Logging Protocols

@Andy_Pavlo // 15-721 // Spring 2018
Today’s Agenda

Logging Schemes
Crash Course on ARIES
Physical Logging
Command Logging
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'

<table>
<thead>
<tr>
<th>NAME</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>O.D.B.</td>
<td>$100</td>
</tr>
<tr>
<td>El-P</td>
<td>$666</td>
</tr>
<tr>
<td>Andy</td>
<td>$888</td>
</tr>
</tbody>
</table>

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 = 'Andy'
```

<table>
<thead>
<tr>
<th>NAME</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>O.D.B.</td>
<td>$110</td>
</tr>
<tr>
<td>EL-P</td>
<td>$732</td>
</tr>
<tr>
<td>Andy</td>
<td>$888</td>
</tr>
</tbody>
</table>
**LOGICAL LOGGING EXAMPLE**

**Logical Log**

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

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

<table>
<thead>
<tr>
<th>NAME</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>O.D.B.</td>
<td>$110</td>
</tr>
<tr>
<td>EL-P</td>
<td>$732</td>
</tr>
<tr>
<td>Andy</td>
<td>$888</td>
</tr>
</tbody>
</table>
Logical Log

**UPDATE employees**

**SET salary = salary * 1.10**

**UPDATE employees**

**SET salary = 900**

**WHERE name = 'Andy'**

<table>
<thead>
<tr>
<th>NAME</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>O.D.B.</td>
<td>$110</td>
</tr>
<tr>
<td>EL-P</td>
<td>$732</td>
</tr>
<tr>
<td>Andy</td>
<td>$900</td>
</tr>
</tbody>
</table>

Logical Log
**Logical Logging Example**

**Logical Log**

```
UPDATE employees SET salary = salary * 1.10
UPDATE employees SET salary = 900 WHERE name = 'Andy'
```

<table>
<thead>
<tr>
<th>NAME</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>O.D.B.</td>
<td>$110</td>
</tr>
<tr>
<td>EL-P</td>
<td>$732</td>
</tr>
<tr>
<td>Andy</td>
<td>$990</td>
</tr>
</tbody>
</table>
LOGICAL LOGGING EXAMPLE

**Logical Log**

UPDATE employees SET salary = salary * 1.10

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

<table>
<thead>
<tr>
<th>NAME</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>O.D.B</td>
<td>$110</td>
</tr>
<tr>
<td>EL-P</td>
<td>$732</td>
</tr>
<tr>
<td>Andy</td>
<td>$990</td>
</tr>
</tbody>
</table>
**LOGICAL LOGGING EXAMPLE**

**Logical Log**

```
UPDATE employees
SET salary = salary * 1.10

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

<table>
<thead>
<tr>
<th>NAME</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>O.D.B.</td>
<td>$110</td>
</tr>
<tr>
<td>EL-P</td>
<td>$732</td>
</tr>
<tr>
<td>Andy</td>
<td>$990</td>
</tr>
</tbody>
</table>

```
UPDATE employees
SET salary = salary * 1.10

UPDATE employees
SET salary = 900 WHERE
name = 'Andy'
```
The “gold standard” for physical logging & recovery in a disk-oriented DBMS is **ARIES**.

→ Algorithms 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.
DISK-ORIENTED LOGGING & RECOVERY

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

→ Algorithms for **Recovery** and **Isolation Exploiting Semantics**

→ 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.
→ Each log record is assigned a unique identifier (LSN).

**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 – RECOVERY PHASES**

**Phase #1: Analysis**
→ Read the WAL to identify dirty pages in the buffer pool and active txns at the time of the crash.

**Phase #2: Redo**
→ Repeat all actions starting from an appropriate point in the log.
→ Log redo steps in case of crash during recovery.

**Phase #3: Undo**
→ Reverse the actions of txns that did not commit before the crash.
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

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>

Master Record

pageLSN
A=99  B=5  C=12
Log Sequence Numbers

WAL (Tail)

\[\text{015: } <T5 \text{ begin}> \]
\[\text{016: } <T5, A, 99, 88> \]
\[\text{017: } <T5, B, 5, 10> \]
\[\text{018: } <T5 \text{ commit}> \]

Non-Volatile Storage

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

Buffer Pool

\begin{align*}
\text{pageLSN} & \quad A=99 \quad B=5 \quad C=12 \\
\text{flushedLSN} & \end{align*}

Master Record

\begin{align*}
\text{pageLSN} & \quad A=99 \quad B=5 \quad C=12 \\
\end{align*}
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

Master Record
pageLSN
A=99 B=5 C=12
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

Master Record
A=99 B=5 C=12
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
DISK-ORIENTED DBMS OVERHEAD

Measured CPU Instructions

- 34% for B-TREE KEYS
- 16% for BUFFER POOL
- 16% for LATCHING
- 12% for LOCKING
- 16% for LOGGING
- 7% for REAL WORK

OLTP THROUGH THE LOOKING GLASS, AND WHAT WE FOUND THERE
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.
→ Battery-backed DRAM is large / finnicky
→ Real NVM is coming...

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

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.
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 ('Prashanth', 'Andy')
```

```
Txn#1001
[people, 888, (isLame=true)]
[people, 999, (isLame=true)]
```
SILOR – ARCHITECTURE

Worker

Logger

Storage

BEGIN
Sql
Program Logic
Sql
Program Logic
; Comm

Free Buffers

Flushing Buffers

Log Files

epoch=100

Epoch Thread

epoch=100
SILOR – ARCHITECTURE

Worker

Logger

Free Buffers

Flushing Buffers

Storage

Log Files

epoch=100

BEGIN
  Sql
  Program Logic
  Sql
  Program Logic
  COMMIT

Epoch Thread

epoch=100
**SILOR – ARCHITECTURE**

![Diagram of SILOR architecture]

- **Worker**: Begins SQL, Program Logic, and COMMIT.
- **Logger**: Free Buffers and Flushing Buffers.
- **Storage**: Log Files.

Log Records

**Log Records**

**Epoch = 100**

**Epoch Thread**
**SILOR – ARCHITECTURE**

**Worker**
- Begins SQL
- Continues Program Logic
- Commits

**Logger**
- **Free Buffers**
- **Flushing Buffers**

**Storage**
- Log Files

**Epoch**
- epoch=100
SILOR – ARCHITECTURE

Worker

Logger
  Free Buffers
  Flushing Buffers

Storage
  Log Files

epoch=100
**SILOR – ARCHITECTURE**

- **Worker**
  - BEGIN
  - Sql
  - Program Logic
  - COMMIT

- **Logger**
  - Free Buffers
  - Flushing Buffers

- **Storage**
  - Log Files

**epoch=200**

**Epoch Thread**

CMU 15-721 (Spring 2018)
**SILOR – ARCHITECTURE**

- **Worker**
  - Begins SQL program logic
  - Ends with COMMIT

- **Logger**
  - Free Buffers
  - Flushing Buffers

- **Storage**
  - Log Files

**Epoch**

**Thread**

**Log File**

**epoch=200**
SILOR – ARCHITECTURE

Worker

BEGIN
Sql
Program Logic
Sql
Program Logic
;
COMMIT

Logger
Free Buffers
Flushing Buffers

Storage
Log Files

epoch=200
SILOR – ARCHITECTURE

Worker

Logger

Storage

Free Buffers

Flushing Buffers

Log Files

epoch=200

BEGIN
Sql
Program Logic
Sql
Program Logic
:COMMIT

Epoch
Thread

CMU 15-721 (Spring 2018)
SILOR – ARCHITECTURE

**Worker**
- Begins
- Program Logic
- Sql
- Program Logic
- COMMIT

**Logger**
- Free Buffers
- Flushing Buffers

**Storage**
- Log Files

**Epoch Thread**
- epoch=200
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

epoch=100

Epoch Thread
SILOR – ARCHITECTURE

epoch=200

epoch=200

epoch=200

epoch=200

epoch=200

pepoch=200
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 \textit{peepoch} file to determine the most recent persistent epoch.
→ Any log record from after the \textit{peepoch} 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

\[ \text{peepoch} = 200 \]
SILOR – RECOVERY PROTOCOL

peepoch = 200
SILOR – RECOVERY PROTOCOL

peepoch = 200
SILOR – RECOVERY PROTOCOL

peepoch=200
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

Logging+Checkpoints

Logging Only

No Recovery

Average Throughput

SiloR: 548K txns/s
LogSilo: 575K txns/s
MemSilo: 592 txns/s
# Recovery Times

<table>
<thead>
<tr>
<th></th>
<th>Recovered Database</th>
<th>Checkpoint</th>
<th>Log</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>YCSB</strong></td>
<td>Size</td>
<td>43.2 GB</td>
<td>36 GB</td>
<td>64 GB</td>
</tr>
<tr>
<td></td>
<td>Recovery</td>
<td>-</td>
<td>33 sec</td>
<td>73 sec</td>
</tr>
<tr>
<td><strong>TPC-C</strong></td>
<td>Size</td>
<td>72.2 GB</td>
<td>16.7 GB</td>
<td>180 GB</td>
</tr>
<tr>
<td></td>
<td>Recovery</td>
<td>-</td>
<td>17 sec</td>
<td>194 sec</td>
</tr>
</tbody>
</table>
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
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.

\[
\begin{align*}
\text{Txn #1} & : A = A + 1 \\
\text{Txn #2} & : A = A \times 3 \\
\text{Txn #3} & : A = A - 5
\end{align*}
\]
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

<table>
<thead>
<tr>
<th>Txn #1</th>
<th>A = A + 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Txn #2</td>
<td>A = A × 3</td>
</tr>
<tr>
<td>Txn #3</td>
<td>A = A - 5</td>
</tr>
</tbody>
</table>
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** \[ A = A \times \text{NOW( )} \] (highlighted)

**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.

\[
\begin{align*}
\text{Txn #1} & : A = A + 1 \\
\text{Txn #2} & : A = A \times \text{NOW}( ) \\
\text{Txn #3} & : A = A - 5
\end{align*}
\]
VOLTDB – ARCHITECTURE

Partitions

Single-threaded Execution Engines
VOLTDB – ARCHITECTURE

Procedure Name
Input Params
```
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 (?, ?, ?);
```
VOLTDB – ARCHITECTURE

Command Log

TxnId
Procedure Name
Input Params
VOLTDB – ARCHITECTURE
VOLTDB – ARCHITECTURE

Snapshots
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.
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
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.

\[
X \leftarrow \text{SELECT } X \text{ FROM } P2 \\
\text{if } (X == \text{ true}) \{ \\
    Y \leftarrow \text{UPDATE } P2 \text{ SET } Y = Y+1 \\
\} \text{ else } \{ \\
    Y \leftarrow \text{UPDATE } P3 \text{ SET } Y = Y+1 \\
\} \text{ return } (Y)
\]
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.

\[
X \leftarrow \text{SELECT X FROM P2}
\]
\[
\text{if (} X == \text{true}\text{)} \{ \\
\quad Y \leftarrow \text{UPDATE P2 SET Y = Y+1} \\
\} \text{ else } \{ \\
\quad Y \leftarrow \text{UPDATE P3 SET Y = Y+1} \\
\}\] return (Y)
Physical logging is a general purpose approach that supports all concurrency control schemes.

Logical logging is faster but not universal.

Non-volatile memory is coming…
NEXT CLASS

Checkpoint Schemes
Facebook’s Fast Restarts