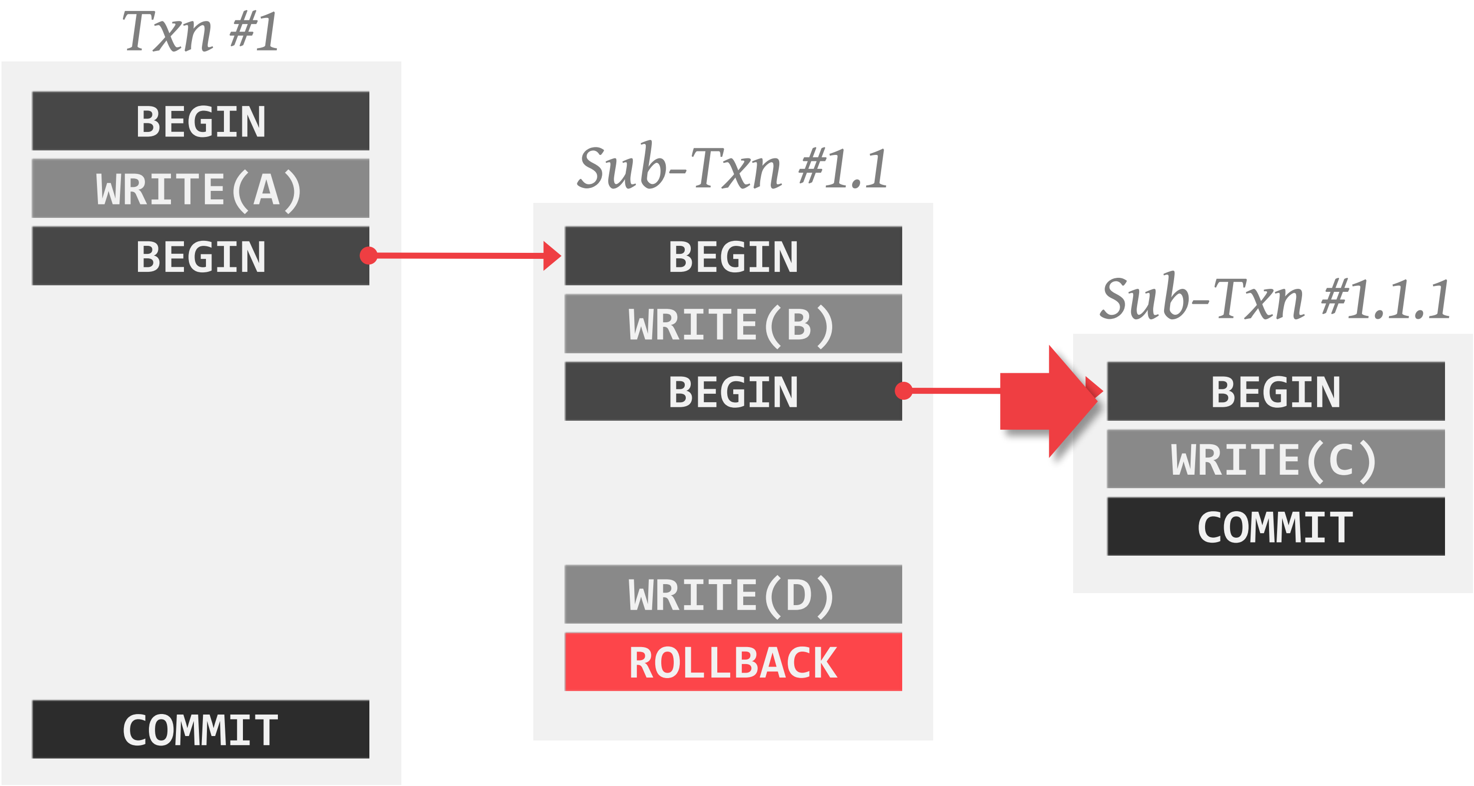
NESTED TRANSACTIONS



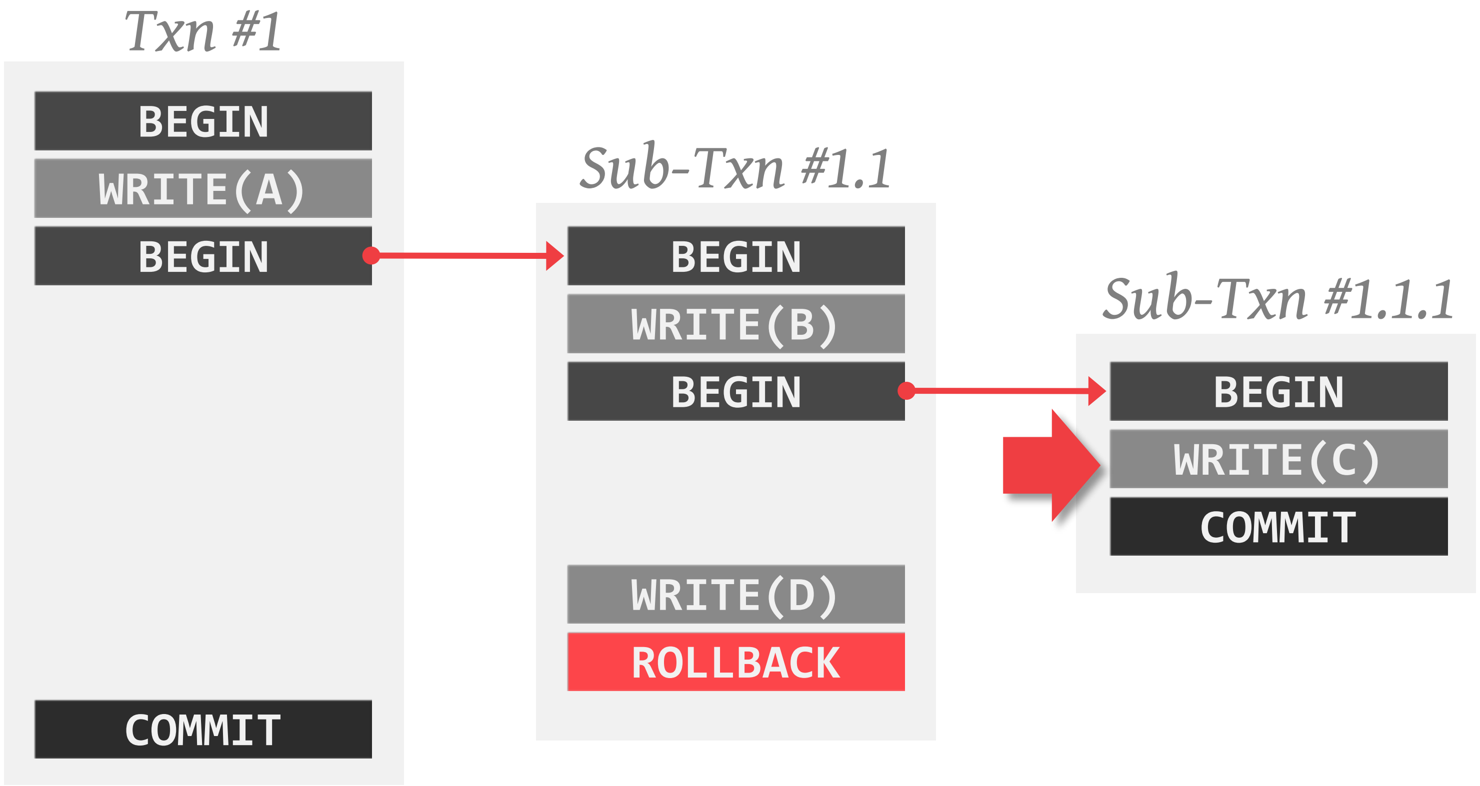
NESTED TRANSACTIONS



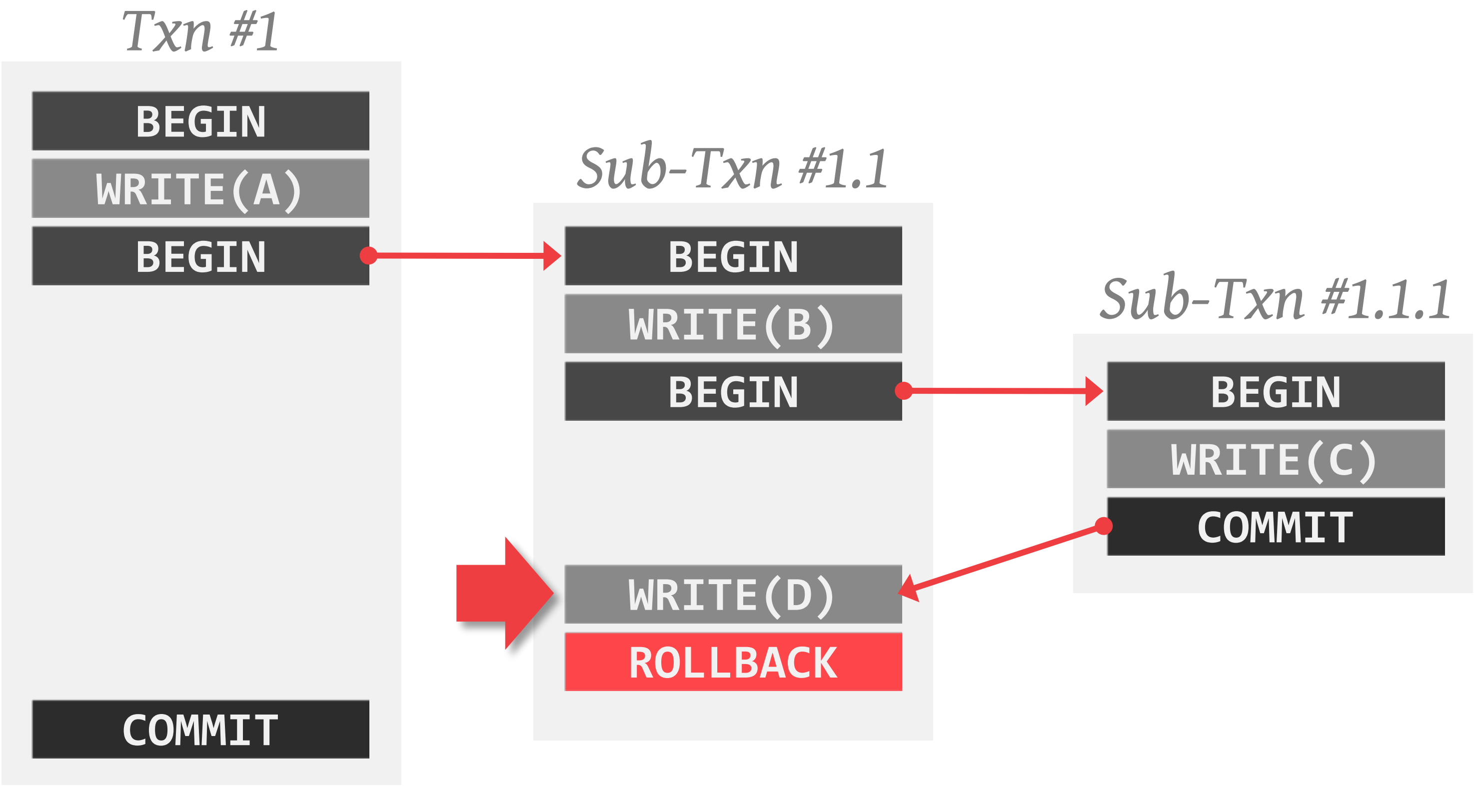
NESTED TRANSACTIONS



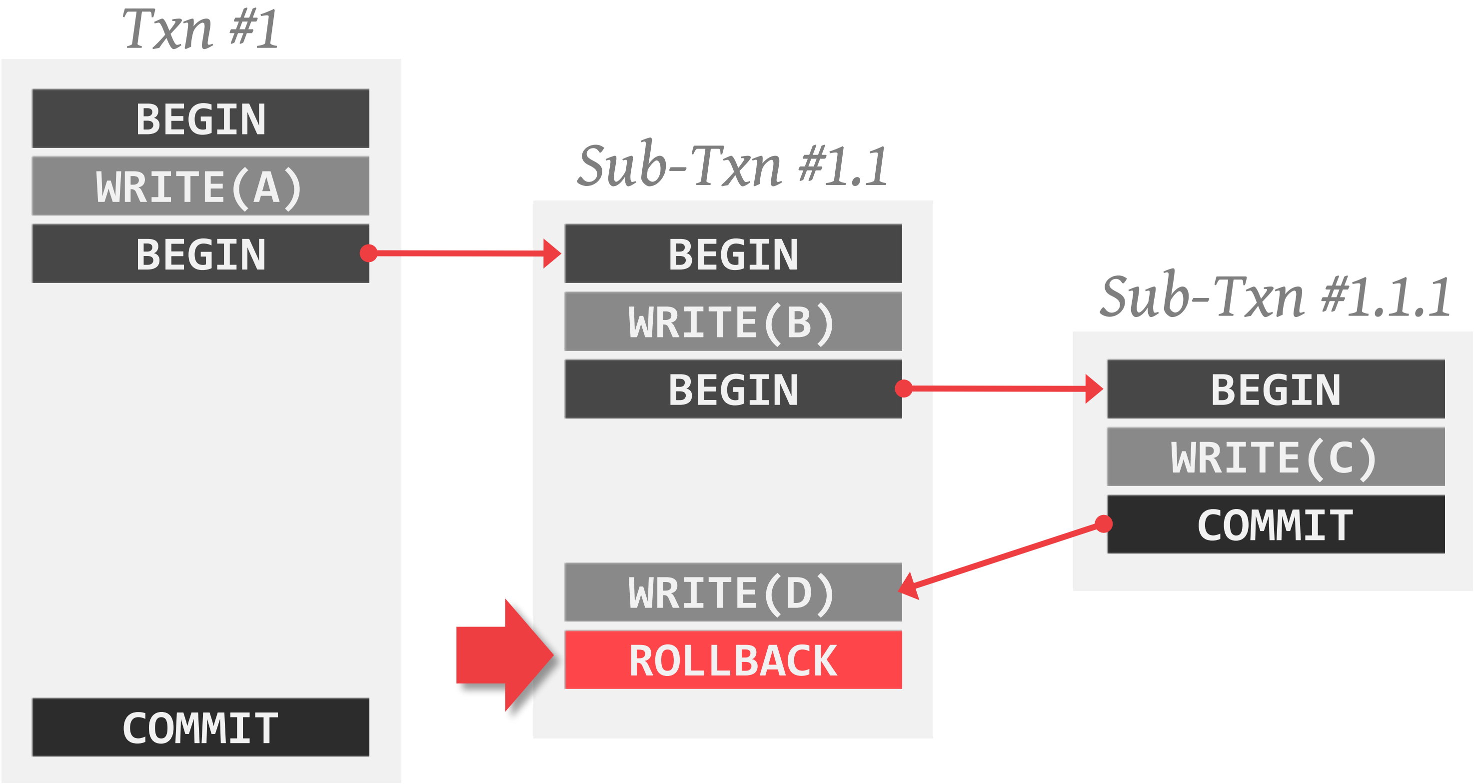
NESTED TRANSACTIONS



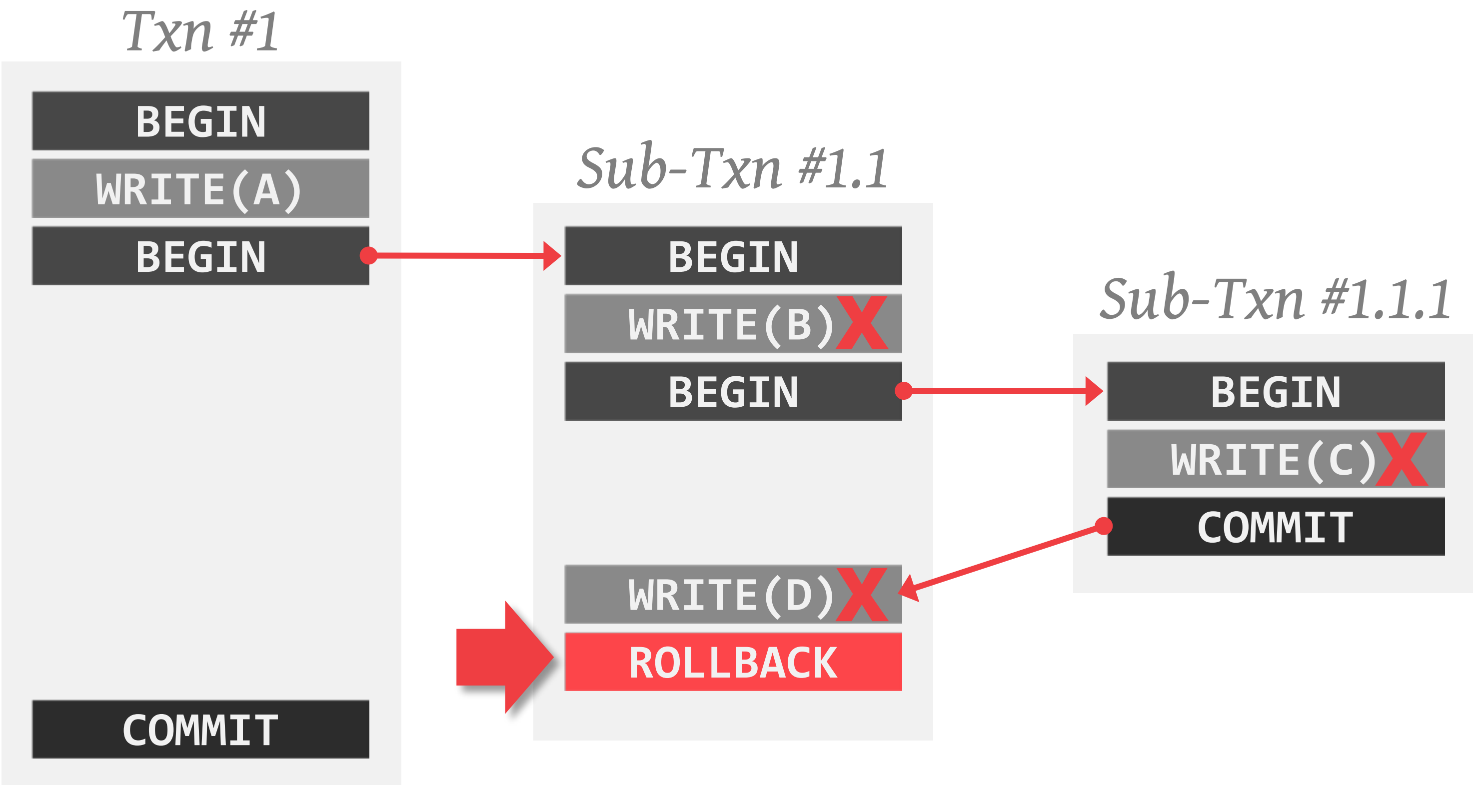
NESTED TRANSACTIONS



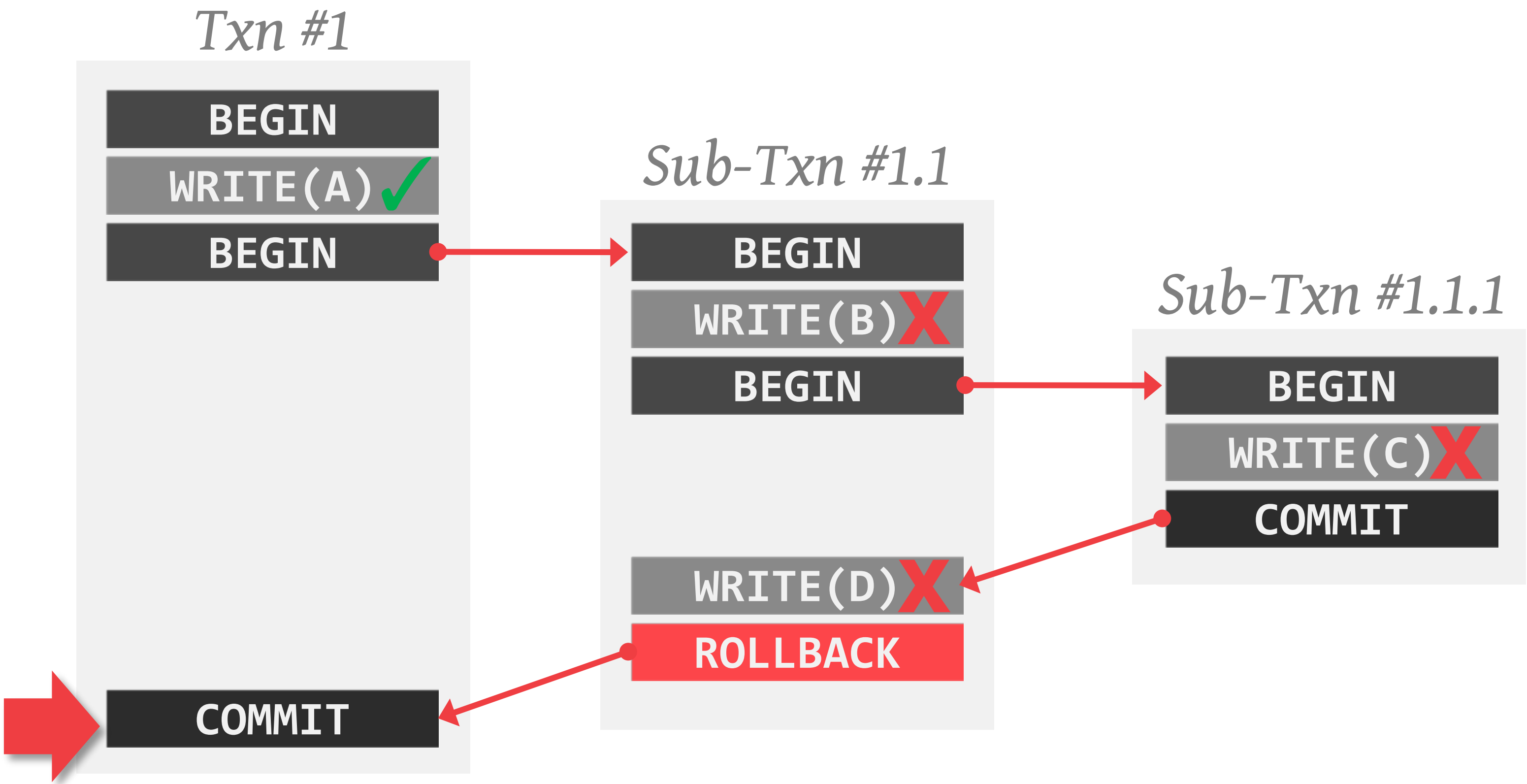
NESTED TRANSACTIONS



NESTED TRANSACTIONS



NESTED TRANSACTIONS



BULK UPDATE PROBLEM

These other txn models are nice, but they still do not solve our bulk update problem.

Chained txns seems like the right idea but they require the application to handle failures and maintain it's own state.

→ Has to be able to reverse changes when things fail.

COMPENSATING TRANSACTIONS

A special type of txn that is designed to semantically reverse the effects of another already committed txn.

Reversal has to be **logical** instead of physical.

→ Example: Decrement a counter by one instead of reverting to the original value.

SAGA TRANSACTIONS

A sequence of chained txns $T_1, \ldots, T_n$ and compensating txns $C_1, \ldots, C_{n-1}$ where one of the following is guaranteed:

→ The txns will commit in the order $T1, \ldots, Tn$

→ The txns will commit in the order $T_1, \ldots, T_j, C_j, \ldots, C_1$ (*where* $j < n$)

SAGAS
SIGMOD, pp. 249-259, 2014.

SAGA TRANSACTIONS

Txn #1	*Txn #2*	*Txn #3*
BEGIN	BEGIN	BEGIN
WRITE(A+1)	WRITE(B+1)	WRITE(C+1)
COMMIT	COMMIT	

Comp Txn #1	*Comp Txn #2*	*Comp Txn #3*
BEGIN	BEGIN	BEGIN
WRITE(A-1)	WRITE(B-1)	WRITE(C-1)
COMMIT	COMMIT	COMMIT

SAGA TRANSACTIONS

Txn #1	*Txn #2*	*Txn #3*
BEGIN	BEGIN	BEGIN
WRITE(A+1)	WRITE(B+1)	WRITE(C+1)
COMMIT	COMMIT	

Comp Txn #1	*Comp Txn #2*	*Comp Txn #3*
BEGIN	BEGIN	BEGIN
WRITE(A-1)	WRITE(B-1)	WRITE(C-1)
COMMIT	COMMIT	COMMIT

SAGA TRANSACTIONS

Txn #1

BEGIN
WRITE(A+1)
COMMIT

Txn #2

BEGIN
WRITE(B+1)
COMMIT

Txn #3

BEGIN
WRITE(C+1)

Comp Txn #1

BEGIN
WRITE(A-1)
COMMIT

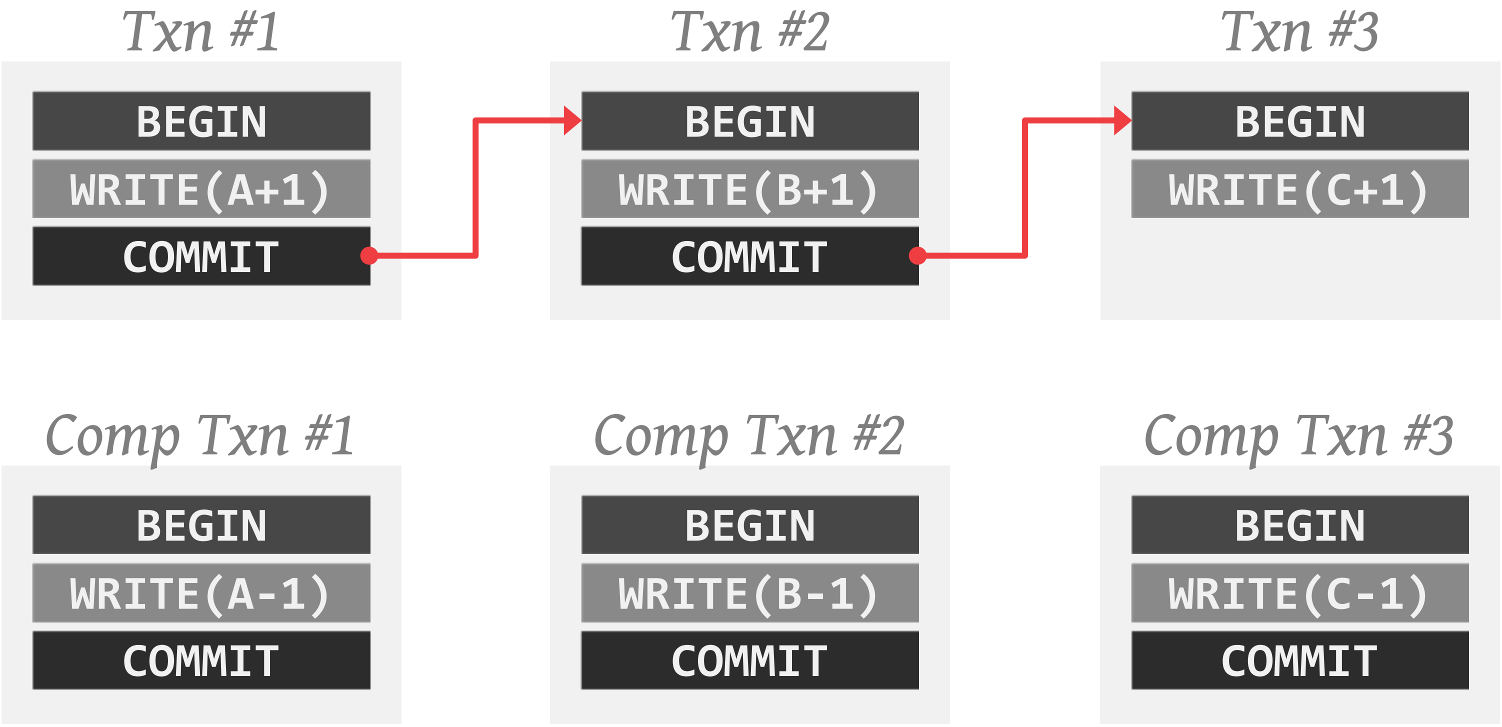
Comp Txn #2

BEGIN
WRITE(B-1)
COMMIT

Comp Txn #3

BEGIN
WRITE(C-1)
COMMIT

SAGA TRANSACTIONS

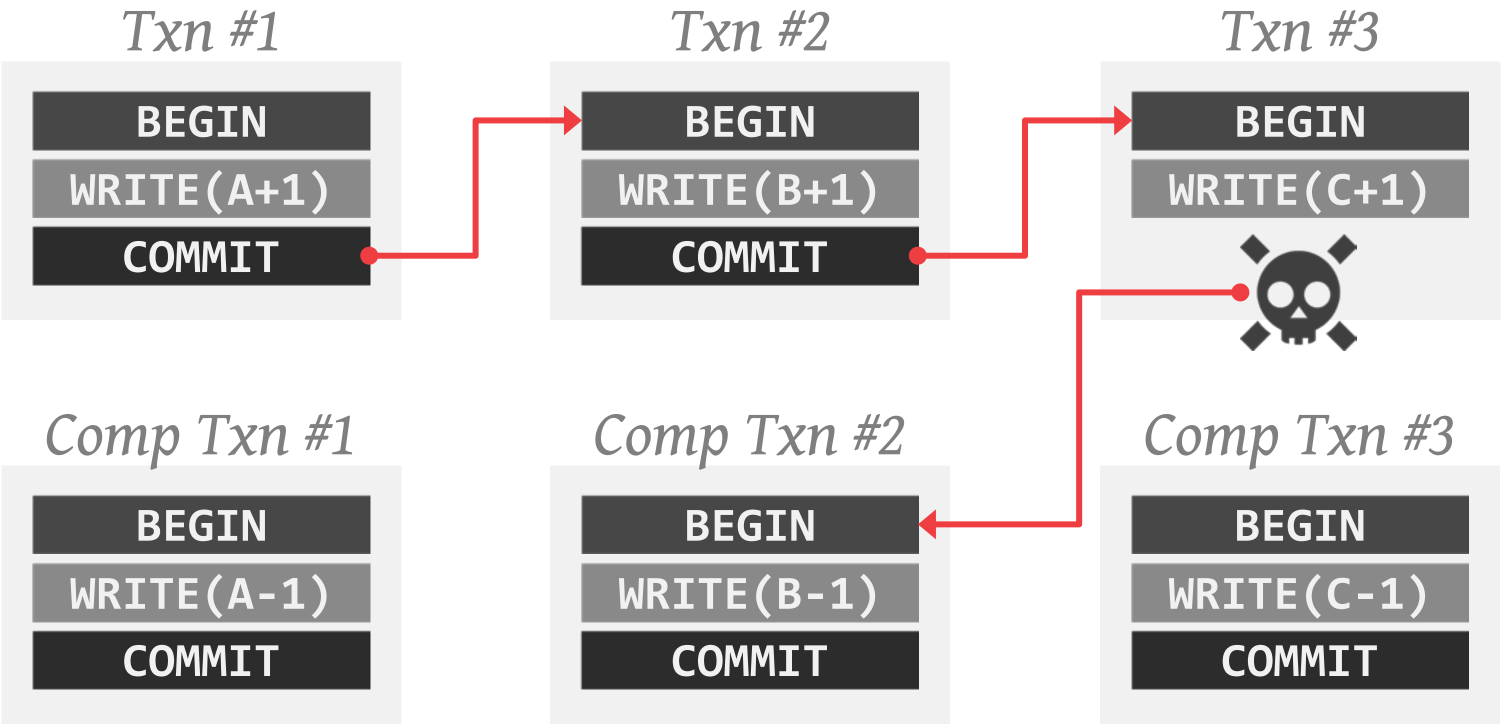


SAGA TRANSACTIONS

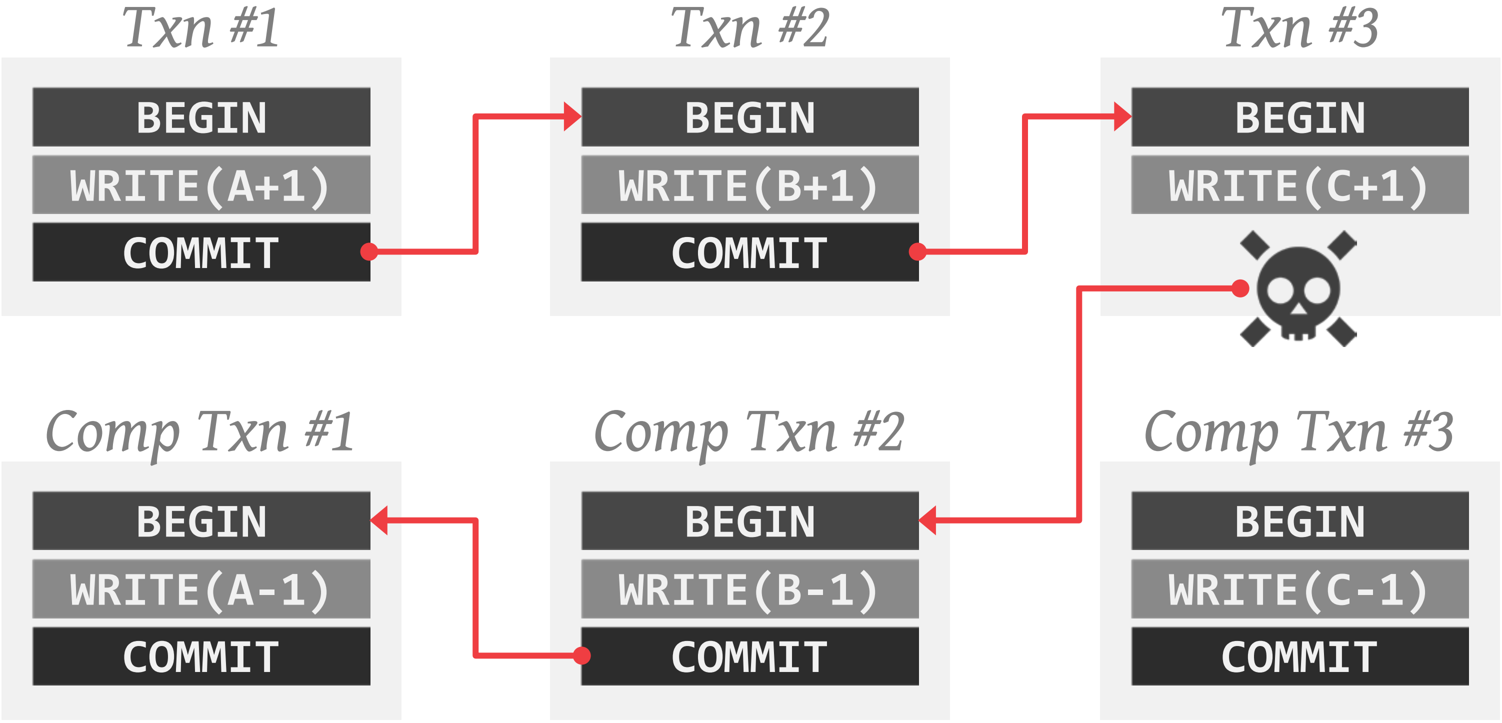
Txn #1	*Txn #2*	*Txn #3*
BEGIN	BEGIN	BEGIN
WRITE(A+1)	WRITE(B+1)	WRITE(C+1)
COMMIT	COMMIT	

Comp Txn #1	*Comp Txn #2*	*Comp Txn #3*
BEGIN	BEGIN	BEGIN
WRITE(A-1)	WRITE(B-1)	WRITE(C-1)
COMMIT	COMMIT	COMMIT

SAGA TRANSACTIONS



SAGA TRANSACTIONS



CONCURRENCY CONTROL

The protocol to allow txns to access a database in a multi-programmed fashion while preserving the illusion that each of them is executing alone on a dedicated system.

→ The goal is to have the effect of a group of txns on the database's state is equivalent to any serial execution of all txns.

Provides **A**tomicity + **I**solation in ACID

TXN INTERNAL STATE

Undo Log Entries
→ Stored in an in-memory data structure.
→ Dropped on commit.

Redo Log Entries
→ Append to the in-memory tail of WAL.
→ Flushed to disk on commit.

Read/Write Set
→ Depends on the concurrency control scheme.

CONCURRENCY CONTROL SCHEMES

Two-Phase Locking (2PL)

→ Assume txns will conflict so they must acquire locks on elements before they are allowed to access them.

Timestamp Ordering (T/O)

→ Assume that conflicts are rare so txns do not need to acquire locks and instead check for conflicts at commit time.

TWO-PHASE LOCKING

Txn #1

TWO-PHASE LOCKING

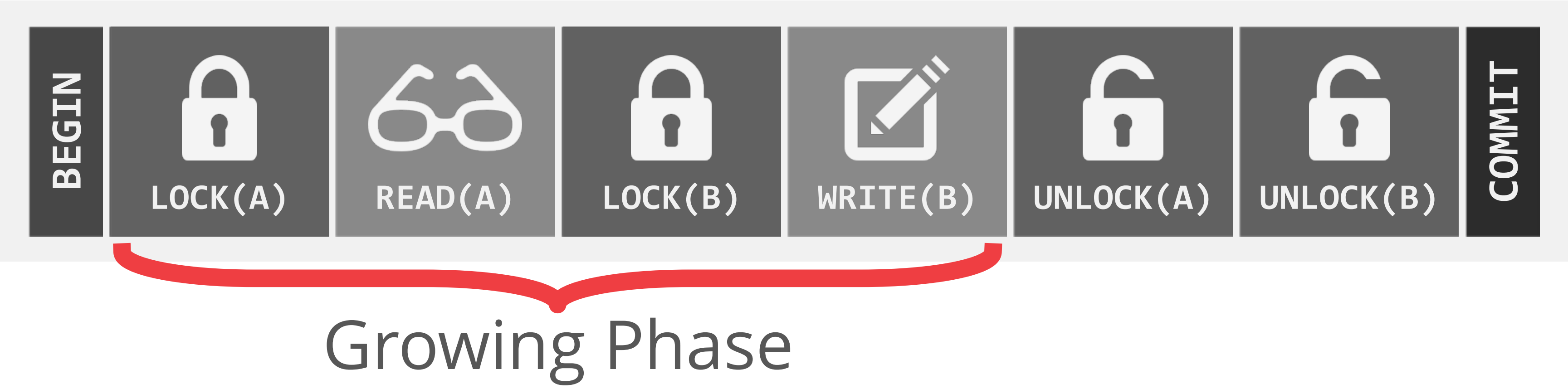
Txn #1

TWO-PHASE LOCKING

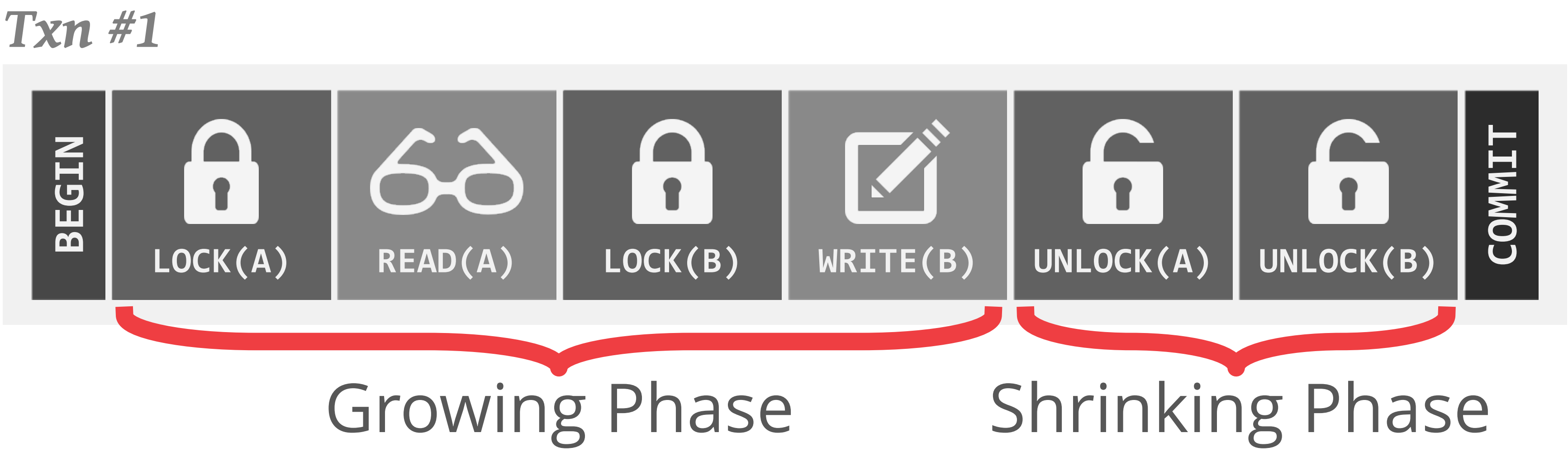
Txn #1

TWO-PHASE LOCKING

Txn #1



TWO-PHASE LOCKING



TWO-PHASE LOCKING

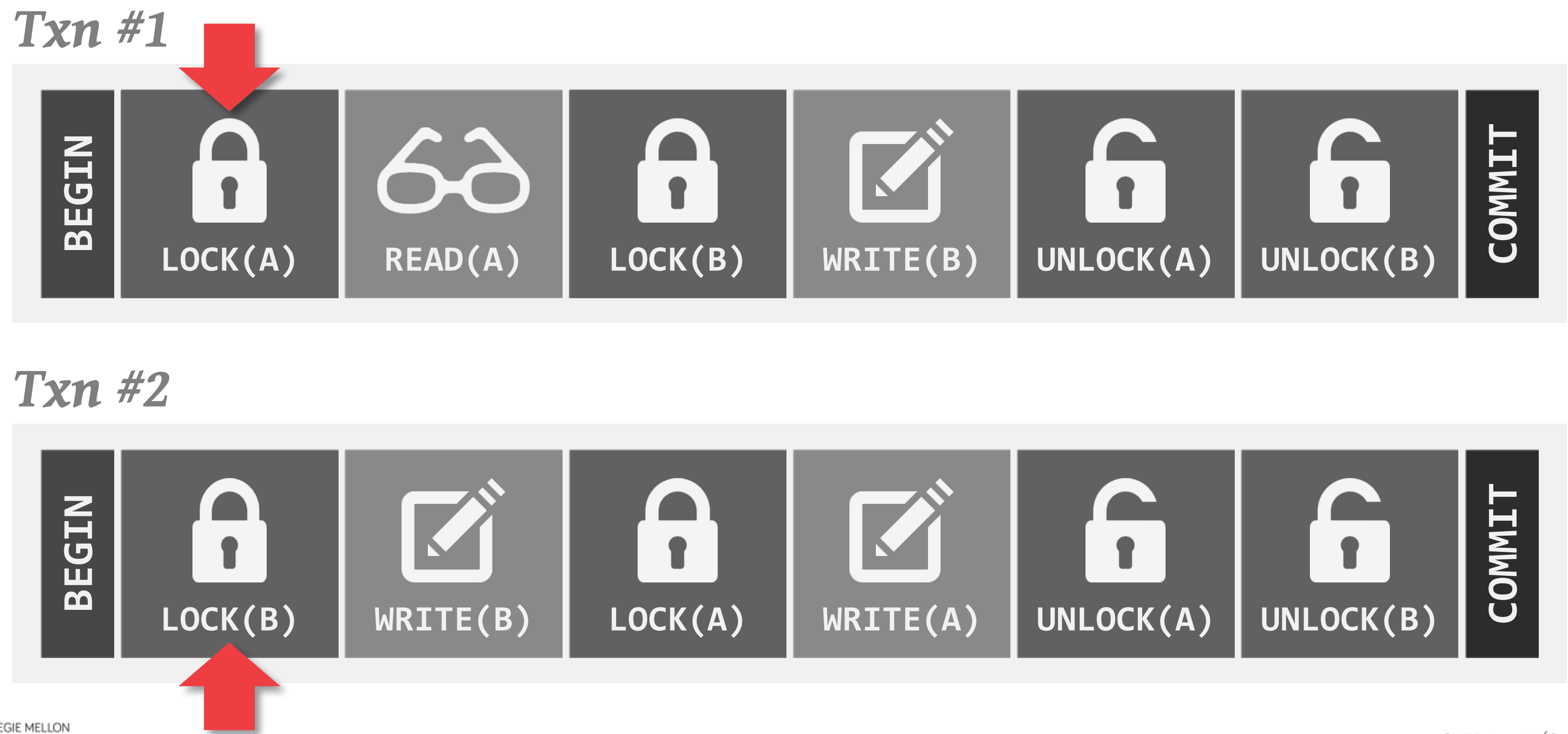
Txn #1

TWO-PHASE LOCKING

Txn #1

Txn #2

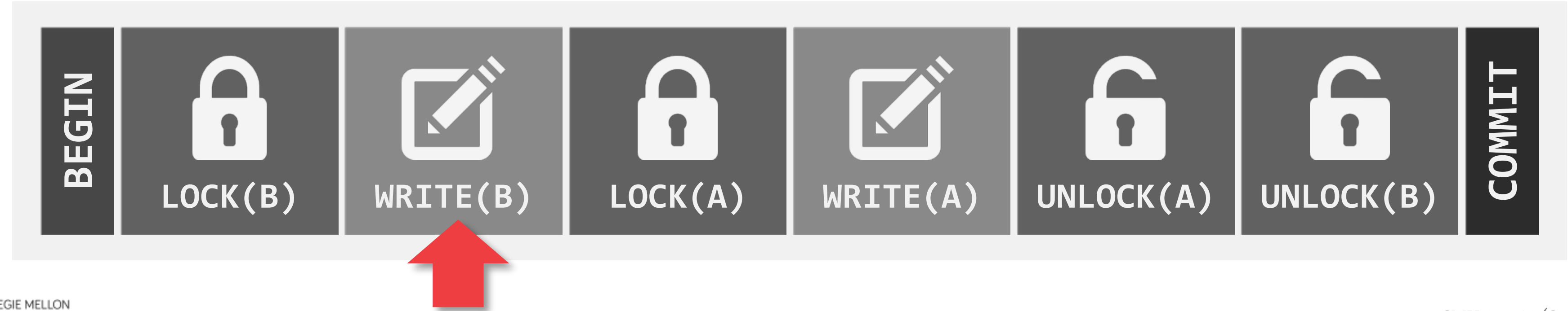
TWO-PHASE LOCKING



TWO-PHASE LOCKING

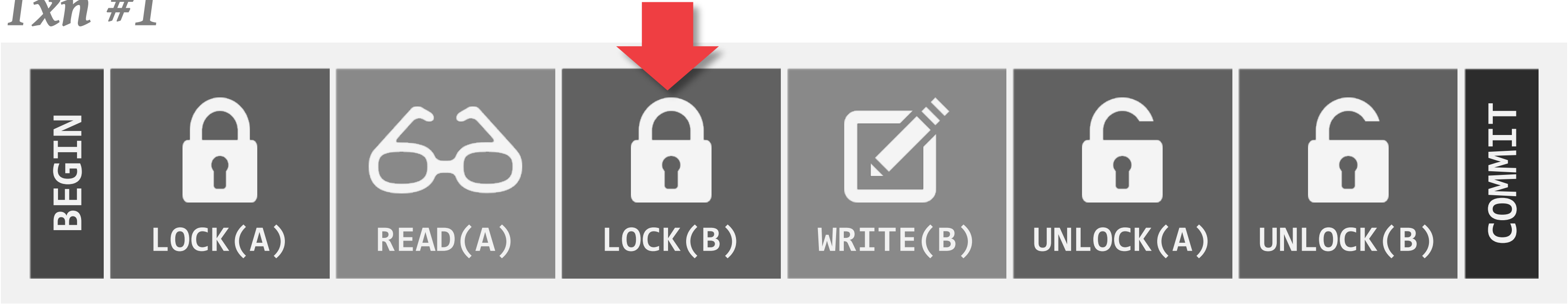
Txn #1

Txn #2

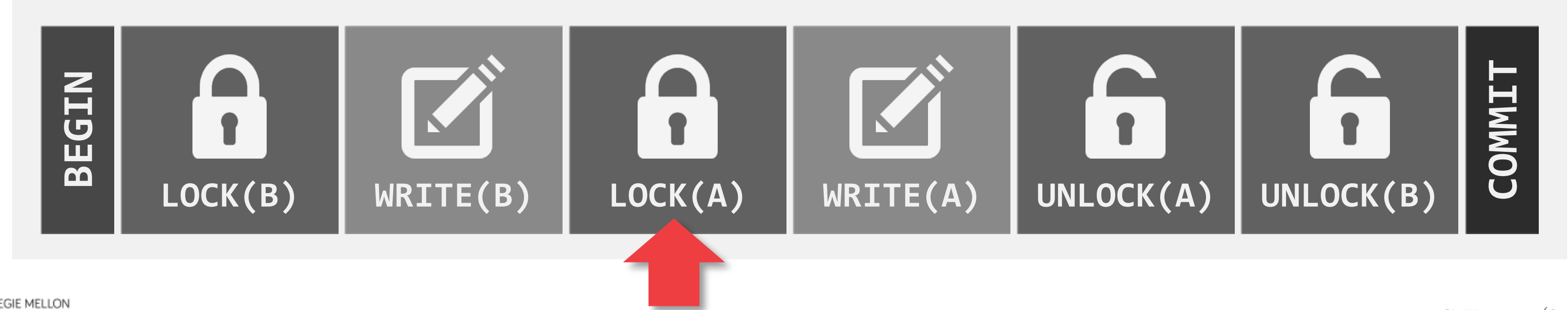


TWO-PHASE LOCKING

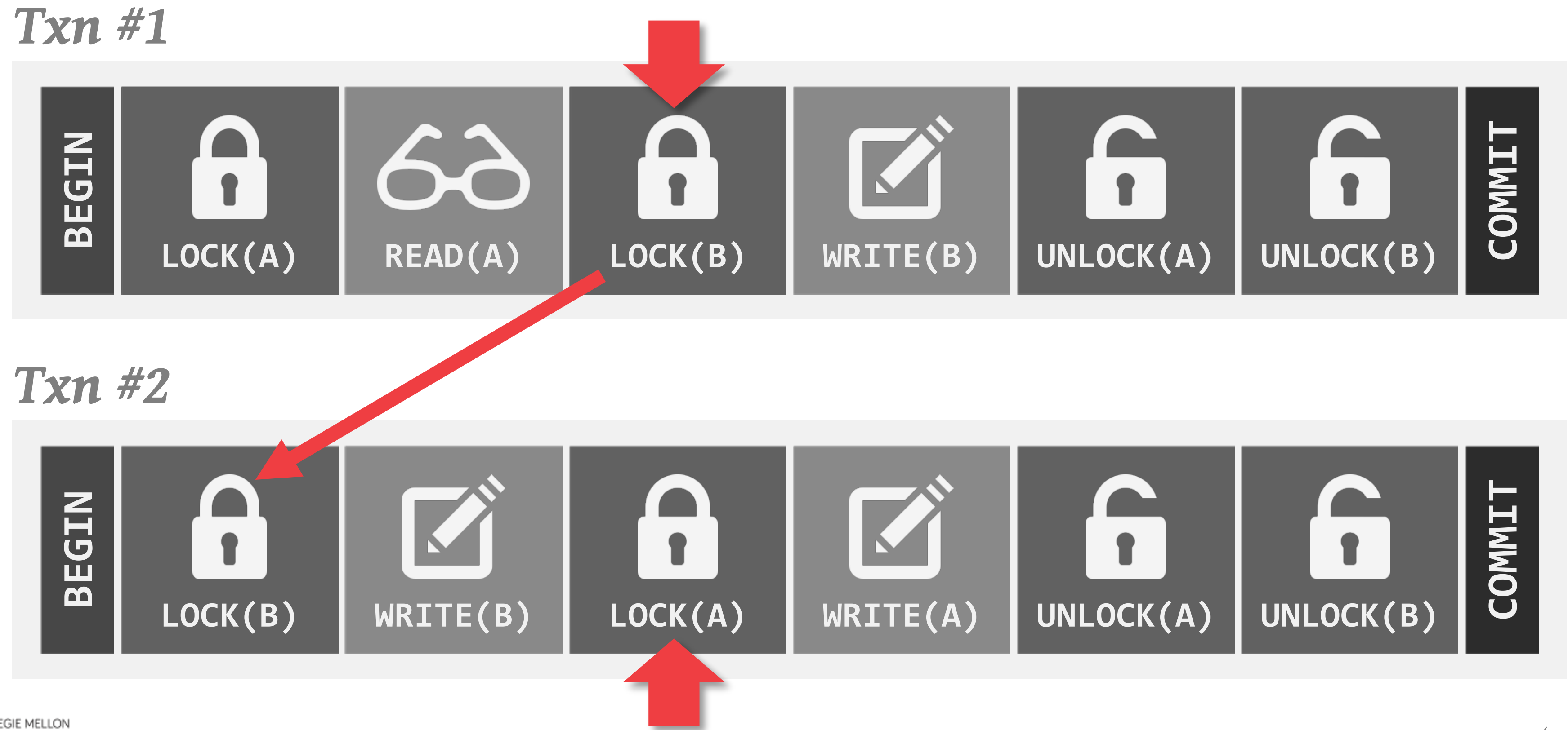
Txn #1



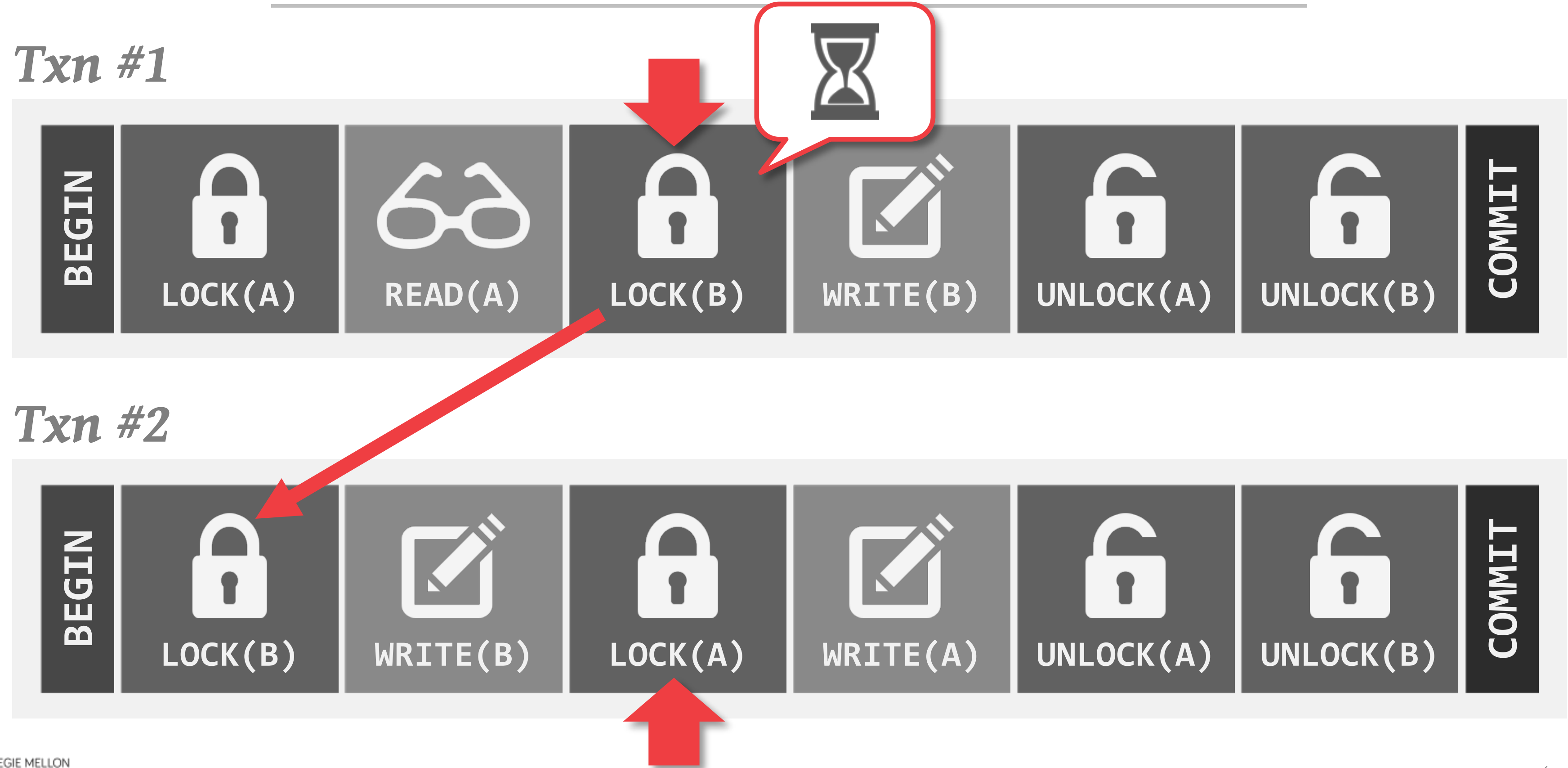
Txn #2



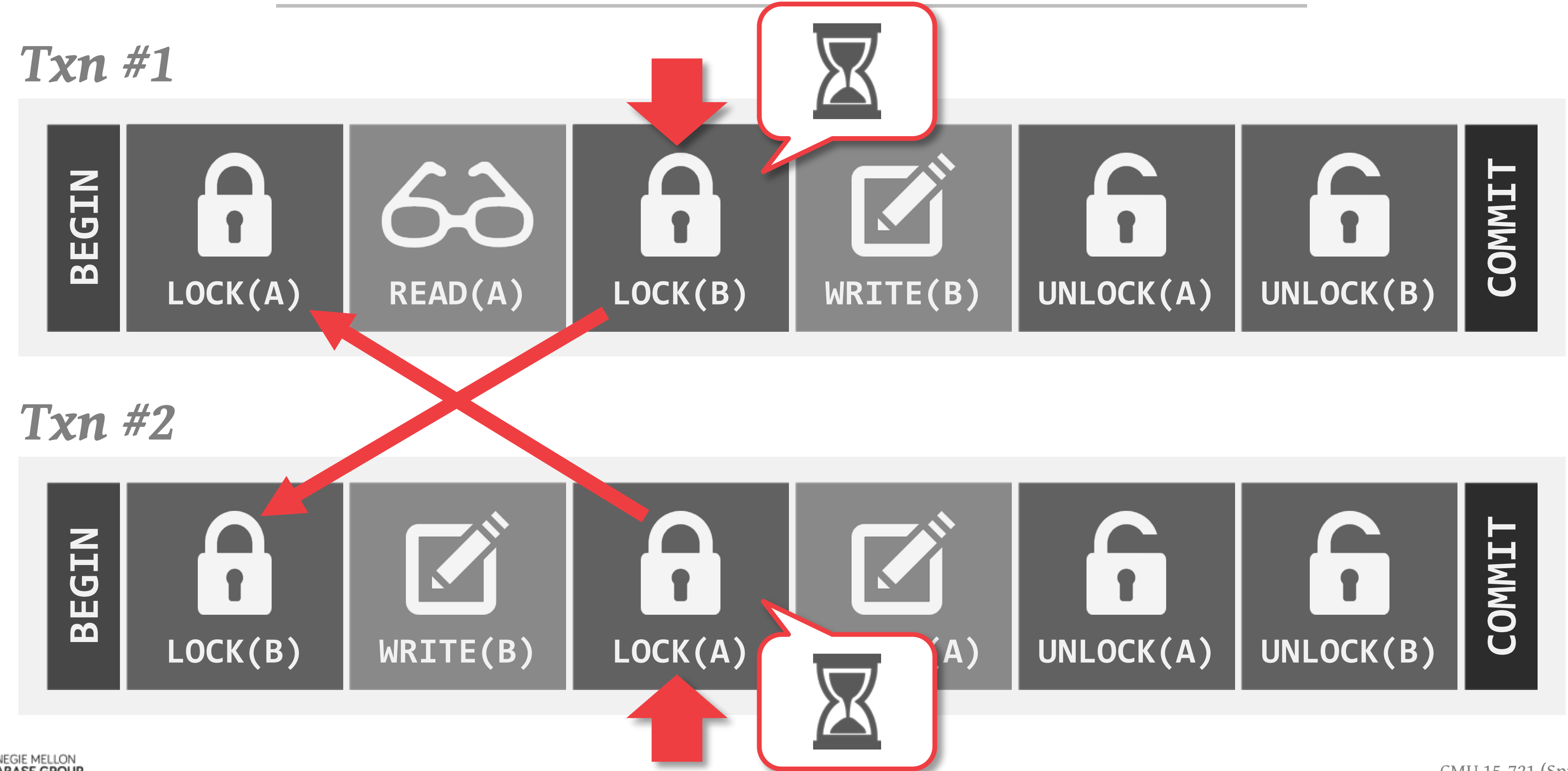
TWO-PHASE LOCKING



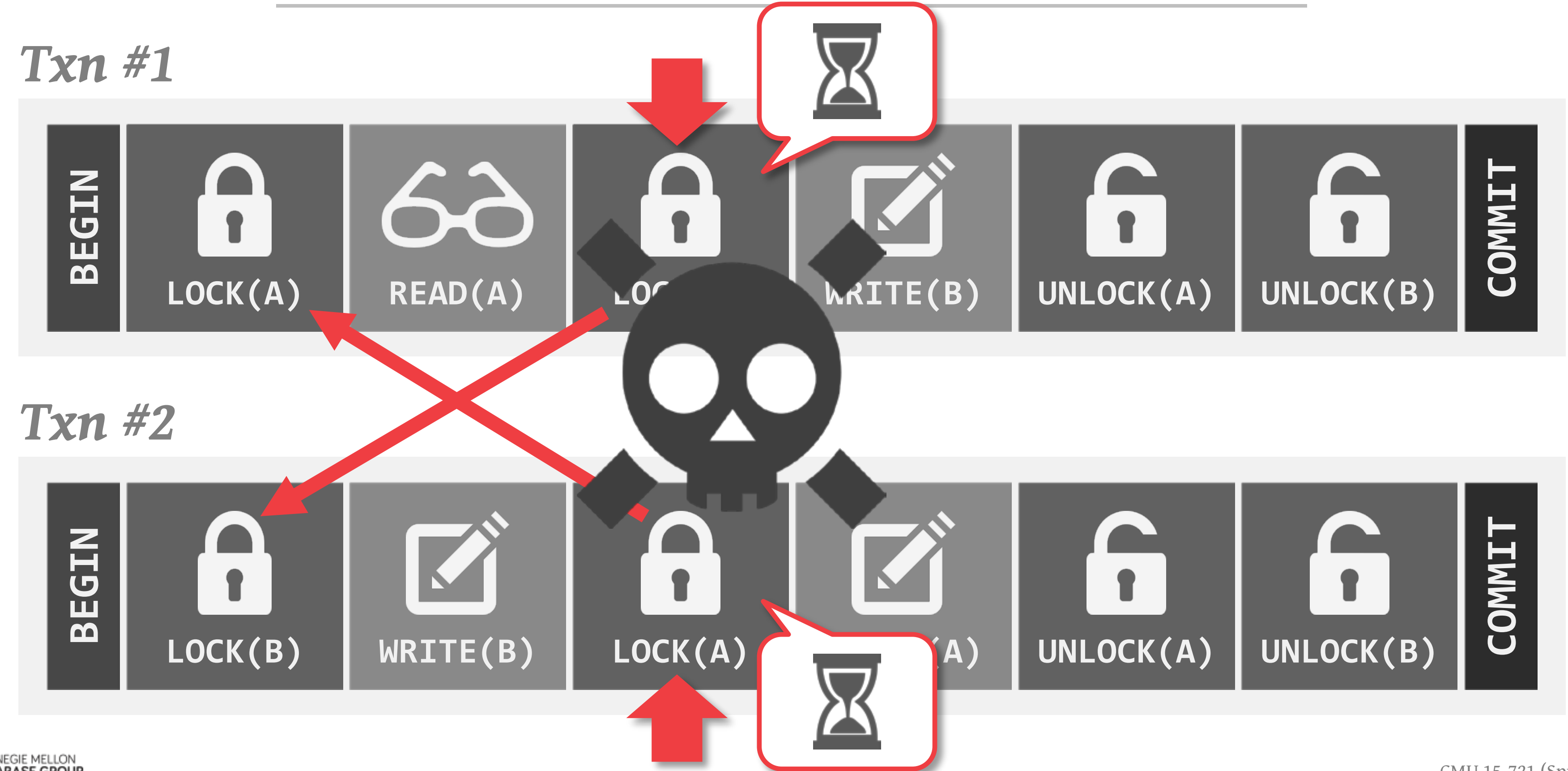
TWO-PHASE LOCKING



TWO-PHASE LOCKING



TWO-PHASE LOCKING



TWO-PHASE LOCKING

Deadlock Detection

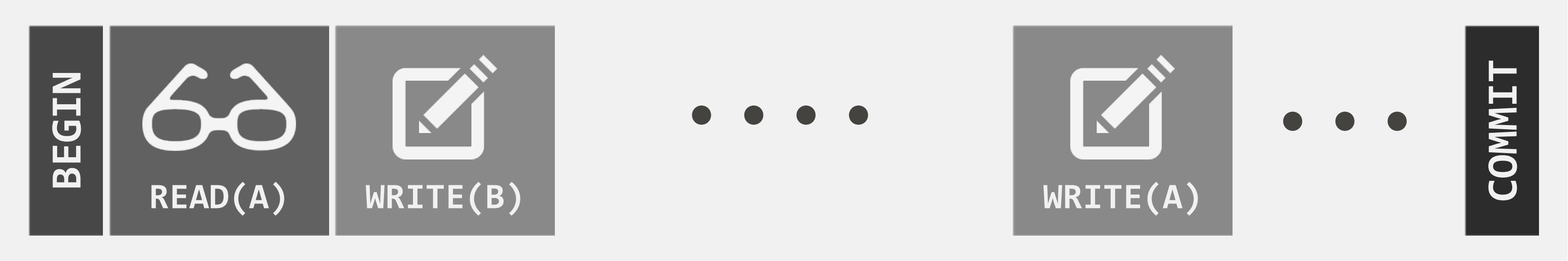
→ Each txn maintains a queue of the txns that hold the locks that it waiting for.
→ A separate thread checks these queues for deadlocks.
→ If deadlock found, use a heuristic to decide what txn to kill in order to break deadlock.

Deadlock Prevention

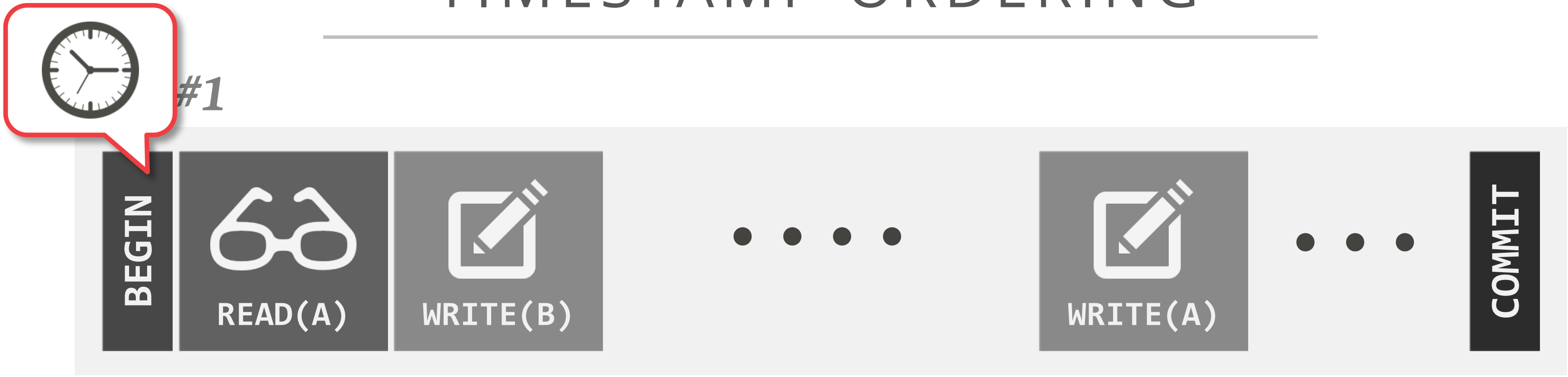
→ Check whether another txn already holds a lock when another txn requests it.
→ If lock is not available, the txn will either (1) wait, (2) commit suicide, or (3) kill the other txn.

TIMESTAMP ORDERING

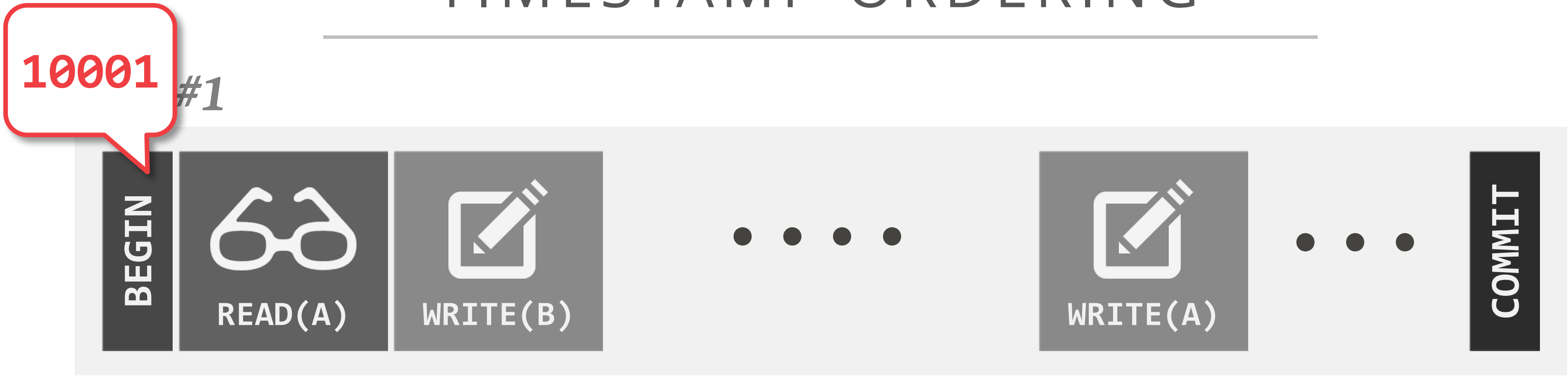
Txn #1



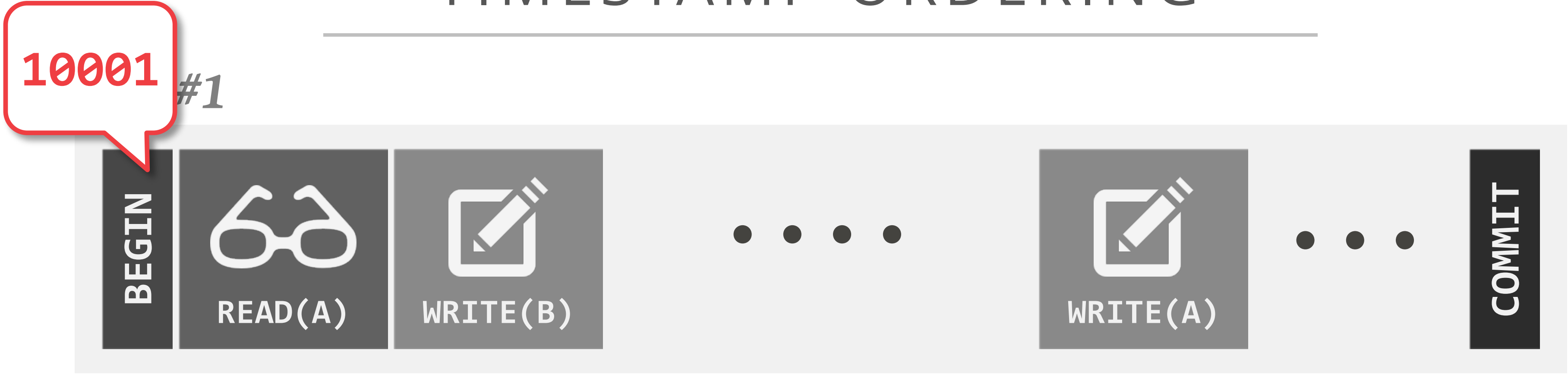
TIMESTAMP ORDERING



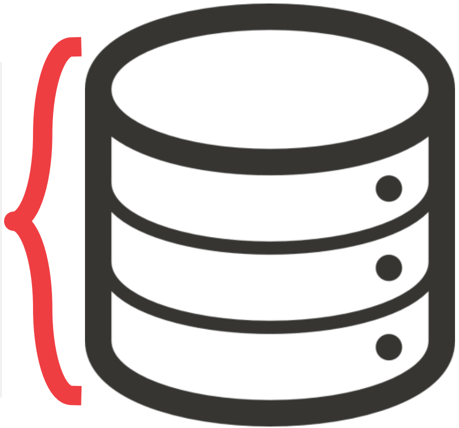
TIMESTAMP ORDERING



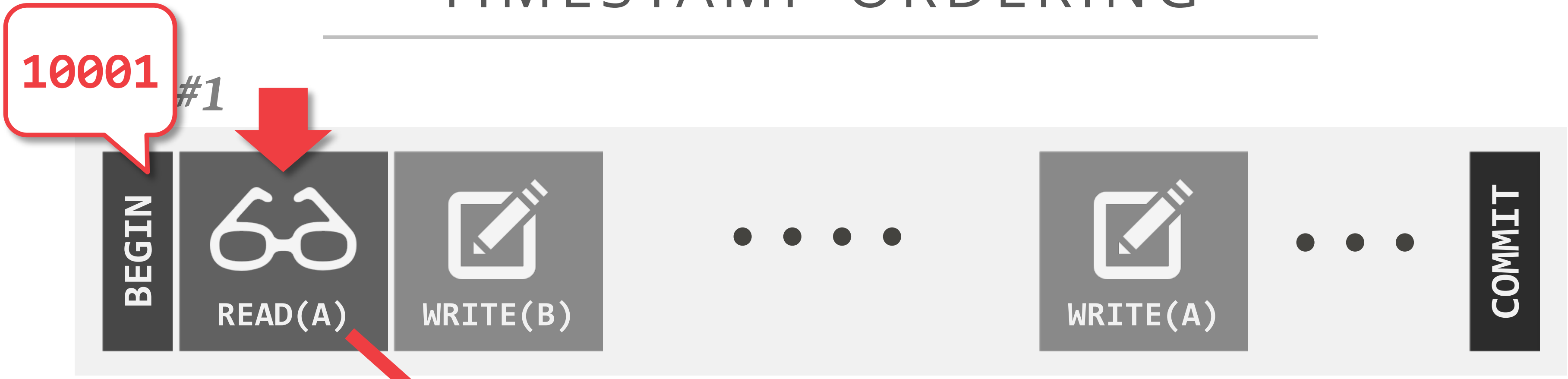
TIMESTAMP ORDERING



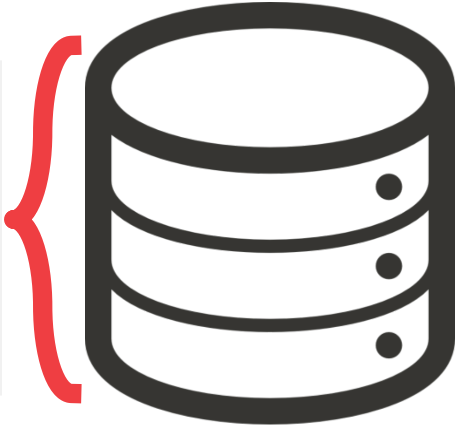
Record	Read Timestamp	Write Timestamp
A	10000	10000
B	10000	10000



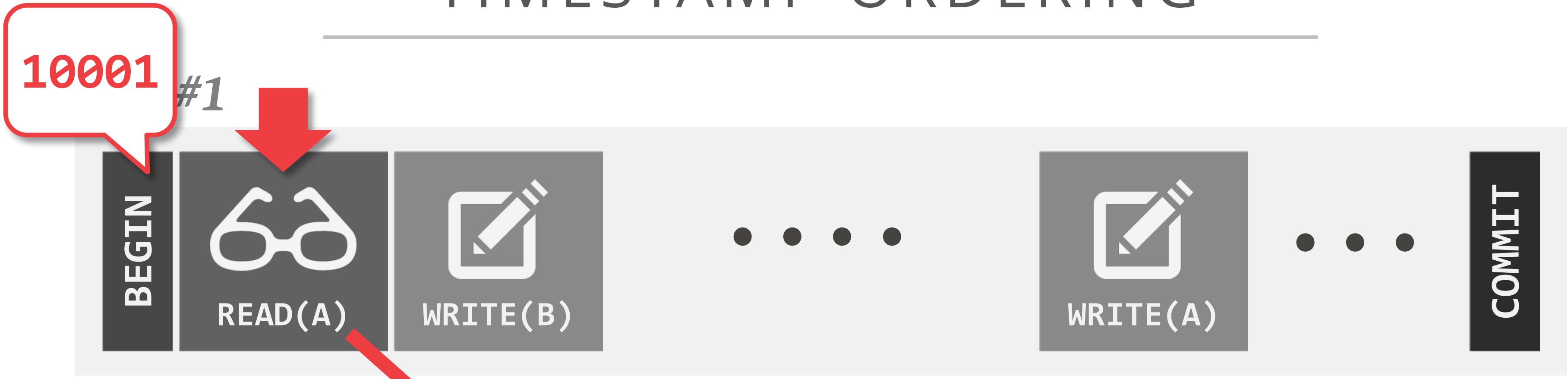
TIMESTAMP ORDERING



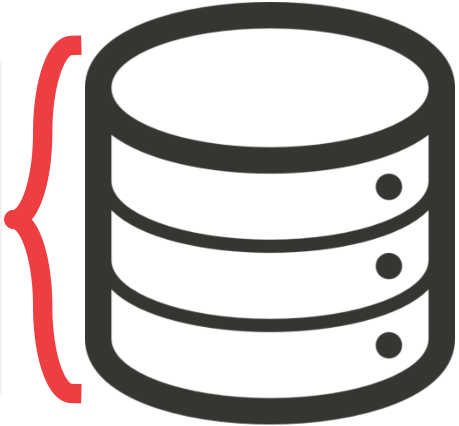
Record	Read Timestamp	Write Timestamp
A	10000	10000
B	10000	10000



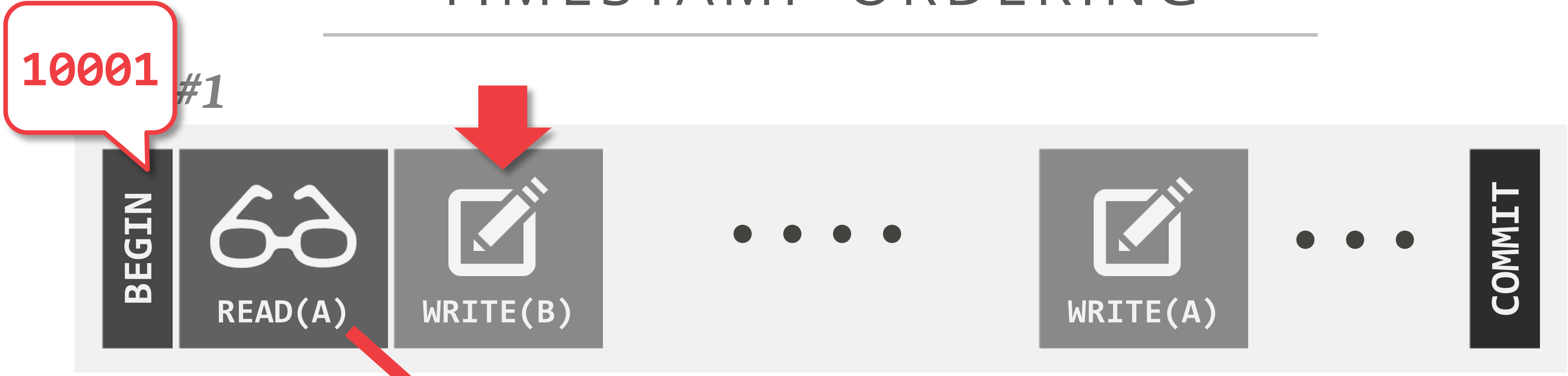
TIMESTAMP ORDERING



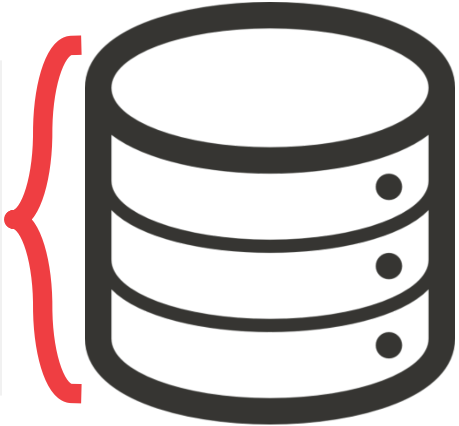
Record	Read Timestamp	Write Timestamp
A	10001	10000
B	10000	10000



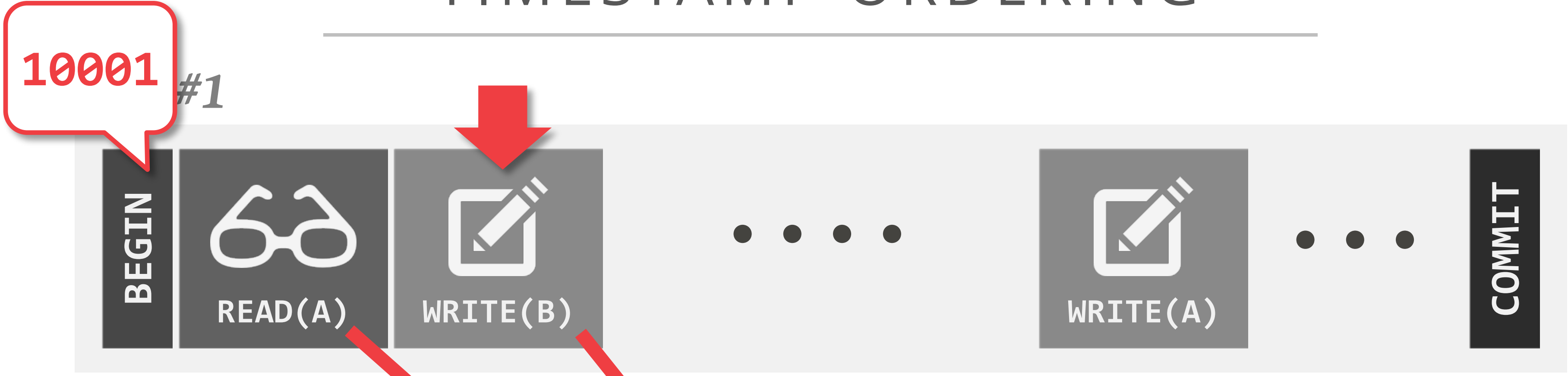
TIMESTAMP ORDERING



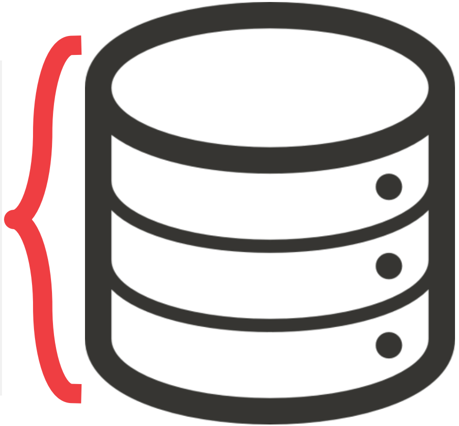
Record	Read Timestamp	Write Timestamp
A	10001	10000
B	10000	10000



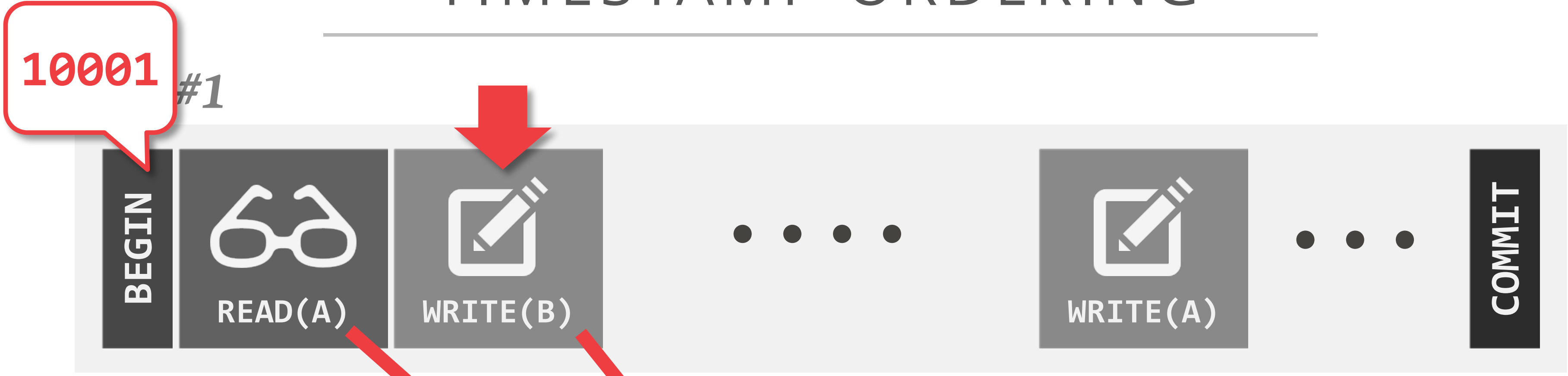
TIMESTAMP ORDERING



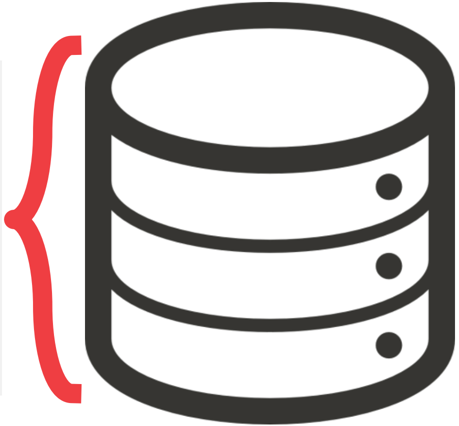
Record	Read Timestamp	Write Timestamp
A	10001	10000
B	10000	10000



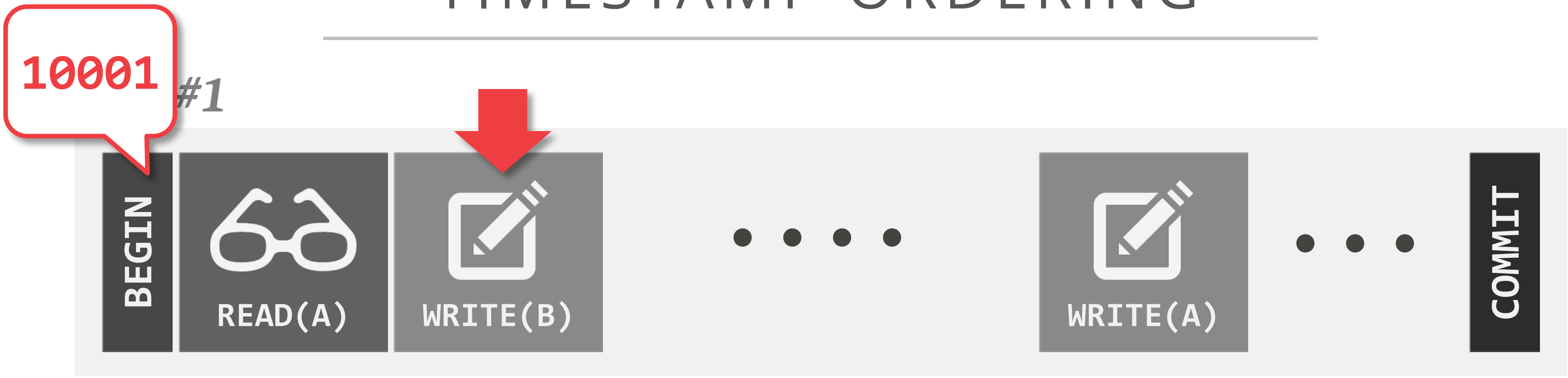
TIMESTAMP ORDERING



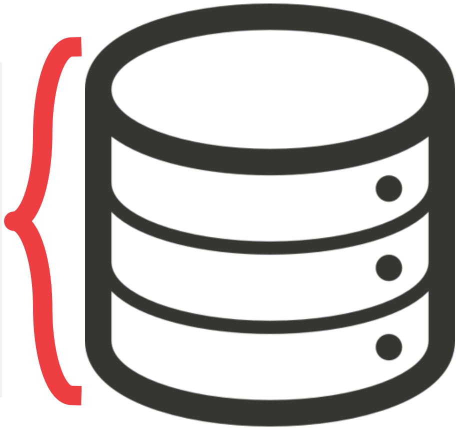
Record	Read Timestamp	Write Timestamp
A	10001	10000
B	10000	10001



TIMESTAMP ORDERING

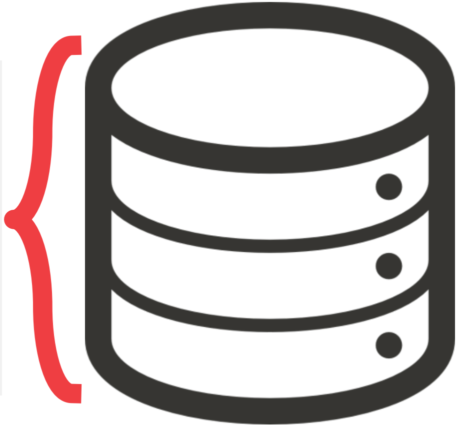


Record	Read Timestamp	Write Timestamp
A	10001	10000
B	10000	10001



TIMESTAMP ORDERING

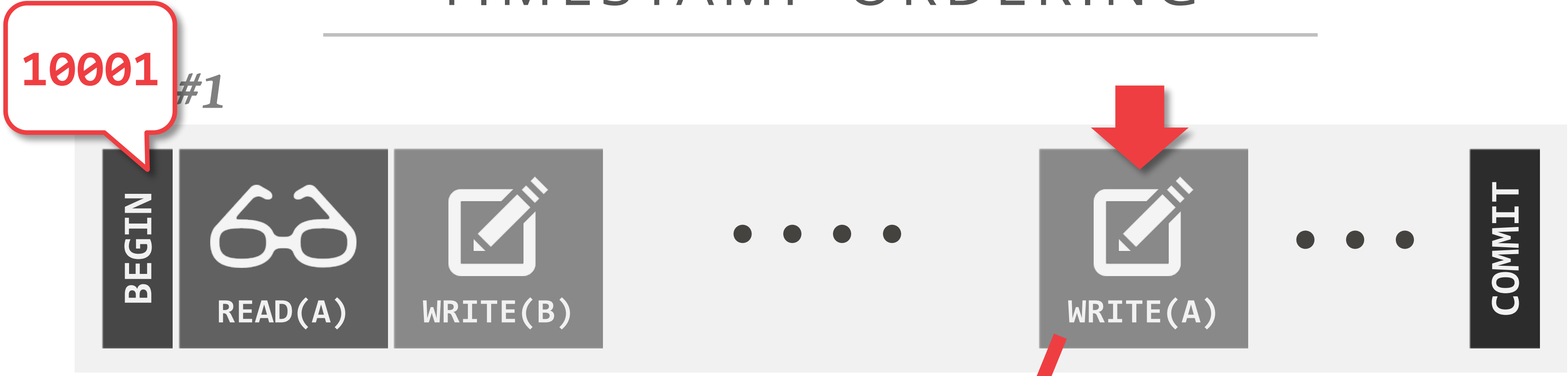
Record	Read Timestamp	Write Timestamp
A	10001	10000
B	10000	10001



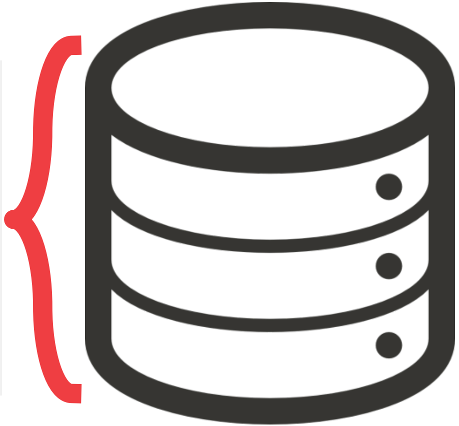
TIMESTAMP ORDERING

Record	Read Timestamp	Write Timestamp
A	10001	10005
B	10000	10001

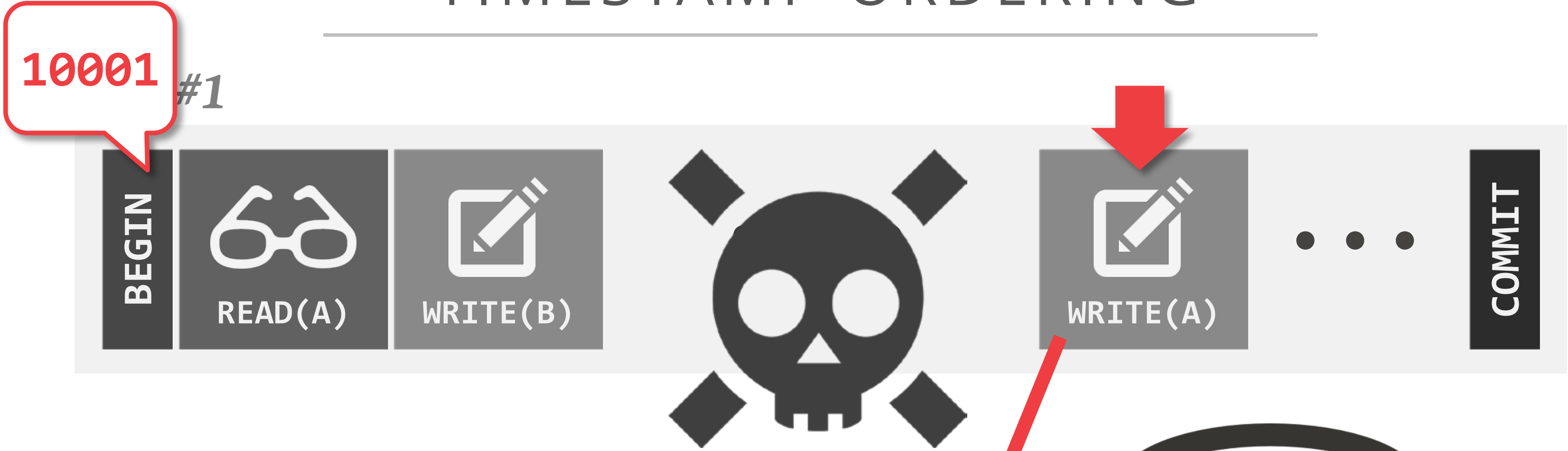
TIMESTAMP ORDERING



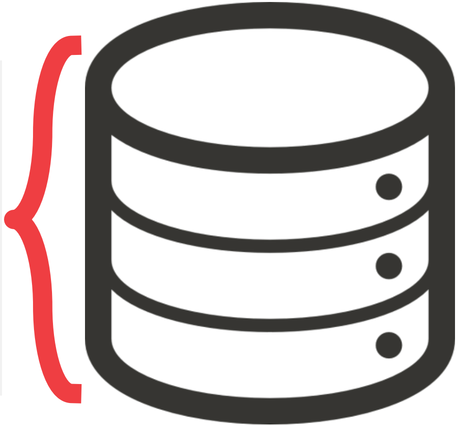
Record	Read Timestamp	Write Timestamp
A	10001	10005
B	10000	10001



TIMESTAMP ORDERING



Record	Read Timestamp	Write Timestamp
A	10001	10005
B	10000	10001



TIMESTAMP ORDERING

Basic T/O

→ Check for conflicts on each read/write.

→ Copy tuples on each access to ensure repeatable reads.

Multi-Version Concurrency Control (MVCC)

→ Create a new version of a tuple whenever a txn modifies it. Use timestamps as version id.

→ Check visibility on every read/write.

Optimistic Currency Control (OCC)

→ Store all changes in private workspace.

→ Check for conflicts at commit time and then merge.

CONCURRENCY CONTROL SCHEMES

DL_DETECT	2PL w/ Deadlock Detection
NO_WAIT	2PL w/ Non-waiting Prevention
WAIT_DIE	2PL w/ Wait-and-Die Prevention
TIMESTAMP	Basic T/O Algorithm
MVCC	Multi-Version T/O
OCC	Optimistic Concurrency Control

CONCURRENCY CONTROL SCHEMES

DL_DETECT	2PL w/ Deadlock Detection
NO_WAIT	2PL w/ Non-waiting Prevention
WAIT_DIE	2PL w/ Wait-and-Die Prevention
TIMESTAMP	Basic T/O Algorithm
MVCC	Multi-Version T/O
OCC	Optimistic Concurrency Control

CONCURRENCY CONTROL SCHEMES

DL_DETECT	2PL w/ Deadlock Detection
NO_WAIT	2PL w/ Non-waiting Prevention
WAIT_DIE	2PL w/ Wait-and-Die Prevention
TIMESTAMP	Basic T/O Algorithm
MVCC	Multi-Version T/O
OCC	Optimistic Concurrency Control

CONCURRENCY CONTROL SCHEMES

DL_DETECT	2PL w/ Deadlock Detection
NO_WAIT	2PL w/ Non-waiting Prevention
WAIT_DIE	2PL w/ Wait-and-Die Prevention
TIMESTAMP	Basic T/O Algorithm
MVCC	Multi-Version T/O
OCC	Optimistic Concurrency Control

CONCURRENCY CONTROL SCHEMES

DL_DETECT	2PL w/ Deadlock Detection
NO_WAIT	2PL w/ Non-waiting Prevention
WAIT_DIE	2PL w/ Wait-and-Die Prevention
TIMESTAMP	Basic T/O Algorithm
MVCC	Multi-Version T/O
OCC	Optimistic Concurrency Control

CONCURRENCY CONTROL SCHEMES

DL_DETECT	2PL w/ Deadlock Detection
NO_WAIT	2PL w/ Non-waiting Prevention
WAIT_DIE	2PL w/ Wait-and-Die Prevention
TIMESTAMP	Basic T/O Algorithm
MVCC	Multi-Version T/O
OCC	Optimistic Concurrency Control

CONCURRENCY CONTROL SCHEMES

DL_DETECT	2PL w/ Deadlock Detection
NO_WAIT	2PL w/ Non-waiting Prevention
WAIT_DIE	2PL w/ Wait-and-Die Prevention
TIMESTAMP	Basic T/O Algorithm
MVCC	Multi-Version T/O
OCC	Optimistic Concurrency Control

1000-CORE CPU SIMULATOR

DBx1000 Database System

→ In-memory DBMS with pluggable lock manager.
→ No network access, logging, or concurrent indexes

MIT Graphite CPU Simulator

→ Single-socket, tile-based CPU.
→ Shared L2 cache for groups of cores.
→ Tiles communicate over 2D-mesh network.

STARING INTO THE ABYSS: AN EVALUATION OF CONCURRENCY CONTROL WITH ONE THOUSAND CORES
VLDB, pp. 209-220, 2014.

TARGET WORKLOAD

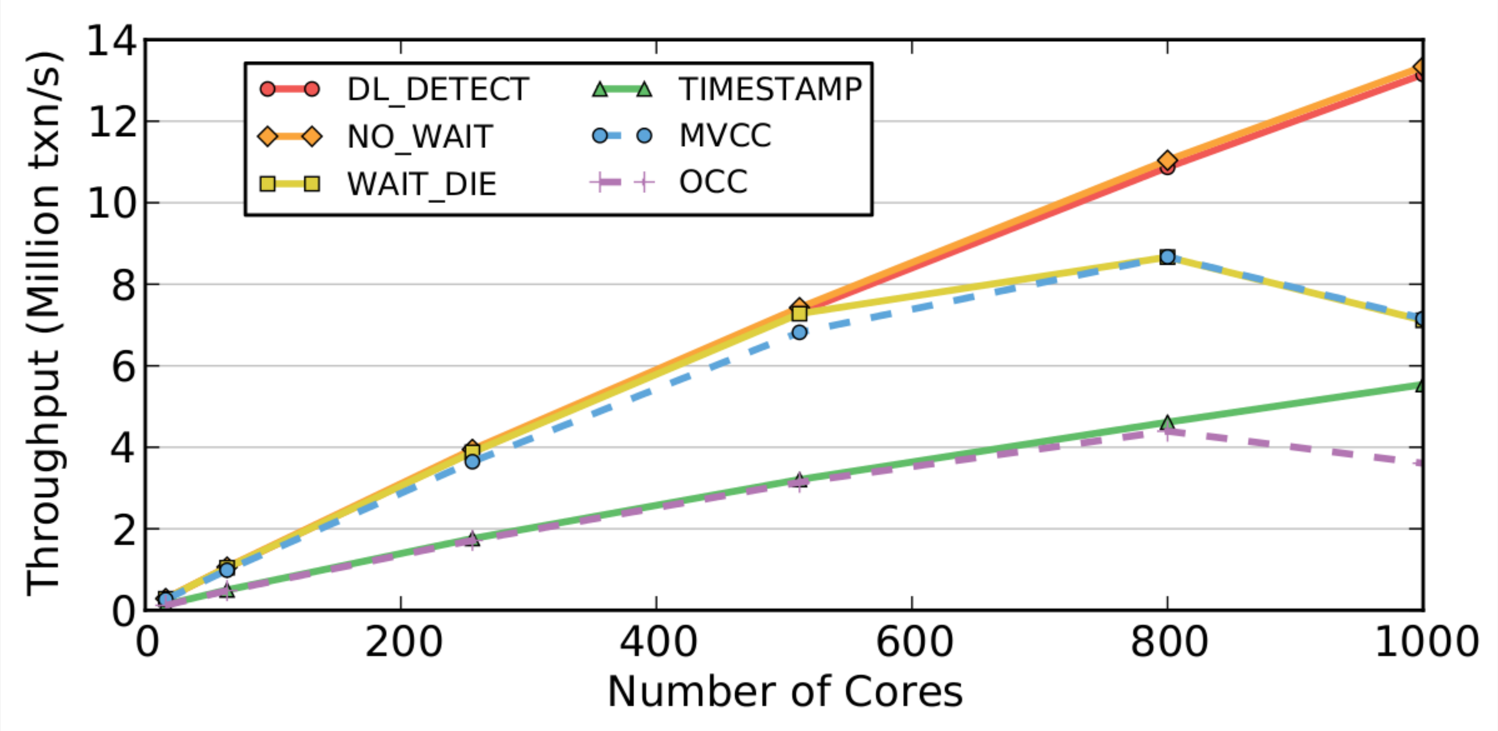
Yahoo! Cloud Serving Benchmark (YCSB)
→ 20 million tuples
→ Each tuple is 1KB (total database is ~20GB)

Each transactions reads/modifies 16 tuples.

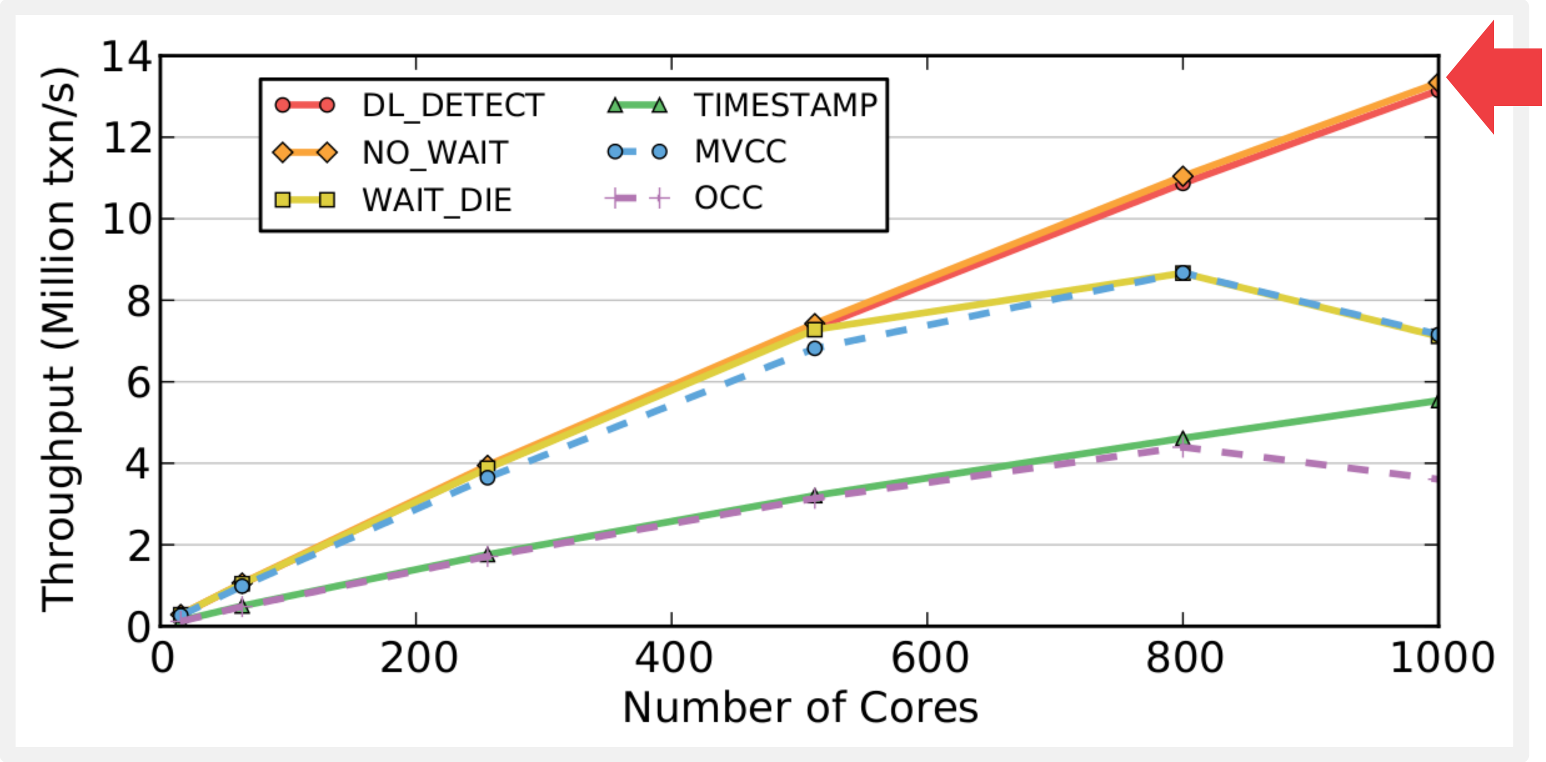
Varying skew in transaction access patterns.

Serializable isolation level.

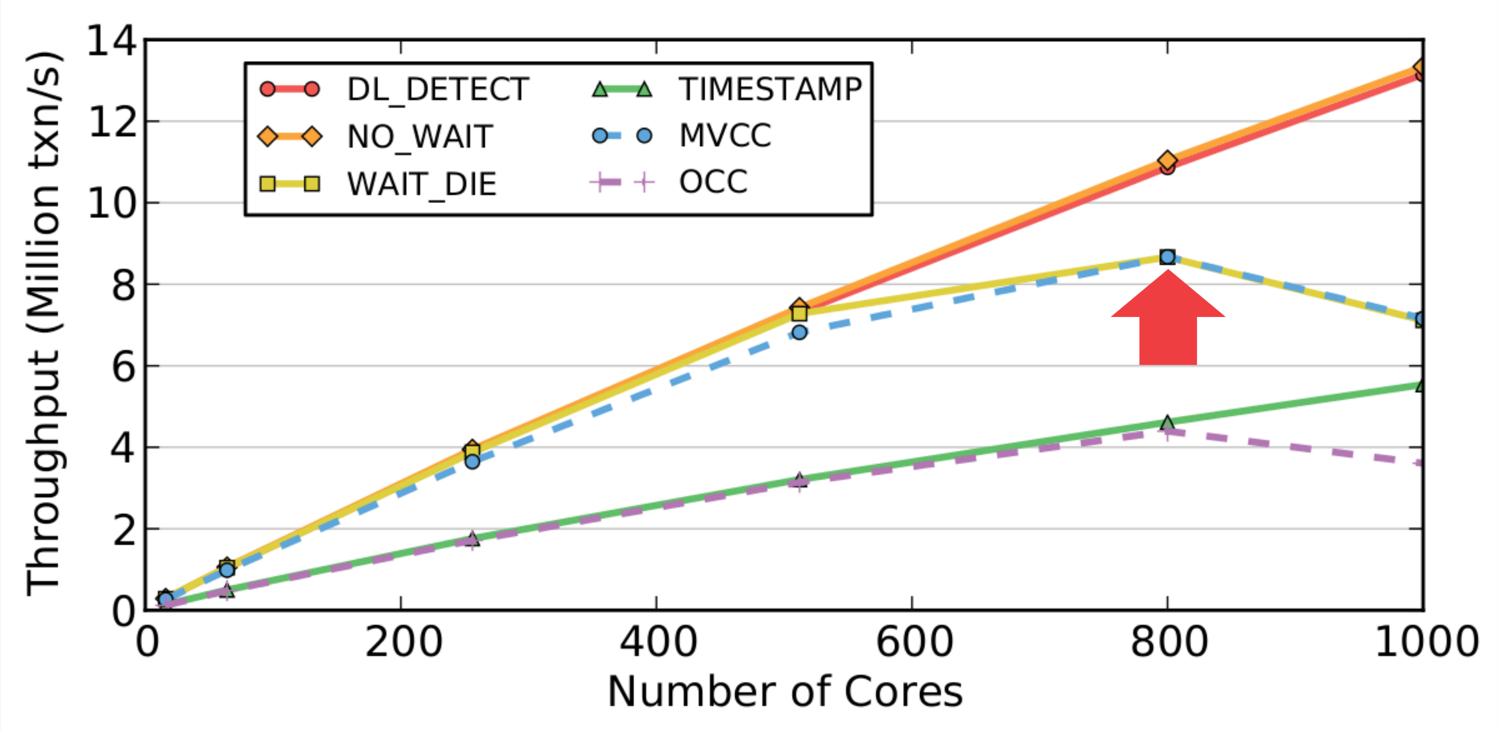
READ-ONLY WORKLOAD



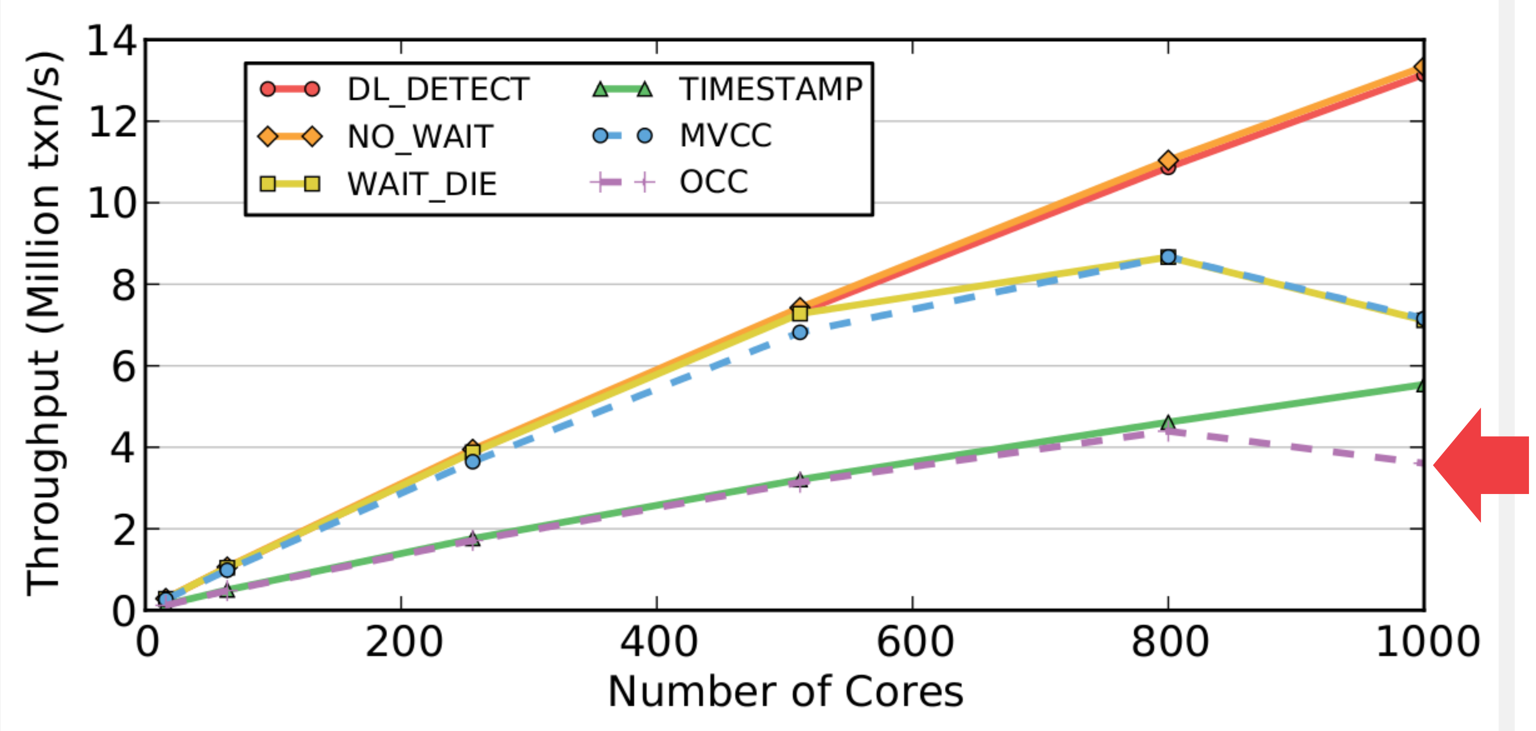
READ-ONLY WORKLOAD



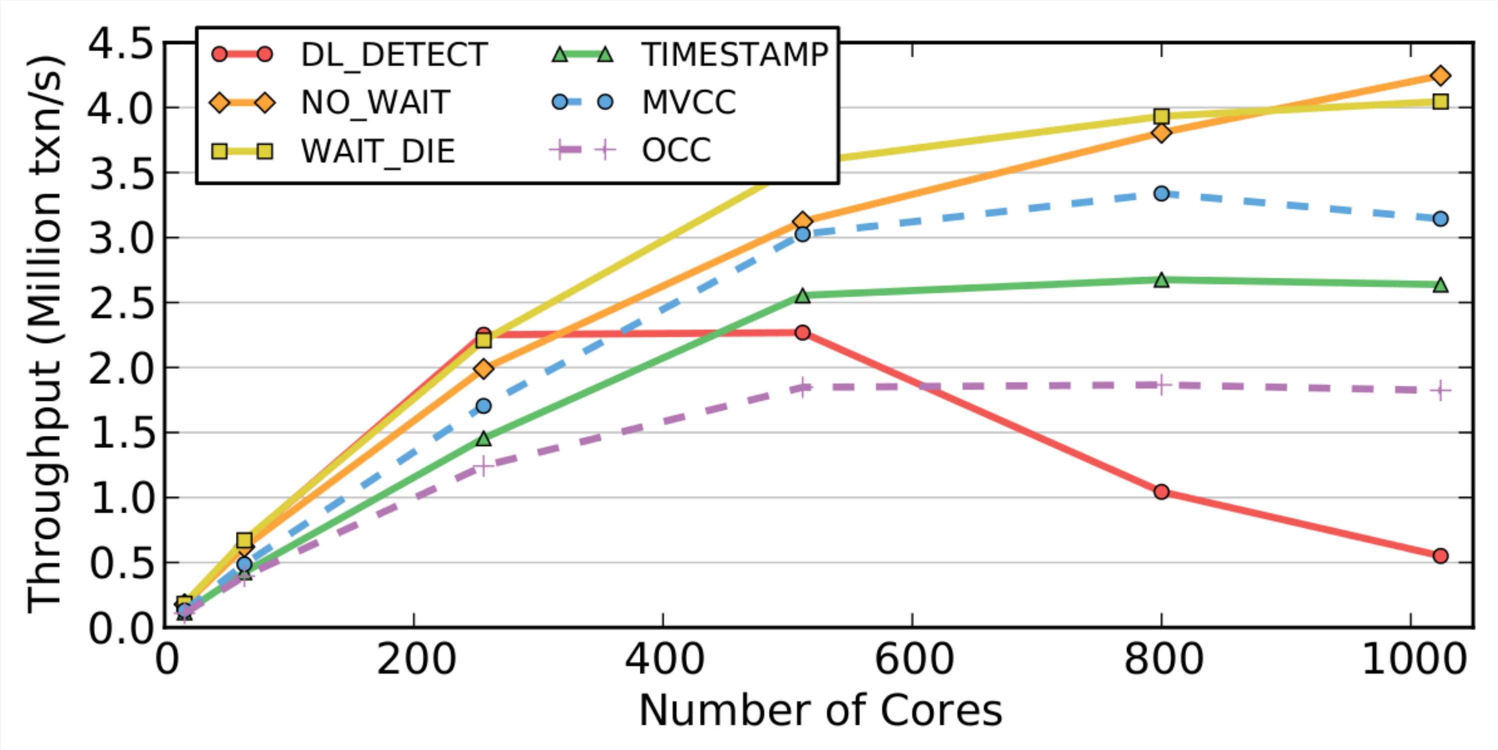
READ-ONLY WORKLOAD



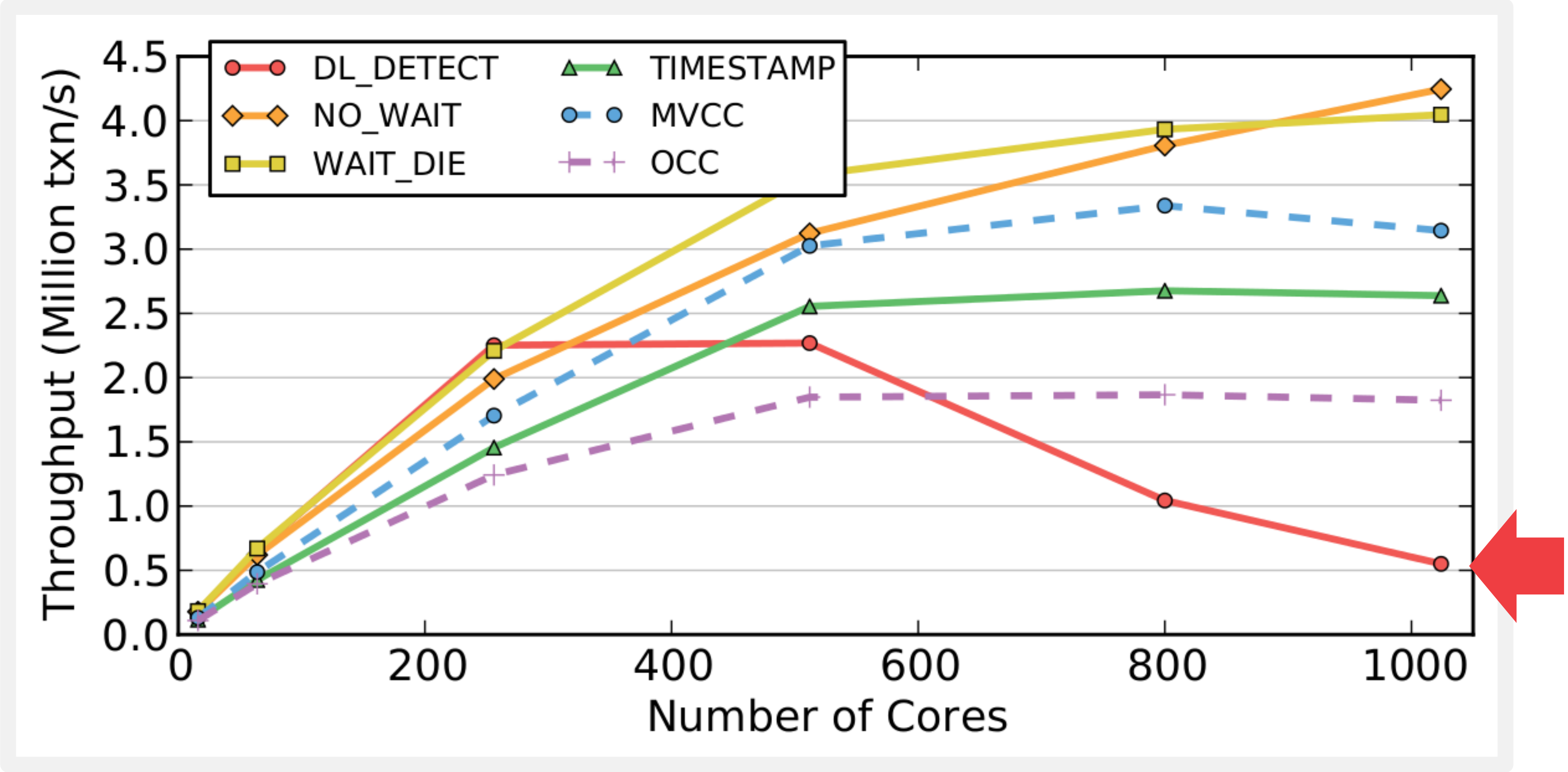
READ-ONLY WORKLOAD



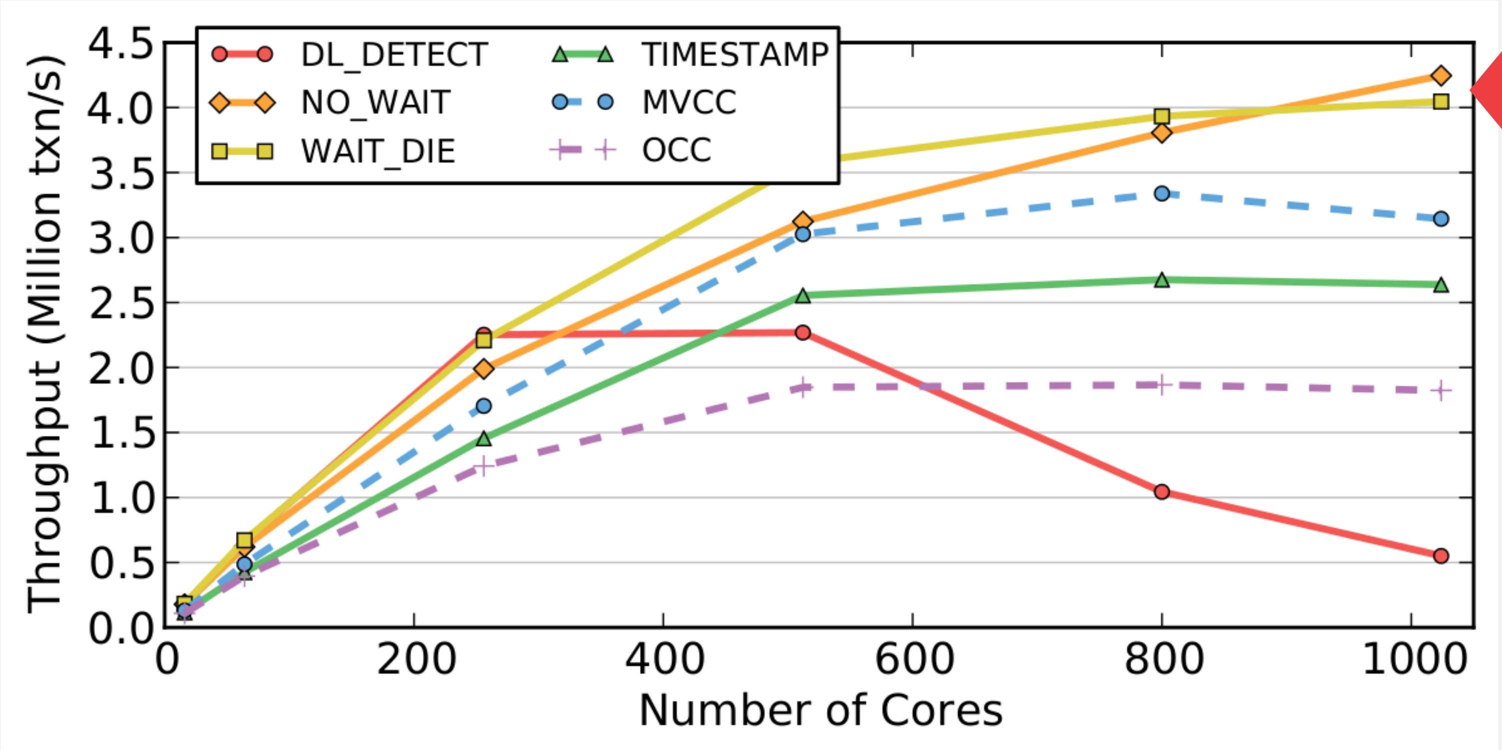
WRITE-INTENSIVE / MEDIUM-CONTENTION



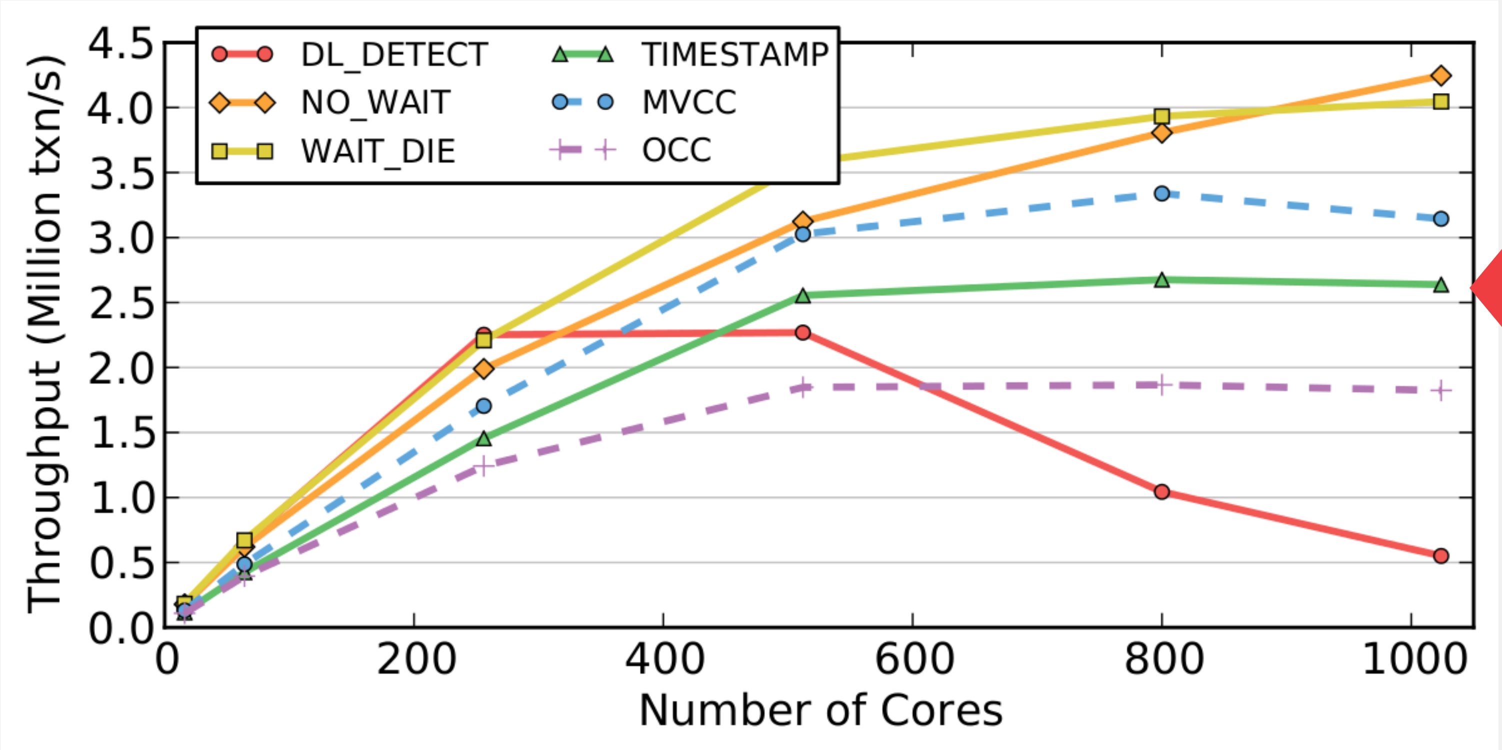
WRITE-INTENSIVE / MEDIUM-CONTENTION



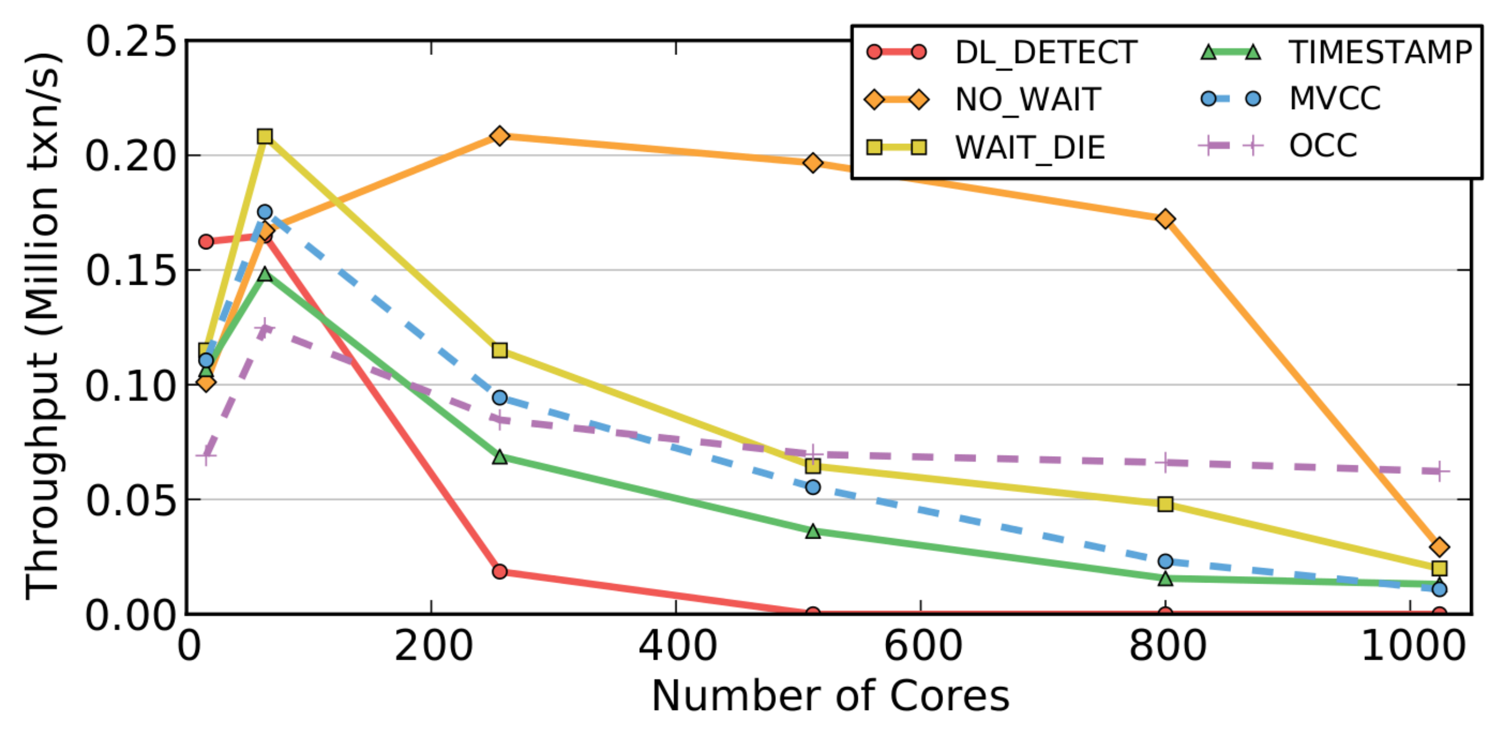
WRITE-INTENSIVE / MEDIUM-CONTENTION



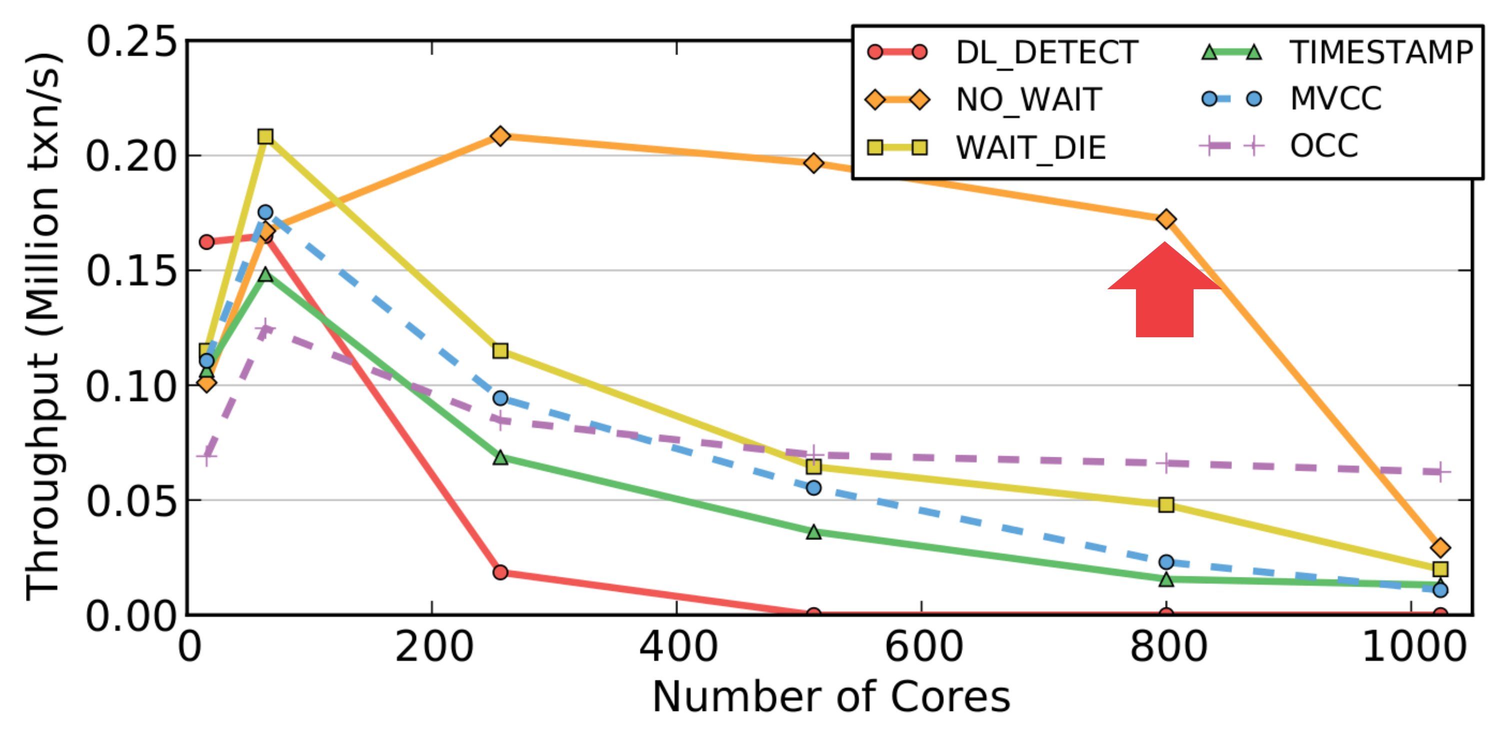
WRITE-INTENSIVE / MEDIUM-CONTENTION



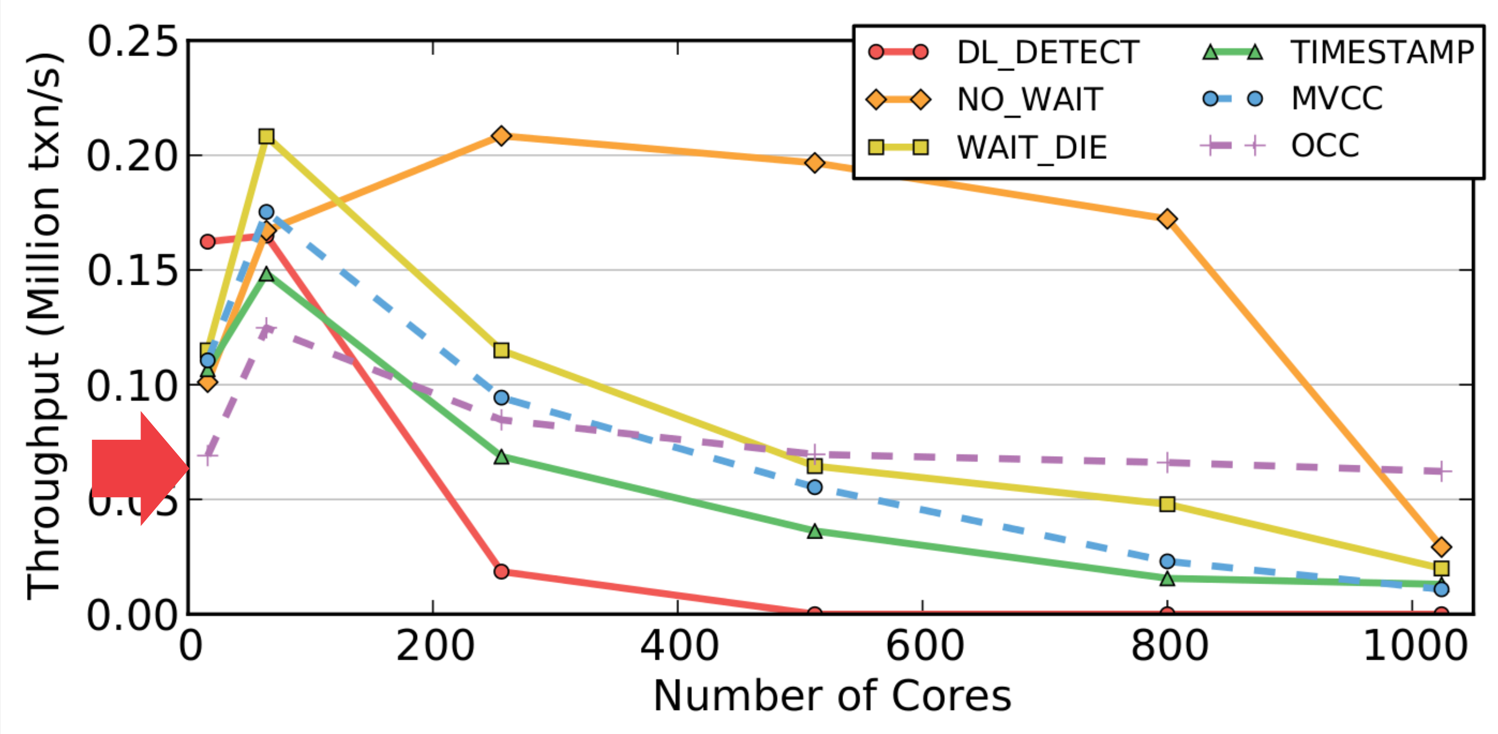
WRITE-INTENSIVE / HIGH-CONTENTION



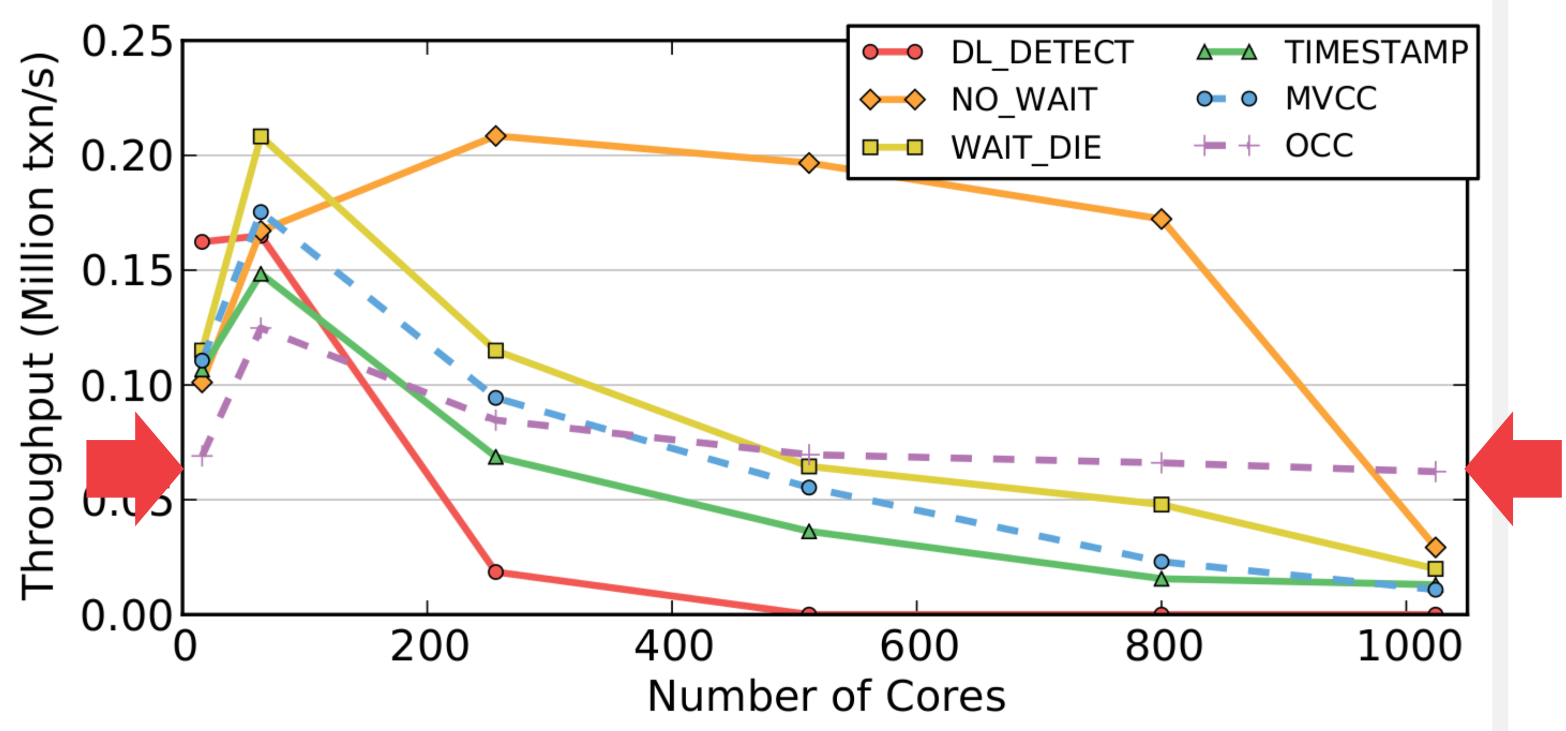
WRITE-INTENSIVE / HIGH-CONTENTION



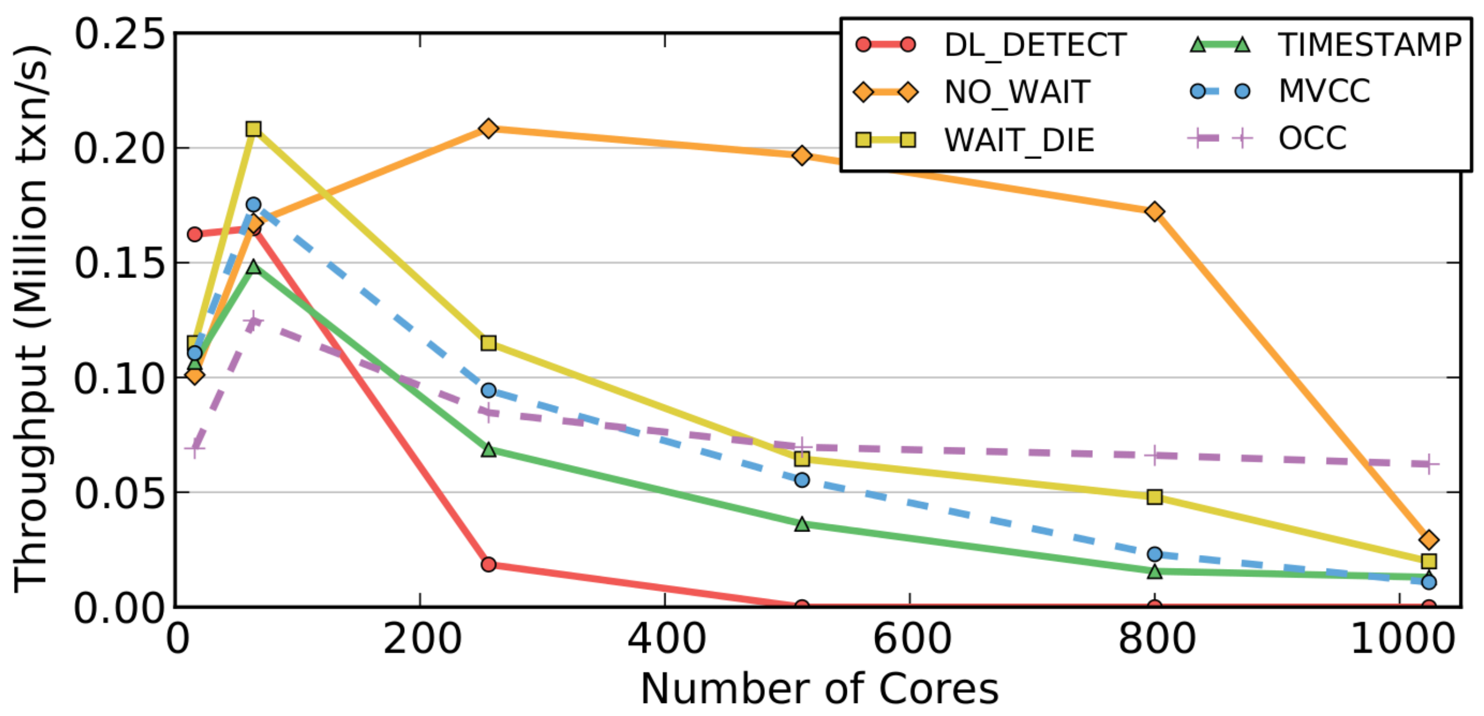
WRITE-INTENSIVE / HIGH-CONTENTION



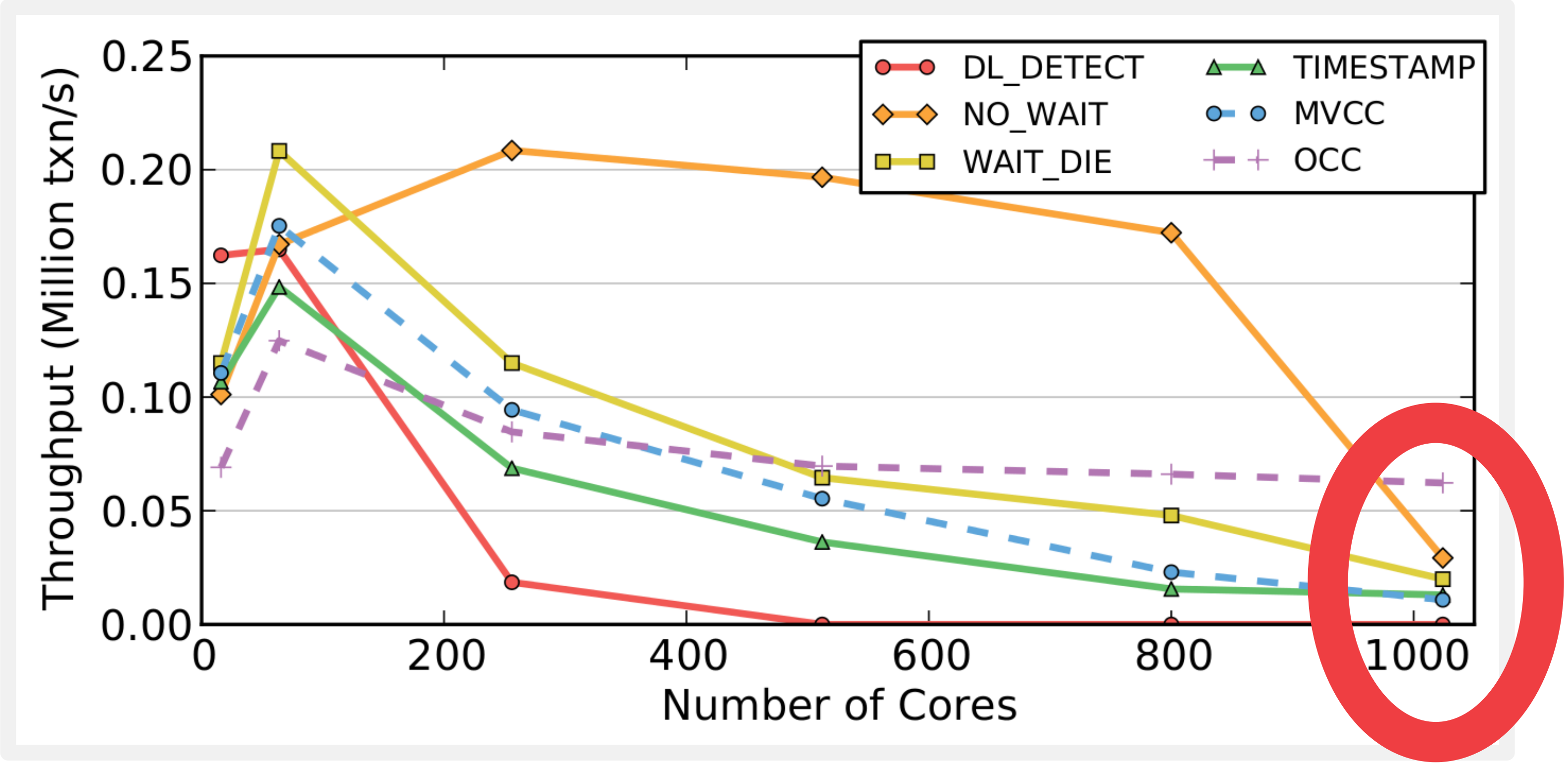
WRITE-INTENSIVE / HIGH-CONTENTION



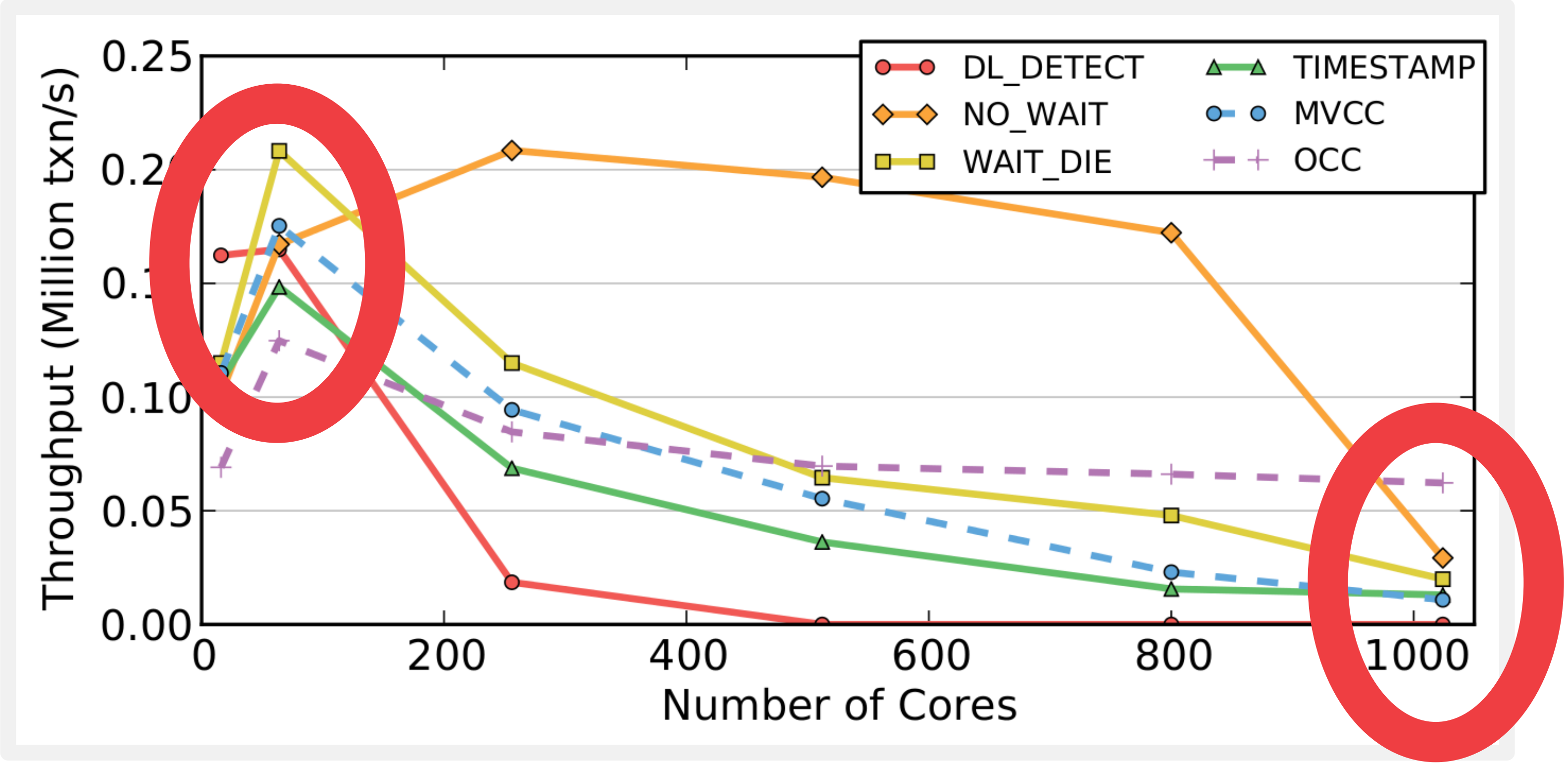
WRITE-INTENSIVE / HIGH-CONTENTION



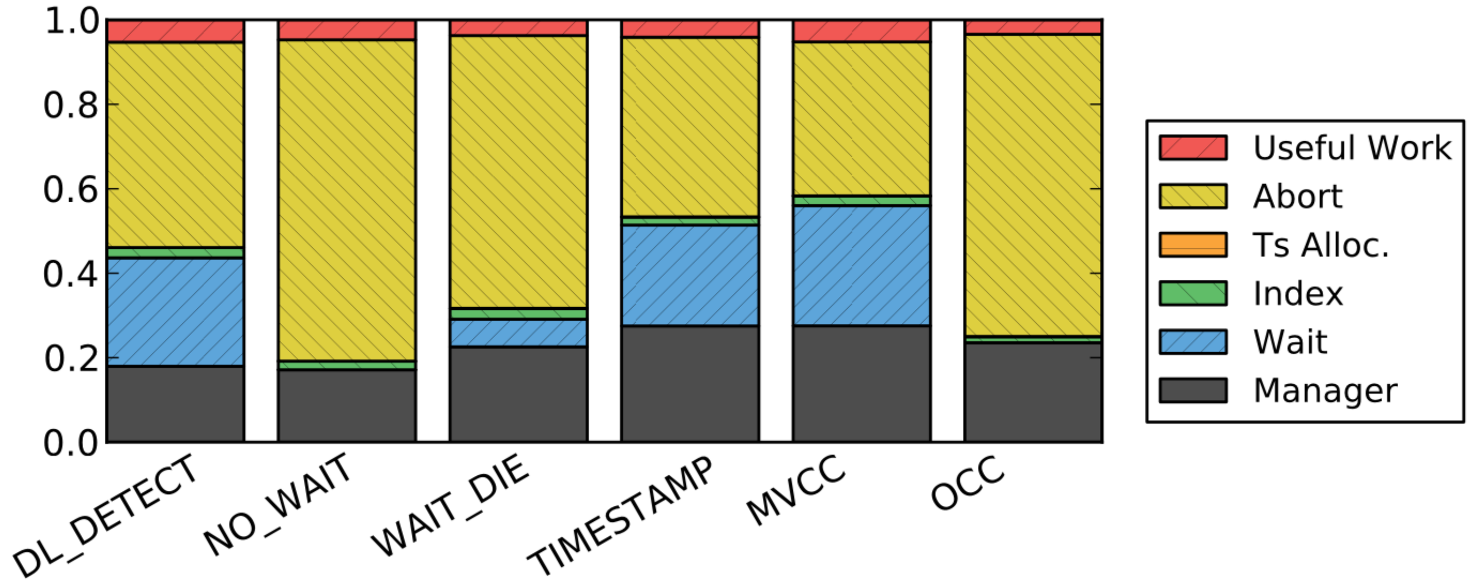
WRITE-INTENSIVE / HIGH-CONTENTION



WRITE-INTENSIVE / HIGH-CONTENTION



WRITE-INTENSIVE / HIGH-CONTENTION



BOTTLENECKS

Lock Thrashing
→ DL_DETECT, WAIT_DIE

Timestamp Allocation
→ All T/O algorithms + WAIT_DIE

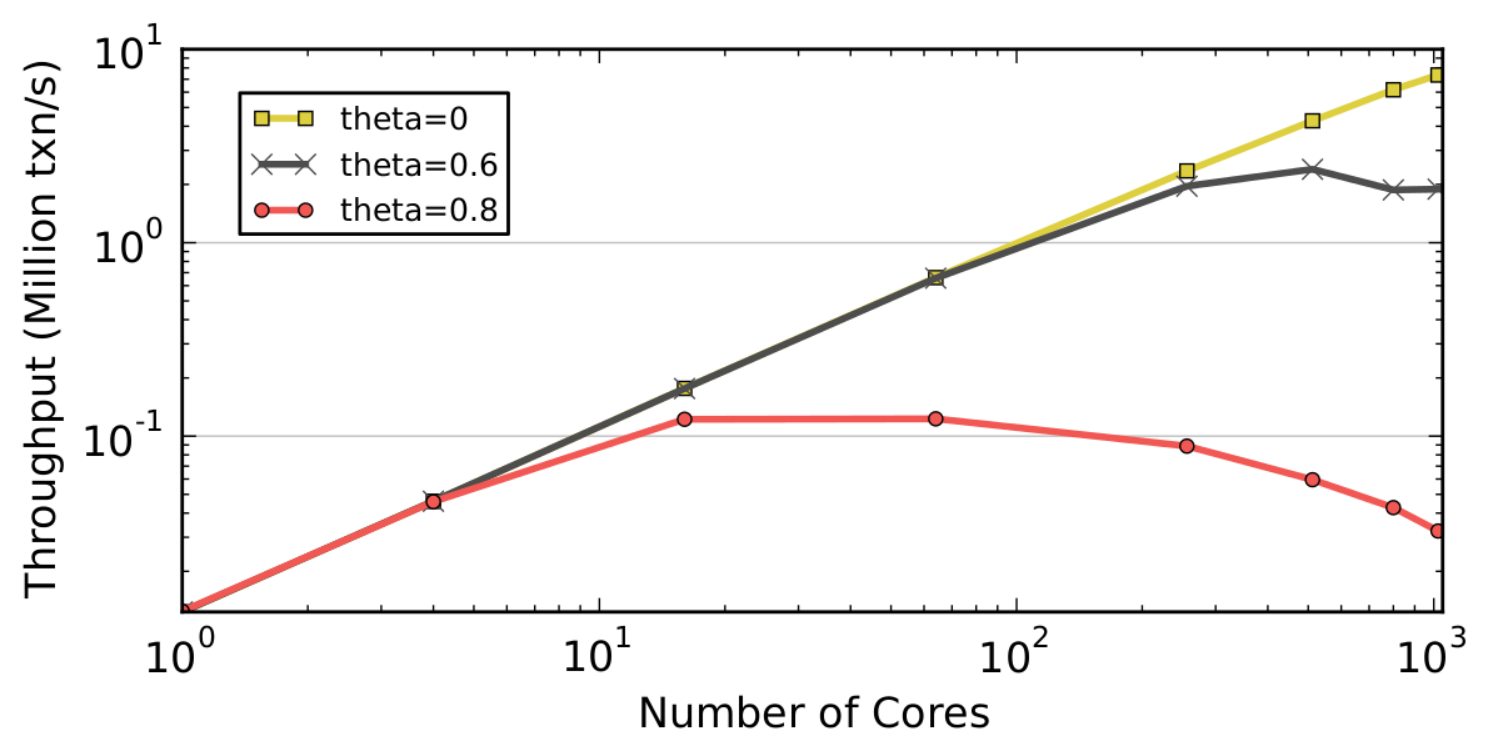
Memory Allocations
→ OCC + MVCC

LOCK THRASHING

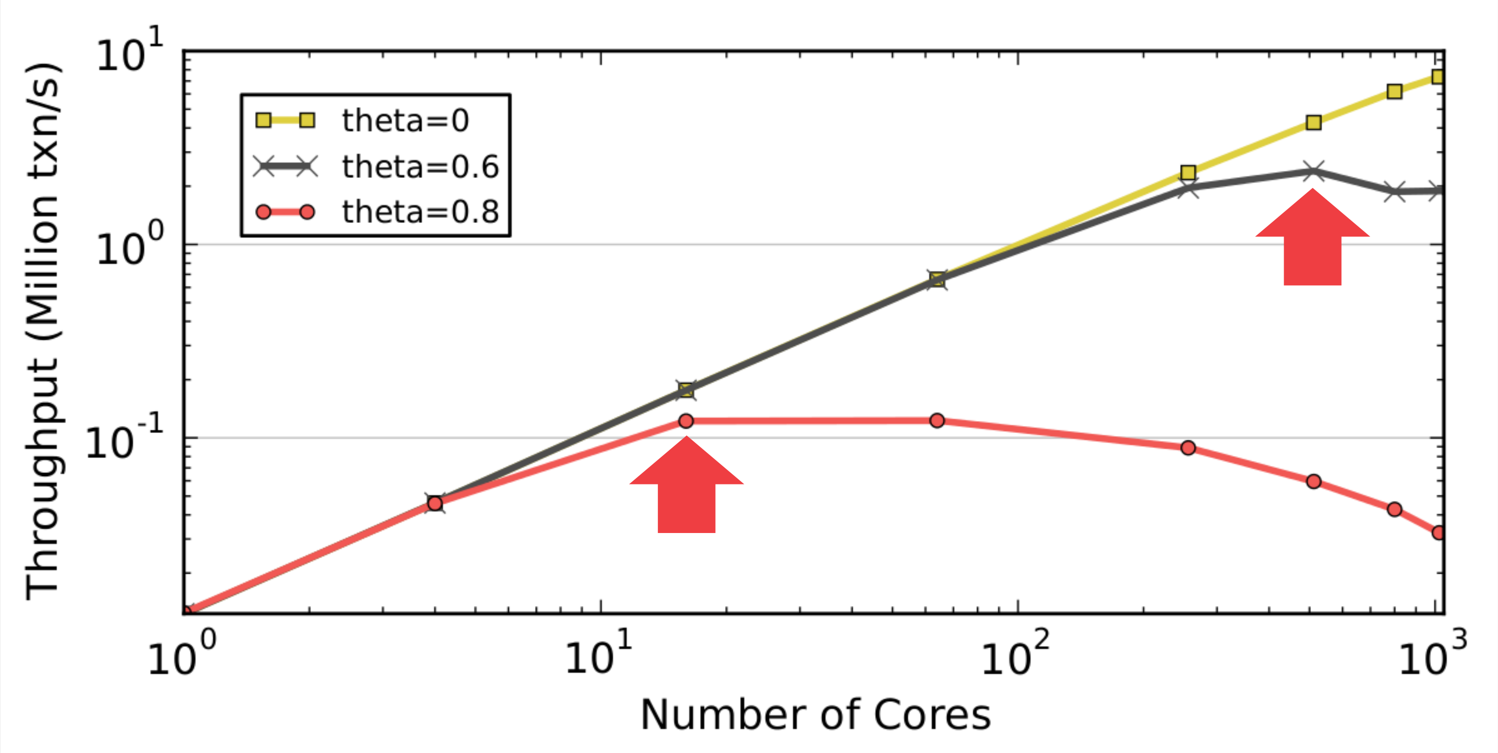
Each txn waits longer to acquire locks, causing other txn to wait longer to acquire locks.

Can measure this phenomenon by removing deadlock detection/prevention overhead.
→ Force txns to acquire locks in primary key order.
→ Deadlocks are not possible.

LOCK THRASHING



LOCK THRASHING



CK THRASHING

converts the update lock to a write lock. This lock conversion can't lead to a lock conversion deadlock, because at most one transaction can have an update lock on the data item. (Two transactions must try to convert the lock at the same time to create a lock conversion deadlock.) On the other hand, the benefit of this approach is that an update lock does not block other transactions that read without expecting to update later on. The weakness is that the request to convert the update lock to a write lock may be delayed by other read locks. If a large number of data items are read and only a few of them are updated, the tradeoff is worthwhile. This approach is used in Microsoft SQL Server. SQL Server also allows update locks to be obtained in a SELECT (i.e., read) statement, but in this case, it will not downgrade the update locks to read locks, since it doesn't know when it is safe to do so.

Lock Thrashing

By reducing the frequency of lock conversion deadlocks, we have dispensed with deadlock as a major performance consideration, so we are left with blocking situations. Blocking affects performance in a rather dramatic way. Until lock usage reaches a saturation point, it introduces only modest delays—significant, but not a serious problem. At some point, when too many transactions request locks, a large number of transactions suddenly become blocked, and few transactions can make progress. Thus, transaction throughput stops growing. Surprisingly, if enough transactions are initiated, throughput actually decreases. This is called **lock thrashing** (see Figure 6.7). The main issue in locking performance is to maximize throughput without reaching the point where thrashing occurs.

One way to understand lock thrashing is to consider the effect of slowly increasing the **transaction load**, which is measured by the number of active transactions. When the system is idle, the first transaction to run cannot block due to locks, because it's the only one requesting locks. As the number of active transactions grows, each successive transaction has a higher probability of becoming blocked due to transactions already running. When the number of active transactions is high enough, the next transaction to be started has virtually no chance of running to completion without blocking for some lock. Worse, it probably will get some locks before encountering one that blocks it, and these locks contribute to the likelihood that other active transactions will become blocked. So, not only does it not contribute to increased throughput, but by getting some locks that block other transactions, it actually reduces throughput. This leads to thrashing, where increasing the workload decreases the throughput.

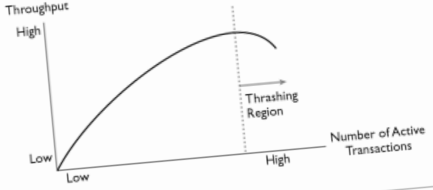
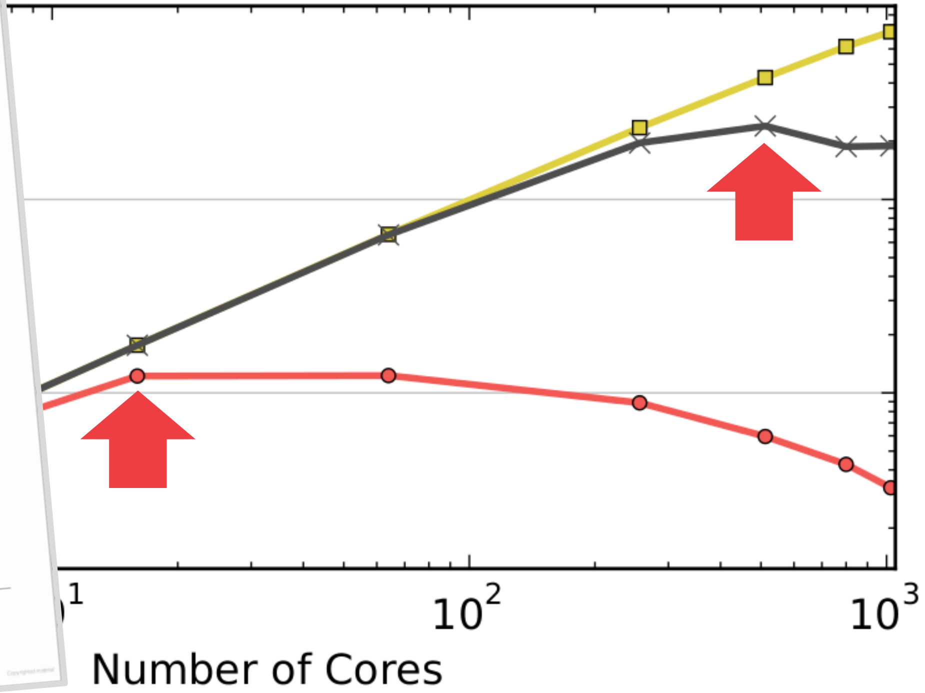
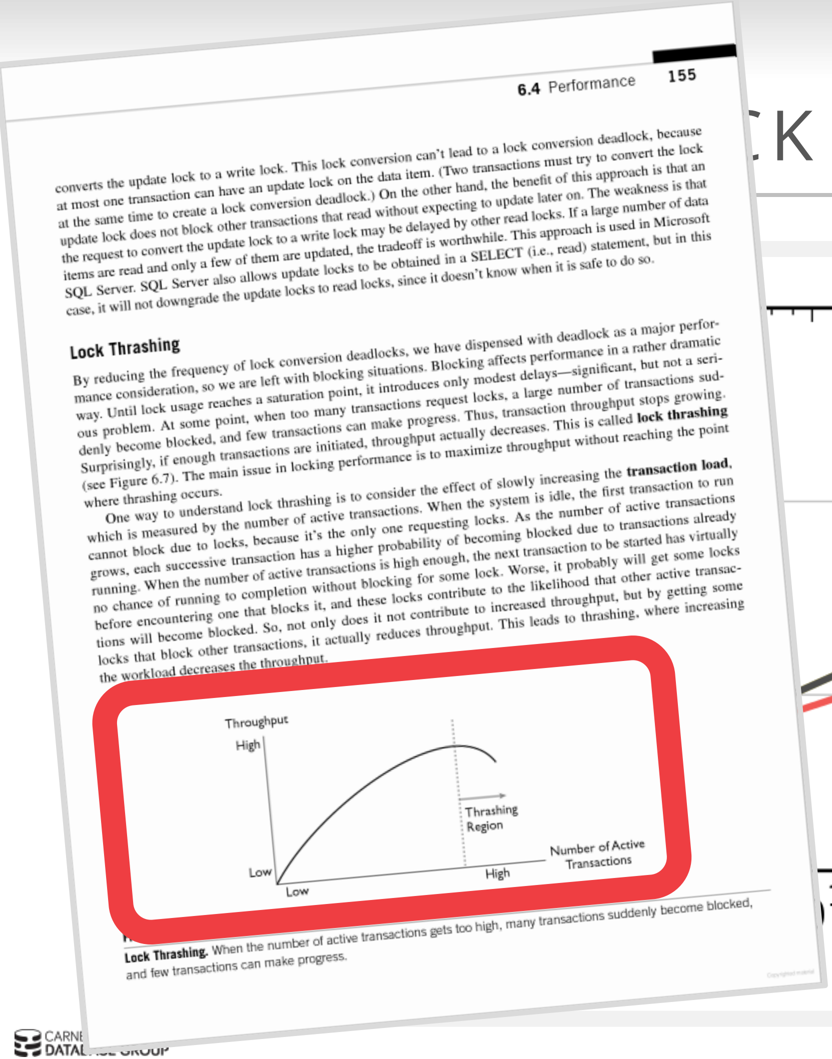


FIGURE 6.7
Lock Thrashing. When the number of active transactions gets too high, many transactions suddenly become blocked, and few transactions can make progress.



CK THRASHING



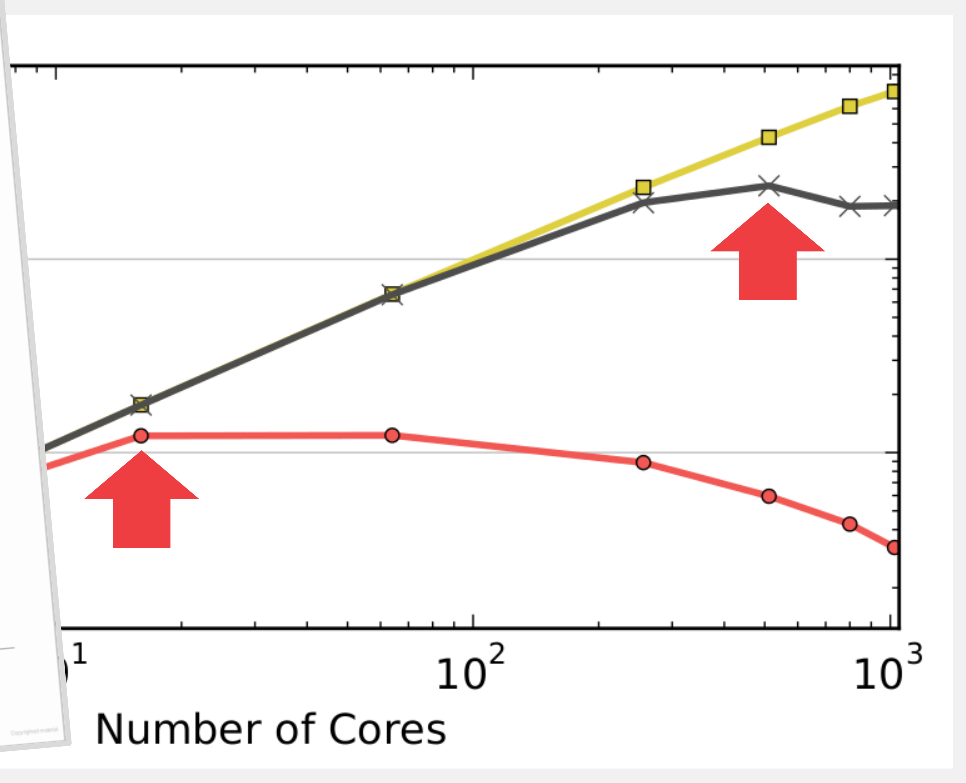
converts the update lock to a write lock. This lock conversion can't lead to a lock conversion deadlock, because at most one transaction can have an update lock on the data item. (Two transactions must try to convert the lock at the same time to create a lock conversion deadlock.) On the other hand, the benefit of this approach is that an update lock does not block other transactions that read without expecting to update later on. The weakness is that the request to convert the update lock to a write lock may be delayed by other read locks. If a large number of data items are read and only a few of them are updated, the tradeoff is worthwhile. This approach is used in Microsoft SQL Server. SQL Server also allows update locks to be obtained in a SELECT (i.e., read) statement, but in this case, it will not downgrade the update locks to read locks, since it doesn't know when it is safe to do so.

Lock Thrashing

By reducing the frequency of lock conversion deadlocks, we have dispensed with deadlock as a major performance consideration, so we are left with blocking situations. Blocking affects performance in a rather dramatic way. Until lock usage reaches a saturation point, it introduces only modest delays—significant, but not a serious problem. At some point, when too many transactions request locks, a large number of transactions suddenly become blocked, and few transactions can make progress. Thus, transaction throughput stops growing. Surprisingly, if enough transactions are initiated, throughput actually decreases. This is called **lock thrashing** (see Figure 6.7). The main issue in locking performance is to maximize throughput without reaching the point where thrashing occurs.

One way to understand lock thrashing is to consider the effect of slowly increasing the **transaction load**, which is measured by the number of active transactions. When the system is idle, the first transaction to run cannot block due to locks, because it's the only one requesting locks. As the number of active transactions grows, each successive transaction has a higher probability of becoming blocked due to transactions already running. When the number of active transactions is high enough, the next transaction to be started has virtually no chance of running to completion without blocking for some lock. Worse, it probably will get some locks before encountering one that blocks it, and these locks contribute to the likelihood that other active transactions will become blocked. So, not only does it not contribute to increased throughput, but by getting some locks that block other transactions, it actually reduces throughput. This leads to thrashing, where increasing the workload decreases the throughput.

Lock Thrashing. When the number of active transactions gets too high, many transactions suddenly become blocked, and few transactions can make progress.



TIMESTAMP ALLOCATION

Mutex
→ Worst option.

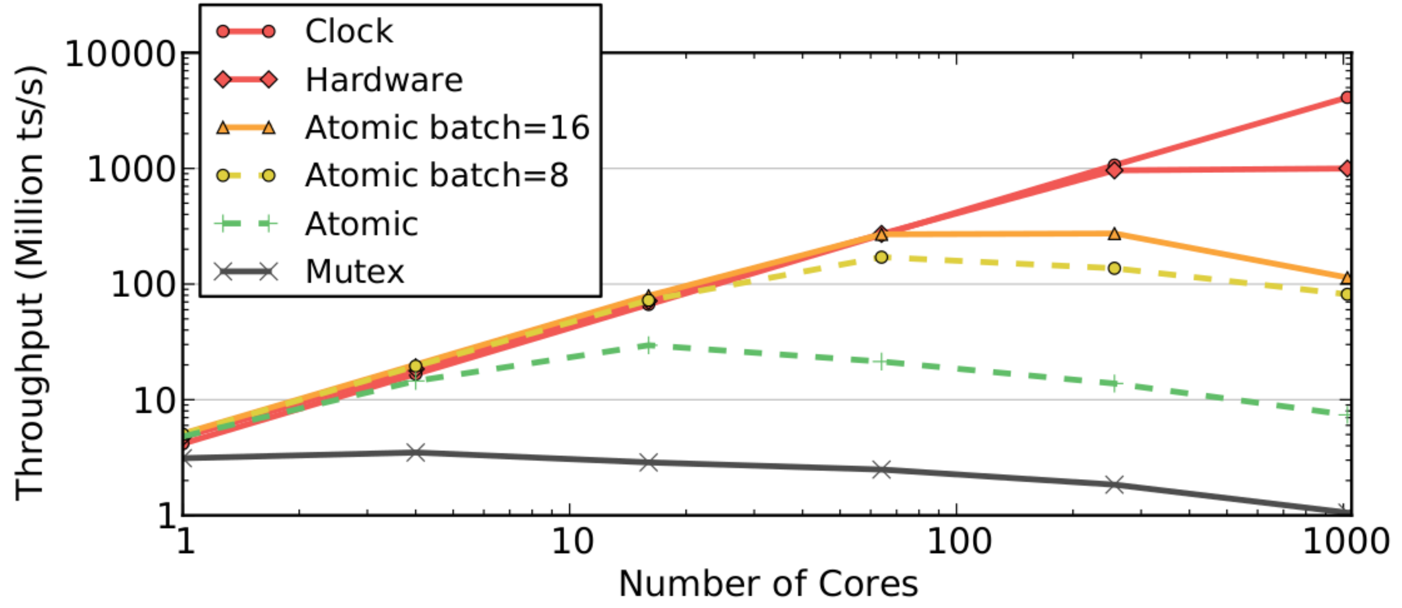
Atomic Addition
→ Requires cache invalidation on write.

Batched Atomic Addition
→ Needs a back-off mechanism to prevent fast burn.

Hardware Clock
→ Not sure if it will exist in future CPUs.

Hardware Counter
→ Not implemented in existing CPUs.

TIMESTAMP ALLOCATION



MEMORY ALLOCATIONS

Copying data on every read/write access slows down the DBMS because of contention on the memory controller.

→ In-place updates and non-copying reads are not affected as much.

Default libc `malloc` is slow. Never use it.

PARTITION-LEVEL LOCKING

The database is split up into horizontal partitions:

→ Each partition is assigned a single-threaded execution engine that has exclusive access to its data.

→ In-place updates.

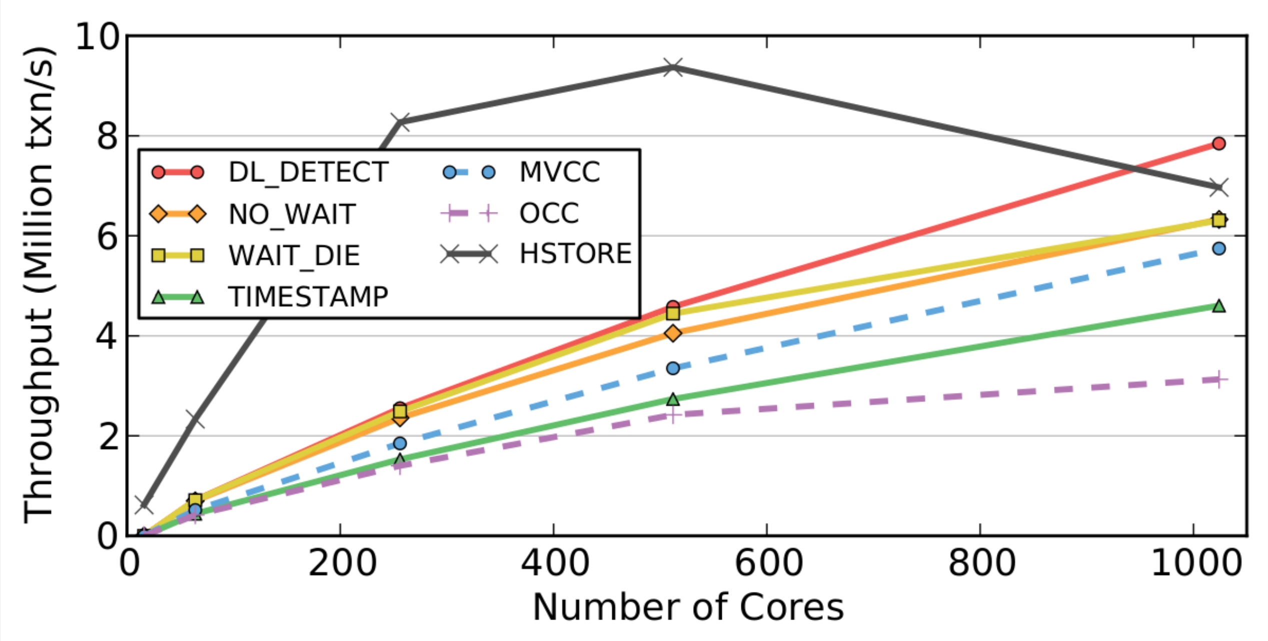
Only one txn can execute at a time per partition.

→ Order txns based on when they arrive at the DBMS.

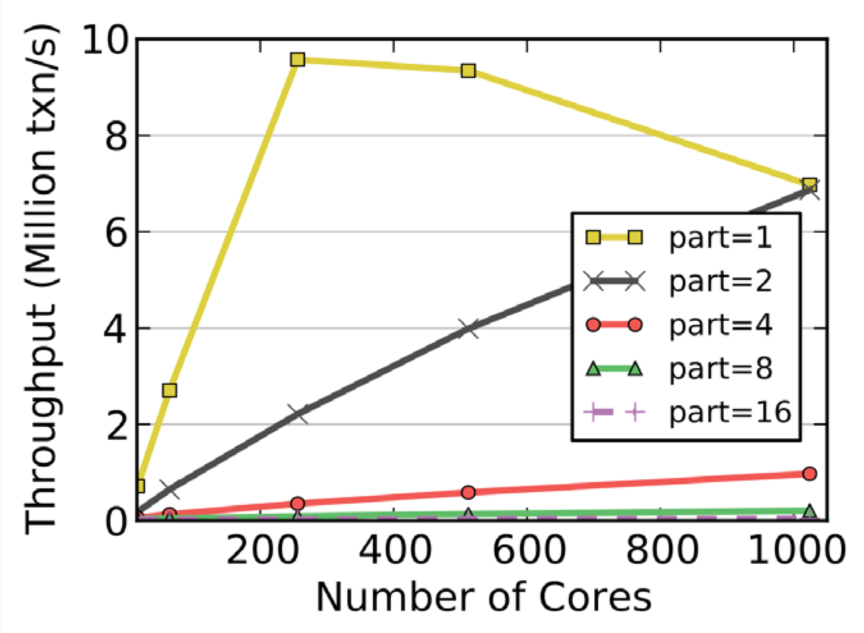
→ A txn acquires the lock for a partition when it has the lowest timestamp.

→ It is not allowed to access any partition that it does not hold the lock for.

READ-ONLY WORKLOAD



MULTI-PARTITION WORKLOADS



PARTING THOUGHTS

Concurrency control is hard to get correct and perform well.

Evaluation did not consider HTAP workloads.

NEXT CLASS

Isolation Levels

Modern MVCC