# Lecture #05: Database Storage (Part II)

**15-445/645 Database Systems (Spring 2025)**
https://15445.courses.cs.cmu.edu/spring2025/
Carnegie Mellon University
Jignesh Patel

## 1 Log-Structured Storage

There are several problems associated with the Slotted-Page tuple-oriented architecture discussed previously in lecture:

- **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.

### Log-Structured Storage Overview

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 then writes out the changes sequentially to disk (**SSTable**). The records stored in these structures contain the tuple's unique identifier, 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**, which is a fast operation due to being in-memory. Once **MemTable** fills up, the DBMS will serialize the logs it stores and write an **SSTable** to disk. 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.

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

### 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**, any log files can be compacted together. 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. **Tiering** is another log compaction method that will not be covered in this course.

### 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)

## 2 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. The DBMS uses a page layout similar to a slotted page, and tuples are typically sorted in the page based on key.

## 3 Data Representation

The data in a tuple is essentially just a byte array prefixed with a header that contains metadata. It doesn't keep track of what kinds of values the attributes are. It is up to the DBMS to keep track of that and interpret the bytes. A *data representation* scheme is how a DBMS stores the bytes for a value.

DBMSs want to make sure the tuple attributes are word-aligned so that the CPU can access them without any unexpected behavior or additional work. Two approaches are usually taken:

- **Padding:** Add empty bits after attributes to ensure that they are word aligned.
- **Reordering:** Switch the order of attributes in the tuple's physical layout so they are word aligned.

There are five high level datatypes that can be stored in tuples: integers, variable-precision numbers, fixed-point precision numbers, variable length values, and dates/times.

### Integers

Most DBMSs store integers in the same way as C/C++. These values are fixed length.

Examples: `INTEGER`, `BIGINT`, `SMALLINT`, `TINYINT`.

### Variable Precision Numbers

These are inexact, variable-precision numeric types that use the "native" C/C++ types specified by the IEEE-754 standard. These values are also fixed length.

Operations on variable-precision numbers are faster to compute than arbitrary precision numbers because

the CPU can execute instructions on them directly. However, there may be rounding errors when performing computations due to the fact that some numbers cannot be represented precisely.

Examples: `FLOAT`, `REAL`.

### Fixed-Point Precision Numbers

These are numeric data types with arbitrary precision and scale. They are typically stored in an exact, variable-length binary representation (almost like a string) with additional metadata describing the length of the data and where the decimal should be.

These data types are used when rounding errors are unacceptable, but the DBMS pays a performance penalty to get this accuracy.

Examples: `NUMERIC`, `DECIMAL`.

### Variable-Length Data

These represent data types of arbitrary length. They are typically stored with a header that keeps track of the length of the string to make it easy to jump to the next value. It may also contain a checksum for the data.

Most DBMSs do not allow a tuple to exceed the size of a single page. The ones that do store the data on a special "overflow" page and have the tuple contain a reference to that page. These overflow pages can contain pointers to additional overflow pages until all the data can be stored.

Some systems will let you store these large values in an external file, and then the tuple will contain a pointer to that file. For example, if the database is storing photo information, the DBMS can store the photos in the external files rather than having them take up large amounts of space in the DBMS. One downside of this is that the DBMS cannot manipulate the contents of this file. Thus, there are no durability or transaction protections.

Examples: `VARCHAR`, `VARBINARY`, `TEXT`, `BLOB`.

### Dates and Times

Representations for date/time vary for different systems. Typically, these are represented as some unit time (micro/milli)seconds since the unix epoch.

Examples: `TIME`, `DATE`, `TIMESTAMP`.

### Null Data Types

There are three common apporaches to represent nulls in a DBMS.

- **Null Column Bitmap Header:** Store a bitmap in a centralized header that specifies what attributes are null. This is the most common approach.
- **Special Values:** Designate a value to represent `NULL` for a data type (e.g., `INT32_MIN`).
- **Per Attribute Null Flag:** Store a flag that marks that a value is null. This apporach is **NOT** recommended because it is not memory-efficient. For each value, the DBMS has to use more than just a single bit to avoid messing up the word alignment.

## 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 and columns 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.