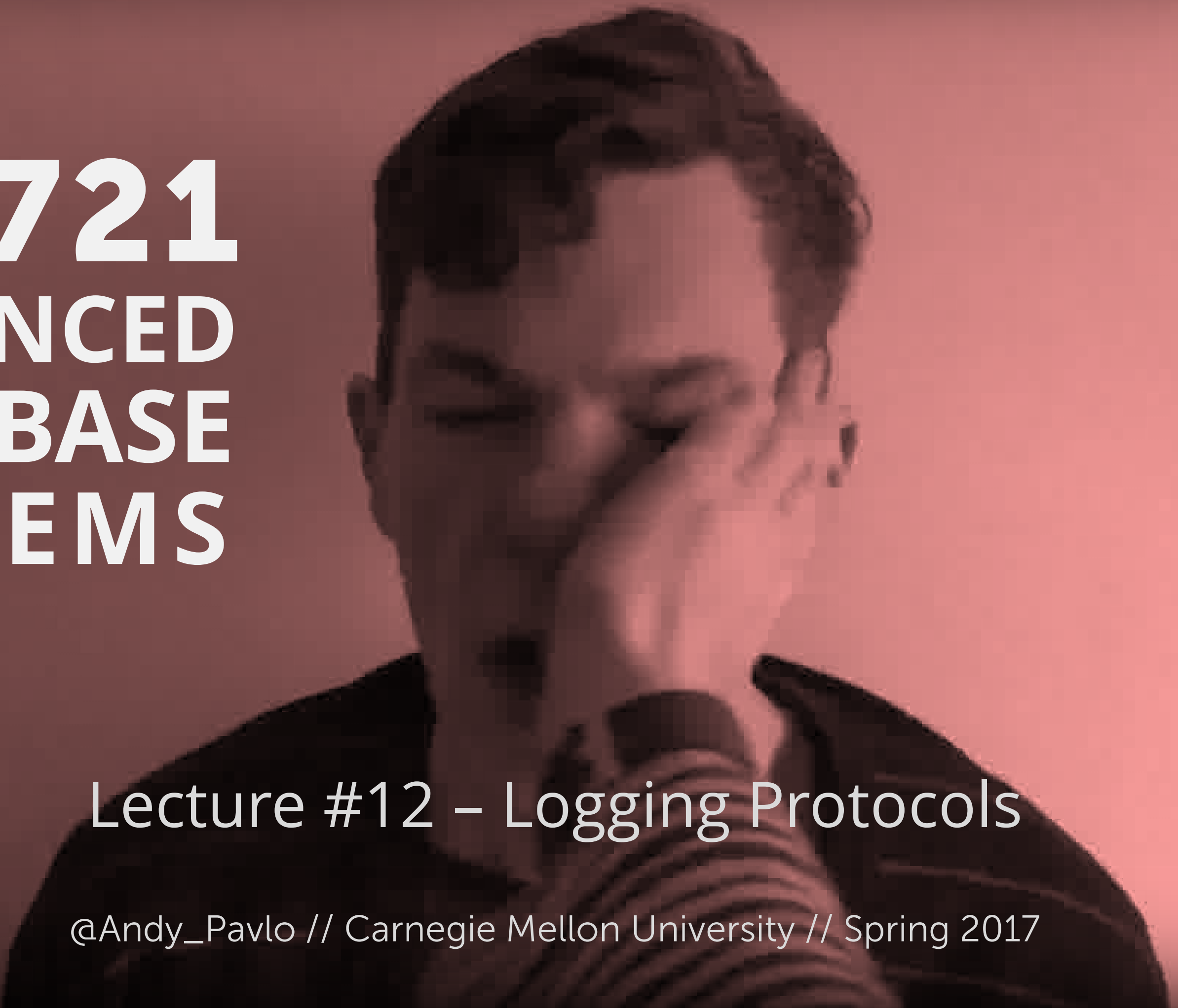
15-721 ADVANCED DATABASE SYSTEMS



Lecture #12 – Logging Protocols

@Andy_Pavlo // Carnegie Mellon University // Spring 2017

TODAY'S AGENDA

Logging Schemes
Crash Course on ARIES
Physical Logging
Command Logging

LOGGING & RECOVERY

Recovery algorithms are techniques to ensure database **consistency**, txn **atomicity** and **durability** despite failures.

Recovery algorithms have two parts:
→ Actions during normal txn processing to ensure that the DBMS can recover from a failure.
→ Actions after a failure to recover the database to a state that ensures atomicity, consistency, and durability.

LOGGING SCHEMES

Physical Logging

→ Record the changes made to a specific record in the database.
→ Example: Store the original value and after value for an attribute that is changed by a query.

Logical Logging

→ Record the high-level operations executed by txns.
→ Example: The `UPDATE`, `DELETE`, and `INSERT` queries invoked by a txn.

PHYSICAL VS. LOGICAL LOGGING

Logical logging writes less data in each log record than physical logging.

Difficult to implement recovery with logical logging if you have concurrent txns.

→ Hard to determine which parts of the database may have been modified by a query before crash.
→ Also takes longer to recover because you must re-execute every txn all over again.

LOGICAL LOGGING EXAMPLE

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

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

NAME	SALARY
O.D.B.	$100
El-P	$666
Andy	$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 = 'Andy'
```

NAME	SALARY
O.D.B.	$100
El-P	$666
Andy	$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 = 'Andy'
```

NAME	SALARY
O.D.B.	$110
El-P	$732
Andy	$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 = 'Andy'
```

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

Logical Log

```
UPDATE employees SET
salary = salary * 1.10

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

LOGICAL LOGGING EXAMPLE

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

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

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

Logical Log

```
UPDATE employees SET
salary = salary * 1.10

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

LOGICAL LOGGING EXAMPLE

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

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

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

Logical Log

```
UPDATE employees SET
salary = salary * 1.10

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

LOGICAL LOGGING EXAMPLE

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

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

NAME	SALARY
O.D.B.	$110
El-P	$732
Andy	$990

Logical Log

```
UPDATE employees SET
salary = salary * 1.10

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

LOGICAL LOGGING EXAMPLE

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

```
UPDATE employees
   SET salary = 900
 WHERE name = ‘Andy’
```

NAME	SALARY
O.D.B	$110
El-P	$732
Andy	$990

Logical Log

```
UPDATE employees SET
salary = salary * 1.10

UPDATE employees SET
salary = 900 WHERE
name = ‘Andy’
```

LOGICAL LOGGING EXAMPLE

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

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

Logical Log

```
UPDATE employees SET
salary = salary * 1.10

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

NAME	SALARY
O.D.B	$110
El-P	$732
Andy	$990

SALARY
$110
$732
$900

DISK-ORIENTED LOGGING & RECOVERY

The "gold standard" for physical logging & recovery in a disk-oriented DBMS is **ARIES**.

→ **A**lgorithms for **R**ecovery and **I**solation **E**xploiting **S**emantics
→ Invented by IBM Research in the early 1990s.

Relies on STEAL and NO-FORCE buffer pool management policies.

ARIES: A TRANSACTION RECOVERY METHOD SUPPORTING FINE-GRANULARITY LOCKING AND PARTIAL ROLLBACKS USING WRITE-AHEAD LOGGING
ACM Transactions on Database Systems 1992

ARIES – MAIN IDEAS

Write-Ahead Logging:

→ Any change is recorded in log on stable storage before the database change is written to disk.

Repeating History During Redo:

→ On restart, retrace actions and restore database to exact state before crash.

Logging Changes During Undo:

→ Record undo actions to log to ensure action is not repeated in the event of repeated failures.

ARIES – RUNTIME LOGGING

For each modification to the database, the DBMS appends a record to the tail of the log.

When a txn commits, its log records are flushed to durable storage.

ARIES – RUNTIME CHECKPOINTS

Use fuzzy checkpoints to allow txns to keep on running while writing checkpoint.

→ The checkpoint may contain updates from txns that have not committed and may abort later on.

The DBMS records internal system state as of the beginning of the checkpoint.

→ Active Transaction Table (ATT)

→ Dirty Page Table (DPT)

LOG SEQUENCE NUMBERS

Every log record has a globally unique ***log sequence number*** (LSN) that is used to determine the serial order of those records.

The DBMS keeps track of various LSNs in both volatile and non-volatile storage to determine the order of almost **everything** in the system…

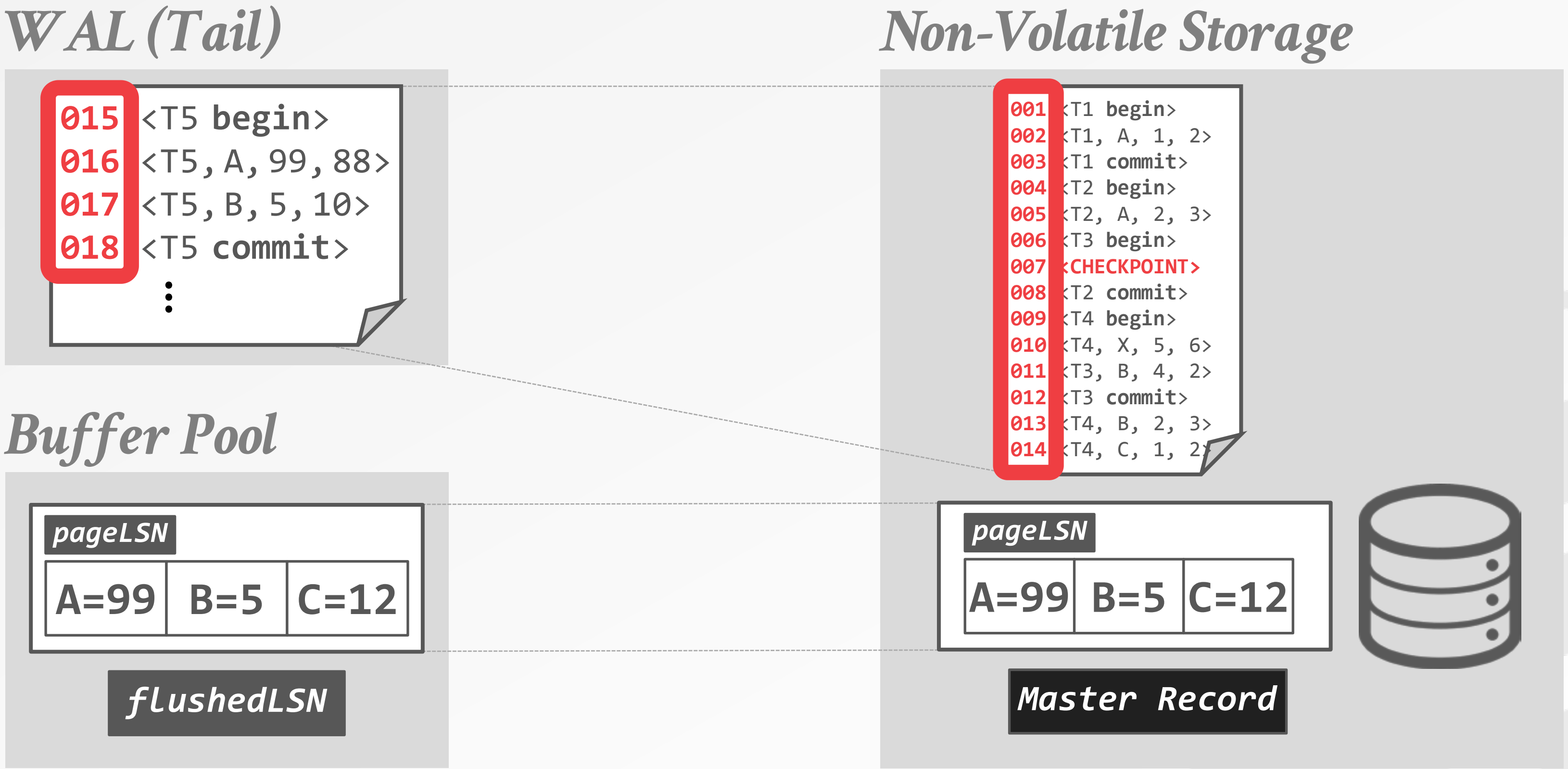
LOG SEQUENCE NUMBERS

Each page contains a ***pageLSN*** that represents the LSN of the most recent update to that page.

The DBMS keeps track of the max log record written to disk (***flushedLSN***).

For a page ***i*** to be written, the DBMS must flush log at least to the point where $\boldsymbol{pageLSN_i \leq flushedLSN}$

LOG SEQUENCE NUMBERS



LOG SEQUENCE NUMBERS

WAL (Tail)

```
015:<T5 begin>
016:<T5,A,99,88>
017:<T5,B,5,10>
018:<T5 commit>
⋮
```

Non-Volatile Storage

```
001:<T1 begin>
002:<T1, A, 1, 2>
003:<T1 commit>
004:<T2 begin>
005:<T2, A, 2, 3>
006:<T3 begin>
007:<CHECKPOINT>
008:<T2 commit>
009:<T4 begin>
010:<T4, X, 5, 6>
011:<T3, B, 4, 2>
012:<T3 commit>
013:<T4, B, 2, 3>
014:<T4, C, 1, 2>
```

Buffer Pool

pageLSN

A=99	B=5	C=12

flushedLSN

pageLSN

A=99	B=5	C=12

Master Record

LOG SEQUENCE NUMBERS

WAL (Tail)

```
015:<T5 begin>
016:<T5,A,99,88>
017:<T5,B,5,10>
018:<T5 commit>
⋮
```

Buffer Pool

pageLSN

A=99	B=5	C=12

flushedLSN

Non-Volatile Storage

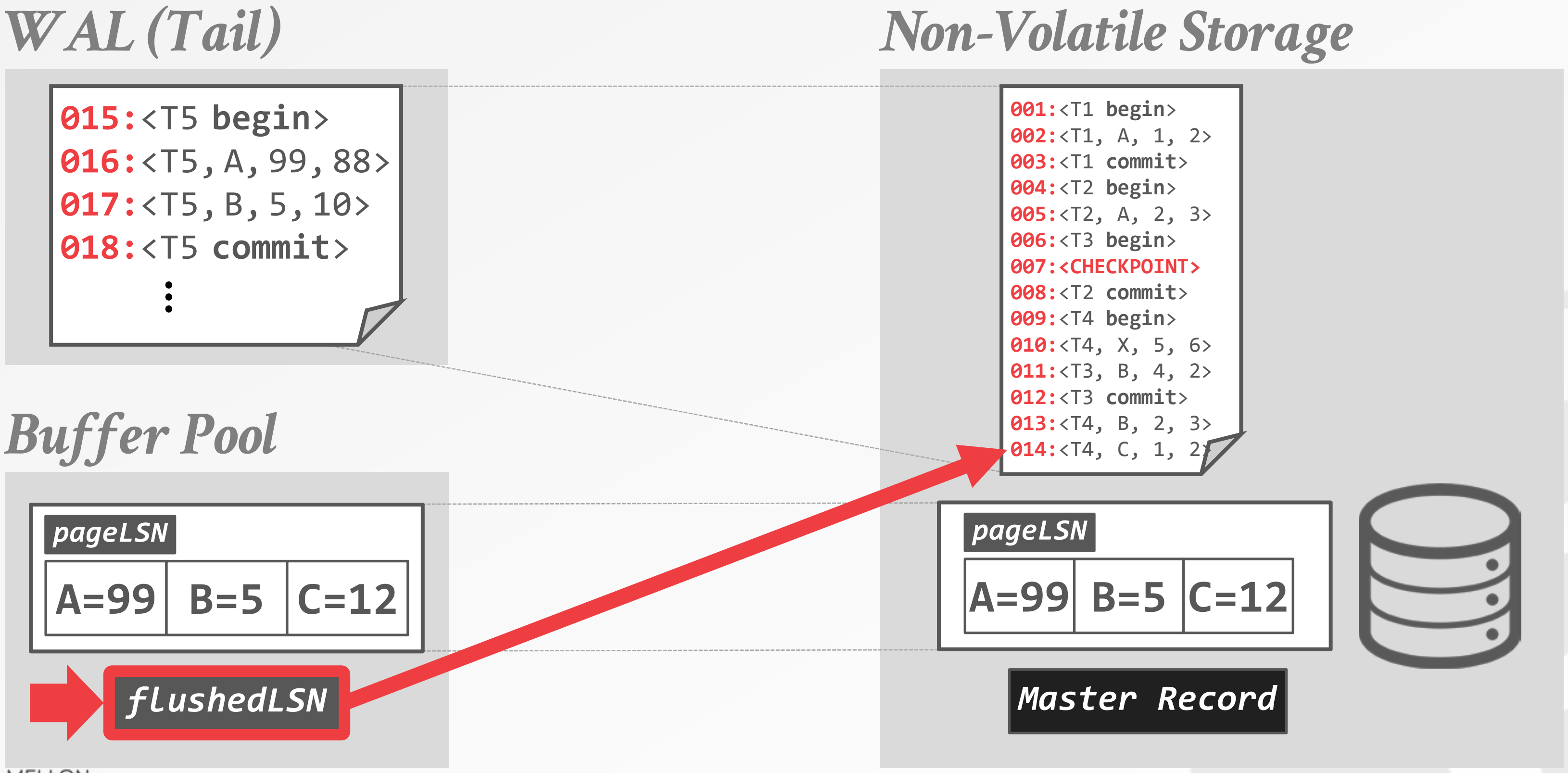
```
001:<T1 begin>
002:<T1, A, 1, 2>
003:<T1 commit>
004:<T2 begin>
005:<T2, A, 2, 3>
006:<T3 begin>
007:<CHECKPOINT>
008:<T2 commit>
009:<T4 begin>
010:<T4, X, 5, 6>
011:<T3, B, 4, 2>
012:<T3 commit>
013:<T4, B, 2, 3>
014:<T4, C, 1, 2>
```

pageLSN

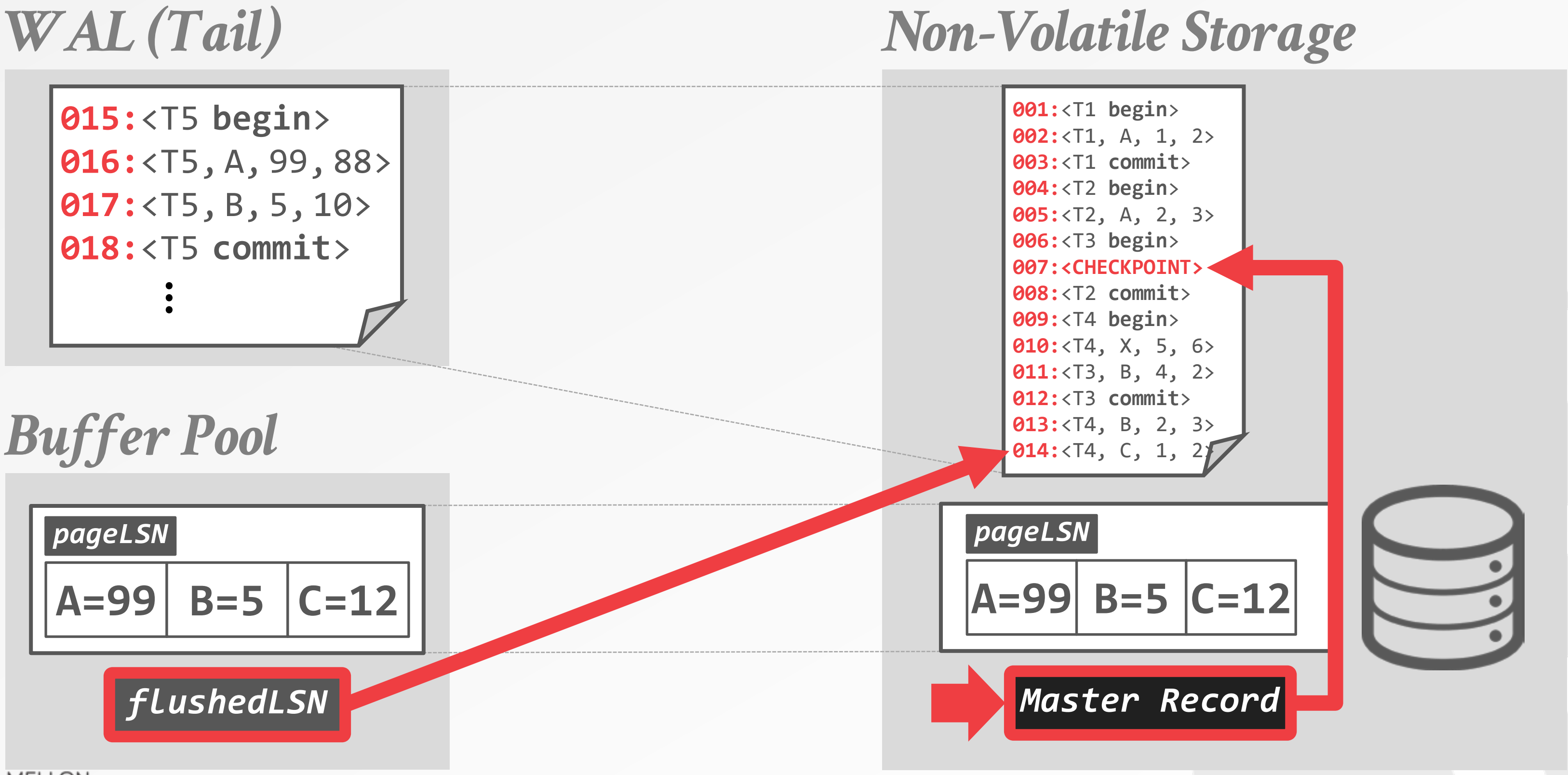
A=99	B=5	C=12

Master Record

LOG SEQUENCE NUMBERS



LOG SEQUENCE NUMBERS



LOG SEQUENCE NUMBERS

WAL (Tail)

015:<T5 begin>
016:<T5,A,99,88>
017:<T5,B,5,10>
018:<T5 commit>
⋮

Non-Volatile Storage

001:<T1 begin>
002:<T1, A, 1, 2>
003:<T1 commit>
004:<T2 begin>
005:<T2, A, 2, 3>
006:<T3 begin>
007:<CHECKPOINT>
008:<T2 commit>
009:<T4 begin>
010:<T4, X, 5, 6>
011:<T3, B, 4, 2>
012:<T3 commit>
013:<T4, B, 2, 3>
014:<T4, C, 1, 2>

Buffer Pool

pageLSN

A=99	B=5	C=12

flushedLSN

pageLSN

A=99	B=5	C=12

Master Record

LOG SEQUENCE NUMBERS

WAL (Tail)

```
015:<T5 begin>
016:<T5,A,99,88>
017:<T5,B,5,10>
018:<T5 commit>
⋮
```

Non-Volatile Storage

```
001:<T1 begin>
002:<T1, A, 1, 2>
003:<T1 commit>
004:<T2 begin>
005:<T2, A, 2, 3>
006:<T3 begin>
007:<CHECKPOINT>
008:<T2 commit>
009:<T4 begin>
010:<T4, X, 5, 6>
011:<T3, B, 4, 2>
012:<T3 commit>
013:<T4, B, 2, 3>
014:<T4, C, 1, 2>
```

Buffer Pool

pageLSN

A=99		C=12

flushedLSN

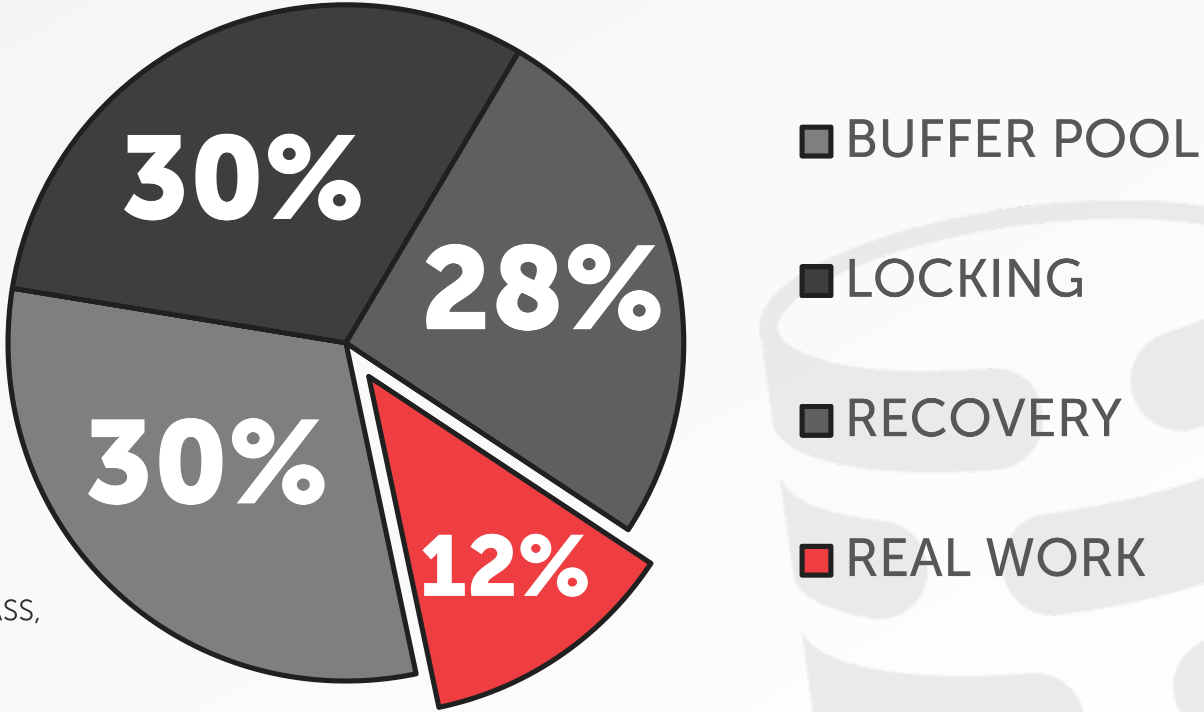
pageLSN

A=99	B=5	C=12

Master Record

DISK-ORIENTED DBMS OVERHEAD

Measured CPU Cycles



OLTP THROUGH THE LOOKING GLASS, AND WHAT WE FOUND THERE
SIGMOD, pp. 981-992, 2008.

OBSERVATION

Often the slowest part of the txn is waiting for the DBMS to flush the log records to disk.

Have to wait until the records are safely written before the DBMS can return the acknowledgement to the client.

GROUP COMMIT

Batch together log records from multiple txns and flush them together with a single `fsync`.

- → Logs are flushed either after a timeout or when the buffer gets full.
- → Originally developed in IBM IMS FastPath in the 1980s

This amortizes the cost of I/O over several txns.

EARLY LOCK RELEASE

A txn's locks can be released before its commit record is written to disk as long as it does not return results to the client before becoming durable.

Other txns that read data updated by a **pre-committed** txn become dependent on it and also have to wait for their predecessor's log records to reach disk.

IN-MEMORY DATABASE RECOVERY

Recovery is slightly easier because the DBMS does not have to worry about tracking dirty pages in case of a crash during recovery.

An in-memory DBMS also does not need to store undo records.

But the DBMS is still stymied by the slow sync time of non-volatile storage

OBSERVATION

The early papers (1980s) on recovery for in-memory DBMSs assume that there is non-volatile memory.

This hardware is still not widely available so we want to use existing SSD/HDDs.

A RECOVERY ALGORITHM FOR A HIGH-PERFORMANCE MEMORY-RESIDENT DATABASE SYSTEM
SIGMOD 1987

SILO – LOGGING AND RECOVERY

SiloR uses the epoch-based OCC that we discussed previously.

It achieves high performance by parallelizing all aspects of logging, checkpointing, and recovery.

Again, Eddie Kohler is unstoppable.

FAST DATABASES WITH FAST DURABILITY AND RECOVERY THROUGH MULTICORE PARALLELISM
OSDI 2014

SILOR – LOGGING PROTOCOL

The DBMS assumes that there is one storage device per CPU socket.

→ Assigns one logger thread per device.
→ Worker threads are grouped per CPU socket.

As the worker executes a txn, it creates new log records that contain the values that were written to the database (i.e., REDO).

SILOR – LOGGING PROTOCOL

Each logger thread maintains a pool of log buffers that are given to its worker threads.

When a worker's buffer is full, it gives it back to the logger thread to flush to disk and attempts to acquire a new one.

→ If there are no available buffers, then it stalls.

SILOR – LOG FILES

The logger threads write buffers out to files

→ After 100 epochs, it creates a new file.

→ The old file is renamed with a marker indicating the max epoch of records that it contains.

Log record format:

→ Id of the txn that modified the record (TID).

→ A set of value log triplets (Table, Key, Value).

→ The value can be a list of attribute + value pairs.

SILOR – LOG FILES

```
root@magneto:/var/lib/mysql# ls -lah
total 5.5G
drwxr-x---  5 mysql mysql 4.0K Dec 22 07:56 .
drwxr-xr-x 69 root  root  4.0K Dec 16 20:22 ..
-rw-rw----  1 mysql mysql   56 Aug 16  2015 auto.cnf
-rw-------  1 mysql mysql 1.7K Dec 16 20:22 ca-key.pem
-rw-r--r--  1 mysql mysql 1.1K Dec 16 20:22 ca.pem
-rw-r--r--  1 mysql mysql 1.1K Dec 16 20:22 client-cert.pem
-rw-------  1 mysql mysql 1.7K Dec 16 20:22 client-key.pem
-rw-r-----  1 mysql mysql 1.1K Dec 16 20:29 ib_buffer_pool
-rw-rw----  1 mysql mysql  76M Dec 21 08:38 ibdata1
-rw-r-----  1 mysql mysql 500M Dec 22 07:00 ib_logfile0
-rw-r-----  1 mysql mysql 500M Dec 21 08:39 ib_logfile1
-rw-rw----  1 mysql mysql 4.4G Dec 21 08:38 magneto.log
-rw-rw----  1 mysql mysql  55M Dec 21 08:38 magneto-slow.log
drwxr-x---  2 mysql mysql 4.0K Dec 16 20:27 mysql
-rw-r--r--  1 root  root     6 Dec 16 20:27 mysql_upgrade_info
```

SILOR – LOG FILES

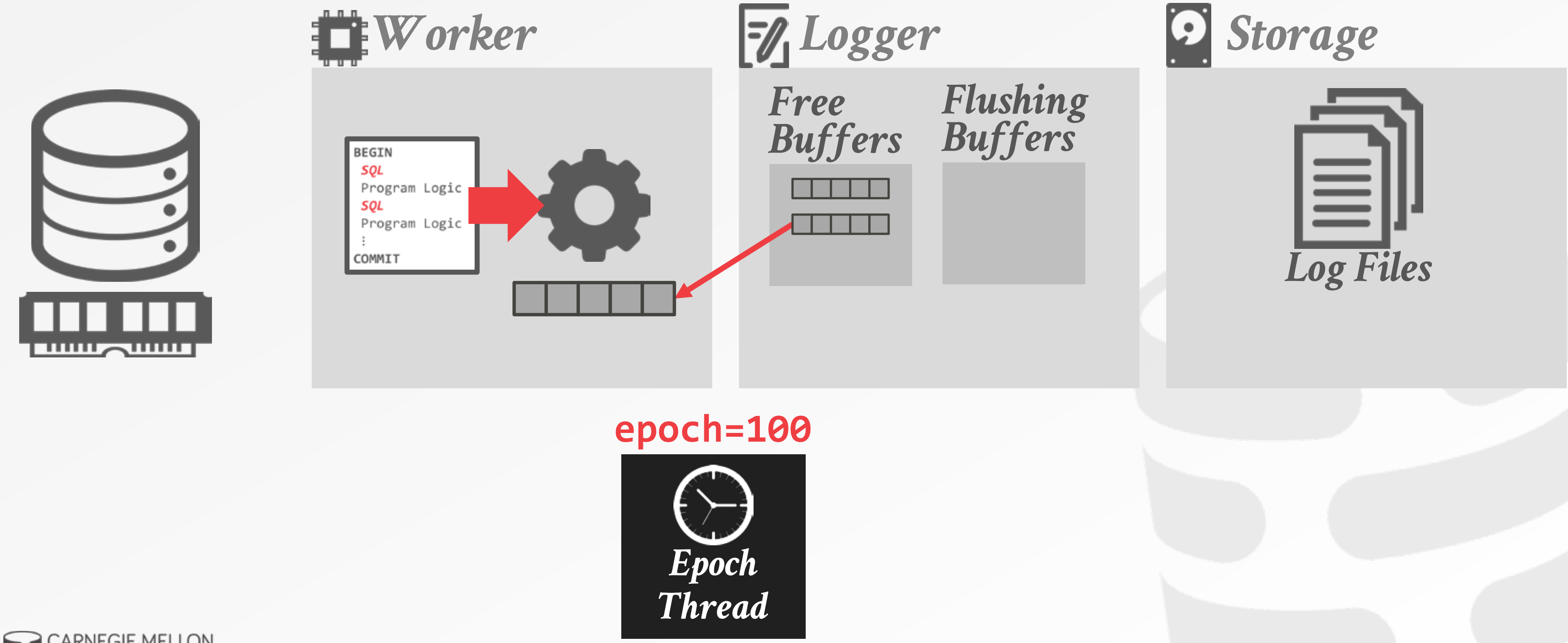
The logger threads write buffers out to files
- → After 100 epochs, it creates a new file.
- → The old file is renamed with a marker indicating the max epoch of records that it contains.

Log record format:
- → Id of the txn that modified the record (TID).
- → A set of value log triplets (Table, Key, Value).
- → The value can be a list of attribute + value pairs.

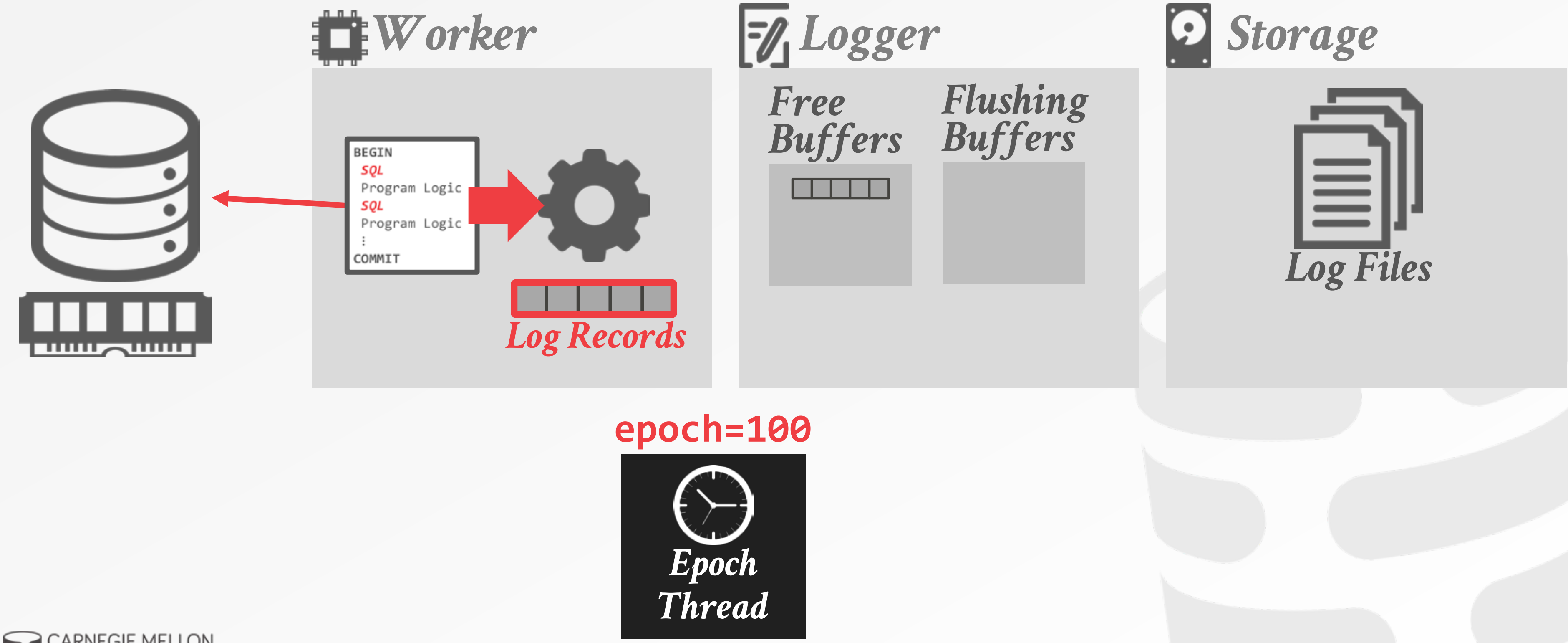
```
UPDATE people
   SET isLame = true
 WHERE name IN ('Dana','Andy')
```

```
Txn#1001
[people, 888, (isLame→true)]
[people, 999, (isLame→true)]
```

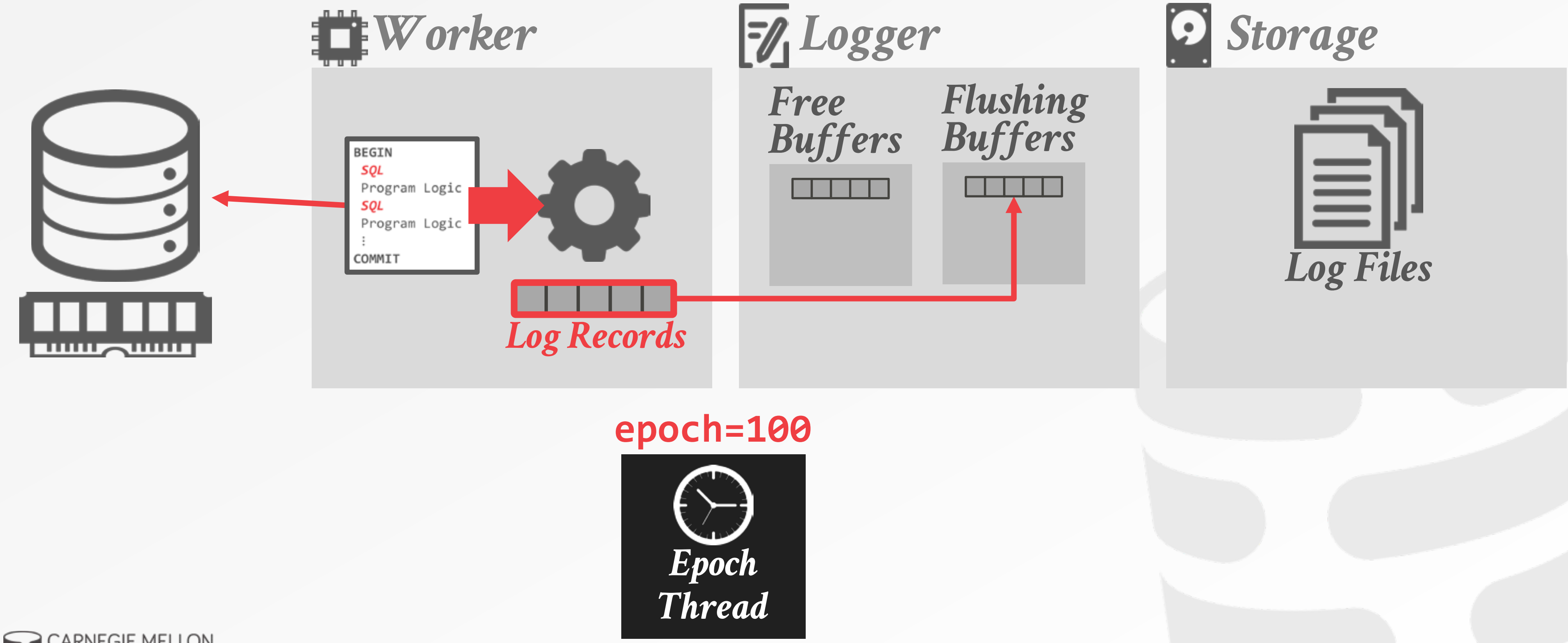
SILOR – ARCHITECTURE



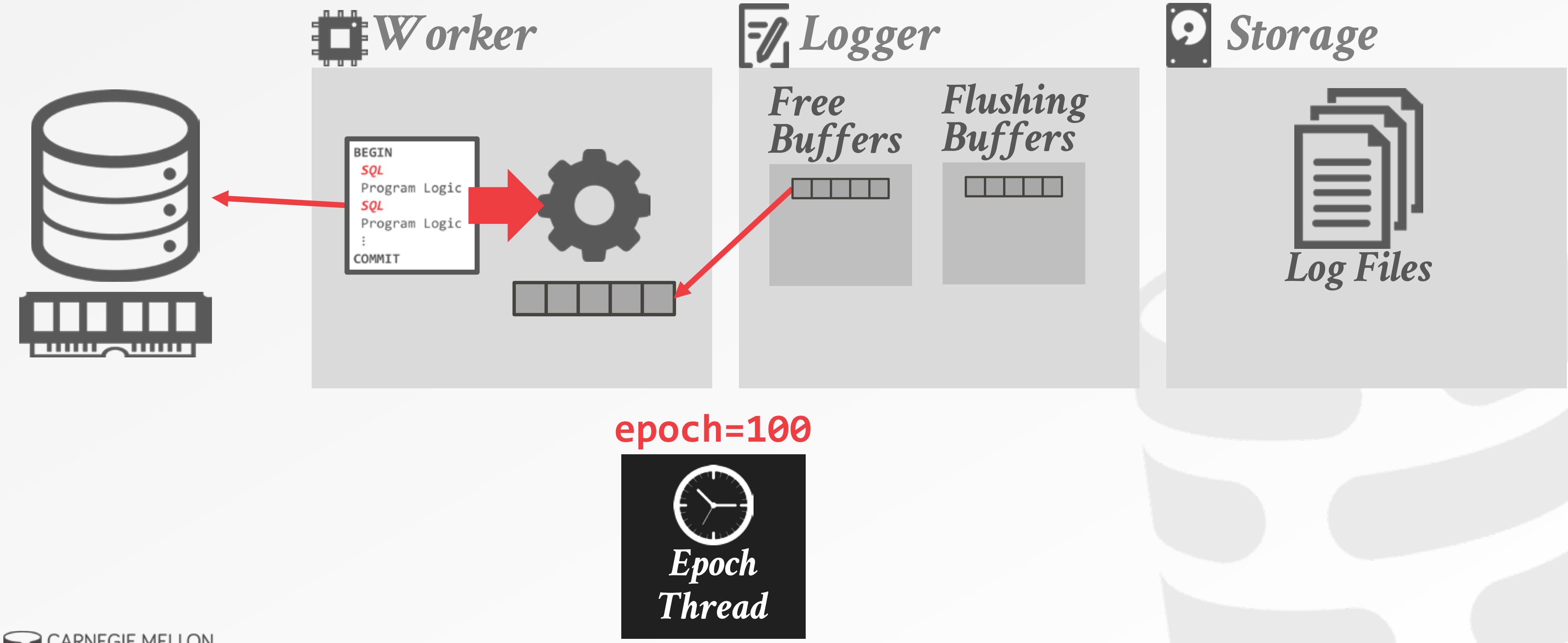
SILOR – ARCHITECTURE



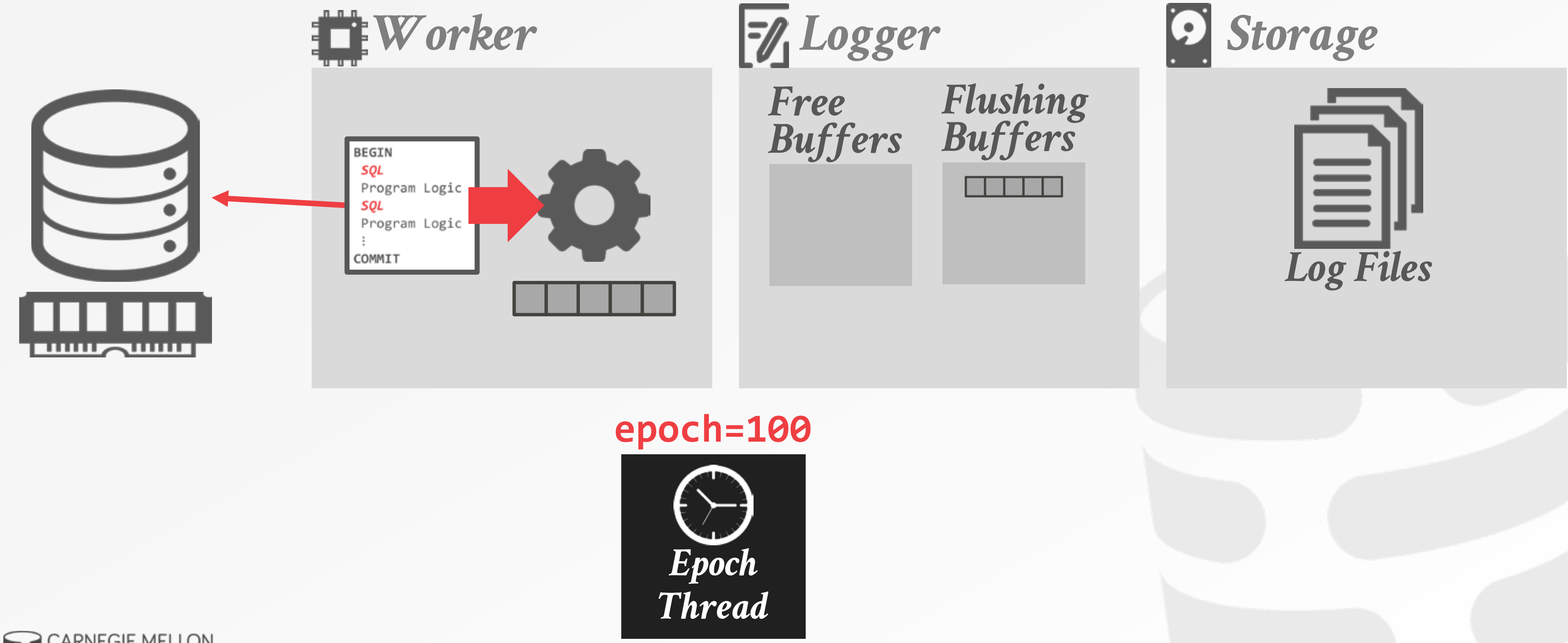
SILOR – ARCHITECTURE



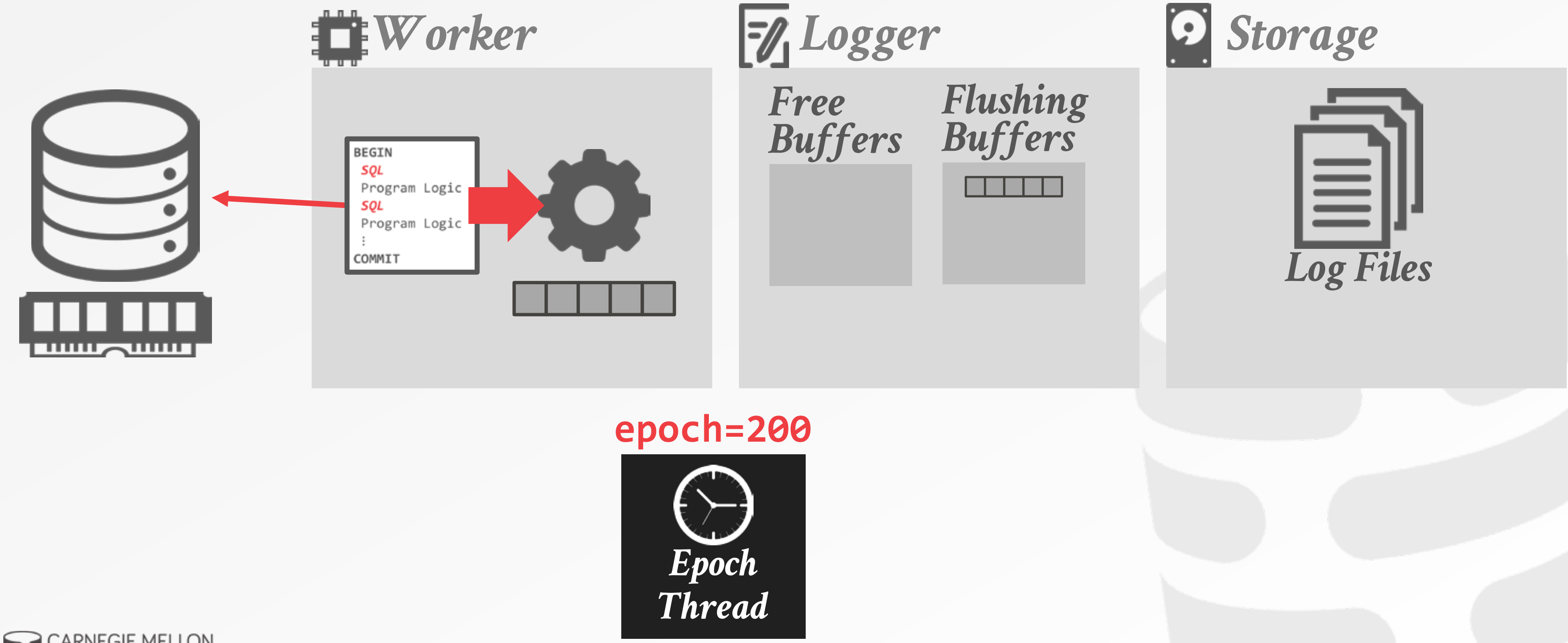
SILOR – ARCHITECTURE



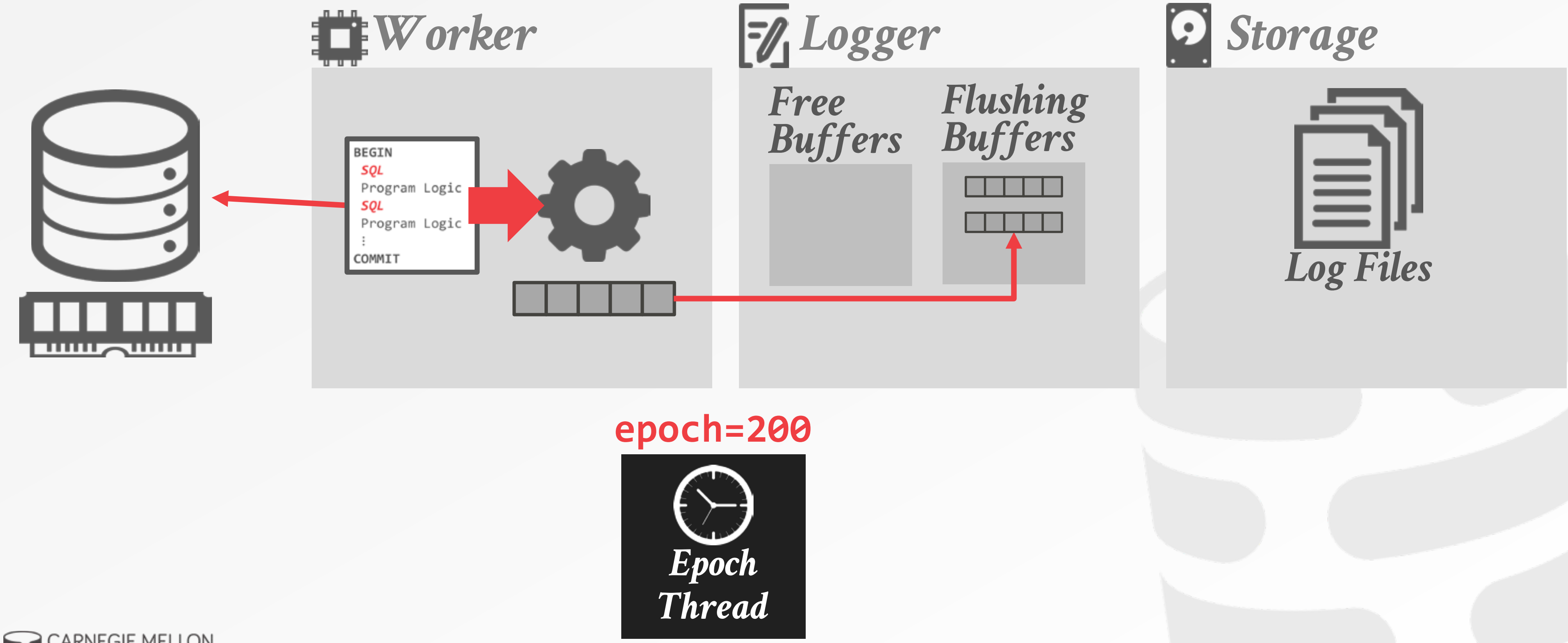
SILOR – ARCHITECTURE



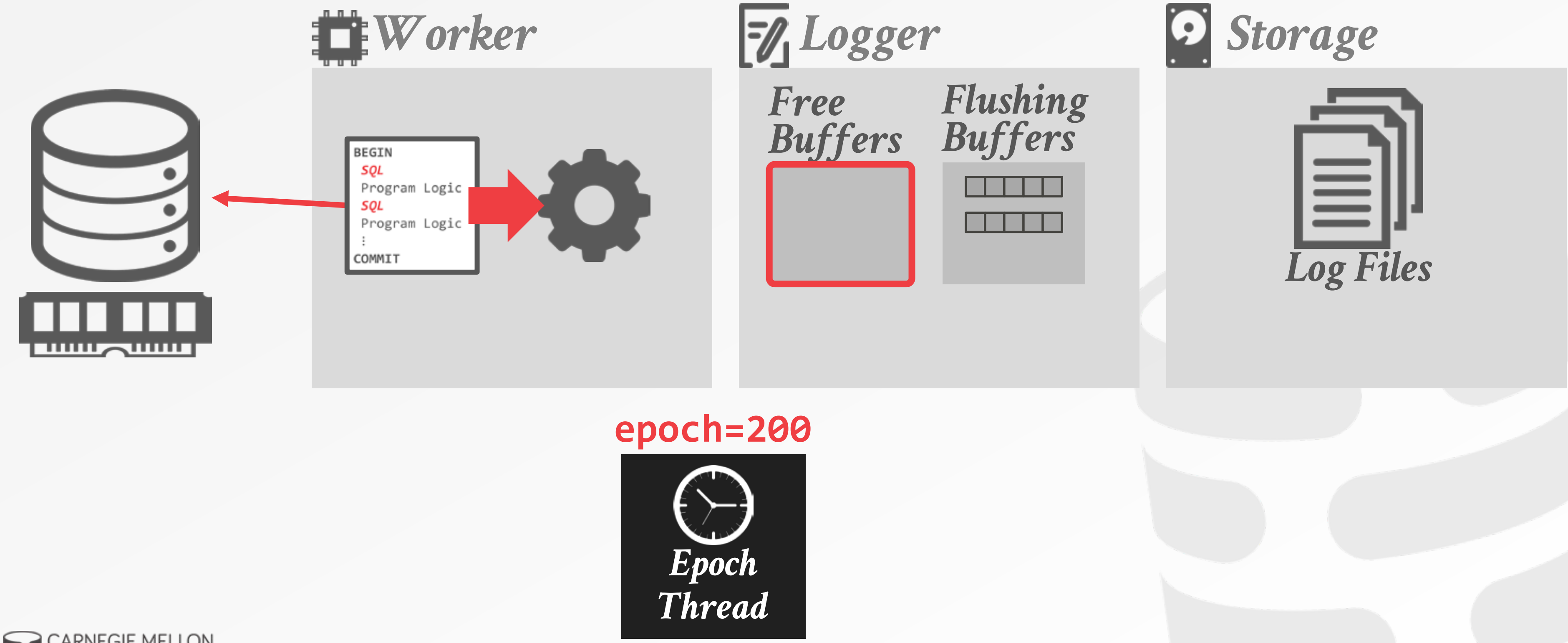
SILOR – ARCHITECTURE



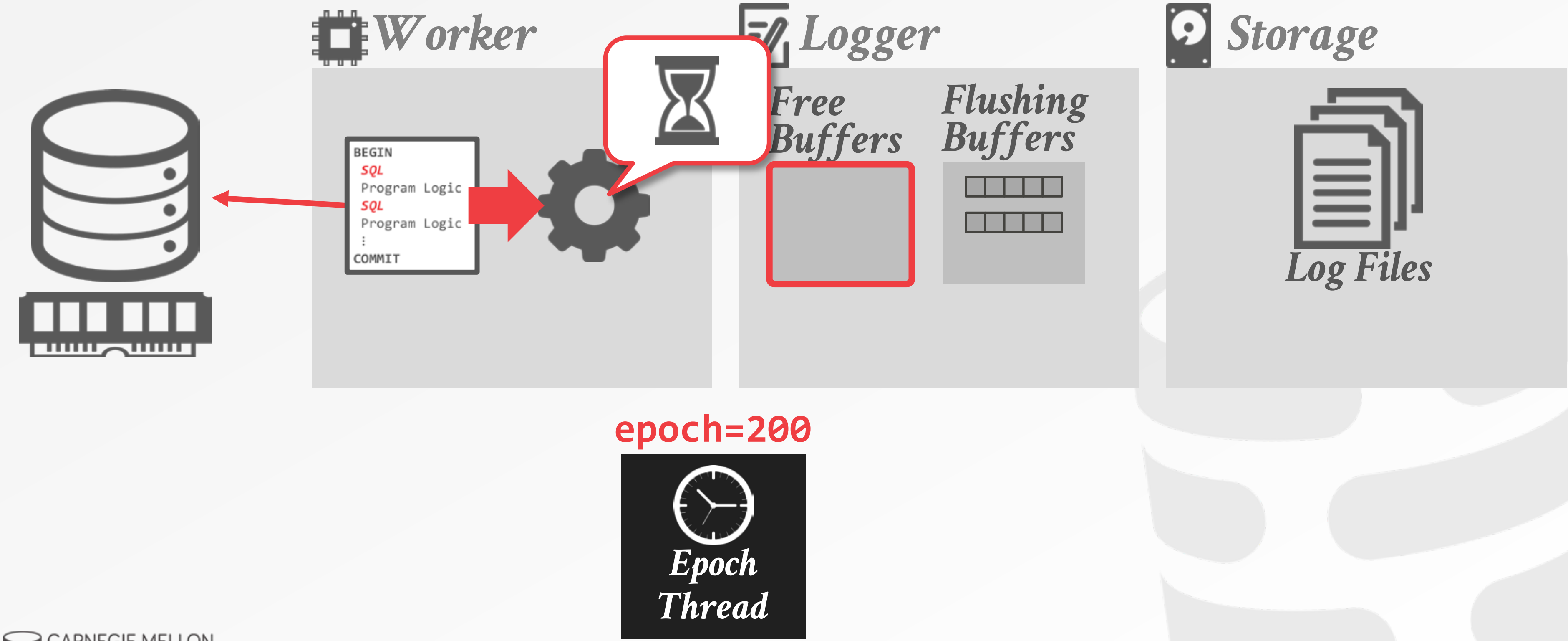
SILOR – ARCHITECTURE



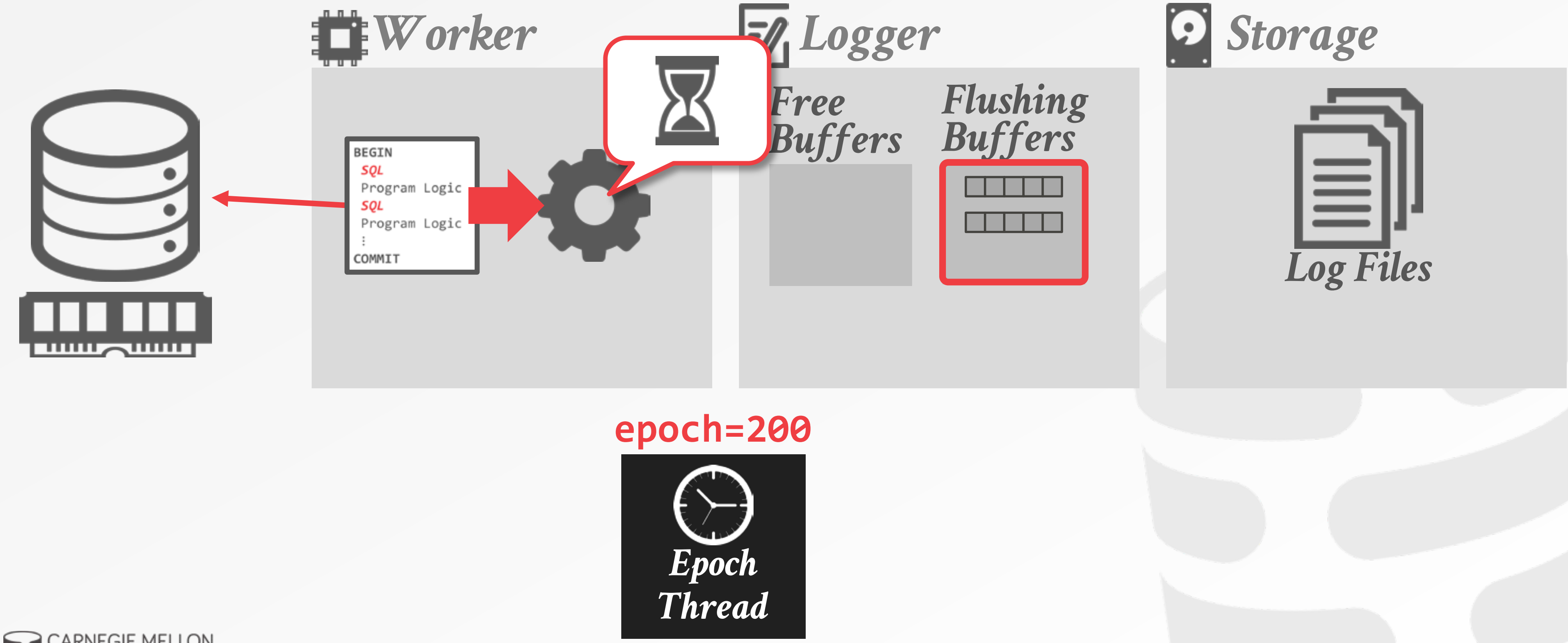
SILOR – ARCHITECTURE



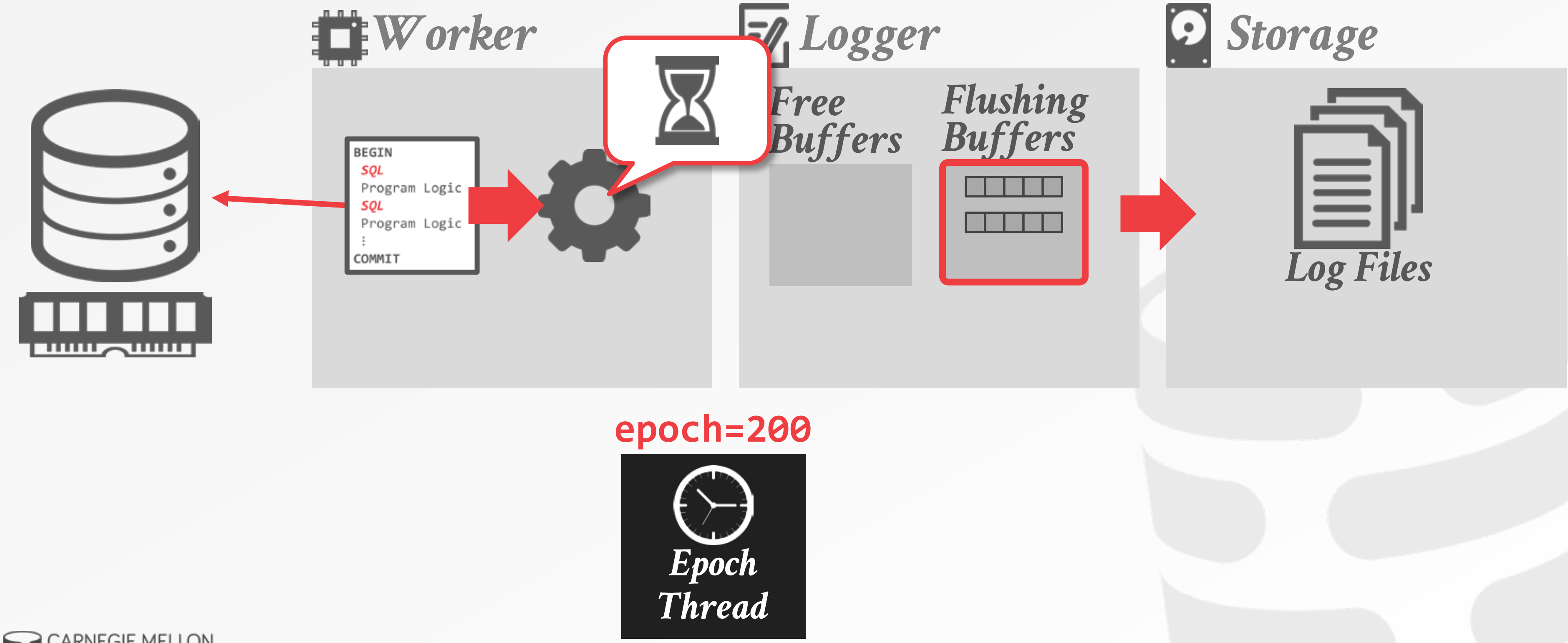
SILOR – ARCHITECTURE



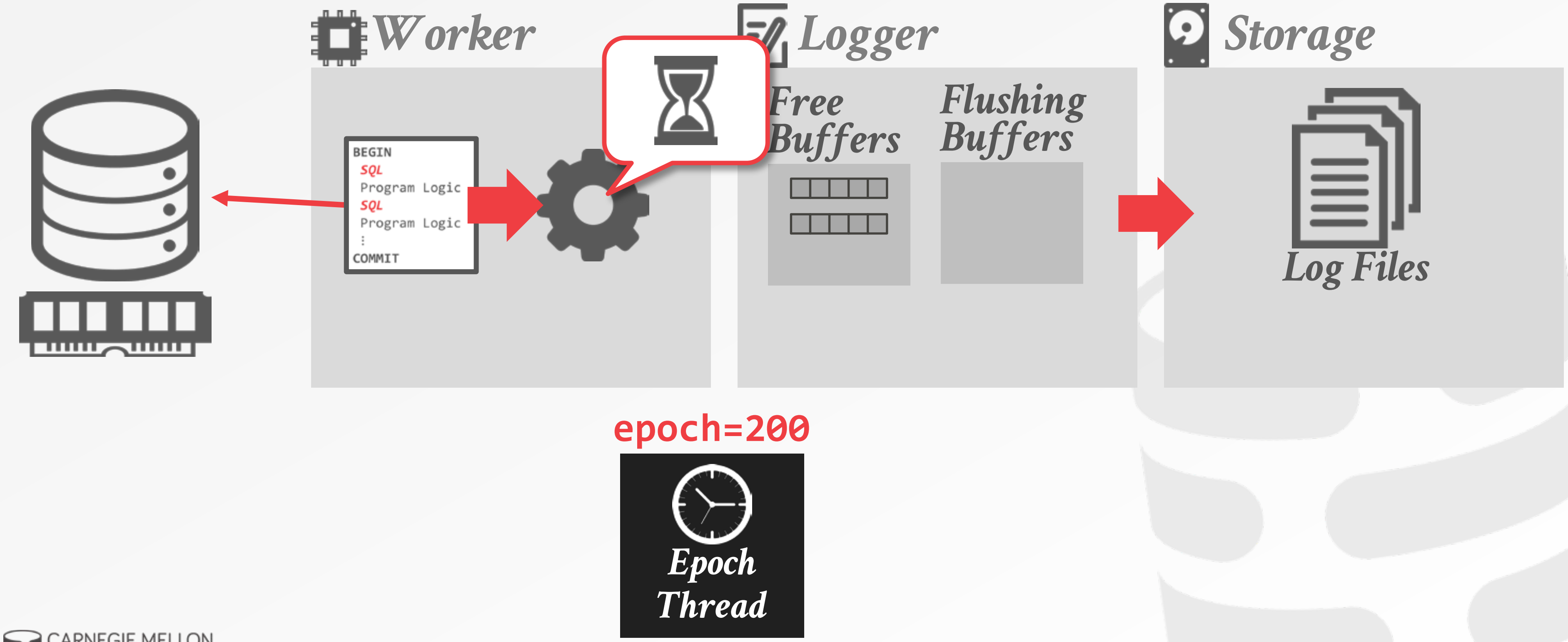
SILOR – ARCHITECTURE



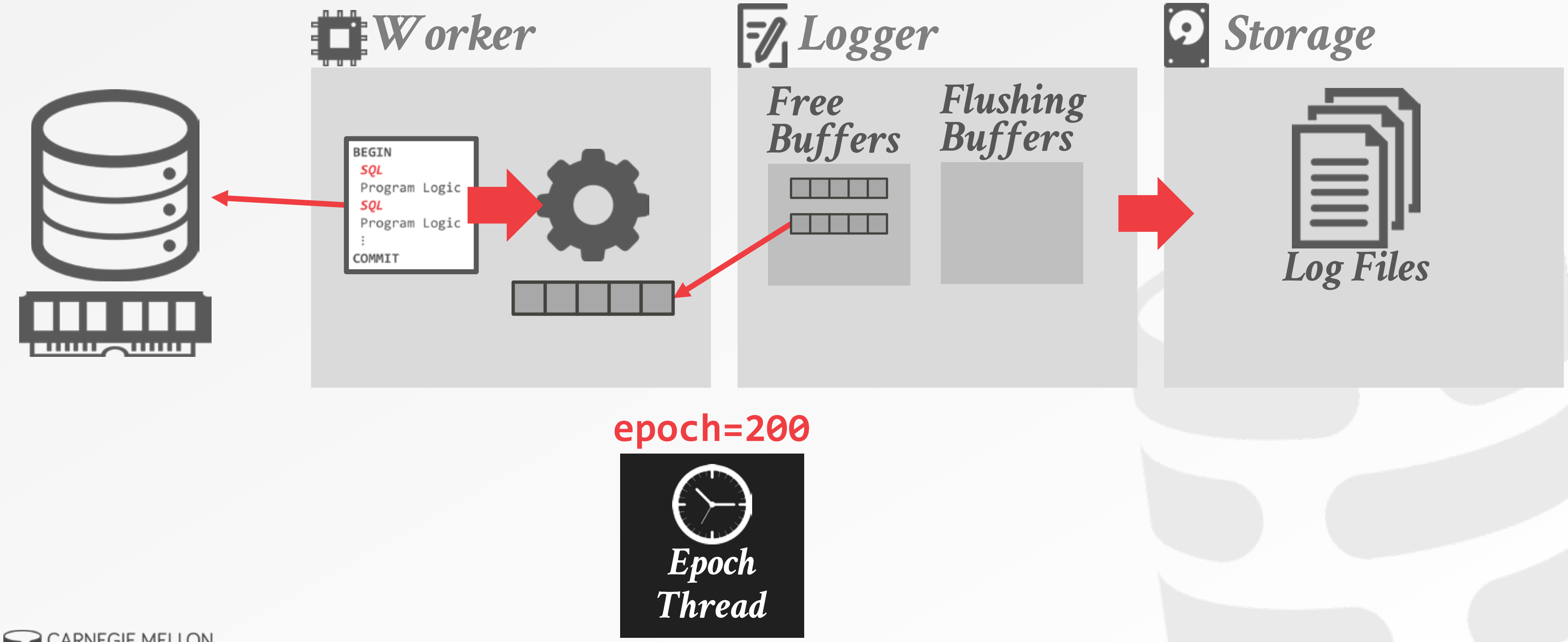
SILOR – ARCHITECTURE



SILOR – ARCHITECTURE



SILOR – ARCHITECTURE



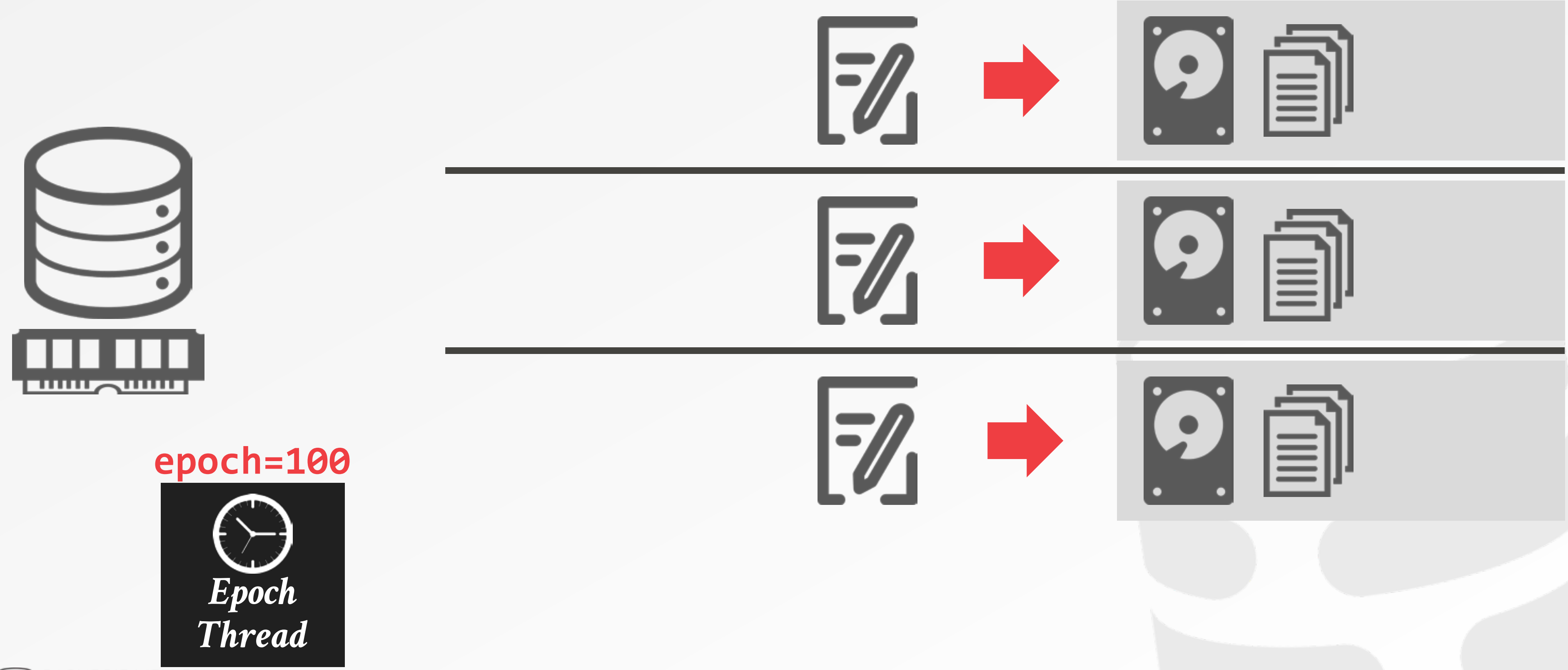
SILOR – PERSISTENT EPOCH

A special logger thread keeps track of the current persistent epoch (***pepoch***)

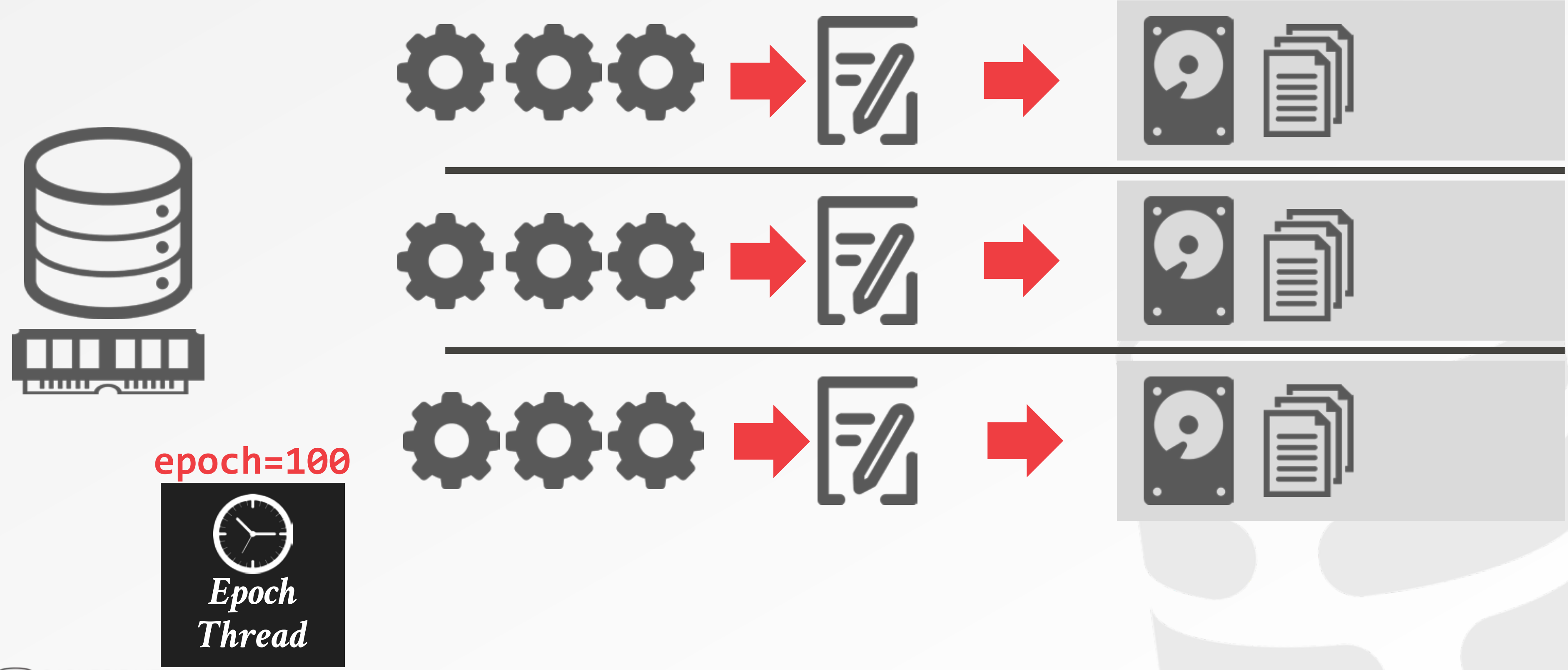
→ Special log file that maintains the highest epoch that is durable across all loggers.

Txns that executed in epoch ***e*** can only release their results when the ***pepoch*** is durable to non-volatile storage.

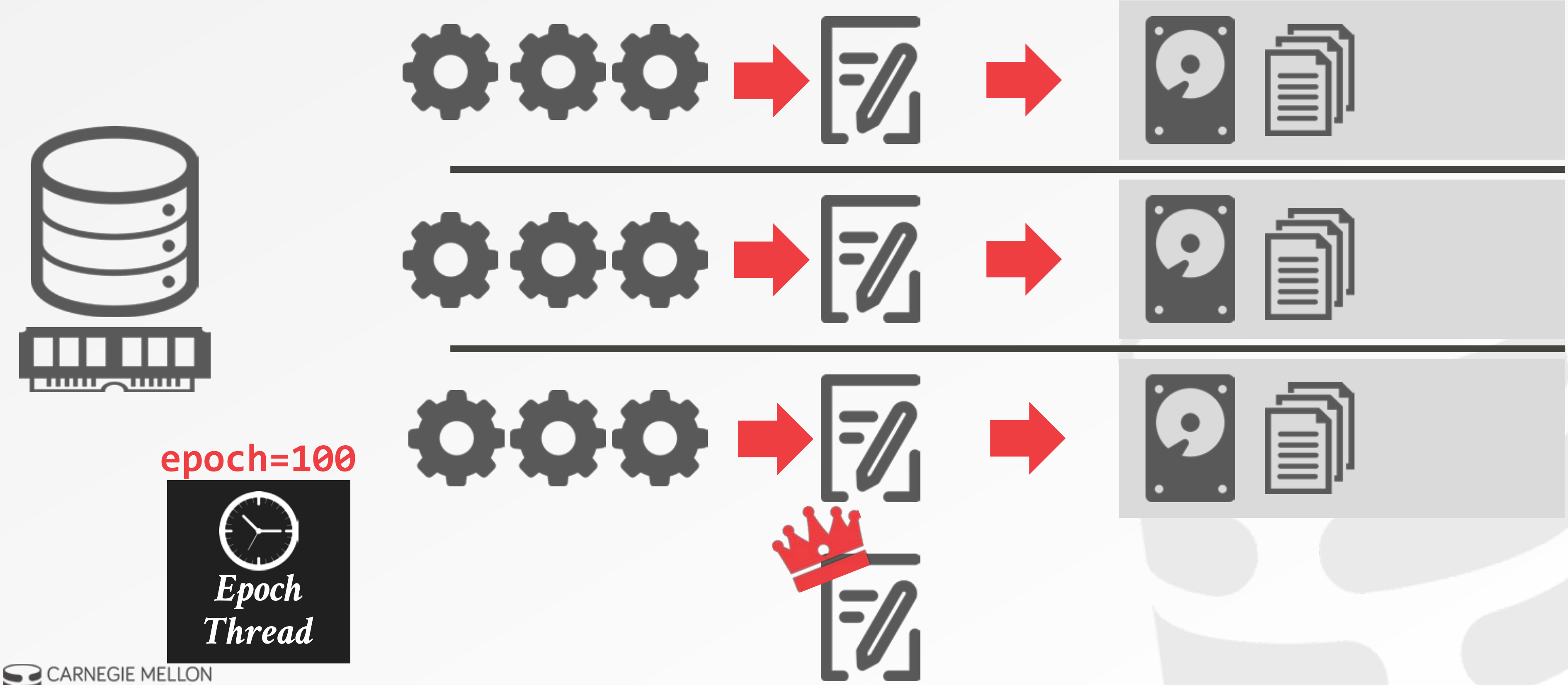
SILOR – ARCHITECTURE



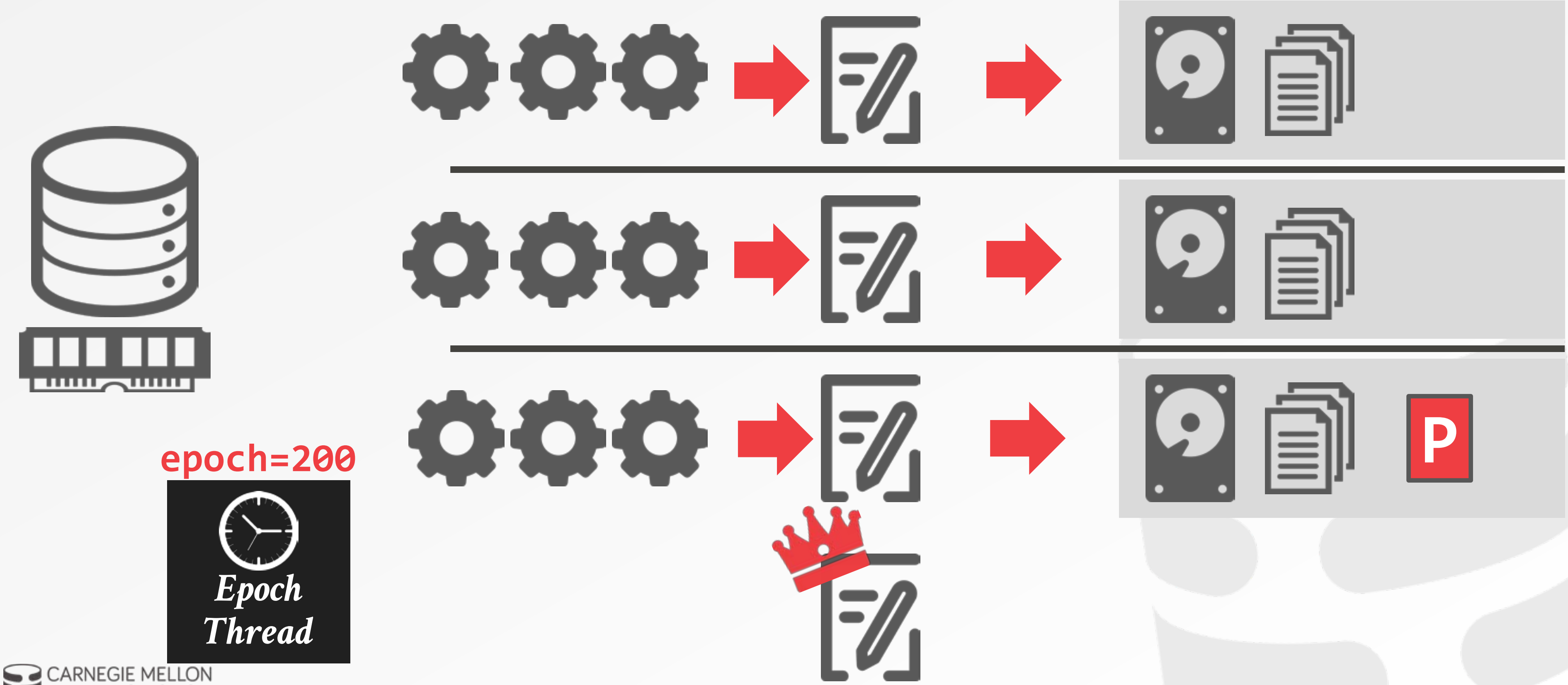
SILOR – ARCHITECTURE



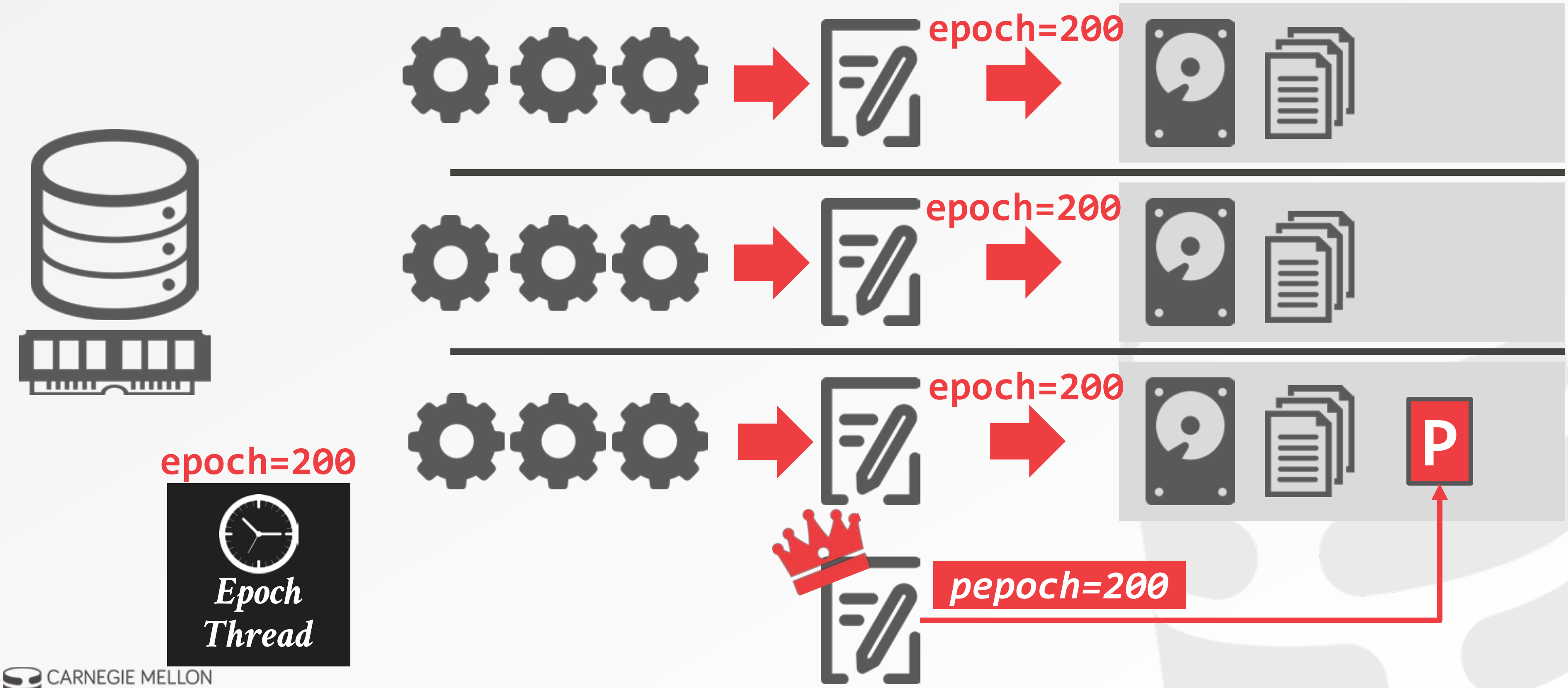
SILOR – ARCHITECTURE



SILOR – ARCHITECTURE



SILOR – ARCHITECTURE



SILOR – RECOVERY PROTOCOL

Phase #1: Load Last Checkpoint

→ Install the contents of the last checkpoint that was saved into the database.
→ All indexes have to be rebuilt.

Phase #2: Replay Log

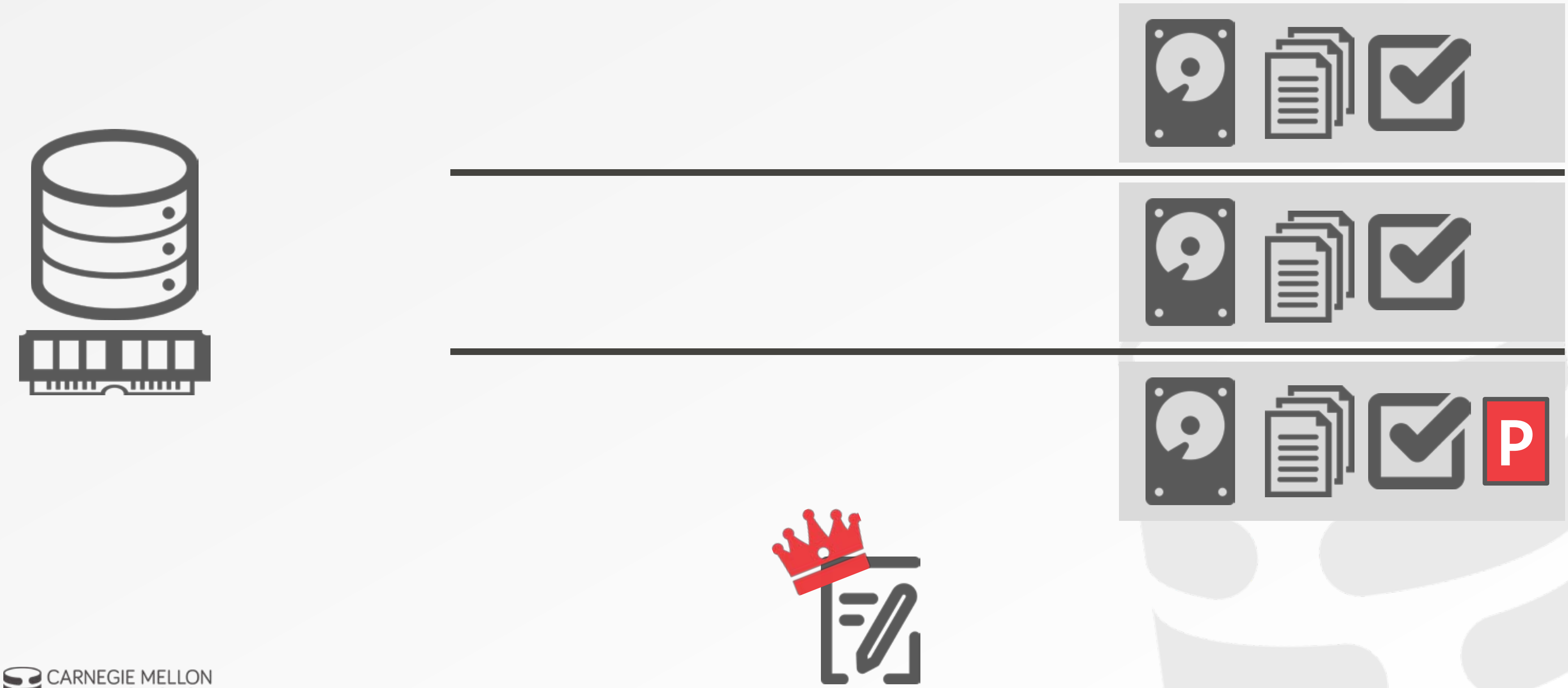
→ Process logs in reverse order to reconcile the latest version of each tuple.

LOG RECOVERY

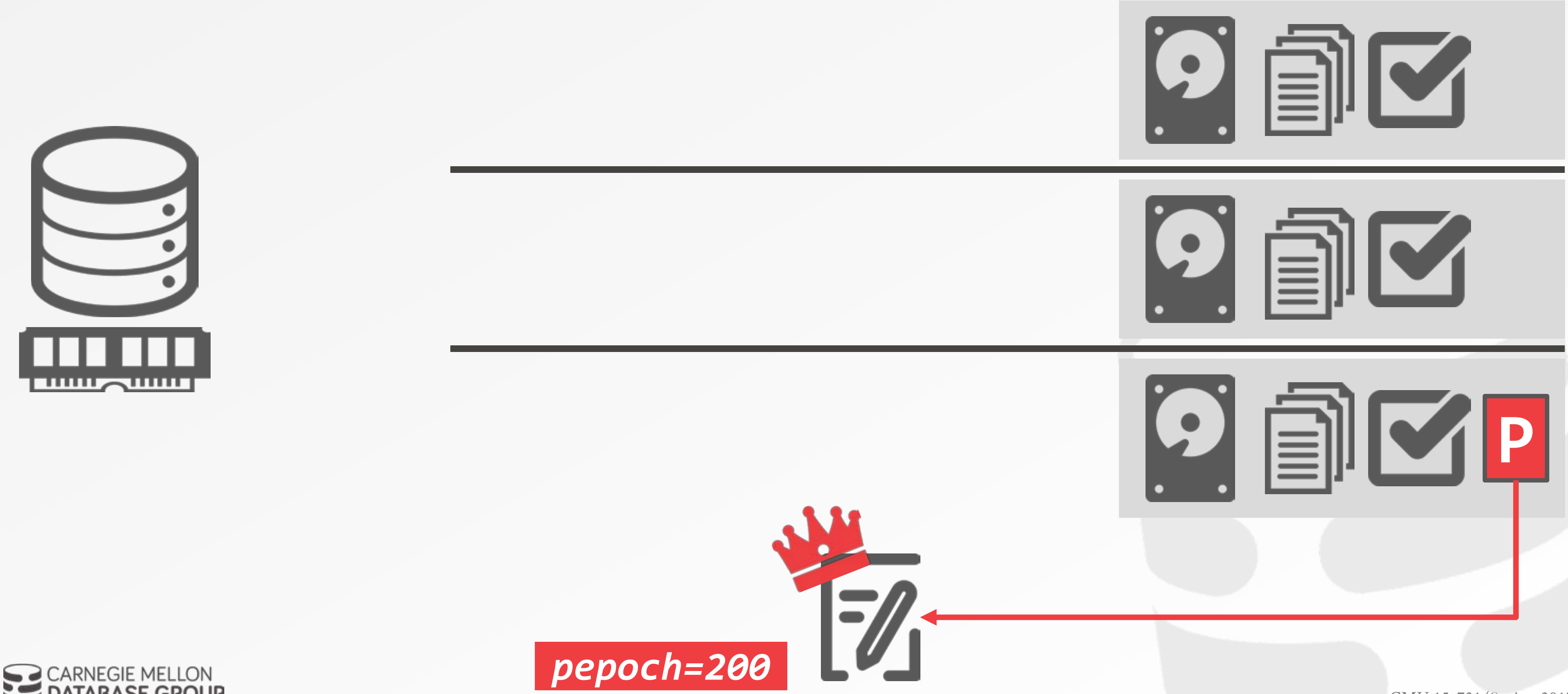
First check the ***pepoch*** file to determine the most recent persistent epoch.
→ Any log record from after the ***pepoch*** is ignored.

Log files are processed from newest to oldest.
→ Value logging is able to be replayed in any order.
→ For each log record, the thread checks to see whether the tuple already exists.
→ If it does not, then it is created with the value.
→ If it does, then the tuple's value is overwritten only if the log TID is newer than tuple's TID.

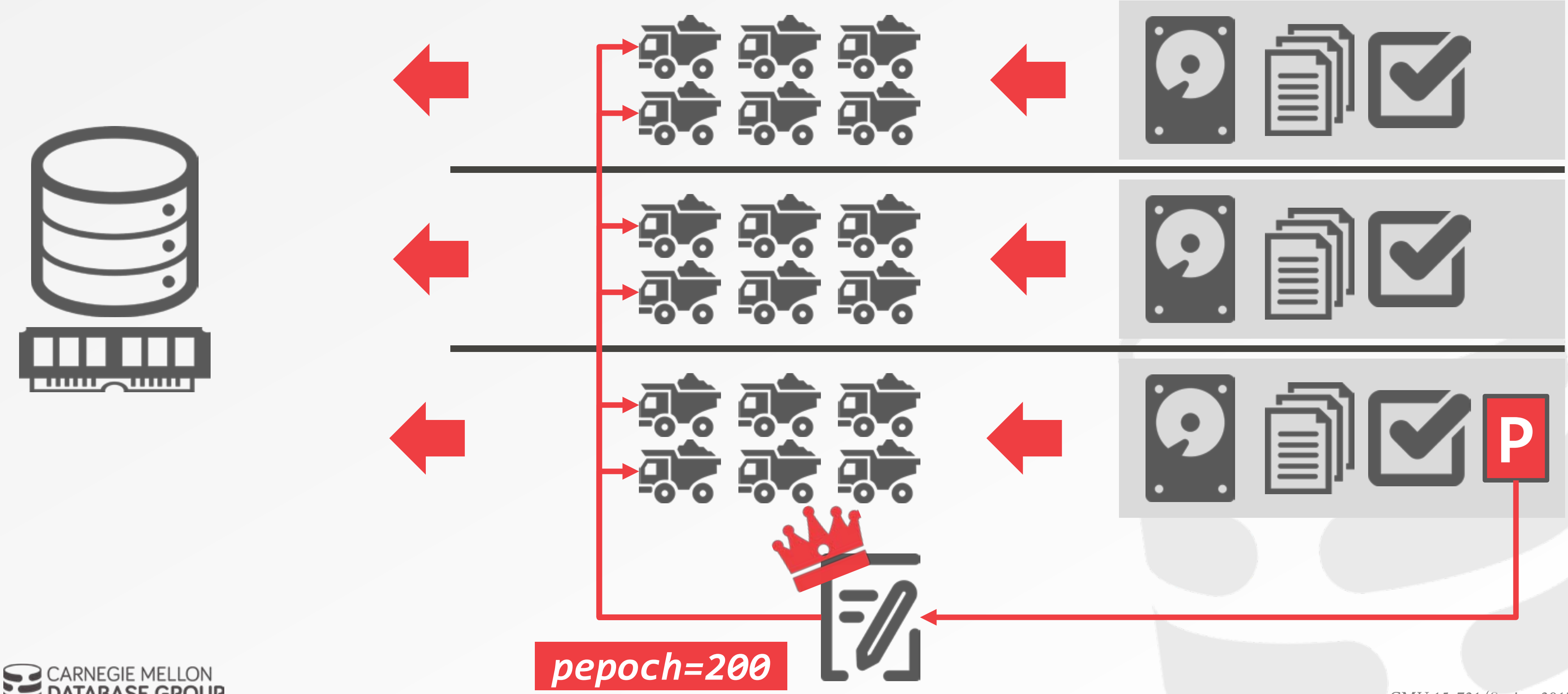
SILOR – RECOVERY PROTOCOL



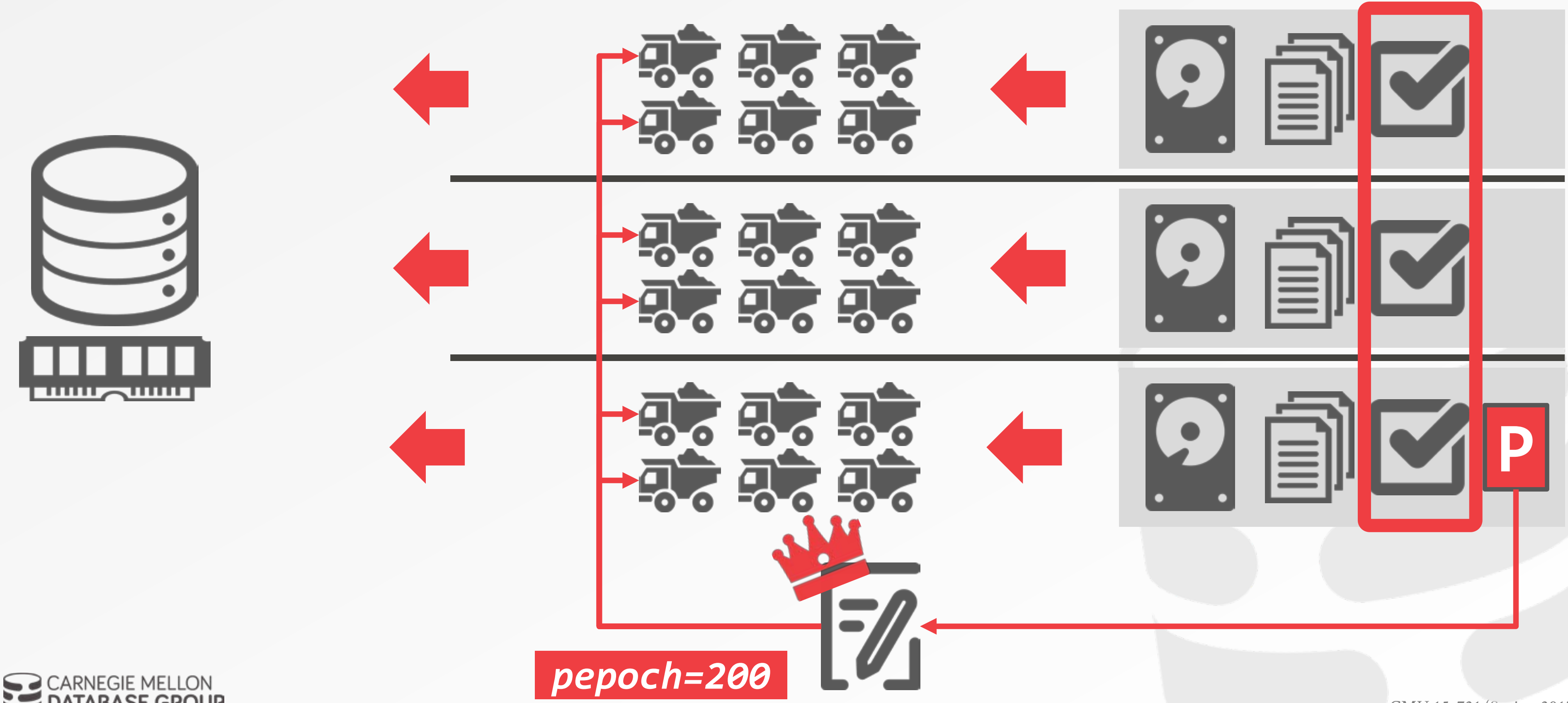
SILOR – RECOVERY PROTOCOL



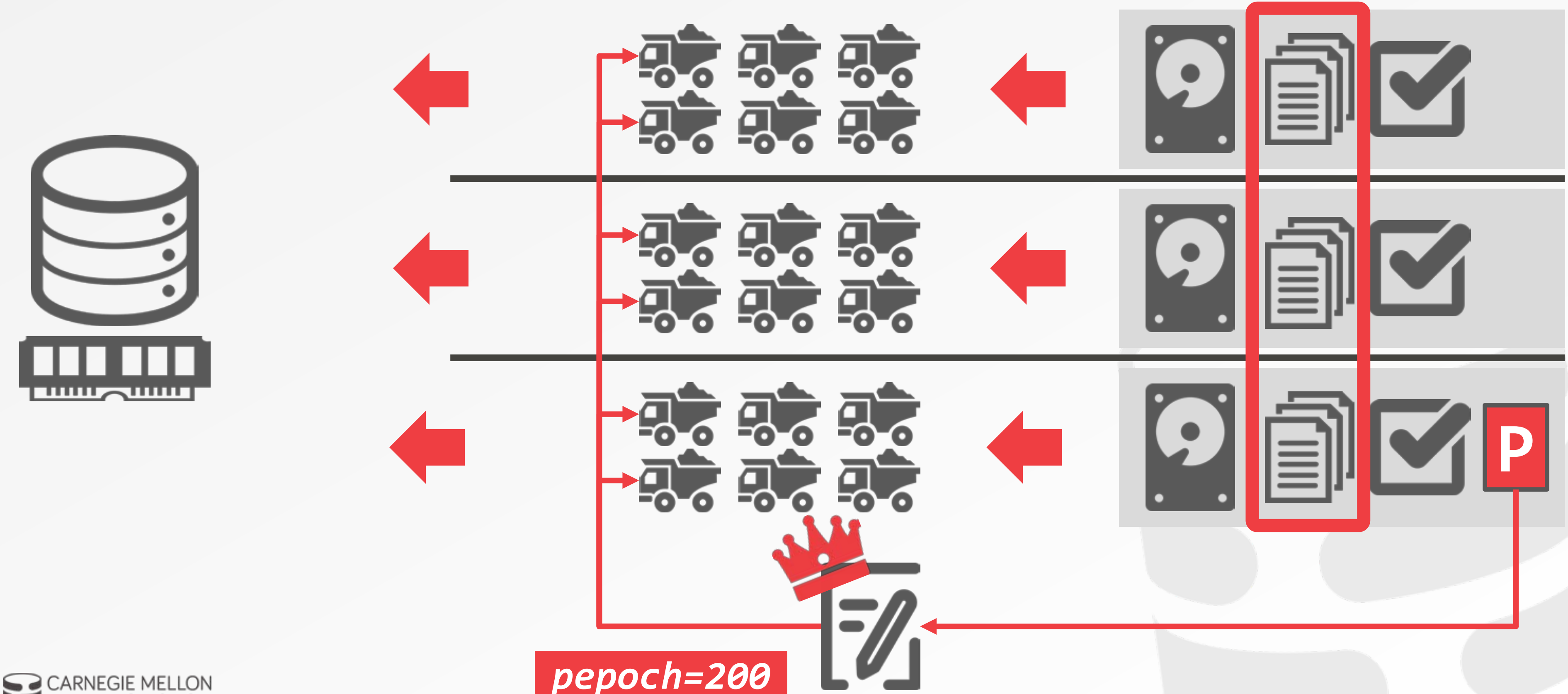
SILOR – RECOVERY PROTOCOL



SILOR – RECOVERY PROTOCOL



SILOR – RECOVERY PROTOCOL



OBSERVATION

The txn ids generated at runtime are enough to determine the serial order on recovery.

This is why SiloR does not need to maintain separate log sequence numbers for each entry.

EVALUATION

Comparing Silo performance with and without logging and checkpoints

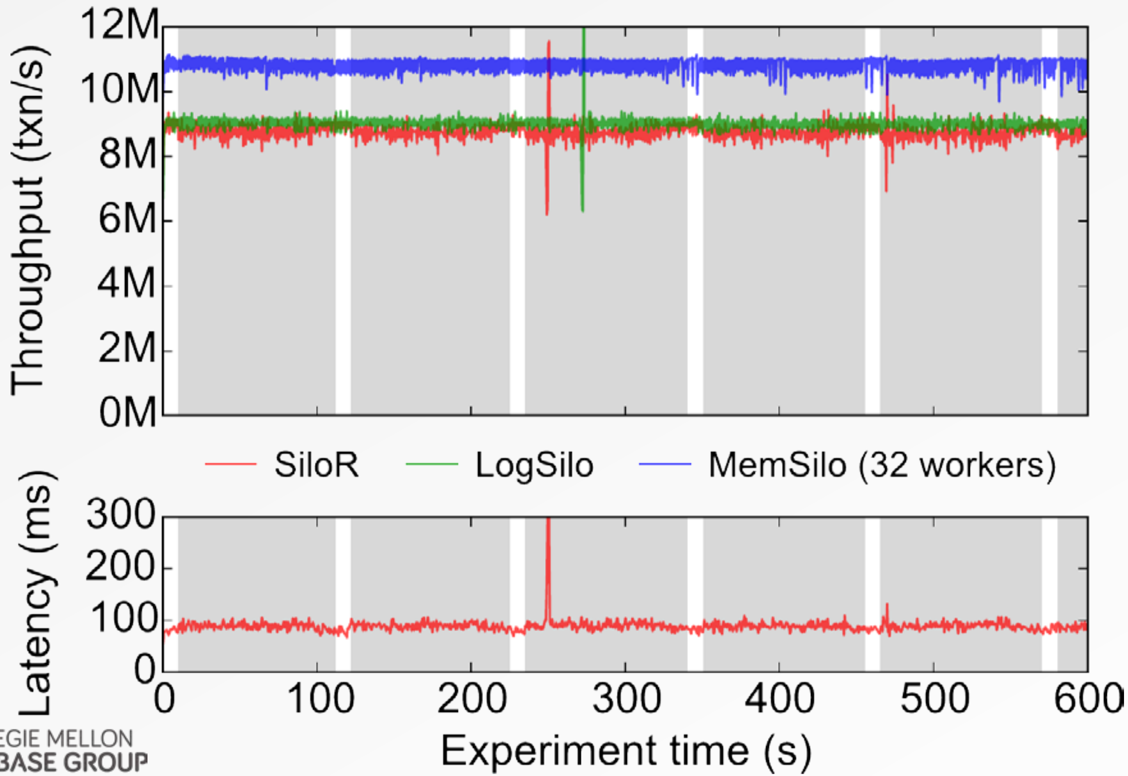
YCSB + TPC-C Benchmarks

Hardware:

→ Four Intel Xeon E7-4830 CPUs (8 cores per socket)
→ 256 GB of DRAM
→ Three Fusion ioDrive2
→ RAID-5 Disk Array

YCSB-A

70% Reads / 30% Writes



Average Throughput

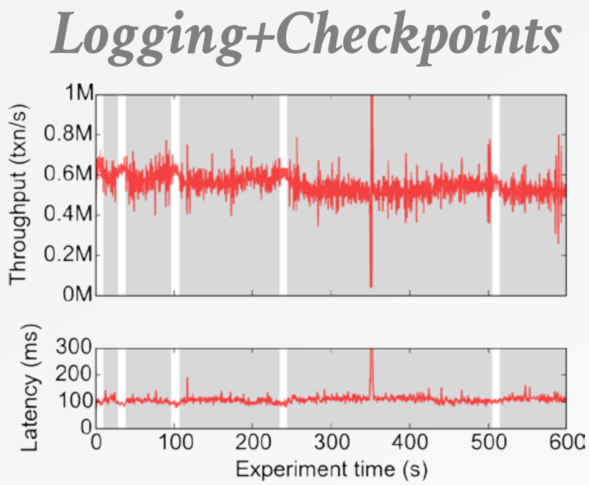
SiloR: 8.76M txns/s

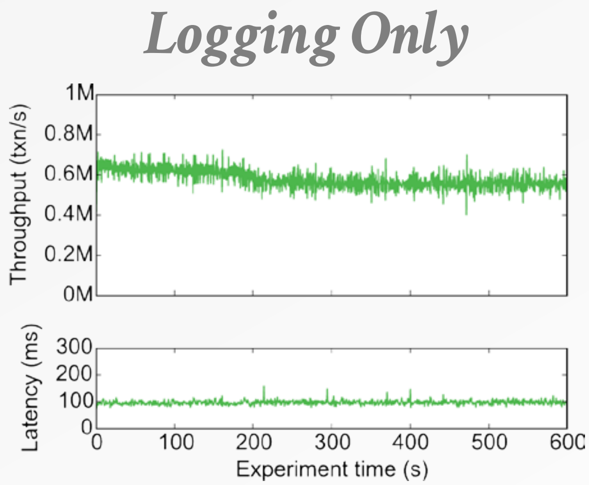
LogSilo: 9.01M txns/s

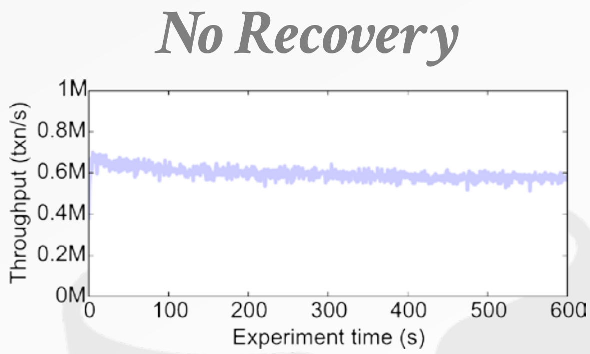
MemSilo: 10.83M txns/s

TPC-C

28 workers, 4 loggers, 4 checkpoint threads







Average Throughput

SiloR: 548K txns/s

LogSilo: 575K txns/s

MemSilo: 592 txns/s

RECOVERY TIMES

		Recovered Database	Checkpoint	Log	Total
YCSB	Size	43.2 GB	36 GB	64 GB	100 GB
	Recovery	-	33 sec	73 sec	106 sec
TPC-C	Size	72.2 GB	16.7 GB	180 GB	195.7 GB
	Recovery	-	17 sec	194 sec	211 sec

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.

`A=100`

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.

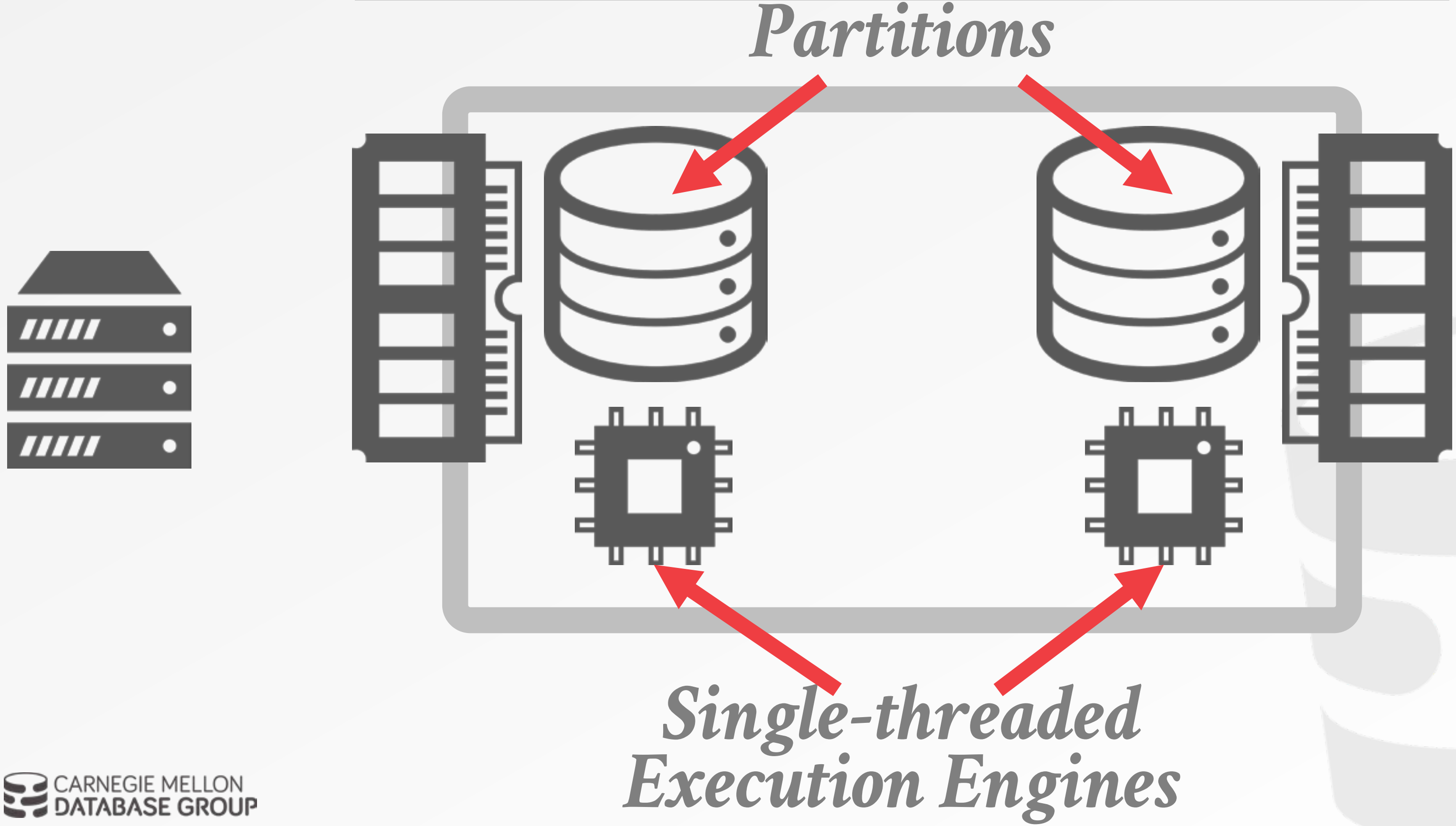
→ The txn logic is deterministic.

Txn #1 `A = A + 1`

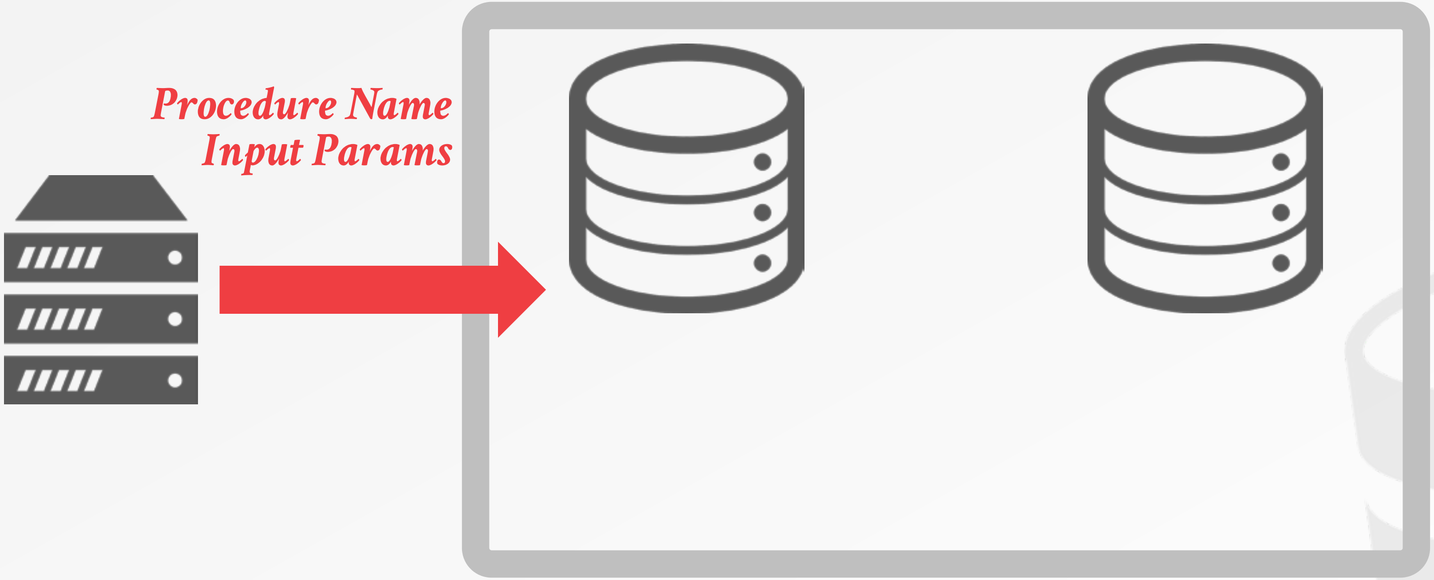
Txn #2 `A = A × NOW()`

Txn #3 `A = A - 5`

VOLTDB – ARCHITECTURE



VOLTDB – ARCHITECTURE



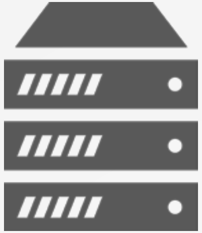
VoteCount:

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

InsertVote:

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

Proc
In



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

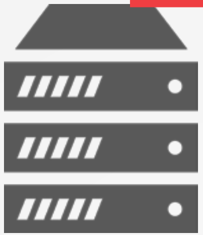
VoteCount:

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

InsertVote:

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

Proc



```
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 = ?;
```

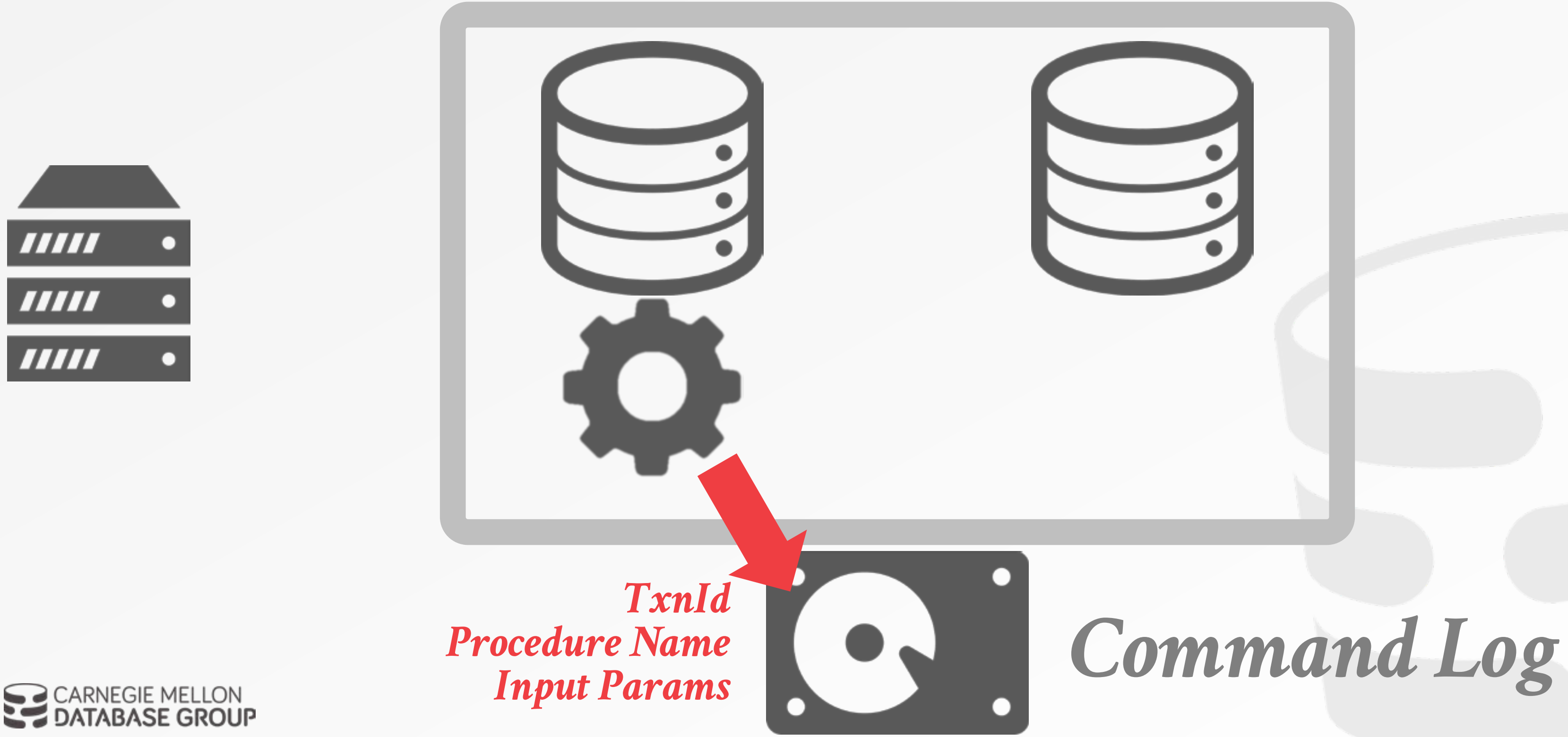
InsertVote:

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

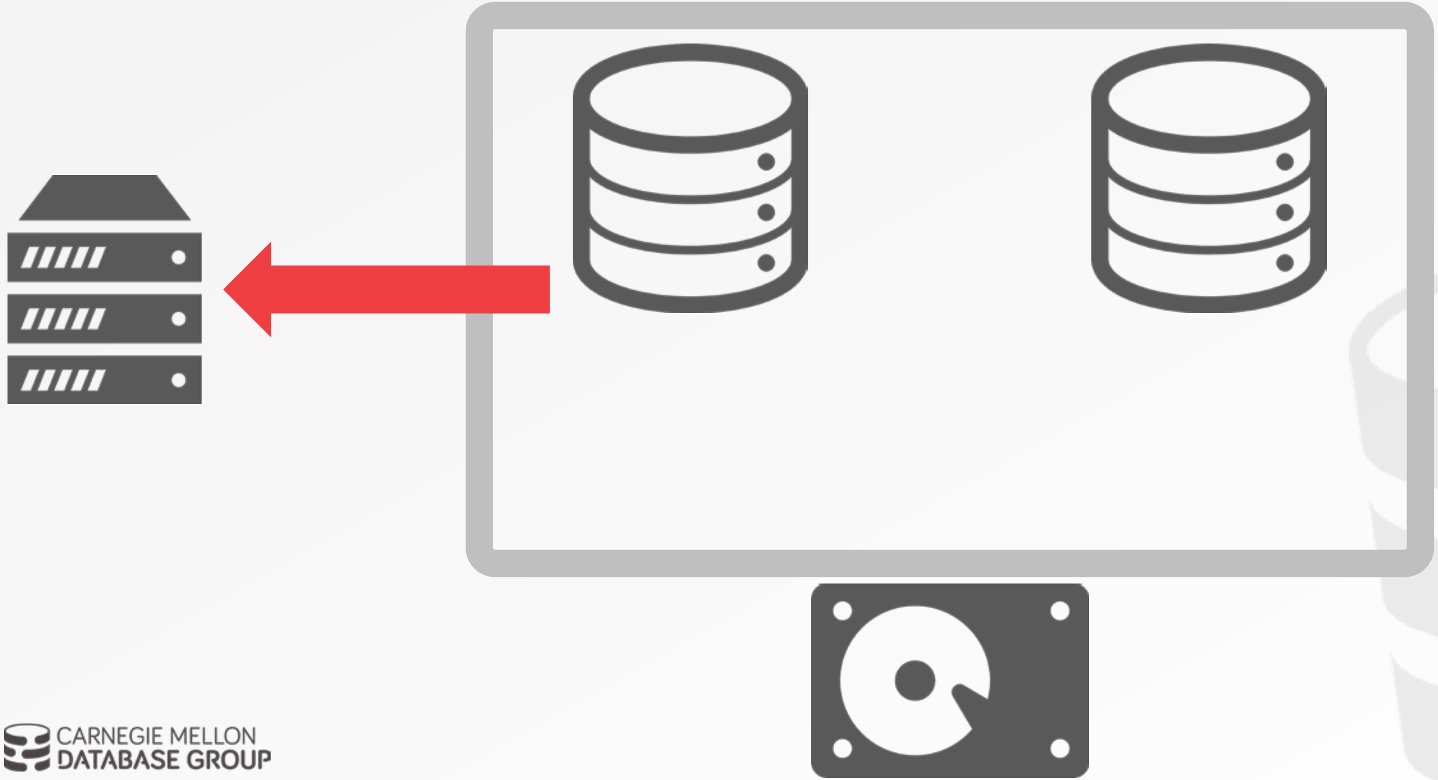
```
run(phoneNum, contestantId, currentTime) {
  result = execute(VoteCount, phoneNum);
  if (result > MAX_VOTES) {
    return (ERROR);
  }
  execute(InsertVote, phoneNum,
                      contestantId,
                      currentTime);
  return (SUCCESS);
}
```

Proc
In

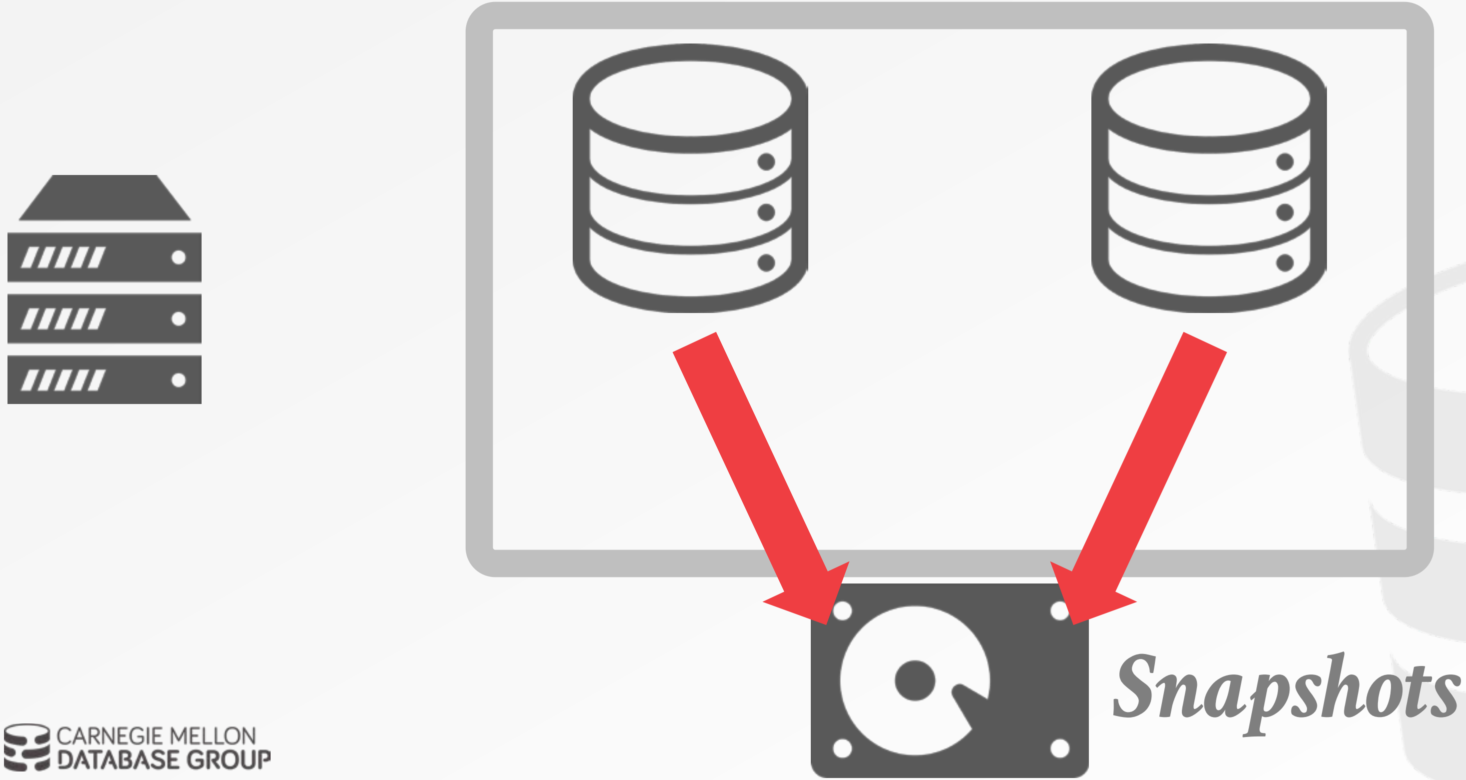
VOLTDB – ARCHITECTURE



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 – RECOVERY PROTOCOL

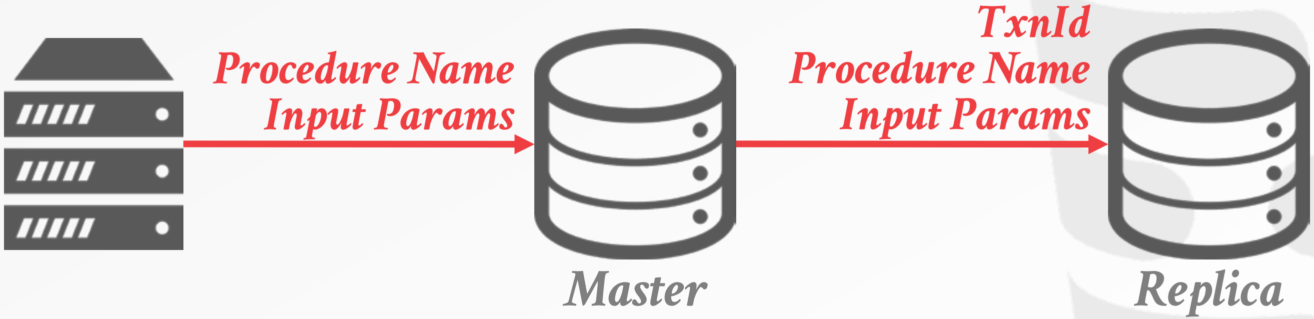
The DBMS loads in the last complete checkpoint from disk.

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.

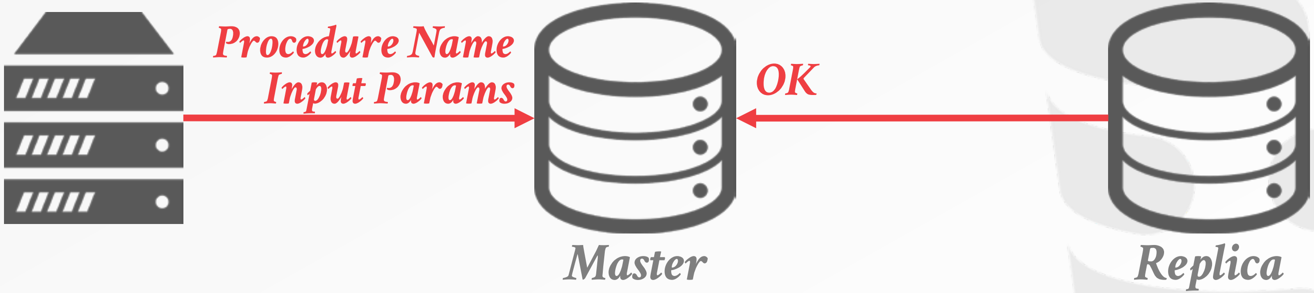
→ DBMS does not need to use **Two-Phase Commit**



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.

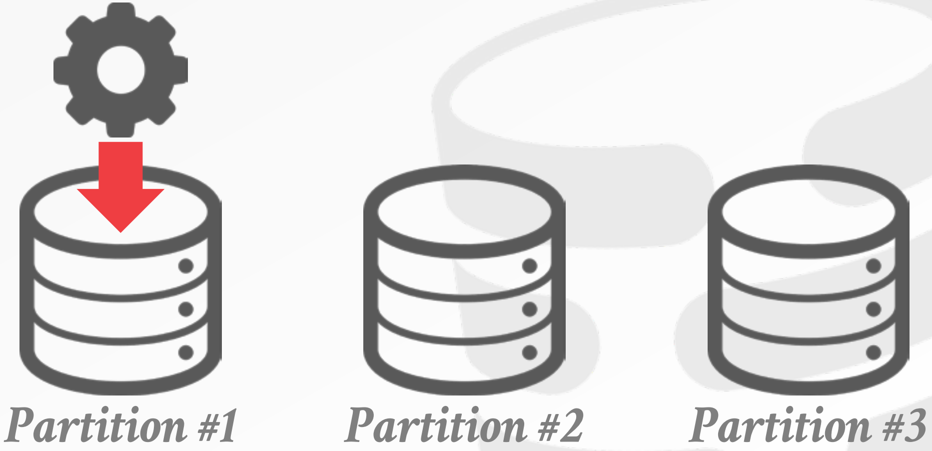
```
X ← SELECT X FROM P2
if (X == true) {
  Y ← UPDATE P2 SET Y = Y+1
} else {
  Y ← UPDATE P3 SET Y = Y+1
}
return (Y)
```

Partition #1 *Partition #2* *Partition #3*

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.

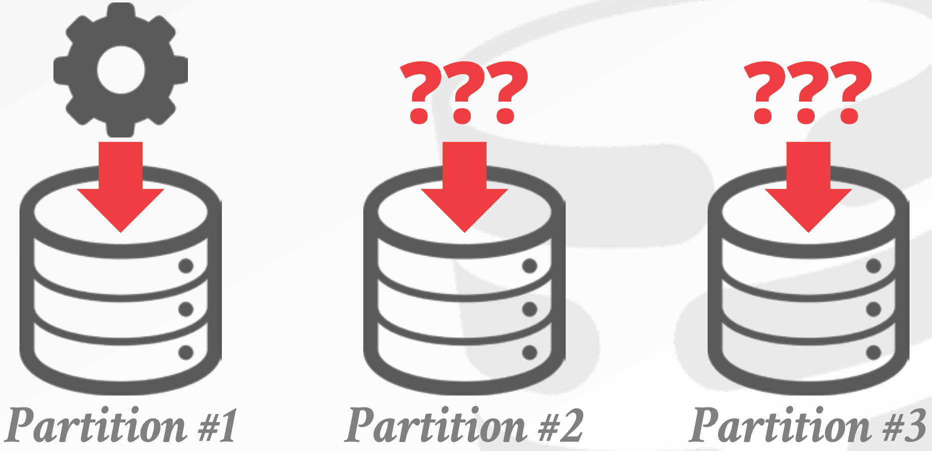
```
X ← SELECT X FROM P2
if (X == true) {
  Y ← UPDATE P2 SET Y = Y+1
} else {
  Y ← UPDATE P3 SET Y = Y+1
}
return (Y)
```



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)
```



PARTING THOUGHTS

Physical logging is a general purpose approach that supports all concurrency control schemes.

Logical logging is faster but not universal.

NEXT CLASS

Checkpoint Schemes

Facebook's Fast Restarts