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**

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

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

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'
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'

<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

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

UPDATE employees SET
salary = salary * 1.10

UPDATE employees SET
salary = 900 WHERE
name = 'Andy'
LOGICAL LOGGING EXAMPLE

**Logical Log**

<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
- SET salary = 900
  WHERE name = 'Andy'

---

CMU 15-721 (Spring 2017)
**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>$110</td>
</tr>
<tr>
<td>EL-P</td>
<td>$732</td>
</tr>
<tr>
<td>Andy</td>
<td>$990</td>
</tr>
</tbody>
</table>

**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'
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 – 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.
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 \textit{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 (\textit{flushedLSN}).

For a page \textit{i} to be written, the DBMS must flush log at least to the point where \textit{pageLSN}_i \leq \textit{flushedLSN}
**LOG SEQUENCE NUMBERS**

**WAL (Tail)**

- 015: `<T5 begin>`
- 016: `<T5, A, 99, 88>`
- 017: `<T5, B, 5, 10>`
- 018: `<T5 commit>`
- ... (more entries)

**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>

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> · · ·

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
**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**

<table>
<thead>
<tr>
<th>pageLSN</th>
<th>flushedLSN</th>
</tr>
</thead>
<tbody>
<tr>
<td>A=99</td>
<td>B=5</td>
</tr>
<tr>
<td>C=12</td>
<td></td>
</tr>
</tbody>
</table>

**Master Record**

<table>
<thead>
<tr>
<th>pageLSN</th>
</tr>
</thead>
<tbody>
<tr>
<td>A=99</td>
</tr>
<tr>
<td>B=5</td>
</tr>
<tr>
<td>C=12</td>
</tr>
</tbody>
</table>
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  C=12
flushedLSN

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

Measured CPU Cycles

- 30% BUFFER POOL
- 30% LOCKING
- 30% RECOVERY
- 12% 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.
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.
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.
**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.
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 thetxn 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 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.

```
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-r------- 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-r------- 1 mysql mysql 4.4G Dec 21 08:38 magneto.log
-rw-r------- 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

**Worker**

```
BEGIN
SQL
Program Logic
SQL
Program Logic
;
COMMIT
```

**Logger**

- Free Buffers
- Flushing Buffers

**Storage**

Log Files

**Epoch**

epoch=100
SILOR – ARCHITECTURE

Worker

Logger

Storage

epoch=100

Log Records
SILOR – ARCHITECTURE

Worker

Logger

Storage

epoch=100
SILOR – ARCHITECTURE

Worker

Logger

Storage

Free Buffers

Flushing Buffers

Log Files

epoch=100

Epoch Thread
SILOR – ARCHITECTURE

Worker

Logger

Storage

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**

epoch=200
SILOR – ARCHITECTURE

Worker

BEGIN SQL Program Logic SQL Program Logic : COMMIT

Free Buffers

Flushing Buffers

Storage

Log Files

epoch=200
epoch=200
SILOR – ARCHITECTURE

Worker → Free Buffers → Flushing Buffers → Storage

epoch=200
SILOR – ARCHITECTURE

Worker

Logger

Storage

epoch=200

Free Buffers

Flushing Buffers

Log Files

CMU 15-721 (Spring 2017)
SILOR – PERSISTENT EPOCH

A special logger thread keeps track of the current persistent epoch (\(pePOCH\))

\[ \rightarrow \text{Special log file that maintains the highest epoch that is durable across all loggers.} \]

Txns that executed in epoch \(e\) can only release their results when the \(pePOCH\) is durable to non-volatile storage.
SILOR – ARCHITECTURE

epoch=100
SILOR – ARCHITECTURE

epoch=100

Epoch Thread
SILOR – ARCHITECTURE

epoch=100

Epoch Thread
SILOR – ARCHITECTURE

Epoch

Thread

epoch=200

epoch=200

epoch=200

epoch=200

epoch=200

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

peepoch=200
SILOR – RECOVERY PROTOCOL

peepoch=200
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> Size</td>
<td>43.2 GB</td>
<td>36 GB</td>
<td>64 GB</td>
<td>100 GB</td>
</tr>
<tr>
<td>Recovery</td>
<td>-</td>
<td>33 sec</td>
<td>73 sec</td>
<td>106 sec</td>
</tr>
<tr>
<td><strong>TPC-C</strong> Size</td>
<td>72.2 GB</td>
<td>16.7 GB</td>
<td>180 GB</td>
<td>195.7 GB</td>
</tr>
<tr>
<td>Recovery</td>
<td>-</td>
<td>17 sec</td>
<td>194 sec</td>
<td>211 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.

\[ A = 100 \]

| Txn #1 | \( A = A + 1 \) |
| Txn #2 | \( A = A \times 3 \) |
| 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.

$A = 100$

Txn #1: $A = A + 1$

Txn #2: $A = A \times 3$

Txn #3: $A = A - 5$

$A = 298$
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.

\[
\begin{align*}
A &= 100 \\
Txn\ #1 &\quad A = A + 1 \\
Txn\ #2 &\quad A = A \times \text{NOW}() \\
Txn\ #3 &\quad A = A - 5
\end{align*}
\]
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.
VOLTDB – ARCHITECTURE

Partitions

Single-threaded Execution Engines
VOLTDB – ARCHITECTURE

Procedure Name
Input Params
Procedure `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 (?, ?, ?);
### Procedure:

```java
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);
}
Snapshots
The DBMS logs the txn command *before* it starts executing once atxn 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
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
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.

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

\[
X \leftarrow \text{SELECT } X \text{ FROM P2} \\
\text{if } (X == \text{true}) \{ \\
\text{Y } \leftarrow \text{UPDATE P2 SET Y = Y+1} \\
\} \text{ else } \{ \\
\text{Y } \leftarrow \text{UPDATE P3 SET Y = Y+1} \\
\} \text{ return (Y)}
\]
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 ← 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.
Checkpoinat Schemes
Facebook’s Fast Restarts