Check for
updates

Write Amplification Versus Read Perspiration

BY PAT HELLAND

THE TRADEOFFS BETWEEN WRITE AND READ

Increasingly in computing systems, when you write something into durable storage it is in need of reorganization later. Personally, I'm pretty darned disorganized and I lose stuff a lot. This causes extensive searching, sometimes to no avail. It is, however, easier to "store" stuff by setting it down wherever I feel like it.

In computing, there's an interesting trend where writing creates a need to do more work. You need to reorganize, merge, reindex, and more to make the stuff you wrote more useful. If you don't, you must search or do other work to support future reads.

INDEXING WITHIN A DATABASE

My first job programming was to implement a database system. In 1978, my colleague and I didn't even know what that was! We voraciously read every ACM SIGMOD (Special Interest Group on Management of Data) and TODS (Transactions on Database Systems) paper we could lay our hands on. We learned about this interesting and confusing concept of a relational database and how indexing can optimize access while being transparent to the application. Of course, updating an index meant another two disk accesses since the indices of a B+ tree didn't fit in memory. We understood that the additional work to make database changes was worth it if you were

*It's Not Your Grandmother's
Database Anymore

ever going to read it later.

The next perplexing question was, how much should be indexed? Should we index every column? When should a pair of columns be indexed together? The more indexing we did, the faster the read queries would become. The more indexing we did, the more our ability to update became slower than molasses.

I learned that this is a common tradeoff. Reading fast frequently means writing slow.

ROW-STORE VERSUS COLUMN-STORE

I've focused most of my misspent career on distributed systems and OLTP (online transaction processing)-style databases. It's natural for me to associate high-performance updates with what today is called a *row-store*.

Another approach is to organize data by columns: Take a bunch of rows and organize the data by its column values. Every row containing the state of California, for example, keeps just the single column's data together. Columnar databases are super-fast for doing queries because many logical rows with the same value are physically close to each other.

Updating a *column-store* is not as easy, though. Typically, updates are kept separately in an integrated row-store. Queries check the small row-store in a fashion that's relatively fast because it's small. These queries are combined with the results of the faster column-store to give a unified accurate answer. Periodically, the new row-store updates are merged with the column-store to make a new column-store. This may be done in a cascading fashion

Linear searches for your wallet might be tractable in a small apartment but not so much when the search space gets bigger in a larger home in the suburbs.

somewhat like the merges in an LSM (log-structured merge) tree, described in the next section.

When inserting into a column-store (or really its attached row-store), you are incurring a debt to be paid later. This debt to rewrite and integrate the new data is a form of *write amplification* where a single write turns into more writes later.

LSM: LOG-STRUCTURED MERGE TREES

Log-structured merge trees were first proposed in 1996.⁶ The idea is to track changes to a key-value store as transactions, with new values kept in memory. As transactions commit, the sorted collection of recent key-value pairs can be written to disk in a uniquely named file. This file contains the sorted key-value pairs along with an index into the keys in the file. Once written to disk, the newly committed changes do not need to be kept in memory.

Now, if you keep doing this, looking up values by key starts looking like what happens to me when I try to find something I set down in some random place. Linear searches for your wallet might be tractable in a small apartment but not so much when the search space gets bigger in a larger home in the suburbs. To reduce the *read perspiration*, LSM trees invest energy to organize the data by rewriting it as you go.

When a new file is freshly written from the storage engine, it has a bunch of key-value pairs. To make it easy to find keys, these are merged with files that were written earlier. Each LSM tree has some form of fan-out where lower levels of the tree (with keys written earlier) are kept

across more files. For example, you may have 10 times as many files at level 1 as at the brand-new level 0. Each file at level 1 has approximately one-tenth as large a key range represented but approximately 10 times the amount of update time represented. Similarly, moving down to level 2 results in 100 files, each with a narrower key range and longer time range.

The depth of an LSM tree depends on the fan-out, the size of each file, and the number of key-value pairs in the tree. In general, most of the storage is in the lowest level of the tree.

So, within this basic LSM structure that is gaining so much popularity, there are varieties of implementation choices. Consider:

➔ **Leveling merges.** When a new file is added to a level, pick the next file in the round-robin traversal and merge it with the files in the next level below. Suppose you pick a fan-out of 10; you will find that the key range in the file dropping down typically covers the key range in about 10 files in the level below. You merge 11 files together as one drops down onto 10 and you get 11 files out. Now, the next level has gotten fatter by one file, so you repeat and merge down again.

➔ **Tiering merges.** In this different but related approach, you let a bunch of files stack up on each level before doing the merge. Say you stack up 10 files before you merge down at each level. That dramatically reduces the amount of merging required.

Leveling merges have a large write amplification. Each write of a new key-value pair to level 0 will be rewritten 10 or 11 times at each level it moves through. On the other

Search is in many ways a variation of database indexing.

hand, they have a small read perspiration, as a reader typically checks only one place per level.

Tiering merges have a much lower write amplification but a larger read perspiration. Because new files stack up at each level before merging, there's less merging and hence less writing. On the other hand, reads need to check a lot more places, leading to the larger read perspiration.

There's a bunch of fun work lately on the tradeoffs of these schemes.^{2,5}

INDEXING AND SEARCHING

Search is in many ways a variation of database indexing. In database indices, the notion of identity exists hidden within the database as a row-id or a primary key. Within a relational system, updates to indices are transactionally integrated, and the user sees only a performance difference.

Search systems are a bit different in that they deal with documents. Most search systems asynchronously update the search index *after* the change to the document occurs. This is knit together with some form of document identity.³

Search makes reading the documents a lot easier. It dramatically lowers the read perspiration. Updates to the documents asynchronously impose a debt onto the system to get them indexed. Creating and merging search indices is a complex job that I think of as a form of write amplification.

To index, you need to scour the corpus to find recently written or updated documents. Each of these needs to have an identifier and then needs to be processed to locate the search terms (sometimes called *n*-grams;

<https://en.wikipedia.org/wiki/N-gram>). Each of these many n -grams found in a typical document then needs to be sent to an indexer that covers one of many shards. So, the document identifier now is associated with each term (or n -gram) located in the searchable document—all of this because the user did a write or created a document!

I worked for a few years on an Internet-scale search engine and know how they work. I'm still in awe that all this machinery can keep up with the work involved in all that write amplification. It's a lot of work for each document written—and there are lots and lots of documents.

Internet-scale search systems clearly offer excellent and low read perspiration.

LARGE-SCALE CACHES

Lots of big Internet systems have ginormous caches. Consider a product catalog at a big ecommerce retailer. Whenever anything changes, lots of servers are updated with the new product description. This makes for a very easy and fast read in exchange for a lot of writes.

NORMALIZATION AND DENORMALIZATION

Growing up in the relational database world, I was imbued with the determination to have normalized data contained in the database. Working to avoid update anomalies was deemed to be extremely important. Performing a large number of joins to get an answer was a small penalty to pay to ensure the database wasn't damaged by an errant update.

Increasingly, I view this as the equivalent of throwing salt over your shoulder if you spill some. Yeah... I've seen

others do it, but I'm not sure I should.

Most systems are getting more and more distributed. Most of these have key-value pairs containing their data, which is sharded for scale. By grouping related data into the value of a pair—typically in a JSON (JavaScript Object Notation) representation or something similar—it's easy to grab the value, perhaps as a string, and squirt it over to the distant system issuing the request.

If you *were* to normalize the data in this big and sharded system, the normalized values would not be on the same shard together. Doing a distributed join is more annoying than doing a centralized join.

Related articles

➡ Immutability Changes Everything

We need it, we can afford it,
and the time is now

Pat Helland

<https://queue.acm.org/detail.cfm?id=2884038>

➡ Disambiguating Databases

Use the database built
for your access model.

Rick Richardson

<https://queue.acm.org/detail.cfm?id=2696453>

➡ The Pathologies of Big Data

Scale up your datasets enough and
all your apps will come undone.

Adam Jacobs

<https://queue.acm.org/detail.cfm?id=1563874>

To cope with this, people superimpose versioning on their data. It's not perfect but it's less challenging than distributed joins or trying to do massive updates across the denormalized data. The classic example for the value of normalization in databases is a denormalized table with employees, their manager, and their manager's phone number.⁴ Because the manager's phone number is copied in many tables for many employees, it's hard to change it. Increasingly, I see systems store “as-of” data in their denormalized structures—for example, the manager's phone is captured “as-of” June 1.

Large-scale distributed systems put a lot of pressure on the semantics of a consistent read. This, in turn, can be seen as a tension between write amplification and read perspiration.

CONCLUSION

We've looked at just a few of the examples where there are tradeoffs in our systems between write and read.¹ It is endemic in so many environments. We see emerging systems that adapt and optimize for these tradeoffs as they watch their usage patterns. Fun stuff!

References

1. Athanassoulis, M., Kester, M. S., Maas, L. M., Stoica, R., Idreos, S., Ailamaki, A., Callaghan, M. 2016. Designing access methods: the RUM conjecture. In *Proceedings of the 19th International Conference on Extending Database Technology* (EDBT).
2. Dayan, N., Idreos, S. 2018. Dostoevsky: better space-time tradeoffs for LSM-tree-based key-value stores via adaptive removal of superfluous merging. In *Proceedings of the International Conference on Management of Data* (ACM SIGMOD), 505–520.
3. Helland, P. 2019. Identity by any other name. *Communications of the ACM* 62(4), 80.
4. Helland, P. 2007. Normalization is for sissies; <https://blogs.msdn.microsoft.com/pathelland/2007/07/23/normalization-is-for-sissies/>.
5. Luo, C., Carey, M. J. Forthcoming. LSM-based storage techniques. *Computing Surveys*. arXiv:1812.07527.
6. O'Neil, P., Cheng, E., Gawlick, D., O'Neil, E. 1996. The log-

structured merge-tree (LSM-tree). *Acta Informatica* 33(4).

Pat Helland *has been implementing transaction systems, databases, application platforms, distributed systems, fault-tolerant systems, and messaging systems since 1978. For recreation, he occasionally writes technical papers. He currently works at Salesforce.*

Copyright © 2019 held by owner/author. Publication rights licensed to ACM.

SHAPE THE FUTURE OF COMPUTING!

We're more than computational theorists, database managers, UX mavens, coders and developers. We're on a mission to solve tomorrow. ACM gives us the resources, the access and the tools to invent the future.

Join ACM today at acm.org/join

BE CREATIVE. STAY CONNECTED. KEEP INVENTING.

Association for
Computing Machinery

Advancing Computing as a Science & Profession