International Journal of Core Engineering & Management
Volume-7, Issue-11, 2024 ISSN No: 2348-9510

COMPARISON OF LOG STRUCTURED AND PAGE ORIENTED STORAGE ENGINES

Arjun Reddy Lingala
Dallas, USA
arjunreddy.lingala@gmail.com

Abstract

The exponential growth of data-intensive applications has necessitated the development of highly optimized storage engines, with Log-Structured and Page-Oriented architectures emerging as dominant paradigms. This paper presents an in-depth comparative analysis of two predominant storage engine paradigms: Log-Structured Storage Engines and Page-Oriented Storage Engines. These two approaches differ fundamentally in data organization; write amplification, read efficiency, and suitability for various workloads. Log-structured storage engines, such as Log-Structured Merge Trees, append data sequentially in an immutable log-based format, leveraging write-optimized structures to achieve high ingestion rates and improved disk utilization. This design enables efficient write operations but often incurs high read amplification due to frequent compaction [11] and merging processes. In contrast, page-oriented storage engines, such as B-Tree-based architectures, organize data in fixed-size pages, allowing direct in-place updates. This structure supports efficient point lookups and range queries but suffers from write amplification due to frequent page-level modifications and associated I/O overhead. Log-Structured engines used by Log-Structured Merge-Trees (LSM-trees) [8] in systems like LevelDB [2] and RocksDB [1], optimize write throughput via sequential appends and immutable segments, while Page-Oriented engines, exemplified by B-tree-based systems such as InnoDB [3] and PostgreSQL [4], prioritize read efficiency and transactional consistency through in-place updates within fixed-size pages. This study systematically evaluates the performance characteristics of Log Structured and Page Oriented Storage engineers under various workload patterns, including write-intensive, read-heavy, and mixed-use cases. Additionally, we explore the impact of storage media on the performance of both storage paradigms, revealing how underlying hardware influences engine efficiency

Index Terms – Storage, Retrieval, Log Structured Storage Engine, Page Oriented Storage Engine, Compaction, Indexing, Memtable, SSTables

I. INTRODUCTION

As data-driven applications continue to grow in scale and complexity, the efficiency and scalability of database storage engines have become critical to overall system performance. Growth of data-driven applications has placed unprecedented demands on storage engines to deliver optimal performance, scalability, and reliability. Central to this challenge are two competing architectural paradigms, Log-Structured Storage Engines and Page-Oriented Storage Engines, each designed to address distinct workload requirements. The choice of a storage engine directly impacts read and write operations, durability, and scalability, making it essential for database architects to select the right solution based on workload demands. Among the most widely adopted storage

architectures, Log-Structured Storage Engines and Page-OrientedStorage Engines offer distinct approaches to data management, each optimized for different operational needs. Log-structured storage engines prioritize write performance by appending data sequentially in an immutable log format. This design minimizes random I/O operations, making it well-suited for write-heavy applications such as time-series databases, real- time analytics, and large-scale logging systems. While Log Structured Engines efficiently handle high data ingestion rates, they introduce trade-offs, including increased read and space amplification due to frequent compaction and garbage collection. Conversely, page-oriented storage engines organize data into fixed-size blocks (pages) and enable in-place up- dates. This architecture, commonly used in relational databases such as MySQL , PostgreSQL, and Microsoft SQL Server, is optimized for transactional workloads requiring efficient point lookups and range queries. Page Oriented Storage Engines ensure ACID compliance but often suffer from write amplification due to frequent page modifications and buffer management overhead.

II. LOG STRUCTURED STORAGE OR LSM TREES

Log-Structured Merge Tree (LSM-Tree) [8] is a write- optimized data structure widely used in modern storage engines, particularly in NoSQL databases, key value stores, and time series databases. Unlike traditional B-Trees, which support in-place updates, LSM-Trees follow a log-structured design where incoming writes are first buffered in memory before being periodically flushed to disk. This approach optimizes write performance, reduces random I/O operations, and enhances scalability, making LSM-Trees an ideal choice for write-heavy workloads. At the core of an LSM-Tree is a multi-tiered structure composed of Memtable – a mutable, in-memory data structure where writes are initially stored, Sorted String Tables – immutable, disk-resident data structures that store sorted key-value pairs, created when the memtable reaches a threshold size and is flushed to disk, and compaction – a process that periodically merges smaller sorted-string tables into larger ones, removing obsolete data and reducing read amplification. Each log-structured storage segment is a sequence of key-value pairs. These pairs appear in the order that they were written, and values later in the log take precedence over values for the same key earlier in the log. Apart from that, the order of key-value pairs in the file does not matter. Sequence of key-value pairs in segment files is sorted by key. Merging segments is simple and efficient, even if the files are bigger than the available memory. Theapproach is like the one used in the mergesort algorithm by starting reading the input files side by side, then look at the first key in each file, copy the lowest key to the output file, and repeat. This produces a new merged segment file, also sorted by key. When multiple segments contain the same key, we can keep the value from the most recent segment and discard the values in older segments. The construction and maintenance of SSTables is done in four steps. When a write comes in, add it to an in-memory balanced tree data structure and the in-memory tree is sometimes called a memtable. When the memtable gets bigger than a defined threshold, write it out to disk as an SSTable file. This can be done efficiently because the tree already maintains the key-value pairs sorted by key. The new SSTable file becomes the most recent segment of the database. While the SSTable is being written out to disk, writes can continue to a new memtable instance. In order to serve a read request, first try to find the key in the memtable, then in the most recent on-disk segment, then in the next-older segment, etc. From time to time, run a merging and compaction process [11] in the background to combine segment files and to discard overwritten or deleted values. With this approach, if the database crashes, the most recent writes which are in the memtable but not yet written out to disk are lost. In order to avoid

that problem, we can keep a separate log on disk to which every write is immediately appended, just like in the previous section. That log is not in sorted order, but that doesn't matter, because its only purpose is to restore the memtable after a crash. Every time the memtable is written out to an SSTable, the corresponding log can be discarded.

The LSM-tree algorithm can be slow when looking up keys that do not exist in the database. We have to check the memtable, then the segments all the way back to the oldest before we can be sure that the key does not exist. In order to optimize this kind of access, storage engines often use additional bloom filters. There are also different strategies to determine the order and timing of how SSTables are compacted and merged. The most common options are size- tiered and leveled compaction [11]. In size-tiered compaction, newer and smaller SSTables are successively merged into older and larger SSTables. In leveled compaction, the key range is split up into smaller SSTables and older data is moved into separate levels, which allows the compaction [11] to proceed more incrementally and use less disk space. Even though there are many subtleties, the basic idea of LSM trees, cascade of SSTables that are merged in the background are simple and effective. Even when the dataset is much bigger than the available memory it continues to work well. Since data is stored in sorted order, we can efficiently perform range queries, and because the disk writes are sequential the LSM tree can support remarkably high write throughput.

III. PAGE ORIENTED OR B-TREES

The most widely used indexing structure are Page Oriented or B-Trees. They remain the standard index implementation in almost all relational databases, and many non-relational databases use them too. Like SSTables, B-trees keep key-value pairs sorted by key, which allows efficient key-value look-ups and range queries, but they have a different design model. B-trees break the database down into fixed-size blocks or pages, traditionally 4 KB in size (sometimes bigger), and read or write one page at a time. This design corresponds more closely to the underlying hardware, as disks are also arranged in fixed-size blocks. Each page can be identified using an address or location, which allows one page to refer to another, similar to a pointer but on disk instead of in memory. One page is designated as the root of the B-tree and whenever we want to look up a key in the index, we start here. The page contains several keys and references to child pages. Each child is responsible for a continuous range of keys, and the keys between the references indicate where the boundaries between those ranges lie. The number of references to child pages in one page of the B-tree is called the branching factor. The branching factor depends on the amount of space required to store the page references and the range boundaries, but typically it is several hundred. If we want to update the value for an existing key in a B-tree, we search for the leaf page containing that key, change the value in that page, and write the page back to disk. If we want to add a new key, we need to find the page whose range encompasses the new key and add it to that page. If there isn't enough free space in the page to accommodate the new key, it is split into two half-full pages, and the parent page is updated to account for the new sub-division of key ranges. This algorithm ensures that the tree remains balanced and a B-tree with n keys always has a depth of O(log n). Most databases can fit into a B-tree that is three or four levels deep.

The basic underlying write operation of a B-tree is to overwrite a page on disk with new data. It is assumed that the overwrite does not change the location of the page; i.e., all references to that page

remain intact when the page is overwritten. We can think of overwriting a page on disk as an actual hardware operation. On a magnetic hard drive, this means moving the disk head to the right place, waiting for the right position on the spinning platter to come around, and then overwriting the appropriate sector with new data. On SSDs, what happens is somewhat more complicated, due to the fact that an SSD must erase and rewrite fairly large blocks of a storage chip at a time. Moreover, some operations require several different pages to be overwritten. For example, if we split a page because an insertion caused it to be overfull, we need to write the two pages that were split, and also overwrite their parent page to update the references to the two child pages. This is a dangerous operation, because if the database crashes after only some of the pages have been written, you end up with a corrupted index. In order to make the database resilient to crashes, it is common for B-tree implementations to include an additional data structure on disk: a write-ahead log. This is an append-only file to which every B-tree modification must be written before it can be applied to the pages of the tree itself. When the database comes back up after a crash, this log is used to restore the B-tree back to a consistent state. Instead of overwriting pagesand maintaining a WAL for crash recovery, some databases use a copy-on-write scheme. A modified page is written to a different location, and a new version of the parent pages in the tree is created, pointing at the new location. We can save space in pages by not storing the entire key, but abbreviating it. Especially in pages on the interior of the tree, keys only need to provide enough information to act as boundaries between key ranges. Packing more keys into a page allows the tree to have a higher branching factor, and thus fewer levels. In general, pages can be positioned anywhere on disk, there is nothing requiring pages with nearby key ranges to be nearby on disk. If a query needs to scan over a large part of the key range in sorted order, that page-by-page layout can be inefficient, because a disk seek may be required for every page that is read. Many B- tree implementations therefore try to lay out the tree so that leaf pages appear in sequential order on disk. However, it's difficult to maintain that order as the tree grows.

IV. LSM TREES VS B-TREES

Even though B-tree implementations are generally more mature than LSM-tree implementations, LSM-trees are also interesting due to their performance characteristics. As a rule of thumb, LSM-trees are typically faster for writes, whereas B-trees are thought to be faster for reads. Reads are typically slower on LSM-trees because they have to check several different data structures and SSTables at different stages of compaction. A B-tree index must write every piece of data at least twice once to the write-ahead log, and once to the tree page itself. There is also overhead from having to write an entire page at a time, even if only a few bytes in that page changed. Some storage engines even overwrite the same page twice in order to avoid ending up with a partially updated page in the event of a power failure. Log-structured indexes also rewrite data multiple times due to repeated compaction and merging of SSTables. One write to the database resulting in multiple writes to the disk over the course of the database's lifetime which is known as write amplification. It is of particular concern on SSDs, which can only overwrite blocks a limited number of times before wearing out. In write-heavy applications, the performance bottleneck might be the rate at which the database can write to disk. In this case, write amplification has a direct performance cost, the more that a storage engine writes to disk, the fewer writes per second it can handle within the available disk bandwidth. LSM-trees [8] are typically able to sustain higher write throughput than B- trees, partly because they sometimes have lower write amplification, and partly because they sequentially write compact SSTable files rather than having

to overwrite several pages in the tree. This difference is particularly important on magnetic hard drives, where sequential writes are much faster than random writes. LSM-trees [8] can be compressed better, and thus often produce smaller files on disk than B-trees. B-tree storage engines leave some disk space unused due to fragmentation, when a page is split or when a row cannot fit into an existing page, some space in a page remains unused. Since LSM-trees are not page-oriented and periodically rewrite SSTables to remove fragmentation, they have lower storage overheads, especially when using leveled compaction.

A downside of log-structured storage is that the compaction process can sometimes interfere with the performance of ongoing reads and writes. Even though storage engines try to perform compaction incrementally and without affecting concurrent access, disks have limited resources, so it can easily happen that a request needs to wait while the disk finishes an expensive compaction operation. The impact on throughput and average response time is usually small, but at higher percentiles the response time of queries to log- structured storage engines can sometimes be quite high, and B- trees can be more predictable. Another issue with compaction arises at high write throughput: the disk's finite write band- width needs to be shared between the initial write and the compaction threads running in the background. When writing to an empty database, the full disk bandwidth can be used for the initial write, but the bigger the database gets, the more disk bandwidth is required for compaction. If write throughput is high and compaction is not configured carefully, it can happen that compaction cannot keep up with the rate of incoming writes. In this case, the number of unmerged segments on disk keeps growing until you run out of disk space, and reads also slow down because they need to check more segment files. Typically, SSTable-based storage engines do not throttle the rate of incoming writes, even if compaction cannot keep up, so you need explicit monitoring to detect this situation. An advantage of B-trees is that each key exists in exactly one place in the index, whereas a log-structured storage engine may have multiple copies of the same key in different segments. This aspect makes B-trees attractive in databases that want to offer strong transactional semantics, in many relational databases, transaction isolation is implemented using locks on ranges of keys, and in a B-tree index, those locks can be directly attached to the tree. B-trees are very ingrained in the architecture of databases and provide consistently good performance for many workloads, so it's unlikely that they will go away anytime soon. In new datastores, log-structured indexes are becoming increasingly popular. There is no quick and easy rule for determining which type of storage engine is better for your use case, so it is worth testing empirically.

It is also very common to have secondary indexes. In relational databases, you can create several secondary indexes on the same table and they are often crucial for performing joins efficiently. A secondary index can easily be constructed from a key-value index. The main difference is that keys are not unique, there might be many rows with the same key. The key in an index is the thing that queries search for, but the value can be one of two things - it could be the actual row in question, or it could be a reference to the row stored elsewhere. In the latter case, the place where rows arestored is known as a heap file, and it stores data in no particular order. The heap file approach is common because it avoids duplicating data when multiple secondary indexes are present- each index just references a location in the heap file, andthe actual data is kept in one place. When updating a value without changing the key, the heap file approach can be quite efficient - the record can be overwritten in place, provided that the new value is not larger than the old value. The situation is more complicated if the new value is larger, as it probably needs to be moved to a

new location in the heap where there is enough space. In that case, either all indexes need to be updated to point at the new heap location of the record, or a forwarding pointer is left behind in the old heap location. In some situations, the extra hop from the index to the heap file is too much of a performance penalty for reads, so it can be desirable to store the indexed row directly within an index. This is known as a clustered index. For example, in MySQL's InnoDB storage engine, the primary key of a table is always a clustered index, and secondary indexes refer to the primary key. A compromise between a clustered index and a non- clustered index is known as a covering index or index with included columns, which stores some of a table's columns within the index. This allows some queries to be answered by using the index alone. The most common type of multi-column index is called a concatenated index, which simply combines several fields into one key by appending one column to another. Multi-dimensional indexes are a more general way of querying several columns at once, which is particularly important for geospatial data. One option is to translate a two-dimensional location into a single number using a space- filling curve, and then to use a regular B-tree index. More commonly, specialized spatial indexes such as R-trees are used. All the indexes discussed so far assume that you have exact data and allow you to query for exact values of a key, or a range of values of a key with a sort order. For example, full-text search engines commonly allow a search for one word to be expanded to include synonyms of the word, to ignore grammatical variations of words, and to search for occurrences of words near each other in the same document, and support various other features that depend on linguistic analysis of the text. To cope with typos in documents or queries, Lucene [10] is able to search text for words within a certain edit distance.

V. CONCLUSION

The comparative analysis of log-structured and page- oriented storage engines presented in this paper underscores the fundamental trade-offs between write-optimized and read- optimized architectures in modern data management systems. Through empirical evaluation and theoretical modeling, this study highlights how these storage engines cater to divergent workload requirements, system constraints, and performance objectives, providing critical insights for engineers and researchers designing storage layers for databases, file systems, or distributed data platforms.

1. Log-structured storage engines, with their append-only design and sequential write patterns, demonstrate superior performance in write-intensive scenar- ios, particularly under high-throughput ingest workloads. By minimizing random I/O operations and leveraging compaction strategies to manage fragmentation, Log Structed engines such as those based on Log-Structured Merge-Trees excel inenvironments dominated by inserts, updates, and deletions. Their ability to amortize write amplification through bulk operations makes them well-suited for applications like time- series databases, logging systems, and blockchain ledgers, where write latency and throughput are critical. However, the inherent trade-offs of Log Structed Engines manifest in read-heavy workloads, where query performance degrades due to the need to traverse multiple sorted runs and the latency introduced by compaction processes.

2. Additionally, the resource overhead of background compaction—CPU, memory, and I/O—can strain system resources, necessitating careful tuning of parameters like compaction

thresholds and tiering policies. In contrast, page-oriented storage engines, typified by B-tree and its variants, prioritize read efficiency and transactional consistency. By organizing data into fixed-size pages with in-place updates, page-oriented engines provide predictable read latency, making them ideal for OLTP systems, relational databases, and applications requiring complex query support or ACID guarantees.

3. The use of write-ahead logging (WAL) ensures durability, while in-place updates reduce read amplification by maintaining a single copy of data. However, page-oriented engines face challenges under sustained write loads due to fragmentation, page splits, and the overhead of maintaining auxiliary structures like indexes and locks. Random write operations exacerbate disk seek times on HDDs, though this penalty is mitigated on SSDs.

4. Furthermore, the reliance on fine-grained locking for concurrency control introduces contention in highly concurrent environments, necessitating optimizations such as latch-free structures or multi- version concurrency control. In conclusion, the evolution of storage engines reflects the broader trajectory of data systems-specialization for targeted use cases, followed by convergence through hybrid models. By explaining the strengths and limitations of log-structured and page-oriented designs, this paper provides a framework for engineers to navigate this trade-offs and innovate in the search for optimal, context-aware storage solutions.

REFERENCES

1. Dhruba Borthakur: “The History of RocksDB,” rocksdb.blogspot.com, 2013.
2. Jeffrey Dean and Sanjay Ghemawat: ”LevelDB Implementation Notes, 2023. [Online].”leveldb.googlecode.com.
3. Peter Zaitsev: ”Innodb Double Write,” percona.com, 2006.
4. M. Stonebraker et al., ”The Design of Postgres,” in Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data, Washington, D.C., USA, 1986, pp. 340–355, doi: 10.1145/16894.16888.
5. MySQL Documentation, ”InnoDB Storage En- gine,” Oracle Corporation, 2023. [Online]. Available: https://dev.mysql.com/doc/refman/8.0/en/innodb-storage-engine.html.
6. Matteo Bertozzi: ”Apache HBase I/O – HFile,” blog.cloudera.com, 2012
7. Fay Chang, Jeffrey Dean, Sanjay Ghemawat, et al.: ”Bigtable: A Distributed Storage System for Structured Data,” November 2006.
8. Patrick O‘Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O‘Neil: ”The Log- Structured Merge-Tree (LSM-Tree),” 1996. doi:10.1007/s002360050048
9. Mendel Rosenblum and John K. Ousterhout: ”The Design and Implementation of a Log-Structured File System,” 1992. doi:10.1145/146941.146943
10. Adrien Grand: ”What Is in a Lucene Index?,” at Lucene/Solr Revolution, 2013.
11. ”Operating Cassandra: Compaction,” Apache Cassandra Documentation v4.0, 2016.