Lecture #02: In-Memory Databases

15-721 Advanced Database Systems (Spring 2020)
https://15721.courses.cs.cmu.edu/spring2020/
Carnegie Mellon University
Prof. Andy Pavlo

1 Background

The history of DBMSs development is about dealing with the limitations of hardware. The first DBMSs in the 1970s were designed in environment with the following characteristics:

- Uniprocessor (single core CPU)
- RAM was severely limited
- Database had to be stored on disk
- Disk is slow. **Seriously slow**

But now DRAM capacities are large and inexpensive enough that most structured databases (gigabytes/low terabytes) will entirely fit in memory. This merits us to rethink all aspects of the DBMS to account for this. This course is about ways to do this.

2 Disk-Oriented Database Management Systems

For a disk oriented DBMS, the system architecture is predicated on the assumption that data is stored in non-volatile memory. This means that the DBMS may have to read data from disk during query execution.

In a disk-based system, only approximately 7% of instructions are for the execution of transaction logic in OLTP workloads [2]. The majority of the DBMS's instructions time are in managing three of its key components: (1) buffer pool, (2) concurrency control, (3) logging/recovery.

Buffer Pool

The DBMS organizes the database as a set of fixed-length blocks called *slotted pages*. The system uses an in-memory (volatile) *buffer pool* to cache the blocks cached from disk.

- When a query accesses a page, the DBMS checks to see if that page is already in memory.
- If not, the DBMS retrieves the memory from disk and copies it into a frame in its buffer pool. A pointer to the frame is returned for further operations.
- If there is no free frame to store the new page, DBMS finds a page to evict. If the evicted page is dirty, DBMS writes it on disk.
- Once the page is in memory, the DBMS translates any on-disk addresses to their in-memory addresses.
- Every tuple access has to go through the buffer pool manager regardless of whether that data will always be in memory. Even if we provide the DBMS with large enough memory to store the entire database in memory, these operations still occur.

Concurrency Control

In a disk oriented DBMS, the system assumes that a transaction could stall at any time when it tries to access data that is not in memory.

The system's concurrency control protocol allows the DBMS to execute other transactions at the same time to improve performance while still preserving atomicity and isolation guarantees. Since locks are stored separately in memory, however, a DBMS spends extra time to locate lock owners.

Logging and Recovery

Most DBMS use `STEAL + NO-FORCE` buffer pool policies so all modifications have to be flushed to the WAL before a transaction can commit [1]. Log entries contain before and after image of record modified. The DBMS flushes WAL pages to disk separately from corresponding modified database pages, so it takes extra work to keep track of what log record is responsible for what page (e.g., LSN).

3 In-Memory Database Management Systems

The system architecture assumes that the primary storage location of the database is in memory. This means that the DBMS does not need to perform extra steps during execution to handle the case where it has to retrieve data from disk. If disk I/O is no longer the slowest resource, much of the DBMS architecture will have to change to account for other bottlenecks: [4]

- Locking/latching
- Cache-line misses
- Pointer chasing
- Predicate evaluation
- Data movement and copying
- Networking (between application and DBMS)

Data Organization

We no longer need to use the slotted page layout in an in-memory DBMS as we do not have to worry about packing pages data onto disk. We also do not have to store the data close to each other as we are storing them on disk. Instead, an in-memory DBMS splits the data for tuples into fixed-length and variable-length pools. Indexes use direct pointers instead of record ids to the fixed-length data for each tuple. These tuples then have 64-bit pointers to any variable-length values stored in a separate memory location.

Concurrency Control

In-memory DBMSs still use either a pessimistic or optimistic concurrency control schemes to interleave transactions. They will use modern variants of these algorithms that are designed for in-memory data storage. The new bottleneck is contention caused from transactions trying to access data at the same time.

One key difference is that an in-memory DBMS can store locking information about each tuple together with its data. This is because the cost of a transaction acquiring a lock is the same as accessing data. Contrast this with disk-oriented DBMSs where locks are physically stored separate from their tuples because the tuples may get swapped out to disk.

Indexes

Like with concurrency control schemes, in-memory DBMSs will use data structures for their indexes that are optimized for fast, in-memory access. In-memory DBMSs will not log index updates. Instead, the system will rebuild the indexes upon restart when it loads the database back into memory. This avoids the runtime overhead of logging updates to indexes during transaction execution.

Query Processing

The best strategy for executing a query plan in a DBMS changes when all the data is already in memory. Sequential scans are no longer significantly faster than random access.

The traditional tuple-at-a-time iterator model is too slow because of function calls.

Logging and Recovery

The DBMS still needs WAL on non-volatile storage since the system could halt at anytime. In many cases, however, it may be possible to use more lightweight logging schemes (e.g., only store redo information). For example, since there are no "dirty pages", the DBMS does not need to maintain LSNs throughout the systems. In-memory DBMSs still takes checkpoints to reduce the amount of log that the system has to replay during recovery.

4 Concurrency Control

A DBMS's *concurrency control protocol* to allow transactions to access a database in a multi-programmed fashion while preserving the illusion that each of them is executing alone on a dedicated system. The goal is to have the effect of a group of transactions on the database's state is equivalent to any serial execution of all transactions. There are two high-level categories of concurrency control schemes:

1. **Two-Phase Locking (Pessimistic):** Assume transactions will conflict so they must acquire locks on database objects before they are allowed to access them.
2. **Timestamp Ordering (Optimistic):** Assume that conflicts are rare so transactions do not need to first acquire locks on database objects and instead check for conflicts at commit time.

5 Two-Phase Locking

There are two ways to deal with deadlocks in a two-phase locking (2PL) concurrency control protocol:

- **Deadlock Detection:** If deadlock is found, use a heuristic to decide what transaction to kill in order to break deadlock.
- **Deadlock Prevention:** If lock is not available, then make a decision about how to proceed.

6 Timestamp Ordering Concurrency Control

Use timestamps to determine the order of transactions.

Basic T/O Protocol

Every transaction is assigned a unique timestamp when they arrive in the system. The DBMS maintains separate timestamps in each tuple's header of the last transaction that read that tuple or wrote to it. Each transaction check for conflicts on each read/write by comparing their timestamp with the timestamp of the tuple they are accessing. The DBMS needs copy a tuple into the transaction's private workspace when reading a tuple to ensure repeatable reads.

Optimistic Concurrency Control (OCC)

Store all changes in private workspace. Check for conflicts at commit time and then merge. First proposed in 1981 at CMU by Kung and Robinson [3].

The protocol puts transactions through three phases during its execution:

1. **Read Phase:** Transaction's copy tuples accessed to private work space to ensure repeatable reads, and keep track of read/write sets.
2. **Validation Phase:** When the transaction invokes `COMMIT`, the DBMS checks if it conflicts with other transactions. Parallel validation means that each transaction must check the read/write set of other transactions that are trying to validate at the same time. Each transaction has to acquire locks for its write set records in some global order. Original OCC uses serial validation.
   The DBMS can proceed with the validation in two directions:
   - Backward Validation: Check whether the committing transaction intersects its read/write sets with those of any transactions that have already committed.
   - Forward Validation: Check whether the committing transaction intersects its read/write sets with any active transactions that have not yet committed.
3. **Write Phase:** The DBMS propagates the changes in the transactions write set to the database and makes them visible to other transactions' items. As each record is updated, the transaction releases the lock acquired during the Validation Phase

Timestamp Allocation

There are different ways for the DBMS to allocate timestamps for transactions [5]. Each have their own performance trade-offs.

- **Mutex:** This is the worst option. Mutexes are always a terrible idea.
- **Atomic Addition:** Use compare-and-swap to increment a single global counter. Requires cache invalidation on write.
- **Batched Atomic Addition:** Use compare-and-swap to increment a single global counter in batches. Needs a back-off mechanism to prevent fast burn.
- **Hardware Clock:** The CPU maintains an internal clock (not wall clock) that is synchronized across all cores. Intel only. Not sure if it will exist in future CPUs.
- **Hardware Counter:** Single global counter maintained in hardware. Not implemented in any existing CPUs.

7 Performance Bottlenecks

All concurrency control protocols have performance and scalability problems when there are a large number of concurrent threads and large amount of contention (i.e., the transactions are all trying to read/write to the same set of tuples) [5].

Lock Thrashing:

- Each transaction waits longer to acquire locks, causing other transaction to wait longer to acquire locks.
- Can measure this phenomenon by removing deadlock detection/prevention overhead.

Memory Allocation

- Copying data on every read/write access slows down the DBMS because of contention on the memory controller.
- Default libc malloc is slow. Never use it.

References

[1] M. J. Franklin. Concurrency control and recovery. In *Computing Handbook, Third Edition: Information Systems and Information Technology*, pages 12: 1–21. 2014. URL http://db.lcs.mit.edu/6.893/F04/ccandr.pdf.

[2] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. OLTP through the looking glass, and what we found there. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 981–992, 2008. doi: http://doi.acm.org/10.1145/1376616.1376713.

[3] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Datab. Syst*, 6(2), June 1981. URL https://dl.acm.org/citation.cfm?id=319567.

[4] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era: (it's time for a complete rewrite). In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 1150–1160, 2007. URL http://hstore.cs.brown.edu/papers/hstore-endofera.pdf.

[5] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker. Staring into the abyss: an evaluation of concurrency control with one thousand cores. In *VLDB '14: Proceedings of the VLDB Endowment*, volume 8, pages 209–220, November 2014. URL https://dl.acm.org/citation.cfm?id=2735511.