DOI:10.1145/2844112

Article development led by acmqueue
queue.acm.org

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

BY PAT HELLAND

Immutability Changes Everything

THERE IS AN inexorable trend toward storing and sending immutable data. We *need immutability* to coordinate at a distance, and we *can afford immutability* as storage gets cheaper. This article offers an amuse-bouche of repeated patterns of computing that leverage immutability. Climbing up and down the compute stack really does yield a sense of déjà vu all over again.

It was not that long ago that computation was expensive, disk storage was expensive, DRAM (dynamic random access memory) was expensive, but coordination with latches was cheap. Now all these have changed using cheap computation (with many-core), cheap commodity disks, and cheap DRAM and SSDs (solid-state drives), while coordination with latches has become harder because latch latency loses lots of instruction opportunities. Keeping immutable copies of lots of data is now affordable, and one payoff is reduced coordination challenges.

Storage is increasing as the cost per terabyte of disk keeps dropping. This means a lot of data can be kept for a long time. Distribution is increasing as more and more data and work are spread across a great distance. Data within a data center seems "far away." Data within a many-core chip may seem "far away." Ambiguity is increasing when trying to coordinate with systems that are far away—more stuff has happened since you have heard the news. Can you take action with incomplete knowledge? Can you wait for enough knowledge?

Turtles all the way down.[17] As various technological areas have evolved, they have responded to these trends of increasing storage, distribution, and ambiguity by using immutable data in some very fun ways. This article explores how apps use immutability in their ongoing work, how they generate an immutable dataset for later offline analysis, how SQL can expose and process immutable snapshots, and how massively parallel big-data work relies on immutable datasets. This leads to looking at the ways in which semantically immutable dataset may be altered while remaining immutable.

Next, the article considers how updatability is layered atop the creation of new immutable files via techniques such as LSF (log-structured file system), COW (copy-on-write), and LSM (log-structured merge-tree). How do replicated and distributed file systems depend on immutability to eliminate anomalies? Hardware folks have joined the party by leveraging these tricks in SSDs and HDDs (hard-disk drives). Immutability is a key architectural concept at many layers of the stack, as shown in Figure 1.

Finally, the article looks at some of the trade-offs of using immutable data.

IMAGE BY EVA OZKOIDI

Accountants Don't Use Erasers

Many kinds of computing are *append-only*. This section looks at some of the ways this is commonly accomplished.

In append-only computing, observations are recorded forever (or for a long time). Derived results are calculated on demand (or periodically precalculated).

This is similar to a DBMS in which transaction logs record all the changes made to the database. High-speed appends are the only way to change the log. From this perspective, the contents of the database hold a caching of the latest record values in the logs. The truth is the log. The database is a cache of a subset of the log. That cached subset happens to be the latest value of each record and index value from the log.

Accounting: Observed and derived facts. Accountants don't use erasers; otherwise they may go to jail. All entries in a ledger remain in the ledger. Corrections can be made but only by making new entries in the ledger. When a company's quarterly results are published, they include small corrections to the previous quarter. Small fixes are OK. They are append-only, too.

Some entries describe *observed facts*. For example, receiving a debit or credit against a checking account is an observed fact. Some entries describe *derived facts*, meaning that based on the observations, something new can be calculated. For example, amortized capital expenses based upon a rate and a cost are derived facts. Another example is the current bank account balance with applied debits and credits.

Append-only distributed single master. Single-master computing means changes are ordered somehow. The order can come from a centralized master or some Paxos-like[11] distributed protocol providing serial ordering. Changes are semantically applied one at a time and are layered over their predecessors. New values supersede old ones. The granularity of this may be a set of records in a relational store or a new version of a document. Distributed single-master computing means there is a space of data (relational records, documents, export files, and more) that emanates from one logical location with new versions over time.

Distributed computing "back in the day." Before telephones, people used messengers—often kids walking through town to deliver the message. Alternatively, the postal service delivered the messages, which took a long time. Sometimes people used fancy forms with many layers, each a different color. They had multiple sections on the page. Each participant filled out a section (pressing hard with the pen), then tore off the back page of the form and filed it. Each participant got the data needed and added more data to the form. Earlier sections could not be updated; data could only be appended to the end.

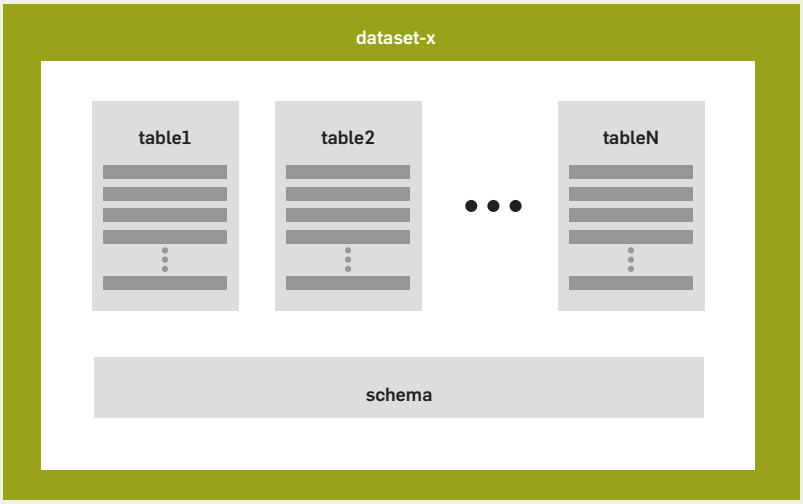
Figure 1. Immutability is a key architectural concept at many layers of the stack.

Layers	Usages of Immutable Data
Append-only apps	App over immutable data: record facts, then derive
App-generated datasets	Generate immutable data
Massively parallel big data	Read and write immutable datasets
SQL snapshots and datasets	Generate immutable data
Subjectively immutable datasets	Interpret data as immutable
LSF, LSM, and COW	Expose change over immutable files by append
Immutable files	Replication of files/blocks without update anomalies
Wear leveling on SSD	Change via COW to spread physical update blocks
Shingles on HDD	Change via COW to allow large physical rewrites

Figure 2. Characteristics of inside data and outside (immutable) data.

	Inside Data	Outside Data
Changeable	Yes!	No! Immutable
Granularity	Relational field	Document, file, or message
Representation	Typically relational	Typically semi-structured
Schema	Prescriptive	Descriptive
Identity	No identity: Data by values	Identity: URL, Msg#, Doc-ID...
Versioning	No versioning: data by value	Versions may augment identity

Figure 3. A dataset is a logical set of immutable tables and its schema.



Before computers, workflow was frequently captured in paper forms with multiple parts on the form and multiple pages (for example, "Fill out Part 3 and keep the goldenrod page from the back"). This "distributed computing" was append-only. New messages were new additions to the form—each was a version and each was immutable. You were never allowed to overwrite what had been written.

Data on the Outside vs. Data on the Inside

Surprisingly (to database old-timers), not all data is kept in relational database systems. This section (based on an earlier paper[7]) discusses some of the implications of unlocking data.

Data on the inside refers to what is kept and managed by a classic relational database system and its surrounding application code. Sometimes this is referred to as a *service*.

Data on the inside lives in a transactional world with changes applied in a serializable fashion (or something close to that).

Data on the outside is prepared as messages, files, documents, and/or Web pages. These are sent out from a service into the world. It is also possible that outside data has been created by some other mechanism than one using databases.

Data on the outside:

- *is immutable.* Once it is written, it is never changed.
- *is unlocked.* It is not locked in the database. A copy is extracted and sent outside.
- *has identity.* When sent outside, these files, documents, and messages have a unique identity (perhaps a URL).
- *may be versioned.* Updates are not updates but new versions with a new unique identifier.

Contrasting inside vs. outside. There are deep differences in the representation, meaning, and usage of inside data versus outside data. Increasingly, data is being kept as outside (immutable) data (see Figure 2).

Referencing Immutable Data

The dataset is a collection of data with a unique ID. Some datasets have structures that look like a number of tables with schema. How are these datasets referenced by a relational database, and how do relational operators span both the DBMS and dataset?

A dataset is a fixed and immutable set of tables. The schema for each table is captured in the dataset. The contents of each table are captured when the dataset is created. Since the dataset is immutable, it is created, may be consumed for reading, and then deleted. A dataset may be relational, or they may have some other representation such as a graph, a hierarchy such as JSON (JavaScript Object Notation), or any other representation (Figure 3). A dataset is a logical set of immutable tables along with its schema.

A dataset may be referenced by an RDBMS (relational DBMS). The metadata is visible to the DBMS. The data can be accessed for a read, even though it may not be updated. The dataset may be semantically present within the relational system even if it

is physically stored elsewhere. Because the dataset is immutable, there is no need for locking and no worries about controlling updates.

Relational work on immutable datasets. A functional calculation takes a set of inputs and predictably creates a set of outputs. This can happen with a query against locked or snapshot data in a relational database, and it can happen on a big-data MapReduce-style system. In both cases, there is still an unchanging collection of data. With snapshots or some form of isolation, database data becomes semantically immutable for the duration of the calculation. With big-data calculations, the inputs are typically stored in GFS (Google File System) or HDFS (Hadoop Distributed File System) files.

There is no semantic obstacle to doing JOINs across data stored inside a relational database and data stored in an external dataset. Locking (or snapshot isolation) provides a version of the relational database, which may be joined. A named and frozen dataset may be joined with relational data (see Figure 4). You can meaningfully apply relational operations across data held in a DBMS and data held in an immutable dataset.

In some ways, the ability to work across immutable datasets and relational databases is surprising. An immutable dataset is defined with an identity and an optional version. Its schema, which describes the shape and form of the dataset at the time of its creation, is descriptive, whereas the schema held in the RDBMS is prescriptive.

This tailoring of the schema to meld the two connects the schema of the dataset (describing its data when written) with the schema of the RDBMS (describing its data as of the snapshot). Also, the JOINs and other relational operators must necessarily combine the contents of the dataset as interpreted as a set of relational tables. This sidesteps the notion of identity within the dataset and focuses exclusively on the tables as interpreted as a set of values held within rows and columns.

Immutability Is in the Eye of the Beholder

A consumer may see a dataset as immutable even if they change under the covers.

A dataset is semantically immutable. It has a set of tables, rows, and columns. It may also have semi-structured data (for example, JSON). It may have application-specific data in a proprietary format.

Dataset may be defined as a SELECTION, PROJECTION, or JOIN over a previously existing dataset. Semantically, all that data is now a part of the new dataset.

What is important about a dataset is it appears to be unchanging from the standpoint of the reader.

Optimizing a dataset for read patterns. Datasets are semantically immutable but can be physically changed. You can add an index or two. It is OK to denormalize tables to optimize for read access. Datasets can be partitioned and the pieces placed close to their readers. A column-oriented representation of a dataset may also make sense.

You can make a copy of a table with far fewer columns to optimize for quick access (a skinny table). The column values can be left in both the skinny table and fat table.

By watching and monitoring the read usage of a dataset, you may realize new optimizations (for example, new indices) are possible.

Immutability is the backbone of big data. Massively parallel computations are based on immutable inputs and functional calculations. MapReduce[3] and Dryad[9] both take immutable files as input. The work is cut into pieces, each with immutable input. This functional calculation (using immutable inputs) is idempotent, making it possible to fail and restart. Immutability is the backbone of big data. MapReduce performs functional computations over immutable data to create immutable outputs. Failure and restart, so essential to reliable big data, are based on the idempotent nature of functional computation over immutable inputs.

Immutability as a semantic prism. Datasets show an immutable semantic prism, even if the underlying representation is augmented or completely replaced. The King James Bible is, character for character, immutable—even when it is printed in a different font; even when digitized; even when accompanied by different pictures.

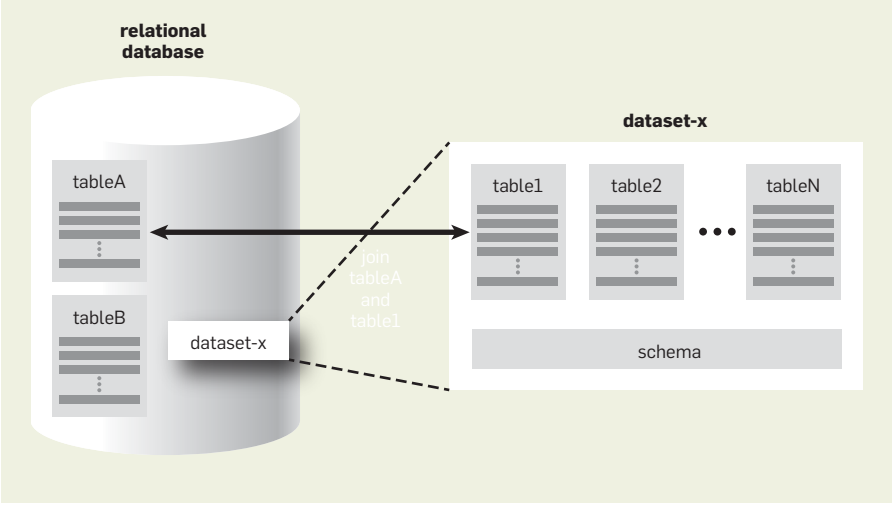
Is a dataset changed if there is a lossless transformation to a new schema representation? Can the new address field have more capacity? Can the enum values be mapped to a new underlying representation? Can the data be mapped from UTF-8 to UTF-16 encoding?

Having the right bits is not enough. You have to know how to interpret them. For example, "President Bush" had a different meaning in 1990 than in 2005. The word "napkin" is interpreted differently in the U.S. and the U.K.

Descriptive metadata when immutable. When an immutable dataset is created, the semantics of the data may not be changed. The contents may only be described as they are at the time the dataset is created.

Most programmers are used to SQL DDL (Data Definition Language) sup-

Figure 4. Immutable dataset may be joined with relational data.



porting dynamic changes in the metadata for their tables. This happens at a transaction boundary and can prescribe a new schema for the existing data. SQL DDL can be thought of as *prescriptive metadata* since it is prescribing the representation (which may change). Immutable datasets have *descriptive metadata* that explains what is there.

Of course, it is possible to create a new dataset that refers to one or more existing datasets in order to create a new representation of their data. Each new dataset has a unique ID. There is nothing wrong with having a dataset implemented by reference and not by value.

Normalization is for sissies. The goal of normalization is to eliminate update anomalies. When the data is not stored in a normalized fashion, updates might yield unpleasant results. The classic example is an imperfectly normalized table in which each employee has his or her manager's name and phone number. This makes it very difficult to update the manager's phone number since it is stored in many places. Normalization is very important in a database designed for updating.

Normalization is not necessary in an immutable dataset, however. The only reason to normalize immutable datasets may be to reduce the storage necessary for them. On the other hand, denormalized datasets may be easier and faster to process as inputs to a computation.

Versions Are Immutable, Too!

Each version is immutable. This section looks first at multiversion concurrency control; then techniques such as LSM that provide a semantic of change within a transactional space while generating immutable data that describes the state of these changes; finally, it looks at the world through the lens of COW, in which high-performance updates are implemented by writing new immutable data.

Versions and history. Versions should have immutable names. Other than the first version of something, a new version captures a replacement for or an augmentation of an earlier version. A *linear version history* is sometimes referred to as being strongly consistent: one version replaces another; there is one parent and one child; each version is immutable; each version has an identity. The alternative to linear version history is a DAG (directed acyclic graph) of version history, in which there are many parents and/or many children. This is sometimes called eventual consistency.

Multiversion concurrency control. Strongly consistent, or ACID (atomicity, consistency, isolation, durability), transactions appear as if they run in a serial order. This is sometimes called serializability.[2]

The database changes version by version. Transaction T1 is a version and later transaction T2 is a version. Transactions layer new versions of record and index changes atop earlier versions. The new versions can be captured as snapshots of the entire database (although this would not result in high performance).

Alternatively, the new version can be captured as changes to the previous version. In this way, a key-value store can be built, and a relational database can be built atop a key-value store. Records are deleted by adding tombstones. Changing the database is done by adding new records to the key-value store.

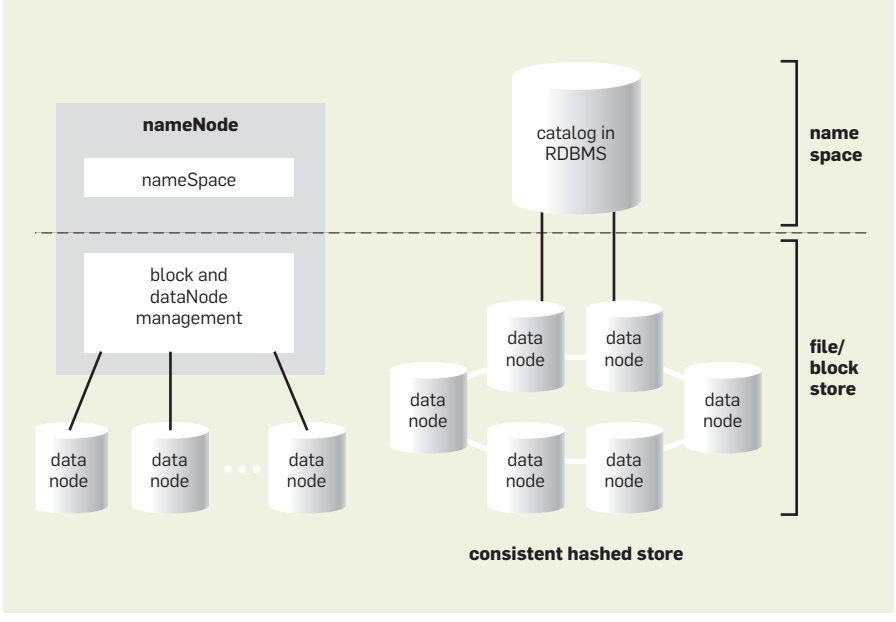
If a timestamp is added to each new version, it is possible to show the state of the database at a given point in time. This allows the user to navigate the state of the database to any older version. Ongoing work can see a stable snapshot of a version of the database.

LSM: Reorganizing immutable stuff. LSM presents a façade of change atop immutable files. With an LSM tree,[15] changes to the key-value store are accomplished by writing new versions of the affected records. These new versions are logged to an immutable file. Periodically, the new versions of the key values are sorted by key and written to an immutable file known as a Level 0 file within the LSM tree. Level 0 files are merged into a collection of Level 1 files (typically 10 Level 1 files, each containing one-tenth of the key range). Similarly, Level 1 files are merged with Level 2 files on a 10-to-1 basis. As you move down the LSM tree, each level has 10 times as many files. Reading a record typically involves searching one file per level. As the LSM files merge, new immutable files with new identities can be written.

Go ahead ... have a COW! An LSM tree can create changeable data out of immutable files by performing a COW. The granularity of the copy is typically a key-value pair. For a relational database, this can be a key-value pair for each record or each index entry. The changes are copied into the log and then into the LSM tree (and copied a few more times for merges).

High-performance COW happens with logging and classic DBMS performance techniques. The new versions are captured in memory and logged for failure recovery. The identity of each log file is a unique ID, and the log files are immutable. Each new log file can record the history of its preceding log files and

Figure 5. Immutable blocks over a consistent-hashing store.



even the identity of upcoming log files. Having one of the recent log-file IDs means the entire LSM key-value store can be reconstructed.

Keeping the Stone Tablets Safe

Many file systems keep immutable files consisting of immutable blocks. This section explores at a high level the implementation of GFS and HDFS and the implications of what can be done with these files. It discusses the vagaries of files that can be renamed and considers the value of storing immutable data within a consistent hash store.

Log-structured files: Running in circles. An early example of reifying change through immutability is the log-structured file system.[16] In this wonderful invention, file-system writes are always appended to the end of a circular buffer. Occasionally, enough metadata to reconstruct the file system is added to the circular buffer. Old data must be copied forward so it is not overwritten.

Log-structured file systems have some interesting performance characteristics, both good and bad. Today they are an important technique. As technology trends continue to move in the direction of recent years, they will become even more important.

Files, blocks, and replication. GFS,[5] HDFS,[1] and others offer highly available files. Each file is a bunch of blocks (also called chunks). The file consists of a file name and a description of the blocks needed to provide a bytestream. Each block is replicated in the cluster for durability and high availability. They are typically replicated three times over different fault zones in the data center.

Each file is immutable and (typically) single-writer. The file is created, and one process can append to it. The file lives for a while and is eventually deleted. Multiwriters are difficult, and GFS had some challenges with this.[13]

Immutable files and immutable blocks empower this replication. The file system has no concept of a change to a complete file. Each block's immutability allows it to be easily replicated without any update anomalies because it does not get updated.

Widely sharing immutable files is safe. An immutable file has an identity and contents, neither of which can change. You can copy an immutable file whenever and wherever you want and share the immutable copies across users. As long as you manage reference counts (so you know when it is OK to delete it), you can use one copy of the file to share across many users. You can distribute immutable files wherever you want. With the same identity and same contents, the files are location independent.

Names and immutability ... A slippery slope. GFS and HDFS both provide immutable files. Immutable blocks (chunks) are replicated across *data nodes*. Immutable files are a sequence of blocks, each of which is identified with a GUID (globally unique identifier). The contents of a file are immutable and labeled with a GUID. The file-ID GUID always refers to exactly one file and its contents.

GFS and HDFS also provide a namespace that can be changed. The logical name of an immutable file may be changed. File names may be rebound to different contents. Users must take great care to ensure they have predictable results when changing file names. Is something really immutable when its name can change?

Immutable data and consistent hashing. Consider a strongly consistent file system in which a single master is controlling a namespace (perhaps a Posix-style namespace). Looking up a file results in a GUID that is used to find an immutable bytestream.

Now consider a store implemented with consistent hashing.[10] It is well understood that consistent hashing offers very robust rebalancing under failures and/or additional capacity. It also has somewhat chaotic placement behavior while the ring is adjusting to changes. At times, some participants have seen the changes and others have not. When reading and updating within a consistent-hashing key-value store, the read occasionally yields an older version of the value. To cope with this, the application must be designed to make the data eventually consistent.[4] This is a burden and makes application development more difficult.

When storing immutable data within a consistent-hashing ring, you cannot get stale versions of the data. Each block stored has the only version it will ever have. This provides the advantages of a self-managing and master-less file store while avoiding the anomalies and challenges of eventual consistency as seen by the application (Figure 5).

Using an eventually consistent store to hold immutable data also means log writes can have more predictable SLAs (service-level agreements) by allowing the replicas to land in less predictable locations in the cluster. In a distributed cluster, you can know *where* you are writing or you can know *when* the write will complete but not both.[8] By preallocating files from the strongly consistent catalog, log writes using the file IDs need only to touch weakly consistent servers to be able to retry getting the blocks durable in a bounded time.

Immutability and decentralized recovery. Separating the namespace from block-placement control has a number of advantages. The consistent-hashing ring can take writes and reads even when the ring is in flux.

Although the catalog is a central point for access, it does not have the same varying load a name node does when handling failures in the cluster. The larger the cluster, the more data nodes will fail, each necessitating many controlling operations to elevate the replica count back to three. While this traffic happens, operations to read and write from the cluster will experience SLA variation. Immutability allows decentralized recovery of data-node failures with more predictable SLAs.

Hardware Changes Toward Unchanging

The trend toward leveraging immutability in new designs is so pervasive it can be seen in a number of hardware areas. Here, I examine the implementation of SSDs and some new trends in hard disks.

SSDS and wear leveling. The flash chip within most SSDs is broken into physical blocks, each of which has a finite number of times it may be written before it begins to wear out and give increasingly unreliable results. Consequently, chip designers have a feature known as wear leveling[12] to mitigate this aspect of flash. Wear leveling is a form of COW and treats each version of the block as an immutable version.

Each new block or update to a block in the logical address space of the flash chip is mapped to a different physical block. Each new write (or update to a new block) is written to a different phys-

ical block in a circular fashion, evening out the writes so each physical block is written about as often as the others.

Hard disks: Getting the shingles. As hard-disk manufacturers strive to increase the areal density of the data on disk, some physical headaches have intervened. Current designs have a much larger write track than read track. Writes overlap the previous ones in a fashion evocative of laying shingles on a roof—hence the name *shingled disk systems*.[6]

In shingled disks, a large band of data is written as layered write tracks forming a shingle pattern, partially overwriting the preceding tracks. The data in the middle of the band cannot be overwritten without trashing the remaining part of the band.

To overcome this, the hardware disk controllers implement log-structured file systems within the disk controller.[14] The operating system is unaware of the use of shingles. What is written to the disk (that is, the band of data written with shingles) remains unchanged until it is discarded. The user of the disk (for example, the operating system) perceives the ability to update in place.

Immutability May Have Some Dark Sides

As immutability is leveraged in all these ways, there are trade-offs to be managed. Denormalized documents help with read performance at the expense of extra storage cost. Data is copied many times with COW. This is exacerbated when these mechanisms are layered.

Denormalization: Nimble but fat. Denormalization consumes storage as a data item is copied multiple times in a dataset. It is good in that it eliminates JOINs to put the data together, making the use of the data more efficient. Immutable data has more choices for its representation. It can be normalized for space optimization or denormalized for read usage.

Write amplification vs. read perspiration. Data may be copied many times with COW (for example, with log-structured file systems, log-structured merge systems, wear leveling in SSDs, and shingle management in HDD). This is known as *write amplification*.[18]

In many cases, there is a relationship between the amount of write amplification and the difficulty involved in reading the data being managed. For example, some LSM systems will do more or less copying as the data is reorganized and merged. If the data is aggressively merged and reorganized, then fewer places need checking to read a record. This can reduce the cost of reading at the expense of additional writing.

Conclusion

Designs are driving toward immutability, which is needed to coordinate at ever increasing distances. Given space to store data for a long time, immutability is affordable. Versioning provides a changing view, while the underlying data is expressed with new contents bound to a unique identifier.

▸ *Copy-on-write.* Many emerging systems leverage COW semantics to provide a façade of change while writing immutable files to an underlying store. In turn, the underlying store offers robustness and scalability because it is storing immutable files. For example, many key-value systems are implemented with LSM trees (for example, HBase, BigTable, and LevelDB).

▸ *Clean replication.* When data is immutable and has a unique identifier, many challenges with replication are eased. There is never a worry about finding a stale version of the data because no stale versions exist. Consequently, the replication system may be more fluid and less picky about where it allows a replica to land. There are also fewer replication bugs.

▸ *Immutable datasets.* Immutable datasets can be combined by reference with transactional database data and offer clean semantics when the dataset project relational schema and tables. Looking at the semantics projected by an immutable dataset, you can create a new version of it optimized for a different usage pattern but still projecting the same semantics. Projections, redundant copies, denormalization, indexing, and column stores are all examples of optimizing immutable data while preserving its semantics.

▸ *Parallelism and fault tolerance.* Immutability and functional computation are keys to implementing big data. C

Related articles on queue.acm.org

If You Have Too Much Data, then "Good Enough" Is Good Enough
Pat Helland
http://queue.acm.org/detail.cfm?id=1988603

Enhanced Debugging with Traces
Peter Phillips
http://queue.acm.org/detail.cfm?id=1753170

Condos and Clouds
Pat Helland
http://queue.acm.org/detail.cfm?id=2398392

References

1. Apache Hadoop; http://en.wikipedia.org/wiki/Apache_Hadoop.
2. Bernstein, P., Hadzilacos, V. and Goodman, N. *Concurrency Control and Recovery in Database Systems.* Addison Wesley, 1987.
3. Dean, J. and Ghemawat, S. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th Annual Symposium on Operating System Design and Implementation*, 2004.
4. DeCandia, G. et al. Dynamo: Amazon's highly available key-value store. In *Proceedings of the 21st Annual ACM Symposium on Operating Systems Principles*, 2007.
5. Ghemawat, S., Gobioff, H. and Leung, S. The Google File System. In *Proceedings of the 19th Annual ACM Symposium on Operating Systems Principle*, 2003.
6. Gibson, G. and Ganger, G. Principles of operation for shingled disk devices. *Carnegie Mellon University Parallel Data Lab Technical Report CMU-PDL-11-107*, 2011.
7. Helland, P. Data on the outside versus data on the inside. In *Proceedings of the Conference on Innovative Database Research*, 2005.
8. Helland, P. Heisenberg was on the write track. *Abstract: Proceedings of the Conference on Innovative Database Research*, 2014.
9. Isard, M., Budiu, M., Yu, Y., Birrell, A. and Fetterly, D. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the European Conference on Computer Systems*, 2007.
10. Karger, D., Lehman, E., Leighton, T., Panigraphy, R., Levine, M. and Lewin, D. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, 1997.
11. Lamport, L. The part-time parliament. *ACM Transactions on Computer Systems 16*, 2 (1998), 133-169.
12. Lofgren, K., Normal, R., Thelin, G. and Gupta, A. Wear-leveling techniques for flash EEPROM systems. US Patent #6850443, 2003, SanDisk, Western Digital.
13. McKusick, M. and Quinlan, S. GFS: Evolution on fast forward. *ACM Queue 7*, 7 (2009).
14. New, R. and Williams, M. Log-structured file system for disk drives with shingled writing. US Patent #7996645, 2003, Hitachi.
15. O'Neil, P., Cheng, E., Gawlick, D. and O'Neil, E. The log-structured merge-tree (LSM-tree). *Acta Informatica 33*, 4 (1996).
16. Rosenblum, M. and Ousterhout, J. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems 10*, 1 (1992), 26–52.
17. Wikipedia. Turtles all the way down; http://en.wikipedia.org/wiki/Turtles_all_the_way_down.
18. Wikipedia. Write amplification; http://en.wikipedia.org/wiki/Write_amplification.

Pat Helland has been implementing transaction systems, databases, application platforms, distributed systems, fault tolerant systems, and messaging systems since 1978. He currently works at Salesforce.