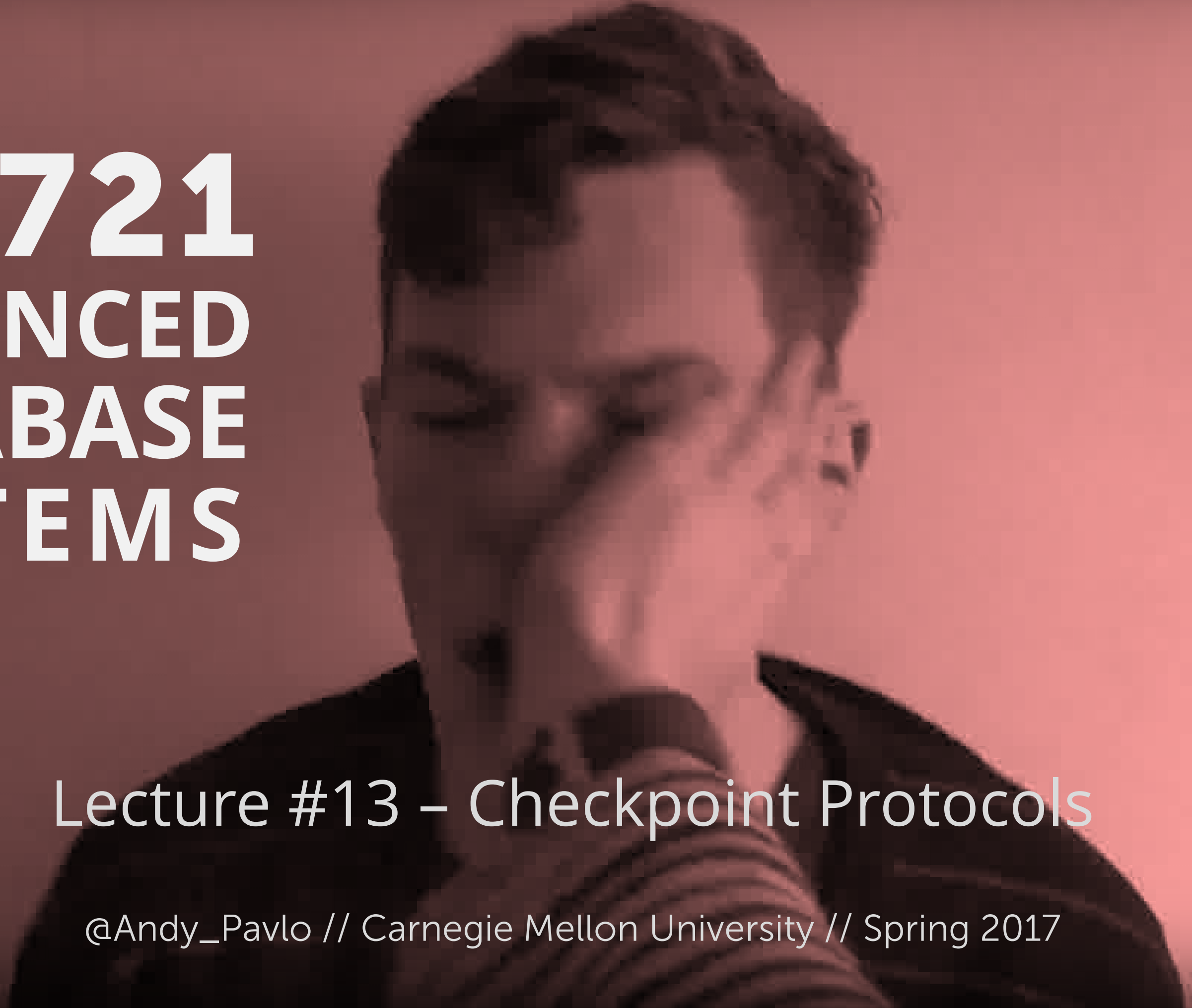
15-721 ADVANCED DATABASE SYSTEMS



Lecture #13 – Checkpoint Protocols

@Andy_Pavlo // Carnegie Mellon University // Spring 2017

TODAY'S AGENDA

Course Announcements
In-Memory Checkpoints
Shared Memory Restarts

COURSE ANNOUNCEMENTS

Autolab should be on-line now.

Project #2 is now due **March 9th @ 11:59pm**

Project #3 proposals are still due **March 21st**

OBSERVATION

Logging allows the DBMS to recover the database after a crash/restart. But this system will have to replay the entire log each time.

Checkpointing allows the systems to ignore large segments of the log to reduce recovery time.

IN-MEMORY CHECKPOINTS

There are different approaches for how the DBMS can create a new checkpoint for an in-memory database.

The choice of approach in a DBMS is tightly coupled with its concurrency control scheme.

The checkpoint thread scans each table and writes out data asynchronously to disk.

IDEAL CHECKPOINT PROPERTIES

Do **not** slow down regular txn processing.

Do **not** introduce unacceptable latency spikes.

Do **not** require excessive memory overhead.

LOW-OVERHEAD ASYNCHRONOUS CHECKPOINTING
IN MAIN-MEMORY DATABASE SYSTEMS
SIGMOD 2016

CONSISTENT VS. FUZZY CHECKPOINTS

Approach #1: Consistent Checkpoints

→ Represents a consistent snapshot of the database at some point in time. No uncommitted changes.

→ No additional processing during recovery.

Approach #2: Fuzzy Checkpoints

→ The snapshot could contain records updated from transactions that have not finished yet.

→ Must do additional processing to remove those changes.

FREQUENCY

Checkpointing too often causes the runtime performance to degrade.

→ The DBMS will spend too much time flushing buffers.

But waiting a long time between checkpoints is just as bad:

→ It will make recovery time much longer because the DBMS will have to replay a large log.

IN-MEMORY CHECKPOINTS

Approach #1: Naïve Snapshots

Approach #2: Copy-on-Update Snapshots

Approach #3: Wait-Free ZigZag

Approach #4: Wait-Free PingPong

FAST CHECKPOINT RECOVERY ALGORITHMS
FOR FREQUENTLY CONSISTENT APPLICATIONS
SIGMOD 2011

NAÏVE SNAPSHOT

Create a consistent copy of the entire database in a new location in memory and then write the contents to disk.

→ The DBMS blocks all txns during the checkpoint.

Two approaches to copying database:

→ Do it yourself (tuple blocks only).

→ Let the OS do it for you (everything).

HYPER – FORK SNAPSHOTS

Create a snapshot of the database by forking the DBMS process.

→ Child process contains a consistent checkpoint if there are not active txns.

→ Otherwise, use the in-memory undo log to roll back txns in the child process.

Continue processing txns in the parent process.

HYPER: A HYBRID OLTP&OLAP MAIN MEMORY DATABASE SYSTEM BASED ON VIRTUAL MEMORY SNAPSHOTS
ICDE 2011

COPY-ON-UPDATE SNAPSHOT

During the checkpoint, txns create new copies of data instead of overwriting it.

→ Copies can be at different granularities (block, tuple)

The checkpoint thread then skips anything that was created after it started.

→ Old data is pruned after it has been written to disk

VOLTDB – CONSISTENT CHECKPOINTS

A special txn starts a checkpoint and switches the DBMS into copy-on-write mode.

→ Changes are no longer made in-place to tables.
→ The DBMS tracks whether a tuple has been inserted, deleted, or modified since the checkpoint started.

A separate thread scans the tables and writes tuples out to the snapshot on disk.

→ Ignore anything changed after checkpoint.
→ Clean up old versions as it goes along.

OBSERVATION

Txns have to wait for the checkpoint thread when using naïve snapshots.

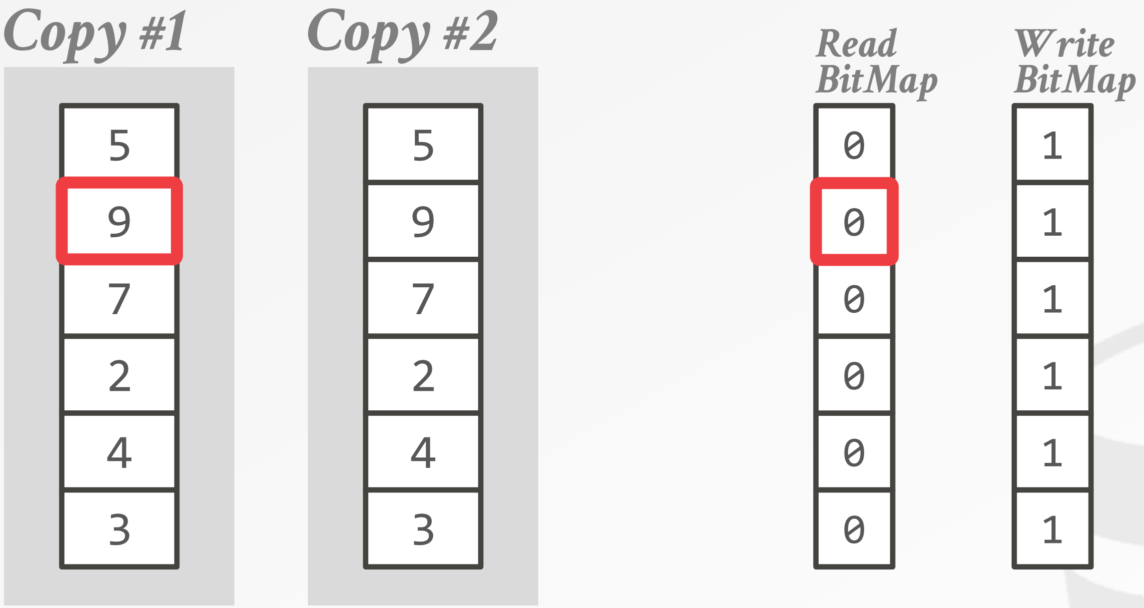
Txns may have to wait to acquire latches held by the checkpoint thread under copy-on-update

WAIT-FREE ZIGZAG

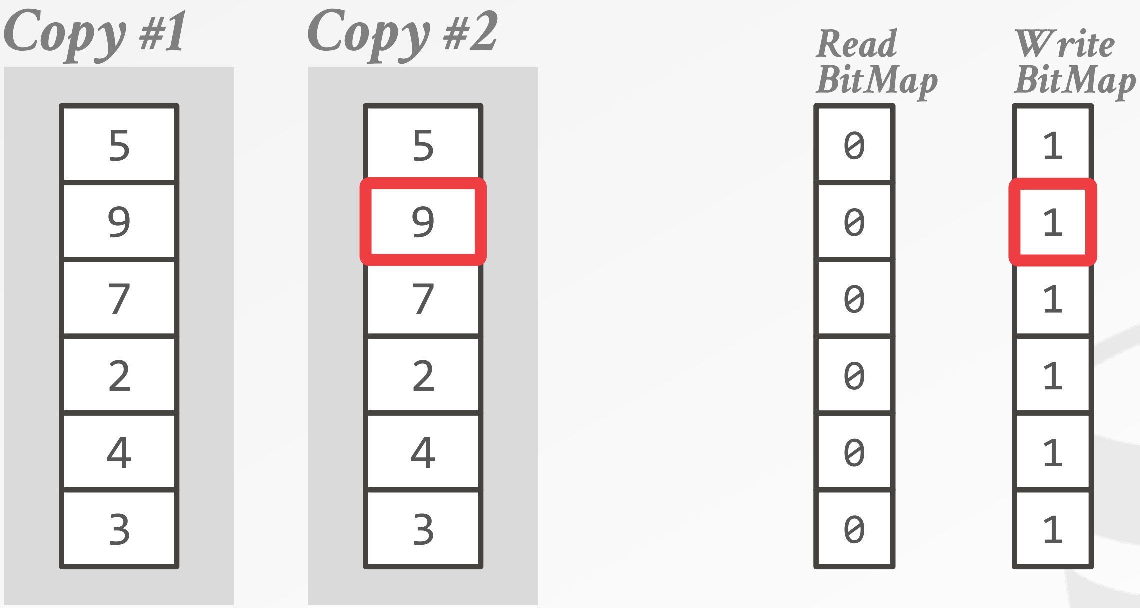
Maintain two copies of the entire database
→ Each txn write only updates one copy.

Use two BitMaps to keep track of what copy a txn should read/write from per tuple.
→ Avoid the overhead of having to create copies on the fly as in the copy-on-update approach.

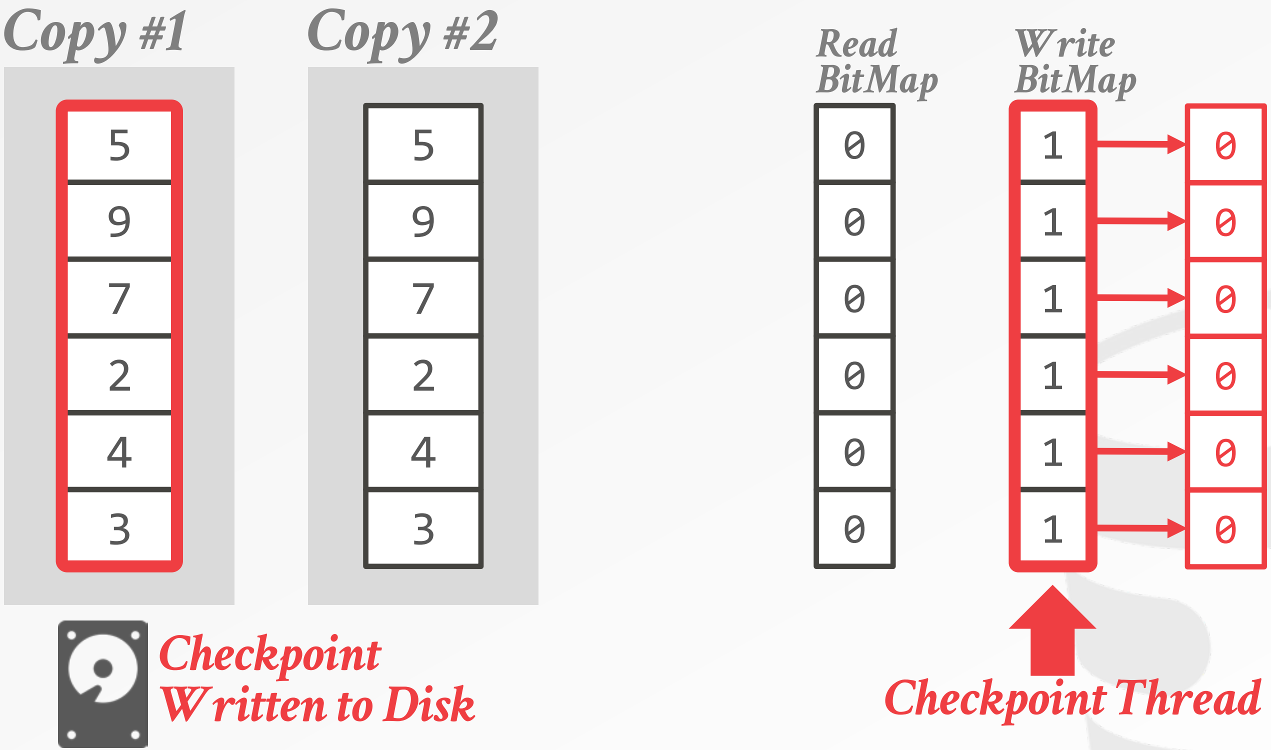
WAIT-FREE ZIGZAG



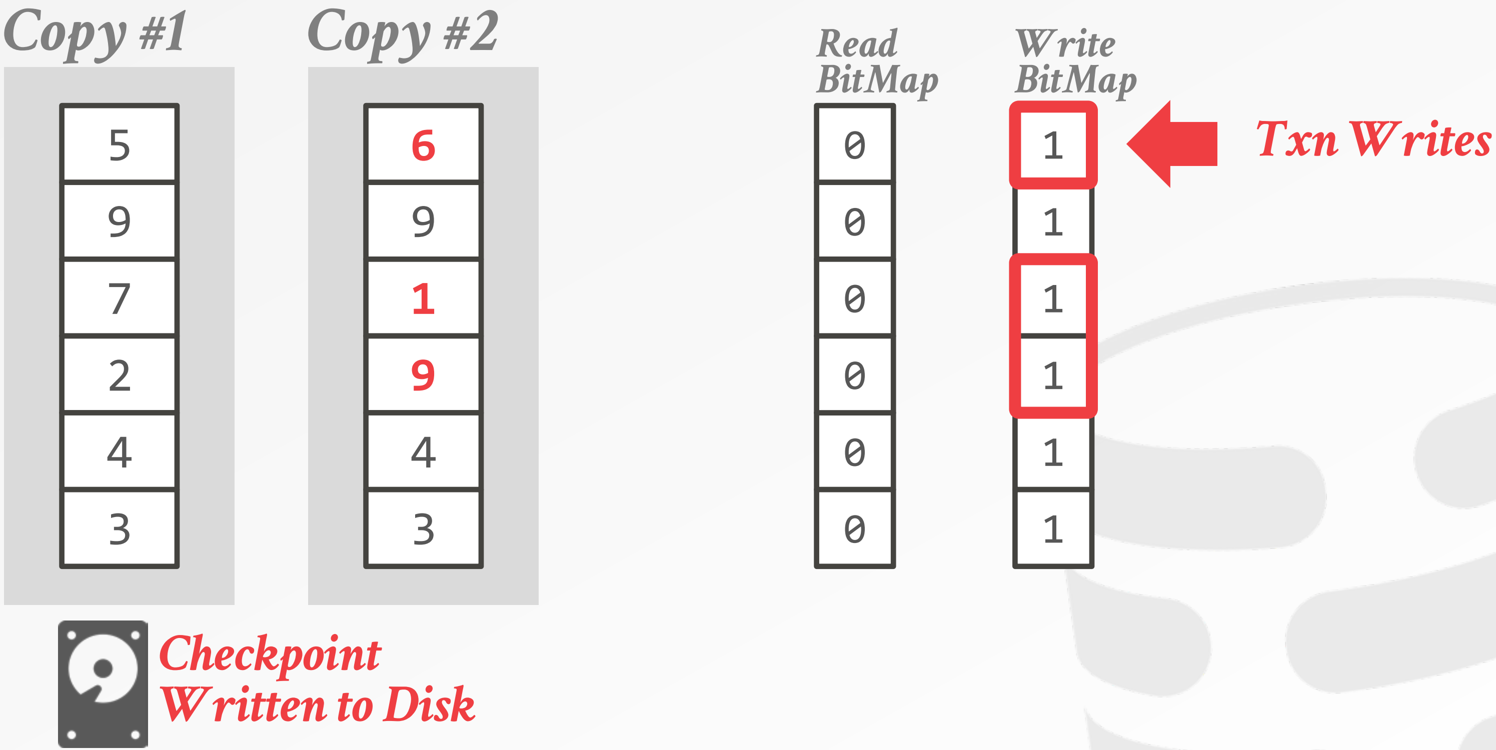
WAIT-FREE ZIGZAG



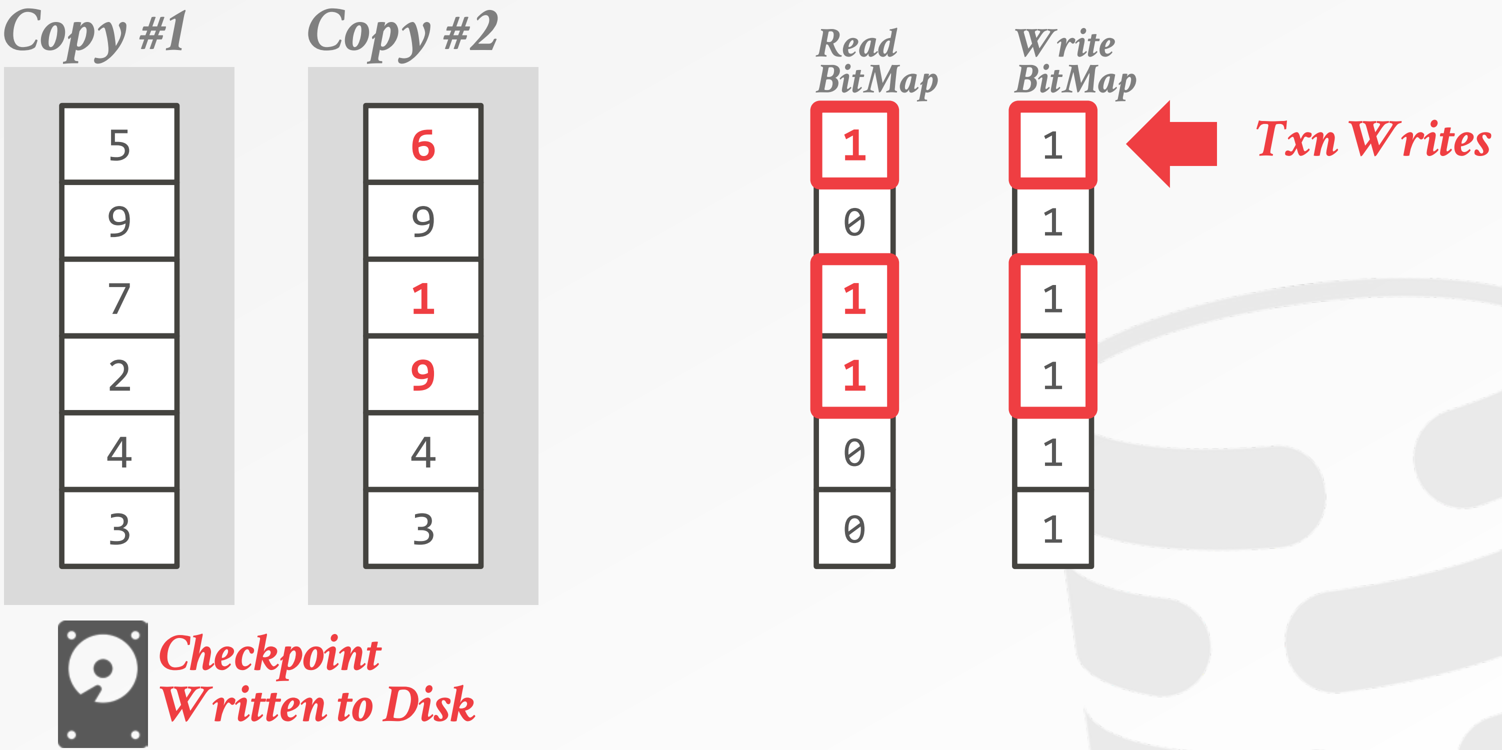
WAIT-FREE ZIGZAG



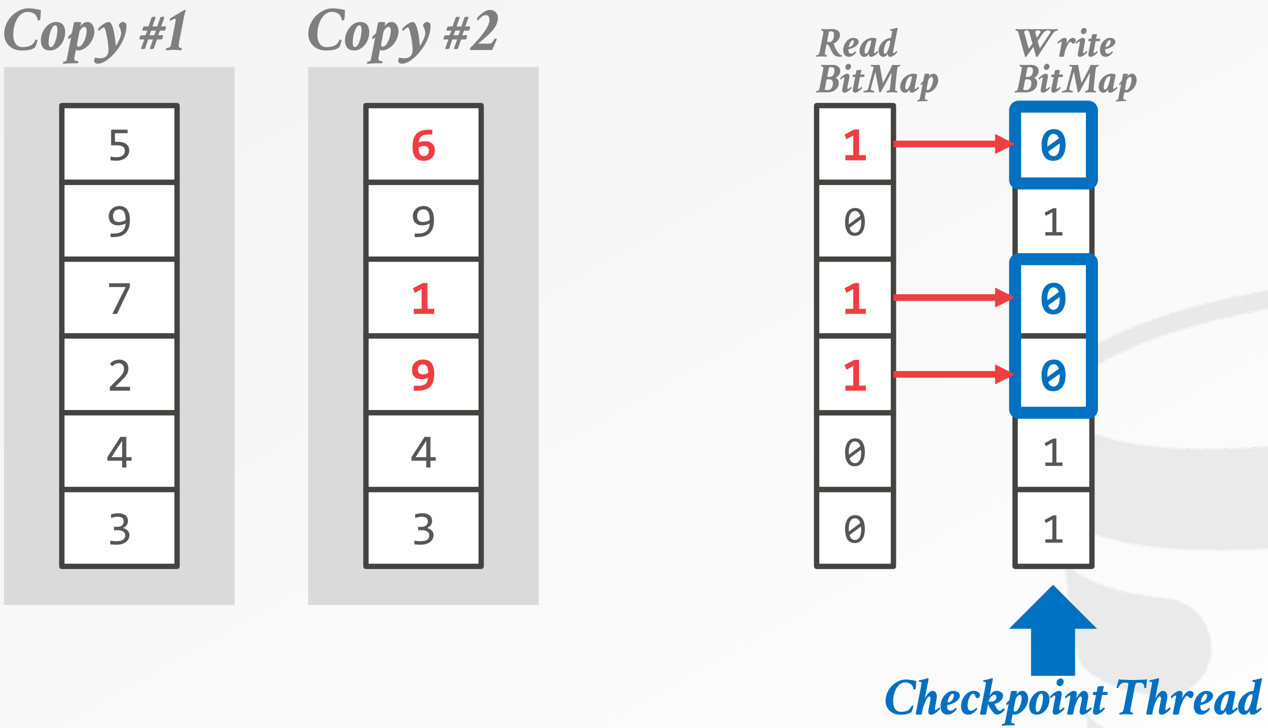
WAIT-FREE ZIGZAG



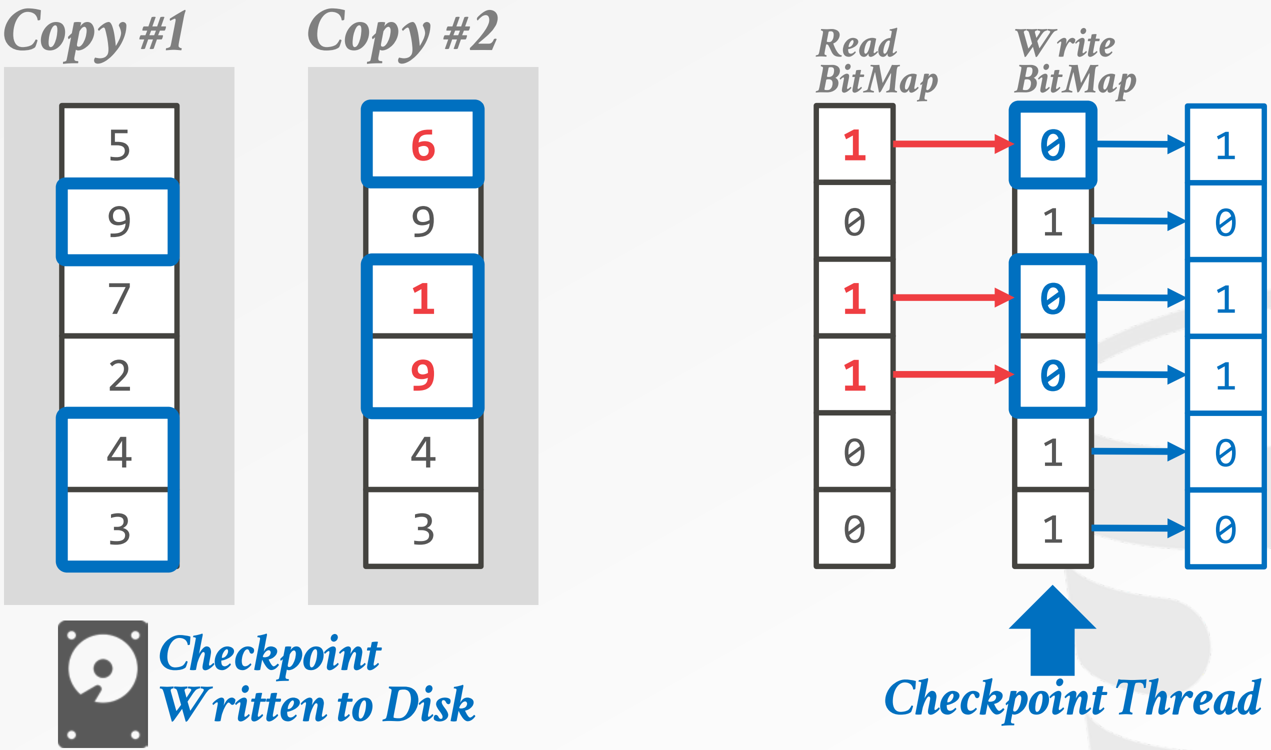
WAIT-FREE ZIGZAG



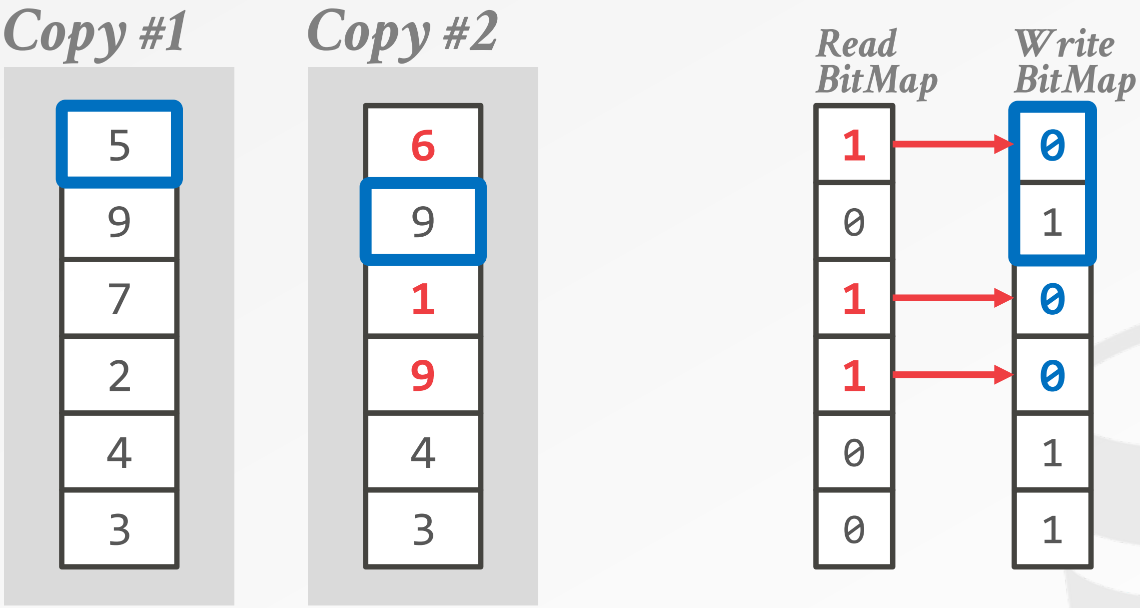
WAIT-FREE ZIGZAG



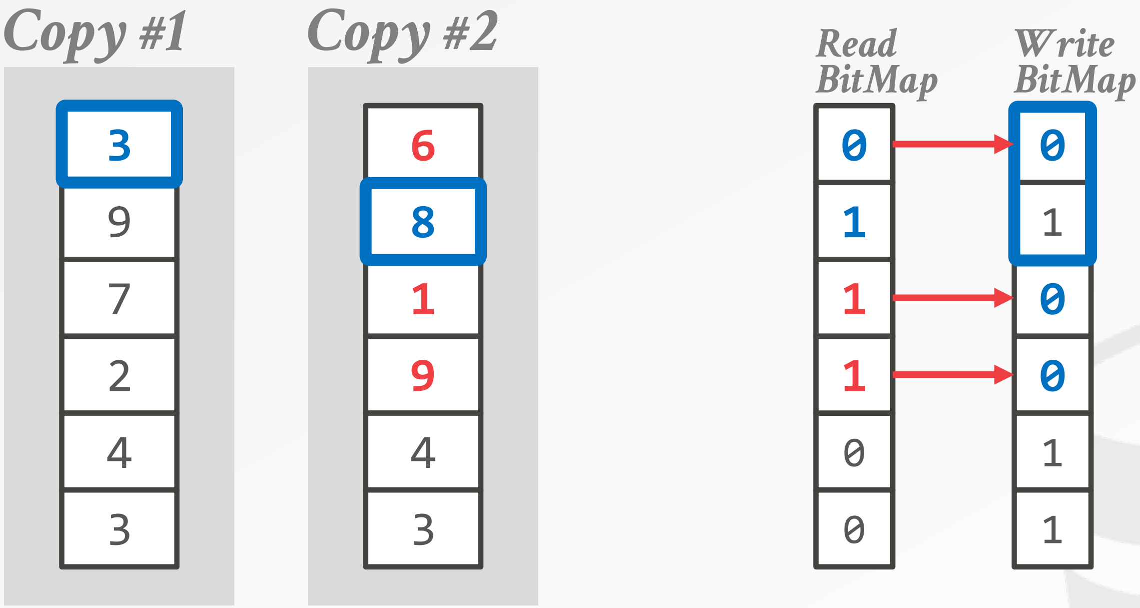
WAIT-FREE ZIGZAG



WAIT-FREE ZIGZAG



WAIT-FREE ZIGZAG



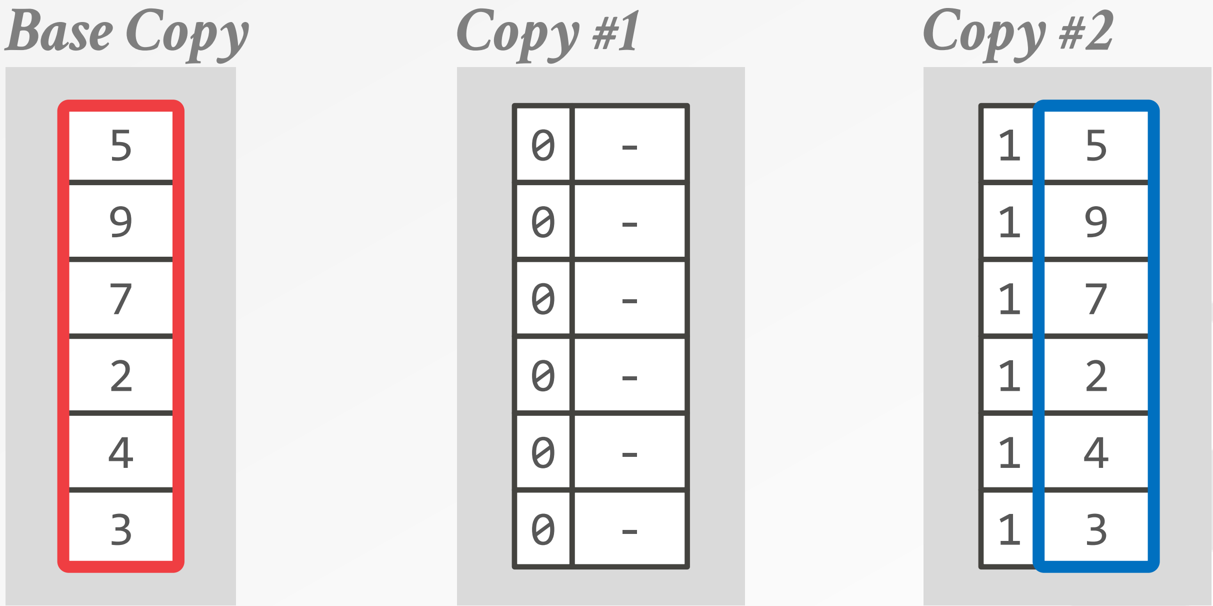
WAIT-FREE PINGPONG

Trade extra memory + CPU to avoid pauses at the end of the checkpoint.

Maintain two copies of the entire database at all times plus extra space for a shadow copy.

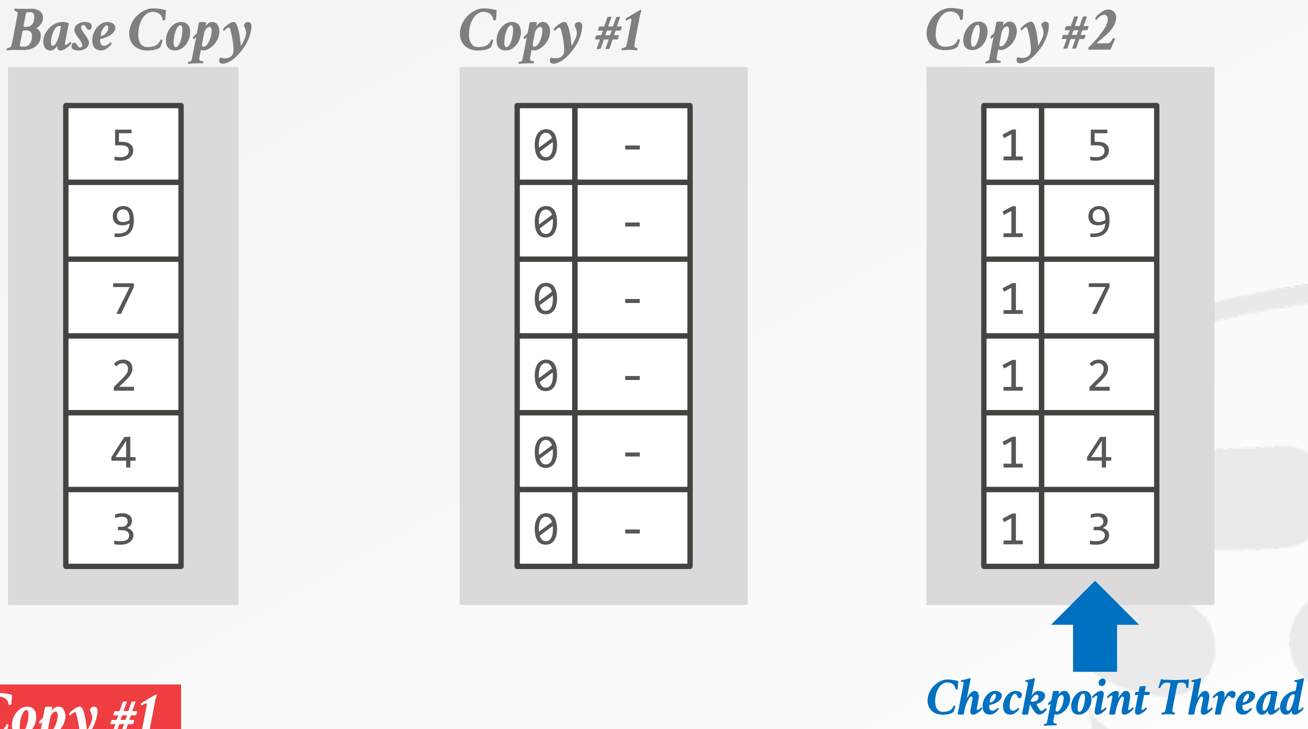
→ Pointer indicates which copy is the current master.
→ At the end of the checkpoint, swap these pointers.

WAIT-FREE PINGPONG



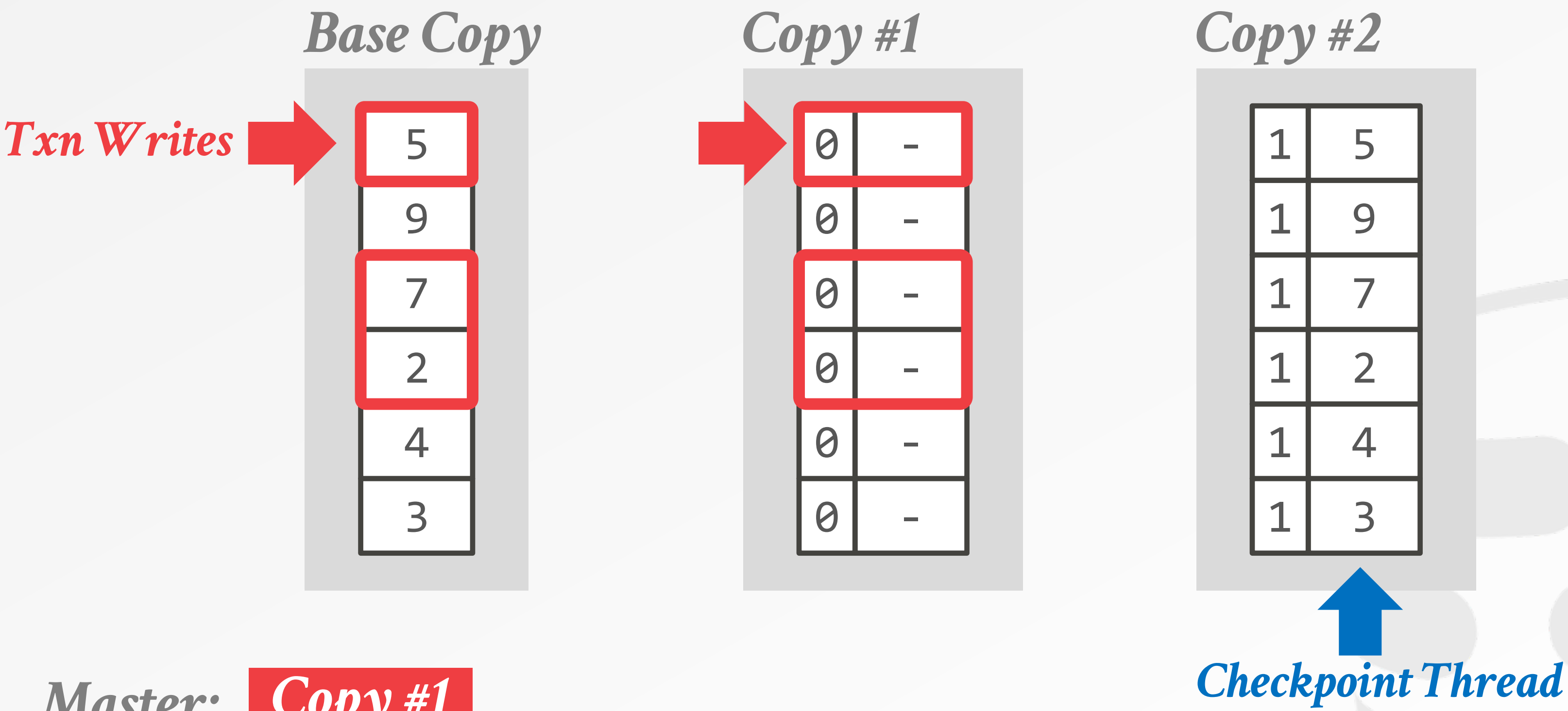
Master: **Copy #1**

WAIT-FREE PINGPONG



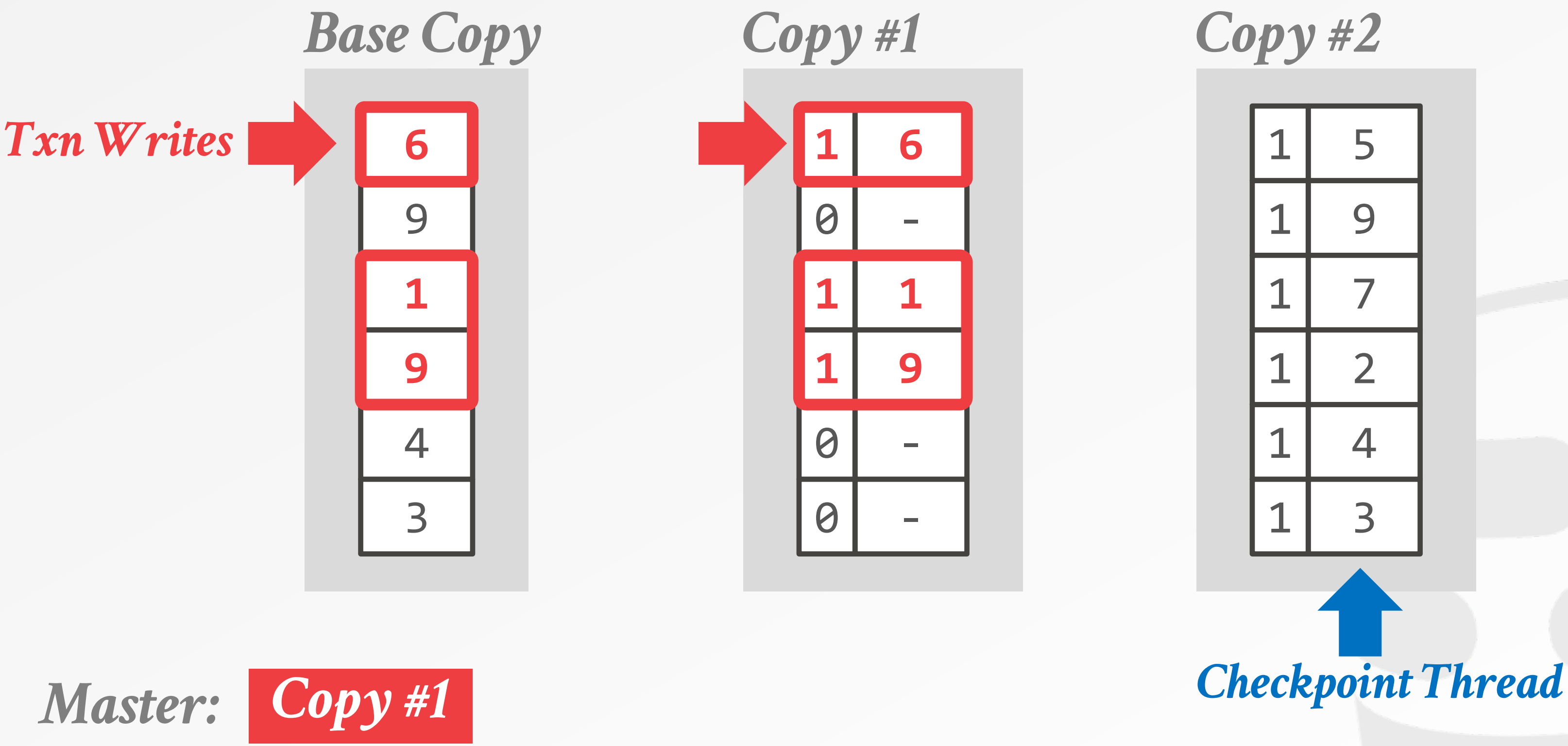
Master: **Copy #1**

WAIT-FREE PINGPONG

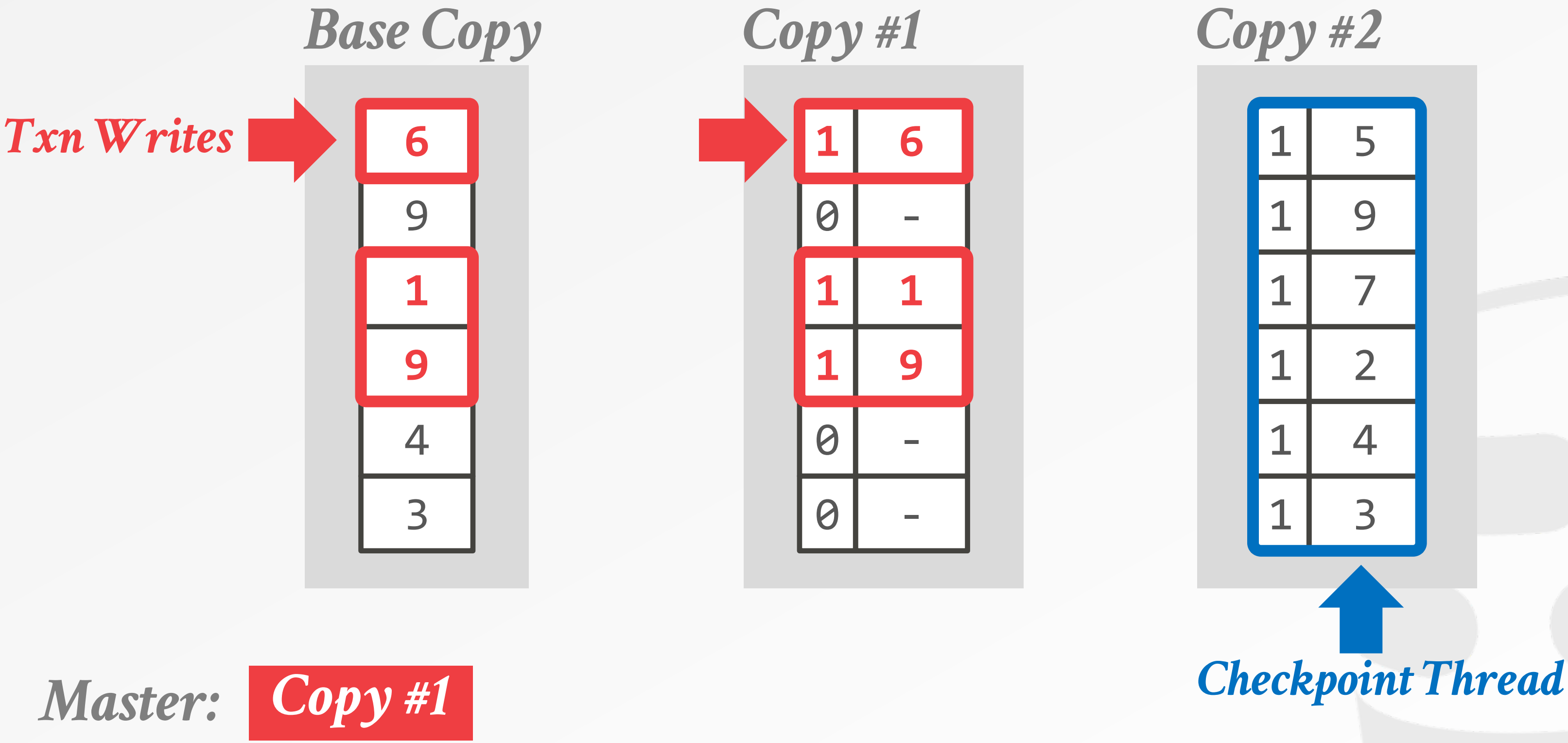


Master: **Copy #1**

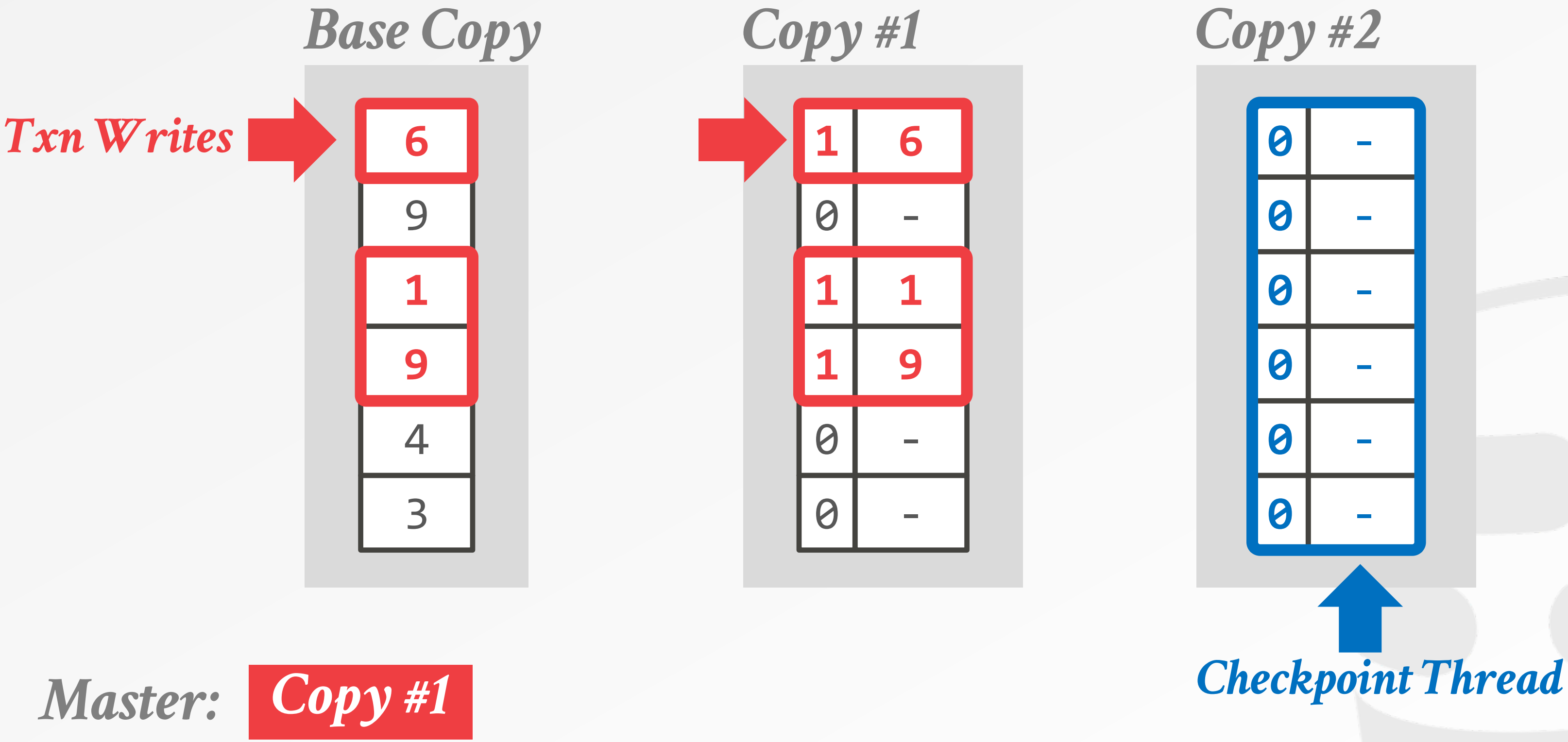
WAIT-FREE PINGPONG



WAIT-FREE PINGPONG



WAIT-FREE PINGPONG



WAIT-FREE PINGPONG

Base Copy

6
9
1
9
4
3

Copy #1

1	6
0	-
1	1
1	9
0	-
0	-

Copy #2

0	-
0	-
0	-
0	-
0	-
0	-

Master: *Copy #1*

WAIT-FREE PINGPONG

Base Copy

6
9
1
9
4
3

Copy #1

1	6
0	-
1	1
1	9
0	-
0	-

Copy #2

0	-
0	-
0	-
0	-
0	-
0	-

Master: *Copy #2*

WAIT-FREE PINGPONG

Base Copy

6
9
1
9
4
3

Copy #1

1	6
0	-
1	1
1	9
0	-
0	-

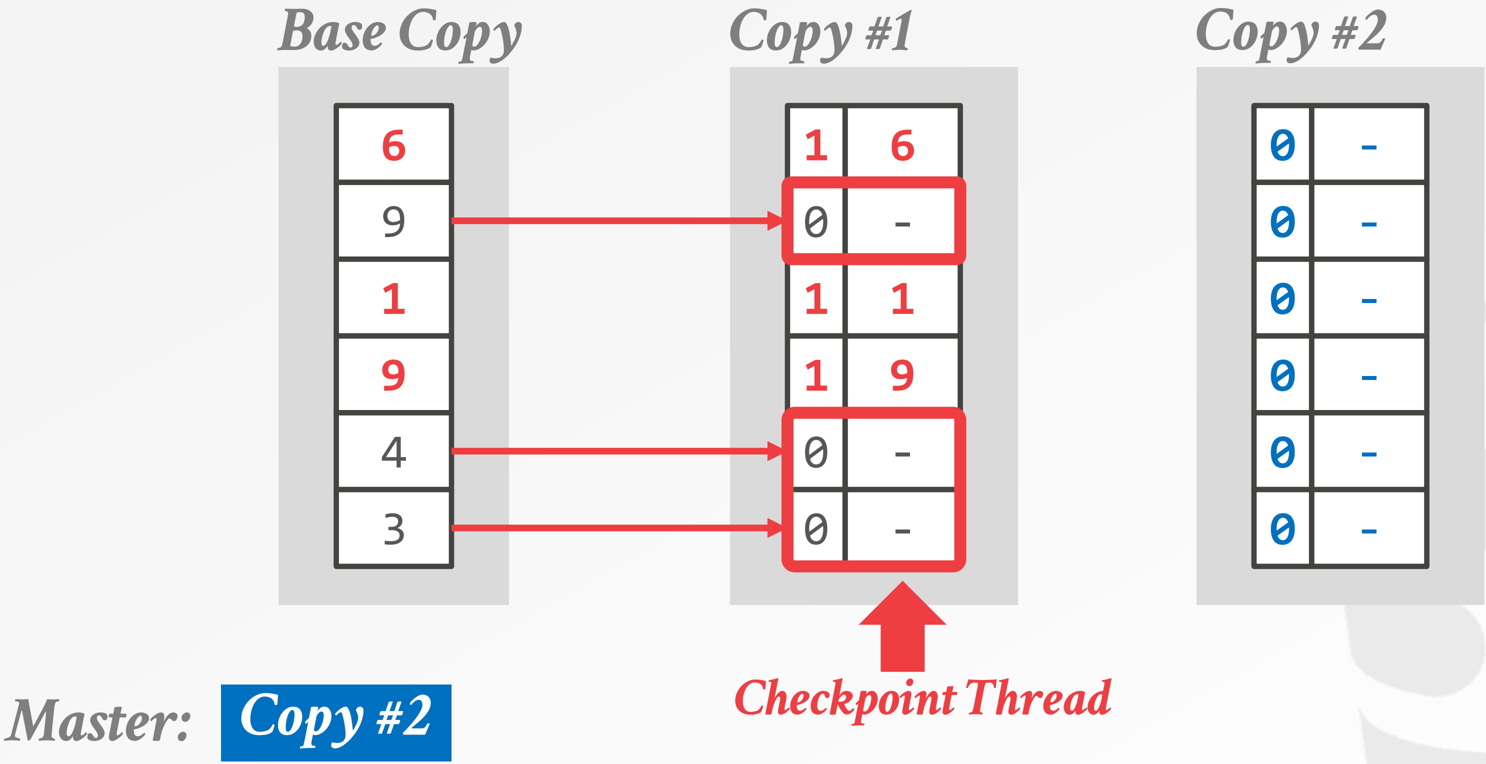
Copy #2

0	-
0	-
0	-
0	-
0	-
0	-

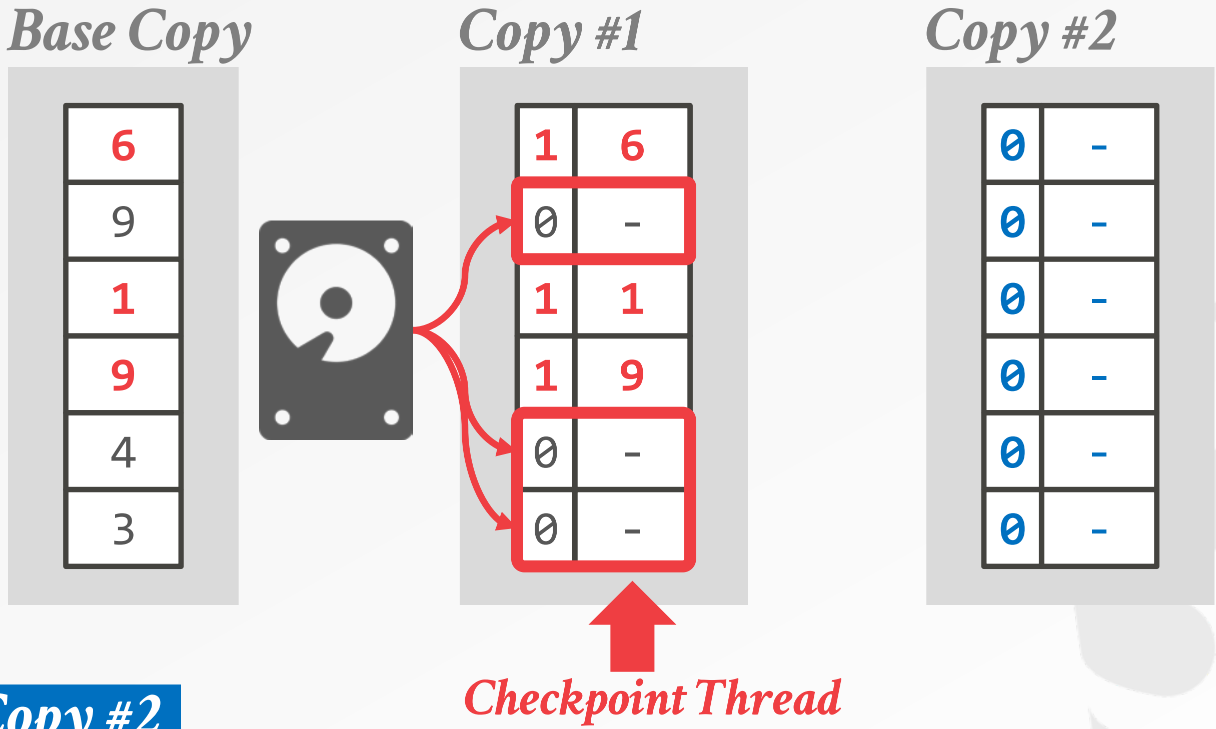
Checkpoint Thread

Master: *Copy #2*

WAIT-FREE PINGPONG



WAIT-FREE PINGPONG



Master: **Copy #2**

CHECKPOINT IMPLEMENTATIONS

Bulk State Copying

→ Pause txn execution to take a snapshot.

Locking

→ Use latches to isolate the checkpoint thread from the worker threads if they operate on shared regions.

Bulk Bit-Map Reset:

→ If DBMS uses BitMap to track dirty regions, it must perform a bulk reset at the start of a new checkpoint.

Memory Usage:

→ To avoid synchronous writes, the method may need to allocate additional memory for data copies.

IN-MEMORY CHECKPOINTS

	Bulk Copying	Locking	Bulk Bit-Map Reset	Memory Usage
Naïve Snapshot	Yes	**No**	**No**	2x
Copy-on-Update	**No**	Yes	Yes	2x
Wait-Free ZigZag	**No**	**No**	Yes	2x
Wait-Free Ping-Pong	**No**	**No**	**No**	3x

OBSERVATION

Not all DBMS restarts are due to crashes.

→ Updating OS libraries
→ Hardware upgrades/fixes
→ Updating DBMS software

Need a way to be able to quickly restart the DBMS without having to re-read the entire database from disk again.

FACEBOOK SCUBA – FAST RESTARTS

Decouple the in-memory database lifetime from the process lifetime.

By storing the database shared memory, the DBMS process can restart and the memory contents will survive.

FAST DATABASE RESTARTS AT FACEBOOK
SIGMOD 2014

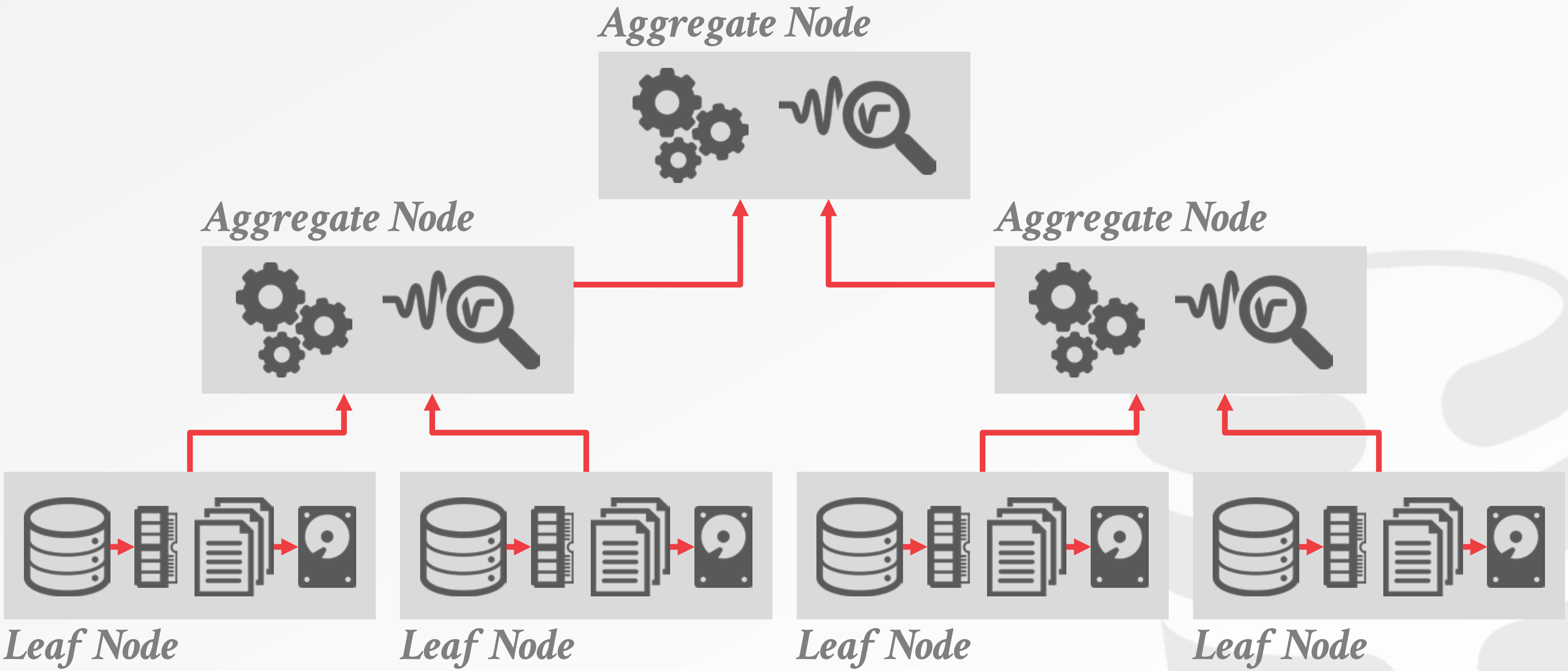
FACEBOOK SCUBA

Distributed, in-memory DBMS for time-series event analysis and anomaly detection.

Heterogeneous architecture

→ **Leaf Nodes**: Execute scans/filters on in-memory data
→ **Aggregator Nodes:** Combine results from leaf nodes

FACEBOOK SCUBA – ARCHITECTURE



SHARED MEMORY RESTARTS

Approach #1: Shared Memory Heaps

→ All data is allocated in SM during normal operations.
→ Have to use a custom allocator to subdivide memory segments for thread safety and scalability.
→ Cannot use lazy allocation of backing pages with SM.

Approach #2: Copy on Shutdown

→ All data is allocated in local memory during normal operations.
→ On shutdown, copy data from heap to SM.

FACEBOOK SCUBA – FAST RESTARTS

When the admin initiates restart command, the node halts ingesting updates.

DBMS starts copying data from heap memory to shared memory.
→ Delete blocks in heap once they are in SM.

Once snapshot finishes, the DBMS restarts.
→ On start up, check to see whether the there is a valid database in SM to copy into its heap.
→ Otherwise, the DBMS restarts from disk.

PARTING THOUGHTS

I think that copy-on-update checkpoints are the way to go especially if you are using MVCC

Shared memory does have some use after all…

NEXT CLASS

Optimizers!

Project #2 is now due **March 9th @ 11:59pm**

Project #3 proposals are still due **March 21st**