15-721 DATABASE SYSTEMS

Lecture #14 – Logging & Recovery (Alternative Methods)

Andy Pavlo // Carnegie Mellon University // Spring 2016

TODAY'S AGENDA

Course Announcements
Physical Logging Clarification
Command Logging
In-Memory Checkpoints
Shared Memory Restarts

COURSE ANNOUNCEMENTS

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

Project #3 proposals are still due **March 14th**

No Mandatory Reading for **March 2nd**

GRADE BREAKDOWN

Reading Reviews (10%)
Project #1 (10%)
Project #2 (25%)
Project #3 (45%)
Final Exam (10%)
Extra Credit (+10%)

LOGICAL LOGGING EXAMPLE

```
UPDATE employees
   SET salary = salary * 1.10
```

```
UPDATE employees
   SET salary = 900
 WHERE name = 'Joy'
```

NAME	SALARY
O.D.B.	$100
El-P	$666
Joy	$888

Logical Log

LOGICAL LOGGING EXAMPLE

```
UPDATE employees
   SET salary = salary * 1.10
```

```
UPDATE employees
   SET salary = 900
 WHERE name = 'Joy'
```

NAME	SALARY
O.D.B.	$100
El-P	$666
Joy	$888

Logical Log

LOGICAL LOGGING EXAMPLE

```
UPDATE employees
   SET salary = salary * 1.10
```

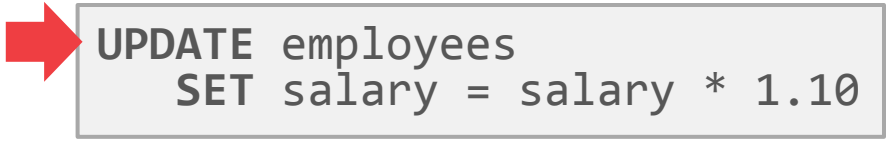
```
UPDATE employees
   SET salary = 900
 WHERE name = 'Joy'
```

NAME	SALARY
O.D.B.	$100
El-P	$666
Joy	$888

Logical Log

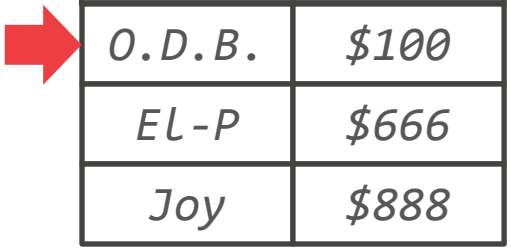
```
UPDATE employees SET
salary = salary * 1.10
```

LOGICAL LOGGING EXAMPLE



```
UPDATE employees
   SET salary = salary * 1.10
```

```
UPDATE employees
   SET salary = 900
 WHERE name = 'Joy'
```



NAME	SALARY
O.D.B.	$100
El-P	$666
Joy	$888

Logical Log

```
UPDATE employees SET
salary = salary * 1.10
```

LOGICAL LOGGING EXAMPLE

```
UPDATE employees
   SET salary = salary * 1.10
```

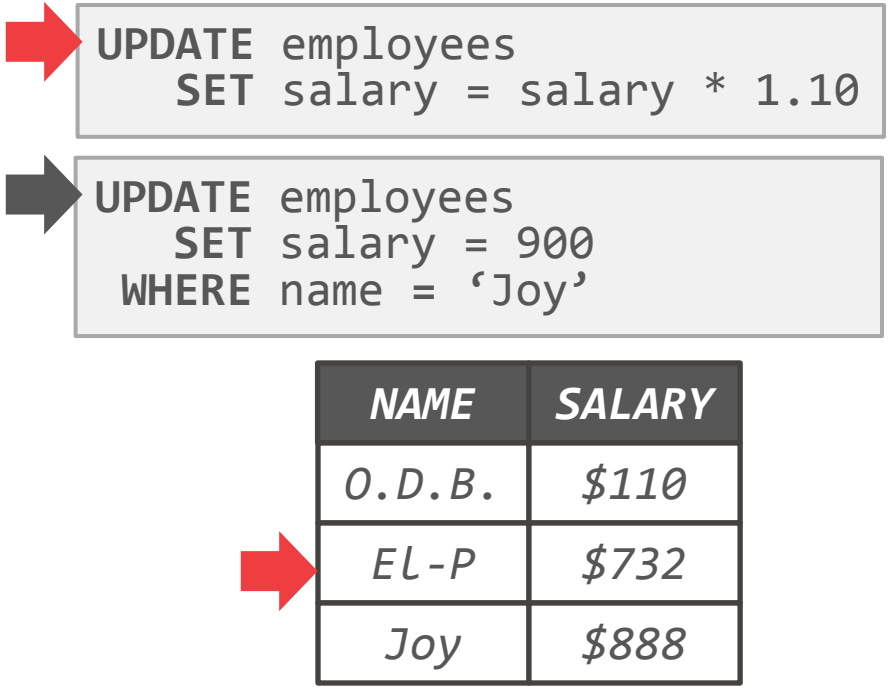
```
UPDATE employees
   SET salary = 900
 WHERE name = 'Joy'
```

NAME	SALARY
O.D.B.	$110
El-P	$732
Joy	$888

Logical Log

```
UPDATE employees SET
salary = salary * 1.10
```

LOGICAL LOGGING EXAMPLE



Logical Log

```
UPDATE employees SET
salary = salary * 1.10

UPDATE employees SET
salary = 900 WHERE
name = 'Joy'
```

LOGICAL LOGGING EXAMPLE

```
UPDATE employees
   SET salary = salary * 1.10
```

```
UPDATE employees
   SET salary = 900
 WHERE name = 'Joy'
```

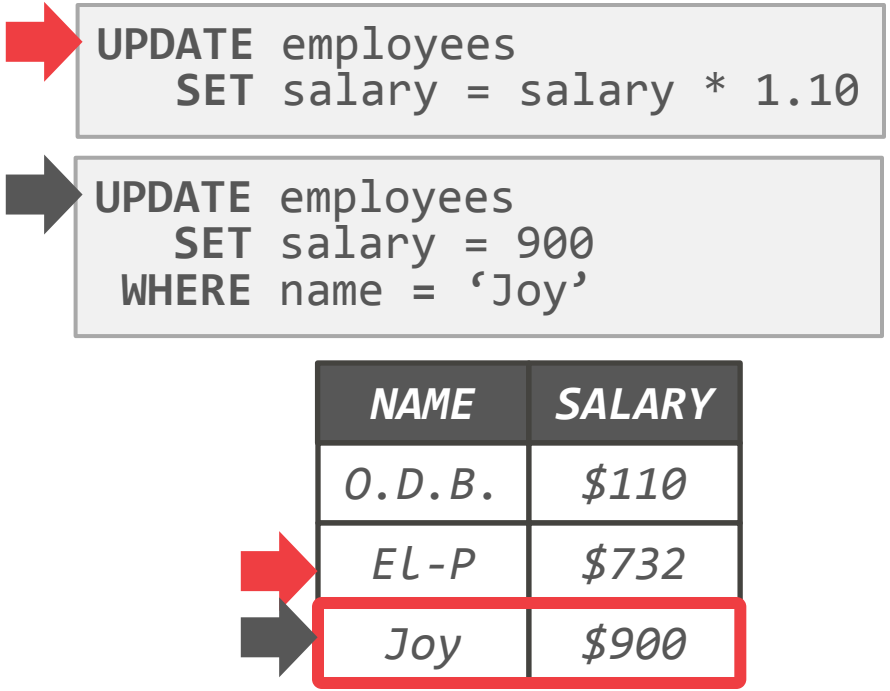
NAME	SALARY
O.D.B.	$110
El-P	$732
Joy	$888

Logical Log

```
UPDATE employees SET
salary = salary * 1.10

UPDATE employees SET
salary = 900 WHERE
name = 'Joy'
```

LOGICAL LOGGING EXAMPLE



Logical Log

```
UPDATE employees SET
salary = salary * 1.10

UPDATE employees SET
salary = 900 WHERE
name = 'Joy'
```

LOGICAL LOGGING EXAMPLE

```
UPDATE employees
   SET salary = salary * 1.10
```

```
UPDATE employees
   SET salary = 900
 WHERE name = 'Joy'
```

NAME	SALARY
O.D.B.	$110
El-P	$732
Joy	$900

Logical Log

```
UPDATE employees SET
salary = salary * 1.10

UPDATE employees SET
salary = 900 WHERE
name = 'Joy'
```

LOGICAL LOGGING EXAMPLE

```
UPDATE employees
   SET salary = salary * 1.10
```

```
UPDATE employees
   SET salary = 900
 WHERE name = 'Joy'
```

NAME	SALARY
O.D.B.	$110
El-P	$732
Joy	$900

Logical Log

```
UPDATE employees SET
salary = salary * 1.10

UPDATE employees SET
salary = 900 WHERE
name = 'Joy'
```

LOGICAL LOGGING EXAMPLE

```
UPDATE employees
   SET salary = salary * 1.10
```

```
UPDATE employees
   SET salary = 900
 WHERE name = 'Joy'
```

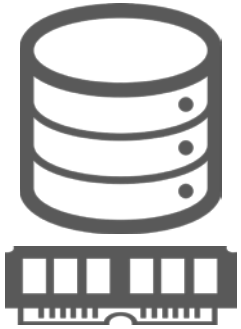
NAME	SALARY
O.D.B.	$110
El-P	$732
Joy	$900

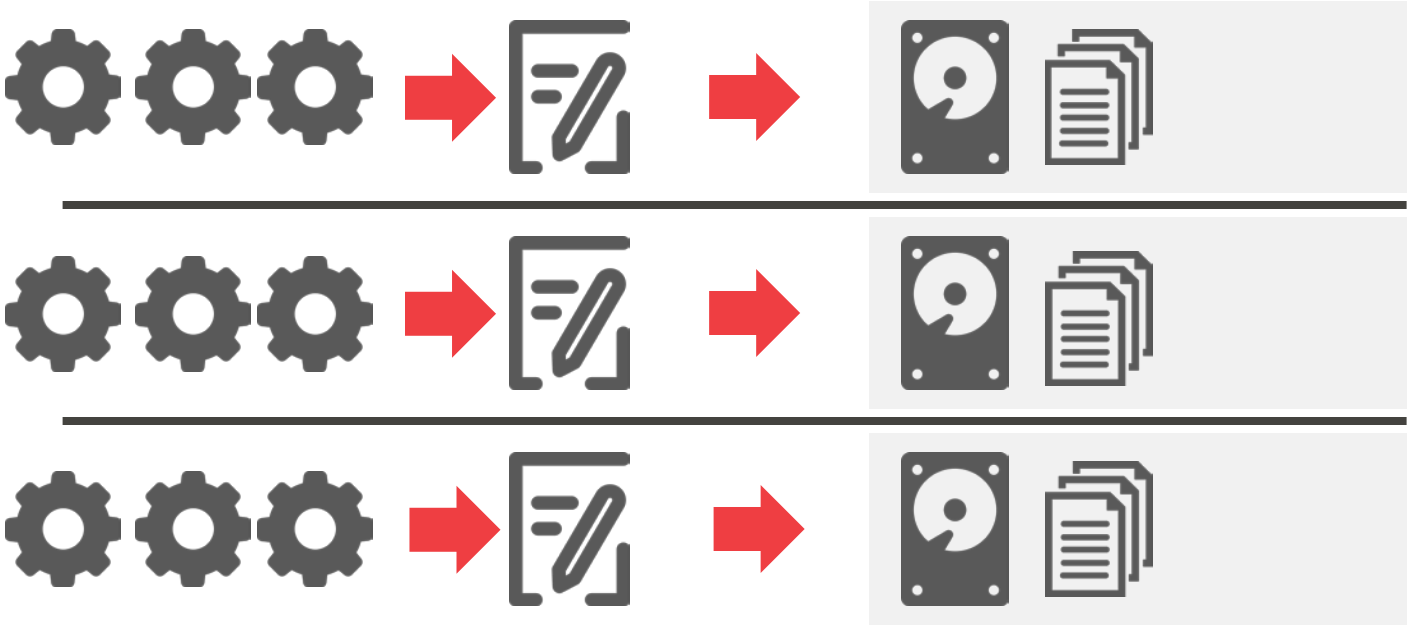
Logical Log

```
UPDATE employees SET
salary = salary * 1.10

UPDATE employees SET
salary = 900 WHERE
name = 'Joy'
```

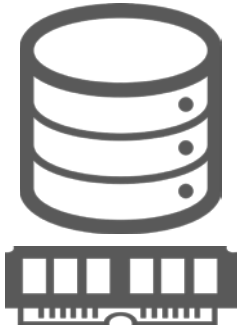
SILOR – ARCHITECTURE

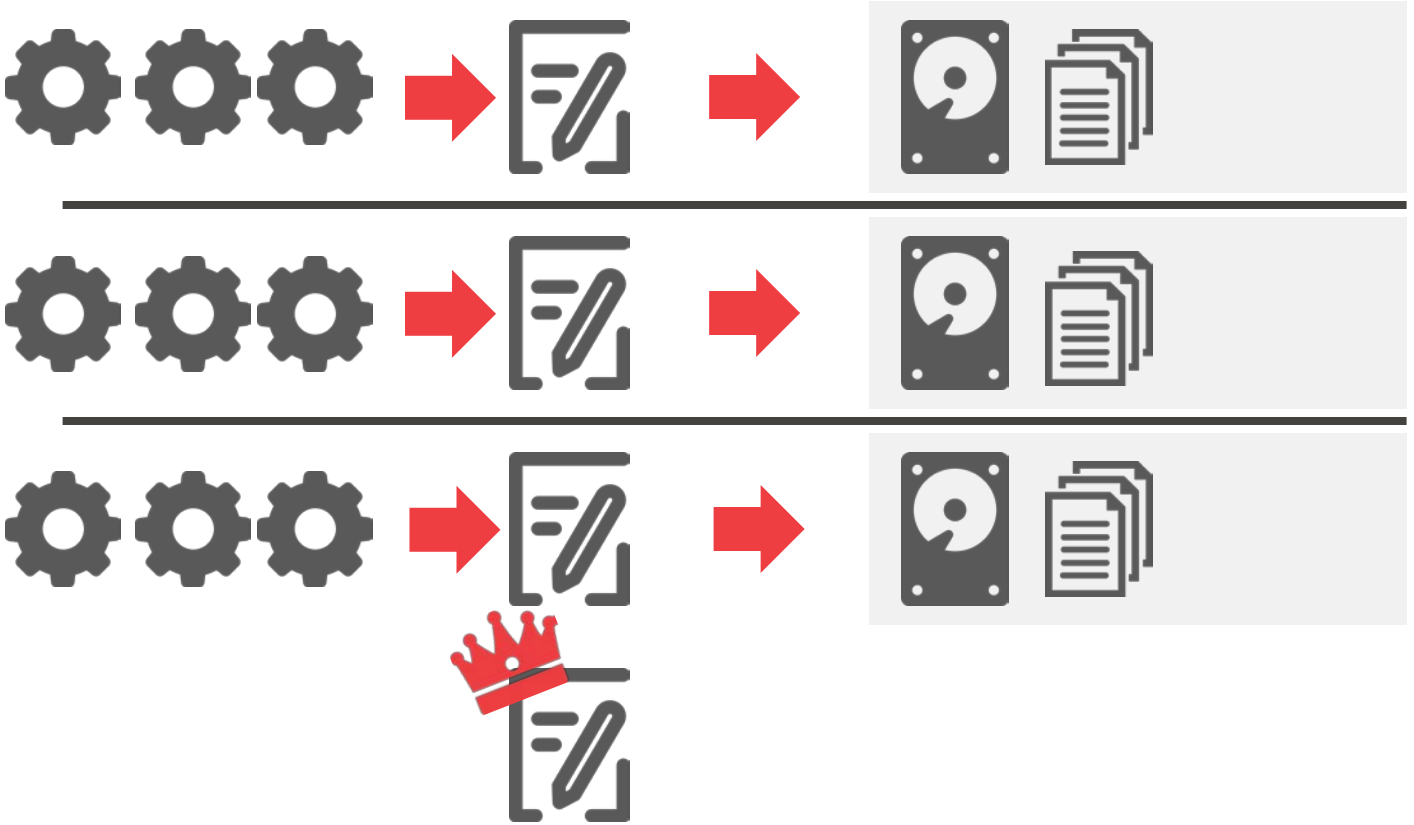




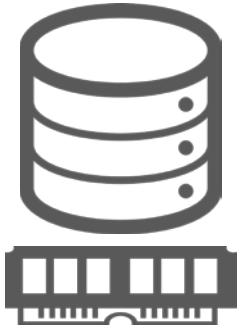
epoch=100

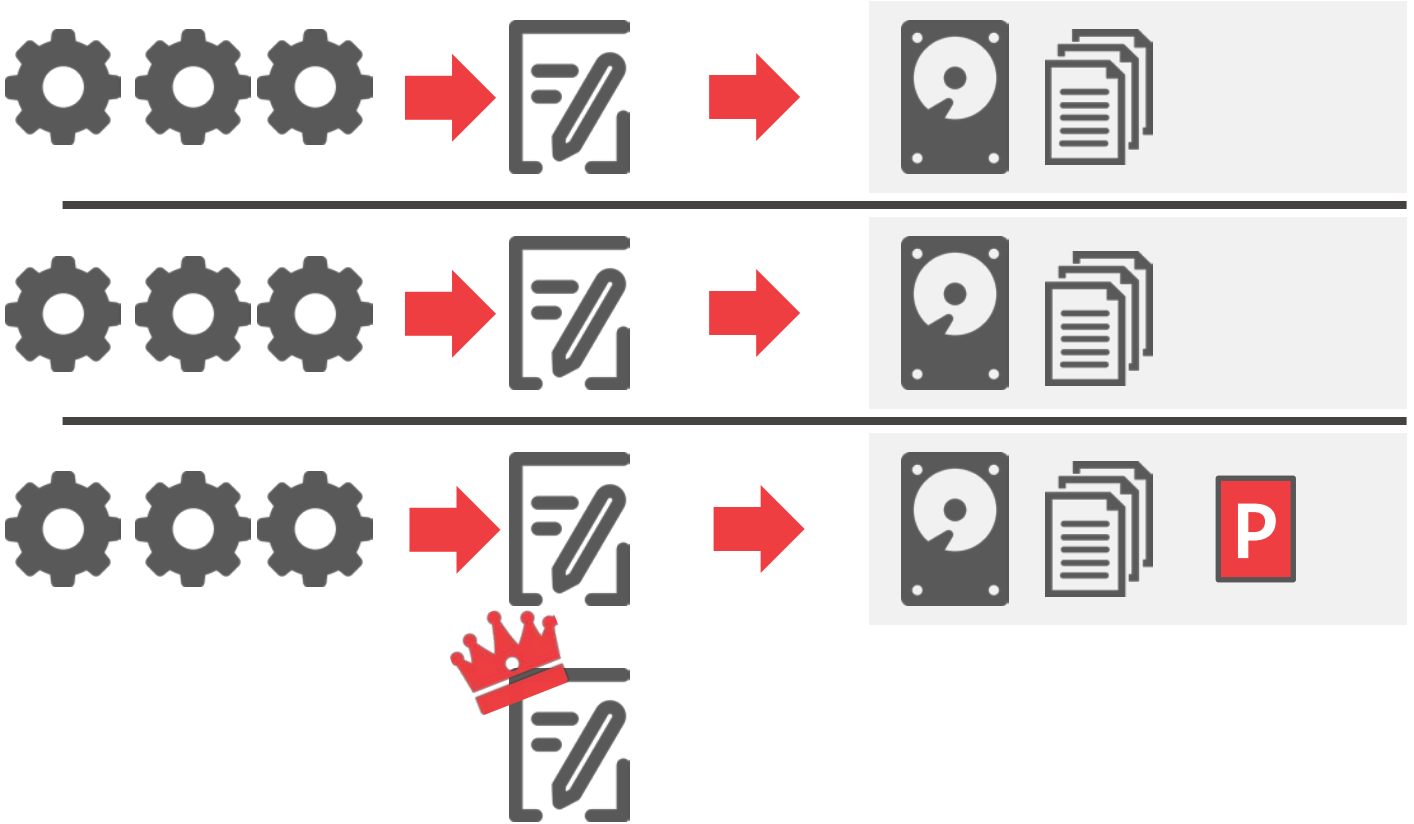
SILOR – ARCHITECTURE



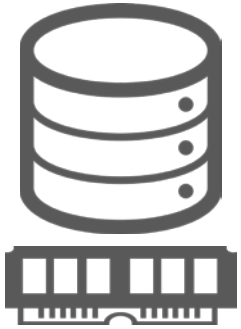


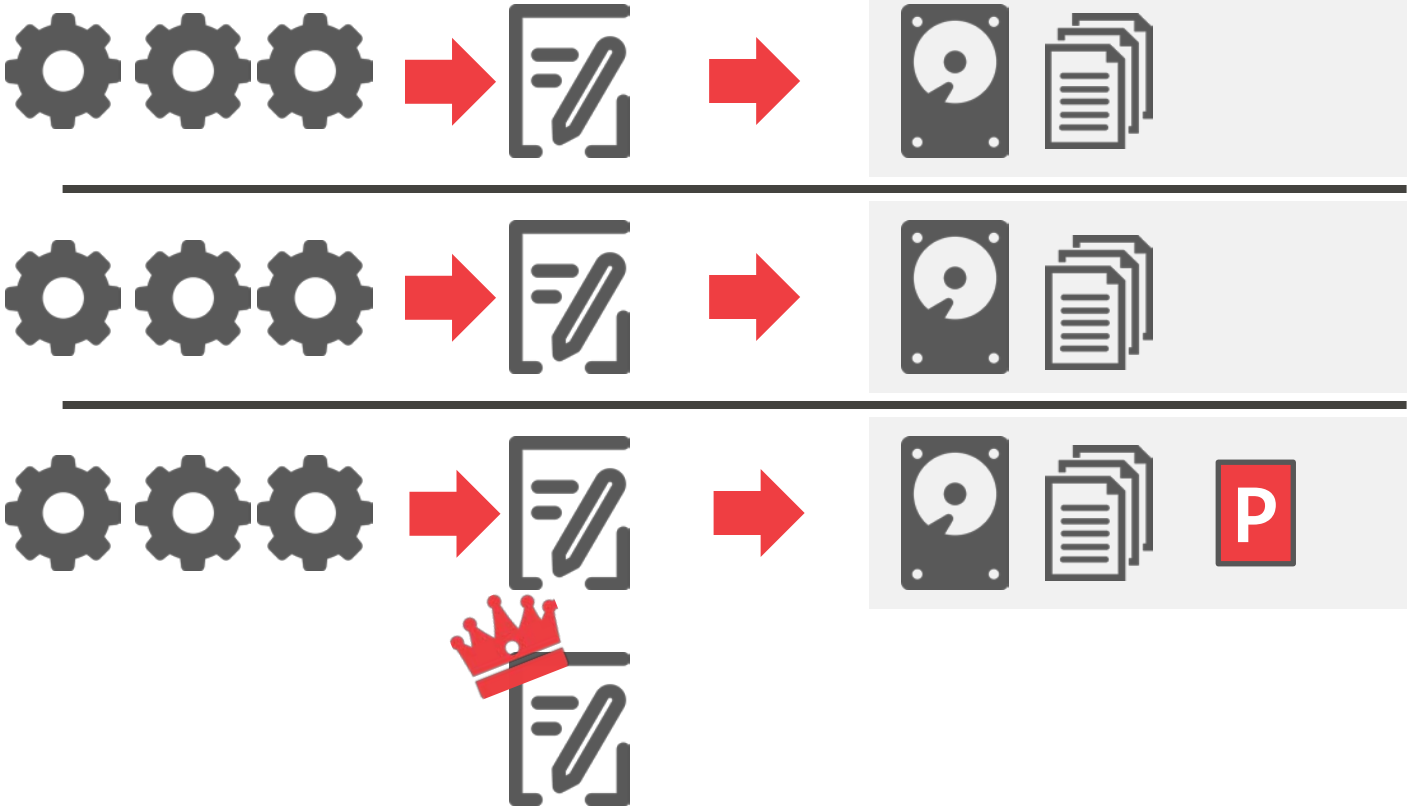
SILOR – ARCHITECTURE





SILOR – ARCHITECTURE

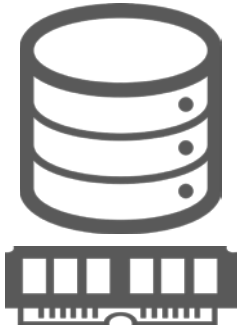


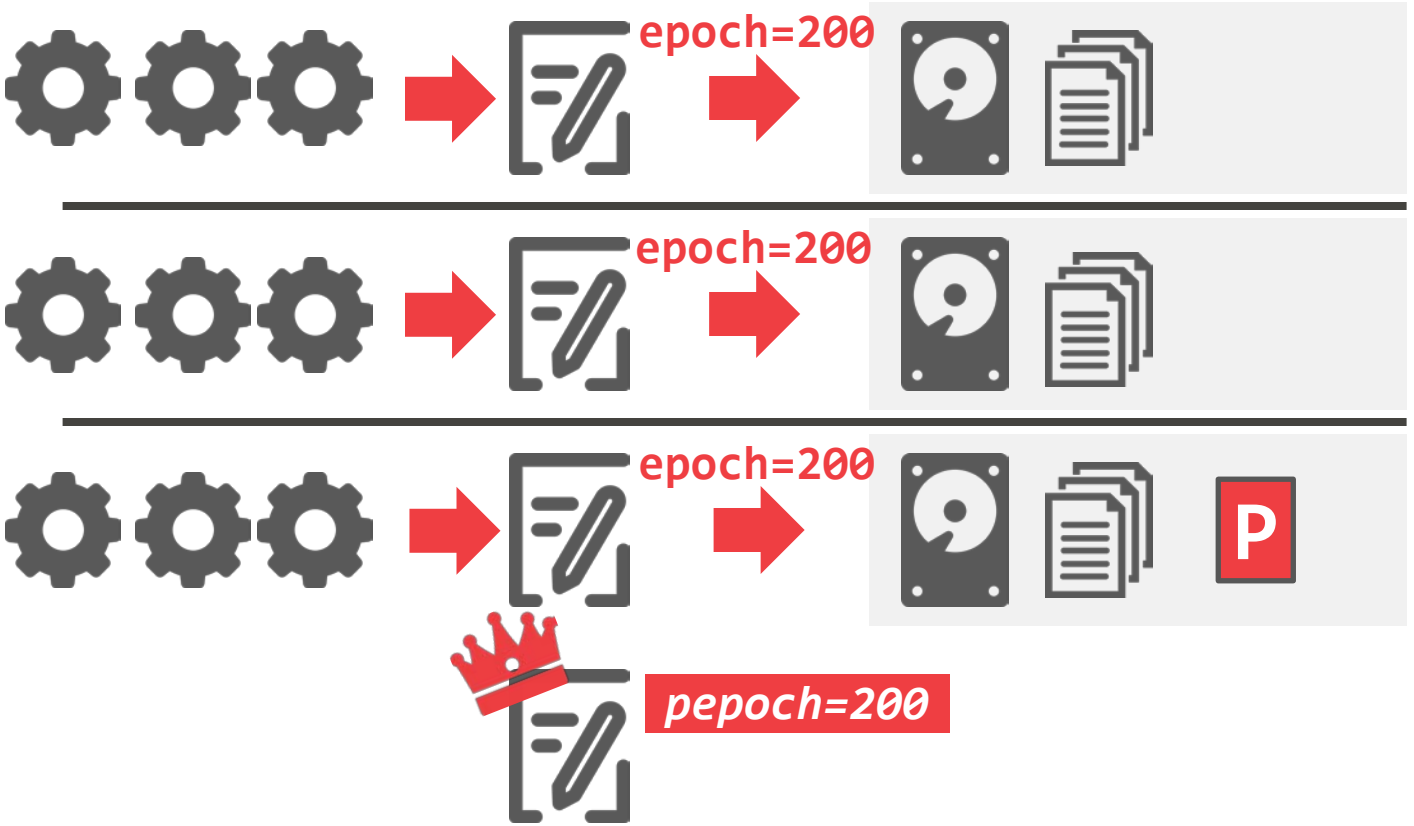


epoch=200

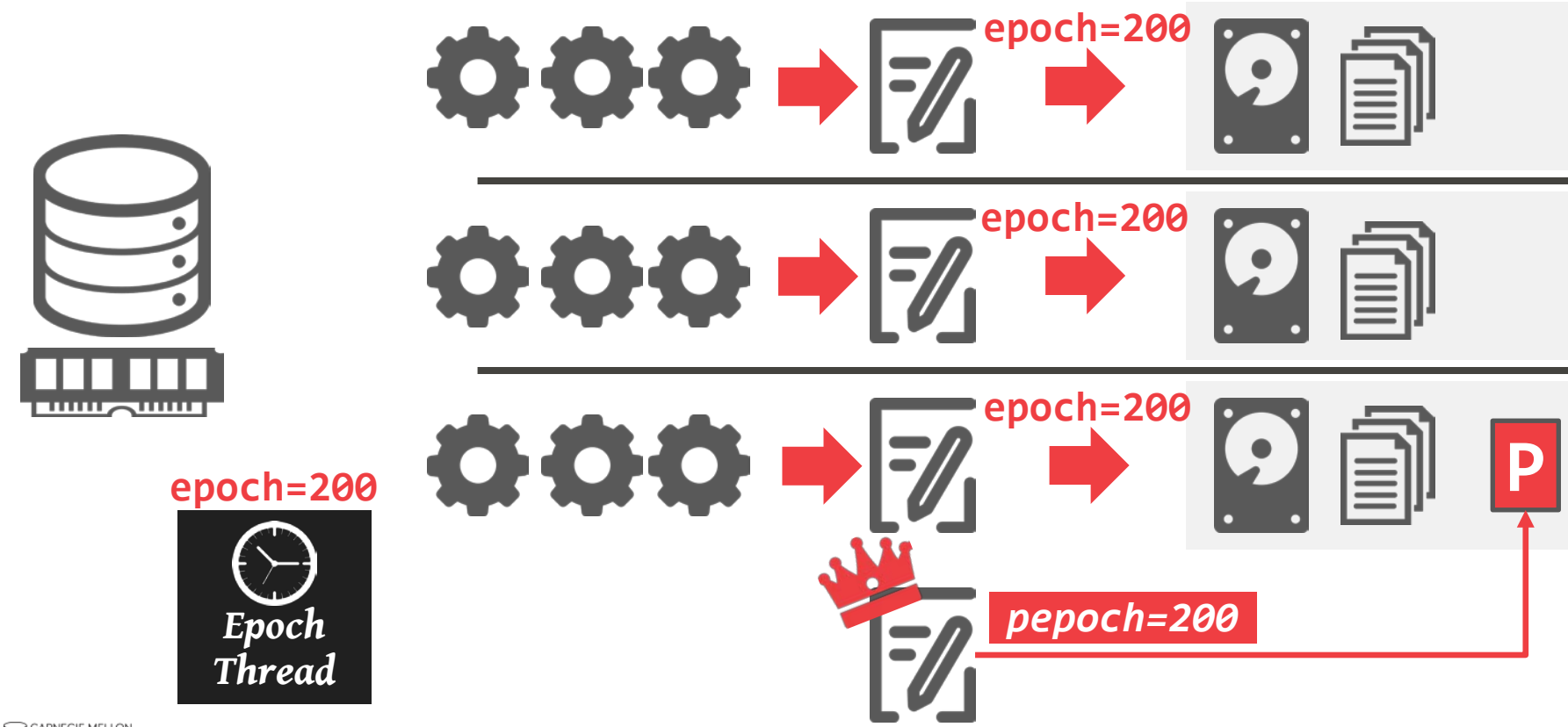
Epoch Thread

SILOR – ARCHITECTURE

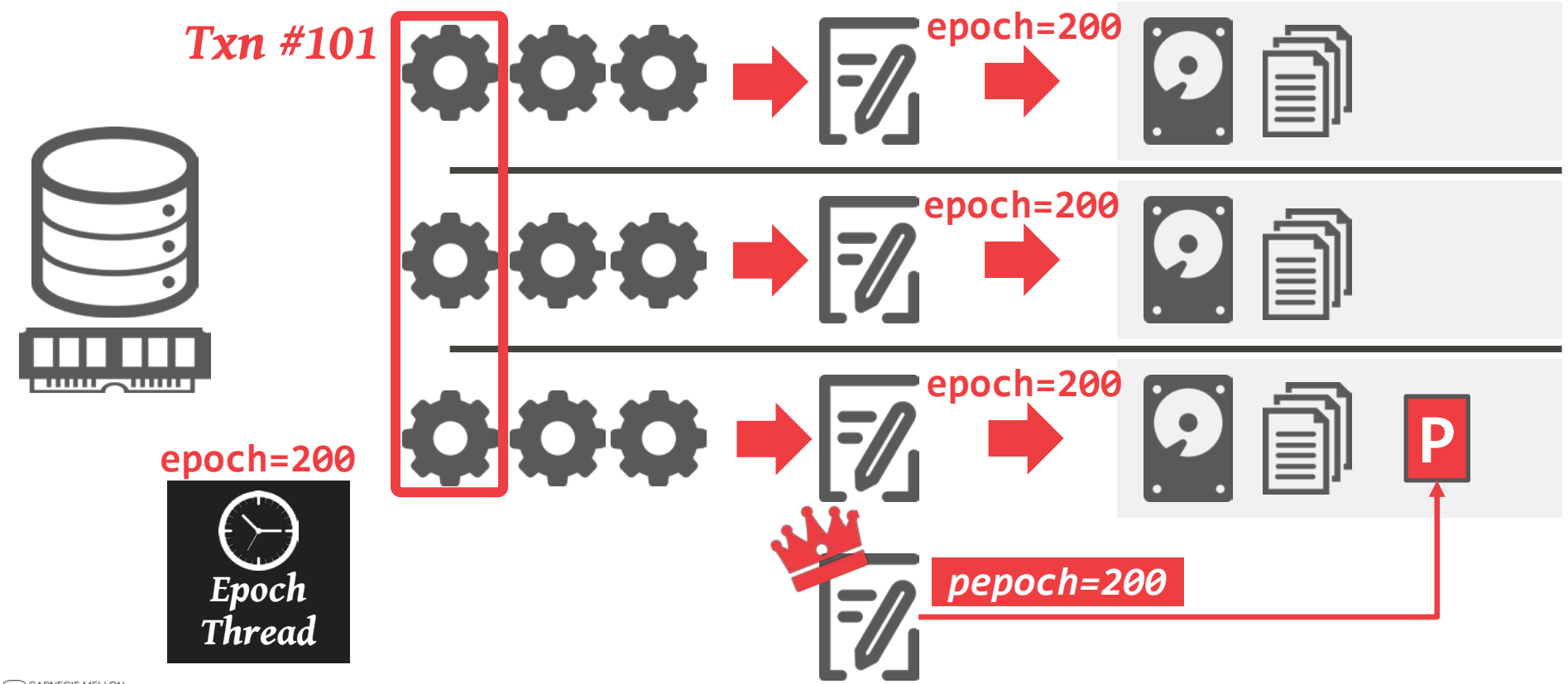




SILOR – ARCHITECTURE



SILOR – ARCHITECTURE



LOGGING SCHEMES

Physical Logging

→ Record the changes made to a specific record in the database.

→ Slower for execution, faster for recovery.

Logical Logging

→ Record the high-level operations executed by txns.

→ Faster for execution, slower for recovery.

OBSERVATION

Node failures in OLTP databases are rare.
→ OLTP databases are not that big.
→ They don't need to run on hundreds of machines.

It's better to optimize the system for runtime operations rather than failure cases.

COMMAND LOGGING

Logical logging scheme where the DBMS only records the stored procedure invocation

→ Stored Procedure Name
→ Input Parameters
→ Additional safety checks

Command Logging = Transaction Logging

RETHINKING MAIN MEMORY OLTP RECOVERY
ICDE 2014

DETERMINISTIC CONCURRENCY CONTROL

For a given state of the database, the execution of a serial schedule will always put the database in the same new state if:

→ The order of txns (or their queries) is defined before they start executing.

→ The txn logic is deterministic.

DETERMINISTIC CONCURRENCY CONTROL

For a given state of the database, the execution of a serial schedule will always put the database in the same new state if:

→ The order of txns (or their queries) is defined before they start executing.

→ The txn logic is deterministic.

DETERMINISTIC CONCURRENCY CONTROL

For a given state of the database, the execution of a serial schedule will always put the database in the same new state if:

→ The order of txns (or their queries) is defined before they start executing.

→ The txn logic is deterministic.

Txn #1 `A = A + 1`

Txn #2 `A = A × 3`

Txn #3 `A = A - 5`

DETERMINISTIC CONCURRENCY CONTROL

For a given state of the database, the execution of a serial schedule will always put the database in the same new state if:

→ The order of txns (or their queries) is defined before they start executing.

→ The txn logic is deterministic.

A=298

Txn #1 `A = A + 1`

Txn #2 `A = A × 3`

Txn #3 `A = A - 5`

DETERMINISTIC CONCURRENCY CONTROL

For a given state of the database, the execution of a serial schedule will always put the database in the same new state if:

→ The order of txns (or their queries) is defined before they start executing.

→ The txn logic is deterministic.

Txn #1 `A = A + 1`

Txn #2 `A = A × NOW()`

Txn #3 `A = A - 5`

DETERMINISTIC CONCURRENCY CONTROL

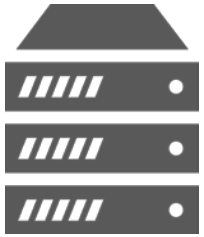
For a given state of the database, the execution of a serial schedule will always put the database in the same new state if:

→ The order of txns (or their queries) is defined before they start executing.

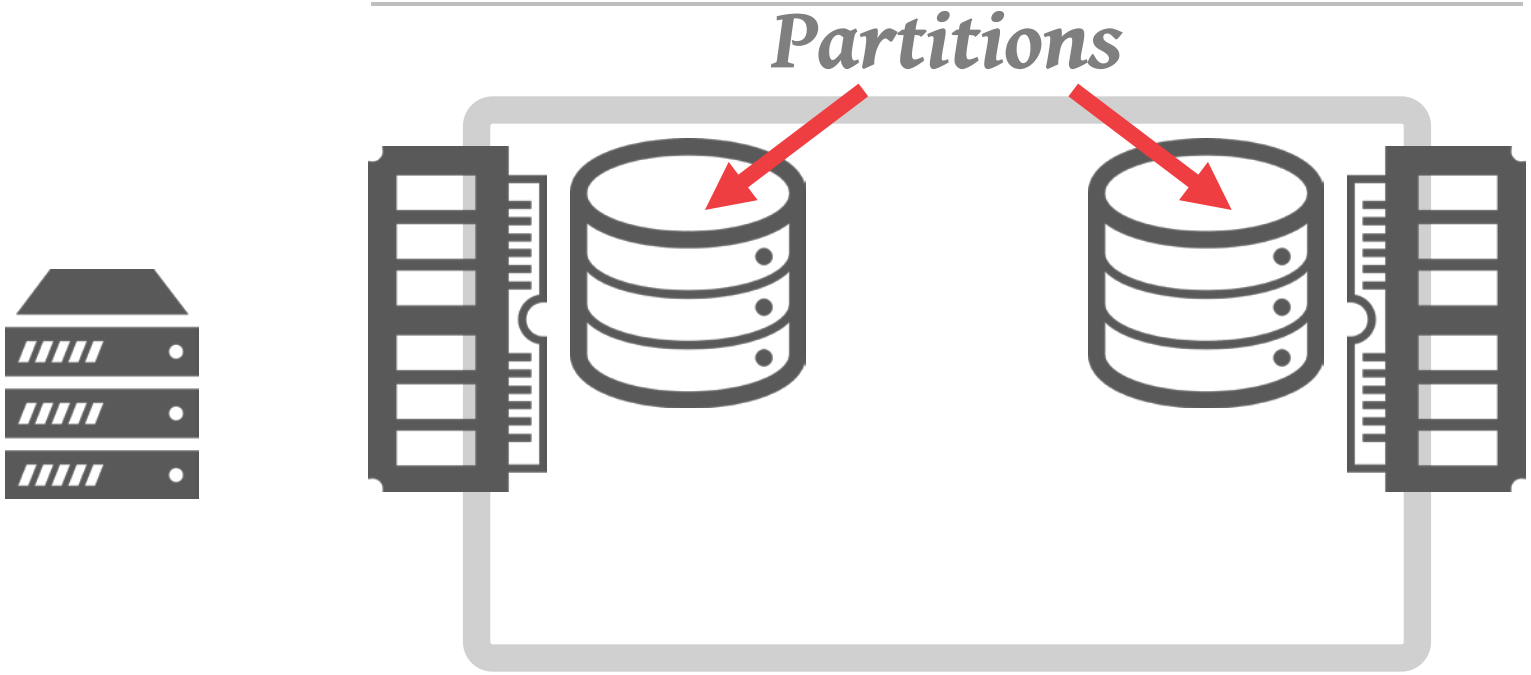
→ The txn logic is deterministic.

Txn #1	`A = A + 1`
Txn #2	`A = A × NOW()`
Txn #3	`A = A - 5`

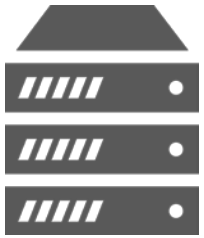
VOLTDB – ARCHITECTURE

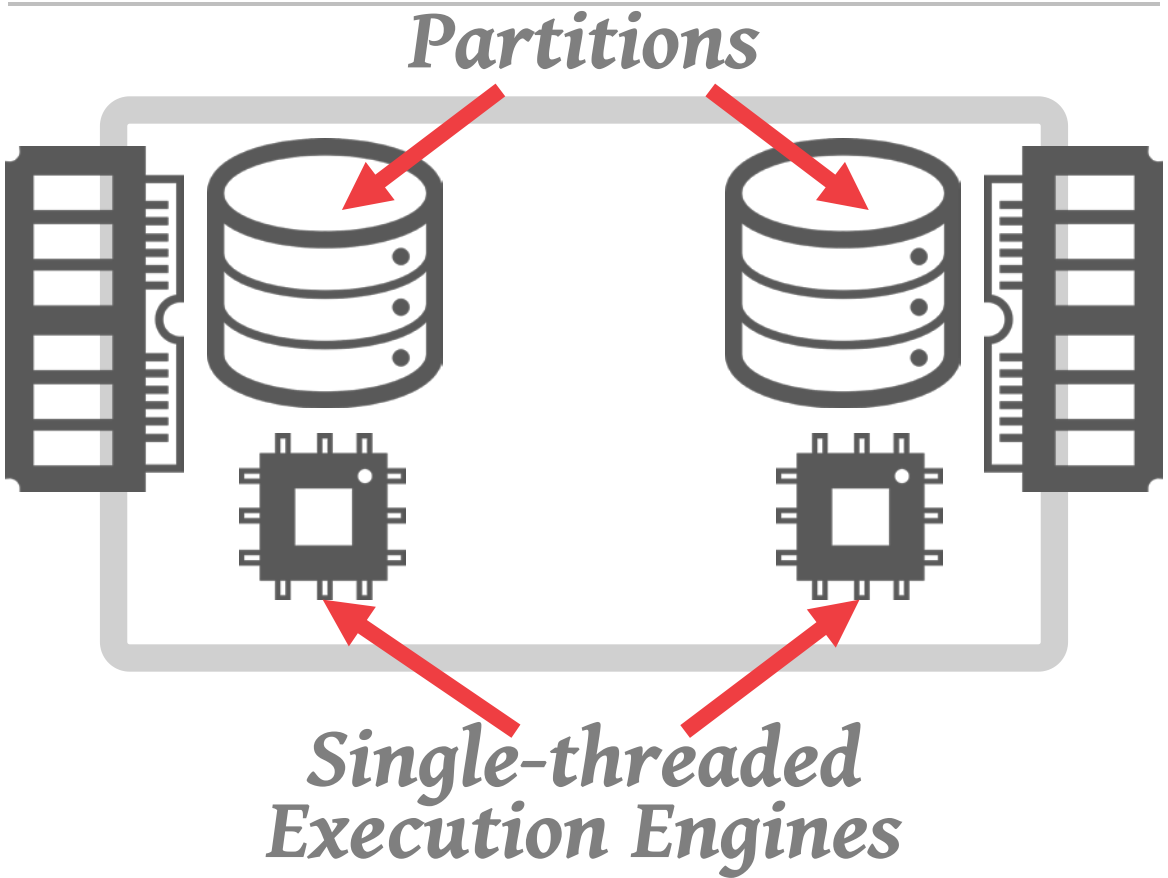


VOLTDB – ARCHITECTURE

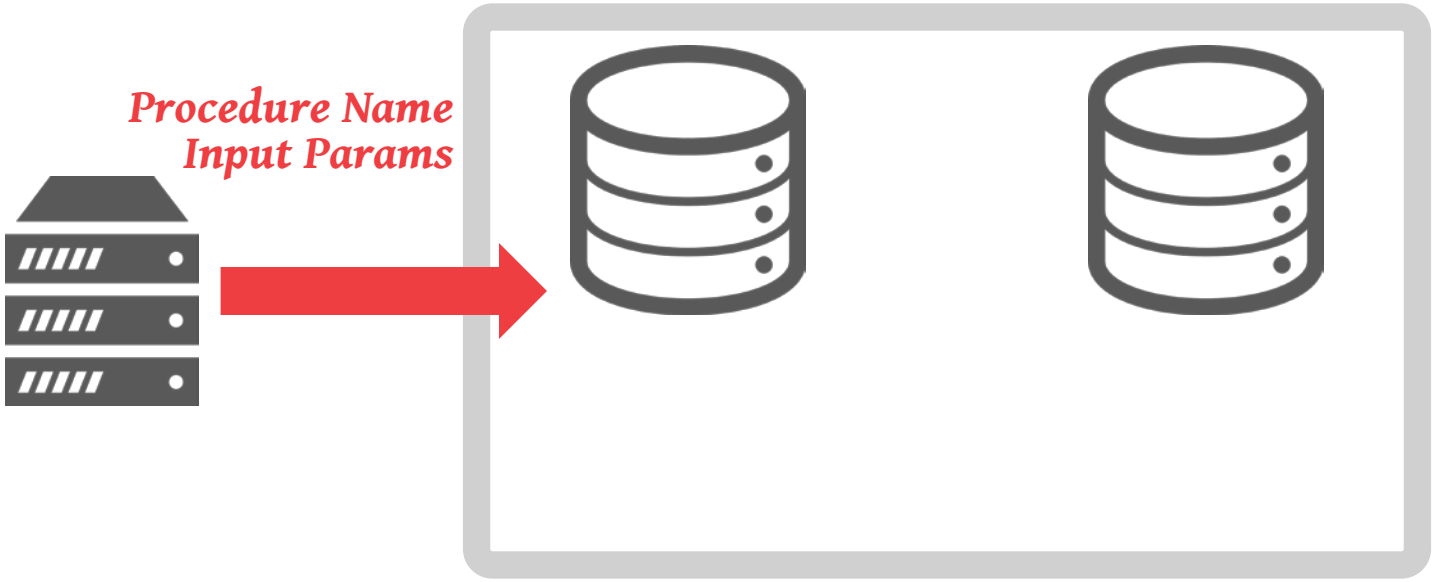


VOLTDB – ARCHITECTURE





VOLTDB – ARCHITECTURE



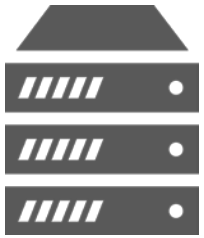
VoteCount:

```
SELECT COUNT(*)
  FROM votes
 WHERE phone_num = ?;
```

InsertVote:

```
INSERT INTO votes
 VALUES (?, ?, ?);
```

Proce
Inp



```
run(phoneNum, contestantId, currentTime) {
  result = execute(VoteCount, phoneNum);
  if (result > MAX_VOTES) {
    return (ERROR);
  }
  execute(InsertVote, phoneNum,
                      contestantId,
                      currentTime);
  return (SUCCESS);
}
```

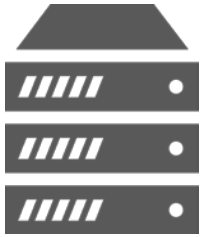
VoteCount:

```
SELECT COUNT(*)
  FROM votes
 WHERE phone_num = ?;
```

InsertVote:

```
INSERT INTO votes
 VALUES (?, ?, ?);
```

Proce
Inp



```
run(phoneNum, contestantId, currentTime) {
  result = execute(VoteCount, phoneNum);
  if (result > MAX_VOTES) {
    return (ERROR);
  }
  execute(InsertVote, phoneNum,
                      contestantId,
                      currentTime);
  return (SUCCESS);
}
```

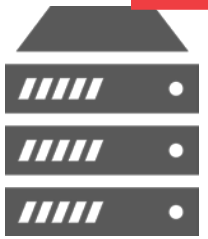
VoteCount:

```
SELECT COUNT(*)
  FROM votes
 WHERE phone_num = ?;
```

InsertVote:

```
INSERT INTO votes
 VALUES (?, ?, ?);
```

Proce



```
run(phoneNum, contestantId, currentTime) {
  result = execute(VoteCount, phoneNum);
  if (result > MAX_VOTES) {
    return (ERROR);
  }
  execute(InsertVote, phoneNum,
                      contestantId,
                      currentTime);
  return (SUCCESS);
}
```

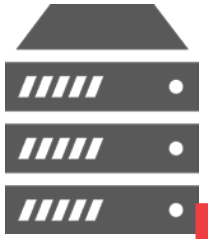
VoteCount:

```
SELECT COUNT(*)
  FROM votes
 WHERE phone_num = ?;
```

InsertVote:

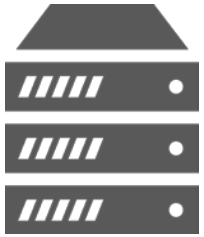
```
INSERT INTO votes
 VALUES (?, ?, ?);
```

Proce
Inp

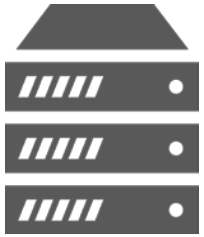


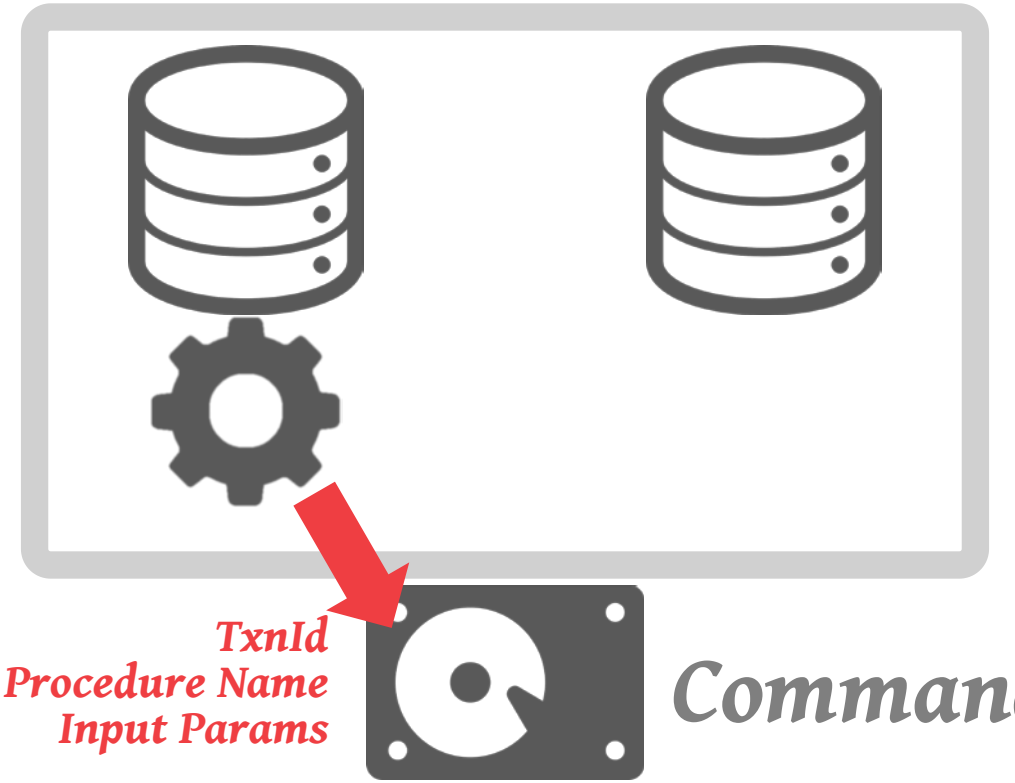
```
run(phoneNum, contestantId, currentTime) {
  result = execute(VoteCount, phoneNum);
  if (result > MAX_VOTES) {
    return (ERROR);
  }
  execute(InsertVote, phoneNum,
                      contestantId,
                      currentTime);
  return (SUCCESS);
}
```

VOLTDB – ARCHITECTURE

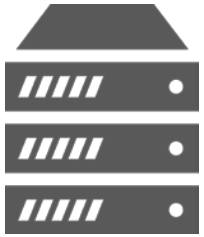


VOLTDB – ARCHITECTURE

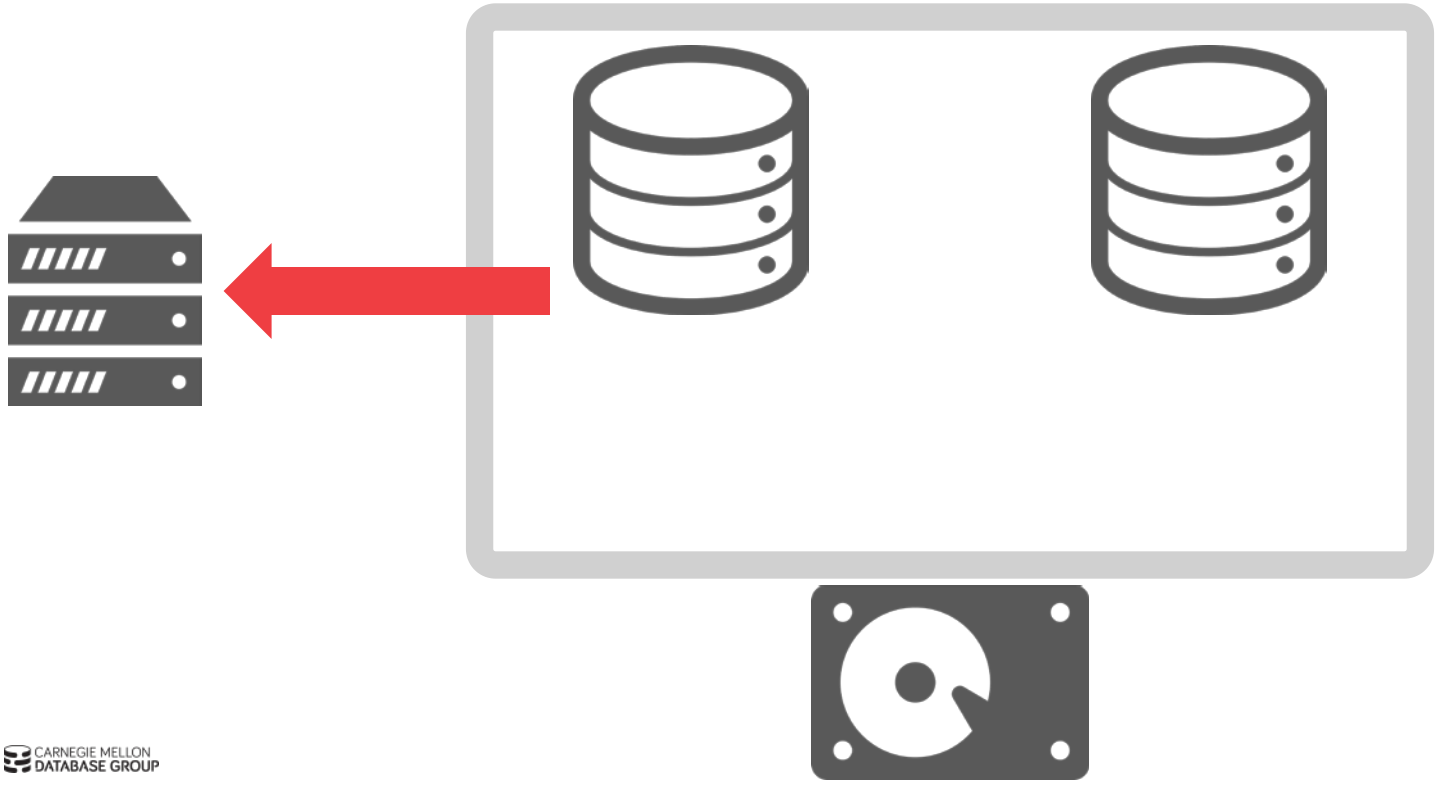




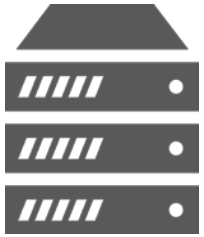
VOLTDB – ARCHITECTURE

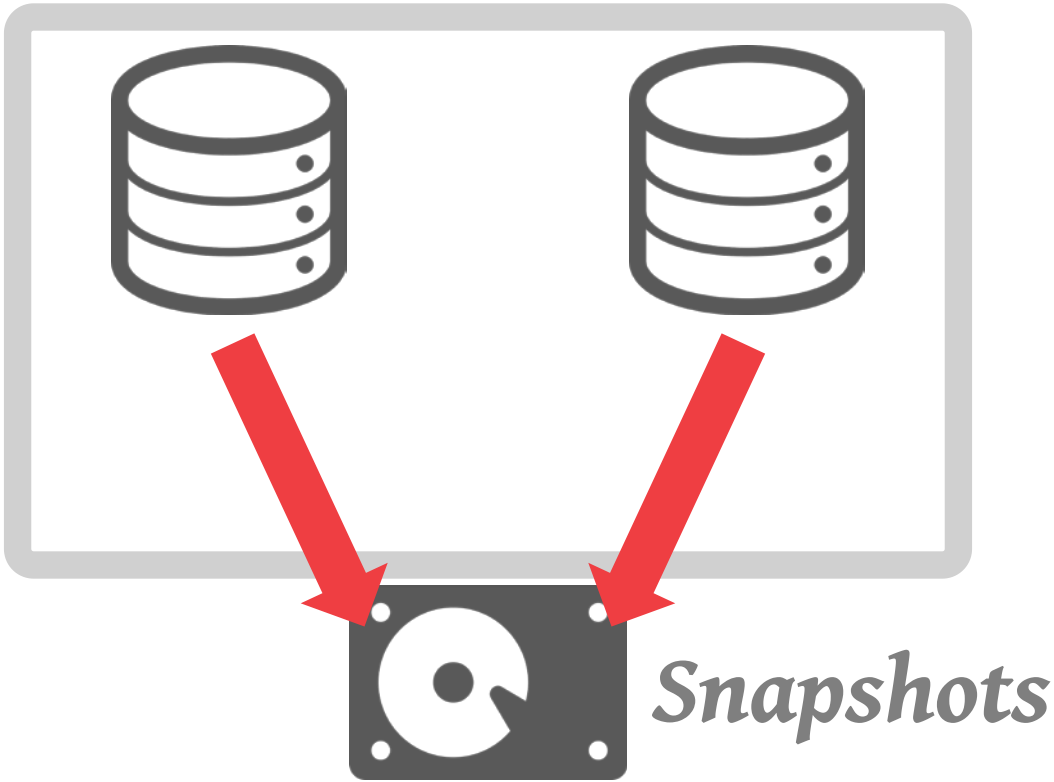


VOLTDB – ARCHITECTURE



VOLTDB – ARCHITECTURE





VOLTDB – LOGGING PROTOCOL

The DBMS logs the txn command **before** it starts executing once a txn has been assigned its serial order.

The node with the txn's "base partition" is responsible for writing the log record.
→ Remote partitions do not log anything.
→ Replica nodes have to log just like their master.

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.

VOLTDB – RECOVERY PROTOCOL

The DBMS loads in the last complete checkpoint from disk.

Nodes then re-execute all of the txns in the log that arrived after the checkpoint started.

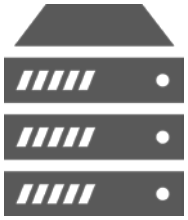
→ The amount of time elapsed since the last checkpoint in the log determines how long recovery will take.

→ Txns that are aborted the first still have to be executed.

VOLTDB – REPLICATION

Executing a deterministic txn on the multiple copies of the same database in the same order provides strongly consistent replicas.

→ DBMS does not need to use **Two-Phase Commit**



Master

Replica

VOLTDB – REPLICATION

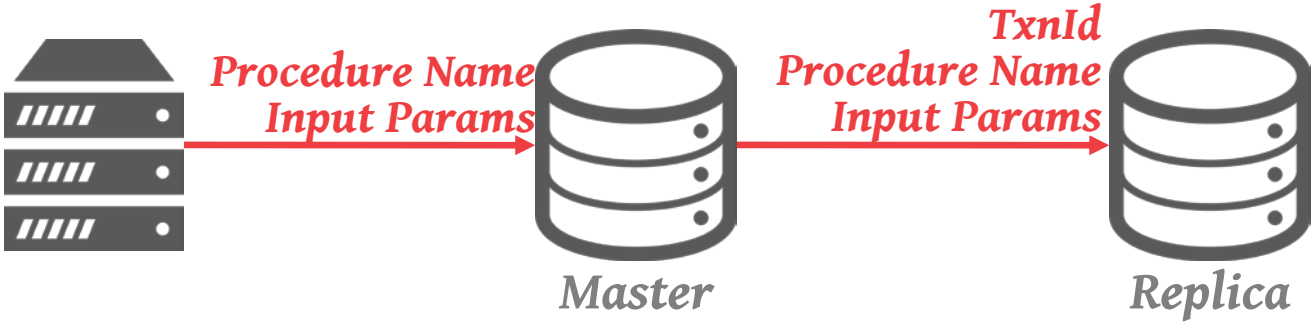
Executing a deterministic txn on the multiple copies of the same database in the same order provides strongly consistent replicas.

→ DBMS does not need to use **Two-Phase Commit**

VOLTDB – REPLICATION

Executing a deterministic txn on the multiple copies of the same database in the same order provides strongly consistent replicas.

→ DBMS does not need to use **Two-Phase Commit**



VOLTDB – REPLICATION

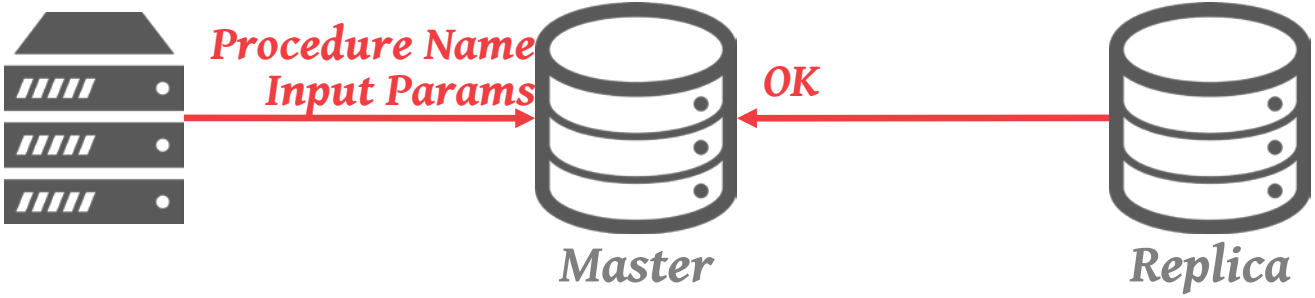
Executing a deterministic txn on the multiple copies of the same database in the same order provides strongly consistent replicas.

→ DBMS does not need to use **Two-Phase Commit**

VOLTDB – REPLICATION

Executing a deterministic txn on the multiple copies of the same database in the same order provides strongly consistent replicas.

→ DBMS does not need to use **Two-Phase Commit**



PROBLEMS WITH COMMAND LOGGING

If the log contains multi-node txns, then if one node goes down and there are no more replicas, then the entire DBMS has to restart.

PROBLEMS WITH COMMAND LOGGING

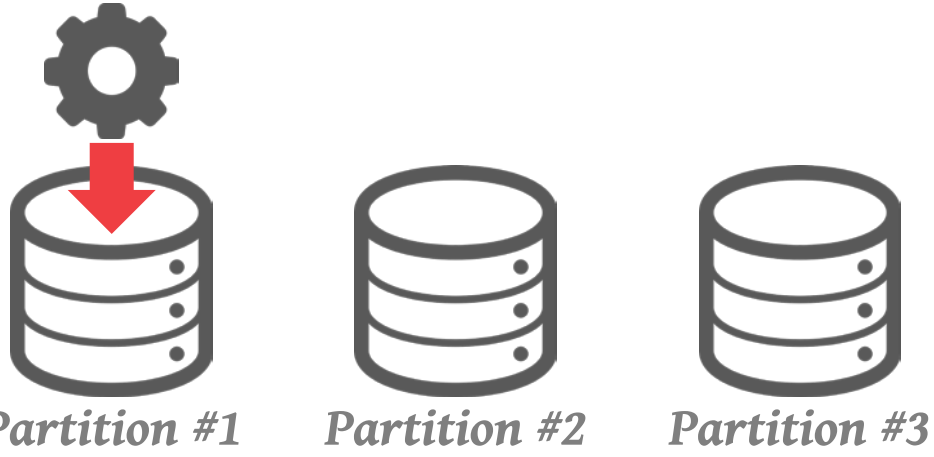
If the log contains multi-node txns, then if one node goes down and there are no more replicas, then the entire DBMS has to restart.

```
X ← SELECT X FROM P2
if (X == true) {
  Y ← UPDATE P2 SET Y = Y+1
} else {
  Y ← UPDATE P3 SET Y = Y+1
}
return (Y)
```

PROBLEMS WITH COMMAND LOGGING

If the log contains multi-node txns, then if one node goes down and there are no more replicas, then the entire DBMS has to restart.

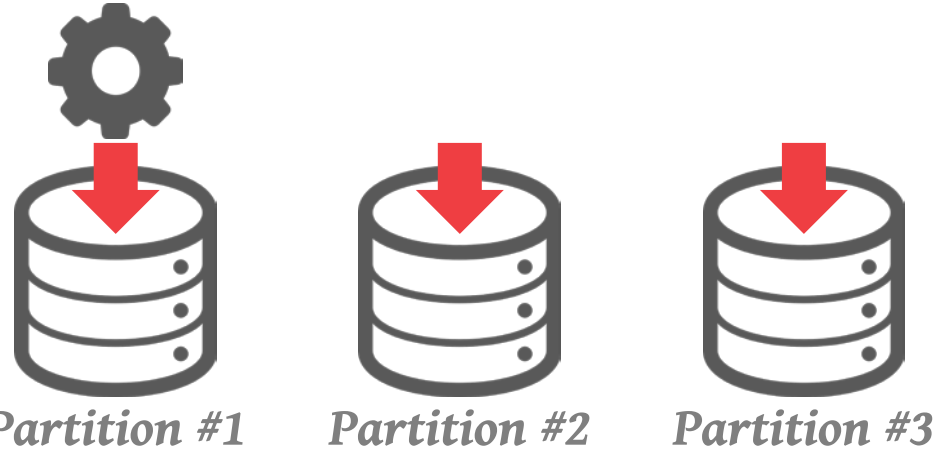
```
X ← SELECT X FROM P2
if (X == true) {
  Y ← UPDATE P2 SET Y = Y+1
} else {
  Y ← UPDATE P3 SET Y = Y+1
}
return (Y)
```



PROBLEMS WITH COMMAND LOGGING

If the log contains multi-node txns, then if one node goes down and there are no more replicas, then the entire DBMS has to restart.

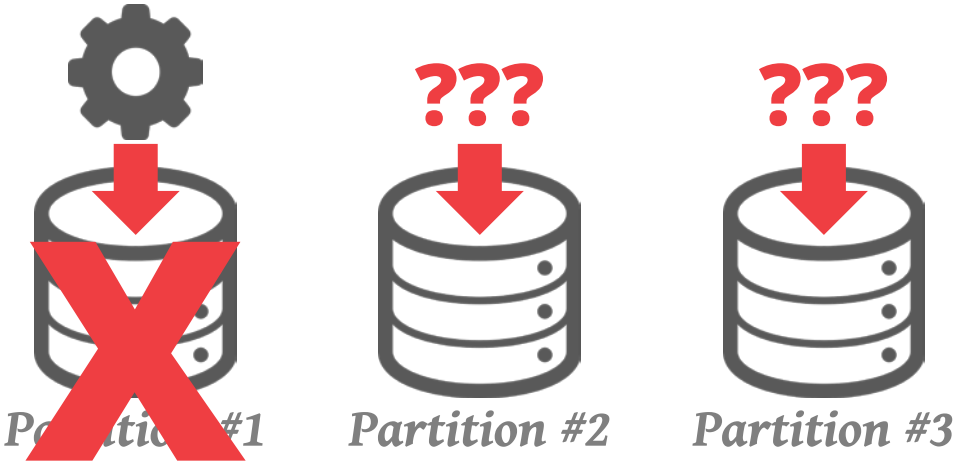
```
X ← SELECT X FROM P2
if (X == true) {
  Y ← UPDATE P2 SET Y = Y+1
} else {
  Y ← UPDATE P3 SET Y = Y+1
}
return (Y)
```



PROBLEMS WITH COMMAND LOGGING

If the log contains multi-node txns, then if one node goes down and there are no more replicas, then the entire DBMS has to restart.

```
X ← SELECT X FROM P2
if (X == true) {
  Y ← UPDATE P2 SET Y = Y+1
} else {
  Y ← UPDATE P3 SET Y = Y+1
}
return (Y)
```



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.

→ If the DBMS provides access to each table's heap then the thread completely ignores indexes.

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.

The copying does not need to be explicit if you fork the DBMS process.

→ Checkpoint is consistent if there are not active txns.

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

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

OBSERVATION

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

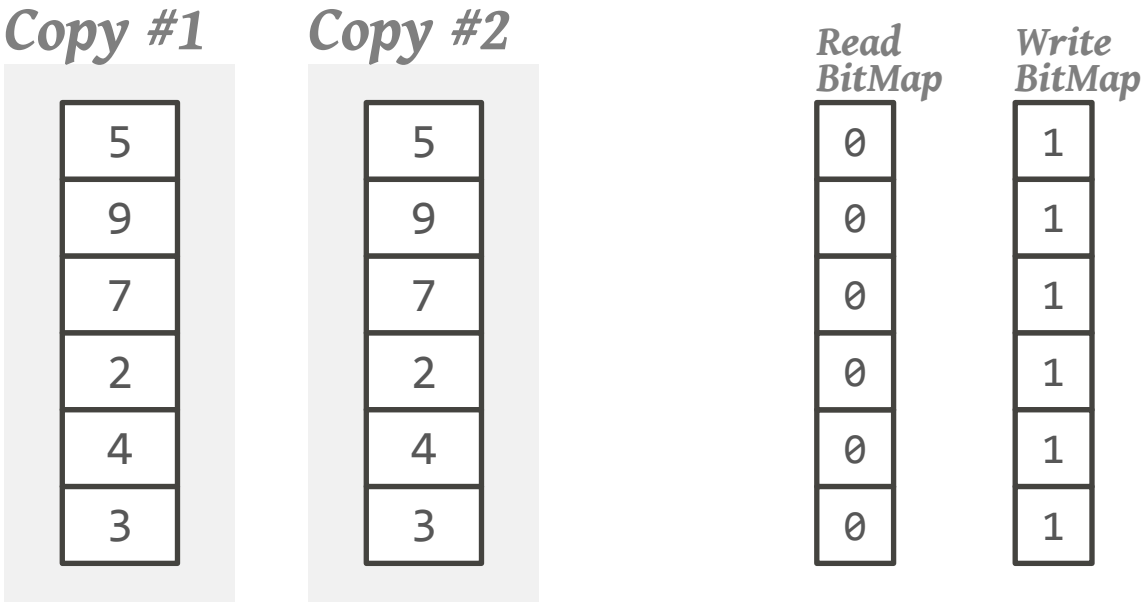
Txns may have to wait to acquire latches held by the checkpoint thread under copy-on-update

WAIT-FREE ZIGZAG

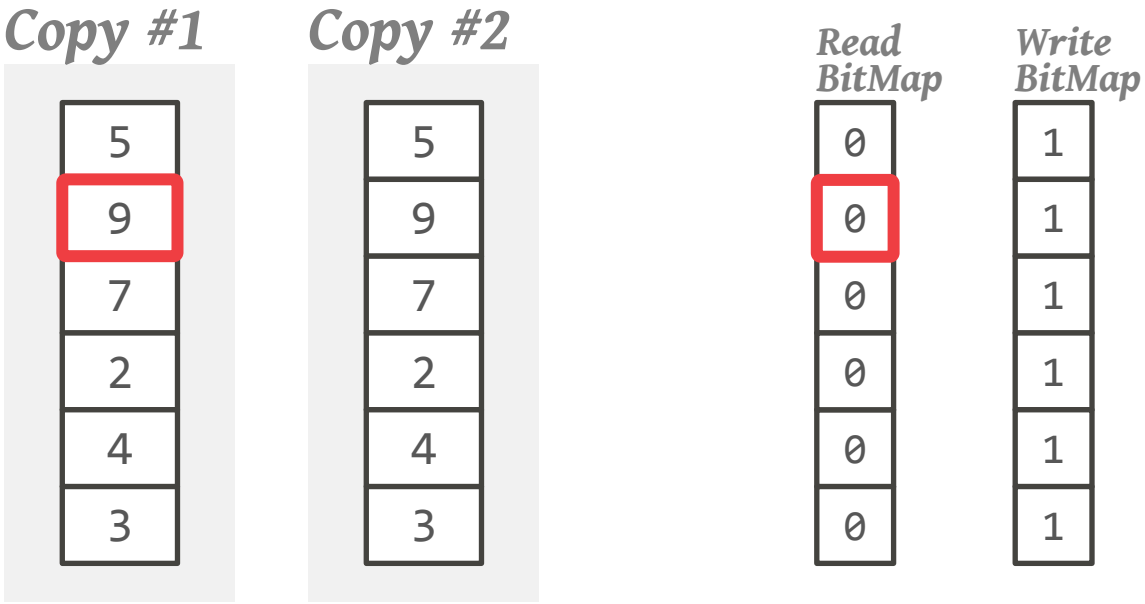
Maintain two copies of the entire database
→ Each txn write only updates one copy.

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.

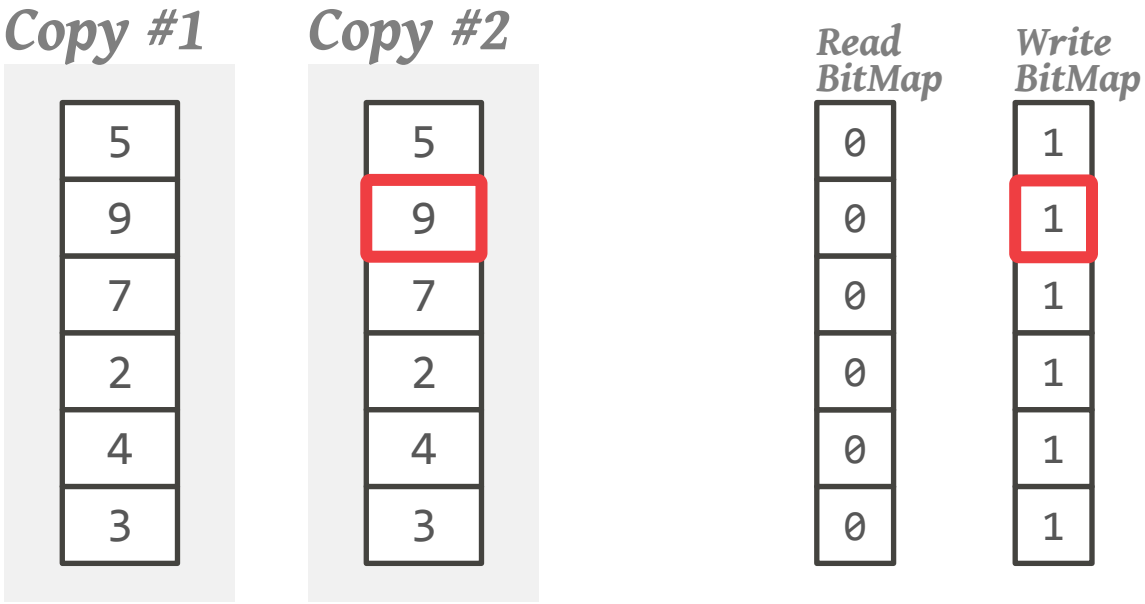
WAIT-FREE ZIGZAG



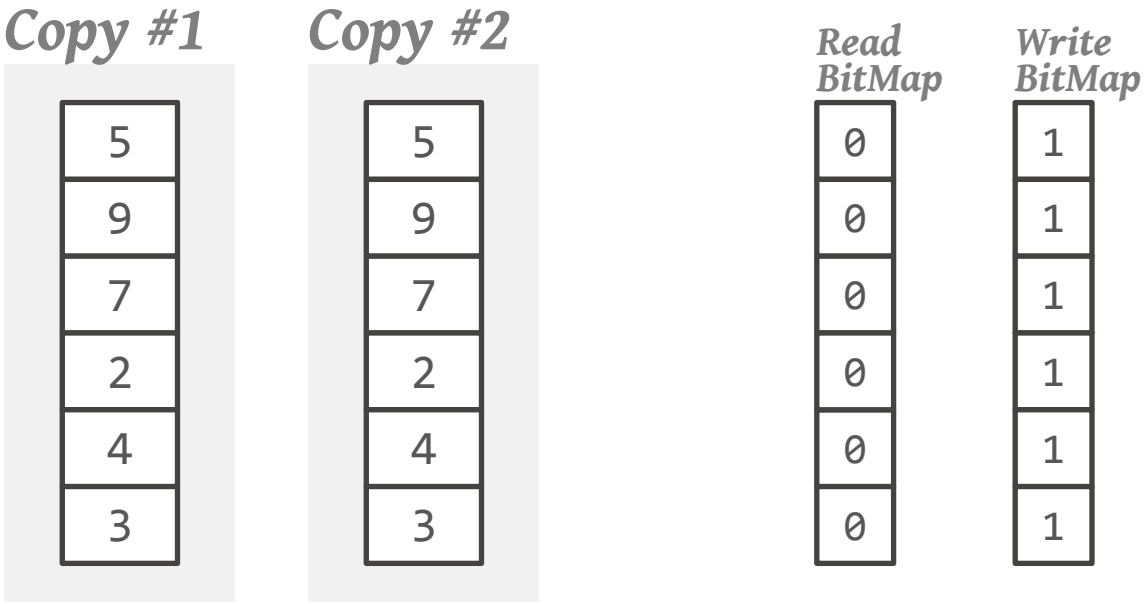
WAIT-FREE ZIGZAG



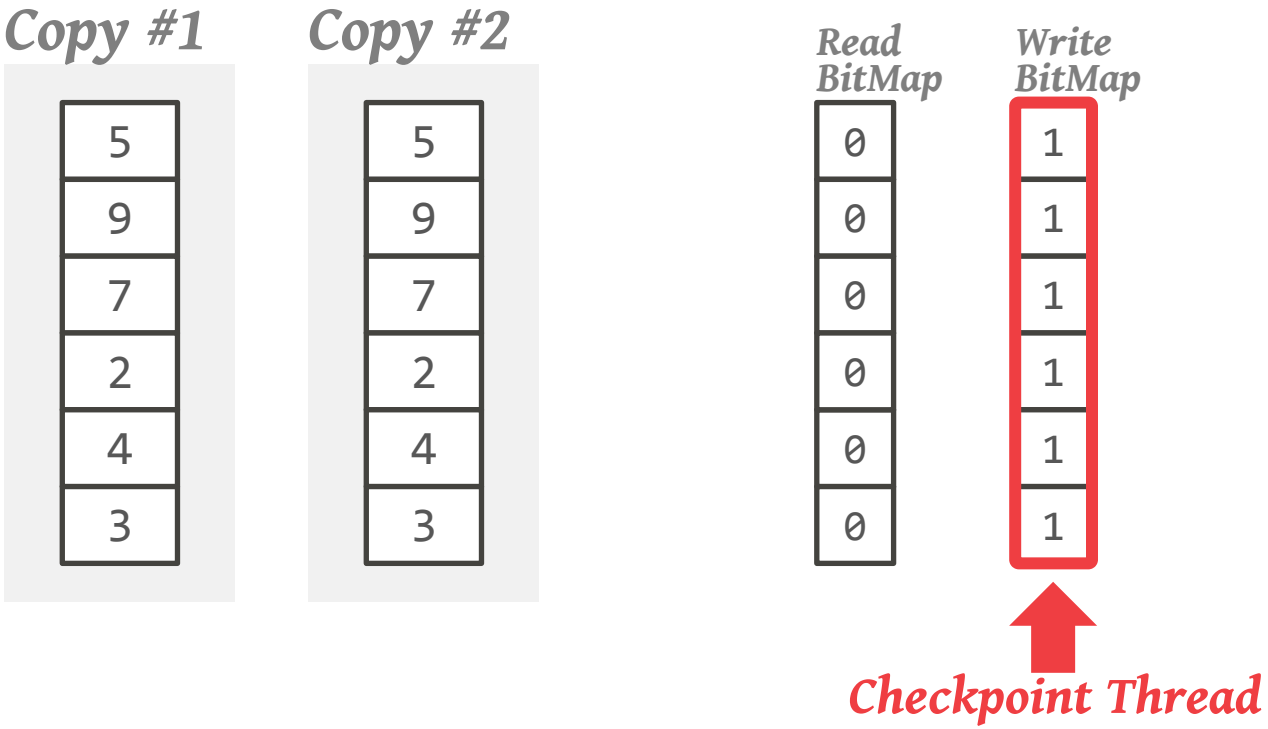
WAIT-FREE ZIGZAG



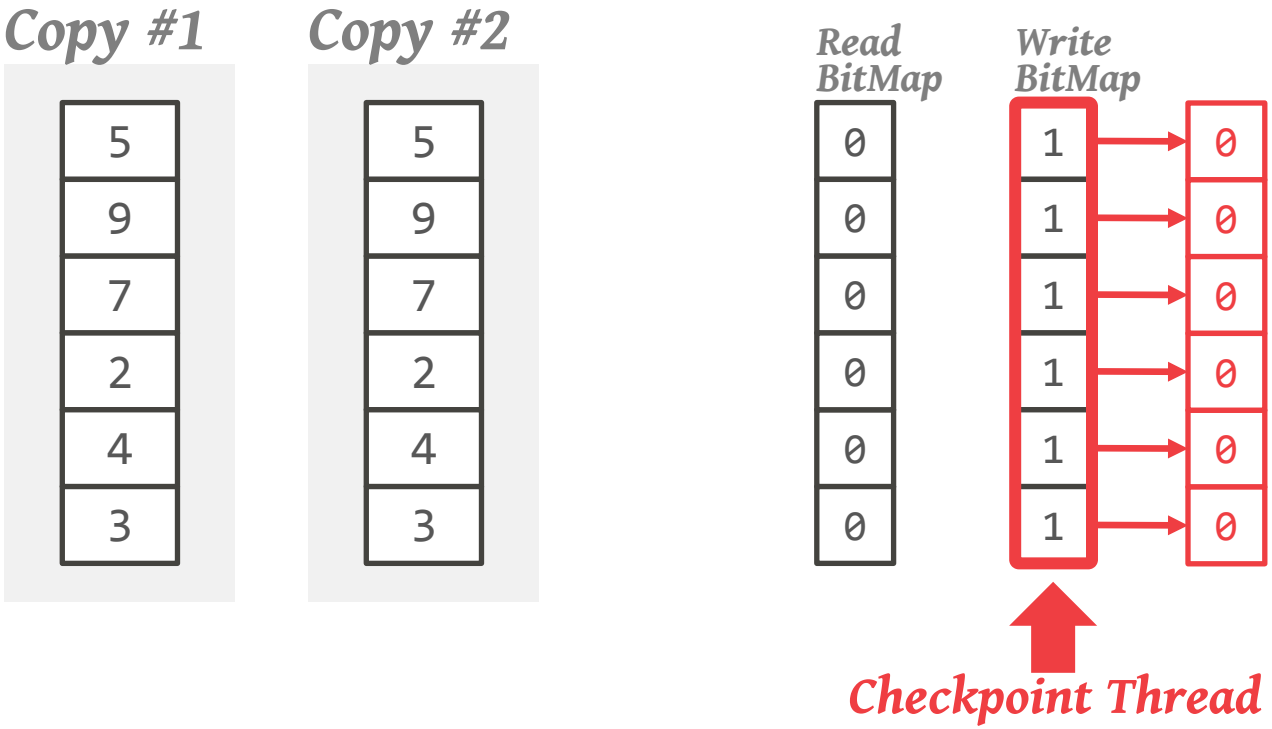
WAIT-FREE ZIGZAG



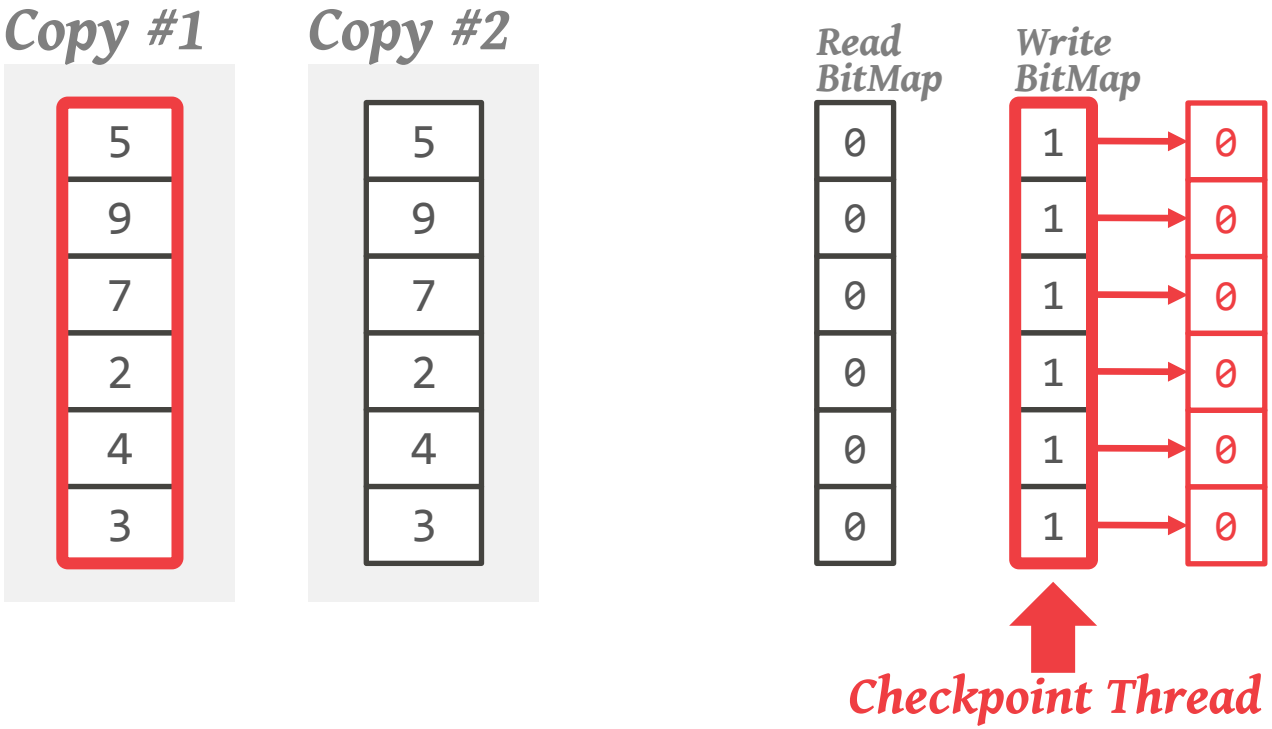
WAIT-FREE ZIGZAG



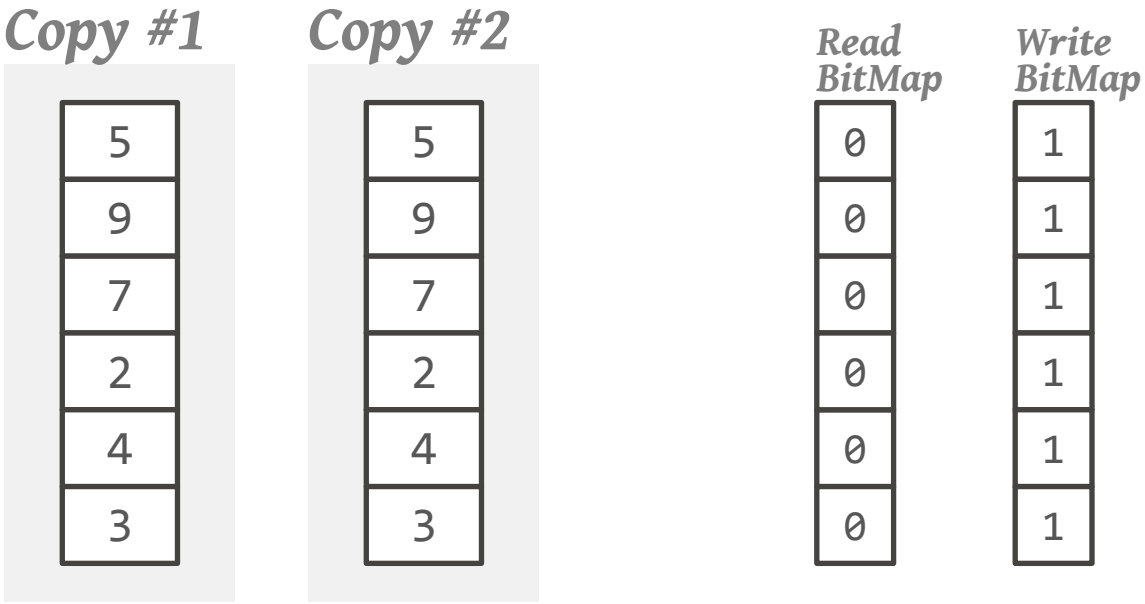
WAIT-FREE ZIGZAG



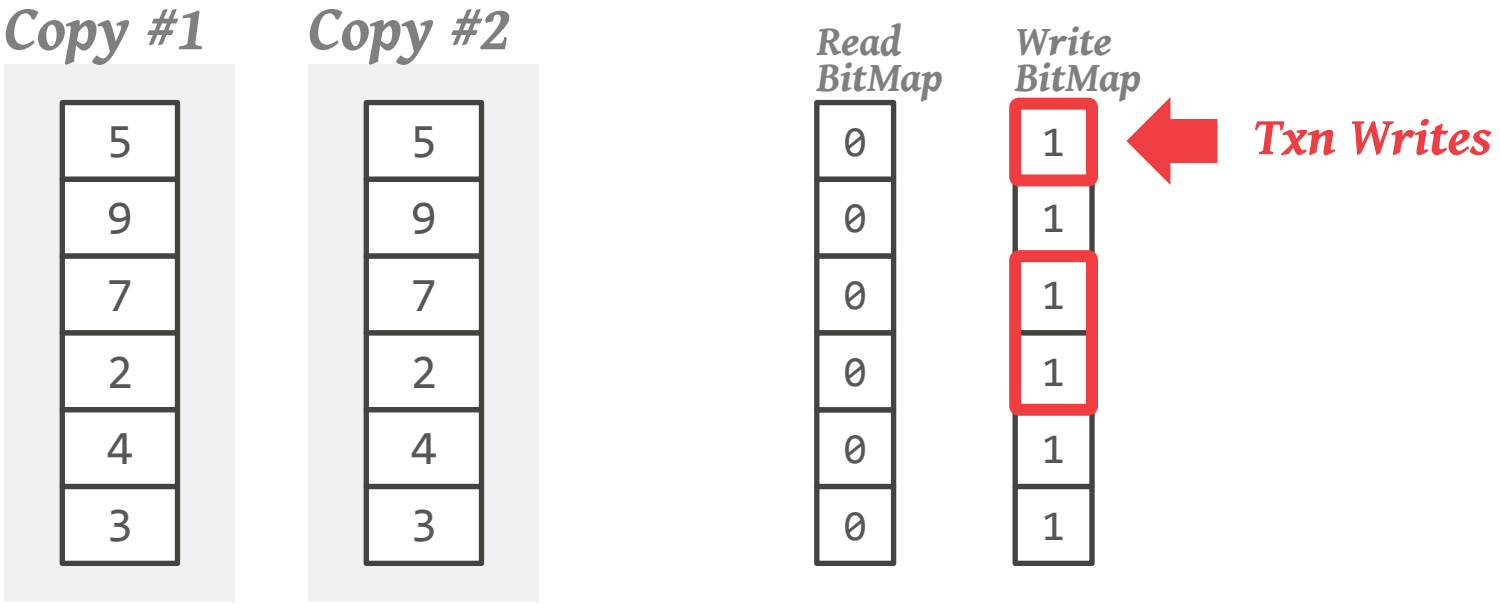
WAIT-FREE ZIGZAG



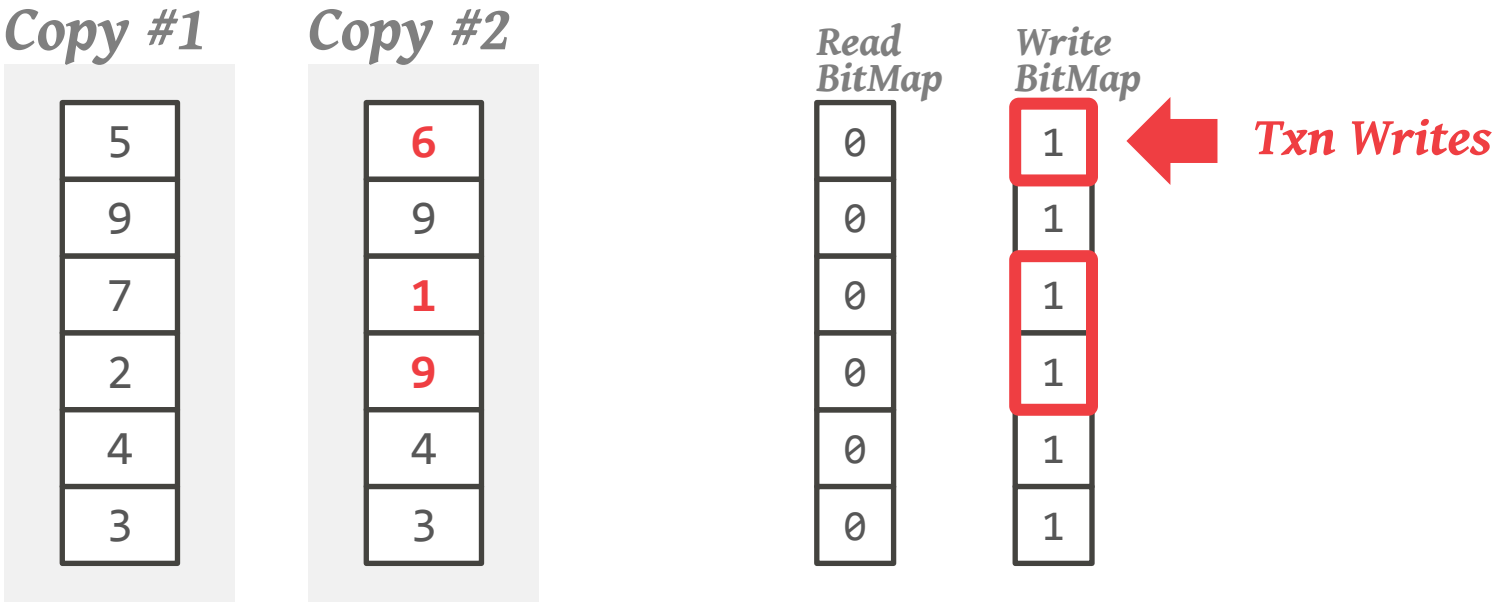
WAIT-FREE ZIGZAG



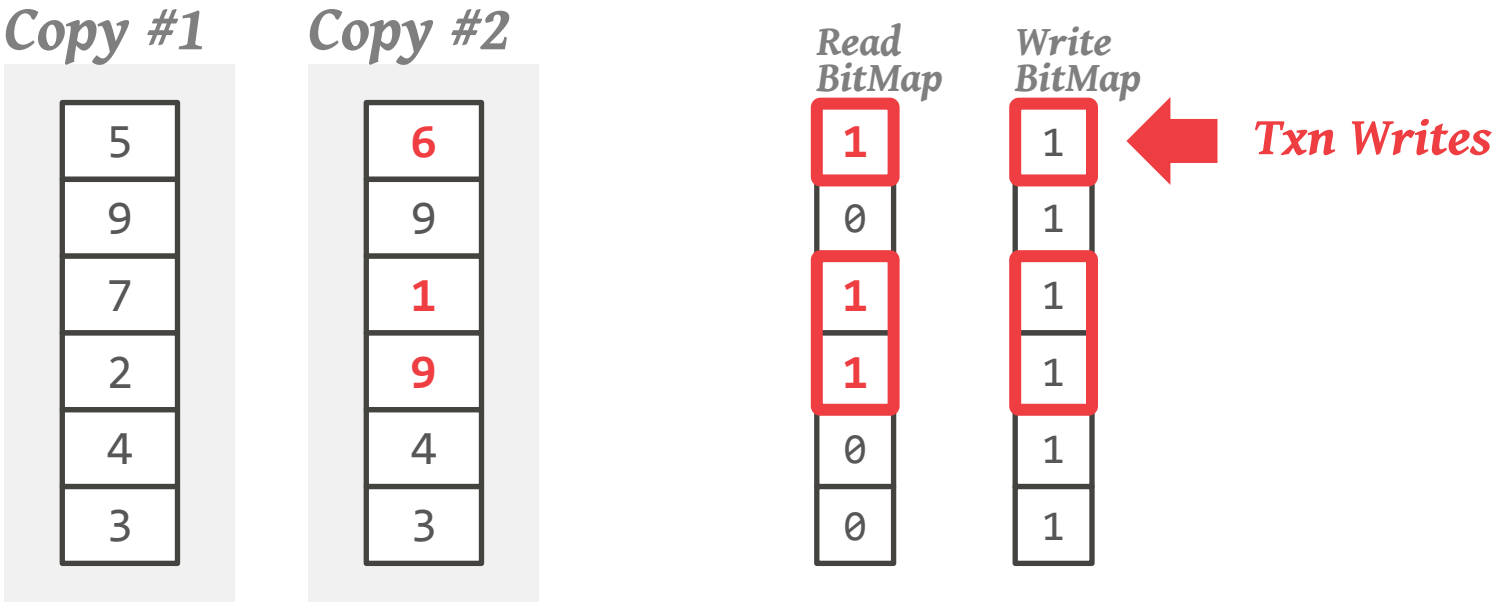
WAIT-FREE ZIGZAG



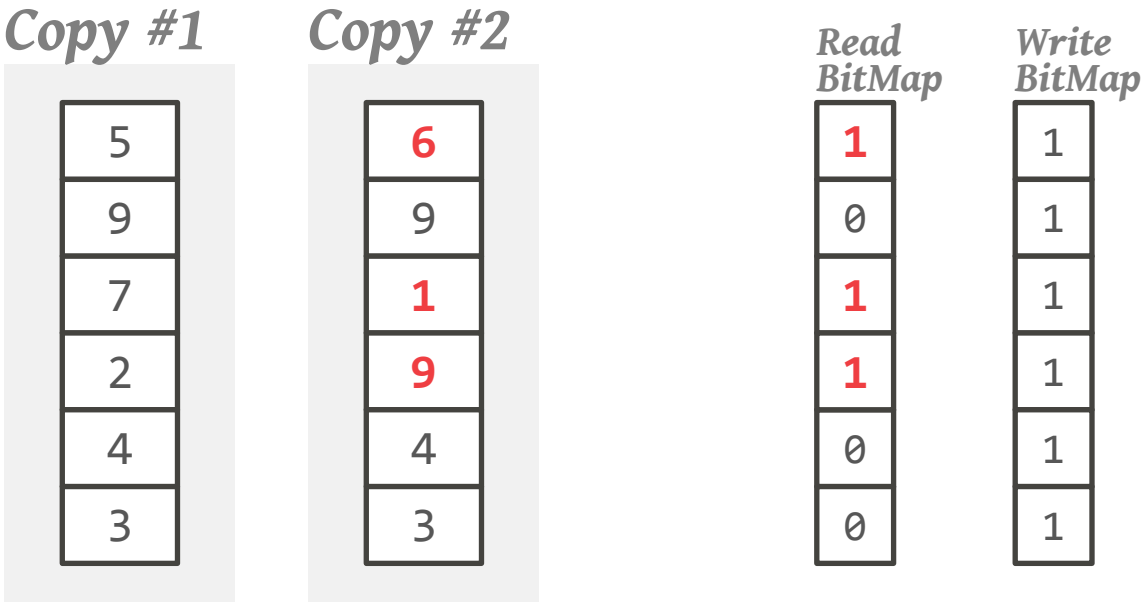
WAIT-FREE ZIGZAG



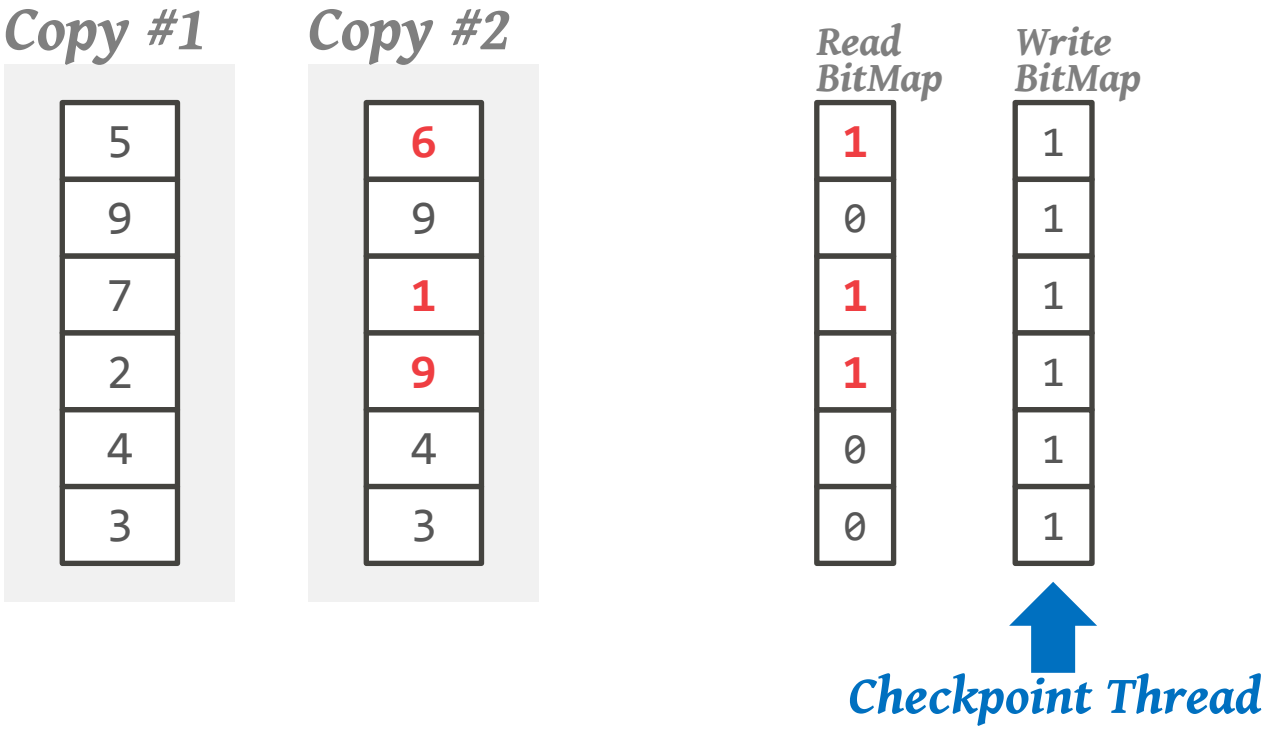
WAIT-FREE ZIGZAG



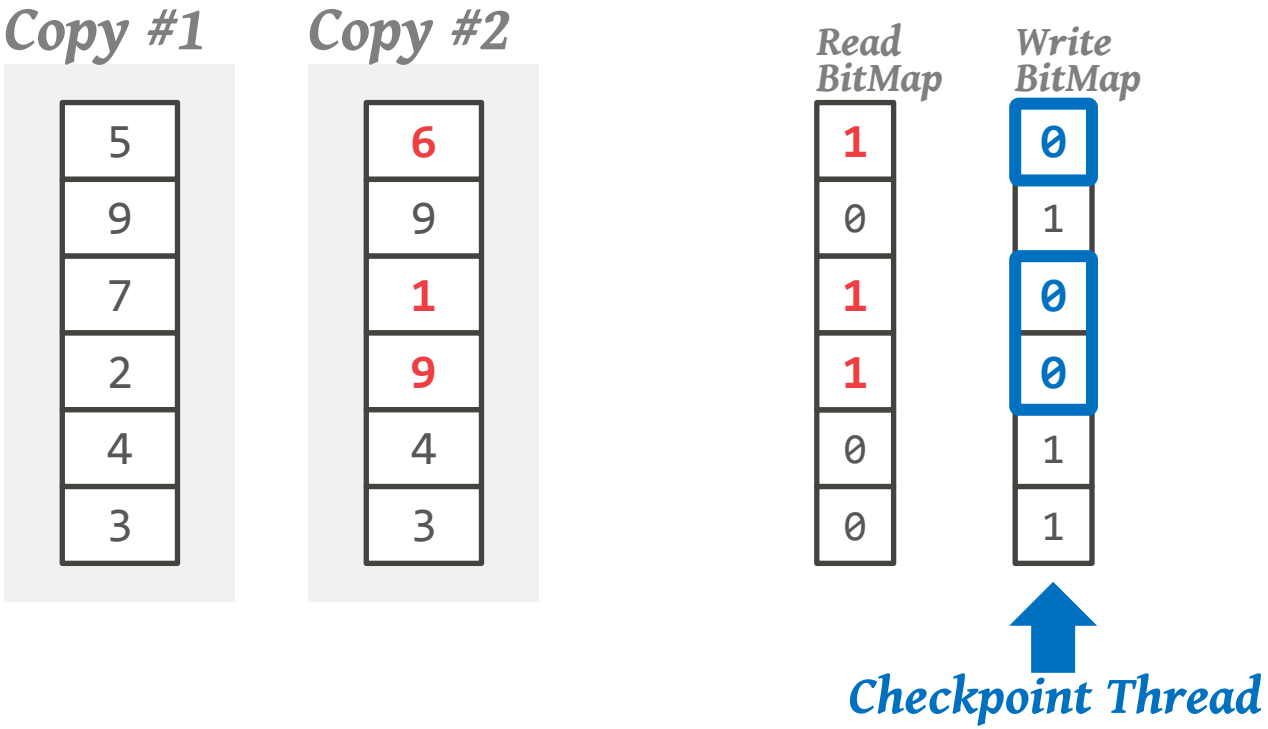
WAIT-FREE ZIGZAG



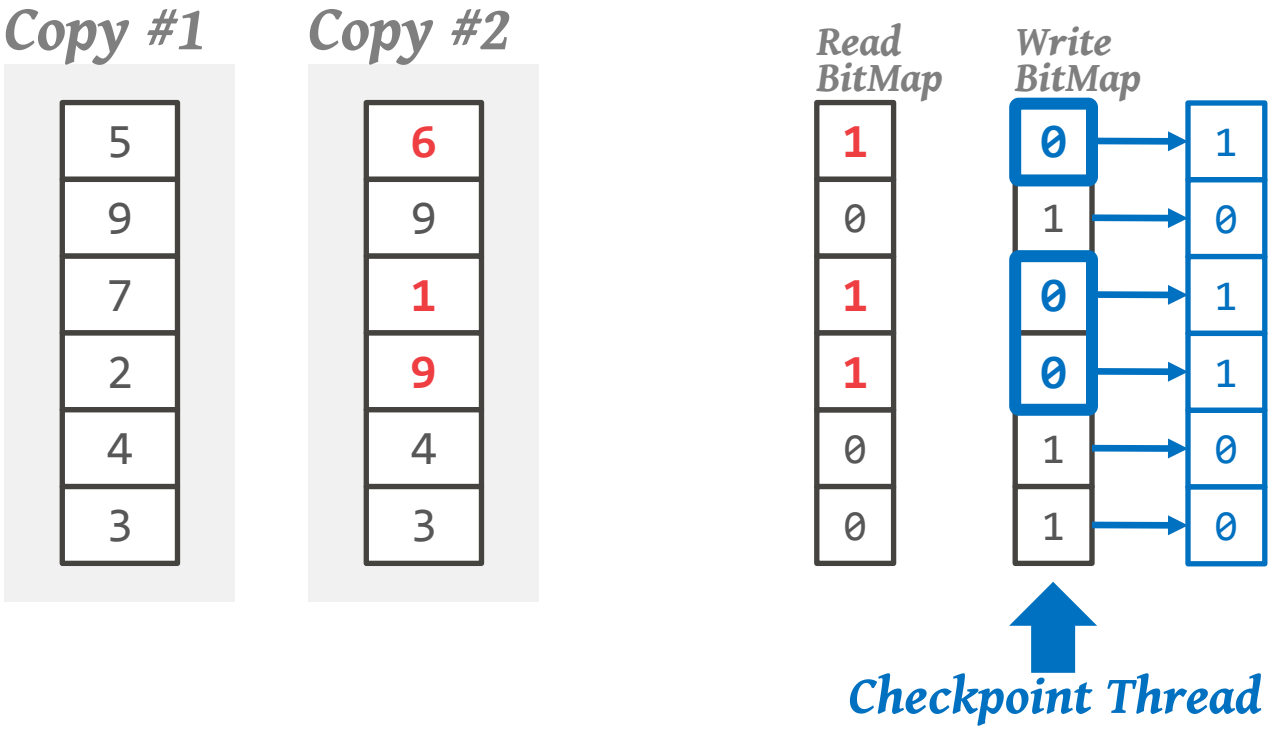
WAIT-FREE ZIGZAG



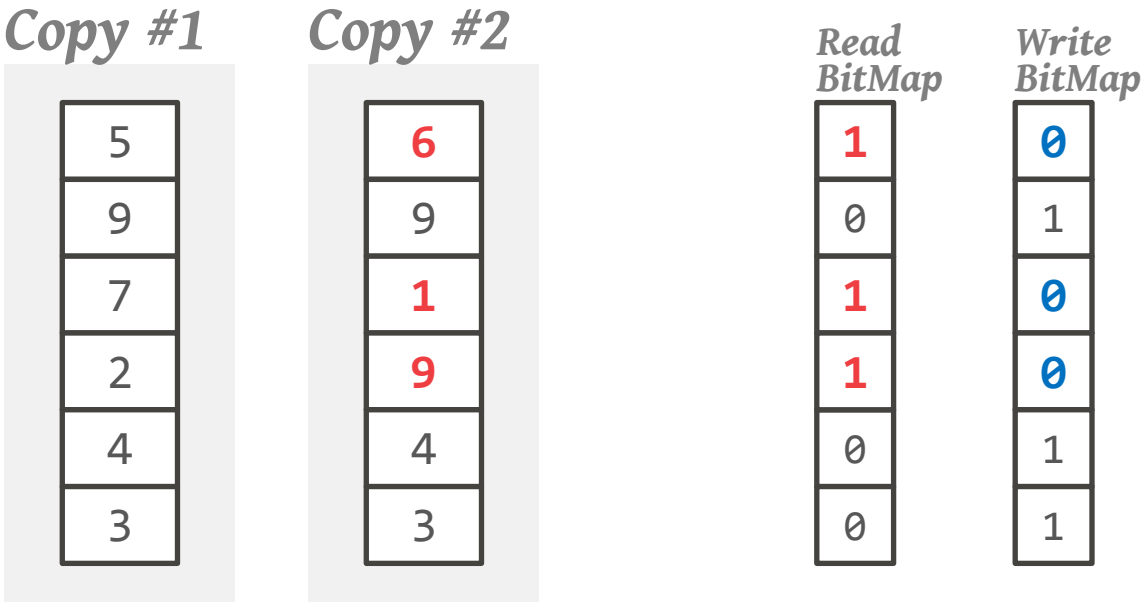
WAIT-FREE ZIGZAG



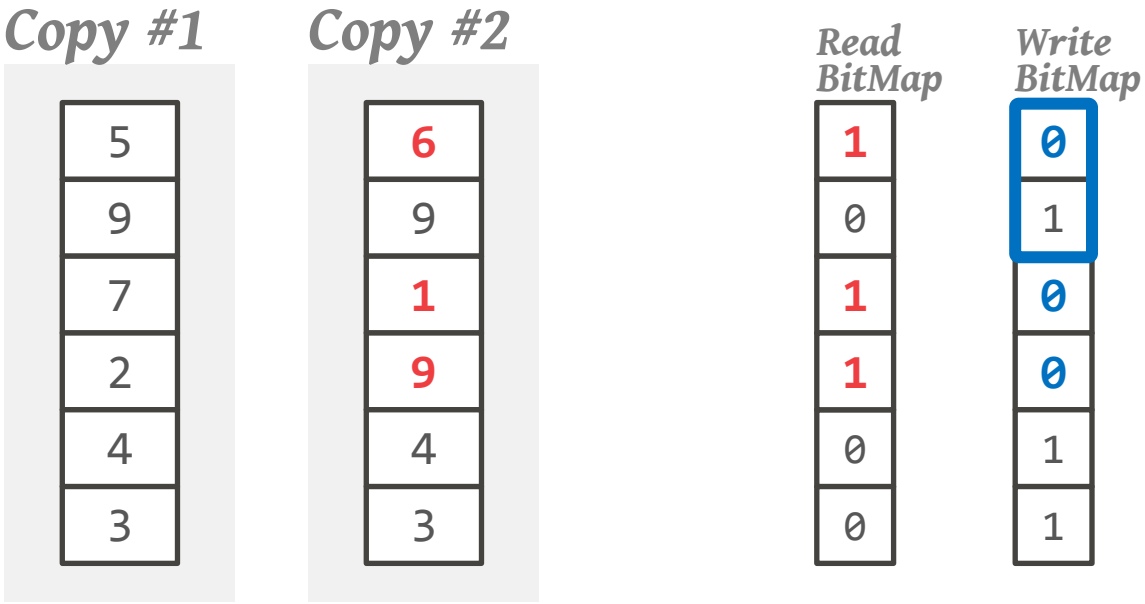
WAIT-FREE ZIGZAG



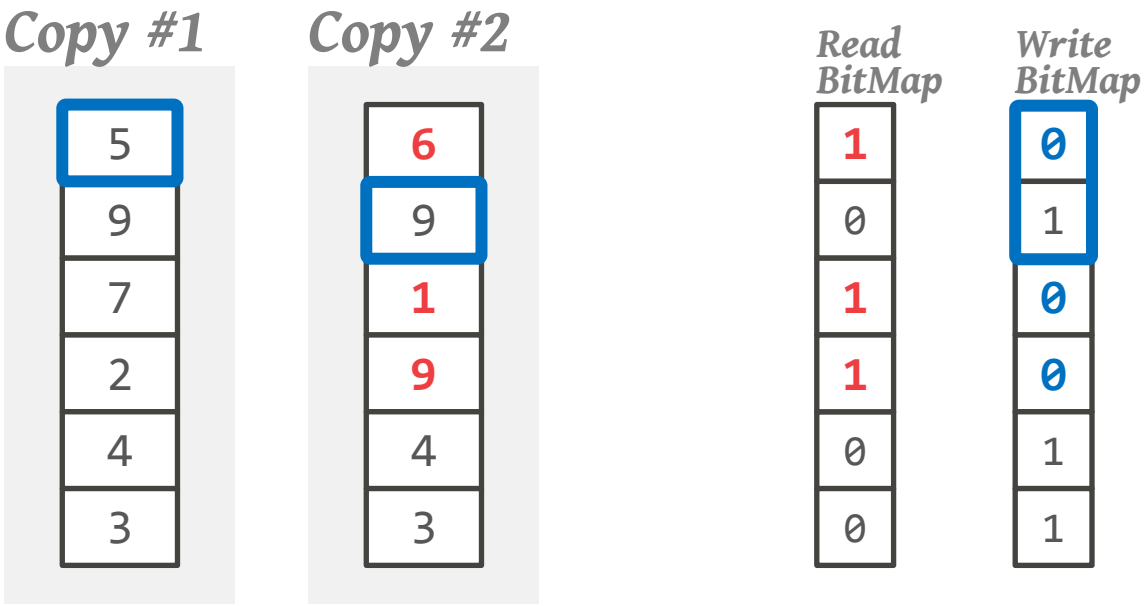
WAIT-FREE ZIGZAG



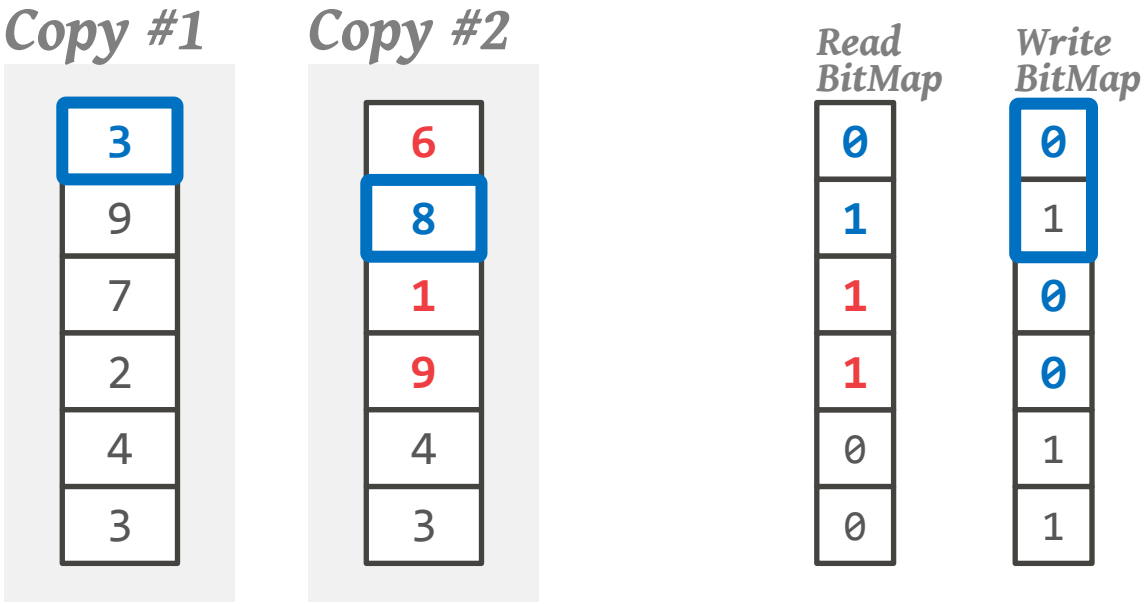
WAIT-FREE ZIGZAG



WAIT-FREE ZIGZAG



WAIT-FREE 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

Copy #1

5
9
7
2
4
3

Copy #2

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

Copy #3

1	5
1	9
1	7
1	2
1	4
1	3

Master: ***Copy #2***

WAIT-FREE PINGPONG

Copy #1

5
9
7
2
4
3

Copy #2

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

Copy #3

1	5
1	9
1	7
1	2
1	4
1	3

Master: ***Copy #2***

WAIT-FREE PINGPONG

Copy #1

5
9
7
2
4
3

Copy #2

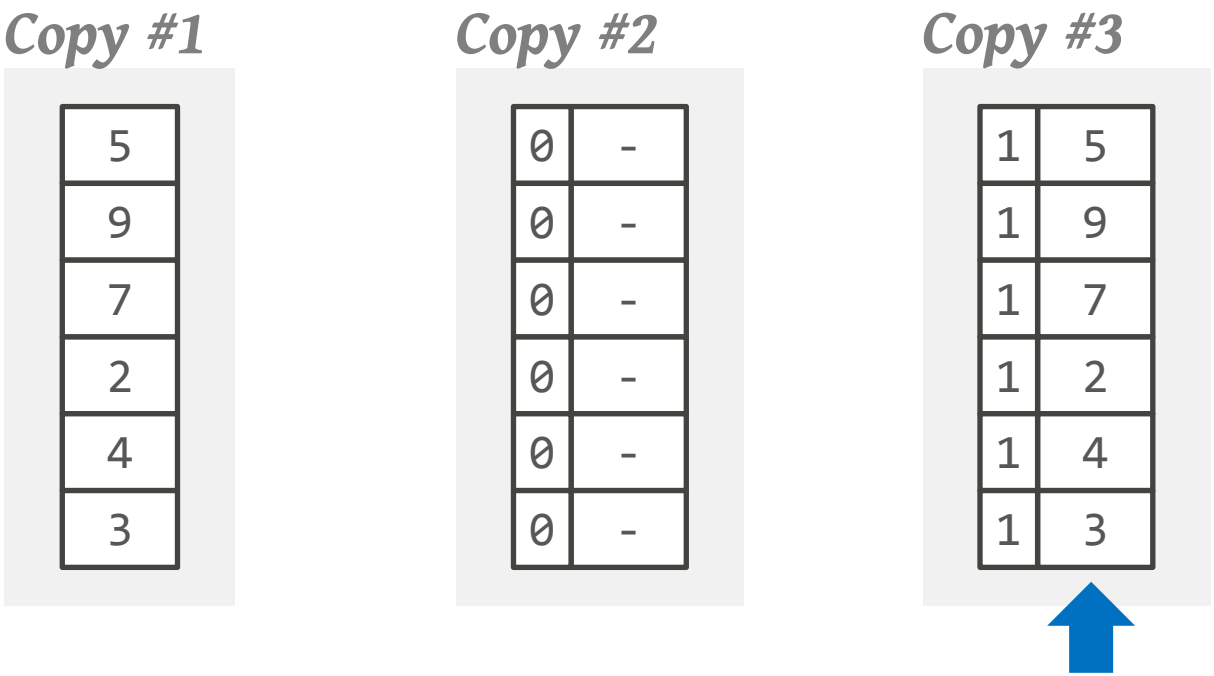
0	-
0	-
0	-
0	-
0	-
0	-

Copy #3

1	5
1	9
1	7
1	2
1	4
1	3

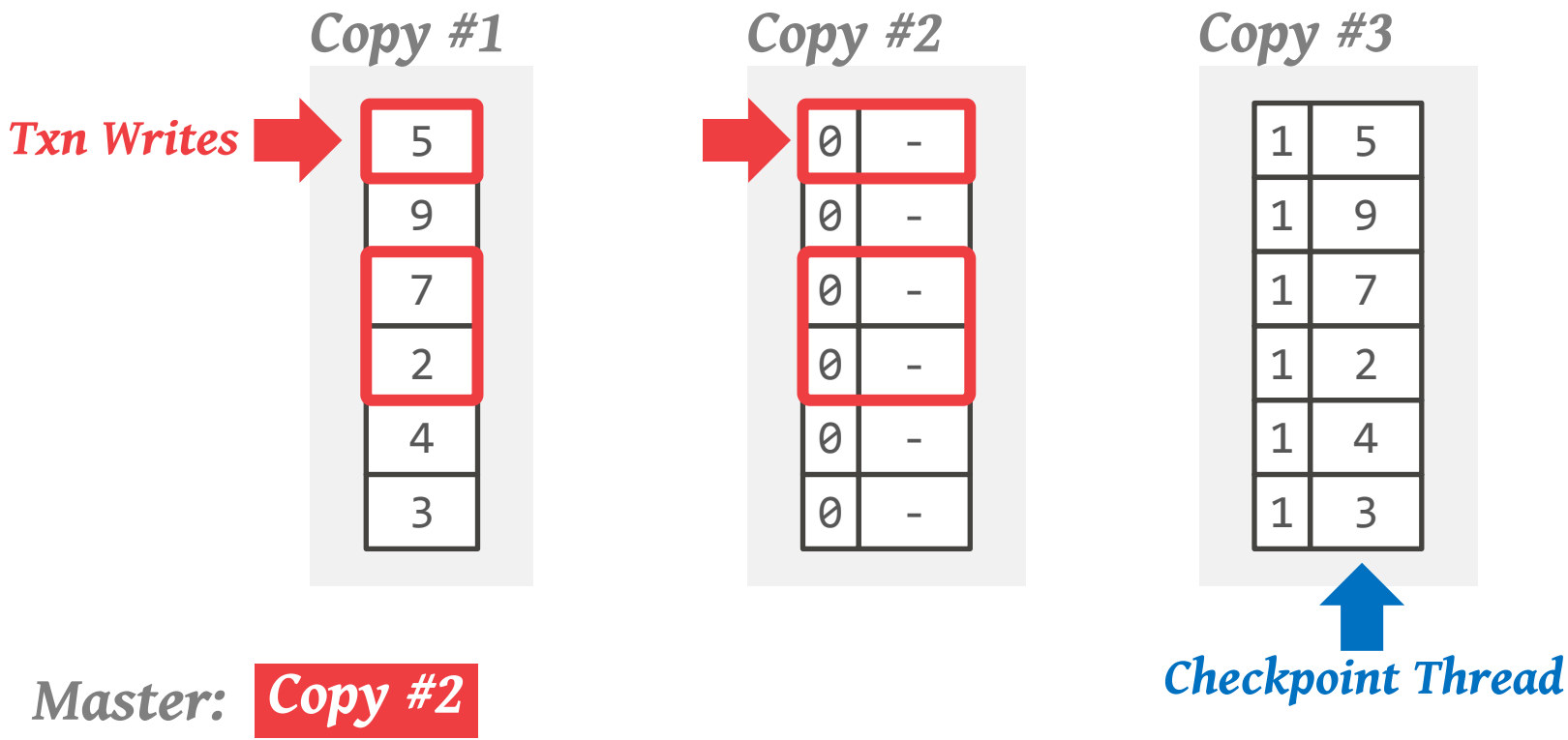
Master: ***Copy #2***

WAIT-FREE PINGPONG

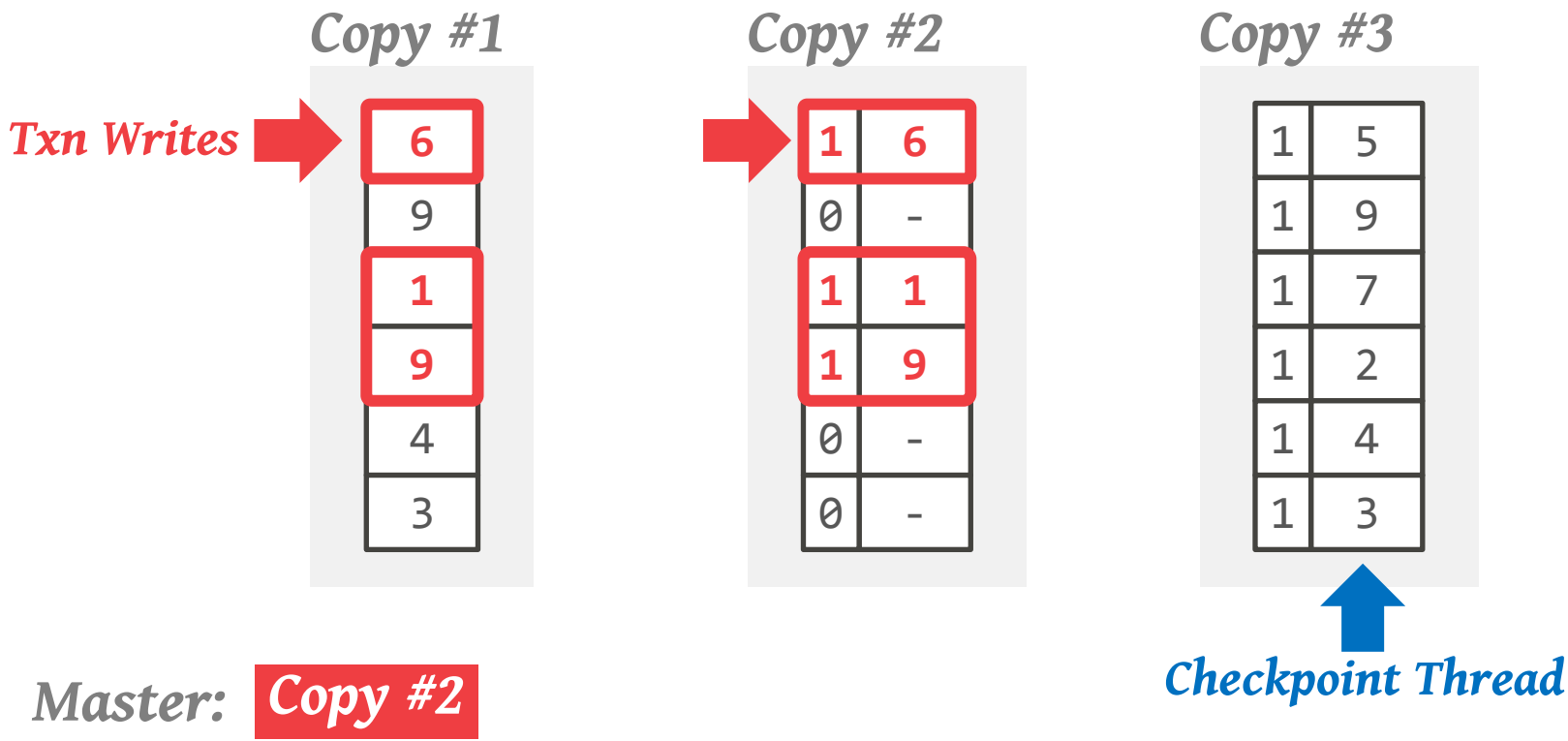


Master: **Copy #2**

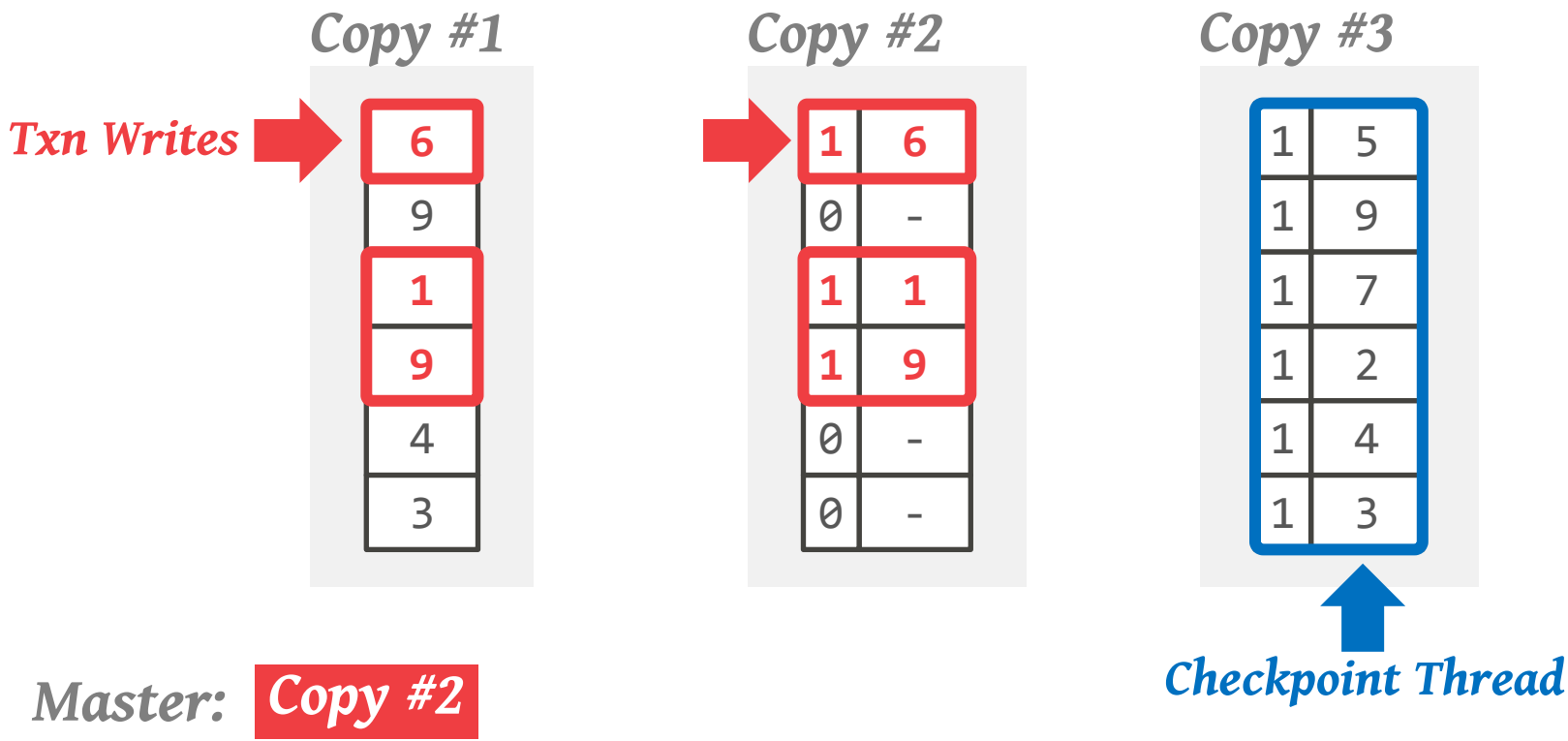
WAIT-FREE PINGPONG



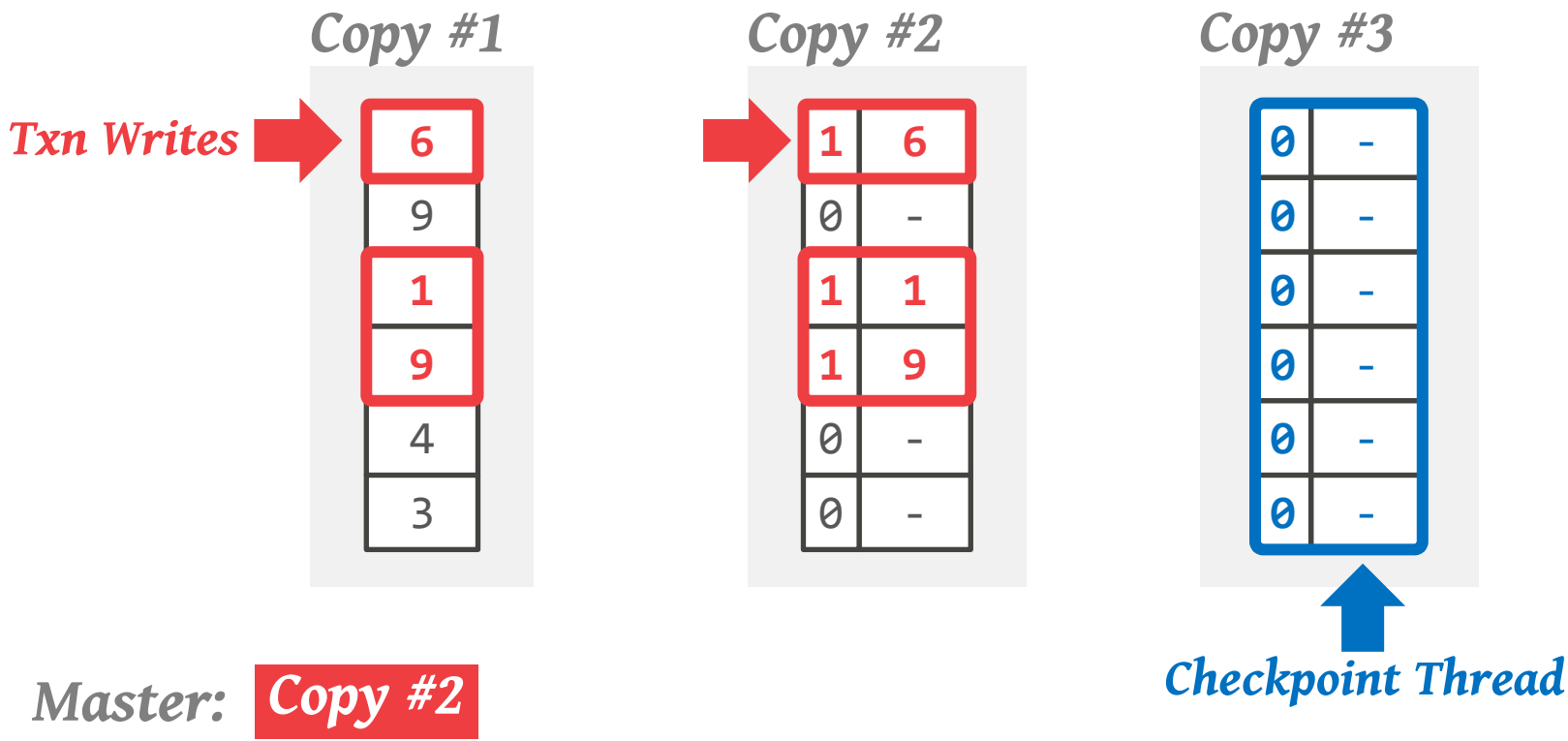
WAIT-FREE PINGPONG



WAIT-FREE PINGPONG

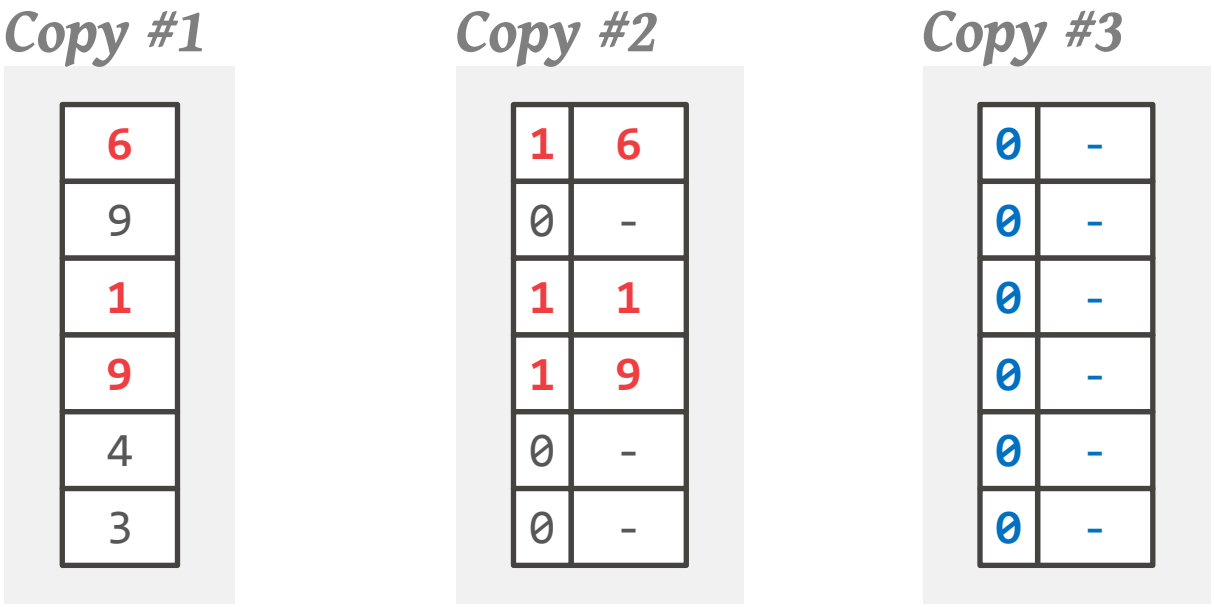


WAIT-FREE PINGPONG



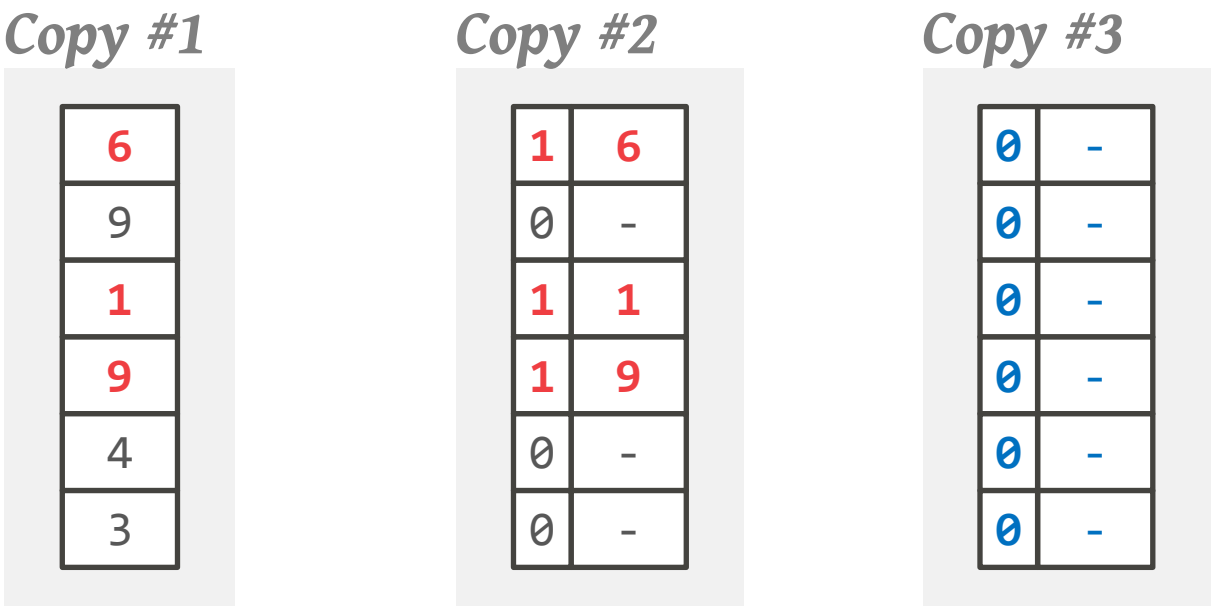
Master: Copy #2

WAIT-FREE PINGPONG



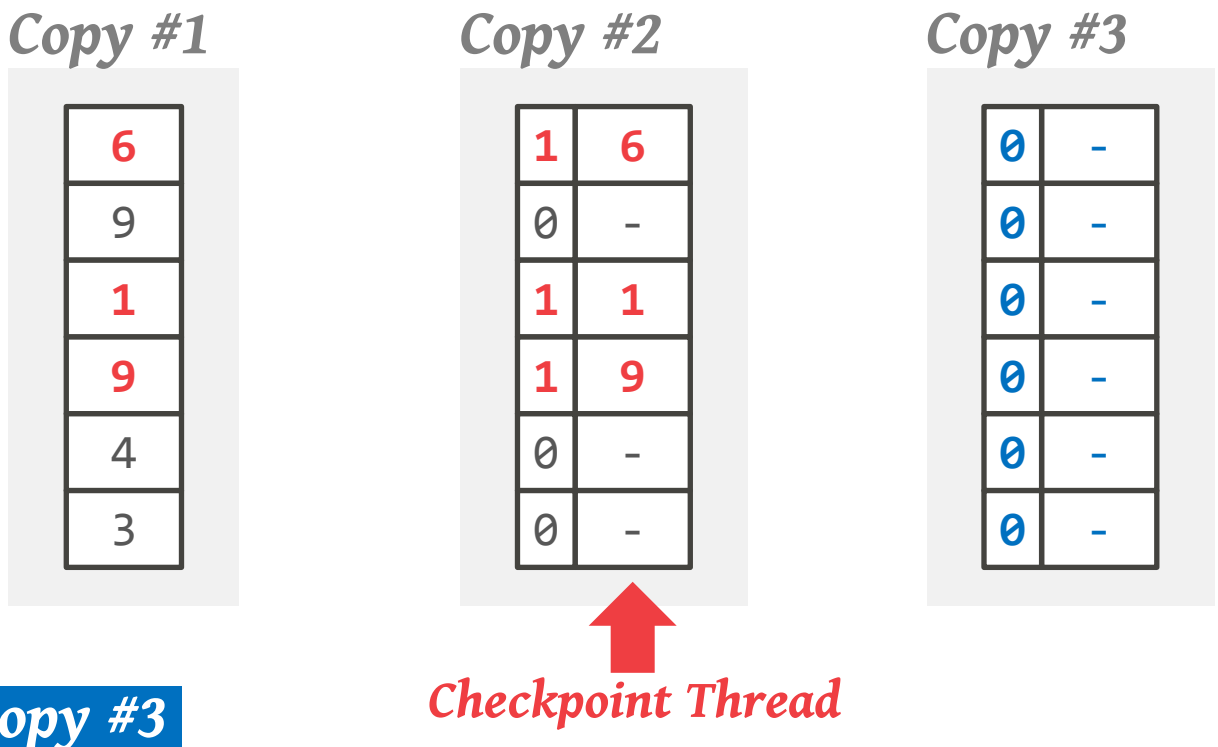
Master: **Copy #2**

WAIT-FREE PINGPONG



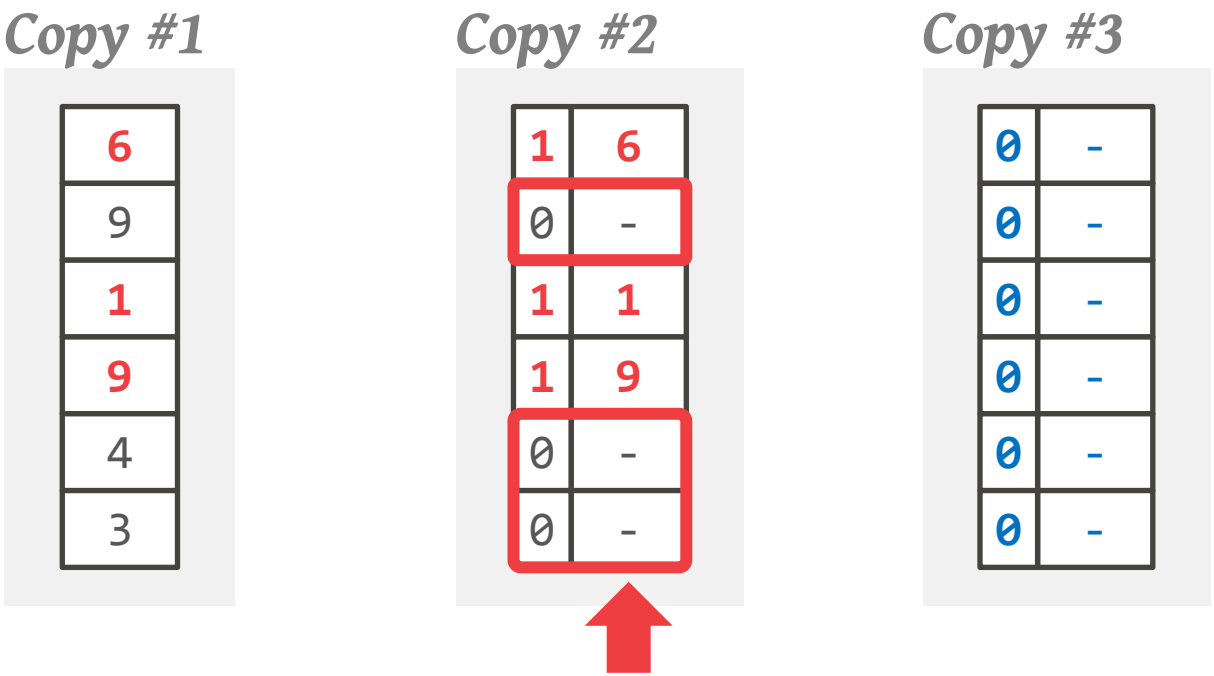
Master: **Copy #3**

WAIT-FREE PINGPONG



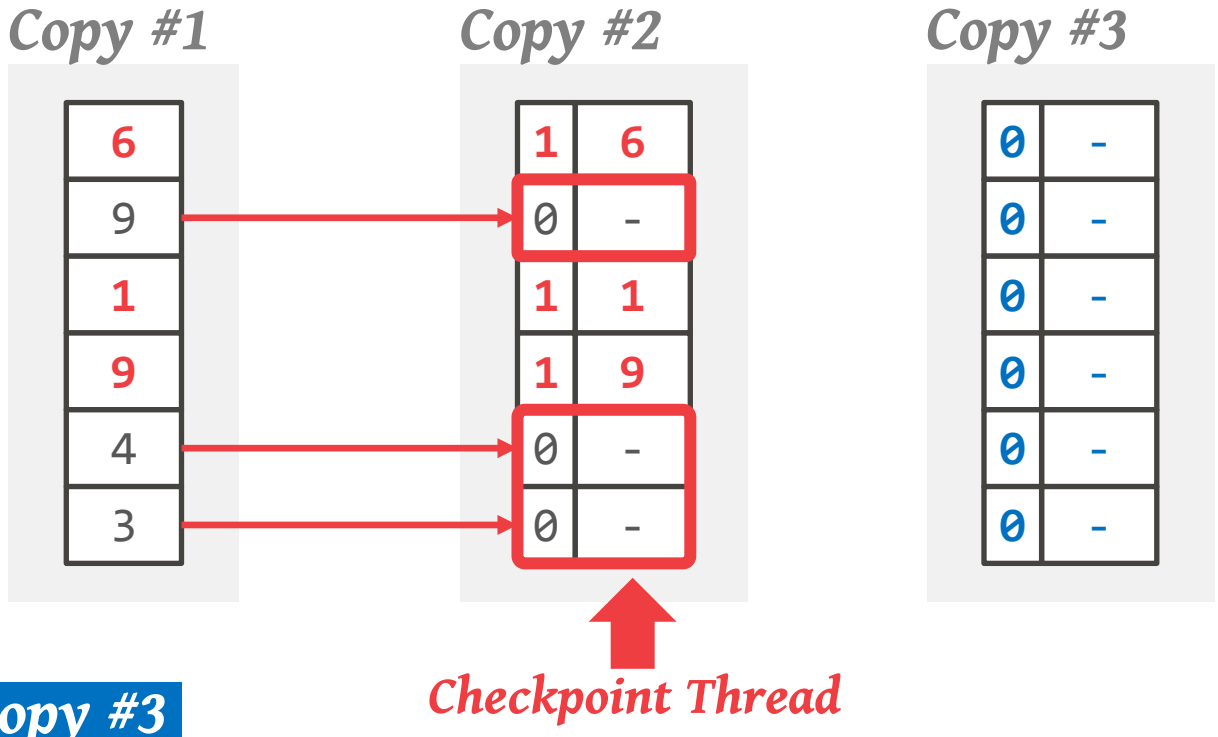
Master: **Copy #3**

WAIT-FREE PINGPONG



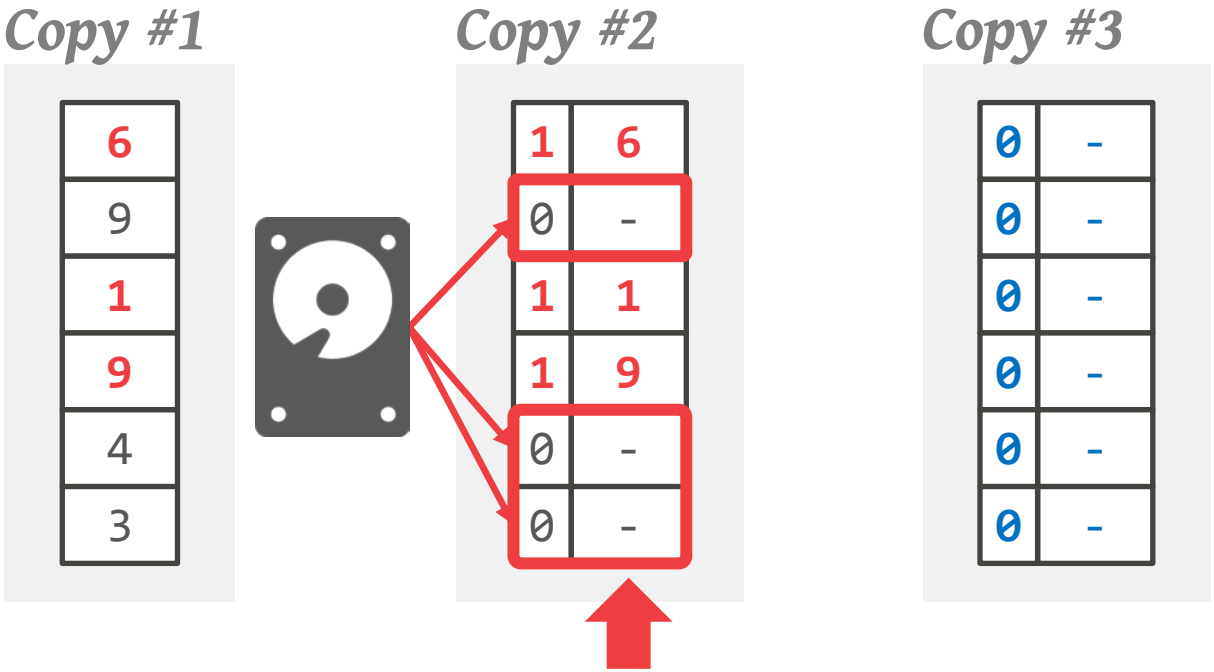
Master: **Copy #3**

WAIT-FREE PINGPONG



Master: **Copy #3**

WAIT-FREE PINGPONG



Master: **Copy #3**

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.

OBSERVATION

Not all DBMS restarts are due to crashes.

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

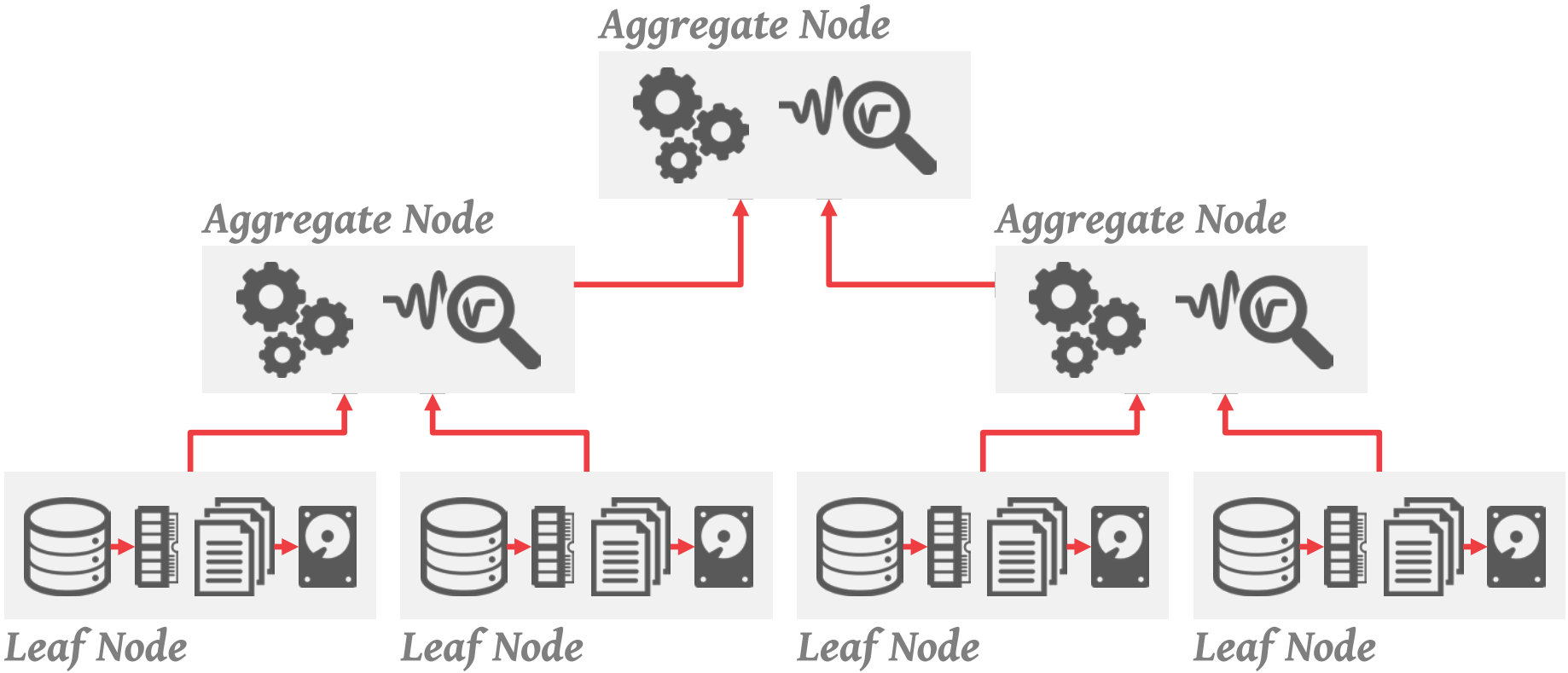
Need a way to be able to quickly restart the DBMS without having to re-read the entire database from disk again.

FACEBOOK SCUBA

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



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.

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.

Approach #2: Copy on Shutdown

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

SCUBA – FAST RESTARTS

When the admin initiates restart command, the leaf 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

Logical logging is faster at runtime but difficult to implement recovery.

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...

SEMESTER PROGRESS

Concurrency Control
Storage Models
Indexes
Scheduling & Execution
Join Algorithms
Logging & Recovery

Compression
Query Optimization
Vectorization
Scan Sharing
JIT Compilation
Mat. Views
NVM / HTM

NEXT CLASS

Project #3 Topics
Extra Credit

Project #2 is now due **March 9th @ 11:59pm**
Project #3 proposals are still due **March 14th**
No Mandatory Reading for **March 2nd**