Recovery algorithms are techniques to ensure database consistency, 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.
OBSERVATION

Many of 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 finally here as of 2019!

This hardware is still not widely available, so we want to use existing SSD/HDDs.
Many of 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 finally here as of 2019!

This hardware is still not widely available, so we want to use existing SSD/HDDs.
IN-MEMORY DATABASE RECOVERY

Slightly easier than in a disk-oriented DBMS because the system must do less work:

→ Do **not** track dirty pages in case of crash during recovery.
→ Do **not** store undo records (only need redo).
→ Do **not** log changes to indexes.

But the DBMS is still stymied by the slow sync time of non-volatile storage.
TODAY’S AGENDA

Logging Schemes
Checkpoint Protocols
Restart Protocols
LOGGING SCHEMES

Approach #1: 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.

Approach #2: Logical Logging
→ Record the high-level operations executed by txns.
→ Example: The UPDATE, DELETE, and INSERT queries invoked by a txn.
LOG FLUSHING

Approach #1: All-at-Once Flushing
→ Wait until a txn has fully committed before writing out log records to disk.
→ Do not need to store abort records because uncommitted changes are never written to disk.

Approach #2: Incremental Flushing
→ Allow the DBMS to write a txn's log records to disk before it has committed.
GROUP COMMIT

Batch together log records from multiple txns and flush them together with a single $\texttt{fsync}$.

$\rightarrow$ Logs are flushed either after a timeout or when the buffer gets full.

$\rightarrow$ 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 if it does not return results to the client before becoming durable.

Other txns that speculatively read data updated by a pre-committed txn become dependent on it and must wait for their predecessor’s log records to reach disk.
OBSERVATION

The delta records in an MVCC DBMS are like the log records generated in physical logging.

Instead of generating separate data structures for MVCC and logging, what if the DBMS could use the same information?
MSSQL CONSTANT TIME RECOVERY

Physical logging protocol that uses the DBMS's MVCC *time-travel* table as the recovery log.

→ The version store is a persistent append-only storage area that is flushed to disk.

→ Leverage versions meta-data to "undo" updates without having to process undo records in WAL.

Recovery time is measured based on the number of version store records that must be read from disk.
### MSSQL: VERSION STORE

#### Main Table

<table>
<thead>
<tr>
<th>COL1</th>
<th>COL2</th>
<th>POINTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₄</td>
<td>xxx</td>
<td>$444</td>
</tr>
<tr>
<td>B₂</td>
<td>yyy</td>
<td>$22</td>
</tr>
<tr>
<td>C₅</td>
<td>zzz</td>
<td>$5</td>
</tr>
</tbody>
</table>

#### Version Store

<table>
<thead>
<tr>
<th>COL1</th>
<th>COL2</th>
<th>POINTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₂</td>
<td>xxx</td>
<td>$222</td>
</tr>
<tr>
<td>B₁</td>
<td>yyy</td>
<td>$11</td>
</tr>
<tr>
<td>A₃</td>
<td>xxx</td>
<td>$333</td>
</tr>
<tr>
<td>C₄</td>
<td>zzz</td>
<td>$4</td>
</tr>
</tbody>
</table>
## MSSQL: VERSION STORE

### Main Table

<table>
<thead>
<tr>
<th>COL1</th>
<th>COL2</th>
<th>POINTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>A4</td>
<td>xxx</td>
<td>$444</td>
</tr>
<tr>
<td>B2</td>
<td>yyy</td>
<td>$22</td>
</tr>
<tr>
<td>C5</td>
<td>zzz</td>
<td>$5</td>
</tr>
</tbody>
</table>

### Version Store

<table>
<thead>
<tr>
<th>COL1</th>
<th>COL2</th>
<th>POINTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>A2</td>
<td>xxx</td>
<td>$222</td>
</tr>
<tr>
<td>B1</td>
<td>yyy</td>
<td>$11</td>
</tr>
<tr>
<td>A3</td>
<td>xxx</td>
<td>$333</td>
</tr>
<tr>
<td>C4</td>
<td>zzz</td>
<td>$4</td>
</tr>
<tr>
<td>A4</td>
<td>xxx</td>
<td>$444</td>
</tr>
</tbody>
</table>
# MSSQL: VERSION STORE

## Main Table

<table>
<thead>
<tr>
<th>COL1</th>
<th>COL2</th>
<th>POINTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>A5</td>
<td>xxx</td>
<td>$555</td>
</tr>
<tr>
<td>B2</td>
<td>yyy</td>
<td>$22</td>
</tr>
<tr>
<td>C5</td>
<td>zzz</td>
<td>$5</td>
</tr>
</tbody>
</table>

## Version Store

<table>
<thead>
<tr>
<th>COL1</th>
<th>COL2</th>
<th>POINTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>A2</td>
<td>xxx</td>
<td>$222</td>
</tr>
<tr>
<td>B1</td>
<td>yyy</td>
<td>$11</td>
</tr>
<tr>
<td>A3</td>
<td>xxx</td>
<td>$333</td>
</tr>
<tr>
<td>C4</td>
<td>zzz</td>
<td>$4</td>
</tr>
<tr>
<td>A4</td>
<td>xxx</td>
<td>$444</td>
</tr>
</tbody>
</table>
MSSQL CTR: PERSISTENT VERSION STORE

Approach #1: In-row Versioning
→ Store small updates to a tuple as a delta record embedded with the latest version in the main table.
→ Same as Cicada "best-effort in-lining" technique.

Approach #2: Off-row Versioning
→ Specialized data table to store the old versions that is optimized for concurrent inserts.
→ Versions from all tables are stored in a single table.
→ Store redo records for inserts on this table in WAL.
MSSQL CTR: IN-ROW VERSIONING

Store small updates to a tuple as a delta record embedded with the latest version in the main table.

The delta record space is **not** pre-allocated per tuple in a disk-oriented DBMS.
MSSQL CTR: IN-ROW VERSIONING

Store small updates to a tuple as a delta record embedded with the latest version in the main table.

The delta record space is **not** pre-allocated per tuple in a disk-oriented DBMS.
MSSQL CTR: IN-ROW VERSIONING

Store small updates to a tuple as a delta record embedded with the latest version in the main table.

The delta record space is **not** pre-allocated per tuple in a disk-oriented DBMS.
MSSQL CTR: RECOVERY PROTOCOL

Phase #1: Analysis
→ Identify the state of every txn in the log.

Phase #2: Redo
→ Recover the main table and version store to their state at the time of the crash.
→ The database is available and online after this phase.

Phase #3: Undo
→ Mark uncommitted txns as aborted in a global txn state map so that future txns ignore their versions.
→ Incrementally remove older versions via logical revert.
MSSQL CTR: LOGICAL REVERT

Approach #1: Background Cleanup
→ GC thread scans all blocks and removes reclaimable versions.
→ If latest version in main table is from an aborted txn, then it will move the committed version back to main table.

Approach #2: Aborted Version Overwrite
→ Txns can overwrite the latest version in the main table if that version is from an aborted txn.
SILO

In-memory OLTP DBMS from Harvard/MIT.
→ Single-versioned OCC with epoch-based GC.
→ Same authors of the Masstree.
→ Eddie Kohler is unstoppable.

SiloR uses physical logging + checkpoints to ensure durability of txns.
→ It achieves high performance by parallelizing all aspects of logging, checkpointing, and recovery.
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.

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

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

Worker

Logger

Free Buffers

Flushing Buffers

Storage

Log Files

epoch=100

Epoch Thread

epoch=100
SILOR: ARCHITECTURE

Worker

Logger

Storage

Free Buffers

Epoch Thread

epoch=100

Flushing Buffers

Log Files
SILOR: ARCHITECTURE

Worker

Logger

Storage

Log Records

Free Buffers
Flushing Buffers

epoch=100

Epoch
Thread

Storage

Log Files
**SILOR: ARCHITECTURE**

Diagram showing the interaction between Worker, Logger, and Storage components. **Worker** processes SQL and Program Logic and generates Log Records. **Logger** manages Free Buffers and Flushing Buffers. **Storage** handles Log Files.

- **Worker** processes SQL and Program Logic and generates Log Records.
- **Logger** manages Free Buffers and Flushing Buffers.
- **Storage** handles Log Files.

**Log Records** are generated as part of the processing flow.

**Epoch=100** is highlighted with an overlay.
SILOR: ARCHITECTURE

**Worker**

- BEGIN
- Sql
- Program Logic
- Sql
- Program Logic
- COMMIT

**Logger**

- Free Buffers
- Flushing Buffers

**Storage**

- Log Files

epoch=100

*Epoch Thread*
SILOR: ARCHITECTURE

Worker

Logger

Storage

epoch=200

epoch=200

Log Files

Free Buffers

Flushing Buffers
SILOR: ARCHITECTURE

Worker

Log Files

Storage

Logger

Free Buffers

Flushing Buffers

epoch=200

Epoch Thread

epoch=200
SILOR: ARCHITECTURE

Worker

Logger

Free Buffers

Flushing Buffers

Storage

Log Files

epoch=200
SILOR: ARCHITECTURE

Worker

Logger

Storage

Free Buffers

Flushing Buffers

Log Files

epoch=200

Epoch
Thread
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=200

epoch=200

epoch=200

epoch=200

pepoch=200

15-721 (Spring 2020)
Phase #1: Load Last Checkpoint
→ Install the contents of the last checkpoint that was saved into the database.
→ All indexes must be rebuilt from checkpoint.

Phase #2: Log Replay
→ Process logs in reverse order to reconcile the latest version of each tuple.
→ The txn ids generated at runtime are enough to determine the serial order on recovery.
SILOR: LOG REPLAY

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 can 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.
OBSERVATION

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

Checkpoints allows the systems to ignore large segments of the log to reduce recovery time.
IN-MEMORY CHECKPOINTS

The different approaches for how the DBMS can create a new checkpoint for an in-memory database are tightly coupled with its concurrency control scheme.

The checkpoint thread(s) scans each table and writes out data asynchronously to disk.
IDEAL CHECKPOINT PROPERTIES

Do not slow down regular txn processing.

Do not introduce unacceptable latency spikes.

Do not require excessive memory overhead.
CONSISTENT VS. FUZZY CHECKPOINTS

Approach #1: Consistent Checkpoints
→ Represents a consistent snapshot of the database at some point in time. No uncommitted changes.
→ No additional processing during recovery.

Approach #2: Fuzzy Checkpoints
→ The snapshot could contain records updated from transactions that committed after the checkpoint started.
→ Must do additional processing to figure out whether the checkpoint contains all updates from those txns.
CHECKPOINT MECHANISM

Approach #1: Do It Yourself
→ The DBMS is responsible for creating a snapshot of the database in memory.
→ Can leverage multi-versioned storage to find snapshot.

Approach #2: OS Fork Snapshots
→ Fork the process and have the child process write out the contents of the database to disk.
→ This copies everything in memory.
→ Requires extra work to remove uncommitted changes.
Create a snapshot of the database by forking the DBMS process.

→ Child process contains a consistent checkpoint if there are not active txns.
→ Otherwise, use the in-memory undo log to roll back txns in the child process.

Continue processing txns in the parent process.
Approach #1: Complete Checkpoint
→ Write out every tuple in every table regardless of whether they were modified since the last checkpoint.

Approach #2: Delta Checkpoint
→ Write out only the tuples that were modified since the last checkpoint.
→ Can merge checkpoints together in the background.
FREQUENCY

Approach #1: Time-based
→ Wait for a fixed period of time after the last checkpoint has completed before starting a new one.

Approach #2: Log File Size Threshold
→ Begin checkpoint after a certain amount of data has been written to the log file.

Approach #3: On Shutdown (Mandatory)
→ Perform a checkpoint when the DBA instructs the system to shut itself down. Every DBMS (hopefully) does this.
# CHECKPOINT IMPLEMENTATIONS

<table>
<thead>
<tr>
<th>Type</th>
<th>Contents</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>MemSQL</td>
<td>Consistent</td>
<td>Complete</td>
</tr>
<tr>
<td>VoltDB</td>
<td>Consistent</td>
<td>Complete</td>
</tr>
<tr>
<td>Altibase</td>
<td>Fuzzy</td>
<td>Complete</td>
</tr>
<tr>
<td>TimesTen</td>
<td>Consistent (Blocking)</td>
<td>Complete</td>
</tr>
<tr>
<td></td>
<td>Fuzzy (Non-Blocking)</td>
<td>Complete</td>
</tr>
<tr>
<td>Hekaton</td>
<td>Consistent</td>
<td>Delta</td>
</tr>
<tr>
<td>SAP HANA</td>
<td>Fuzzy</td>
<td>Complete</td>
</tr>
</tbody>
</table>
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.
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 without having to reload from disk.
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

SELECT COUNT(*) FROM events
WHERE type = 'crash'
AND time = 'Monday'

Query Plan
Fragments
SELECT COUNT(*) FROM events
WHERE type = 'crash'
AND time = 'Monday'

![Query Plan Fragments]
SELECT COUNT(*) FROM events
WHERE type = 'crash'
AND time = 'Monday'
**FACEBOOK SCUBA: ARCHITECTURE**

```sql
SELECT COUNT(*) FROM events
WHERE type = 'crash'
AND time = 'Monday'
```

```
Root
10+20=30
25+15=40
20
```

```
Aggregator
Leaf Node
Leaf Node
Leaf Node
Leaf Node
Leaf Node
Leaf Node
```

10
20
25
15
```
Aggregator
Leaf Node
Leaf Node
```

20
```
Aggregator
Leaf Node
Leaf Node
```

20

...
SELECT COUNT(*) FROM events
WHERE type = 'crash'
AND time = 'Monday'

Root

30 + 40 + 20 = 90

Aggregator

10 + 20 = 30

Aggregator

25 + 15 = 40

Leaf Node

10

Leaf Node

20

Leaf Node

25

Leaf Node

15

...
SHARED MEMORY RESTARTS

Approach #1: Shared Memory Heaps
→ All data is allocated in SM during normal operations.
→ Have to use a custom allocator to subdivide memory segments for thread safety and scalability.
→ Cannot use lazy allocation of backing pages with SM.

Approach #2: Copy on Shutdown
→ All data is allocated in local memory during normal operations.
→ On shutdown, copy data from heap to SM.
SHARED MEMORY RESTARTS

Approach #1: Shared Memory Heaps
→ All data is allocated in SM during normal operations.
→ Have to use a custom allocator to subdivide memory segments for thread safety and scalability.
→ Cannot use lazy allocation of backing pages with SM.

Approach #2: Copy on Shutdown
→ All data is allocated in local memory during normal operations.
→ On shutdown, copy data from heap to SM.
FACEBOOK SCUBA: FAST RESTARTS

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

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

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

Physical logging is a general-purpose approach that supports all concurrency control schemes.
→ Logical logging is faster but not universal.

Copy-on-update checkpoints are the way to go especially if you are using MVCC

Non-volatile memory is here!
NEXT CLASS

Networking Protocols