Mind Your State *for* Your State of Mind

**THE INTERACTIONS
BETWEEN
STORAGE AND
APPLICATIONS CAN
BE COMPLEX AND
SUBTLE.**

PAT HELLAND

Applications have had an interesting evolution as they have moved into the distributed and scalable world. Similarly, storage and its cousin databases have changed side by side with applications. Many times, the semantics, performance, and failure models of storage and applications do a subtle dance as they change in support of changing business requirements and environmental challenges. Adding scale to the mix has really stirred things up. This article looks at some of these issues and their impact on systems.

Before database transactions, there were complexities in updating data, especially if failures happened. This held true even though the systems were centralized and avoided the complexities presented by distribution. Database transactions dramatically simplified the life of

application developers. It was great while it lasted...

As solutions scaled beyond a single database, life got ever more challenging. First, we tried to make multiple databases look like one database. Then, we were hooking multiple applications together using SOA (service-oriented architecture). In SOA, each service had its own discrete database with its own transactions but used messaging to coordinate across boundaries. Soon, we were using microservices, each of which likely did not have its own data but reached directly to a distributed store shared across many separate services. This scaled better—if you got the implementation right.

Different types of distributed stores offer various average speeds, variation in responsiveness, capacity, availability, and durability. Diverse application patterns use the stored data for distinct purposes. They provide various guarantees to their users based largely on their use of storage. These different guarantees from the app sometimes show variations in what the users see in semantics, response time, durability, and more. While these can be surprising, it may be OK. What matters is the fulfillment of the business needs and clarity of expectations.

This article provides a partial taxonomy of diverse storage solutions available over a distributed cluster. Part of this is an exploration of the interactions among different features of a store. The article then considers how distinct application patterns have grown over time to leverage these stores and the business requirements they meet. This may have surprising implications.

THE EVOLUTION OF STATE, STORAGE, AND COMPUTING... AT LEAST SO FAR

This section starts by examining some of the profound changes that have occurred in both storage and computation. The focus then turns to a discussion of both durable state and session state and how they have evolved over time. Finally, there is a brief reminder of how data is treated differently inside a classic database and outside as it moves across trust and transactional boundaries.

Trends in storage and computing

Changes in storage and computing have put demands on how storage is accessed and the expected behavior in doing so. This is especially interesting as work is smeared over pools of small computation known as microservices.

Storage has evolved. It used to be that storage was only directly attached to your computer. Then came shared appliances such as SANs (storage area networks). These are big, expensive devices with a lot of sophisticated software and hardware to provide highly available storage to a bunch of servers attached to them. This led to storage clusters of commodity servers contained in a network.

Computing has evolved. A few decades ago, it was only a single process on a single server. Years went by before people started worrying about communicating across multiple processes on a single server. Then the world moved on with great excitement to RPCs (remote procedure calls) across a tiny cluster of servers. At the time, we didn't think about trust since everyone was in the same trust zone. We were all in the family!

In the 2000s, the concept of services or SOA began

to emerge, sometimes under different names.⁶ The basic aspect of a service is *trust isolation*. This naturally leads to applications and app code encapsulating the data so the distrusted outsider cannot just modify the data with abandon.

As the industry started running stuff at huge scale, it learned that busting a service into smaller microservices has a couple of big advantages:

- ➡ Better engineering. Breaking your services (i.e., trust boundaries) into smaller pieces allows better engineering flexibility as small teams make quicker changes.
- ➡ Better operability. Making these smaller pieces stateless and restartable allows for more resilient operations as failures, rolling upgrades of versions, and adjustments for varying demand are dynamically handled.

Microservices became an essential part of the software engineering and operations landscape.

Computing's use of storage has evolved. It has been quite a wild ride of application changes as their use of storage has evolved:

- ➡ Direct file I/O used careful replacement for recoverability. *Careful replacement* is a technique that is at least as old as the 1960s. It involves thoughtful ordering of changes to durable storage such that failures can be tolerated.

Careful Replacement Variations

- ➡ A write may trash the previous value... write somewhere else first.
 - ➡ A client crash may interrupt a sequence of writes... plan carefully.
-

➡ Transactional changes were supported for application developers, providing a huge improvement. It meant the app developer did not need to be so careful when dealing with storage. It also allowed a grouping of changes to records so a bunch of records were atomically updated. This was a *lot* easier. SANs implemented the required careful replacement for the hardware storage, allowing bigger and better databases. Databases evolved to support two-tier and N-tier applications using transactional updates.

➡ Key-value stores offered more scale but less declarative functionality for processing the application's data. Multirecord transactions were lost as scale was gained.

There have been and continue to be significant changes to the style of computation, to storage, and to how these application patterns are used to access storage.

This is only a partial list of storage and compute models. It is not meant to be complete.

Challenges in modern microservice-based applications

These days, microservices power many scalable apps. Microservices are pools of identical or equivalent services running over a collection of servers. Incoming requests are load balanced across the pool.

When a request waits for a microservice, any one from the same pool will do the job. Sometimes, systems implement *affinitization*, where a subsequent request is likely to go to the same specific microservice. Still, the outcome must be correct if you land on any of the microservices.

Microservices help scalable systems in two broad ways:

- ➔ *Improved software engineering.* Building systems consisting of small and independent microservices results in agility. Teams owning the microservices must be accountable and have independence and ownership. When something needs changing, change it. When something is broken, the owning team is responsible.
- ➔ *Improved operations.* Health-mediated deployment allows for slow rollout of new versions into the running system. By watching the system's health, new versions can be rolled back. These rolling upgrades to the microservices can be sensitive to fault zones so an independent failure during a flaky upgrade is not too damaging. Simply having a lot of separate and equivalent microservices means a failure of one or more of them is automatically repaired.

Durable state is not usually kept in microservices. Instead, it is kept in back-end databases, key-value stores, caches, or other things. The remainder of this article looks at some of these.

Microservices cannot easily update the state across all of the microservices in the pool. This is especially true when they are coming and going willy-nilly. It is common to keep the latest state out of reach of the microservices and provide older versions of the state that are accessible in a scalable cache. Sometimes, this leads to read-through requests by the scalable cache to durable state that is not directly addressable to the calling microservice.

This is now becoming a tried and true pattern. Figure 1 is taken from a 2007 paper by DeCandia et al. on Amazon's Dynamo.² While the nomenclature is slightly different, it shows three tiers of microservices accessing a back-end

FIGURE 1: **EXAMPLE OF AMAZON'S DYNAMO MICROSERVICE ARCHITECTURE**

tier of different stores.

Durable state and session state

Durable state is stuff that gets remembered across

requests and persists across failures. This may be captured as database data, file-system files, key values, and more. Durable state is updated in a number of different ways, largely dependent on the kind of store holding it. It may be changed by single updates to a key value or file, or it may be changed by a transaction or distributed transaction implemented by a database or other store.

Session state is the stuff that gets remembered across requests in a session but not across failures. Session state exists within the endpoints associated with the session. Multioperation transactions use a form of session state.⁷

Session state is hard to do when the session is smeared across service instances. If different microservices in the pool process subsequent messages in the transaction, session state is challenging to implement. It's hard to retain session state at the instance when the next message to the pool may land at a different service instance.

Data on the outside versus data on the inside

The 2005 paper “Data on the Outside Versus Data on the Inside”⁵ speaks about the fundamental differences between data kept in a locked transactional store (e.g., a relational database) and data kept in other representations.

Data on the inside refers to locked transactionally updated data. It lives in one place (e.g., a database) and at one time, the transactional point in time.

Data on the outside is unlocked and immutable, although it may be versioned with a sequence of versions that are in their own right immutable. Outside data always has some form of a unique identifier such as a URI (uniform resource identifier) or a key. The identifier may be implicit

within a session or an environment. Outside data typically is manifest as a message, file, or key-value pair.

THE EVOLUTION OF DURABLE STATE SEMANTICS

Storage systems and databases have evolved through the decades and so have the semantics of updating their state. This section begins in the bad old days when I first started building systems. Back in the '70s and '80s, disk storage had to be carefully updated to avoid trashing disk blocks. From there, we move forward to atomic record updates and the challenges that arose before transactions. When transactions came along a lot of things got a lot easier—if you were making a change at one place and one time. Adding cross-database and cross-time behavior led to the same challenges you had with more primitive storage systems. This was helped by using messaging subsystems to glue stuff together.

Then, an interesting development in storage occurred. Some stores are fast but sometimes return stale values. Others always return the latest value but occasionally stall when one of the servers is slow. This section shows how predictable answers result in unpredictable latencies.¹⁰ Finally, it examines the role immutable data can play in supporting very large systems with predictable answers and response times for some business functions.

Careful replacement of disk blocks

It used to be, back in the '70s and '80s, that a disk write might leave data unreadable. The write went through a number of state changes from the old V1 version, to unreadable garbage, to the new V2 version. When the disk

head was writing a block, the magnetic representation of the bits in the block would be turned to mush on the way to being updated to the new version. A power failure would cause you to lose the old value (see figure 2).

When implementing a reliable application, it's essential that you don't lose the old value of the data. For example, if you're implementing the transaction system for a database, it's really bad to lose the most recently committed transactions because the partially full last block of your transaction log is being rewritten. One trick to avoid this is to take turns writing to mirrored logs on different disks. Only after knowing for sure that mirror A has the new block do you write it to mirror B. After a crash, you rewrite the last block of the log onto both mirrors to ensure a consistent answer.

Another well-known technique, especially for the tail of the log, is called *ping-pong*.⁴ In this approach, the last (and incomplete) block of the log is left where it lies at the

FIGURE 2: **V1 IS TRASHED BEFORE V2 IS WRITTEN**

end of the log. The next version of that block, containing the previous contents and more, is written to a later block. Only after the extended contents are durable on the later block will the new version overwrite the earlier version. In this fashion, there are no windows in which a power failure will lose the contents of the log (see figure 3).

Careful replacement for record writes

Updates to records in pre-database days didn't have transactions. Assuming each record write was atomic, you still couldn't update two records and get any guarantees they would both be updated. Typically, you would write to record X, wait to know it's permanent, and then write to record Y.

FIGURE 3: "PING-PONG" TECHNIQUE DELAYS OVERWRITE OF V1

So, could you untangle the mess if a crash happened?

Frequently, there was an application-dependent pattern that provided insight into the order you needed to write. After a crash and restart:

- ➡ If record A was updated but record B was not written, the application can clean up the mess.
- ➡ If record B was updated but record A was not written, the application could not cope and could not recover.

An example of careful replacement for records is message queuing. If the application writes and confirms the presence of a message in a queue (call it record A), and the work to process that message is idempotent, then the application can cope with crashes based on careful replacement for records. Idempotent means it is correct if restarted.^{4,7}

Transactions and careful replacement

Transactions bundle and solve *careful record replacement*. Multiple application records may be updated in a single transaction, and they are all-or-nothing. The database system ensures that the record updates are atomic.

➡ *Databases automatically handle any challenges with careful storage replacement.* Users are not aware of the funky failure behaviors that may occur when systems crash or power fails. If present, databases also support distributed transactions over a small number of intimate database servers.

➡ *Work across time (i.e., workflow) needs careful transactional replacement.* While the set of records in a transaction is atomically updated with the help of the database, long-running workflows^{3,4} are essential to

accomplish correct work over time. Failures, restarts, and new work can advance the state of the application transaction by transaction. Work across time leverages message processing.

➡ *Work across space* (i.e., across boundaries) *also needs careful transactional replacement*. Different systems, applications, departments, and/or companies have separate trust boundaries and typically don't do transactions across them. Work across space necessitates work across time, transaction by transaction. This leads to messaging semantics.

Messaging semantics

In transactional messaging a transaction makes a bunch of changes to its data and then expresses a desire to send a message. This desire is atomically recorded with the transaction. A transaction may atomically consume an incoming message. That means the work of the transaction, including changes to the application data, occurs if, and only if, the incoming message is consumed.

It is possible to support the semantics of exactly-once delivery. The desire to send is atomically committed with the sending transaction. A committed desire to send a message causes one or more transmissions. The system retries until the destination acknowledges it has received the message in its queue. The message must be processed at the receiver at most once. This means it must be idempotently processed (see figure 4).

There are challenges with at-most-once processing at the destination. To accomplish this, you need to remember the messages you've processed so you don't process them

FIGURE 4: TRANSACTION MESSAGING

twice. But how do you remember the messages? You have to detect duplicates. How long do you remember? Does the destination split? Does the destination move? If you mess this up, will the application process the message more than once? What if the message is being delivered to a microservice-based application? Where is the knowledge of the set of processed messages kept?

Read your writes? Yes? No?

It used to be, back in the day, if you wrote something, you could read it. Now, it's not always that simple. Consider the following:

Linearizable stores offer read-your-write behavior. In a linearizable store each update creates a new version of the value, and the store never returns an old value or a different value. It always returns the latest in a linear series of values.

Linearizable stores will sometimes delay for a loooooong time.

To ensure they always give the correct value, they will always update every replica.

If a server is slow or dead and contains one of the replicas, it may take tens of seconds to decide what to do...

Meanwhile, the user waits.

Nonlinearizable stores do not offer read your writes. A nonlinearizable store means there's no guarantee that a write will update all the replicas. Sometimes, a read may find an old value. Reading and writing a nonlinearizable store has a very consistent response time with much higher probability. A read or write can skip over a sick or dead server. Occasionally, this results in an older value coming back from the skipped server. But, hey, it's fast—and predictably so.

Imagine a key/value store where key-K has value V1 and the store keeps it on servers S1, S2, and S3. You decide to update the value to V2. The store tries to change the values on its three servers, but S2 doesn't answer because it's down. Therefore, the store decides to write V2 onto S1, S3, and S4 so that the new value is always written to three servers. Later, when S2 comes up, a read might find the old value V1. This has the following tradeoffs:

- ➡ The write of three stores always happens quickly.
- ➡ The store is not linearizable and sometimes returns an old value.

This very useful technique underlies a number of scalable storage systems such as Dynamo² and Cassandra.¹¹

Cached data offers scalable read throughput with great response time. Key-value pairs live in many servers and are updated by propagating new versions. Each read hits one of the servers and returns one of the versions (see figure 5).

Different Stores for Different Uses

- OK to stall on reads?
- OK to stall on writes?
- OK to return stale versions?
- You can't have everything!

Immutability: A solid rock to stand on

When you store immutable data, each lookup always returns the same result.⁸ Immutable stores don't ever exhibit update anomalies because you never update them. All you can do is store a brand-new value for an identifier and, later on, delete it. Many application patterns are based on immutable items.

Imagine a system where you are simply recording stuff you've seen. Everything you know is based on observations. The past is never changed—sort of like an accountant's

FIGURE 5: DIFFERENT TYPES OF STORAGE OFFER DIFFERENT GUARANTEES

	FAST PREDICTABLE READS?	FAST PREDICTABLE WRITES?	READ YOUR WRITES?
Linearizable Store	no	no	yes
Non-Linearizable Store	yes	yes	no
Scalable Cache	yes w/scale	no	no

ledger where nothing is updated. You can put a unique ID on each artifact and look at it later but never change it. This is an extremely common pattern.

When keeping immutable objects or values in a key/value store, you never get a stale answer. There's only one immutable value for the unique key. That means a nonlinearizable store offers the one and only correct answer. All the store types give the correct answer, just with different characteristics for read and write latencies (see figure 6). Storing immutable data means you never get a stale version because there isn't one.

SLIP-SLIDIN' AWAY...

This section looks at a number of guarantees that are slipping away. Everyone wishes they had a computational model such as a von Neumann machine,¹² which provides computation, storage, and predictable linear behavior. Once distribution kicks in, however, that's indeed only a wish.

Single-process computation as John von Neumann conceived has evolved to multiprocess- and multiserver-using sessions and session state. These stateful sessions

FIGURE 6: IMMUTABLE DATA ALLOWS “READ YOUR WRITE BEHAVIOR”

	FAST PREDICTABLE READS?	FAST PREDICTABLE WRITES?	READ YOUR WRITES?
Linearizable Store	no	no	immutable
Non-Linearizable Store	yes	yes	immutable
Scalable Cache	yes w/scale	no	immutable

supported composable transactions that spanned multiple records and multiple servers working together. As the work started decomposing into microservices, however, it became hard to use transactions the way they had been used.

To cope with scalable environments, data had to be busted up into key values. Scalable stores worked well for updating a single key at a time but not for atomic transactions across keys. Most of these scalable key-value stores ensured linearizable, strongly consistent updates to their single keys. Unfortunately, these linearizable stores would occasionally cause delays seen by users. This led to the construction of nonlinearizable stores with the big advantage that they have excellent response times for reads and writes. In exchange, they sometimes give a reader an old value.

Finally, this section points out that some uses of data find the correct answer important enough to use careful replacement of the stored values. These uses are not the best for nonlinearizable stores.

Honestly, it ain't like it used to be.

Same process evolves to different process

Applications and the database used to run in the same process. A library call to the database code allowed access to the data. Sometimes, multiple applications were loaded together.

Later, the database and applications were split into different processes connected by a session. The session described the session state and had information about the user, transaction in flight, the application being run, and the cursor state and return values.

Later still, the application and database moved to different servers. The session state made that possible.

Stateful sessions and transactions

Stateful sessions were a natural outcome of shared processes. You knew who you were talking to and you could remember stuff about the other guy.

Stateful sessions worked well for classic SOA. When talking to a service, you expected a long session with state on each side. Stateful sessions meant the application could do multiple interactions within a transaction. In many circumstances, rich and complex transactions could occur over N-tier environments, even across multiple back-end databases using distributed transactions.

Transactions, sessions, and microservices

Microservices leave much to be desired when it comes to session state. Requests are load balanced through a router, and one of many microservice instances is selected. Usually, later traffic is sent to the same instance but not always. You can't count on getting back to where you were.

Without session state, you can't easily create transactions crossing requests. Typically, microservice environments support a transaction within a single request but not across multiple requests.

Furthermore, if a microservice accesses a scalable key-value store as it processes a single request, the scalable key-value store will usually support only atomic updates to a single key. While it won't break the data by failing in the middle of updating a key as older file systems did,

programmers are on their own when changing values tied to multiple keys.

Keys, versions, and nonlinear history

Each key is represented by some number, string, key, or URI. That key can reference something that's immutable. For example, "The *New York Times*, June 1st, 2018, San Francisco Bay Area edition" is immutable across space and time. A key may also reference something that changes over time—for example, "today's *New York Times*."

When a key references something that changes, it can be understood as referencing a sequence of versions, each of which is immutable. By first binding the changing value of the key to a unique version of the key (e.g., `[Key, Version-1]`), you can view the version as immutable data. Each version becomes an immutable thing to be kept. Using the extended `[Key, Version]`, you can reference immutable data in the store.

Version history may be linear, meaning one version supersedes the previous one. This is achieved by using *a linearizable store*. Version history may be a DAG (directed acyclic graph). This happens when writing to a *nonlinearizable store*.

Imagine you have a notepad on which to scribble stuff. But you really have multiple notepads. You scribble stuff on whichever notepad is closest to you at the time. When you want to read the information, you look at the closest notepad even if it's not the one you wrote on most recently. Sometimes, you get two notepads next to each other, look at both, and write something in both to consolidate the scribbles. This is the kind of behavior that comes from a

nonlinearizable store. Updates do not march forward in linear order.

Careful replacement and read your writes

In careful replacement you need to be careful about the ordering of what you update. This is essential to handle some failures, as discussed earlier. Predictable behavior across trust boundaries is needed when working with other companies. It’s also essential when doing long-running workflows.

Careful replacement is predicated on read-your-writes behavior, which depends on a linearizable store. Linearizable stores almost always have the property of occasionally stalling when waiting for a bum server.

SOME EXAMPLE APPLICATION PATTERNS

Let’s look at some application patterns and how they impact the management of durable state (figure 7).

FIGURE 7: APPLICATION PATTERNS

workflow over key-value	a traditional workflow application over a scalable collection of key-value data.
transactional blobs-by-ref	a centralized and transactional system managing very large collections of immutable blobs.
ecommerce—shopping cart	the familiar but still surprising world of ecommerce shopping carts.
ecommerce—product catalog	consider a very large ecommerce product catalog with enormous numbers of product descriptions and huge traffic reading the catalog.

Workflow over key-value with careful replacement

This pattern demonstrates how applications perform workflow when the durable state is too large to fit in a single database.

An object is uniquely identified by its key. Work arrives from the outside via human interaction or messaging. Workflow can be captured in the values. New values replace old ones. The messages are contained as data within the object.⁹

Scalable workflow applications can be built over key-

FIGURE 8: **LINEAR VS. NONLINEAR VERSION HISTORIES**

value stores. You must have single-item linearizability (read your writes). (See figure 8.) With a linear version history, one new version always supersedes the earlier one. A nonlinear history has a DAG version history. In this case, the linearizable behavior of the store also implies that a stall within one of the store servers will stall the write to the store. This is the “must be right” even if it’s not “right now” case.

The workflow implemented by careful replacement will be a mess if you can’t read the last value written. Hence, this usage pattern will *stall* and not be *stale*.

Transactional blobs-by-ref

This is a pretty common application pattern. The application runs using transactions and a relational database. It also stores big blobs such as documents, photos, PDFs, videos, music, and more. The blobs can be large and numerous. Hence, these are a challenge to implement directly in the relational database.

Each of these blobs is an immutable set of bits. To modify a blob (for example, editing a photo), you always create a new blob to replace the old one. The immutable blobs typically have a UUID (universally unique identifier) as their key in a scalable key-value store.

Storing immutable blobs in a nonlinearizable database does not have any problems with returning a stale version. Since there’s only one immutable version, there are no stale versions.

Storing immutable data in a nonlinearizable store enjoys the best of both worlds: it’s both *right* and *right now*.

E-commerce – shopping cart

In e-commerce, each shopping cart is for a separate customer. There's no need or desire for cross-cart consistency. Each shopping cart has a unique identity or key.

Customers are very unhappy if their access to a shopping cart stalls. Large e-commerce sites can measure the percentage of abandoned carts and customer sessions when they get slow. Slow carts correspond to a large drop-off in business. Product catalogs, reviews, and more must be fast and responsive or customers leave.

Shopping carts should be *right now* even if they're not *right*. It is measurably better for business and the customer experience to return a stale or otherwise incorrect answer if it can be done quickly. Users are asked to verify the contents of the shopping cart before confirming the sale.

In a nonlinearizable store, sometimes multiple old versions of the cart exist in the version history DAG. Relatively simple shopping-cart semantics facilitate combining different versions of a single user's shopping cart.²

E-commerce – product catalog

Product catalogs for large e-commerce sites are processed offline and stuffed into large scalable caches. Feeds from partners and crawls of the web are crunched to produce a sanitized and hopefully consistent collection of product-catalog entries.

Each product in the catalog has a unique identifier. Typically, the identifier takes you to a partition of the catalog. The partition has a bunch of replicas, each

FIGURE 8: **LINEAR VS. NONLINEAR VERSION HISTORIES**

containing many product descriptions [see figure 9]. One typical implementation of a scalable product cache has partitions with replicas. In this depiction, the columns are partitions and the rows depict replicas. The back-end processing produces new product descriptions that are distributed with pub-sub. Incoming requests are sent to the partition for the product identifier and then load-balanced across replicas.

Back-end processing of the feeds and crawls, as well as the pub-sub distribution of updates to the caches, are

throughput-sensitive, not latency-sensitive. Different replicas may be updated asynchronously, meaning it is not surprising to read a new version of the description, retry, and then get an old version from a cache replica that's not yet updated.

User lookups are very sensitive to latency. Just as shopping cart response times must be fast, product-catalog lookups must be fast. It is common for a client working to display the description of a product to wait for an answer, time out, and retry to a different replica if necessary to ensure the latency for the response is fast.

Note that the management of the short latency depends on the fact that any version of the product-catalog description is OK. This is another example of the business needing an answer *right now* more than it needs the answer to be *right*.

Search

Say you're building a search system for the contents of the web. Web crawlers feed search indexers. Each document is given a unique ID. Search terms are identified for each document. The index terms are assigned to a shard.

Updates to the index are not super-latency-sensitive. Mostly, changes observed by crawling the web are not latency-sensitive. Other than time-sensitive news feeds, the changes need not be immediately visible. When a random document is produced at some remote location in the world, it might take a while to be seen.

Search results are, however, sensitive to latency. In general, a search request from a user is fed into servers that ask all of the shards for matching results. This looks

FIGURE 9: PARTITIONS WITH REPLICAS

a lot like the product catalog depicted in figure 9, but the user requests hit all the shards, not just one of them.
It's very important that searches get quick results, or

users will get frustrated. This is aggravated by the need to hear back from all the servers. If any server is a laggard, the response is delayed. The mechanism for coping with this at Google is beautifully described in the 2013 article “The Tail at Scale.”¹

In search, it is OK to get stale answers, but the latency for the response must be short. There’s no notion of linearizable reads nor of read-your-writes. Search clearly needs to return answers *right now* even if they’re not *right*.

It’s about the application pattern

Each application pattern shows different characteristics and tradeoffs, shown in figure 10.

CONCLUSION

State means different things. Session state captures stuff across requests but not across failures. Durable state

FIGURE 10: APPLICATION PATTERN TRADEOFFS

APPLICATION PATTERN	PREDICTABLE READ LATENCY?	PREDICTABLE WRITE LATENCY?	READS YOUR WRITES?	TRADEOFFS
careful replacement with k/v	no	no	yes	works across multiple key/values
tx'l blobs for ref	yes	yes	immutable	non-linearizable plus immutable
ecommerce—shopping cart	yes	yes	no	sometimes give stale results
ecommerce—product catalog	yes	no	no	scalable cache means that stale is ok
search	yes	no	no	scalable cache plus search

Related articles

Non-volatile Storage

Mihir Nanavati et al.

Implications of the Datacenter's
Shifting Center

<https://queue.acm.org/detail.cfm?id=2874238>

Network Applications Are Interactive

Antony Alappatt

The network era requires new models, with
interactions instead of algorithms.

<https://queue.acm.org/detail.cfm?id=3145628>

Storage Systems: Not Just a Bunch of Disks Anymore

Erik Riedel

The sheer size and scope of data available
today puts tremendous pressure on storage
systems to perform in ways never imagined.
<https://queue.acm.org/detail.cfm?id=864059>

remembers stuff across failures.

Increasingly, most scalable computing consists of microservices with *stateless interfaces*. Microservices need partitioning, failures, and rolling upgrades, and this implies that stateful sessions are problematic. Microservices may call other microservices to read data or get stuff done.

Transactions across stateless calls are usually not supported in microservice solutions. Microservices and their load-balanced service pools make server-side session state difficult, which, in turn, makes it hard to have transactions across calls and objects. Without transactions, coordinated

changes across objects in durable state need to use the careful replacement technique in which updates are ordered, confirmed, and idempotent. This is challenging to program but is a natural consequence of microservices, which have emerged as the leading technique to support scalable applications.

Finally, different applications demand different behaviors from durable state. Do you want it *right* or do you want it *right now*? Human beings usually want an answer *right now* rather than *right*. Many application

solutions based on object identity may be tolerant of stale versions. Immutable objects can provide the best of both worlds by being both *right* and *right now*.

Consider your application's requirements carefully. If you aren't careful, you will have problems with your state that you will definitely mind.

References

1. Dean, J., Barosso, L. A. 2013. The tail at scale. *Communications of the ACM* 56 [2], 74-80.
2. 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. *Proceedings of the 21st ACM SIGOPS Symposium on Operating System Principals*, 205-220.
3. Garcia-Molina, H., Salem, K. 1987. Sagas. *Proceedings of the ACM SIGMOD Conference on Management of Data*, 249-259; <https://www.cs.cornell.edu/landrulcs711/2002/fall/readings/sagas.pdf>
4. Gray, J., Reuter, A. 1992. *Transaction Processing: Concepts and Techniques*, Burlington, MA, Morgan Kaufmann, 508-509.
5. Helland, P. 2005. Data on the outside versus data on the inside. *Proceedings of the Conference on Innovative Database Research*.
6. Helland, P. 2002. Fiefdoms and emissaries; http://download.microsoft.com/documents/uk/lmsdn/architecture/connected/fiefdoms_emissaries.ppt
7. Helland, P. 2012. Idempotence is not a medical condition. *acmqueue* 10[4], 56-65.
8. Helland, P. 2016. Immutability changes everything.

- acmqueue* 13[9]; <https://queue.acm.org/detail.cfm?id=2884038>.
9. Helland, P. 2016. Life beyond distributed transactions. *acmqueue* 14[5]; <https://queue.acm.org/detail.cfm?id=3025012>.
 10. Helland, P. 2016. Standing on distributed shoulders of giants. *acmqueue* 14[2]; <https://queue.acm.org/detail.cfm?id=2953944>.
 11. Lakshman, A., Malik, P. 2010. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review* 44[2], 35-40.
 12. von Neumann, J. 1993. First draft of a report on the EDVAC. *IEEE Annals of the History of Computing* 15[4], 27-75.

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 © 2018 held by owner/author. Publication rights licensed to ACM.