ALGORITHMS Behind Modern Storage Systems

DIFFERENT USES FOR READ-OPTIMIZED B-TREES AND WRITE-OPTIMIZED LSM-TREES

ALEX PETROV

The amounts of data processed by applications are constantly growing. With this growth, scaling storage becomes more challenging. Every database system has its own tradeoffs. Understanding them is crucial, as it helps in selecting the right one from so many available choices.

Every application is different in terms of read/write workload balance, consistency requirements, latencies, and access patterns. Familiarizing yourself with database and storage internals facilitates architectural decisions, helps explain why a system behaves a certain way, helps troubleshoot problems when they arise, and fine-tunes the database for your workload.

It's impossible to optimize a system in all directions. In an ideal world there would be data structures guaranteeing the best read and write performance with no storage overhead, but, of course, in practice it's not possible.

This article takes a closer look at two storage system design approaches used in a majority of modern databases—read-optimized B-trees[1] and write-optimized LSM (log-structured merge)-trees[5]— and describes their use cases and tradeoffs.

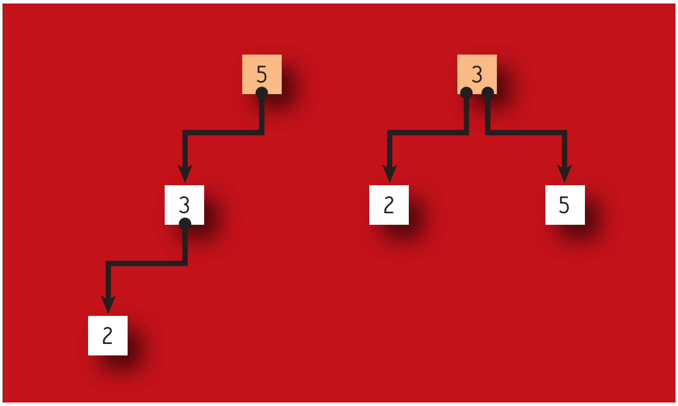
B-TREES

B-trees are a popular read-optimized indexing data structure and generalization of binary trees. They come in many variations and are used in many databases (including MySQL InnoDB[4] and PostgreSQL[7]) and even file systems (HFS+[8], HTrees in ext4[9]). The *B* in *B*-tree stands for *Bayer*, the author of the original data structure, or *Boeing*, where he worked at that time.

In binary tree every node has two children (referred as a left and a right child). Left and right subtrees hold the keys that are less than and greater than the current node key, respectively. To keep the tree depth to a minimum, a binary tree has to be balanced: when randomly ordered keys are being added to the tree, it is natural that one side of the tree will eventually get deeper than the other.

One way to rebalance a binary tree is to use so-called rotation: rearrange nodes, pushing the parent node of the longer subtree down below its child and pulling this child up, effectively placing it instead of its parent. Figure 1 is an example of rotation used for balancing in a binary tree. On the left, a binary tree is unbalanced after adding node 2 to it. In order to balance it, node 3 is used as a pivot (the tree is rotated around it). Then node 5, previously a root

FIGURE 1: **EXAMPLE OF ROTATION USED FOR BALANCING IN BINARY TREE**



node and a parent node for 3, becomes its child node. After the rotation step is done, the height of the left subtree decreases by one and the height of the right subtree increases by one. The maximum depth of the tree has decreased.

Binary trees are most useful as in-memory data structures. Because of balancing (the need to keep the depth of all subtrees to a minimum) and low fanout (a maximum of two pointers per node), they don't work well on disk. B-trees allow for storing more than two pointers per node and work well with block devices by matching the node size to the page size (e.g., 4 KB). Some implementations today use larger node sizes, spanning across multiple pages in size.

B-trees have the following properties:

➡ *Sorted.* This allows sequential scans and simplifies lookups.

➡ *Self-balancing.* There's no need to balance the tree during insertion and deletion: when a B-tree node is full, it's split in two; when occupancy of the neighboring nodes falls down to a certain threshold, the nodes are merged. This also means that leaves are equally distant from the root and can be located in the same amount of steps during lookup.

➡ *Guarantee of logarithmic lookup time.* This makes B-trees a good choice for database indexes, where lookup times are important.

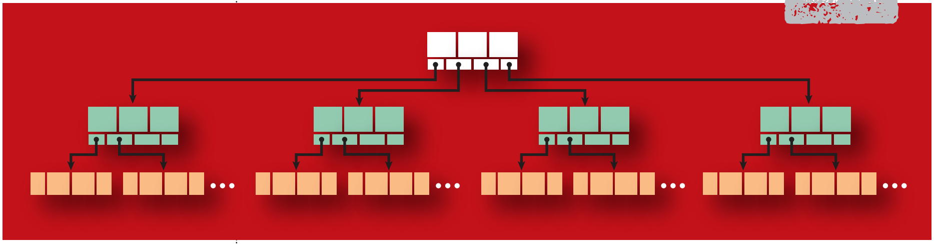
➡ *Mutable.* Inserts, updates, and deletes (also, subsequent splits and merges) are performed on disk in place. To make in-place updates possible, a certain amount of space overhead is required. A B-tree can be organized as a clustered index, where actual data is stored on the leaf nodes or as a heap file with an unclustered B-tree index.

This article discusses the B+tree,[3] which is a modern variant of the B-tree and often used for database storage. The B+tree is different from the original B-tree[1] in that: (a) it has an additional level of linked leaf nodes holding the values; and (b) these values cannot be stored on internal nodes.

Anatomy of the B-tree

Let's first take a closer look at the B-tree building blocks, illustrated in figure 2. B-trees have several node types: root, internal, and leaf. Root (top) is the node that has no parents (i.e., it is not a child of any other node). Internal nodes (middle) have both a parent and children; they connect a root node with leaf nodes. Leaf nodes (bottom) carry the data and have no children. Figure 2 depicts a B-tree with a branching factor of four (four pointers, three keys in internal nodes, and four key/value pairs on leaves).

FIGURE 2: **EXAMPLE OF A B-TREE**



B-trees are characterized by the following:

- *Branching factor*–the number (*N*) of pointers to the child nodes. Along with the pointers, root and internal nodes hold up to *N-1* keys.
- *Occupancy*–how many pointers to child items the node is currently holding, out of the maximum available. For example, if the tree-branching factor is *N*, and the node is currently holding *N/2* pointers, occupancy is 50 percent.
- *Height*–the number of B-tree levels, signifying how many pointers have to be followed during lookup.

Every nonleaf node in the tree holds up to *N* keys (index entries), separating the tree into N+1 subtrees that can be located by following a corresponding pointer. Pointer *i* from an entry K_i points to a subtree in which all the index entries are such that $K_{i-1} <= K_{searched} < K_i$ (where *K* is a set of keys). The first and the last pointers are the special cases, pointing to subtrees in which all the entries are less than or equal to K_0 in the case of the leftmost child, or greater than K_{N-1} in the case of the rightmost child. A leaf node may also hold a pointer to the previous and next nodes on the same level, forming a doubly linked list of sibling nodes. Keys in all the nodes are always sorted.

Lookups

When performing lookups, the search starts at the root node and follows internal nodes recursively down to the leaf level. On each level, the search space is reduced to the child subtree (the range of this subtree includes the searched value) by following the child pointer. Figure 3 shows a lookup in a B-tree making a single root-to-leaf pass, following the pointers "between" the two keys, one of which is greater than (or equal to) the searched term, and the other of which is less than the searched term. When a point query is performed, the search is complete after locating the leaf node. During the range scan, the keys and values of the found leaf, and then the sibling leaf's nodes are traversed, until the end of the range is reached.

In terms of complexity, B-trees guarantee *log(n)* lookup, because finding a key within the node is performed using

FIGURE 3: **SINGLE ROOT-TO-LEAF PASS**

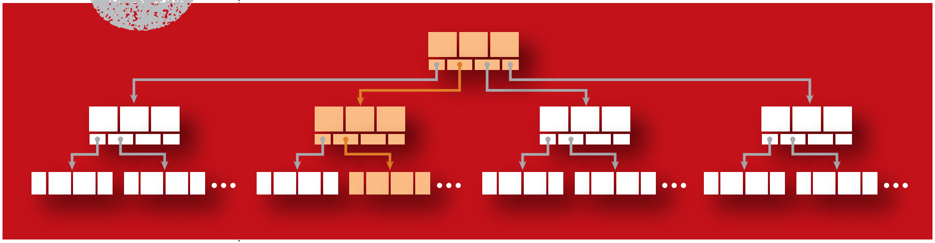
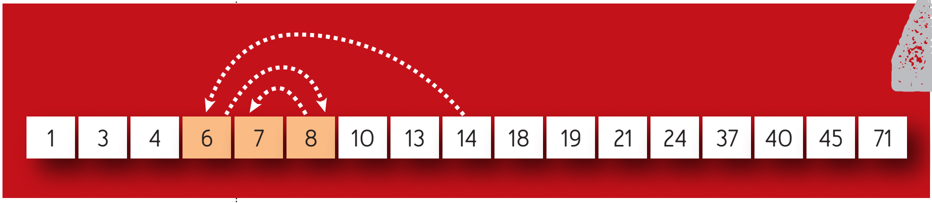


FIGURE 4: **BINARY SEARCH OF A B-TREE**



binary search, shown in figure 4. Binary search is easily explained in terms of searching for words beginning with a certain letter in the dictionary, where all words are sorted alphabetically. First you open the dictionary exactly in the middle. If the searched letter is alphabetically "less than" (appears earlier than) the one opened, you continue your search in the left half of the dictionary; otherwise, you continue in the right half. You keep reducing the remaining page range by half and picking the side to follow until you find the searched letter. Every step halves the search space, therefore making the lookup time logarithmic. Searches in B-trees have logarithmic complexity, since on the node level keys are sorted, and the binary search is performed in order to find a match. This is also why it's important to keep the occupancy high and uniform across the tree.

Insertions, updates, and deletions

When performing insertions, the first step is to locate the target leaf. For that, the aforementioned search algorithm is used. After the target leaf is located, key and value are appended to it. If the leaf does not have enough free space, the situation is called overflow, and the leaf has to be split in two. This is done by allocating a new leaf, moving half the elements to it and appending a pointer to this newly allocated leaf to the parent. If the parent doesn't have free space either, a split is performed on the parent level as well. The operation continues until the root is reached. When the root overflows, its contents are split between the newly allocated nodes, and the root node itself is overwritten in order to avoid relocation. This also implies that the tree (and its height) always grows by splitting the root node.

LSM-TREES

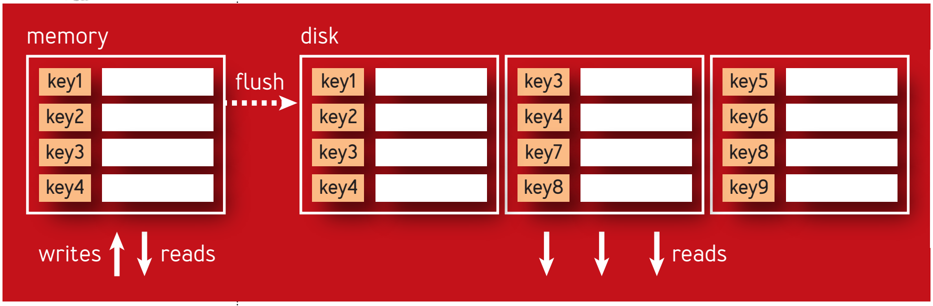
The log-structured merge-tree is an immutable disk-resident write-optimized data structure. It is most useful in systems where writes are more frequent than lookups that retrieve the records. LSM-trees have been getting more attention because they can eliminate random insertions, updates, and deletions.

Anatomy of the LSM-tree

To allow sequential writes, LSM-trees batch writes and updates in a memory-resident table (often implemented using a data structure allowing logarithmic time lookups, such as a binary search tree or skip list) until its size reaches a threshold, at which point it is written on disk (this operation is called a *flush*). Retrieving the data requires searching all disk-resident parts of the tree, checking the in-memory table, and merging their contents before returning the result. Figure 5 shows the structure of an LSM-tree: a memory-resident table used for writes. Whenever the memory table is large enough, its sorted contents are written on disk. Reads are served, hitting

5

FIGURE 5: **STRUCTURE OF AN LSM TREE**



both disk- and memory-resident tables, requiring a merge process to reconcile the data.

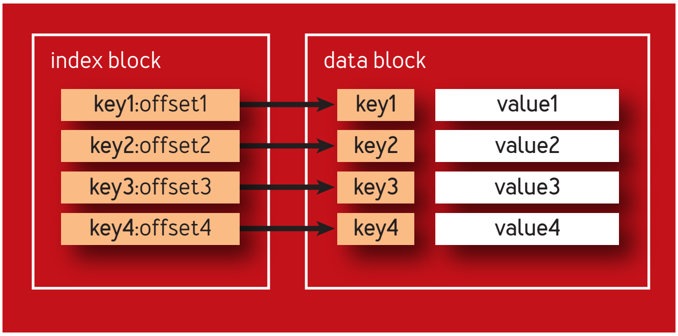
Sorted String Tables

Many modern LSM-tree implementations (such as RocksDB and Apache Cassandra) implement disk-resident tables as SSTables (Sorted String Tables), because of their simplicity (easy to write, search, and read) and merge properties (during the merge, source SSTable scans and merged result writes are sequential).

An SSTable is a disk-resident ordered immutable data structure. Structurally, an SSTable is split in two parts: data and index blocks, as shown in figure 6. A data block consists of sequentially written unique key/value pairs, ordered by key. An index block contains keys mapped to data-block pointers, pointing to where the actual record is located. An index is often implemented using a format optimized for quick searches, such as a B-tree, or using a hash table for a point query. Every value item in an SSTable has a timestamp associated with it. This specifies

FIGURE 6: **STRUCTURE OF AN SSTABLE**

6



the write time for inserts and updates (which are often indistinguishable) and removal time for deletes.

SSTables have some nice properties:

➡ Point queries (i.e., finding a value by key) can be done quickly by looking up the primary index.

➡ Scans (i.e., iterating over all key/value pairs in a specified key range) can be done efficiently simply by reading key/value pairs sequentially from the data block.

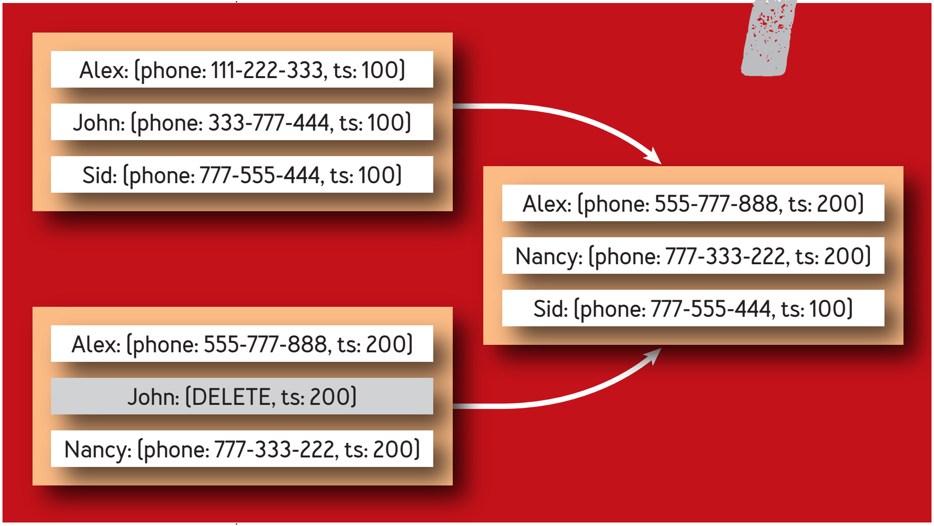
An SSTable represents a snapshot of all database operations over a period in time, as the SSTable is created by the *flush* process from the memory-resident table that served as a buffer for operations against the database state for this period.

Lookups

Retrieving data requires searching all SSTables on disk, checking the memory-resident tables, and merging their contents together before returning the result. The merge step during the read is required since the searched data can reside in multiple SSTables.

The merge step is also necessary to ensure that the deletes and updates work. Deletes in an LSM-tree insert placeholders (often called *tombstones*), specifying which key was marked for deletion. Similarly, an update is just a record with a bigger timestamp. During the read, the records that get shadowed by deletes are skipped and not returned to the client. A similar thing happens with the updates: out of two records with the same key, only the one with the later timestamp is returned. Figure 7 shows a merge step reconciling the data stored in separate tables for the same key: as shown here, the record for Alex was

FIGURE 7: **EXAMPLE OF A MERGE STEP**


Alex: (phone: 111-222-333, ts: 100)
John: (phone: 333-777-444, ts: 100)
Sid: (phone: 777-555-444, ts: 100)
Alex: (phone: 555-777-888, ts: 200)
John: (DELETE, ts: 200)
Nancy: (phone: 777-333-222, ts: 200)
Alex: (phone: 555-777-888, ts: 200)
Nancy: (phone: 777-333-222, ts: 200)
Sid: (phone: 777-555-444, ts: 100)

written with timestamp 100 and updated with a new phone and timestamp 200; the record for John was deleted. The other two entries are taken as is, as they're not shadowed.

To reduce the number of searched SSTables and to avoid checking every SSTable for the searched key, many storage systems employ a data structure known as a Bloom filter[10]. This is a probabilistic data structure that can be used to test whether an element is a member of the set. It can produce false positive matches (i.e., state that the element is a member of the set, while it is not, in fact, present there) but cannot produce false-negatives (i.e., if a negative match is returned, the element is guaranteed not to be a member of the set). In other words, a Bloom filter is used to tell if the key "might be in an SSTable" or "is definitely not in an SSTable." SSTables for which a Bloom filter has returned a negative match are skipped during the query.

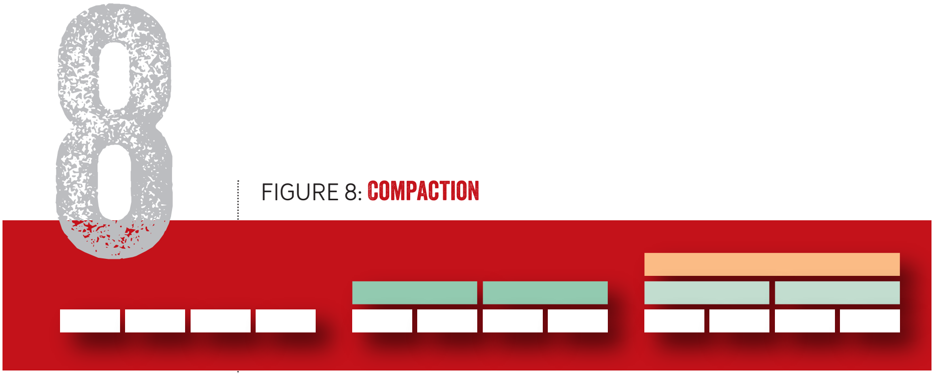
LSM-tree maintenance

Since SSTables are *immutable*, they are written sequentially and hold no reserved empty space for in-place modifications. This means insert, update, or delete operations would require rewriting the whole file. All operations modifying the database state are "batched" in the memory-resident table. Over time, the number of disk-resident tables will grow (data for the same key located in several files, multiple versions of the same record, redundant records that got shadowed by deletes), and the reads will continue getting more expensive.

To reduce the cost of reads, reconcile space occupied by shadowed records, and reduce the number of disk-resident tables, LSM-trees require a *compaction* process that reads complete SSTables from disk and merges them. Because SSTables are sorted by key and compaction works like merge-sort, this operation is very efficient: records are read from several sources sequentially, and merged output can be appended to the results file right away, also sequentially. One of the advantages of merge-sort is that it can work efficiently even for merging large files that don't fit in memory. The resulting table preserves the order of the original SSTables.

During this process, merged SSTables are discarded and replaced with their "compacted" versions, shown in figure 8. Compaction takes multiple SSTables and merges them into one. Some database systems logically group the tables of the same size to the same "level" and start the

FIGURE 8: **COMPACTION**



merge process whenever enough tables are on a particular level. After compaction, the number of SSTables that have to be addressed is reduced, making queries more efficient.

ATOMICITY AND DURABILITY

To reduce the number of I/O operations and make them sequential, both B-trees and LSM-trees batch operations in memory before making an actual update. This means that data integrity is not guaranteed during failure scenarios and *atomicity* (applying a series of changes atomically, as if they were a single operation, or not applying them at all) and *durability* (ensuring that in the face of a process crash or power loss, data has reached persistent storage) properties are not ensured.

To solve that problem, most modern storage systems employ WAL (write-ahead logging). The key idea behind WAL is that all the database state modifications are first durably persisted in the append-only log on disk. If the process crashes in the middle of an operation, the log is replayed, ensuring that no data is lost and all changes appear atomically.

In B-trees, using WAL can be understood as writing changes to data files only after they have been logged. Usually log sizes for B-tree storage systems are relatively small: as soon as changes are applied to the persisted storage, they can be discarded. WAL serves as a backup for the in-flight operations: any changes that were not applied to data pages can be redone from the log records.

In LSM-trees, WAL is used to persist changes that have reached the memtables but have not yet been fully flushed on disk. As soon as a memtable is fully flushed and switched so that read operations can be served from the newly created SSTable, the WAL segment holding the data for the flushed memtable can be discarded.

SUMMARIZING

One of the biggest differences between the B-tree and LSM-tree data structures is what they optimize for and what implications these optimizations have.

Let's compare the properties of B-trees with LSM-trees. In summary, B-trees have the following properties:

➡ They are mutable, which allows for in-place updates by introducing some space overhead and a more involved write path, although it does not require complete file rewrites or multisource merges.

➡ They are read-optimized, meaning they do not require reading from (and subsequently merging) multiple sources, thus simplifying the read path.

➡ Writes might trigger a cascade of node splits, making some write operations more expensive.

➡ They are optimized for paged environments (block storage), where byte addressing is not possible.

➡ Fragmentation, caused by frequent updates, might require additional maintenance and block rewrites. B-trees, however, usually require less maintenance than LSM-tree storage.

- Concurrent access requires reader/writer isolation and involves chains of locks and latches.

LSM-trees have these properties:

- They are immutable. SSTables are written on disk once and never updated. Compaction is used to reconcile space occupied by removed items and merge same-key data from multiple data files. Merged SSTables are discarded and removed after a successful merge as part of the compaction process. Another useful property coming from immutability is that flushed tables can be accessed concurrently.
- They are write optimized, meaning that writes are buffered and flushed on disk sequentially, potentially allowing for spatial locality on the disk.
- Reads might require accessing data from multiple sources, since data for the same key, written during different times, might land in different data files. Records have to go through the merge process before being returned to the client.
- Maintenance/compaction is required, as buffered writes are flushed on disk.

EVALUATING STORAGE SYSTEMS

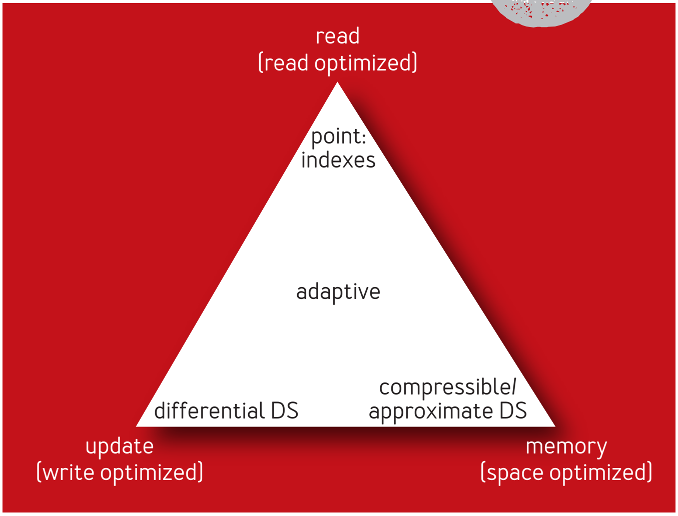
Developing storage systems always presents the same challenges and factors to consider. Deciding what to optimize for has a substantial influence on the result. You can spend more time during write in order to lay out structures for more efficient reads, reserve extra space for in-place updates, facilitate faster writes, and buffer data in memory to ensure sequential write operations. It is impossible, however, to do this all at once. An ideal storage system would have the lowest read cost, lowest write cost, and no overhead. In practice, data structures compromise among multiple factors. Understanding these compromises is important.

Researchers from Harvard's DASlab (Data System Laboratory) summarized the three key parameters database systems are optimized for: read overhead, update overhead, and memory overhead, or RUM. Understanding which of these parameters are most important for your use-case influences the choice of data structures, access methods, and even suitability for certain workloads, as the algorithms are tailored having a specific use-case in mind.

The RUM Conjecture[2] states that setting an upper bound for two of the mentioned overheads also sets a lower bound for the third one. For example, B-trees are read-optimized at the cost of write overhead and having to reserve empty space for the updates (thereby resulting in memory overhead). LSM-trees have less space overhead at a cost of read overhead brought on by having to access multiple disk-resident tables during the read. These three parameters form a competing triangle, and improvement on one side may imply compromise on the other. Figure 9 illustrates the RUM Conjecture.

B-trees optimize for read performance: the index is laid out in a way that minimizes the disk accesses required to traverse the tree. Only a single index file has to be accessed to locate the data. This is achieved by keeping this index file mutable, which also increases write amplification resulting from node splits and merges, relocation, and fragmentation/imbalance-related maintenance. To

FIGURE 9: **THE RUM CONJECTURE**



amortize update costs and reduce the number of splits, B-trees reserve extra free space in nodes on all levels. This helps postpone write amplification until the node is full. In short, B-trees trade update and memory overhead for better read performance.

LSM-trees optimize for write performance. Neither updates nor deletes require locating data on disk (which is the case with B-trees), and they guarantee sequential writes by buffering all insert, update, and delete operations in memory-resident tables. This comes at the price of higher maintenance costs and a need for compaction (which is just a way of mitigating the ever-growing price of reads and reducing the number of disk-resident tables) and more expensive reads (as the data has to be read from multiple sources and merged). At the same time, LSM-trees eliminate memory overhead by not reserving any empty space (unlike B-tree nodes, which have an average occupancy of 70 percent, an overhead required for in-place updates) and allowing block compression because of the better occupancy and immutability of the end file. In short, LSM-trees trade read performance and maintenance for better write performance and lower memory overhead.

There are data structures optimizing for each desired property. Using adaptive data structures allows for better read performance at the price of higher maintenance costs. Adding metadata facilitating traversals (such as fractional cascading) will have an impact on write time and take space, but can improve the read time. Optimizing for memory efficiency using compression (for example, algorithms such as Gorilla compression,[6] delta encoding, and many others) will add some overhead for packing the data on writes and unpacking it on reads. Sometimes, you can trade functionality for efficiency. For example, heap files and hash indexes can provide great performance guarantees and smaller space overhead because of the file format simplicity, for the price of not being able to perform anything but point queries. You can also trade precision for space and efficiency by using approximate data structures, such as the Bloom filter, HyperLogLog, Count-Min sketch, and many others.

The three tunables—read, update, and memory overheads—can help you evaluate the database and gain a deeper understanding of the workloads for which it's best suited. All of them are quite intuitive, and it's often easy to sort the storage system into one of the buckets and guess how it's going to perform, then validate your hypothesis through extensive testing.

Of course, there are other important factors to consider when evaluating a storage system, such as maintenance overhead, operational simplicity, system requirements, suitability for frequent updates and deletes, access patterns, and so on. The RUM Conjecture is just a rule of thumb that helps develop an intuition and provide an initial direction. Understanding your workload is the first step on the way to building a scalable back end.

Some factors may vary from implementation to implementation, and even two databases that use similar storage-design principles may end up performing differently. Databases are complex systems with many moving parts and are an important and integral part of many applications. This information will help you peek under the hood of a database and, knowing the difference between the underlying data structures and their inner doings, decide what's best for you.

Related articles

The Five-minute Rule: 20 Years Later and How Flash Memory Changes the Rules
Goetz Graefe, Hewlett-Packard Laboratories
The old rule continues to evolve, while flash memory adds two new rules.
https://queue.acm.org/detail.cfm?id=1413264

Disambiguating Databases
Rick Richardson
Use the database built for your access model.
https://queue.acm.org/detail.cfm?id=2696453

You're Doing It Wrong
Poul-Henning Kamp
Think you've mastered the art of server performance? Think again.
https://queue.acm.org/detail.cfm?id=1814327

References

1. Comer, D. 1979. The ubiquitous B-tree. *Computing Surveys* 11(2); 121-137; http://citeseerx.ist.psu.edu/viewdoc/

download?doi=10.1.1.96.6637&rep=rep1&type=pdf.

2. Data Systems Laboratory at Harvard. The RUM Conjecture; http://daslab.seas.harvard.edu/rum-conjecture/.
3. Graefe, G. 2011. Modern B-tree techniques. *Foundations and Trends in Databases* 3(4): 203-402; http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.219.7269&rep=rep1&type=pdf.
4. MySQL 5.7 Reference Manual. The physical structure of an InnoDB index; https://dev.mysql.com/doc/refman/5.7/en/innodb-physical-structure.html.
5. O'Neil, P., Cheng, E., Gawlick, D., O'Neil, E. 1996. The log-structured merge(LSM-tree). *Acta Informatica* 33(4): 351-385; http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.44.2782&rep=rep1&type=pdf.
6. Pelkonen, T., Franklin, S.Teller, J., Cavallaro, P., Huang, Q., Meza, J., Veeraraghavan, K. 2015. Gorilla: a fast, scalable, in-memory time series database. *Proceedings of the VLDB Endowment* 8(12): 1816-1827; http://www.vldb.org/pvldb/vol8/p1816-teller.pdf.
7. Suzuki, H. 2015-2018. The internals of PostreSQL; http://www.interdb.jp/pg/pgsql01.html.
8. Apple HFS Plus Volume Format; https://developer.apple.com/legacy/library/technotes/tn/tn1150.html#BTrees
9. Mathur, A., Cao, M., Bhattacharya, S., Dilger, A., Tomas, A., Vivier, L. (2007). The new ext4 filesystem: current status and future plans. Proceedings of the Linux Symposium. Ottawa, Canada: Red Hat. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.111.798&rep=rep1&type=pdf
10. Bloom, Burton H. (1970),"Space/time trade-offs in hash

coding with allowable errors". *Communications of the ACM*, 13 (7): 422–426

Alex Petrov *(http://coffeenco.de/, @ifesdjeen) is an Apache Cassandra committer and storage-systems enthusiast. Over the past several years, he has worked on databases, building distributed systems and data-processing pipelines for various companies.*

CONTENTS