Georgia Tech

Lecture 10: Case Studies

CREATING THE NEXT®

Today's Agenda

Crash Recovery

- Recovery algorithms are techniques to ensure database **consistency**, transaction **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.
  - Reference
  - 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 Systems: 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.

Logging Schemes

Logging Schemes

- **Physical Logging**
  - Record the changes made to a specific location in the database.
  - **Example:** git diff
- **Logical Logging**
  - Record the high-level operations executed by txns.
  - Not necessarily restricted to single page.
  - **Example:** The UPDATE, DELETE, and INSERT queries invoked by a txn.

Physical vs. Logical Logging

- Logical logging requires less data written 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.

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 Optimization

- 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 Optimization

- 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.

Case Study: Microsoft Azure SQL

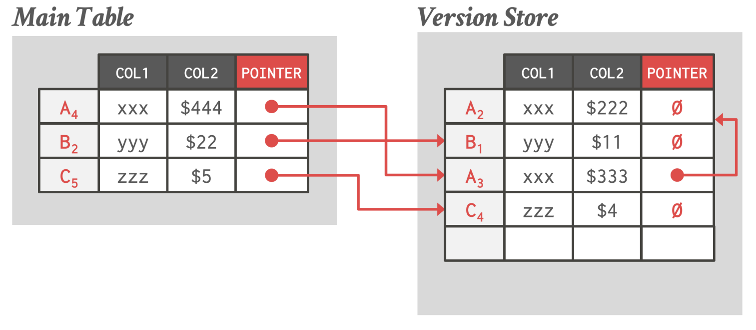
Observation

- The delta records in a DBMS that uses a n **multi-versioned concurrency control** (MVCC) protocol 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?

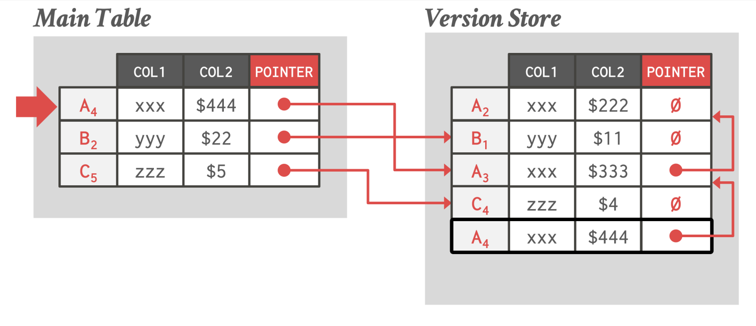
MSSQL: Constant-Time Recovery

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

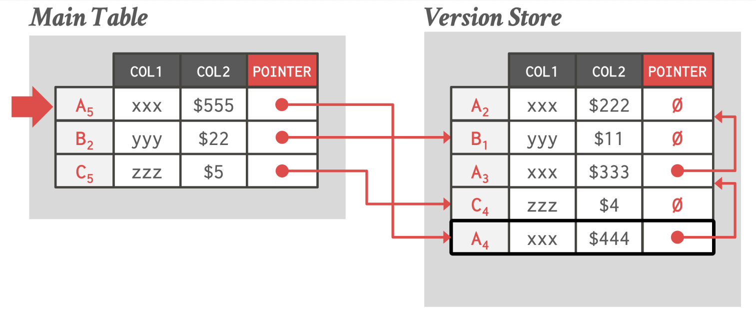
MSSQL: Version Store



MSSQL: Version Store



MSSQL: Version Store

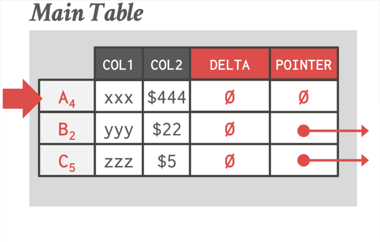


MSSQL CTR: Persistent Version Store

- **Approach 1: In-row Versioning**
  - Store small updates to a tuple as a delta record embedded with the latest version in the main table.
  - "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.

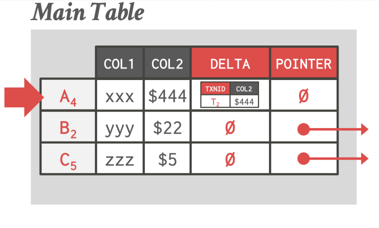
MSSQL CTR: In-row Versioning

- 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.



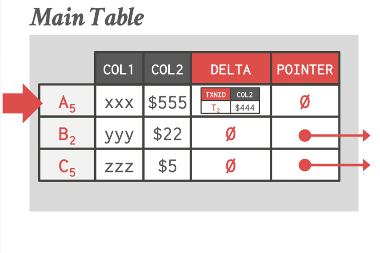
MSSQL CTR: In-row Versioning

- 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 sate 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.

Case Study: SiloR

Silo

- In-memory OLTP DBMS from Harvard/MIT.
  - Single-versioned OCC with epoch-based GC.
  - Same authors of the Masstree (Eddie Kohler et al.).
- **SiloR** uses physical logging + checkpoints to ensure durability of txns.
  - Reference
  - It achieves high performance by parallelizing all aspects of logging, checkpointing, and recovery.

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

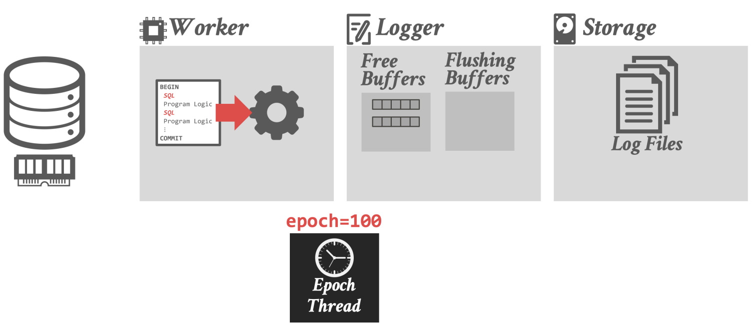
- Each logger thread maintains a pool of log buffers that are given to its worker threads.
- When a worker's buffer is full, it gives it back to the logger thread to flush to disk and attempts to acquire a new one.
  - If there are no available buffers, then it stalls.

SiloR: Log Files

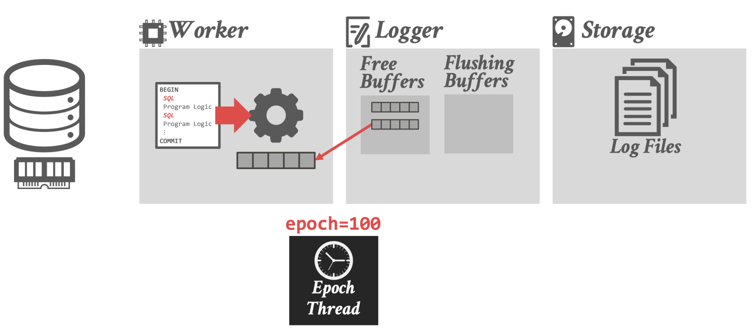
- The logger threads write buffers out to files:
  - After 100 epochs, it creates a new file.
  - The old file is renamed with a marker indicating the max epoch of records that it contains.
- Log record format:
  - Id of the txn that modified the record (TID).
  - A set of value log triplets (Table, Key, Value).
  - The value can be a list of attribute + value pairs.

```sql
UPDATE employees
   SET salary = 1000
   WHERE name IN ('Mozart', 'Beethoven')
```

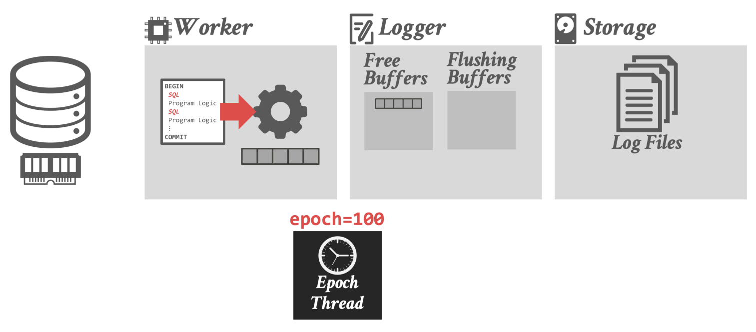
SiloR: Architecture



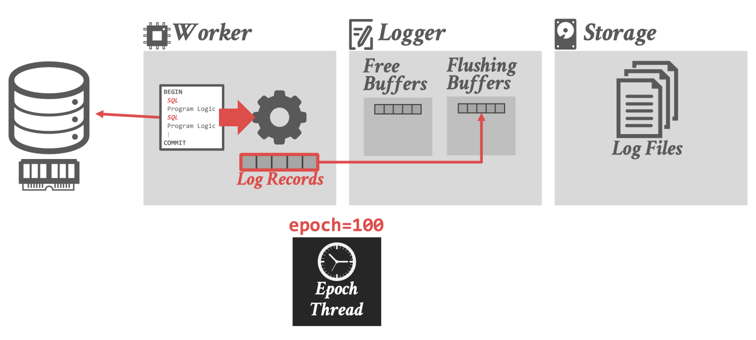
SiloR: Architecture



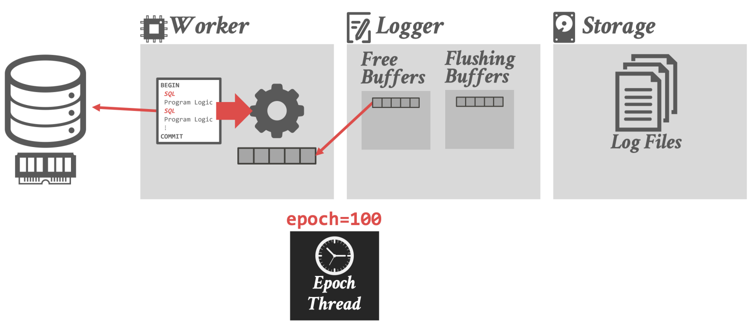
SiloR: Architecture



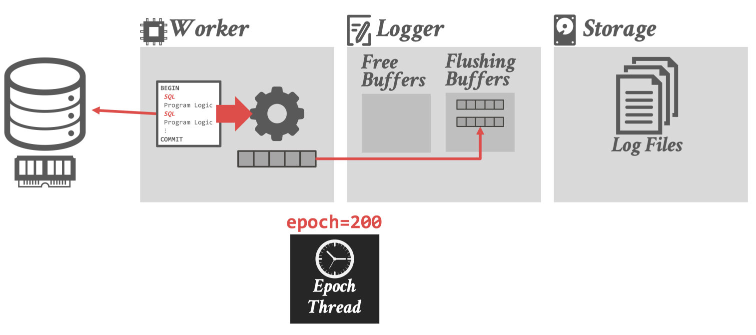
SiloR: Architecture



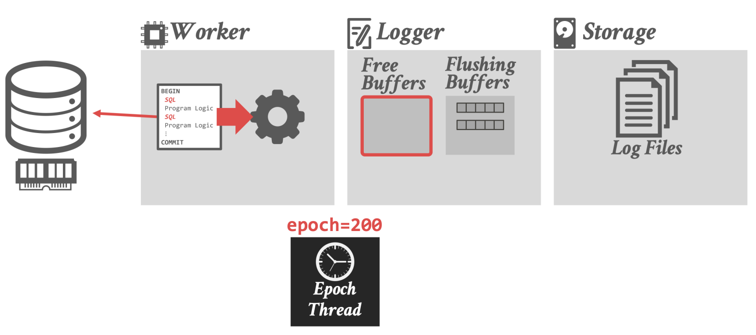
SiloR: Architecture



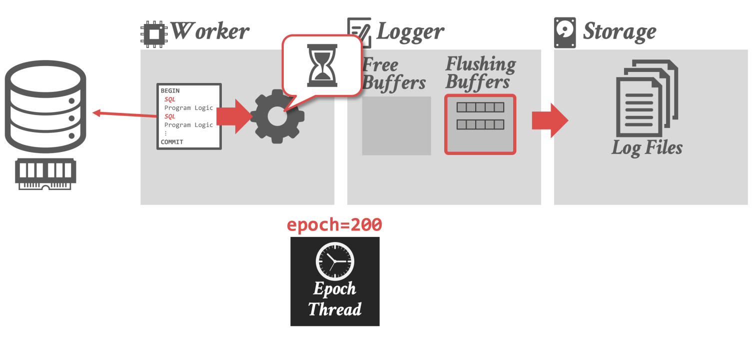
SiloR: Architecture



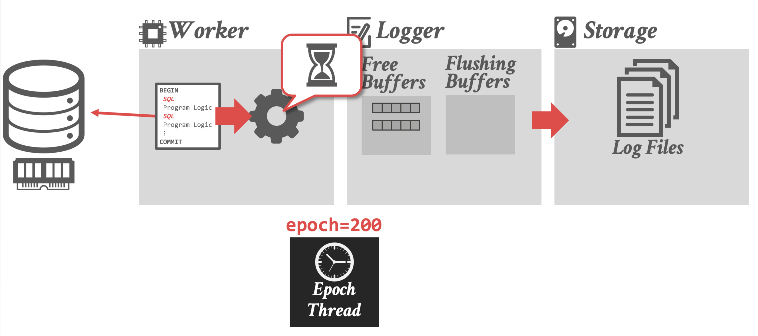
SiloR: Architecture



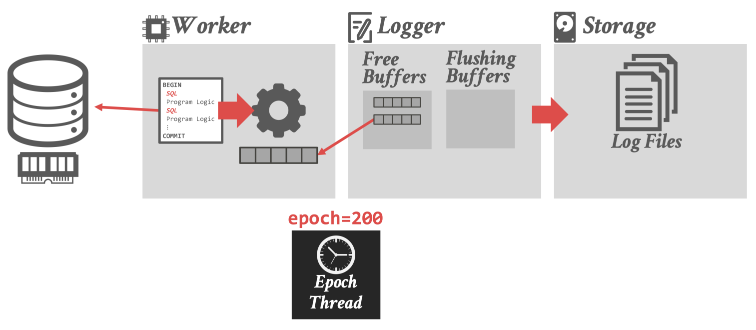
SiloR: Architecture



SiloR: Architecture



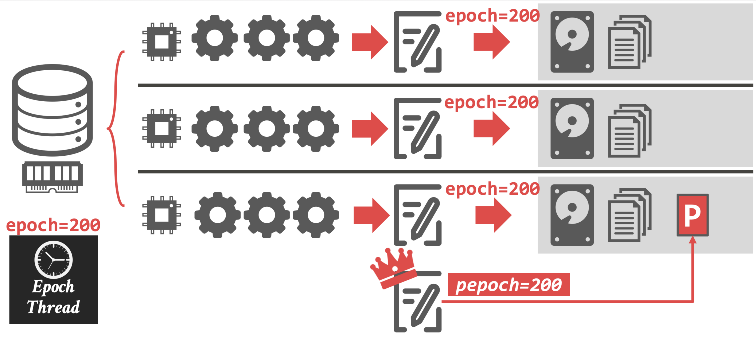
SiloR: Architecture



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 on non-volatile storage.

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 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 **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 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.

Checkpoint Protocols

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.
- Reference

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.

HYPER – OS Fork Snapshots

- 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.
- Reference

Checkpoint Contents

- **Approach 1: Complete Checkpoint**
  - Write out every tuple in every table regardless of whether 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.

Checkpoint 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

	Type	Contents	Frequency
MemSQL	Consistent	Complete	Log Size
VoltDB	Consistent	Complete	Time-Based
Altibase	Fuzzy	Complete	Time-based
TimesTen	Consistent (Blocking)	Complete	On Shutdown
“	Fuzzy (Non-Blocking)	Complete	Time-Based
Hekaton	Consistent	Delta	Log Size
SAP HANA	Fuzzy	Complete	Time-Based

Case Study: Facebook Scuba

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.

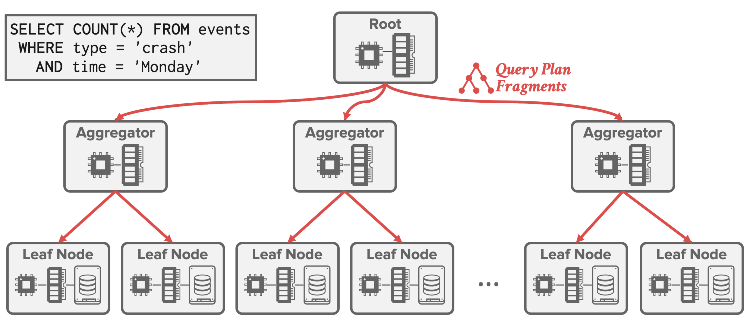
Facebook Scuba: Fast Restarts

- Decouple the in-memory database lifetime from the process lifetime.
- By storing the database in **shared memory**, the DBMS process can restart, and the memory contents will survive without having to reload from disk.

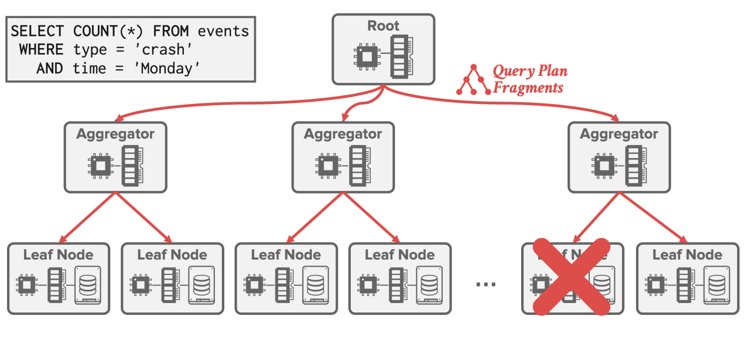
Facebook Scuba

- 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
- Reference

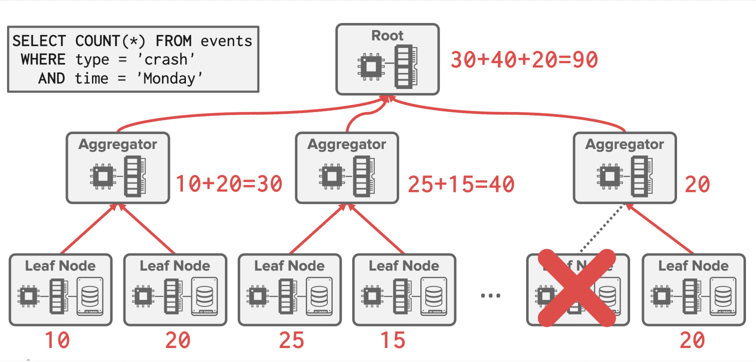
Facebook Scuba: Architecture



Facebook Scuba: Architecture



Facebook Scuba: Architecture



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.
  - Can 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 the there is a valid database in SM to copy into its heap.
  - Otherwise, the DBMS restarts from disk.

Conclusion

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

- Non-Volatile Memory Database Systems