LSM TREES VS B TREE

BTree-MongoDb,Postgres,sql,sqlite,

LsmTree- **Apache Cassandra,** Apache HBase, RocksDB, ScyllaDB

If you compare the performance of just the two without any caching strategy layered on top of it, B+Tree will give better performance. LSM is delayed ordering, B+Tree is always ordered.

LSM CPU usage is higher due to compaction.

LSM space efficiency is superior to btrees, because of the delayed total ordering when it does compaction.

Space Efficiency Gains from Delayed Ordering:

- **Reduced Fragmentation:** Because writes are initially sequential appends to memory and then flushed as sorted files, there's minimal internal fragmentation in the initial stages. SSTables on disk tend to be densely packed.
- **Efficient Handling of Updates and Deletes:** Instead of immediately updating a record in place (like a B-tree), LSM trees often insert a new record representing the update or a "tombstone" record for a delete. During compaction, older versions of the record or deleted records can be efficiently discarded when merging SSTables. This avoids the need to immediately reclaim space and rewrite data as B-trees might do.
- **Bulk Operations during Compaction:** Compaction allows for efficient bulk operations on sorted data. Merging sorted runs is a more space-efficient operation than the random writes and potential node splits in a B-tree. During the merge, duplicate entries or outdated data can be easily identified and eliminated, leading to better space utilization in the long run.
- **Lower Write Amplification (in some cases):** While compaction itself involves writing data, the initial write path in an LSM tree is often more sequential and less prone to the random writes that contribute to higher write amplification in B-trees. This can indirectly lead to better space efficiency by reducing the amount of redundant data written.

SSD’s internally also are LSM like and in theory you can make LSM work well on modern drives.

> If you're always inserting stuff, you want the location that you're inserting stuff into to be easily accessible. With LSM trees, you're inserting stuff sequentially, so it's easy to keep that spot in CPU cache. With B-trees, you have to scan all over the tree to insert stuff, meaning stuff is flying in and out of cache as you're locating and inserting things.

- **MemTable:** While the data is in the MemTable, subsequent inserts are likely to access data in the same memory region. This exhibits **temporal locality** (recently accessed data is likely to be accessed again soon) and **spatial locality** (data located near recently accessed data is also likely to be accessed soon). This increases the chances of finding the necessary data in the CPU cache.
- **SSTable Flushes:** Even when the MemTable is flushed to disk, the operation is a sequential write. The CPU is dealing with a contiguous block of memory as it prepares the sorted data for writing. This sequential access pattern is very cache-friendly.

BTree

Cache Drawbacks:

- **Cache Misses During Traversal:** Each node access during the tree traversal might result in a cache miss if the required node is not already in the CPU cache. Since the nodes can be scattered in memory, there's a lower chance of spatial locality being exploited.
- **Cache Thrashing During Splits:** Node splits can lead to cache thrashing. This happens when the CPU repeatedly loads and evicts different cache lines in a short period, without effectively reusing the data. The operations involved in a split (allocating new memory, copying data, updating pointers) can touch different memory locations, causing data to be loaded into the cache only to be quickly replaced by other data.

In a good B+tree implementation you can get similar or better and stable insert performance.
There are many B+Tree variants to choose from these days that try and compete with some of the above mentioned strengths of LSM but they don't seem to be widely used AFAICT. --Bepsilon Trees, COW B tree

Metrics

In general, there are three critical metrics to measure the performance of a data structure: write amplification, read amplification, and space amplification. This section aims to describe these metrics.

For hard disk drives (HDDs), the cost of disk seek is enormous, such that the performance of random read/write is worse than that of sequential read/write. This article assumes that flash-based storage is used so we can ignore the cost of disk seeks.

Write Amplification

Write amplification is the ratio of the amount of data written to the storage device versus the amount of data written to the database.

For example, if you are writing 10 MB to the database and you observe 30 MB disk write rate, your write amplification is 3.

Flash-based storage can be written to only a finite number of times, so write amplification will decrease the flash lifetime.

There is another write amplification associated with flash memory and SSDs because flash memory must be erased before it can be rewritten.

Read Amplification

Read amplification is the number of disk reads per query.

For example, if you need to read 5 pages to answer a query, read amplification is 5.

Note that the units of write amplification and read amplification are different. Write amplification measures how much more data is written than the application thought it was writing, whereas read amplification counts the number of disk reads to perform a query.

Read amplification is defined separately for point query and range queries. For range queries the range length matters (the number of rows to be fetched).

Caching is a critical factor for read amplification. For example, with a B-tree in the cold-cache case, a point query requires $O(logBN)$ disk reads, whereas in the warm-cache case the internal nodes of the B-tree are cached, and so a B-tree requires at most one disk read per query(only querying the leaf node requires disk access).

Space Amplification

Space amplification is the ratio of the amount of data on the storage device versus the amount of data in the database.

For example, if you put 10MB in the database and this database uses 100MB on the disk, then the space amplification is 10.

Generally speaking, a data structure can optimize for at most two from read, write, and space amplification. This means one data structure is unlikely to be better than another at all three. For example a B-tree has less read amplification than an LSM-tree while an LSM-tree has less write amplification than a B-tree.

Summary

The following table shows the summary of various kinds of amplification:

Data Structure	Write Amplification	Read Amplification
B+ tree	$\Theta(B)$	$O(log_B N/B)$
Level-Based LSM-tree	$\Theta(klog_k N/B)$	$\Theta((log^2 N/B)/logk)$

> Table 1. A summary of the write amplification and read amplification for range queries.

Through comparing various kinds of amplification between B+ tree and Level-based LSM-tree, we can come to a conclusion that Level-based LSM-tree has a better write performance than B+ tree while its read performance is not as good as B+ tree. The main purpose for TiKV to use LSM-tree instead of B-tree as its underlying storage engine is because using cache technology to promote read performance is much easier than promote write performance.