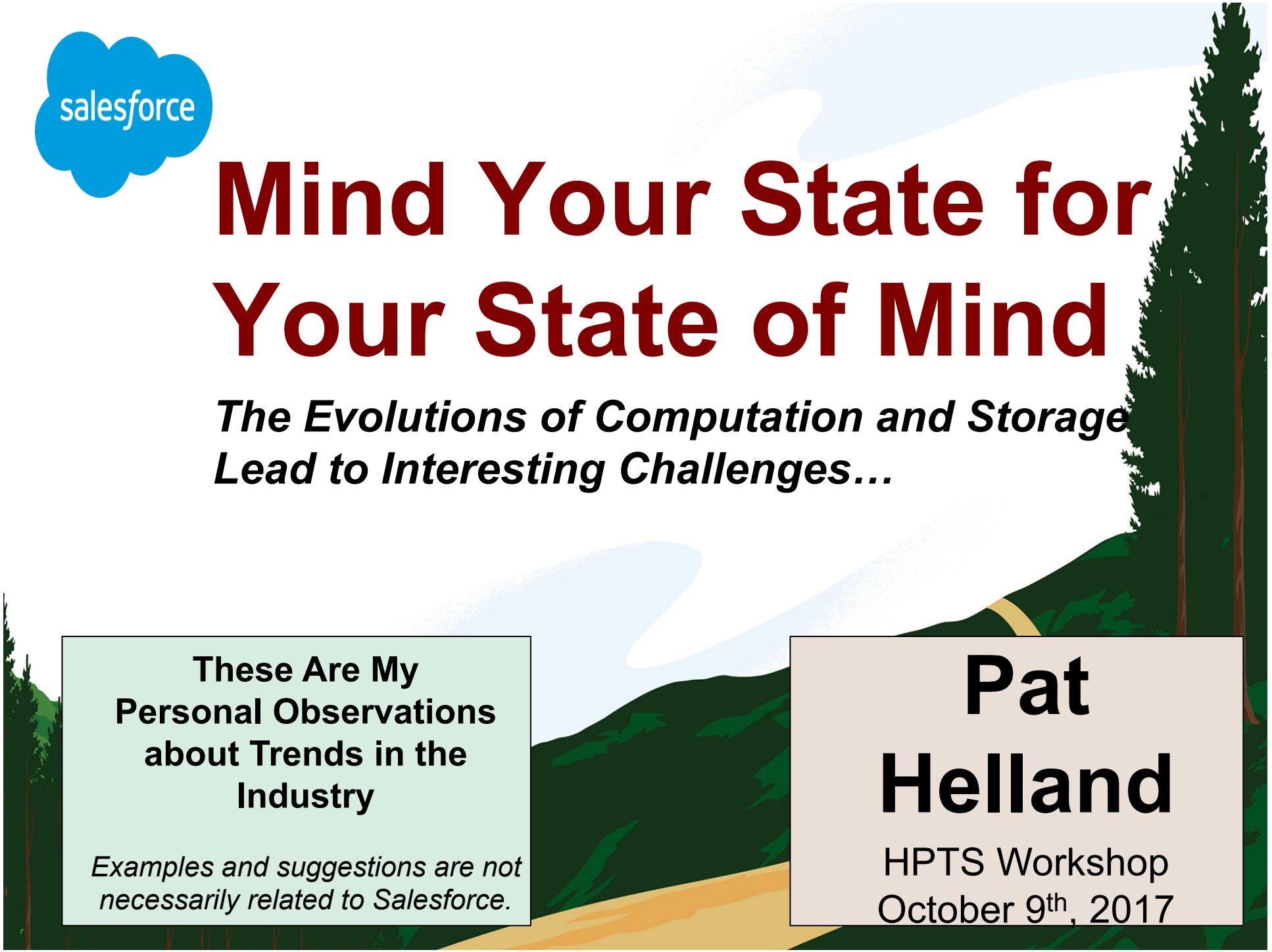


Mind Your State for Your State of Mind

The Evolutions of Computation and Storage Lead to Interesting Challenges…

These Are My Personal Observations about Trends in the Industry

Examples and suggestions are not necessarily related to Salesforce.

Pat Helland

HPTS Workshop
October 9th, 2017

Outline

- Introduction
- What's This State Stuff?
- The Evolution of Durable State Semantics
- Session State Semantics and Transactions
- Identity, Immutability, and Scale
- Some Example Application Patterns
- Conclusion

Trends in Storage and Computing

- **Storage has evolved**
  - Used to be direct attached only
  - Shared appliances (e.g. SAN)
  - Storage clusters contained in a network
  - REST APIs over microservices
- **Computing has evolved**
  - Single process (Mainframe Region)
  - Multiple processes in same server
  - RPC across a tiny cluster
  - Services & SOA (Service Oriented Architecture)
  - Microservices with little or no state

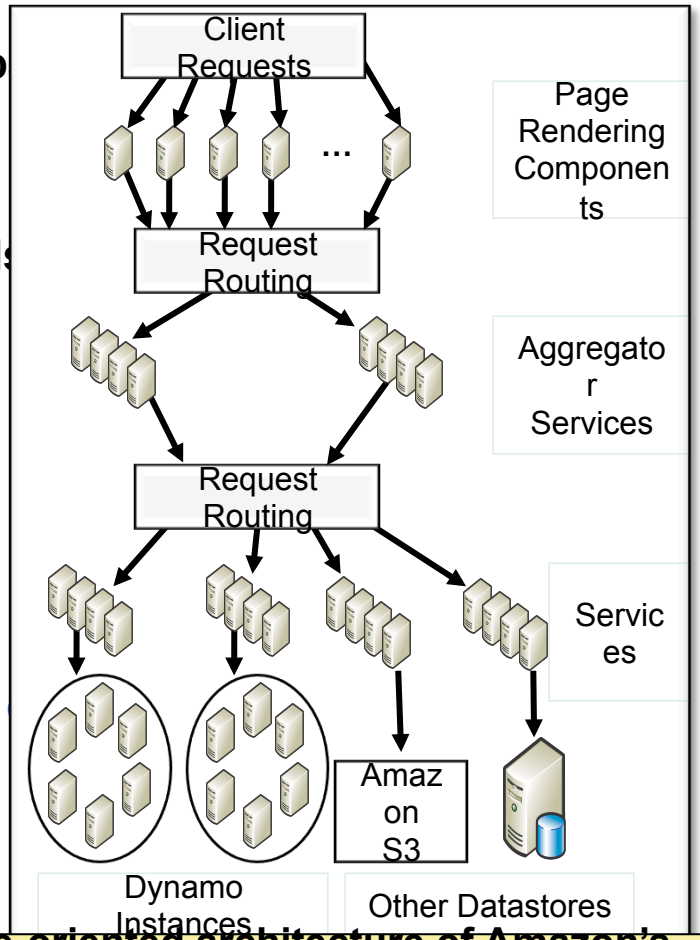
- **Computing's use of storage has evolved**
  - Direct File I/O
    - Use *careful replacement* for recoverability
  - Transactions
    - Implemented careful replacement for the app
    - Later, SANs implemented careful replacement
    - Stateful 2-tier and N-tier transactions
  - Key-Value
    - Typically, atomic per-key updates
  - REST PUTs
    - Invokes the App code of the resource…

Careful Replacement Variations:

1) A write may trash the previous value… Write elsewhere first
2) A client crash may interrupt a sequence of writes… plan carefully!

Challenges in Modern Microservice-based Apps

- **Nowadays, microservices power many scalable app**
  - Pools of equivalent services
  - Incoming requests are load-balanced across the pool
- **Microservices must support many operational needs**
  - Health mediated deploy (canaries)
  - Rolling upgrades (sensitive to fault zones)
  - Fault tolerance
- **Durable state is usually not kept in microservices**
  - Can't effectively update the state across all the services
    - Especially when they are coming and going willy-nilly
  - Typically latest state is kept elsewhere and versions are
  - Sometimes, read-through requests to durable state access information that is NOT in microservices



Service-oriented architecture of Amazon's platform

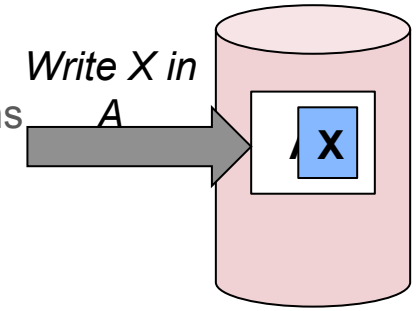
From Dynamo: Amazon's Highly Available Key-value Store
--- SOSP 2007

Outline

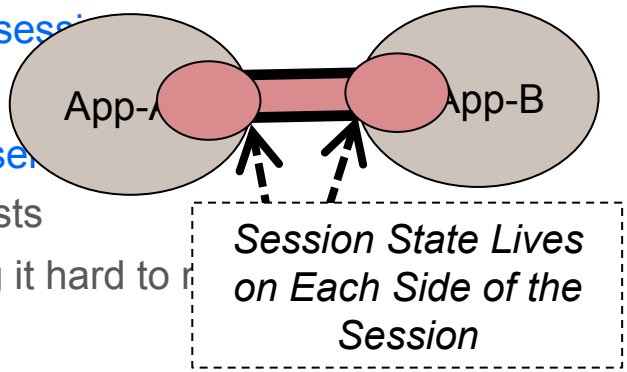
- Introduction
- What's This State Stuff?
- The Evolution of Durable State Semantics
- Session State Semantics and Transactions
- Identity, Immutability, and Scale
- Some Example Application Patterns
- Conclusion

Durable State and Session State

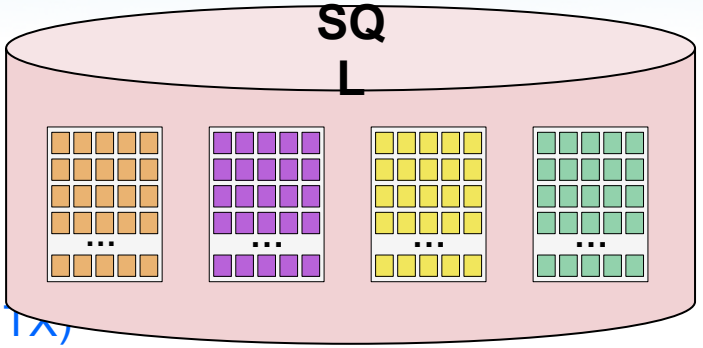
- **Durable State:** *Stuff that gets remembered across requests and persists across failures*
  - What is it?
    - Database data, filesystem files, key-values, caches
  - How is it updated?
    - Single updates, transactions, and/or distributed transactions
    - Careful replacement
    - Messaging semantics
  - Can you read your writes consistently?
    - Weakly consistent stores and caching each make “read your writes” a challenge



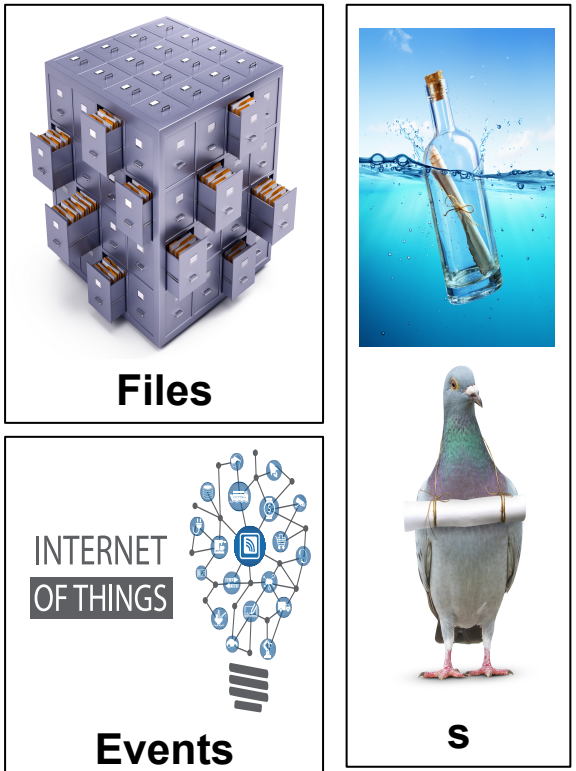
- **Session State:** *Stuff that gets remembered across requests in a session but not across failures*
  - Session state exists within the endpoints associated with the sess
    - Multi-operation transactions are a form of session state
  - Session state is hard to do when the session smears across se
    - Different Microservices in the pool may service later requests
    - Typically session state is kept in a service instance making it hard to r



Data on the Inside vs Data on the Outside Redux



- **Data on the Inside**
  - Classic transactional relational data
  - Tables, rows, columns → *values within cells!*
  - Lives at one place (the DB) and at one time (the TX)
- **Data on the Outside**
  - Messages, files, events, key-value-pairs
  - Unlocked data not stored in a classic database
  - Identity and (optional) versioning for each item
- **Outside Data is immutable (but may be versioned)**
  - Each file/event/message/key has a unique identifier
  - The ID may be a URI, a key, or something else
    - It may be implicit on a session
    - It may be implicit within the environment

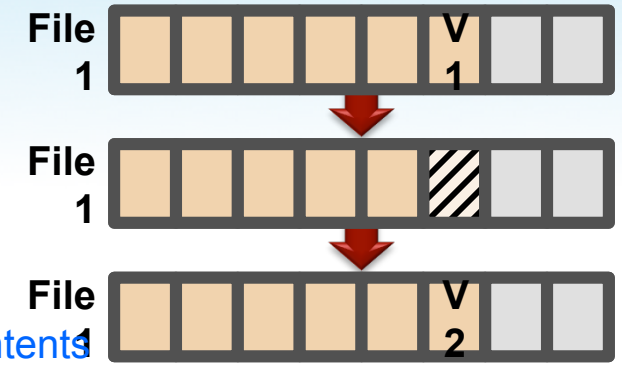


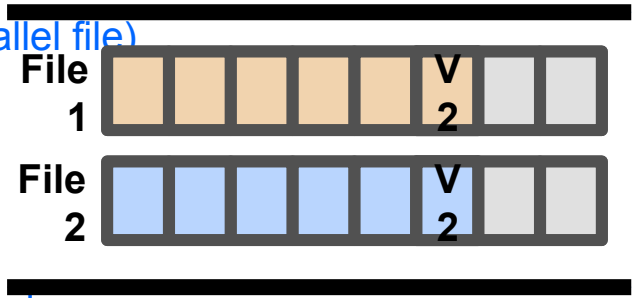
Outline

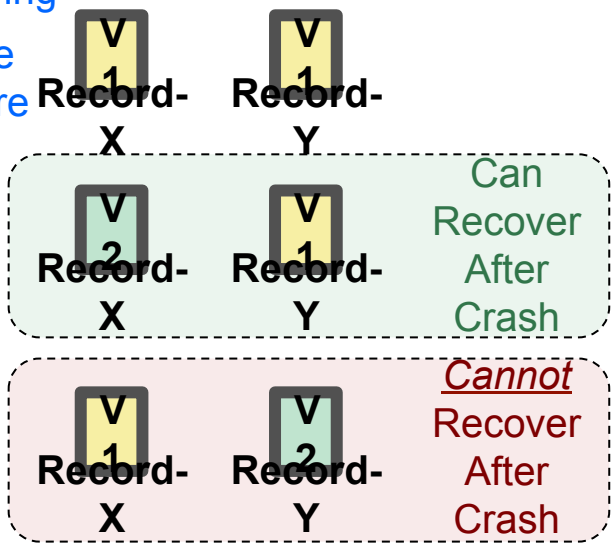
- Introduction
- What's This State Stuff?
- The Evolution of Durable State Semantics
- Session State Semantics and Transactions
- Identity, Immutability, and Scale
- Some Example Application Patterns
- Conclusion

Careful Replacement

- **Used to be: disks might trash a block during a write**
  - WRITE: Old-Value → Unreadable-Trash → New-Value
  - Power failures or other interruptions may destroy the old contents
- **Careful Replacement for Single Disk Block Writes**
  - Write the new value into some other place (e.g., another parallel file)
  - Only after the new value is safe, overwrite the old place
    - Write the tail of the log carefully onto mirrored disks…
- **Careful Replacement for Record Writes**
  - Update to records in pre-SQL databases needed careful ordering
  - In many cases, an update to one record (say Record-X) before updating Record-Y allows the application to recover after failure







Example: Application Queue

Using a record as an entry in a work queue combined with idempotent work will yield a successful restart.

Transactions and Careful Replacement

- **Transactions bundle and solve *careful record replacement***
  - Multiple application records may be updated in a single transaction
  - The database system ensured the record updates were atomic
- **Databases handle challenges with *careful storage replacem***
  - As the database implemented transactions, it was aware of the needs of storage
  - Writes to storage (implementing the database) used careful replacement
  - Distributed transactions handled work across a small number of intimate database servers
- **Work across time (i.e. workflow) needs *careful transactional replacement***
  - While a set of records was atomic, work across time requires careful replacement
  - Failures, restarts, and new work can advance the state of the application TX by TX
- **Work across space (i.e. cross-boundary) needs *careful transactional replacement***
  - Work across space necessitates work across time, TX by TX
  - This leads us to Messaging Semantics…

"It's Déjà Vu All Over Again"

-- Yogi Berra

Messaging Semantics

- **Transactional messaging is pretty cool**
  - A transaction may include the *desire to send* a message
    - Transactional updates happen atomically with desire to
  - A transaction may *atomically consume* an incoming messa
    - Message consumption is atomic with the work of the me
- **Exactly once semantics can be supported**
  - A committed desire to send, causing *one or more sends* (retry until acknowledged)
  - The message must be processed at the receiver at-most-o (idempotent processing)

Challenges with At-Most-Once

Remember the Messages You've Processed

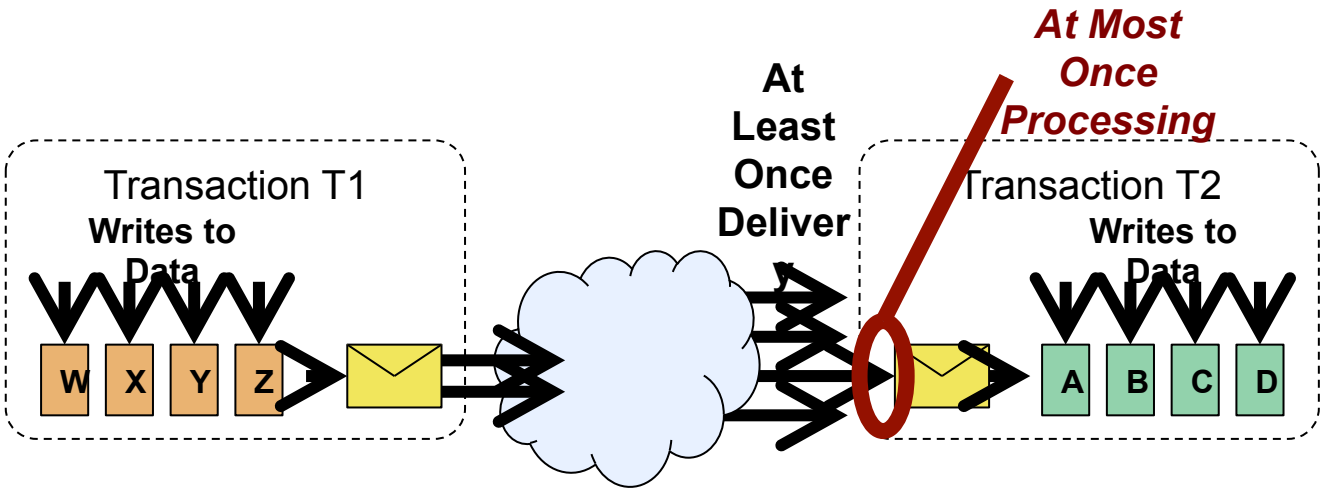
Don't Process the Message Twice

How Do You Remember Messages?

Gotta Detect Duplicates

How Long Do You Remember?

Does the Destination Split? Move?



Read Your Writes? Yes? No?

Used to Be, Back in the Day…
If You Wrote Something, You Could Read It…

- **Linearizable stores offer "Read your writes"**
  - Even as the store scales, as soon as you've written to the store, you can read the latest value
  - Linearizable → Occasionally delay for a LONG time when a server is sick or dead
- **Non-Linearizable stores do NOT offer "Read your writes"**
  - Non-Linearizable → No guarantee that a write will update all the replicas → Might read an old value
  - **Reading and writing have a very consistent SLA**… Skip over sick / dea
- **Cached data offers scalable read throughput with great SLAs**
  - Key-value pairs live in many computers and are updated with versions
  - Reads hit one of the computers and return one of the versions

See "Linearizability versus Serializability" by Peter Bailis	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

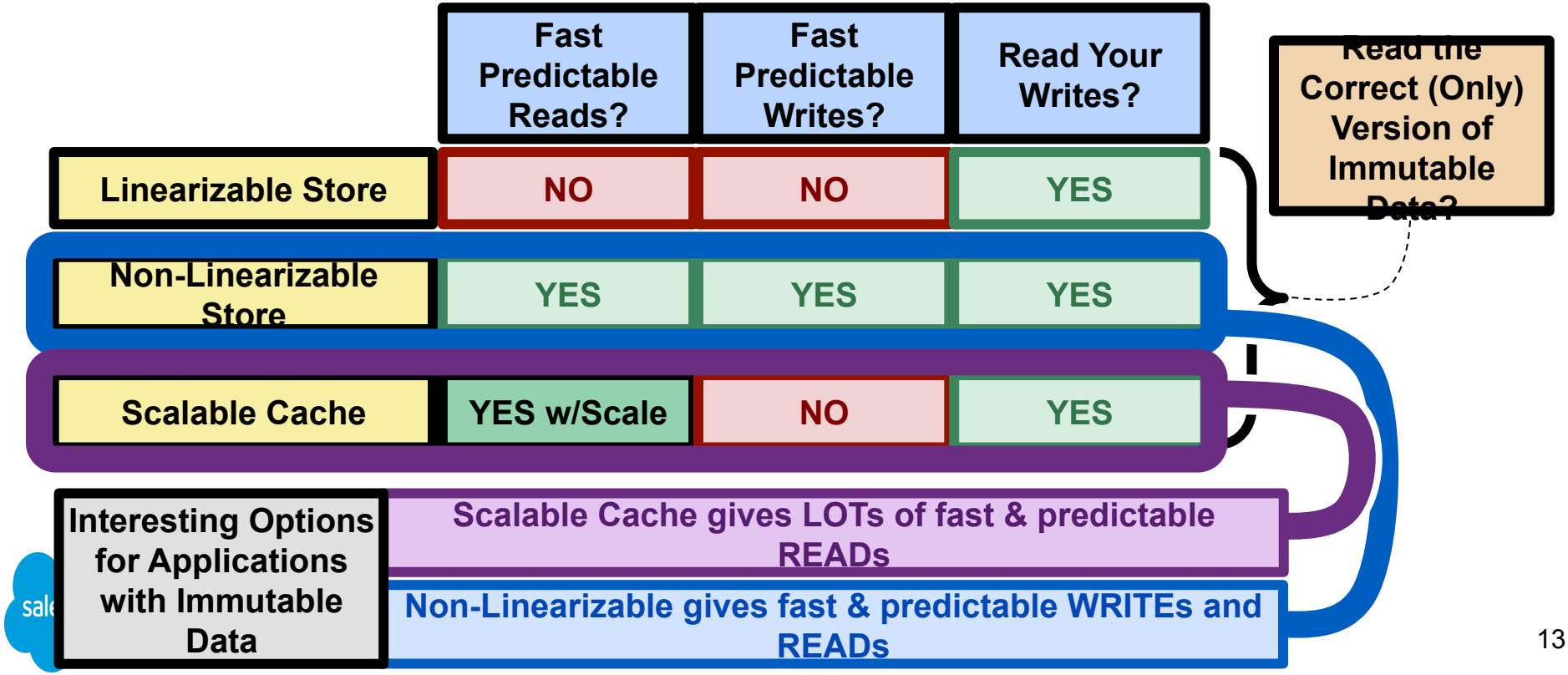
Different Stores for Different Uses

OK to Stall on Read?	OK to Stall on Write?	OK to Return a Stale Version??
		Can't Have Everything!!!

Immutability: A Solid Rock to Stand On

- **Sometimes, we can store immutable things**
  - If you look for it, many application patterns can create immutable items
  - 128-bit UUID is an example of an identity that won't collide with other s
- **Storing immutable things can change the behavior of a store**
  - You never get an old version of the thing because each old version ha ID

Immutability Changes Everything!!!!

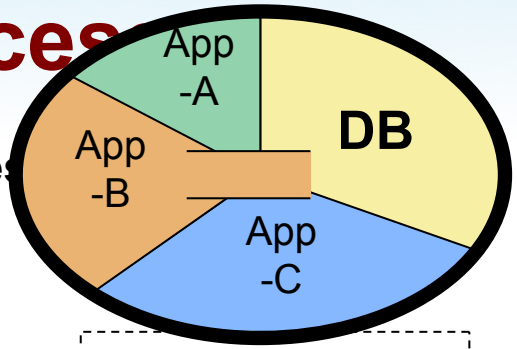


Outline

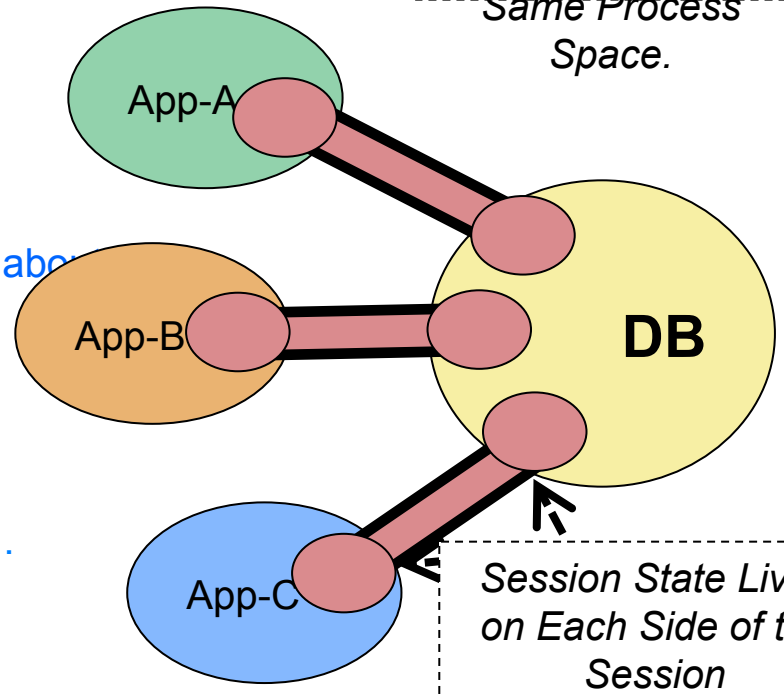
- Introduction
- What's This State Stuff?
- The Evolution of Durable State Semantics
- Session State Semantics and Transactions
- Identity, Immutability, and Scale
- Some Example Application Patterns
- Conclusion

Same Process → Different Proces

- **Applications and databases used to run in the same proces**
  - A library call from the app to the database was easy
  - Sometimes, multiple applications were loaded together
- **Later, the DB and Apps were split apart connected by a session**
  - The session had *session state*
    - User
    - Transaction in flight
    - Application being run
    - Cursor state and return values
  - Each process in the session had information abo the session (i.e. *session state*)
- **Later still, the app and database moved to different servers**
  - Sessions and session state made that work…



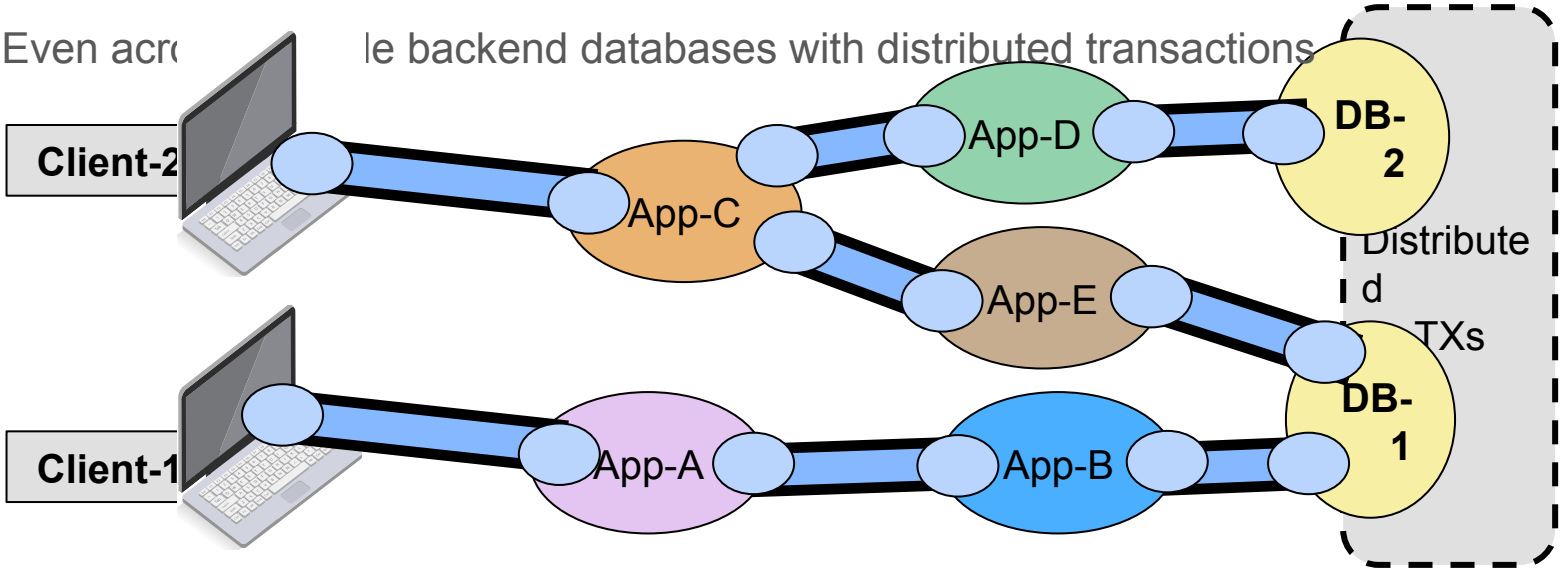
Multiple Applications Coexisted in the Same Process Space.



Session State Lives on Each Side of the Session

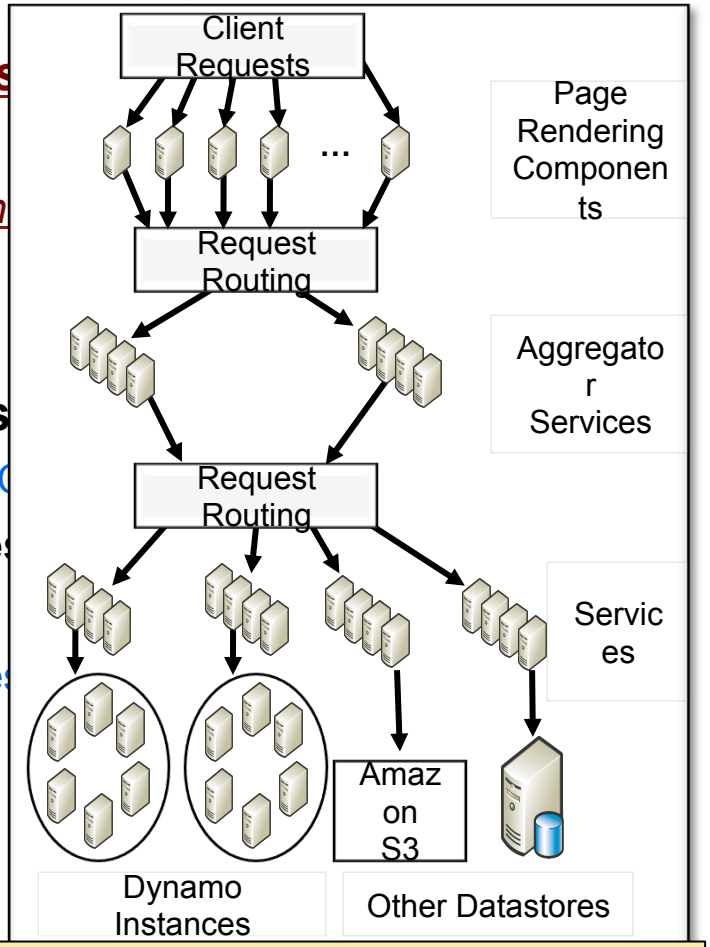
Stateful Sessions and Transactions

- **Stateful sessions were the natural outcome of shared processes**
  - You knew who you were talking to… You used to be in the same process!
  - You knew who you were talking to… You can remember stuff about the other guy
- **Stateful sessions worked well for classic SOA (Service Oriented Architecture)**
  - 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 acr le backend databases with distributed transactions



Transactions, Sessions, and Microservices

- ***Microservices stink when it comes to session s***
  - Requests flow to request routing
  - *Usually, they go back to the same microservice instan*
  - If the individual instance fails then another is used!
    - No more session state!
- **Session state is needed to create cross-reques**
  - The transaction identity and who needs to be in 2 Phase C
- **Microservice transactions are typically 1 store reque**
  - The lack of session state makes multiple updates hard
  - The challenges of 2 Phase Commit make multiple update



Service-oriented architecture of Amazon's platform

From Dynamo: Amazon's Highly Available Key-value Store --- SOSP 2007

Microservices Are Worth the Restrictions!

Fail-fast, Load Balanced, Health Mediated Deploy (Canaries), Rolling Upgrades, Fault Tolerance, and More

It's Not Your Grandmother's Transaction Anymore!

Transactions Only Work on a Single Call to the

Scalable Microservices:

As the Application Microservices Scale, More Instances Are Made

As Microservices Compose, They Call One Another

Scalable Stores:

TX Across Multiple Identities → Distributed TX Across Store Instances

Scalable Linearizability:

Per-Identity
"Read Your Writes"

Scalable Non-

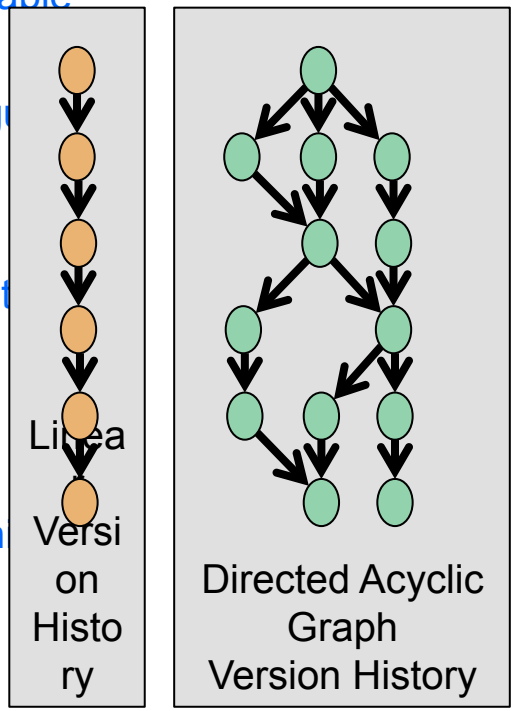
Per-Identity
"Read One or More Old Versions"

Outline

- Introduction
- What's This State Stuff?
- The Evolution of Durable State Semantics
- Session State Semantics and Transactions
- Identity, Immutability, and Scale
- Some Example Application Patterns
- Conclusion

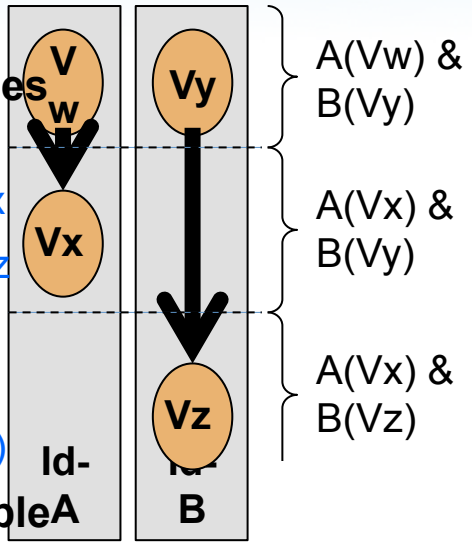
What's Identity ?

- **Each identity is represented by some number, string, key, or**
  - The identity can reference something that's immutable
    - New York Times, October 9th, 2017, San Francisco Bay Area ec
  - The identity can reference something that changes over time
    - Today's New York Times
- **Each version of the identity is immutable**
  - A change makes a new version… Hence, each version is immutable
  - Creating an identity for the immutable version is REALLY useful
  - Now, caching, copying, and referencing are not subject to ambigu
- **Version history may be linear**
  - That's called linearizability (per identity)
  - Requires strongly consistent and ordered per key (identity) updat
- **Version history may be a DAG (directed acyclic graph)**
  - This is called non-linearizabililty (per identity)
  - Independent updates happen separately
  - Concurrent versions come back together representing a fork in h

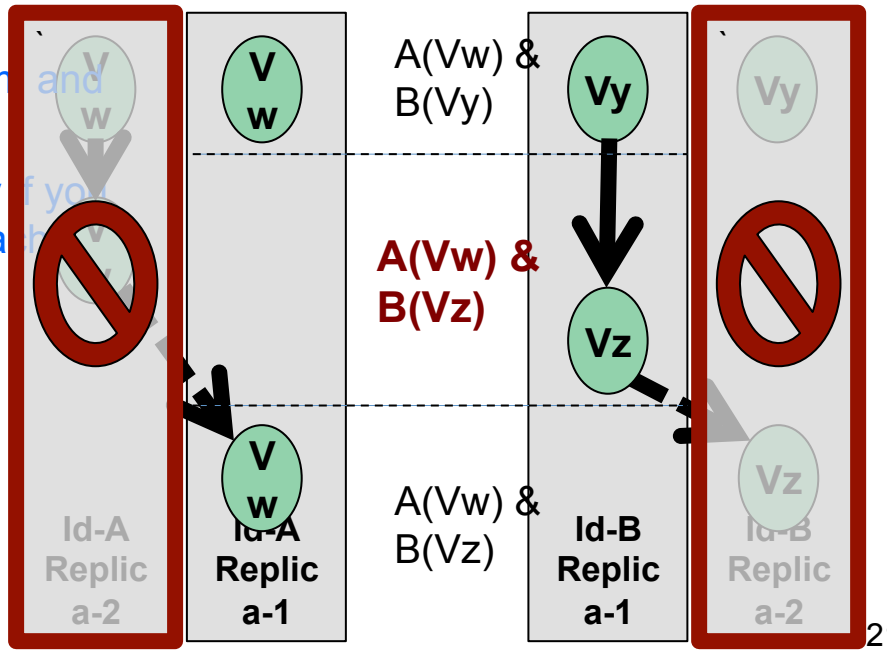


Cross Identity Relationships

- **Using careful replacement across identities is a tried-and-true technique**
  - First update item with Identity-A so Vw → Vx
  - Then update item with Identity-B so Vy → Vz
- **Careful replacement is predictable over linearizable stores**
  - Never read B(Vz) unless you can read A(Vx)
- **Careful replacement over a non-linearizable store will behave unpredictably**
  - You may write a new version of some Id's item and then read the Id and get an older version
  - Cached stores will also behave unpredictably if you are allowed to read stale versions from the cache
- **Careful replacement will be buggy over non-linearizable stores!**



Each Window of Time Where B is Vz also shows A as Vx



How “Append” Blurs Identity

- **HDFS and other “big data” file systems accept WRITEs to append to a big file**
  - It is essential that the appended writes preserve the order
  - It is essential that each append appear exactly once
- **Predictable and repeatable replica data takes careful design**
  - GFS (Google File System) allowed multiple writers to a file and had fixed sized blocks
    - Failures & race conditions sometimes allowed different data per replica → *unpredictable read values*
  - HDFS restricts clients to single writer and avoids this problem (still with fixed sized blocks)
  - Bing’s COSMOS allows multiple writers but has variable length blocks that are shortened for failures
- **APPEND to file-XYZ does not have adequate identity**
  - APPEND has the File’s identity but not the location of the APPEND within the file
  - Big Data system assign the order at the primary block server… this does offer correct semantics
  - Assigning APPEND location means WRITES (appends) may stall when servers fail or stutter
- **Stalls on “Big Data” APPENDS are just fine as it’s usually used for batch**
  - Stalls are less important than throughput unless humans are stuck waiting for the answer

Outline

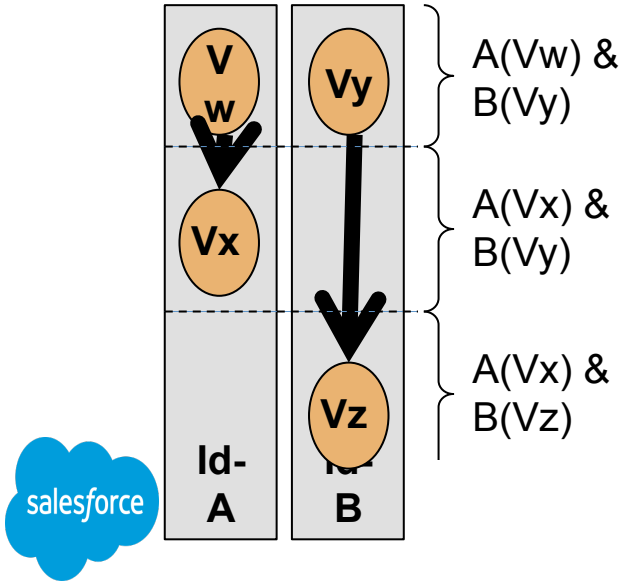
- Introduction
- What's This State Stuff?
- The Evolution of Durable State Semantics
- Session State Semantics and Transactions
- Identity, Immutability, and Scale
- Some Example Application Patterns
- Conclusion

Just a Few Patterns as Examples…

There Are Many More!

Careful Replacement over Key-Value

- **Objects (values) are uniquely identified by their key**
  - Work arrives from outside via messages or human intera
  - Workflow can be captured in the values
  - New values are written to replace the old
  - Messages are contained as data within the objects
- **Scalable applications can be built over key-value st**
  - Single-item linearizabillity (“read your writes”)
  - Correct behavior is more important than occasional stutt

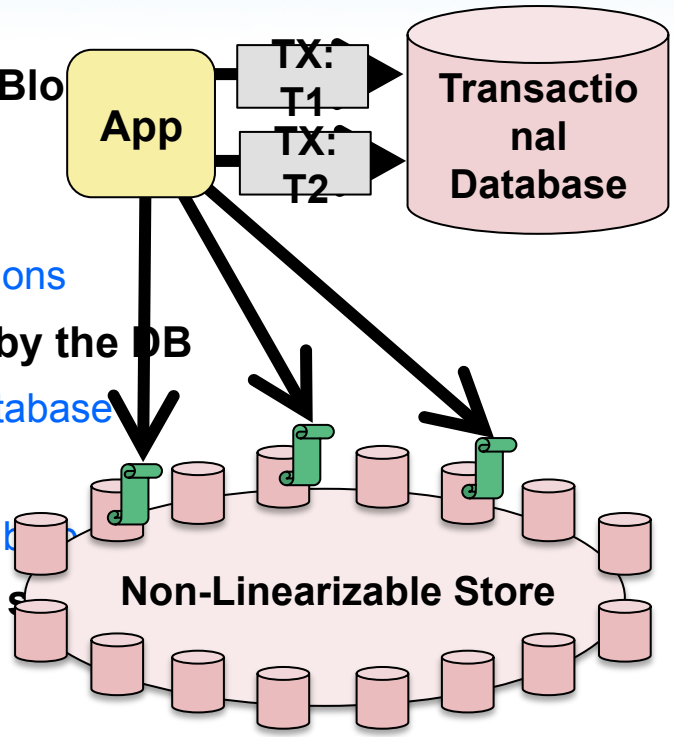


Each Window of Time Where B is Vz also shows A as Vx

Low Latency Predictable Reads?	Low Latency Predictable Writes?	Read Your Writes?
NO	NO	YES

Transactional Blobs-by-Ref

- **Imagine a large relational system storing many large Blo**
  - The blobs are correlated with relational data
  - Blobs are documents, photos, and other stuff
  - Updates to blobs are rare and implemented with new versions
- ***Immutable blobs*** **are placed into the store controlled by the DB**
  - Transaction T1: Allocate BLOB-ID-X and remember in database
    Copy the blob into the blob-store with BLOB-ID-X
    Transaction T2: Remember BLOB-ID-X refers to an intact b
- **The blob store is implemented with *many* commodity s**
  - Don't want a delay when a server gets sick or dies
  - Humans are waiting for blob writes and blob reads
- **Non-linearizable stores have excellent SLAs**
  - Writes put replicas in healthy servers (bypassing sick
  - Reads fetch blobs from any healthy server that answ



Low Latency Predictabl e Reads?	Low Latency Predictabl e Writes?	Read Your Writes?
YES	YES	Immutabl e

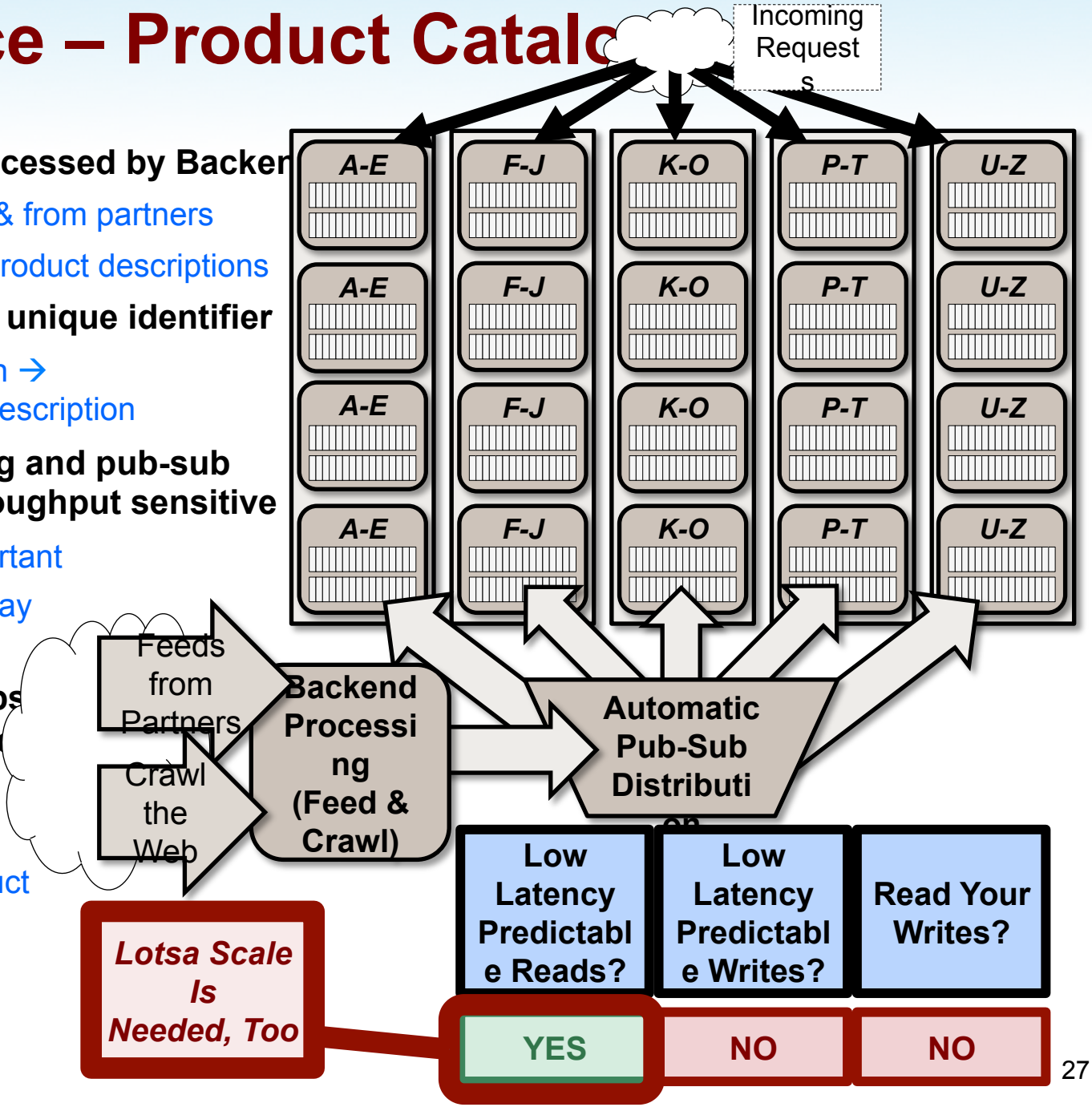
ECommerce – Shopping Cart

- **Each shopping cart is for a separate customer**
  - No problems with cross-customer consistency of reads and writes
- **Customers are *very unhappy* if the shopping cart stalls**
  - Attrition rate from delays is easily measured and a big impact!
  - Product catalog, reviews, and more must be responsive (more below)
  - Shopping cart additions or reads must be responsive
- **Shopping carts should be *Right Now* even if they're not *Right***
  - It is measurably better for the business to show the wrong results than to stall and take too long
  - Users are always asked to verify the cart before completing the purchase
- **In non-linearizable stores, sometimes multiple old versions of the cart exist within the Version History DAG**
  - Items are added and deleted from shopping carts
  - Relatively simple cart semantics facilitates combinir different versions of a single user's shopping cart

Low Latency Predictable Reads?	Low Latency Predictable Writes?	Read Your Writes?
YES	YES	NO

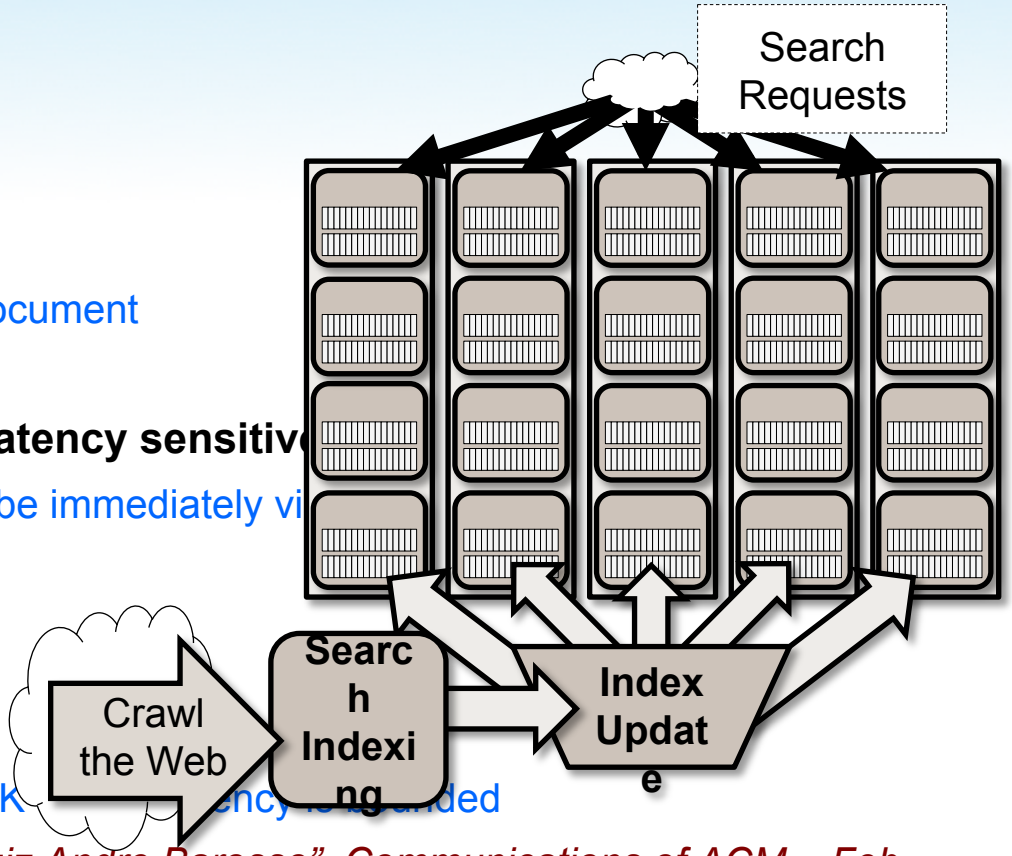
ECommerce – Product Catalo

- **Feeds & Crawls processed by Backen**
  - Data from the web & from partners
  - Produces distilled product descriptions
- **Each product has a unique identifier**
  - Identifier → partition → replica → product description
- **Backend processing and pub-sub distribution are throughput sensitive**
  - Latency is not important
  - Different replicas may be updated asynch
- **User catalog lookups are latency sensitive**
  - Scale is important
  - OK to get a stale version of the product



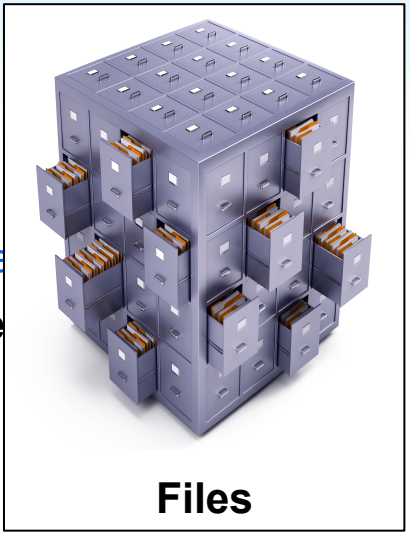
Search

- **Web crawlers feed search indexers**
  - Search terms are identified for each document
  - Search shards are by index terms
- **Updates to the index are not super latency sensitive**
  - Most changes from crawling need not be immediately vi
  - OK to update and not have changes immediately seen
- **Very important that search requests get low latency responses**
  - Retries to other servers in shard are OK
  - See *"Tail at Scale by Jeff Dean and Luiz Andre Barosso" Communications of ACM – Feb 2013*
  - OK to get mixed staleness of answers



Lotsa Scale Is Needed, Too	Low Latency Predictable Reads?	Low Latency Predictable Writes?	Read Your Writes?
	YES	NO	NO

Big Data: Appending to Big Fi



- **HDFS and Hadoop generate data with APPENDs to big files**
  - APPENDS are identified with the file name but not the target byte offse
- **The byte offset is assigned by the primary replica of the storage**
  - The primary must be the one to assign the offset
    - If the primary dies, another server is assigned the role of primary
  - *This is a form of linearizability of the APPENDS*
- **APPENDs will usually be very fast but may be delayed if a server is sick or dead**
  - Dead servers are removed and a protocol is followed to select a new server for the role
- **APPENDs can be delayed while the system copes with a sickness or death**
  - This is OK when the Big Data solution is running a batch job
  - The overall throughput is what matters, not an occasional stutter
- **READs of a Big Data file can bound the SLA**
  - Three servers have the data and it's immutable
  - Read from any of the three and it's great!!

Low Latency Predictable Reads?	Low Latency Predictable Writes?	Read Your Writes?
YES	NO	Immutable

Outline

- Introduction
- What's This State Stuff?
- The Evolution of Durable State Semantics
- Session State Semantics and Transactions
- Identity, Immutability, and Scale
- Some Example Application Patterns
- Conclusion

It's About the Application Pattern!

	Low Latency Predictable Reads?	Low Latency Predictable Writes?	Read Your Writes?	
Careful Replacement (K/V)	NO	NO	YES	*Work across Multiple Key/ Values*
TX'l "Blobs-by-Ref"	YES	YES	Immutable	*Non-Linearizable plus Immutable*
EComm – Shopping Cart	YES	YES	NO	*Sometimes Gives Stale Result*
EComm – Product Catalog	YES	NO	NO	*Scalable Cache → Stale OK*
Search	YES	NO	NO	*Scalable Cache plus Search*
Append to Big Files	YES	NO	Immutable	*File Append Semantics Require Linearizability of Appends*
				Throughput, not Latency Matters

Linearizability and "Read Your Writes" Are Not Always Required in Modern Scalable Applications

How You Use State Depends on Your Application Requirements!

Takeaways

- **"State" means different things:**
  - **Session state:** Stateful sessions remembers stuff; Stateless doesn't remember on the session
  - **Durable state:** Stuff is remembered when you come back to the later
- **Most scalable computing comprises microservices with *stateless interfaces***
  - Microservices need partitioning, failures, and rolling upgrades → stateful sessions are problematic
  - Microservices may call other microservices to read data or to get stuff done
- **Transactions across stateless calls usually aren't supported in microservice solutions**
  - Microservices → no server-side session state → no txs across calls → no txs across objects
- **Coordinated changes use the *careful replacement* technique** *(from computing's early days)*
  - Each update provides a new version of the stuff with a single identity
  - Complex content *within* the new version may include many things including outgoing/ incoming messages
- **Different applications demand different behaviors from durable state**
  - Do you want it ***right*** ("read your writes") or do you want it ***right now*** (bounded and fast SLA)?
    - Humans usually prefer *right now* to *right* !
  - Many app solutions based on object identity may be tolerant of stale versions
  - Immutable objects can provide the best of both by being *right* and *right now*