Lecture #05: Database Storage (Part II)

15-445/645 Database Systems (Spring 2026)
https://15445.courses.cs.cmu.edu/spring2026/
Carnegie Mellon University
Andy Pavlo

1 Tuple-Oriented Storage

The most common way to store tuples on disk is the Tuple-Oriented Storage architecture, using the slotted-page scheme described in previous lectures. Tuples are retrieved using its record ID:

- Check the page directory to find the page position on disk.
- Fetch the page from disk into memory (into the buffer pool).
- Use the slot array to find the tuple's offset within the page.

Inserting a new tuple is simple:

- Check the page directory to find a page with a free slot.
- Fetch the page from disk into memory.
- Use the slot array to check if there is enough free space in the page.
- If not, find another page with a free slot or create a new page.
- Insert the tuple into the page and update the slot array.

However, updating tuples can become expensive:

- Navigate to the tuple using the record ID with the same steps as retrieval.
- If the new value fits in the same space, update in place.
- Otherwise, mark the old value as deleted and insert the new value as if it were a new tuple.

Therefore, while tuple-oriented storage is great for reads, there are several problems associated with it:

- **Fragmentation**: Deletion of tuples can leave gaps in the pages, making them not fully utilized.
- **Useless Disk I/O**: Due to the block-oriented nature of non-volatile storage, the whole block needs to be fetched to update a tuple.
- **Random Disk I/O**: The disk reader could have to jump to 20 different places to update 20 different tuples, which can be very slow.

What if we were working on a system which only allows creation of new pages and no in-place updates (e.g. HDFS, Google Colossus, certain object stores)? The log-structured storage model works with this assumption and addresses some of the problems listed above.

2 Log-Structured Storage

Instead of storing tuples in pages and updating them in-place, Log-Structured Storage maintains a log that records changes to tuples. This idea is based on log-structured file systems (LSFS) [1] and log-structured merge trees (LSM Tree)[2].

The DBMS applies changes to an in-memory data structure (**MemTable**) and writes out the changes sequentially to disk (**SSTable**). The records stored in these structures contain the tuple's unique identifier,

[1] https://doi.org/10.1145/146941.146943
[2] https://doi.org/10.1007/s002360050048

the type of operation (PUT/DELETE), and for a PUT operation, the contents of the tuple. Effectively, you care about the latest values for each key (most recent PUT/DELETE).

Logs are first stored in **MemTable** through fast, in-memory operations. Once **MemTable** fills up, the DBMS serializes the logs it stores and writes them to disk as an **SSTable**. The DBMS also sorts each **SSTable** by key before writing it to disk. Since the **SSTable**s are immutable and written to disk sequentially, this results in less random disk I/O. This workload also maps nicely to **append-only** storage like many cloud storage options, etc.

To read a record, the DBMS first checks **MemTable** to see whether it exists there. If the key does not exist in **MemTable**, then the DBMS has to check the **SSTable**s at each level. A brute force solution is to scan down the **SSTable**s from newest to oldest and perform binary search within each **SSTable** to find the most recent contents of the tuple, which can be slow. To avoid this, the DBMS can maintain an in-memory **SummaryTable** to track additional metadata like min/max key per **SSTable** and a key filter (e.g., Bloom filter) per level.

Compaction

In a write-heavy workload, the DBMS will accumulate a large number of **SSTable**s on disk. Thus, the DBMS can periodically use a sort-merge algorithm to combine **SSTable**s by taking only the most recent change for each tuple. This reduces wasted space and speeds up reads.

There are many ways to compact log files. In **Universal Compaction**, **SSTable**s reside in a single "universal" level. DBMS will trigger compaction when size thresholds are met or too many **SSTable**s overlap in key ranges. This approach works better for insert-heavy workloads and time-oriented queries. In **Level Compaction**, the smallest files are level 0. Level 0 files can be compacted to create a bigger level 1 file, level 1 files can be compacted to a level 2 file, etc. **SSTable**s in the same level are managed with sorted and non-overlapping key ranges (except for level 0, which may have overlapping key ranges). **Level Compaction** works well with read-heavy workloads.

Tradeoffs

The tradeoffs of using Log-Structured Storage are summarized below:

- Fast sequential writes, good for append only storage.
- Reads may be slow.
- Compaction is expensive.
- Write amplification (for each logical write, there could be multiple physical writes during the compaction process).

3 Index-Organized Storage

Both slotted-page storage and log-structured storage rely on an additional index to find individual tuples because the tables are inherently unsorted. In the **index-organized storage** scheme, the DBMS directly stores a table's tuples as the value of an index data structure (e.g. B+ tree, skip list, trie). The DBMS uses a page layout similar to a slotted page, and tuples are typically sorted in the page based on key.

4 System Catalogs

In order for the DBMS to decipher the contents of tuples, it maintains an internal catalog containing metadata about its databases.

Metadata Contents:

- The tables, views, columns and procedures the database has as well as any indexes on those tables.
- Users of the database and what permissions they have.
- Internal statistics about tables (i.e., max value of an attribute).

Most DBMSs store their catalog inside of themselves as tables. They use special code to "bootstrap" these catalog tables.