Immutability Changes Everything

Pat Helland
Salesforce.com
One Market Street, #300
San Francisco, CA 94105 USA
01(415) 546-5881
phelland@salesforce.com

ABSTRACT

There is an inexorable trend towards storing and sending immutable data. We *need immutability* to coordinate at a distance and we *can afford immutability*, as storage gets cheaper.

This paper is simply an amuse-bouche on the 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.

1. INTRODUCTION

It wasn't that long ago that computation was expensive, disk storage was expensive, DRAM 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 SSD, while coordination with latches gets harder because latch latency loses lots of instruction opportunities.

We can now afford to keep immutable copies of lots of data, and one payoff is reduced coordination challenges.

1.1 More Storage, Distribution, & Ambiguity

We have *increasing storage* as the cost per terabyte of disk keeps dropping. This means we can keep lots of data for a long time.

We have *increasing distribution* as more and more data and work are spread across a great distance. Data within a datacenter seems "far away". Data within a many-core chip may seem "far away".

We have *increasing ambiguity* when trying to coordinate with systems that are far away… more stuff has happened since you've heard the news. Can you take action with incomplete knowledge? Can you wait for enough knowledge?

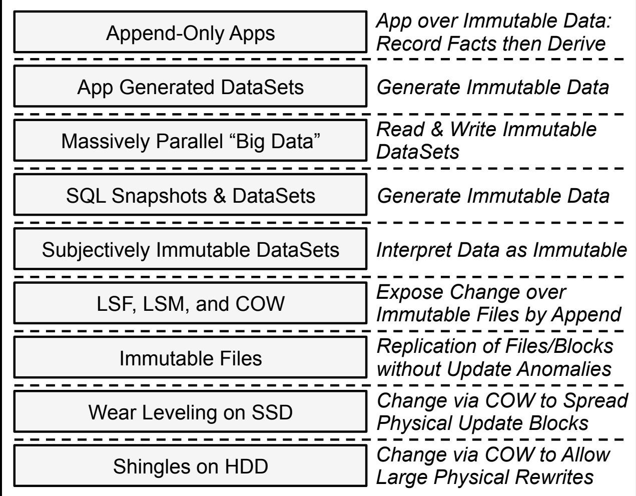
1.2 Turtles All the Way Down [17]

As various technological areas have evolved recently, they have responded to these trends of increasing storage, distribution, and ambiguity by using immutable data in some very fun ways. We will explore how apps use immutability in their ongoing work, how apps generate immutable DataSets for later offline analysis, how SQL can expose and process immutable snapshots, how massively parallel "Big Data" work relies on immutable DataSets. This leads us to looking at the ways in which semantically immutable DataSets may be altered while remaining immutable.

Next, we consider how updatability is layered atop the creation of new immutable files via techniques like LSF (Log Structure File systems), COW (Copy on Write), and LSM (Log Structured Merge trees). We examine how replicated and distributed file systems depend on immutability to eliminate anomalies.

7th Biennial Conference on Innovative Data Systems Research (CIDR '15) January 4-7, 2015, Asilomar, California, USA.

Next, we discuss how the hardware folks have joined the party by leveraging these tricks in SSD and HDD. See Figure 1. Finally, we look at some trade-offs with using immutable data.


Append-Only Apps
App over Immutable Data: Record Facts then Derive
App Generated DataSets
Generate Immutable Data
Massively Parallel "Big Data"
Read & Write Immutable DataSets
SQL Snapshots & 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 1. Immutability is a key architectural concept at many layers of the stack.

2. Accountants Don't Use Erasers

Lots of computing can be characterized as "append-only". This section looks at some of the ways this is commonly accomplished.

2.1 "Append-Only" Computing

May kinds of computing are "Append-Only". Observations are recorded forever (or for a long time). Derived results are calculated on demand (or periodically pre-calculated).

This is similar to a database management system. 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.

2.2 Accounting: Observed & Derived Facts

Accountants don't use erasers or they 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. Based on the observations, we can calculate something new. For example, amortized capital expenses based upon a rate and a cost. Another example is the current bank account balance with applied debits and credits.

2.3 Append-Only Distributed Single Master

Single master computing means somehow we order the changes. The order can come from a centralized master or some Paxos-like [11] distributed protocol providing serial ordering. Somehow, we semantically apply the changes one at a time.

Changes 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.

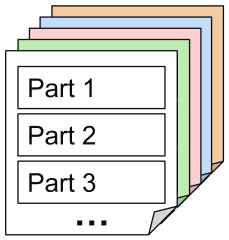
2.4 Distributed Computing "Back in the Day"

Before telephones, people used messengers. These were kids walking through town to deliver the message. Alternatively, the Postal Service delivered the messages. They took a long time…

Sometimes, people used fancy forms to capture the computing. The forms had many layers, each a different color. The forms had multiple sections on the page. A participant in the work filled out the next section (pressing hard with the pen). Then, he tore off the back page of the form and filed it. Each participant got the data they needed and added more data to the form. You can't update earlier sections, only append data to the end. See Figure 2.

Figure 2. Before computers, workflow was frequently captured in paper forms with multiple parts on the form and multiple pages.

"Fill out Part 3 and keep the goldenrod page from the back."



Distributed computing was append-only! New messages, new additions to the form… Each is a version and each is immutable. You were never allowed to overwrite what had been written.

3. Data on the Outside vs. Data on the Inside

Surprisingly (to us database old timers), not all data is kept in relational database systems. This section discusses some of the implications of unlocking data. This section is a subset of [7].

3.1 Data on the Inside

Data on the inside refers to stuff kept and managed by a classic relational database systems 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).

3.2 Data on the Outside

Data on the outside is prepared as messages, files, documents, and/or web pages. These are sent out from a service into the wild and cruel word. 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).
- **Maybe versioned:** Updates aren't updates but new versions with a new unique identifier.

3.3 Contrasting Inside vs. Outside

There are deep differences in the representation, meaning, and usage of inside data versus outside data. Increasingly, we are keeping data as 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*

4. Referencing Immutable Data

In this section, we introduce *DataSets* as a collection of data with a unique ID. Some DataSets have a structure that looks like a number of tables with schema. We consider how these DataSets can be referenced by a relational database and how relational operators may span both the DBMS and DataSet.

4.1 DataSets: Immutable Collections of Data

Let's define a **DataSet** as 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 it is deleted. DataSets may be relational or they may have some other representation such as a graph, a hierarchy (e.g. JSON), or any other representation. See Figure 3.

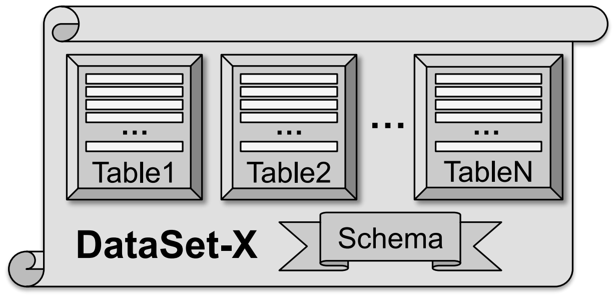


Figure 3: A DataSet is a logical set of immutable tables along with its schema

4.2 DataSets Referenced by a Relational DB

DataSets may be referenced by a relational DBMS. The metadata is visible to the DBMS. The data can be accessed for 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's no need for locking and no worries about controlling updates.

4.3 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. It can happen on a "Big Data" map-reduce style system. In both cases, there is a still and unchanging collection of data. *When we have 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 or HDFS files.

There's no semantic obstacle to doing JOINs across data stored inside a relational database and data stored in external DataSets. Locking (or snapshot isolation) provides a version of the relational database, which may be joined. Named and frozen DataSets may be joined with relational data. See Figure 4.

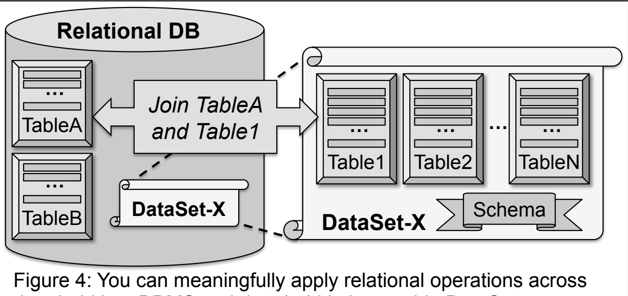


Figure 4: You can meaningfully apply relational operations across data held in a DBMS and data held in immutable DataSets.

In some ways, the ability to work across immutable DataSets and relational databases is surprising. Immutable DataSets are defined with an identity and an optional version. Their schema is the schema that describes the shape and form of the DataSet at the time of its creation. It has descriptive schema rather than the prescriptive schema held in the RDBMS.

This tailoring of the schema to meld the two together 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.

5. Immutability Is in the Eye of the Beholder

In this section, we discuss the ways in which a consumer may see DataSets as immutable even if they change under the covers.

5.1 DataSets Are Semantically Immutable

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

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

What's important about a DataSet is that it appears to be unchanging from the standpoint of the reader.

5.2 Optimizing DataSets for Read Patterns

DataSets are semantically immutable but may be physically changed. You can add an index or two. It's OK to denormalize tables to optimize for read access. DataSets may be partitioned and the pieces placed close to their readers. A column-oriented representation of a DataSet may make a lot of sense, too!

You can make a copy of a table with far fewer columns to optimize for quick access (a skinny table). It's OK to leave the column values in two places, both the skinny table and fat table.

By watching and monitoring the read usage of a DataSet, you may realize new optimizations (e.g. new indices) are possible.

5.3 Immutability and "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. It is OK to fail and restart.

Immutability is the Backbone of "Big Data"

Functional Computation with Immutable Inputs

Failure and Restart Are Based on the Idempotent Nature of Functional Computation over Immutable Inputs

5.4 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… Hmm…

Is a DataSet changed if there is a loss-less transformation to a new schema representation? If the new address field has more capacity, is that OK? If the ENUM values are mapped to a new underlying representation, is that OK? Can we map the data from UTF-8 to the UTF-16 encoding?

It's Not Enough to Have the Right Bits!

You Have to Know How to Interpret Them…

"President Bush" meant a different thing in 1990 versus 2005

The word "Fanny" is interpreted differently in the US vs Australia

You need to know what the Immutable Bits Actually Mean!

5.5 Descriptive Meta-Data when Immutable

Most of us are used to SQL DDL supporting dynamic changes in the metadata for our tables. This happens at a transaction boundary and can prescribe a new schema for the existing data.

When creating an immutable DataSet, the semantics of the data may not be changed. All we can do is describe the contents the way they are at the time the DataSet is created.

SQL DDL can be thought of as *prescriptive meta-data* since it is prescribing the representation (which may change). Immutable DataSets have *descriptive meta-data* that explains what's there.

Of course, you can create new DataSets that refer to one or many other pre-existing DataSets to create a new representation of their data. The new DataSets each have their own unique ID.

There's nothing wrong with having a DataSet implemented by reference and not by value.

5.6 Normalization Is for Sissies

Normalization's goal 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 their manager's name and phone number. This makes it very hard to update the manager's phone number since its stored in lots of places. Normalization is very important in a database designed for update.

Normalization is not necessary in an immutable DataSet.

The only reason to normalize immutable DataSets may be to reduce the storage necessary for them. On the other hand, denormalized DataSets may be easy and faster to process as inputs to a computation.

6. Hey! Versions Are Immutable, Too!

In this section, we consider the use of versions, each of which is immutable. First, we look at multi-version concurrency control. Then, we see how techniques like LSM (Log Structured Merge Trees) provide a semantic of change within a transactional space while generating immutable data that describes the state of these changes. Finally, we explore the world through the lens of copy-on-write in which high-performance updates are implemented by writing new immutable stuff.

6.1 Versions and History

Other than the first version of something, a new version captures a replacement for or an augmentation to an earlier version.

A linear version history is sometimes referred to as being strongly consistent. One version replaces another. There's one parent and one child. Each version is immutable. Each version has an identity. Typically, each version is viewed as a replacement for the earlier versions.

Alternatively, you may have a DAG or *directed acyclic graph of version history*. With a DAG you may have many parents and/or many children. This is sometimes called eventual consistency.

Versions are immutable and should have immutable names.

6.2 Multi-Version Concurrency Control

Strongly consistent (ACID) transactions appear as if they run in a serial order. This is sometimes called serializability [2].

The DB changes version by version.

Transaction T1 is a version. Later, transaction T2 is a version.

Everything changeable can be understood as a bunch of versions.

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 that wouldn't perform too well).

Alternatively, you can capture the new version as changes to the previous version. You can build a key-value store this way. You can build a relational database atop a key-value store. Records are deleted by adding tombstones.

Adding new values to the key-value store changes the database.

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

6.3 LSM: Reorganizing Immutable Stuff

With an LSM (Log Structured Merge 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 out 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 1/10th of the key range). Similarly, Level-1 LSM files are merged with Level-2 LSM 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 we merge the LSM files, we read immutable files and write brand new immutable files with new identities.

LSM presents a facade of change atop immutable files.

6.4 Go Ahead! Have a COW!

How does LSM make changeable stuff out of immutable files? Basically, it performs a COW or copy-on-write. 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 copied into the LSM tree (and copied a few more times for merges).

High-performance copy-on-write 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 even the identity of upcoming log files. If you have one of the recent log-file-IDs, the entire LSM key-value store can be reconstructed.

Other than keeping the starting point for the log somewhere, all the information describing the database state can be kept in immutable files.

7. Keeping the Stone Tablets Safe

Many file systems keep immutable files comprising immutable blocks. This section explore at a high level the implementation of GFS and HDFS and the implications of what can be done with these files. We discuss the vagaries of files that can be renamed. Finally, we consider the value of storing immutable data within a consistent hash store.

7.1 Log Structured Files: Running in Circles!

An early example of reifying change through immutability is Log Structured File Systems [16]. In this wonderful invention, file system writes are always appended to the end of a circular buffer. Occasionally, enough meta-data to reconstruct the file system is added to the circular buffer. Old stuff must be copied forward so it is not overwritten.

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

7.2 Files, Blocks, & Replication

GFS [5], HDFS [1], and others offer highly available files. Each file is a bunch of blocks (or chunks). The file comprises a file name and a description of the blocks needed to provide a byte stream. Each block (called a chunk in GFS) 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. Multi-writers are hard and GFS had some challenges with this as explained in [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 doesn't get updated!

High availability of immutable blocks is available now!

Google, Amazon, Facebook, Yahoo, Microsoft, and more keep petabytes and exabytes of immutable data!

7.3 Widely Sharing Immutable Files Is Safe

Immutable files have an identity and content.

Neither the identity nor the content can change!

You can copy an immutable file whenever and wherever you want. You can share the immutable copies across users. As long as you manage reference counts (so you know when it's 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! See Figure 5.

Published Books are Immutable!

Sometimes later editions repair previous bugs

This is versioning of the book

Versions are immutable objects!

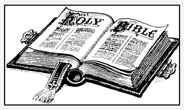


Figure 5

7.4 Names & Immutability… a Slippery Slope

GFS (Google File System) and HDFS (Hadoop Distributed File System) provide immutable files. Immutable blocks (chunks) are replicated across _data nodes_. Immutable files are a sequence of blocks (chunks) each of which is identified with a GUID. 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 to something else. 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?_* What's a name, anyway?

7.5 Immutable Data and Consistent Hashing

Consider a strongly consistent file system. There's a single master controlling a namespace (perhaps a Posix-style namespace). Looking up a file results in a GUID that is used to find an immutable byte stream.

Let's consider a store implemented with a consistent hashing [10]. It's 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. There are times when 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], which is a burden and makes application development more difficult.

When storing immutable data within a consistent hash 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. See Figure 6.

Using an eventually consistent store to hold immutable data also means that log-writes can have more predictable SLAs 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 pre-allocating files from the strongly consistent catalog, log writes using the File-IDs only need to touch weakly consistent servers and can retry to get the blocks durable in a bounded time.

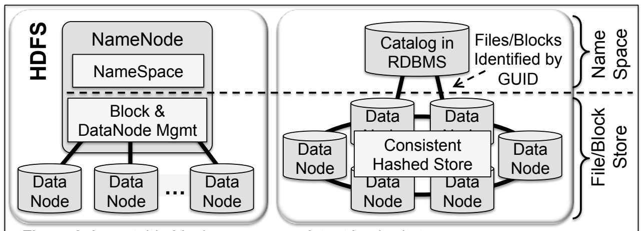


Figure 6: Immutable blocks over a consistent hashed store

7.6 Immutability and Decentralized Recovery

By separating the namespace from block placement control, there are a number of advantages. The consistent hashing ring can take writes and reads even when the ring is under flux.

While the catalog is a central point for access, it does not have the same varying load that 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.

8. Hardware Changes towards Unchanging

The trend to leverage immutability in new designs is so pervasive we see it in a number of hardware areas. We first examine the implementation of SSDs, then some new trends in hard disks.

8.1 SSDs and Wear Leveling

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

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

Wear leveling is a form of copy-on-write and treats each version of the block as an immutable version.

8.2 Hard Disks: Getting the Shingles

As hard disk manufactures strive to get the areal density of the data on disk higher, 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, there is a large band of data that 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's written to the disk (i.e. the band of data written with shingles) remains unchanging until it is discarded. The user of the disk (e.g. the operating system) perceives the ability to update in place.

9. Immutability May Have Some Dark Sides

As we leverage immutability in all these ways, there are tradeoffs to be managed. We see denormalized documents as helping with read performance at the expense of extra storage cost. Data is copied many times when we use copy on write. This is exacerbated when we layer these mechanisms.

9.1 Denormalization: Nimble but Fat

Denormalization consumes storage as a data item is copied multiple times in a DataSet. It's 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. We can normalize for space optimization or denormalize for read usage.

9.2 Write Amplification vs. Read Perspiration

Data may be copied many times when we use copy on write (e.g. 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 (Log Structured Merge tree) systems will do more or less copying as the data is reorganized and merged. If the data is aggressively merged and reorganized, there are fewer places that need checking to read a record. This can reduce the cost of reading at the expense of additional writing.

10. Conclusion

Designs are driving towards immutability. We need immutability to coordinate at ever increasing distances. We can afford immutability given room to store data for a long time. Versioning gives us a changing view of things while the underlying data is expressed with new contents bound to a unique identifier.

Copy-on-Write: Many emerging systems leverage copy-on-write 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 instance, there are many key-value systems implemented with log-structured merge trees (e.g. HBase, BigTable, & LevelDB).

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

Immutable DataSets: Immutable DataSets can be combined by reference with transactional database data and offer clean semantics when the DataSets project relational schema and tables. We can look at the semantics projected by an immutable DataSet and 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 the key to implementing "Big Data".

Immutability does change everything!

11. References

[1] http://en.wikipedia.org/wiki/Apache_Hadoop

[2] Bernstein, P.; Hadzilacos, V.; Goodman, N. (1987). "Concurrency Control and Recovery in Database Systems", *Addison Wesley, ISBN 0-201-10715-5*.

[3] Dean, J.; Ghemawat, S. (2004). "MapReduce; Simplified Data Processing on Large Clusters". OSDI '04: 6th Symposium on Operating System Design & Implementation.

[4] DeCandia, G.; Hastorun, D.; Jampani, M.; Kakulapati, G. Lakshman, A.; Pilchin, A.; Sivasubramanian, S.; Vosshall, P. Vogels, W. (2007). "Dynamo: Amazon's Highly Available Key-Value Store". *Proc of the 21st ACM Symp on Operating Systems Principles.*

[5] Ghemawat, S.; Gobioff, H.; Leung, S. (2003) "The Google File System". *Proceeedings of the 19th ACM Symposium on Operating Systems Principles – SOSP '03*

[6] Gibson, G.; Ganger, G. (2011) "Principles of Operation for Shingled Disk Devices". *Carnegie Mellon University Parallel Data Lab Technical Report CMU-PDL-11-107.*

[7] Helland, P. (2005) "Data on the Outside versus Data on the Inside" *Proceedings of the 2005 CIDR Conference (Conference on Innovative Database Research).*

[8] Helland, P. (2014) "Heisenberg Was on the Write Track". *Abstract: Proceedings of the 2015 CIDR Conference (Conference on Innovative Database Research).*

[9] Isard, M.; Budiu, M.; Yu, Y.; Birrell, A.; Fetterly, D. (2007) "Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks" *European Conf on Computer Systems (EuroSys).*

[10] Karger, D.; Lehman, E.; Leighton, T.; Panigraphy, R.; Levine, M.; Lewin, D. (1997). "Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web". *Proc. of the 29th Annual ACM Symp on Theory of Computing.*

[11] Lamport, L. (1998). "The Part-Time Parliament", *ACM Transactions on Computer Systems (TOCS), Volume 16, Issue 2, May 1998.*

[12] Lofgren, K.; Normal, R.; Thelin, G.; Gupta, A.; (2003). "Wear leveling techniques for flash EEPROM systems". *US Patent # 6850443 (SanDisk, Western Digital).*

[13] McKusick, M.; Quinlan, S.; "GFS: Evolution on Fast Forward" (2009) ACM Queue, August 7, 2009.

[14] New, R.; Williams, M.; (2003). "Log-structured file system for disk drives with shingled writing". *US Patent # 7996645 (Hitachi).*

[15] O'Neil, P; Cheng, E.; Gawlick, D.; O'Neil, E. (1996) "The Log-Structured Merge-Tree (LSM-tree)". *Acta Informatica 33 (4).*

[16] Rosenblum, M.; Ousterhout, J. (1992) "The Design and Implementation of a Log-Structured File System". *ACM Transactions on Computer Systems, Vol. 10, Issue 1.*

[17] http://en.wikipedia.org/wiki/Turtles_all_the_way_down

[18] http://en.wikipedia.org/wiki/Write_amplification