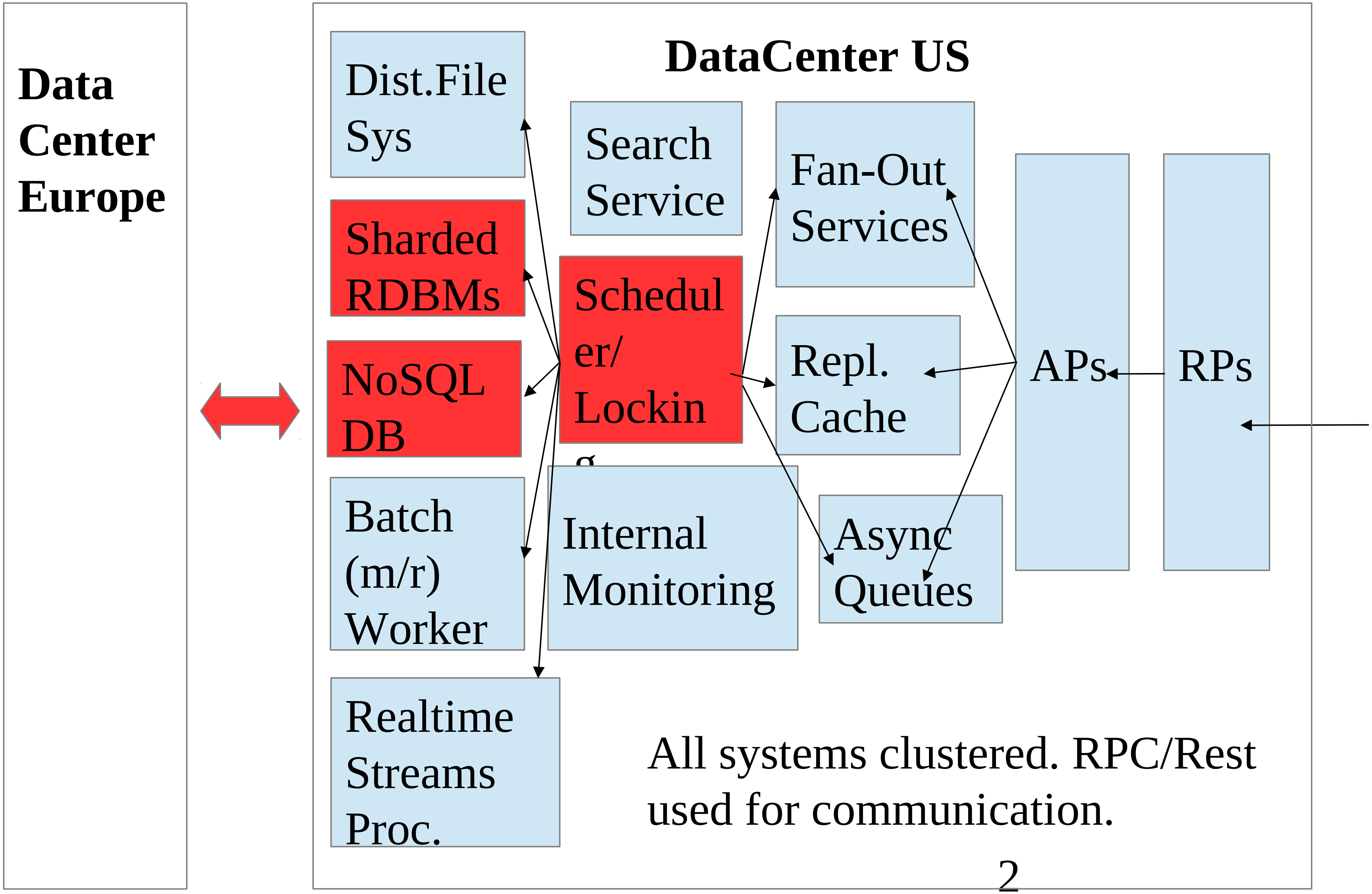
Lecture on

Distributed Services and Algorithms
Part 2

Living (faster) with Uncertainty

Walter Kriha

Distributed Operating Systems II



Data Center Europe

DataCenter US

Dist.File Sys

Sharded RDBMs

NoSQL DB

Batch (m/r) Worker

Realtime Streams Proc.

Search Service

Scheduler/ Locking

Internal Monitoring

Fan-Out Services

Repl. Cache

Async Queues

APs

RPs

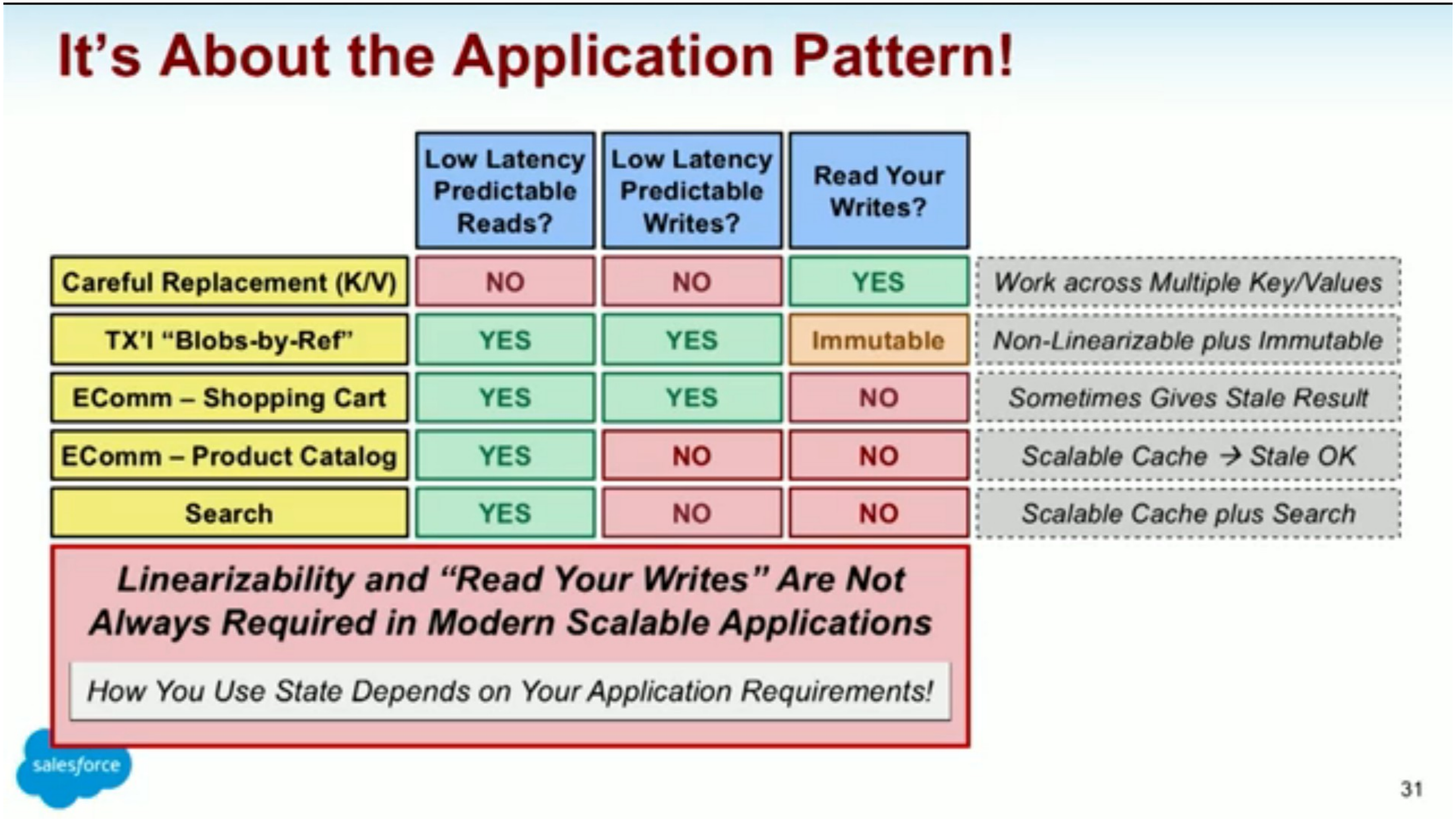
All systems clustered. RPC/Rest used for communication.

When the Truth is Prohibitively Expensive

> “Stop relying on strong consistency. Coordination and distributed transactions are slow and inhibit availability. The cost of knowing the “truth” is prohibitively expensive for many applications. For that matter, what you think is the truth is likely just a partial or outdated version of it.
>
> Instead, choose availability over consistency by making local decisions with the knowledge at hand and design the UX accordingly. By making this trade-off, we can dramatically improve the user’s experience—most of the time.” Tyler Treat, Distributed Systems Are a UX Problem, www.bravenewgeek.com

The quote shows a new understanding of consistency. It all started with CAP and now it is taken further and further, including cheating and bending the problems...

Fast read/write vs. read your writes



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*

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

How You Use State Depends on Your Application Requirements!

Pat Helland, Scale By The Bay 2018, Keynote III Mind Your Sta.mp4

Overview

- classic (ACID) distributed consistency
  Distributed 2P locking
  Distributed 2PC consensus
- ACID 2.0 eventual (coordination-free) consistency
  - CAP and its children, CALM, CRDTs etc.
  - distributed replication (cassandra etc.)
  - CALM (bloom) consistency
  - CRDTs
- Distributed Coordination (chubby, zookeeper)
  - distributed consensus protocols
  - Cluster scheduler (borg)

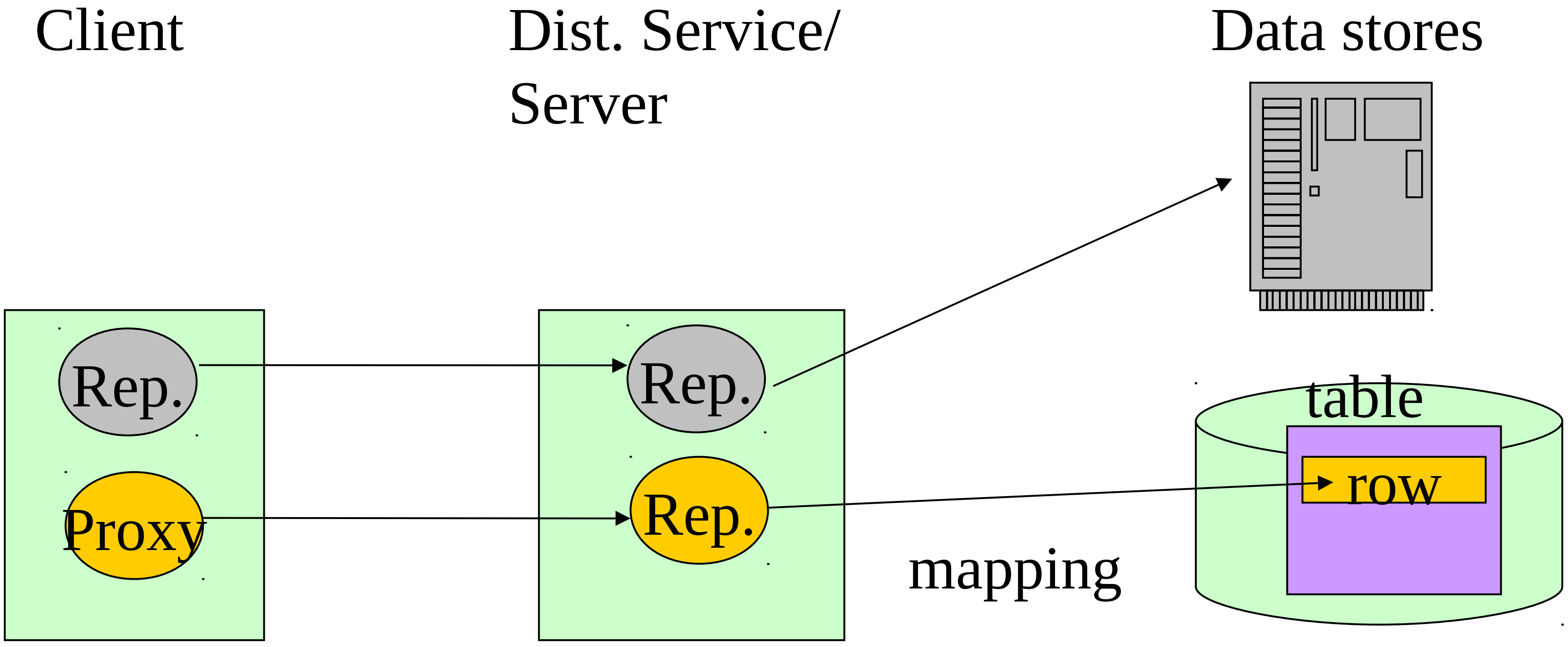
Classic (ACID) Distributed Consistency

- Distributed Objects and Persistence
- ACID
- Transactions
- Isolation Levels
- Two-Phase Locking
- Distributed Transactions
- Two-Phase Commit (2PC)
- Failure Models for 2PC

“Transaction processing expert Phil Bernstein suggests that serializability typically incurs a three-fold performance penalty on a **single-node** database compared to one of the most common weak isolation levels called Read Committed! (P.Bailis, Readings in Database Systems, 2015, ch.6 Weak Isolation and Distribution)

Persistent Distributed Objects

Persistent Object Representations



The real storage object lives in a data store and uses data store concepts for storage, e.g. a row in a table. The service works with object representations ("Incarnations" according to Emmerich) and provides the illusion of a persistent object to clients. The Java Connector Architecture provides an adapter interface for resource managers.

Mechanisms for Persistence

1) Use an SQL Driver to store object state. Suffers from "impedance mismatch" and needs to control locking etc. in the service.
2) Use an object/relational mapper (e.g. EJB/Hibernate) to store object state transparently for the programmer.

Just storing an object is simple. Doing this in a way that protects from concurrent access, system failures and across different data stores is much harder.

Persistent Object Mapping

Object view

Class X {

Int fieldA;

String fieldB; }

Data store view

Create Table Y,

1 integer (primary key) ,
2 string

Mapping specification:

Class X to table Y

fieldA to column 1, tagged as primary key

fieldB to column2

The key to persistent mapping is meta-information. It is used to generate both the object representations for a service and the code necessary for the data store to store the objects with its own mechanisms and objects. Enterprise integration software also specializes in this kind of mapping.

Object Mapping Approaches

Object view

Class X {

Int fieldA;

String fieldB; }

Class Y extends X{

Int fieldC;}

Create Table Z,

0 Type of object

1 integer (primary key) ,
2 string

3 integer (only derived)

what if more derived classes come?

Create Table Y,

1 integer (primary key) ,
2 string

Create Table Z,

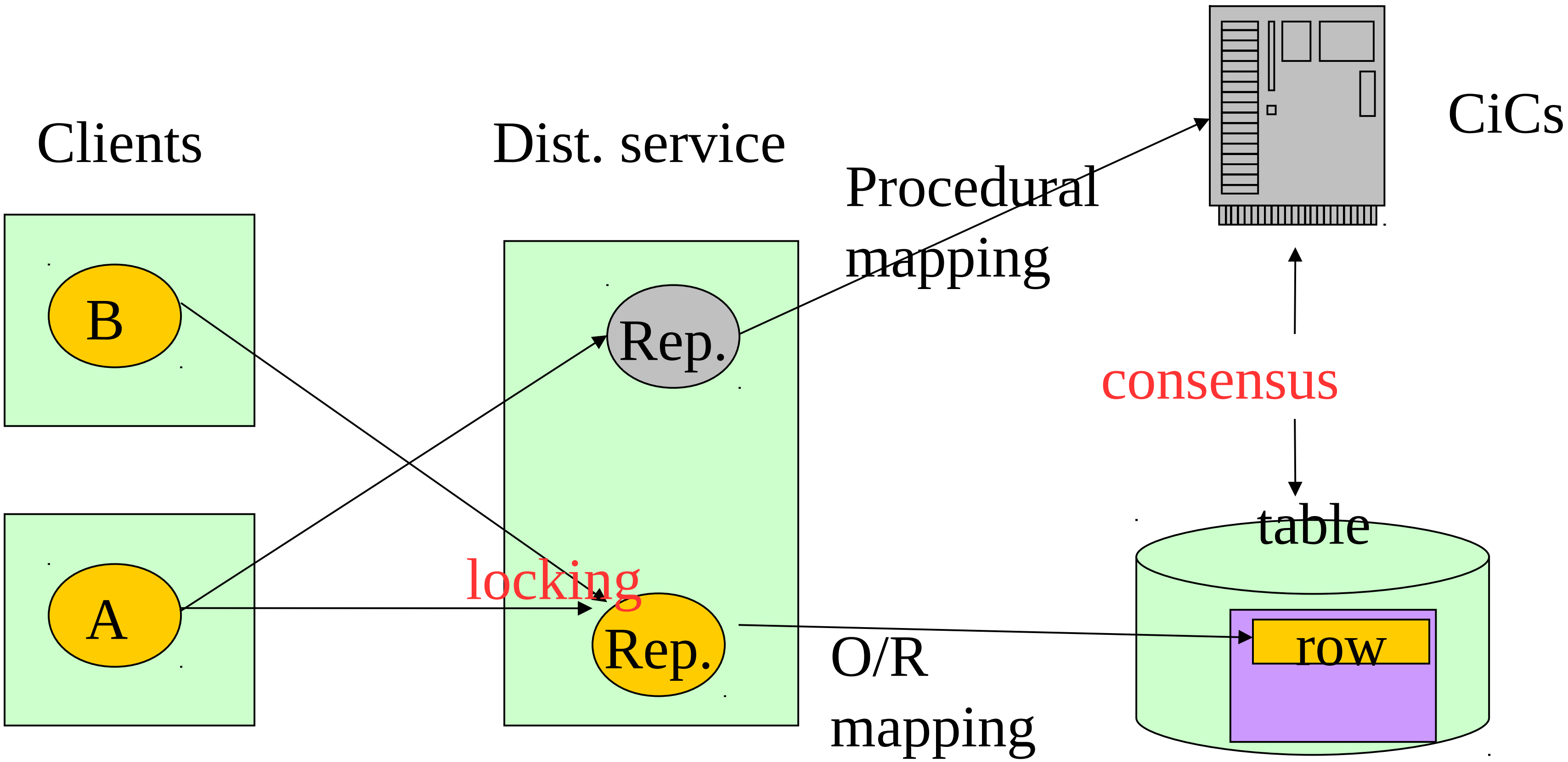
0 foreign key into Y

1 integer (primary key) ,

two table accesses needed

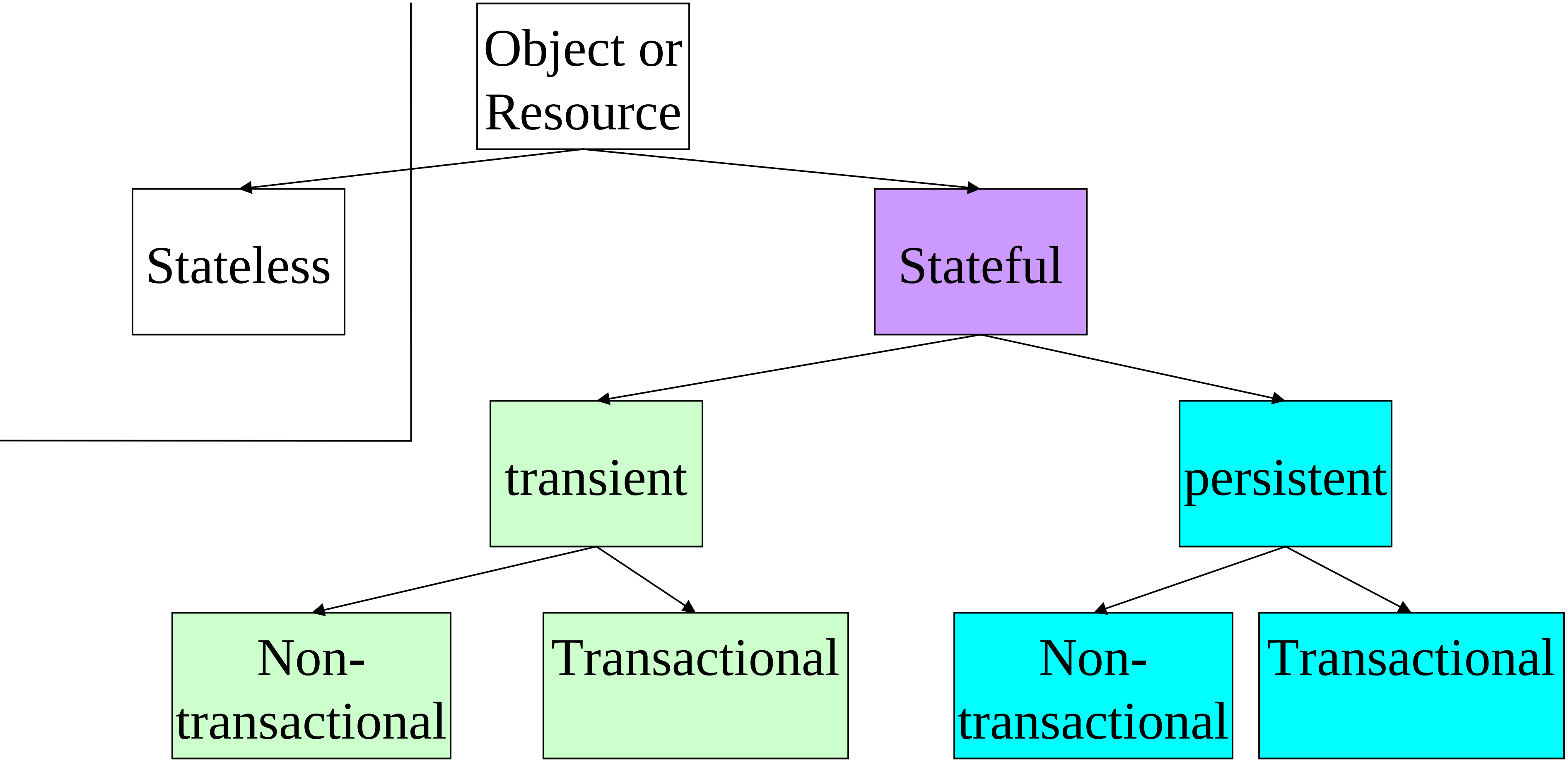
Inheritance creates difficult problems for table mapping. Either performance or flexibility suffer. EJB e.g. does not allow inheritance. A special problem is the extension of a type (class), i.e. to determine all the objects of a type.

Locking and Consensus



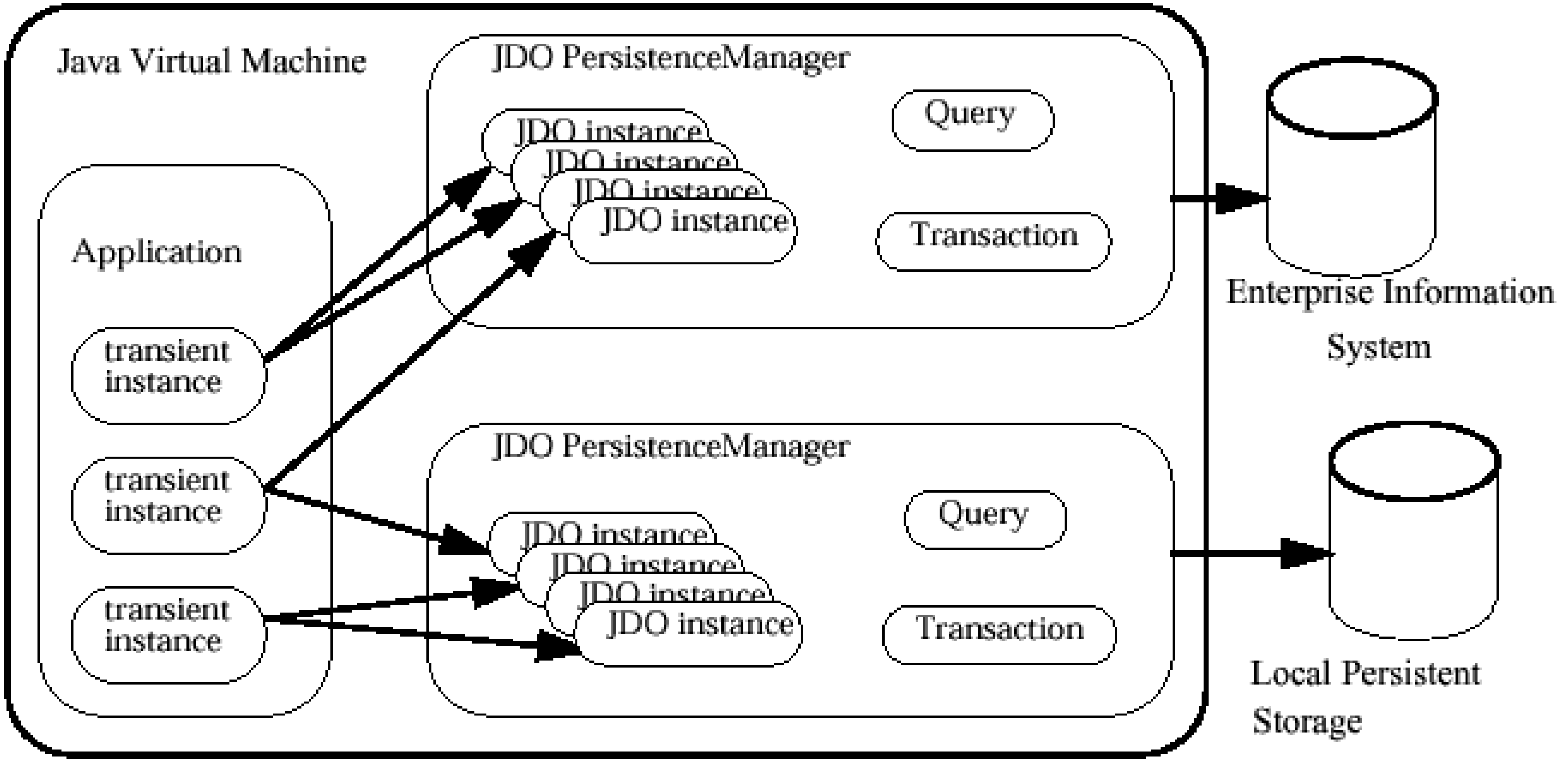
The diagram shows two problems: The yellow object is being shared between clients. Objects have state which needs to be consistent between calls. That's why we need **locking.** Client A uses two objects from different storage systems. Both systems need to agree about changes to achieve atomicity of a unit of work. That's why we need a 2PC **consensus** protocol.

Example: JDO Object Types



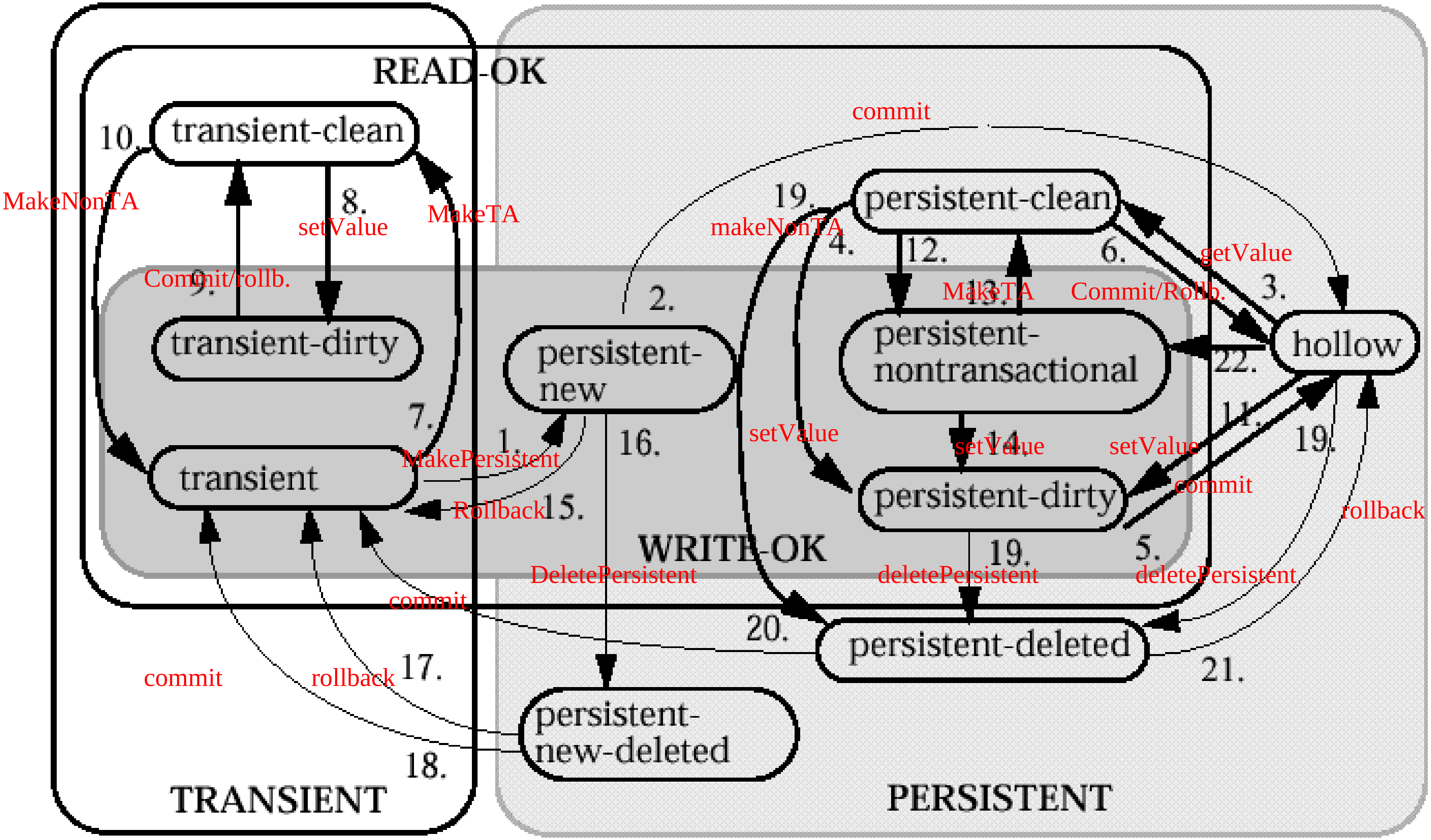
O/R mappers support transactional and non-transactional versions of stateful objects

JDO Architecture

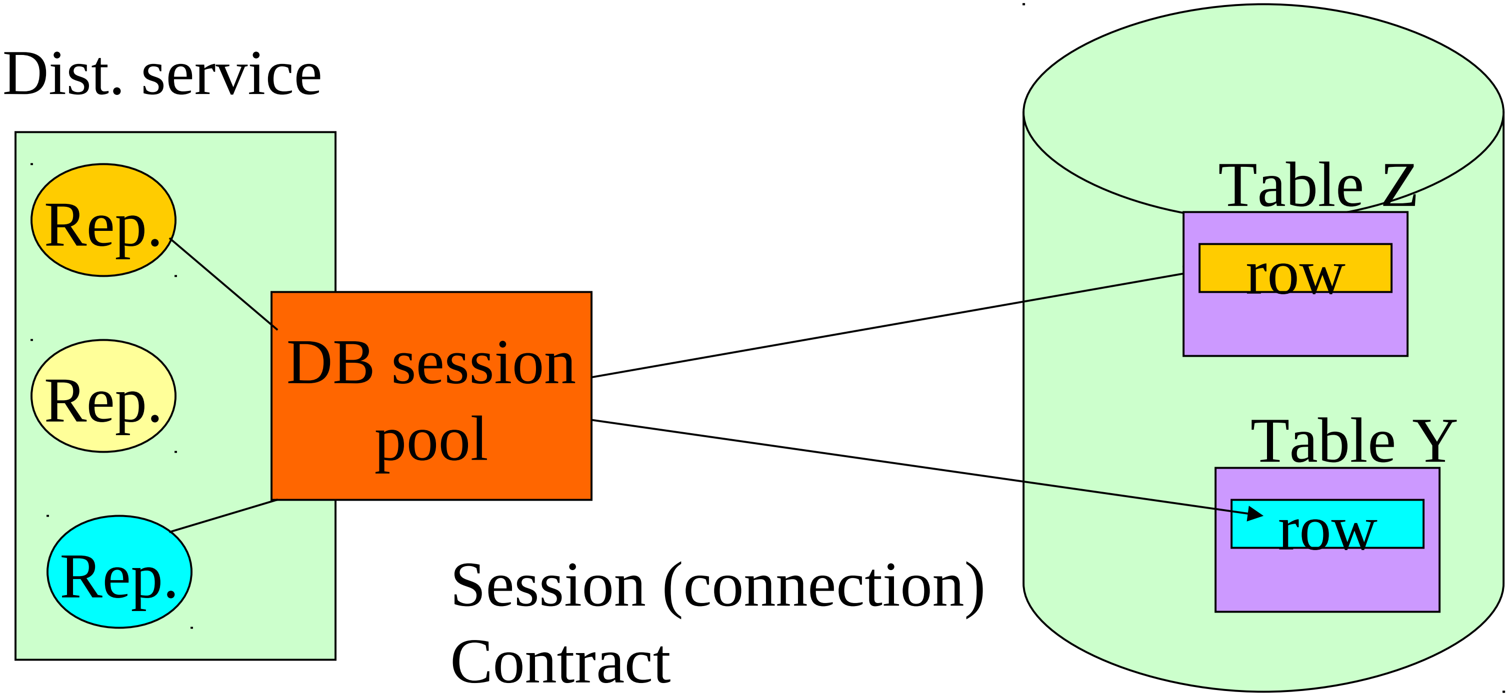


JDO’s are designed to work in a non-managed form (no application server) and a fully managed form. They are supposed to shield applications from different data sources and mapping problems

State Diagram of JDO lifecycle



BTW: Data Store Session Pooling



Dist. service

Rep.

Rep.

Rep.

DB session pool

Table Z

row

Table Y

row

Session (connection)
Contract

The number of channels to a data store is limited and if an object would directly allocate a session (channel) and not return it quickly, system throughput would become marginal. Also, session creation is expensive (security!). Now either clients ask a pool for a session or the container framework automatically allocates and returns sessions. Problems: timeouts, connection recycling,

Transactions

1. ACID
2. Transaction Models
3. Isolation Levels
4. Two-Phase-Locking: Isolation
5. Two-Phase Commit: Consensus

(From Peter Bailis, When ist “ACID” ACID? Rarely!
http://www.bailis.org/blog/when-is-acid-acid-rarely/

Architecture of a Database System:
http://research.microsoft.com/en-us/people/philbe/chapter1.pdf

Classic ACID Definitions

1. Did your PC crash and you lost the changes you made to a word file? The changes were not **DURABLE**
2. Did you move your birthday party to a new location on short notice but couldn’t catch all participants in time so some showed up at the old location and some at the new? Your re-schedule call wasn’t **ATOMIC**
3. Did you and a friend work on a shared file on a server and ended up with some of your changes and some of her changes in the file? Your application did not provide **ISOLATION** between yourself and your friend.
4. Your friend wants to take a day off and asks you to do some of her work on that day (check out a piece of software, modify it, test it, document it and check it in again). You do it (maybe with some more iterations (;-) and next day she starts a new task. You have observed **CONSISTENCY** of the tasks.

Transaction Properties and Mechanisms

Atomic changes over distributed resources → Consensus/Voting algorithm: two phase commit

Consistency → Observation of consistency constraints between objects (or “start consistent, end consistent)

Isolation from concurrent access → Locking mechanisms: 2 phase locking, hierarchical locking

Durability of changes → Transfer of changes to memory objects to persistent storage

Serializability and Isolation

The textbook definition of ACID Isolation is serializability (e.g., Architecture of a Database System, Section 6.2), which states that

the outcome of executing a set of transactions should be equivalent to some serial execution of those transactions.

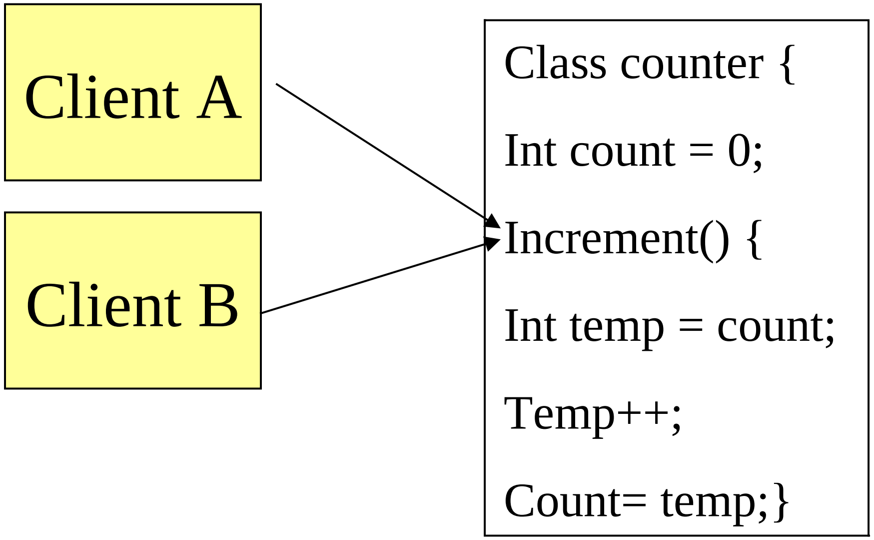
This means that each transaction gets to operate on the database as if it were running by itself, which ensures database correctness, or consistency. A database with serializability ("I" in ACID), provides arbitrary read/write transactions and guarantees consistency ("C" in ACID), or correctness, of the database. Without serializability, ACID, particularly consistency, is generally1 not guaranteed

(From Peter Bailis, When ist "ACID" ACID? Rarely!
http://www.bailis.org/blog/when-is-acid-acid-rarely/

Architecture of a Database System:
http://research.microsoft.com/en-us/people/philbe/chapter1.pdf

Locking

Protect distributed objects : lost updates



```
Class counter {
Int count = 0;
Increment() {
Int temp = count;
Temp++;
Count= temp;}
```

Client A calls increment(). Count is 0, temp becomes 0.

Client A’s thread has used it’s slice and is preempted.

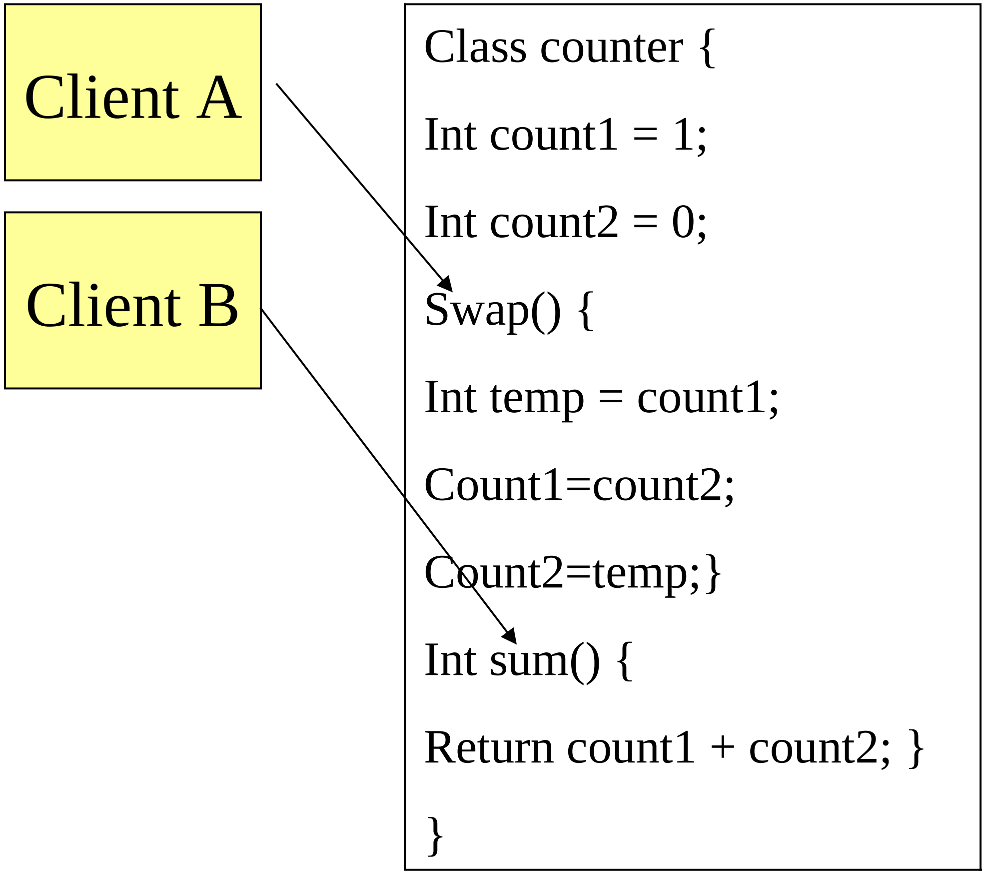
Client B calls increment(). Count is 0, temp becomes 0.

Client B adds 1 to temp and writes it back to count. Count is 1.

Now comes Client A again. Also adds 1 to temp and writes it back to count. Count is 1 and NOT 2 now. We’ve lost one update.

The lost update problem!
Would it help to use count++ ?

Protect distributed objects : inconsistent analysis



```
Class counter {
Int count1 = 1;
Int count2 = 0;
Swap() {
Int temp = count1;
Count1=count2;
Count2=temp;}
Int sum() {
Return count1 + count2; }
}
```

Client A calls swap(). After storing count1 in temp it is set to count2 (0).

Client A’s thread has used it’s slice and is preempted.

Client B calls sum(). Count1 is now 0, and count2 is still 0.

Client B comes back from sum() with result 0.

Now comes Client A again. Writes temp back to count2 . Count2 is now 1 but sum() has reported 0 for both. The analysis of sum() is wrong.

The inconsistent analysis problem!

Use of locking against concurrent access

- Binary locks: e.g. synchronize(object). Will block all clients except of one.

- Modal locks (read lock, write lock): Clients who only want to read can get read locks – many concurrent read locks are possible.

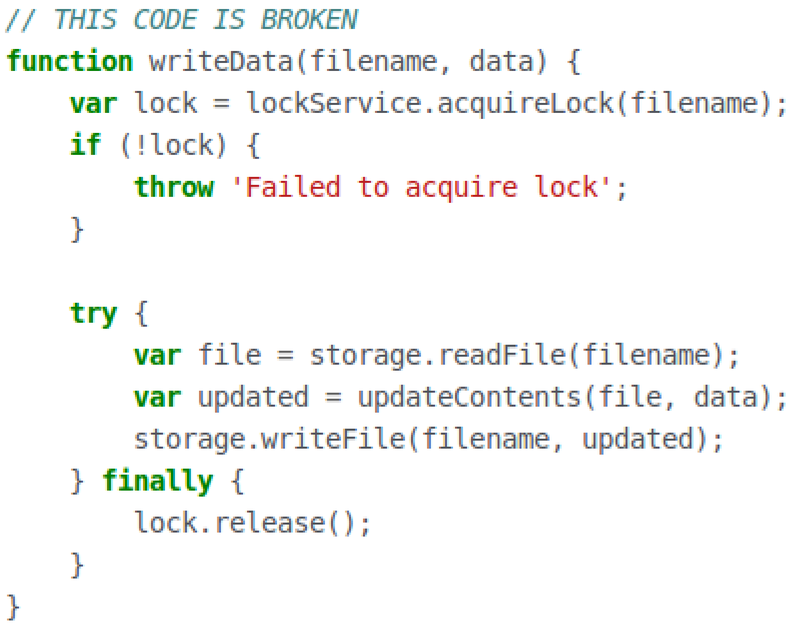
Binary locks are very simple to use but performance suffers badly because they cannot distinguish between reads and writes.

Lock compatibility matrix

	Read lock	Write lock
Read lock	OK	NO
Write lock	NO	NO

The concurrency service will not allow concurrent locks other than read locks. A write lock will exclude all other locks.

Time-Based Leases: Redlock-Algorithm



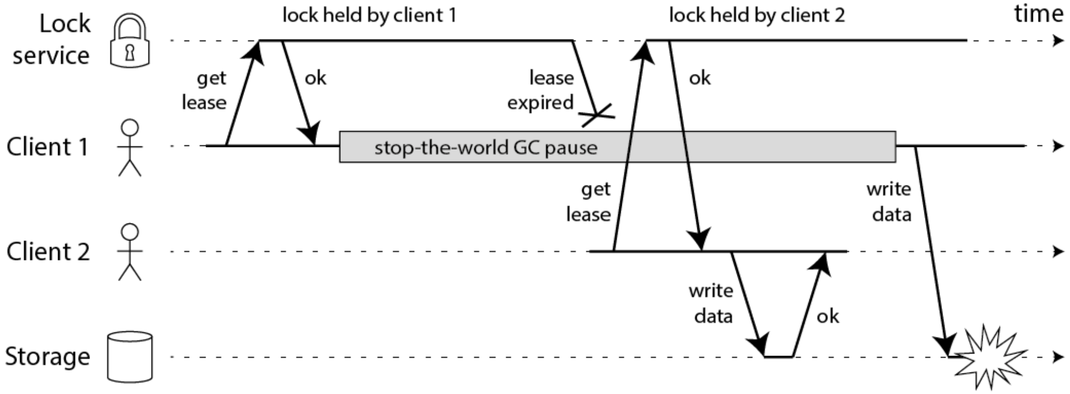
```
// THIS CODE IS BROKEN
function writeData(filename, data) {
    var lock = lockService.acquireLock(filename);
    if (!lock) {
        throw 'Failed to acquire lock';
    }

    try {
        var file = storage.readFile(filename);
        var updated = updateContents(file, data);
        storage.writeFile(filename, updated);
    } finally {
        lock.release();
    }
}
```

The redlock algorithm from Redis uses time-based leases for liveness reasons. From: M.Kleppmann, Designing Data-intensive Applications.

https://martin.kleppmann.com/2016/02/08/how-to-do-distributed-locking.html

(Think asynchonous/partial synchronous systems)

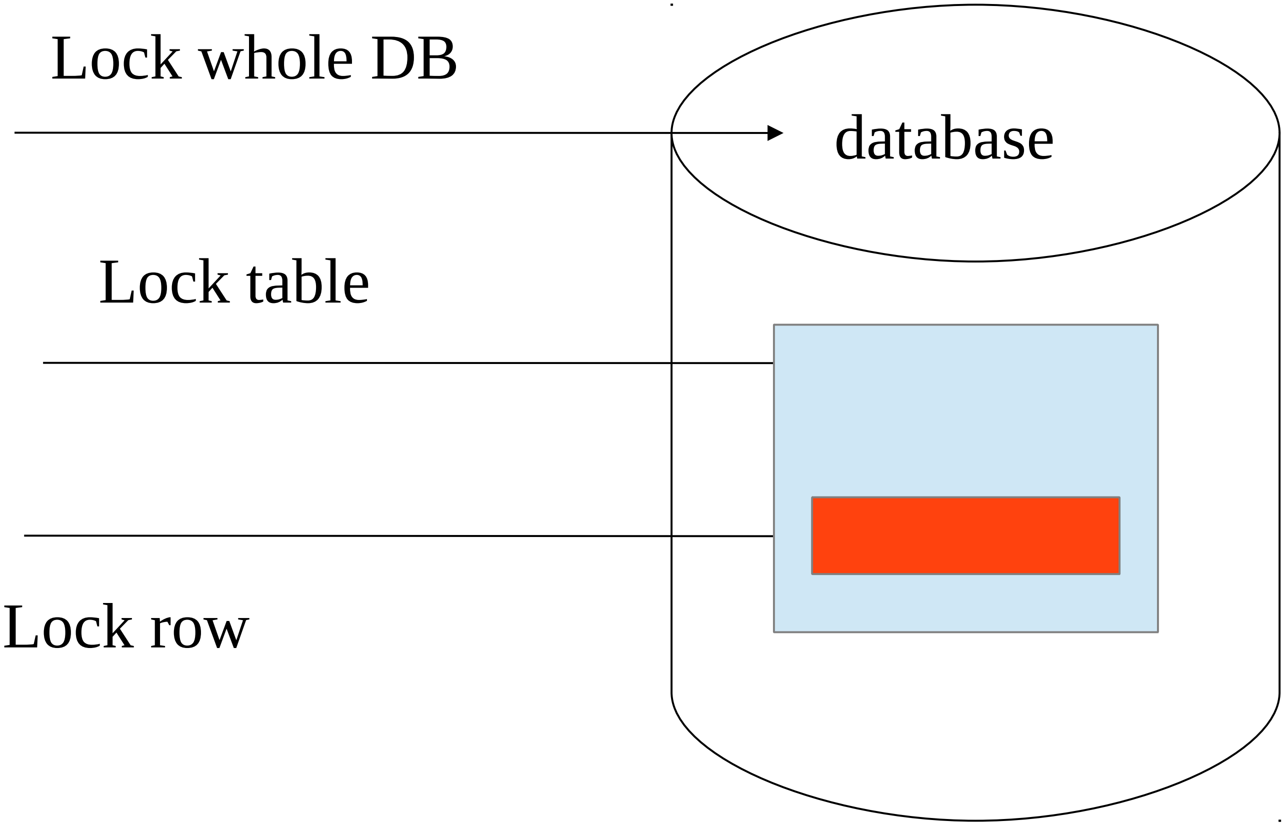
Time-Based Leases: Redlock-Algorithm



Without “fencing” (e.g. sequence number), storage cannot detect expired leases.

https://martin.kleppmann.com/2016/02/08/how-to-do-distributed-locking.html

Lock granularity



Lock whole DB

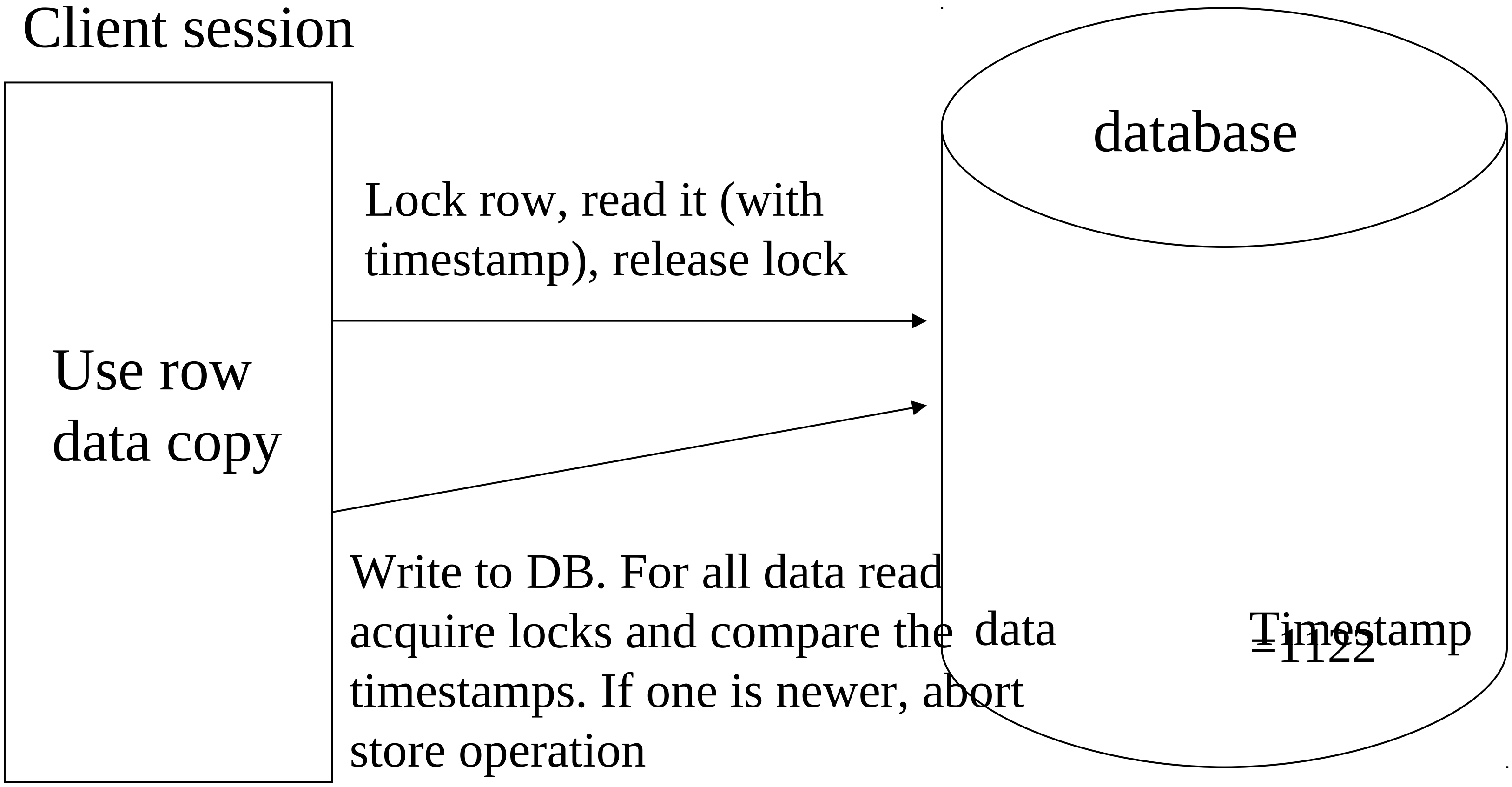
database

Lock table

Lock row

Besides lock mode the granularity of locks will determine overall throughput. The smaller the better.

Optimistic Locking



Client session

Use row data copy

Lock row, read it (with timestamp), release lock

database

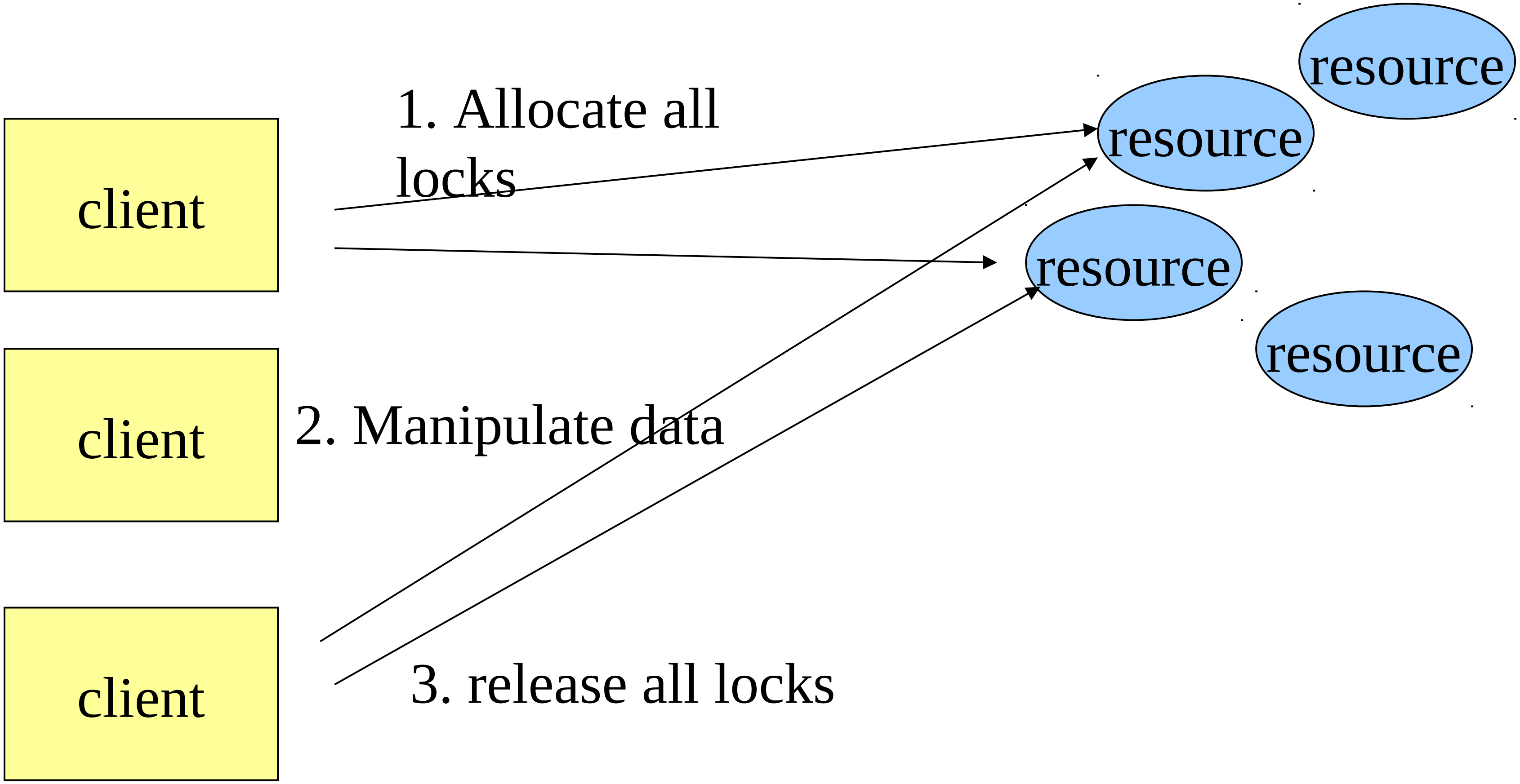
Write to DB. For all data read acquire locks and compare the timestamps. If one is newer, abort store operation

data

Timestamp =1122

Overall throughput is better because locks are held only a very short time. The timestamp compare logic should be a framework mechanism of the client session objects.

Serializability with Two-phase locking



client

client

client

resource

resource

resource

resource

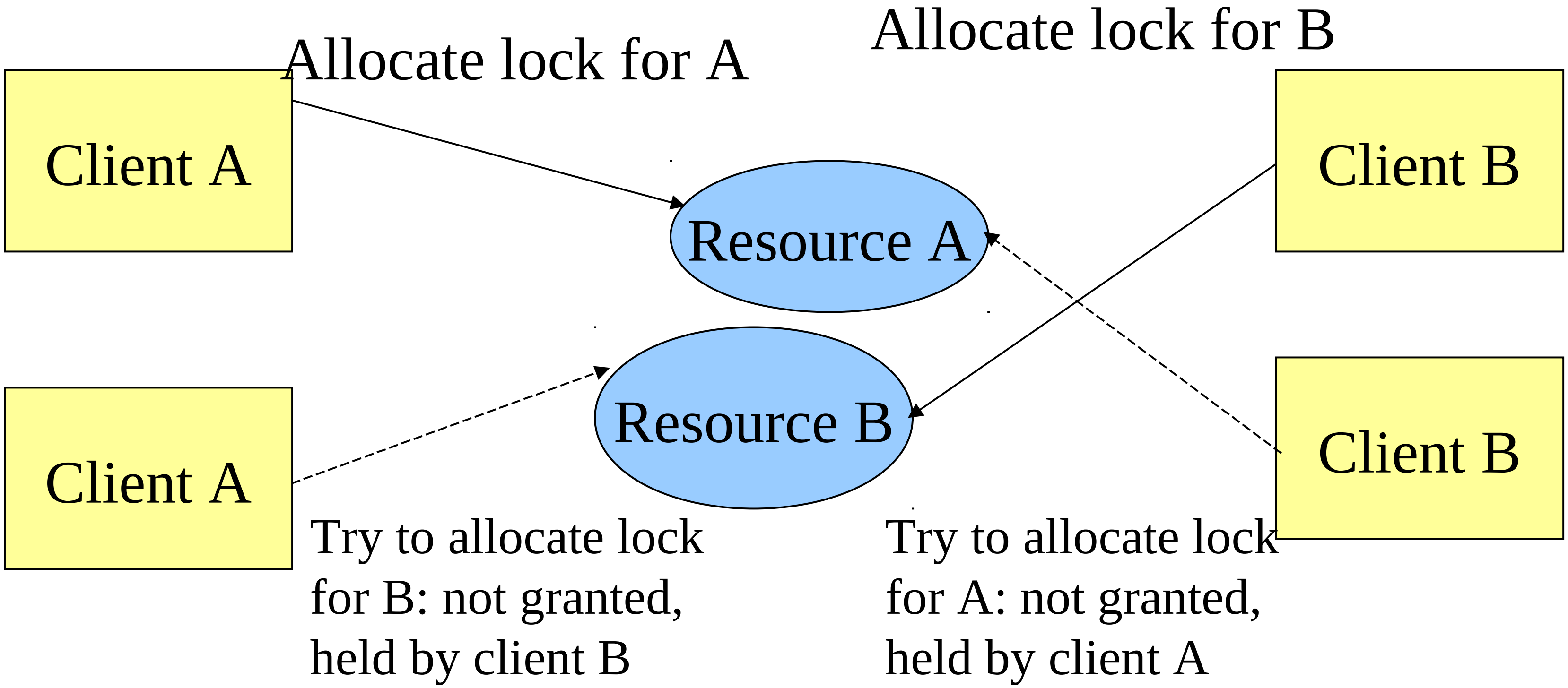
1. Allocate all locks

2. Manipulate data

3. release all locks

A basic requirement for the 2-phase locking protocol is that all locks are allocated first. After the first lock is released NO other locks may be acquired! This will guarantee serializability

Deadlocks



Allocate lock for A

Allocate lock for B

Client A

Resource A

Client B

Resource B

Client A

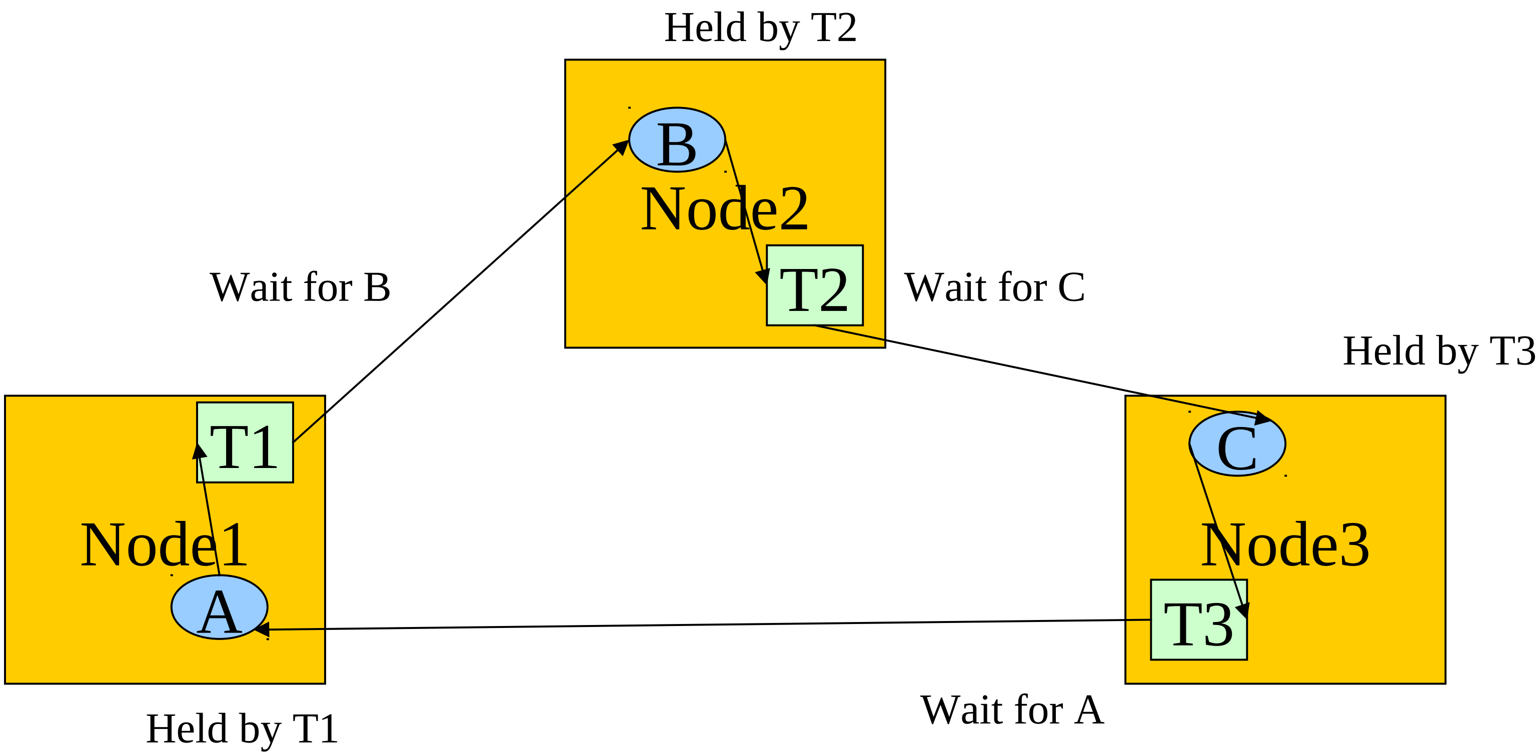
Client B

Try to allocate lock
for B: not granted,
held by client B

Try to allocate lock
for A: not granted,
held by client A

Deadlocks can be detected (e.g. by a database). To prevent deadlocks, always allocate resource locks IN THE SAME ORDER. Process termination must release all locks held by a process.

Distributed Deadlocks



Held by T2

B

Node2

T2

Wait for B

Wait for C

Held by T3

T1

Node1

A

C

Node3

T3

Wait for A

Held by T1

A distributed deadlock does not show locally. How can it be detected?

Exercise:Distributed Deadlocks Detection

Find ways to detect a DD and discuss
a) correctness
b) liveness
c) cost/complexity
d) failure model
e) architecture type

Of your solution.

Some hints: local wait-for-graph, detection server, distributed edge chasing algorithms, stochastic. (from Coulouris et.al. Page 535).

Distributed Transactions

Transaction API

Client:

System is in consistent state

Begin Transaction

Modify objects

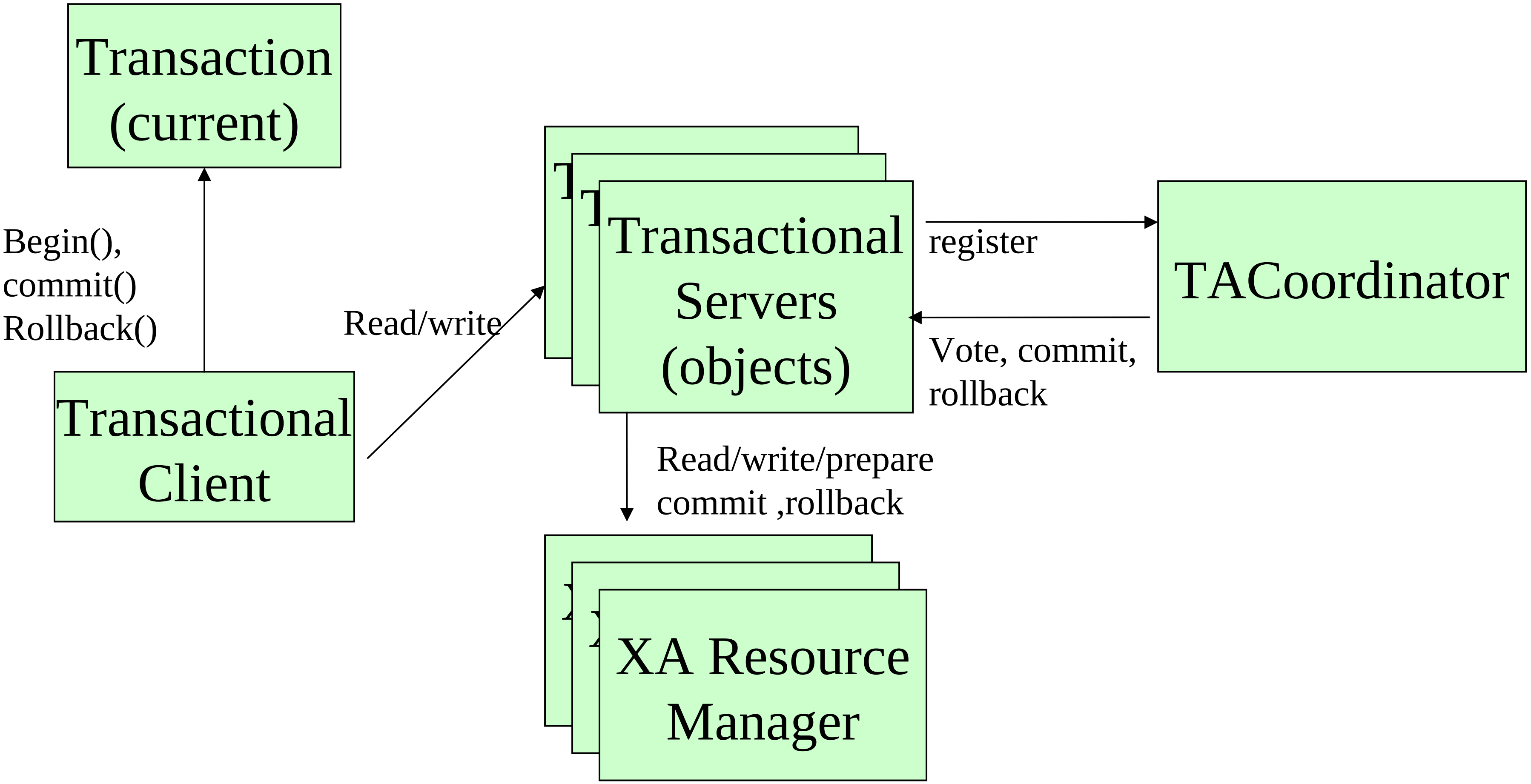
Commit Transaction

System has new, consistent state, all local objects now invalid. The changes are VISIBLE to others.

On Error, either the system or the client can do a “rollback” which takes system state back to the beginning of the TA

Only in case of a successful commit operation becomes the new state durable and visible to others. Please note that “rollback” really means going back to the beginning COMPLETELY. Theoretically the client does not even KNOW that she tried an operation and even log files would have to be cleaned!

Components of distributed transactions



Every resource that implements the XA interface can participate in distributed transactions.

Service Context

Some services need so-called context information to flow with a call. Two prominent ones are:

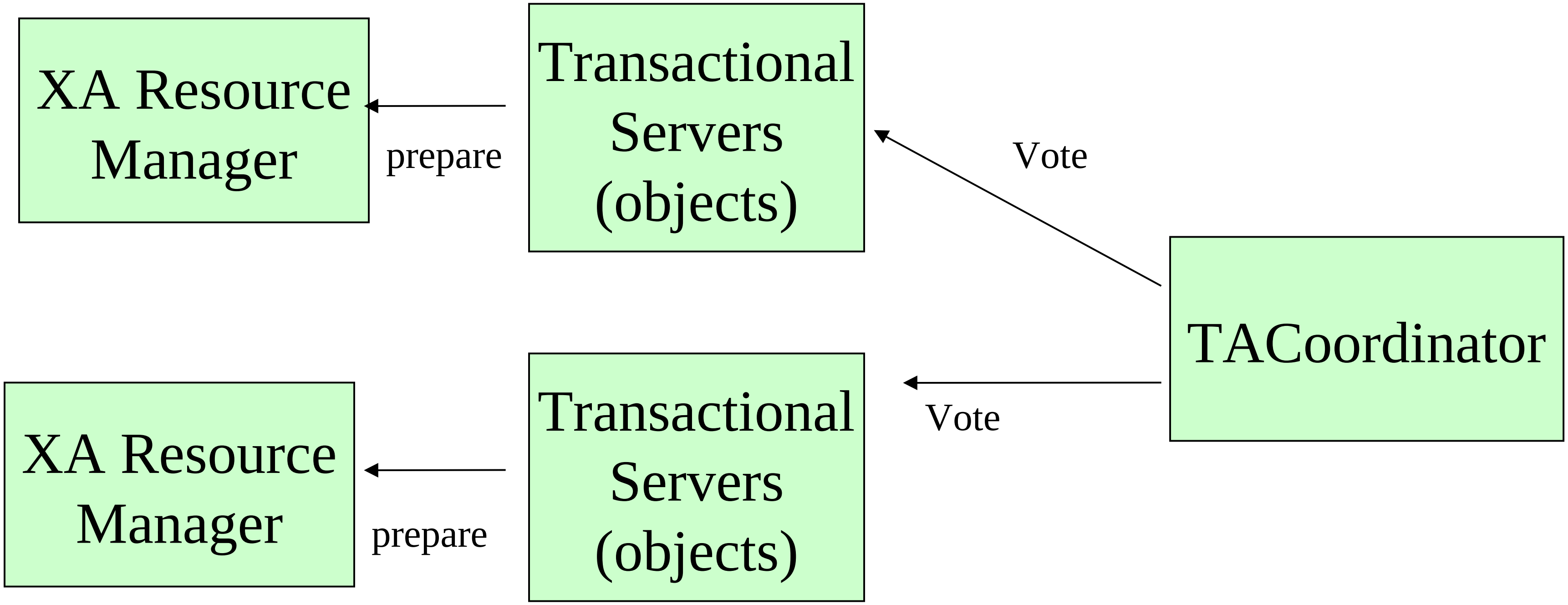
Security (needs to “flow” user information, access rights etc.)

Transactions (need to flow information about on-going transactions to participants)

This additional information needs to be standardized if different vendor implementations of services should interoperate.

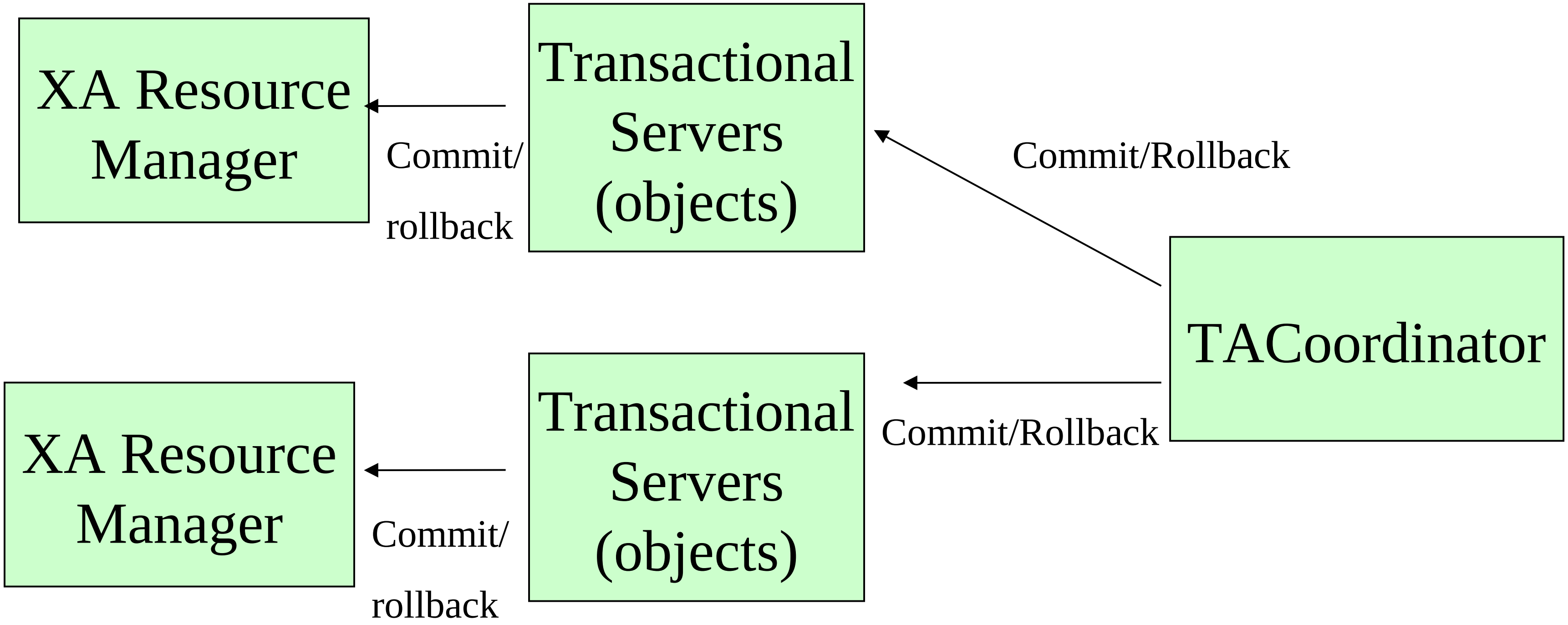
Do you know other “context related” design problems?

Distributed Two-Phase Commit: Vote



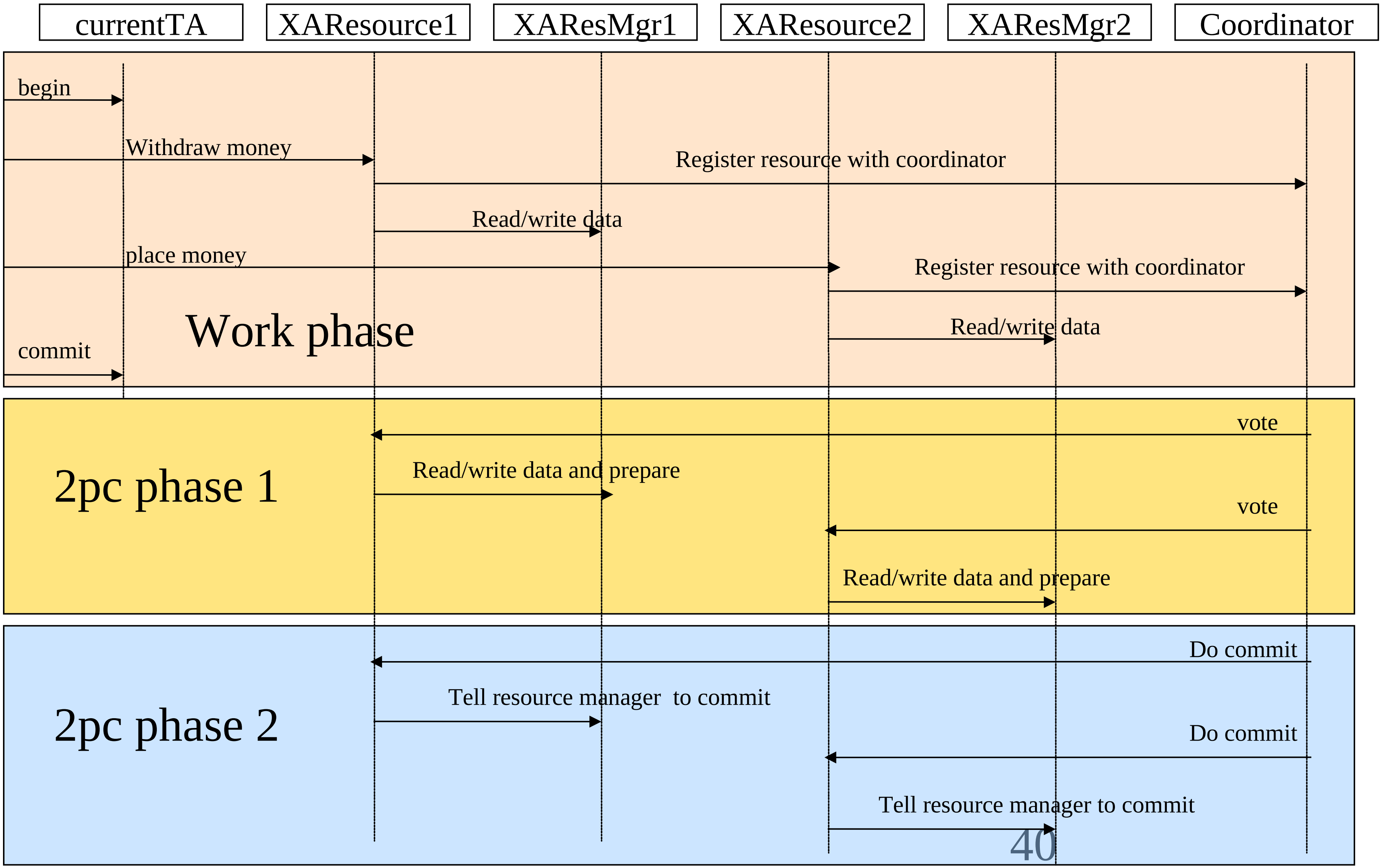
The only way to achieve “atomic” operations in a distributed setting is to ask all the participants. After a client called “commit()” the TA-Coordinator asks all objects which are part of the TA to vote on either a commit or a rollback. The objects in turn ask the resource managers (e.g. DBs) to “prepare” for a commit. After successful return of a prepare the object AND the resource manager have promised to commit the changes if the coordinator sends a commit.

Distributed Two-Phase Commit: Completion



ONLY the coordinator can either commit or abort a TA after the prepare phase. It will call for a commit if the vote phase was successful and all participants have prepared for a following commit. If an error occurred (e.g. a participant was unreachable) the coordinator will call for a rollback.

Example of distributed transactions



currentTA
XAResource1
XAResMgr1
XAResource2
XAResMgr2
Coordinator

begin
Withdraw money
Register resource with coordinator
Read/write data
place money
Register resource with coordinator
Read/write data
commit

Work phase

vote
Read/write data and prepare
vote
Read/write data and prepare

2pc phase 1

Do commit
Tell resource manager to commit
Do commit
Tell resource manager to commit

2pc phase 2

Failure models in distributed TA’s

Work phase:

- A participant crashes or is unavailable in work phase.

The coordinator will call for a rollback.

- The client crashes in work phase (commit is not called). Coordinator will finally time-out the TA and call rollback.

Voting Phase:

- If a resource becomes unavailable or has other problems, the coordinator will call rollback

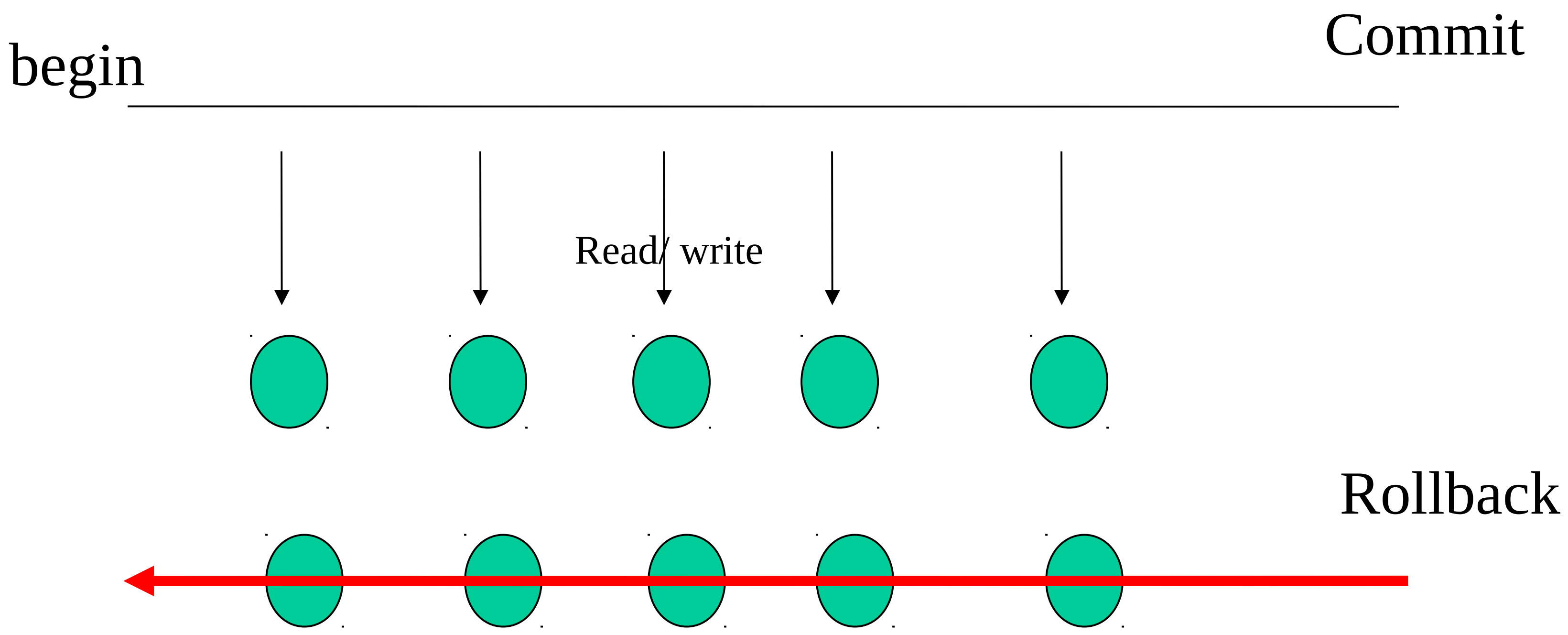
Commit Phase: (server uncertainty)

- a crashed server will consult the coordinator after re-start and ask for the decision (commit or rollback)

Special problems of distributed TA’s

- Resources: Participants in distributed TA’s use up many system resources due to logging all actions to temporary persistent storage. Also considerable parts of a system may get locked during a TA.
- Coordinator – a single point of failure? Even the coordinator must prepare for a crash and log all actions to temporary persistent storage.
- Heuristic outcomes for transactions. Under certain circumstances the outcome of a transaction may only follow a certain heuristic because the real outcome could not be determined. (see exercises)

Transaction Types: flat TA’s



begin

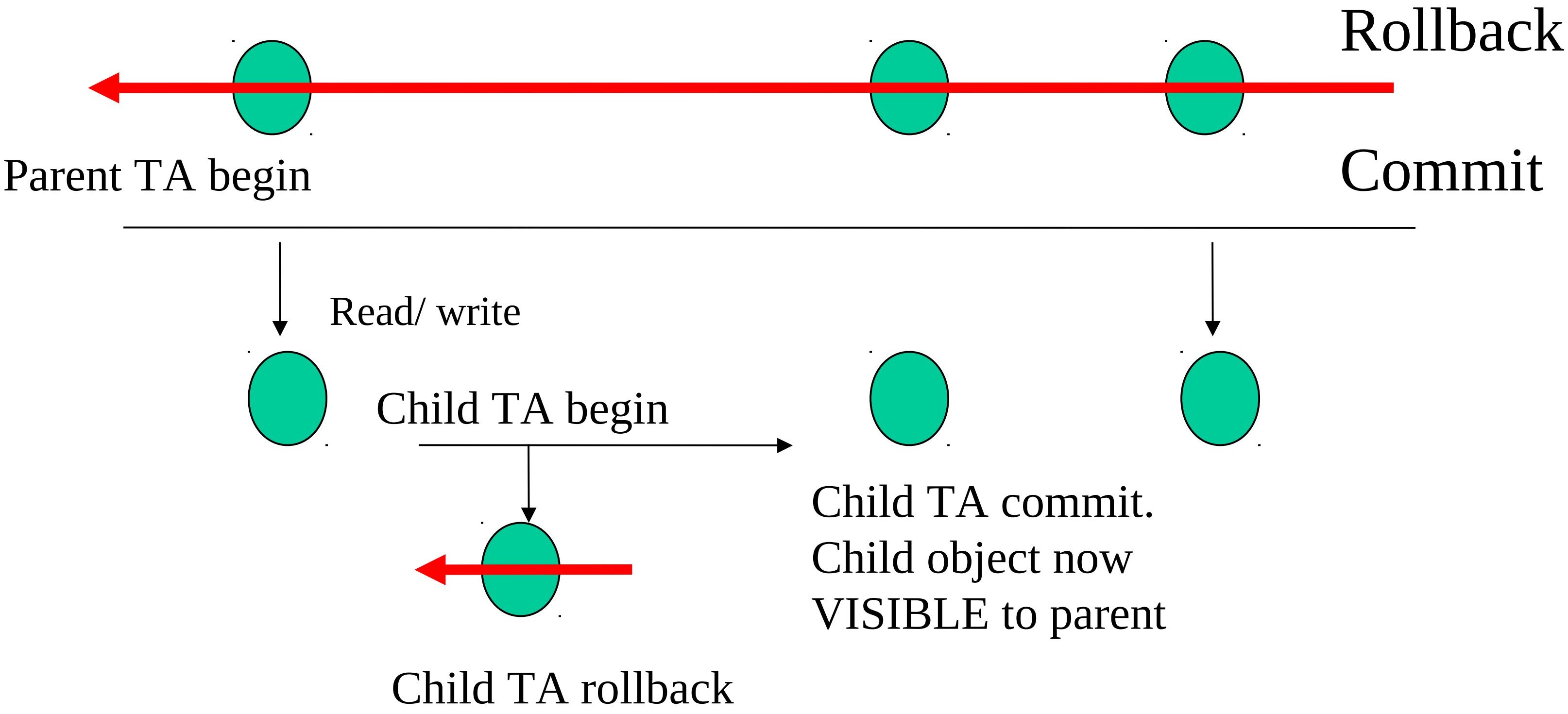
Commit

Read/ write

Rollback

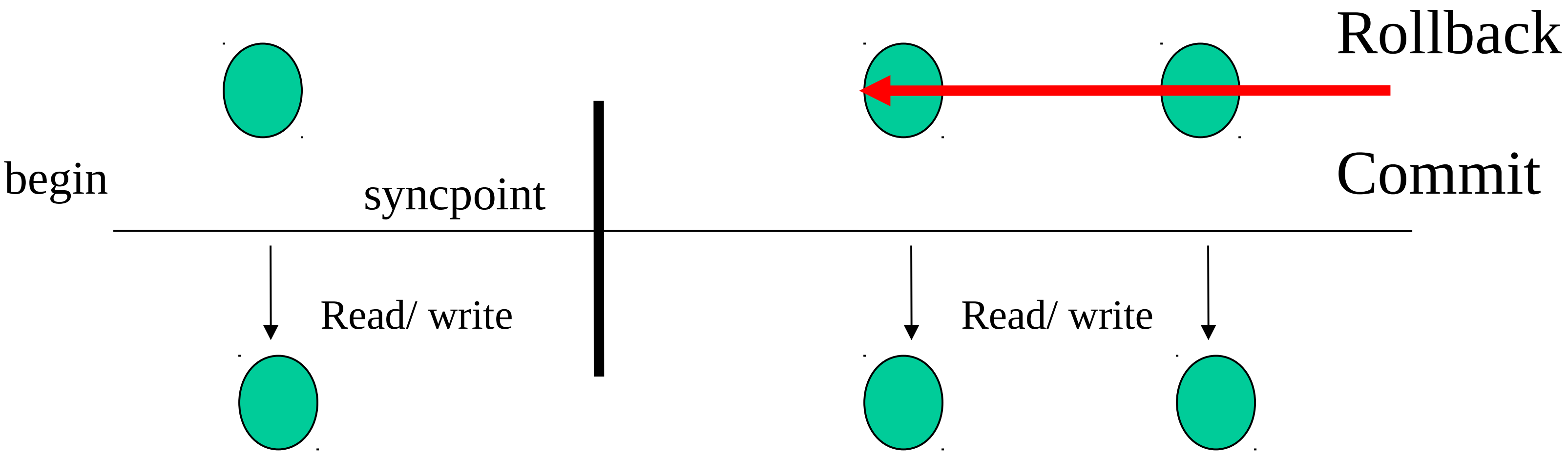
Flat TA’s show the all-or-nothing characteristics of transactions best. ANY failure will cause a complete rollback to the original state. If many objects have been handled this can lose quite a lot of work.

Transaction Types: nested TA's



Nested transactions allow partial rollbacks with a parent transaction. A child TA rollback does not affect the parent TA. But a parent TA rollback will return ALL participants to their initial state. Example: allocation of a travel plan: hotel, flight, rental-car, trips etc. The whole TA should not be aborted only because a certain rental car is not available.

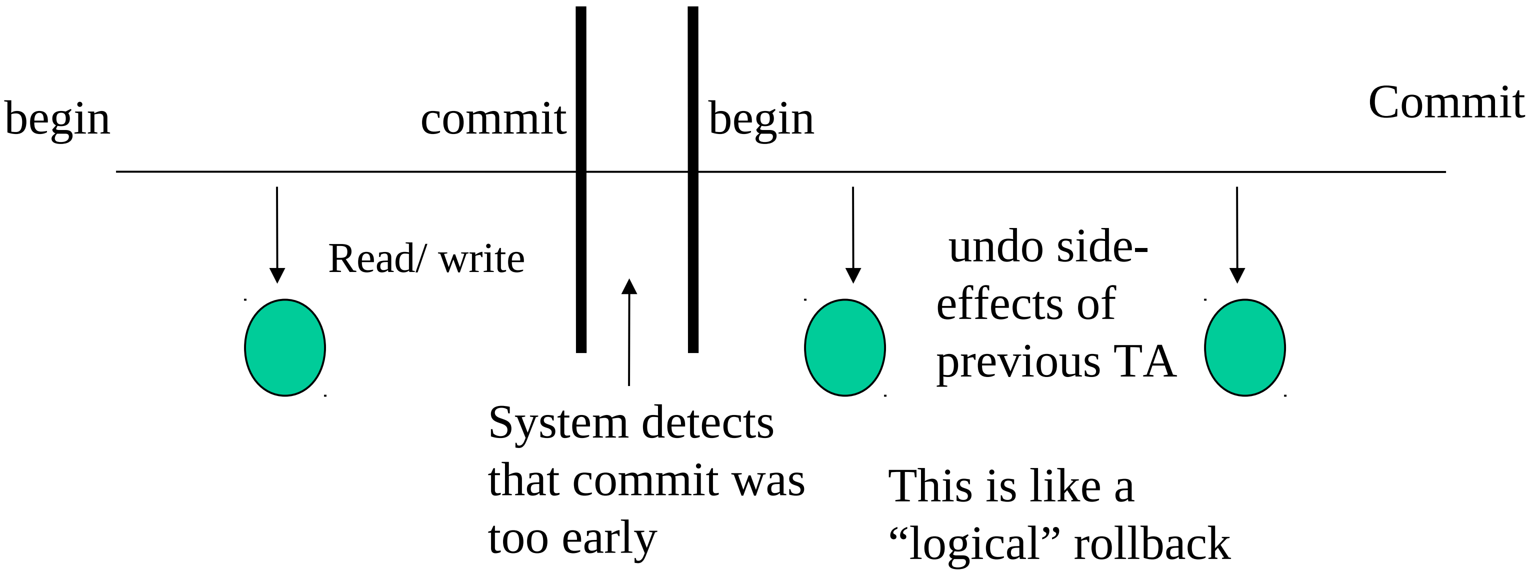
Transaction Types: long-running TA’s



A rollback will
only go back to the
checkpoint state

A problem of long-running transactions is resource allocation as well as the increasing amount of work that would be lost in case of a rollback. Syncpoints move the fallback position forward towards the commit point.

Transaction Types: Compensating TA’s



begin

commit

begin

Commit

Read/ write

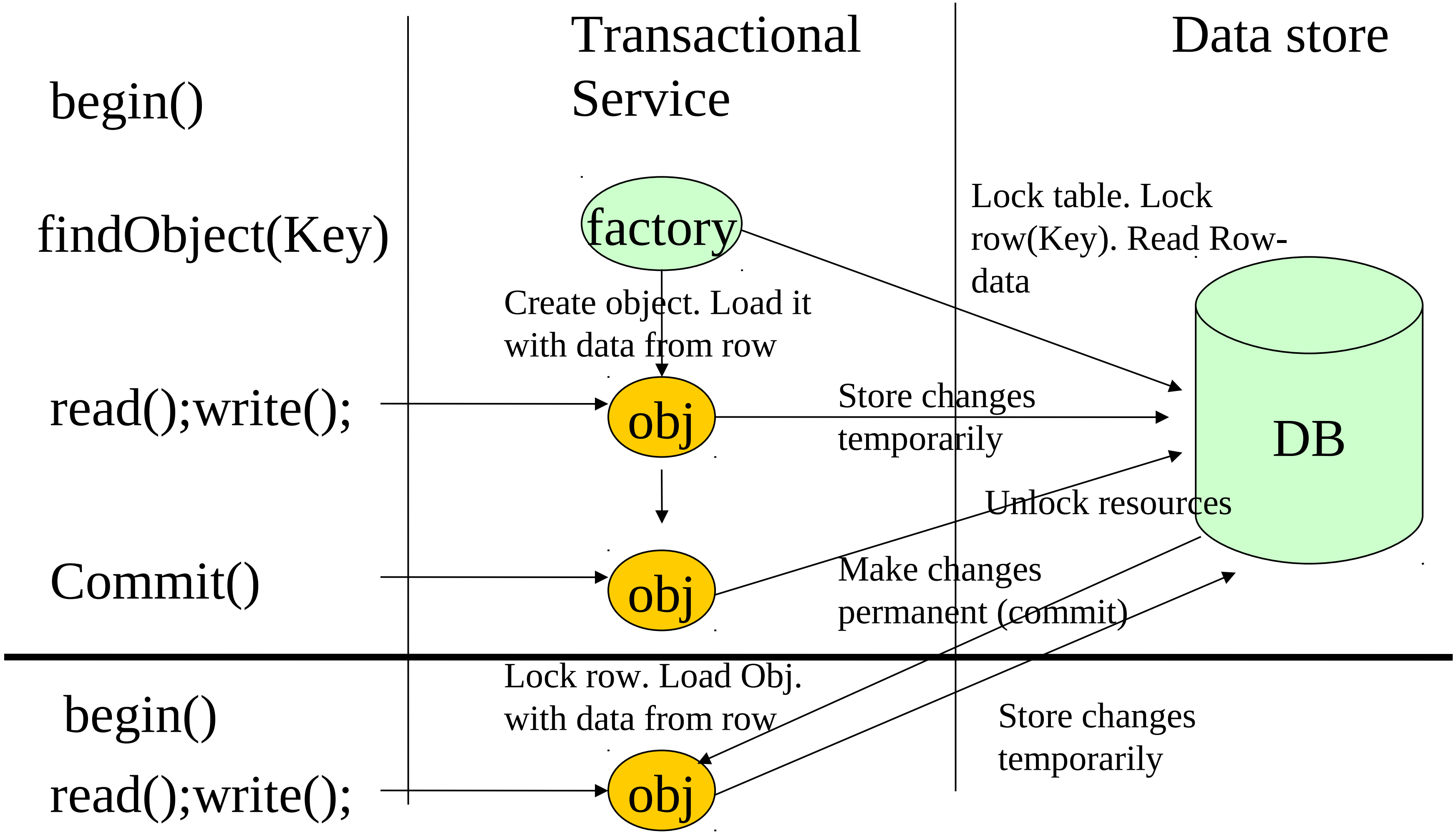
undo side-effects of previous TA

System detects that commit was too early

This is like a “logical” rollback

Transaction throughput increases if objects become visible sooner –e.g. through a lose interpretation of the ISOLATION property. Now we need to COMPENSATE for the previous TA (which can no longer be rolled back). It depends on the application whether such compensating transactions are possible. Compensating transactions are also hand-coded if no transaction monitor/manager is available.

The interplay of transactions and persistence



Transactional Service

Data store

begin()

findObject(Key)

factory

Lock table. Lock row(Key). Read Row-data

Create object. Load it with data from row

read();write();

obj

Store changes temporarily

DB

Unlock resources

Commit()

obj

Make changes permanent (commit)

begin()

Lock row. Load Obj. with data from row

Store changes temporarily

read();write();

obj

The quality of the locks held during a TA is defined through “Isolation levels” in the Resource Manager. Please note that at the beginning of a new TA existing objects are RE-LOADED!

Transaction Isolation Levels: ANSI

The ANSI/ISO SQL standard defines four levels of transaction isolation in terms of three phenomena that must be prevented between concurrent transactions.

dirty reads: A transaction reads data written by concurrent uncommitted transaction.

non-repeatable reads: A transaction re-reads data it has previously read and finds that data has been modified by another transaction (that committed since the initial read).

phantom read : A transaction re-executes a query returning a set of rows that satisfy a search condition and finds that the set of rows satisfying the condition has changed due to another recently-committed transaction.

(From the Postgres manual), see also IBM Knowledge center
“Isolation Levels”

Isolation Levels Explained

Dirty Write (P0): w1(x) … w2(x) Prohibited by RU, RC, RR, 1SR
Dirty Read (P1): w1(x) … r2(x) Prohibited by RC, RR, 1SR
Fuzzy Read (P2): r1(x) … w2(x) Prohibited by RR, 1SR
Phantom (P3): r1(P) … w2(y in P) Prohibited by 1SR

There's a neat kind of symmetry here: P1 and P2 are duals of each other, preventing a read from seeing an uncommitted write, and preventing a write from clobbering an uncommitted read, respectively. P0 prevents two writes from stepping on each other, and we could imagine its dual r1(x) … r2(x)–but since reads don't change the value of x they commute, and we don't need to prevent them from interleaving. Finally, preventing P3 ensures the stability of a predicate P, like a where clause–if you read all people named "Maoonga", no other transaction can sneak in and add someone with the same name until your transaction is done.

"If you're having trouble figuring out what these isolation levels actually allow, you're not alone. The anomalies prevented (and allowed!) by Read Uncommitted, Read Committed, etc are derived from specific implementation strategies. If you use locks for concurrency control, and lock records which are written until the transaction commits (a "long" lock), you prevent P0. If you add a short lock on reads (just for the duration of the read, not until commit time), you prevent P1. If you acquire long locks on both writes and reads you prevent P2, and locking predicates prevents P3. The standard doesn't really guarantee understandable behavior–it just codifies the behavior given by existing, lock-oriented databases." From: https://aphyr.com/posts/327-call-me-maybe-mariadb-galera-cluster (Kyle Kingsbury)

Also see his excellent "jepsen" blog, where he analyzes distributed systems correctness in popular products (https://aphyr.com/tags/Jepsen). A generalized model – independent of implementation – is given in Adya, Liskov, O'Neil, Generalized Isolation Level Defintions, (2000) http://bnrg.cs.berkeley.edu/~adj/cs262/papers/icde00.pdf

Transactions and Isolation Levels: Snapshot Isolation and MVCC

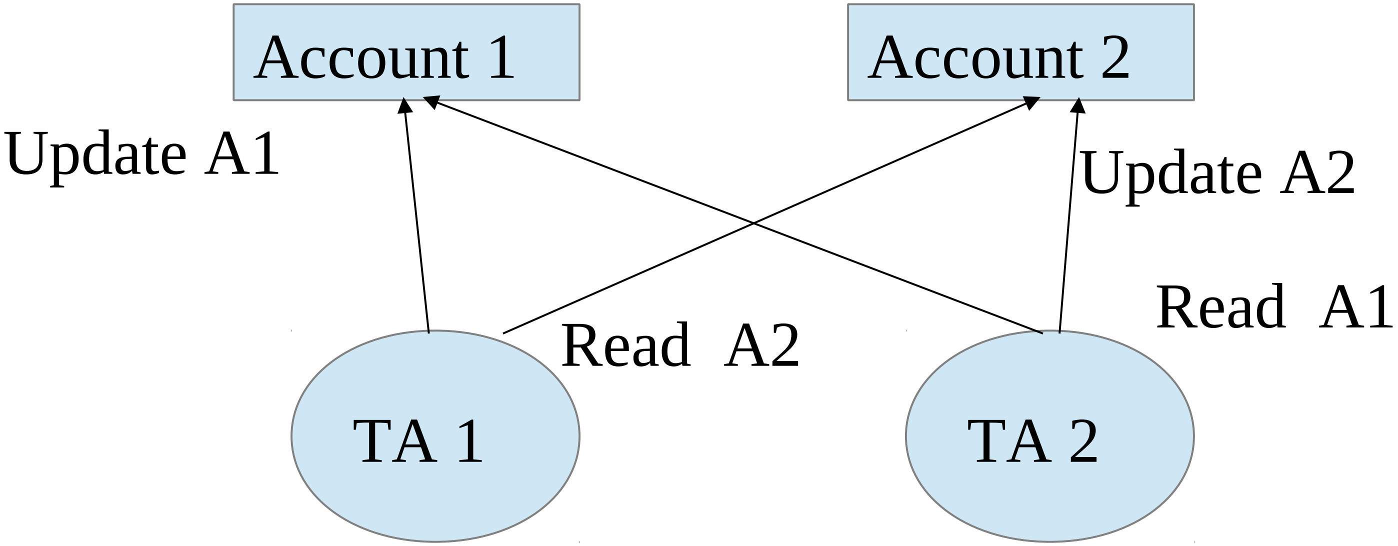
Snapshot Isolation works by guaranteeing a consistent snapshot on all reads when a transaction starts (basically by retrieving the last committed version). Updates will then only be committed, if there is no conflict with concurrent updates.

In effect, multiple concurrent versions exist. Snapshot Isolation does not exhibit inconsistencies described by ANSI, but it is NOT serializable.

Possible effects are write skew anomalies. MVCC based Tas allow a much higher concurrency rate, mostly due to the fact, that read locks are not held and there is no re-verification of values read during the TA.

See wikipedia and M.Herlihy for MVCC, https://aphyr.com/posts/327-call-me-maybe-mariadb-galera-cluster (Kyle Kingsbury) for Snapshot Isolation (Oracle)

Write Skew Anomalies



Account 1

Account 2

Update A1

Update A2

Read A1

Read A2

TA 1

TA 2

TA 1's update on Account 1 depends on information from Account 2. Account 2 changes during TA 1. Both transactions commit on stale read values but the updates do not conflict.

Solution: forced serializability e.g. with “Select for update” command. Does this work for distributed TA's???

Transaction Isolation Levels: Implementations

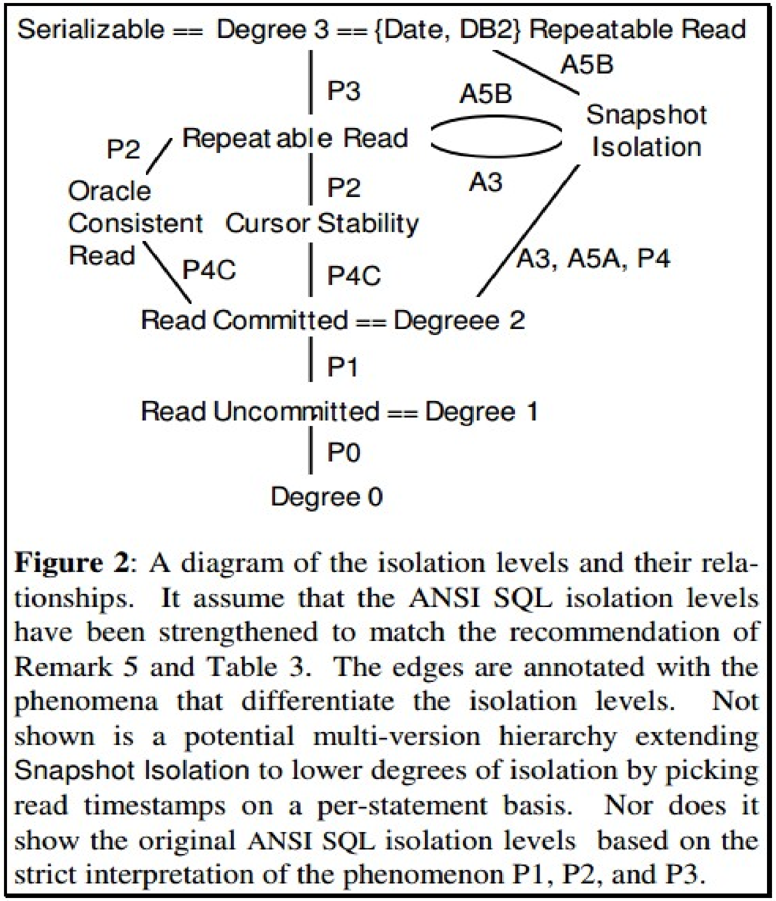


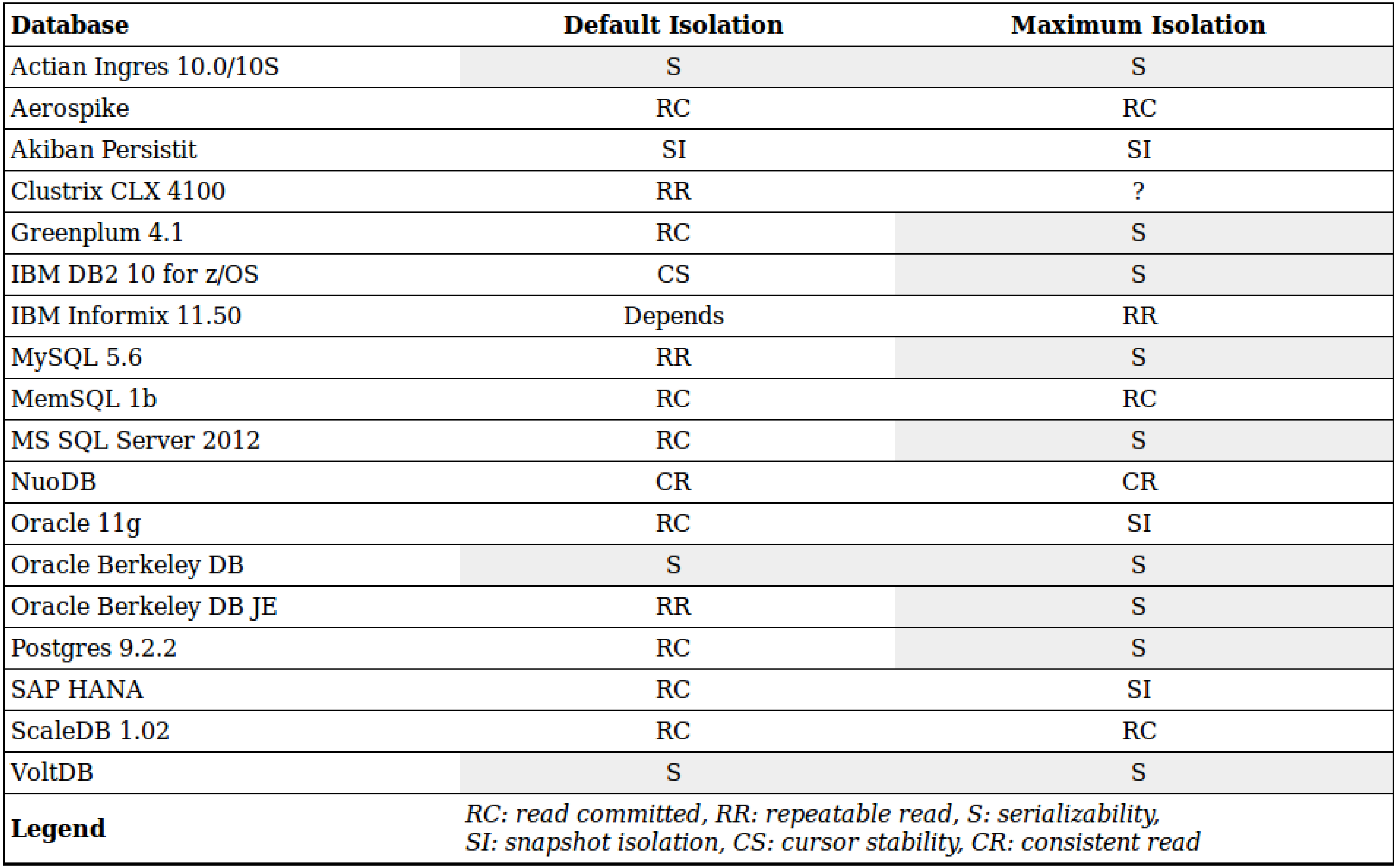
Figure 2: A diagram of the isolation levels and their relationships. It assume that the ANSI SQL isolation levels have been strengthened to match the recommendation of Remark 5 and Table 3. The edges are annotated with the phenomena that differentiate the isolation levels. Not shown is a potential multi-version hierarchy extending Snapshot Isolation to lower degrees of isolation by picking read timestamps on a per-statement basis. Nor does it show the original ANSI SQL isolation levels based on the strict interpretation of the phenomenon P1, P2, and P3.

Again from: https://aphyr.com/posts/327-call-me-maybe-mariadb-galera-cluster.
“SNAPSHOT ISOLATION lies between Read Committed and Serializable, but is neither a superset nor subset of Repeatable Read.”

Transaction Isolation Levels: Phenomena

Isolation Level/Effects	Dirty Read (cells read change during TA. Locked cells can be read)	Non-Repeatable Read (rows change during TA, changed or locked rows can be read)	Phantom Read	Write skew anomalies
Read uncommitted	possible	possible	possbile	possible
Read commited	NO	possible	possible	NO/Yes, depending on implementation
Repeatable read Read stability	NO/NO	NO/NO	NO/possible	NO (no access to locked cells)
Serializable	NO	NO	NO	NO (linearized)
Snapshot Isolation	NO	NO	NO	possible
Cursor Stability		Like read committed		
Consistent Read		Like snapshot isolation?		
Serializable Snapshot Isolation	NO	NO	NO	NO (forced write collisions)

Transaction Isolation Levels: Defaults

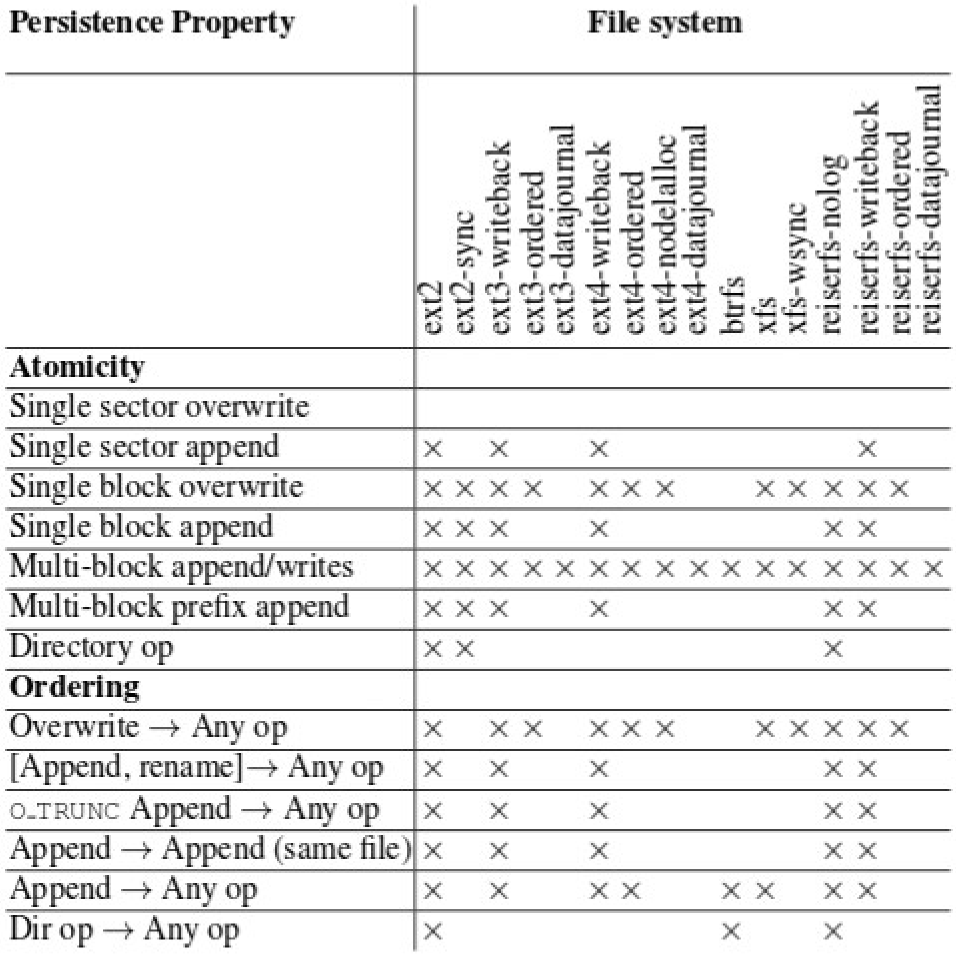


Database	**Default Isolation**	**Maximum Isolation**
Actian Ingres 10.0/10S	S	S
Aerospike	RC	RC
Akiban Persistit	SI	SI
Clustrix CLX 4100	RR	?
Greenplum 4.1	RC	S
IBM DB2 10 for z/OS	CS	S
IBM Informix 11.50	Depends	RR
MySQL 5.6	RR	S
MemSQL 1b	RC	RC
MS SQL Server 2012	RC	S
NuoDB	CR	CR
Oracle 11g	RC	SI
Oracle Berkeley DB	S	S
Oracle Berkeley DB JE	RR	S
Postgres 9.2.2	RC	S
SAP HANA	RC	SI
ScaleDB 1.02	RC	RC
VoltDB	S	S
Legend	*RC: read committed, RR: repeatable read, S: serializability, SI: snapshot isolation, CS: cursor stability, CR: consistent read*	

From Peter Bailis, When is “ACID” ACID? Rarely! Note: Oracle and SAP do not provide serializability at all – even if they use the term!

The Base: File System Consistency

File Systems Block Order Guarantees



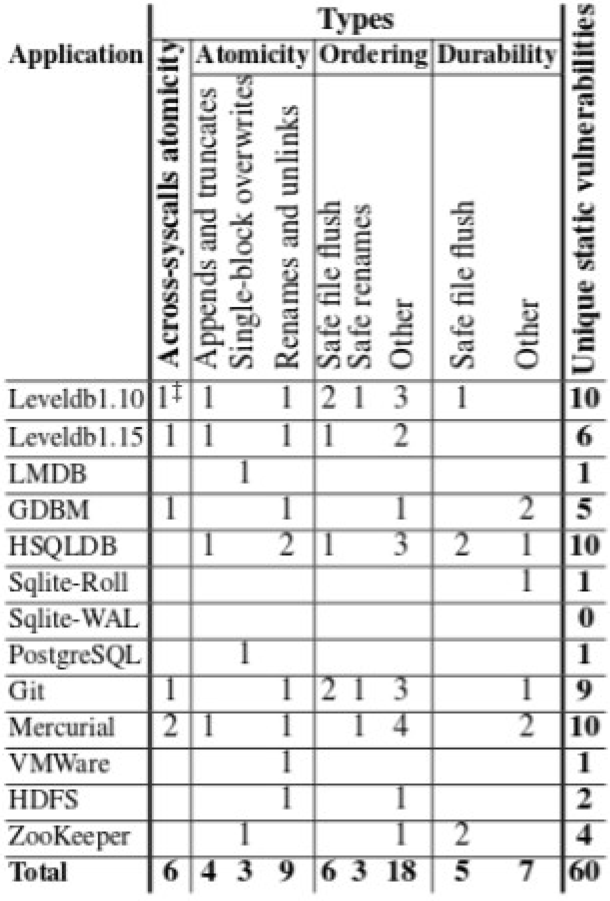
Persistence Property	File system															
	ext2	ext2-sync	ext3-writeback	ext3-ordered	ext3-datajournal	ext4-writeback	ext4-ordered	ext4-nodelalloc	ext4-datajournal	btrfs	xfs	xfs-wsync	reiserfs-nolog	reiserfs-writeback	reiserfs-ordered	reiserfs-datajournal
Atomicity																
Single sector overwrite																
Single sector append	×		×			×								×		
Single block overwrite	×	×	×	×		×	×	×			×	×	×	×	×	
Single block append	×	×	×			×							×	×		
Multi-block append/writes	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
Multi-block prefix append	×	×	×			×							×	×		
Directory op	×	×											×			
Ordering																
Overwrite → Any op	×		×	×		×	×	×			×	×	×	×	×	
[Append, rename]→ Any op	×		×			×							×	×		
O_TRUNC Append → Any op	×		×			×							×	×		
Append → Append (same file)	×		×			×							×	×		
Append → Any op	×		×			×	×			×	×		×	×		
Dir op → Any op	×									×			×			

Table 1: **Persistence Properties.** *The table shows atomicity and ordering persistence properties that we empirically determined for different configurations of file systems. X → Y indicates that X is persisted before Y. [X,Y] → Z indicates that Y follows X in program order, and both become durable before Z. A × indicates that we have a reproducible test case where the property fails in that file system.*

BOB, the Block Order Breaker, is used to find out what behaviours are exhibited by a number of modern file systems that are relevant to building crash consistent applications.

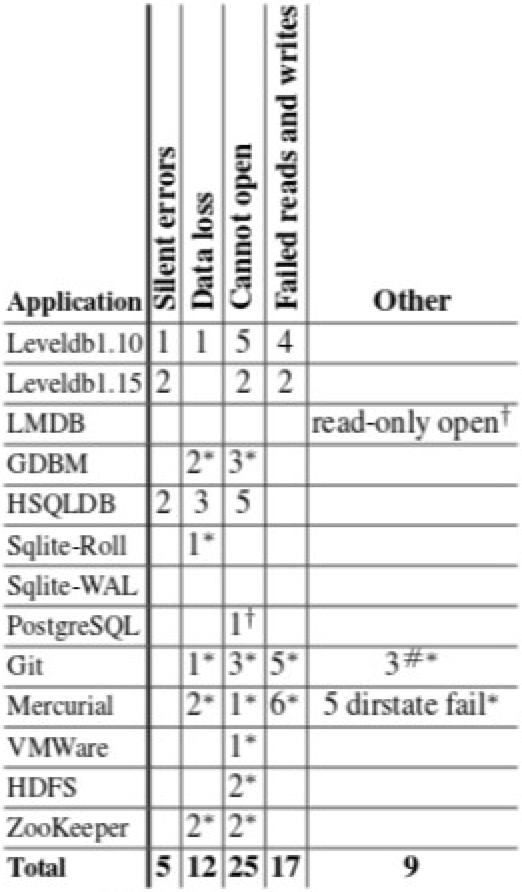
from: http://blog.acolyer.org/2016/02/11/fs-not-equal/

Crash-Consistent Applications



Application	Types									
		Atomicity			Ordering			Durability		
	Across-syscalls atomicity	Appends and truncates	Single-block overwrites	Renames and unlinks	Safe file flush	Safe renames	Other	Safe file flush	Other	Unique static vulnerabilities
Leveldb1.10	1‡	1		1	2	1	3	1		**10**
Leveldb1.15	1	1		1	1		2			**6**
LMDB			1							**1**
GDBM	1			1			1		2	**5**
HSQLDB		1		2	1		3	2	1	**10**
Sqlite-Roll									1	**1**
Sqlite-WAL										**0**
PostgreSQL			1							**1**
Git	1			1	2	1	3		1	**9**
Mercurial	2	1		1		1	4		2	**10**
VMWare				1						**1**
HDFS				1			1			**2**
ZooKeeper			1				1	2		**4**
Total	**6**	**4**	**3**	**9**	**6**	**3**	**18**	**5**	**7**	**60**

(a) Types.



Application	Silent errors	Data loss	Cannot open	Failed reads and writes	Other
Leveldb1.10	1	1	5	4	
Leveldb1.15	2		2	2	
LMDB					read-only open†
GDBM		2*	3*		
HSQLDB	2	3	5		
Sqlite-Roll		1*			
Sqlite-WAL					
PostgreSQL			1†		
Git		1*	3*	5*	3#*
Mercurial		2*	1*	6*	5 dirstate fail*
VMWare			1*		
HDFS			2*		
ZooKeeper		2*	2*		
Total	**5**	**12**	**25**	**17**	**9**

(b) Failure Consequences.

ALICE, the Application Level Intelligent Crash Explorer explore the crash recovery on top of file systems.

All File Systems are Not Created Equal: On the Complexity of Crafting Crash Consistent Applications – Pillai et al. 2014
Discussion: http://blog.acolyer.org/2016/02/11/fs-not-equal/

Eventually Consistent Storage Systems

Consistency without Coordination

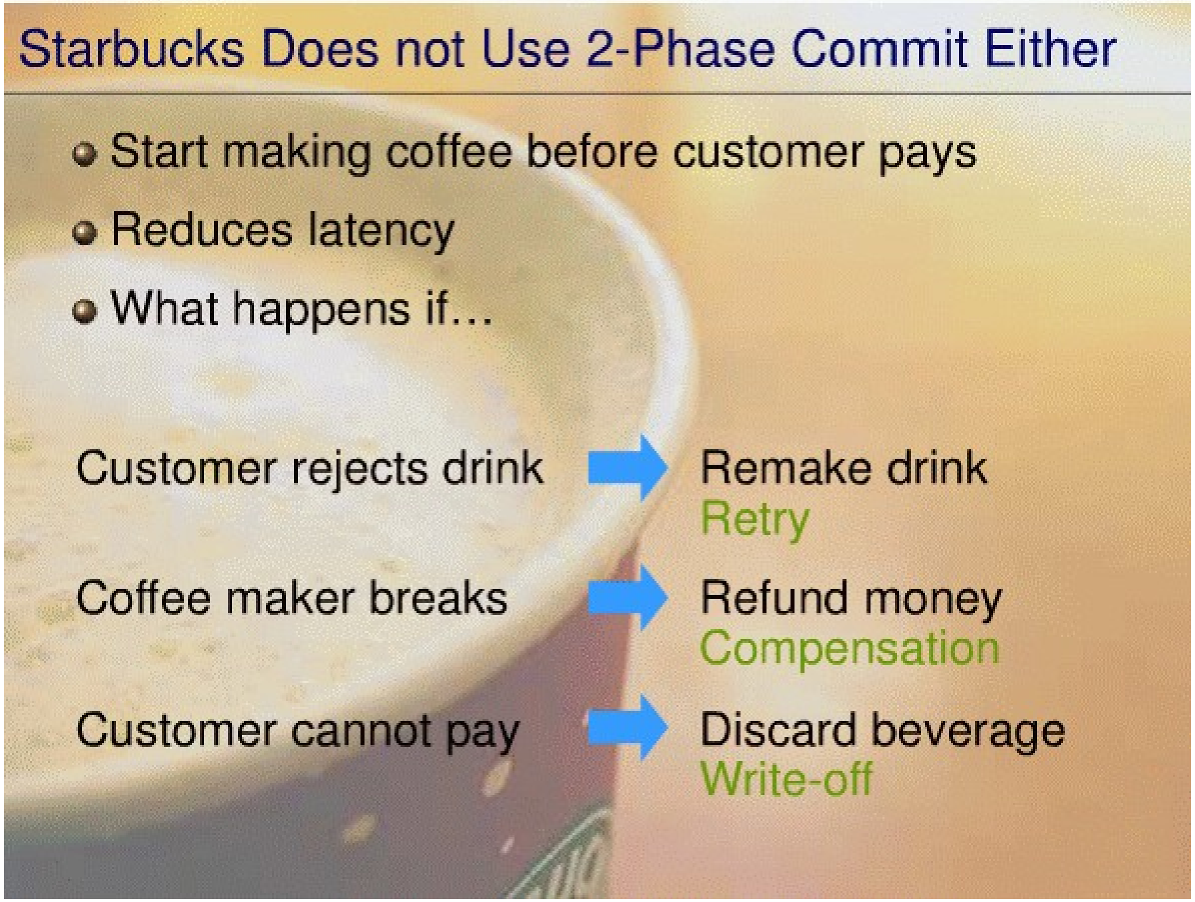
> “The rise of Internet-scale geo-replicated services has led to upheaval in the design of modern data management systems. Given the availability, latency, and throughput penalties associated with classic mechanisms such as serializable transactions, a broad class of systems (e.g., “NoSQL”) has sought weaker alternatives that reduce the use of expensive coordination during system operation, often at the cost of application integrity. When can we safely forego the cost of this expensive coordination, and when must we pay the price?”

From: Coordination Avoidance in Distributed Databases by
Peter David Bailis, Doctor of Philosophy in Computer Science
University of California, Berkeley
http://www.bailis.org/papers/bailis-thesis.pdf

Forces behind NoSQL

- Web2.0 brings user generated content: much more writes!
- Social Networks create huge datasets with structures difficult for relational Dbs (graph processing, sparse data)
- Fast growing sites need a storage layer that scales horizontally
- Fast growing sites want schema-less storage
- Data frequently unstructured – need map/reduce for scan
- Data processing mostly sequential
- Queries against RDBMs anyway too expensive and not possible with shards
- Joins and queries scale against RAM based caches/DBs
- Application servers scale better horizontally than RDBMs
- Scaling storage needs to be automatic
- User data allow less than ACID processing sometimes
- ACID does not work across shards very well

RDBMs did too much in some cases and too little in others (M.Stonebreaker).



Starbucks Does not Use 2-Phase Commit Either

- Start making coffee before customer pays
- Reduces latency
- What happens if…

Customer rejects drink → Remake drink
Retry

Coffee maker breaks → Refund money
Compensation

Customer cannot pay → Discard beverage
Write-off

The fundamental scaling Problems of RDBMs

The poor time complexity characteristics of SQL joins; O(m+n) or worse

The difficulty in horizontally scaling; Loss of joins or jumping between nodes

The unbounded nature of queries; A query can kill a DB und load

Optimized for storage efficiency (no duplicates), integrity and flexibility of access through arbitrary joins.

```
SELECT user_id, sum(amount) AS
total_amount
FROM orders
GROUP BY user_id
ORDER BY total_amount DESC
LIMIT 5
```

A.Debrie, SQL, NoSQL, and Scale: How DynamoDB scales where relational databases don't. https://www.alexdebrie.com/posts/dynamodb-no-bad-queries/

NoSQL Storage Systems

NoSQL Design Patterns

Stores with probabilistic guarantees:

AP-Stores: Dynamo, Cassandra, CouchDB, MongoDB, Riak etc.

Stores with strong guarantees:

CALM (bloom lang),

CRDTs: Convergent/Consistent Replicated Data Types

NoSQL Design Patterns

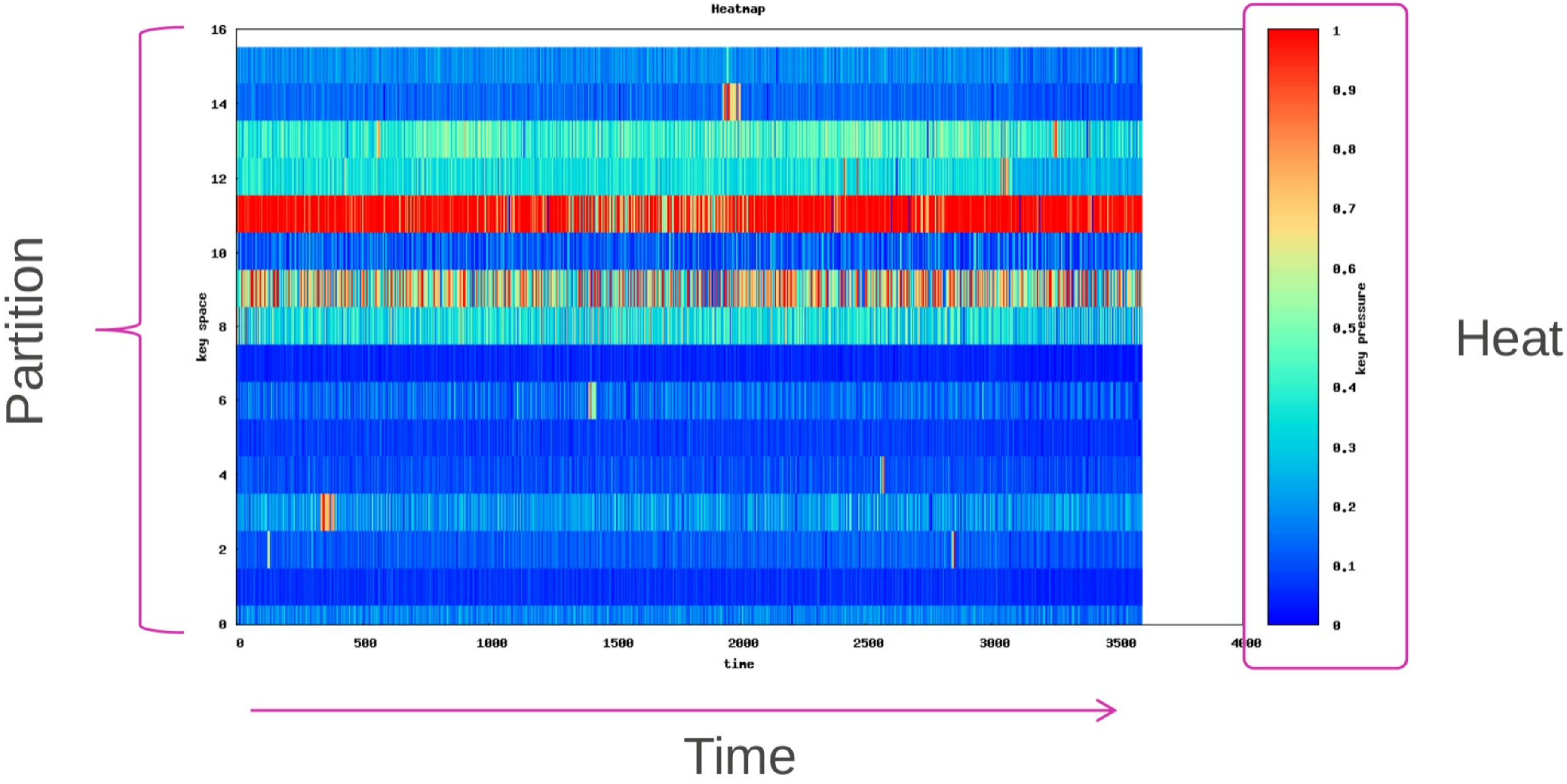
- Partition keys with many distinct values
- All data in one table with hierarchical modeling and de-normalization
- Values evenly requested
- Use composite secondary keys for 1:n, n:n, queries
- limited query responses (with paging token)

- understand the use case
- know you access patterns before data layout
- Avoid relational modeling
- data integrity is an application concern
- data storage efficiency is no concern

Great overview from R. Houlihan at re:event 2018:

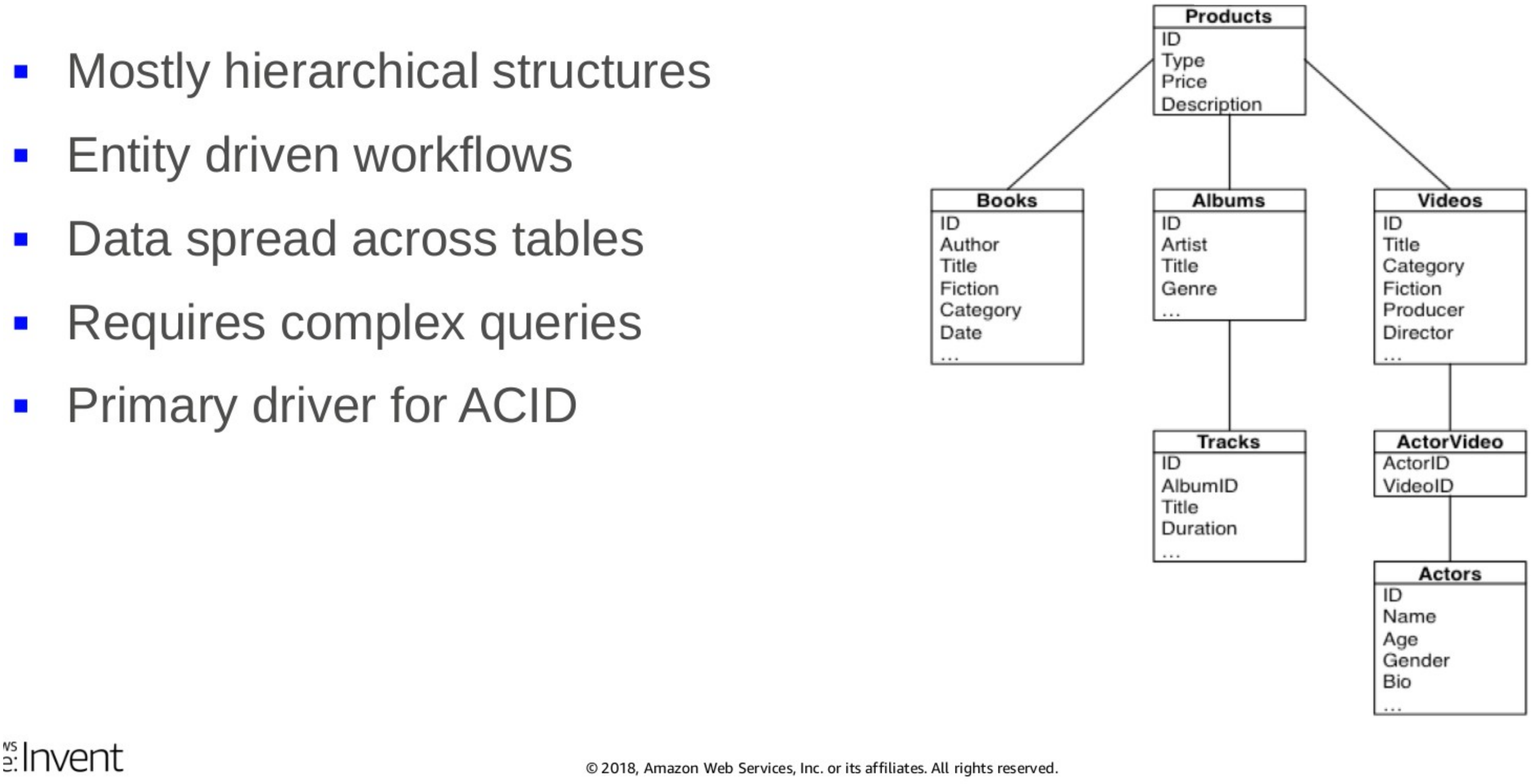
https://www.portal.reinvent.awsevents.com/connect/sessionDetail.ww?SESSION_ID=22972

AntiPattern1: hot keys



From Houlihan. A key with a small range of values prevents horizontal scaling

AntiPattern2: Relational OLTP



- Mostly hierarchical structures
- Entity driven workflows
- Data spread across tables
- Requires complex queries
- Primary driver for ACID

Products
ID
Type
Price
Description

Books
ID
Author
Title
Fiction
Category
Date
…

Albums
ID
Artist
Title
Genre
…

Videos
ID
Title
Category
Fiction
Producer
Director
…

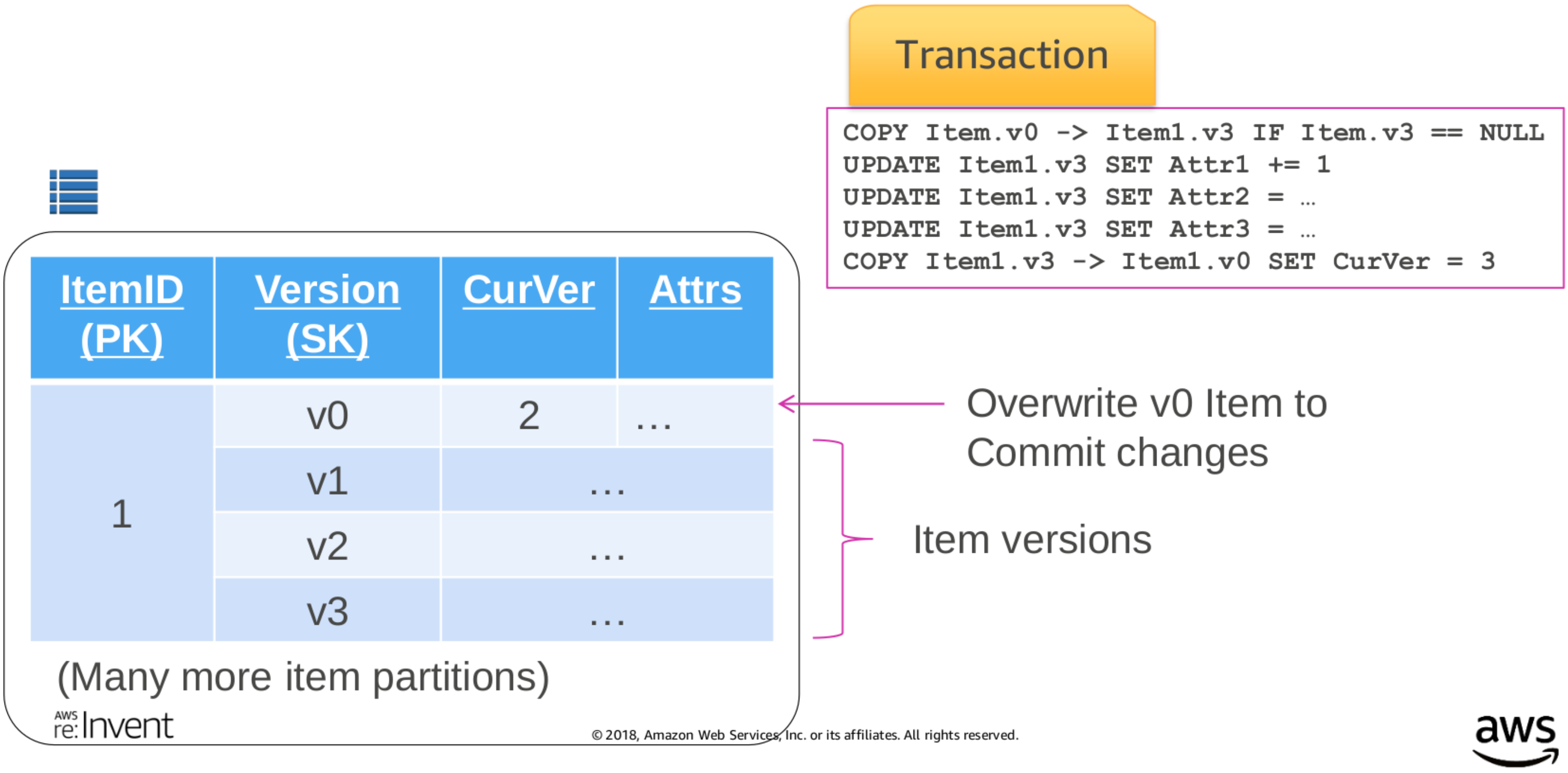
Tracks
ID
AlbumID
Title
Duration
…

ActorVideo
ActorID
VideoID

Actors
ID
Name
Age
Gender
Bio
…

From Houlihan. Relational OLTP is flexible (analytical) but does not scale horizontally

AntiPattern3: Overwriting data



```
COPY Item.v0 -> Item1.v3 IF Item.v3 == NULL
UPDATE Item1.v3 SET Attr1 += 1
UPDATE Item1.v3 SET Attr2 = …
UPDATE Item1.v3 SET Attr3 = …
COPY Item1.v3 -> Item1.v0 SET CurVer = 3
```

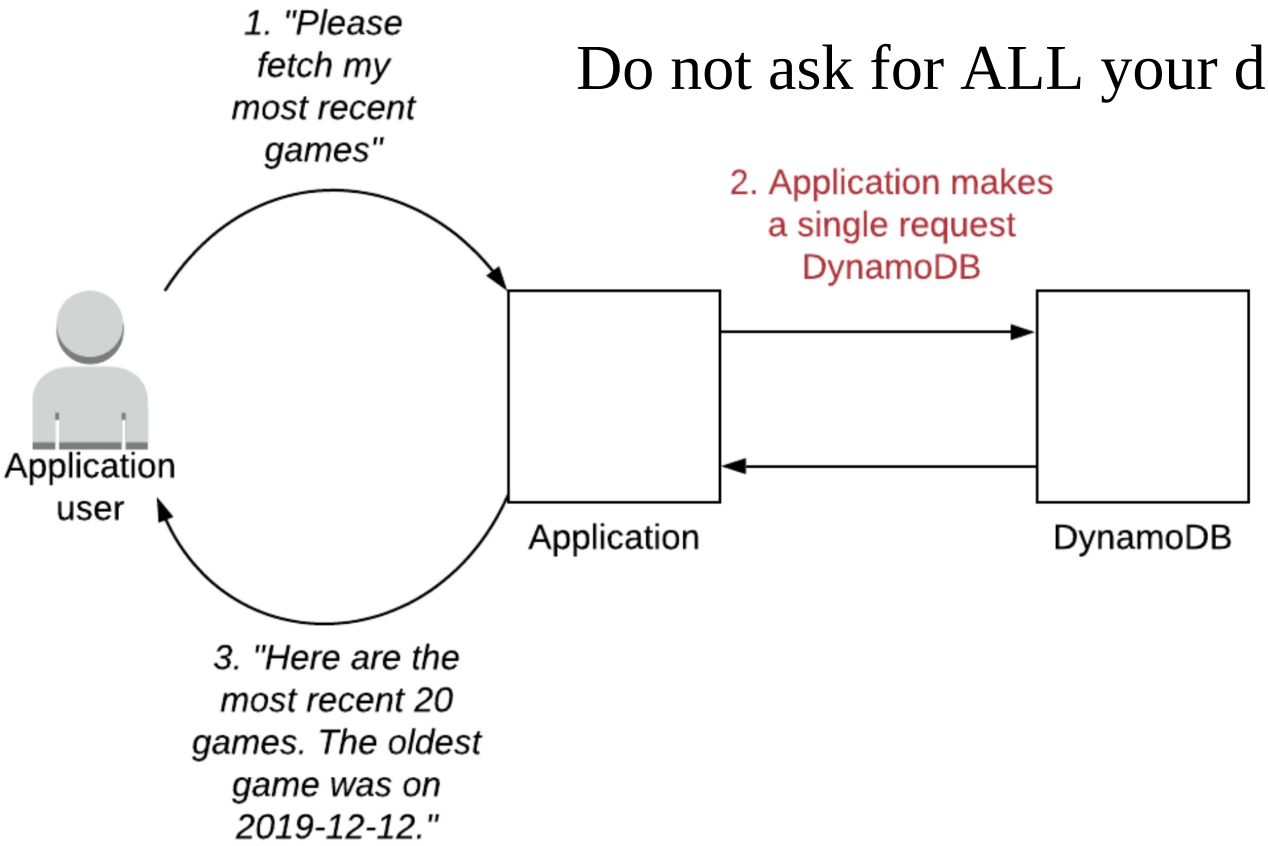
ItemID (PK)	Version (SK)	CurVer	Attrs
1	v0	2	…
	v1	…	
	v2	…	
	v3	…	

(Many more item partitions)

From Houlihan. Keep versions and update only current version. This can easily be done atomically.

AntiPattern 4: Pagination

Do not ask for ALL your data!



1. "Please fetch my most recent games"

2. Application makes a single request DynamoDB

3. "Here are the most recent 20 games. The oldest game was on 2019-12-12."

Application user

Application

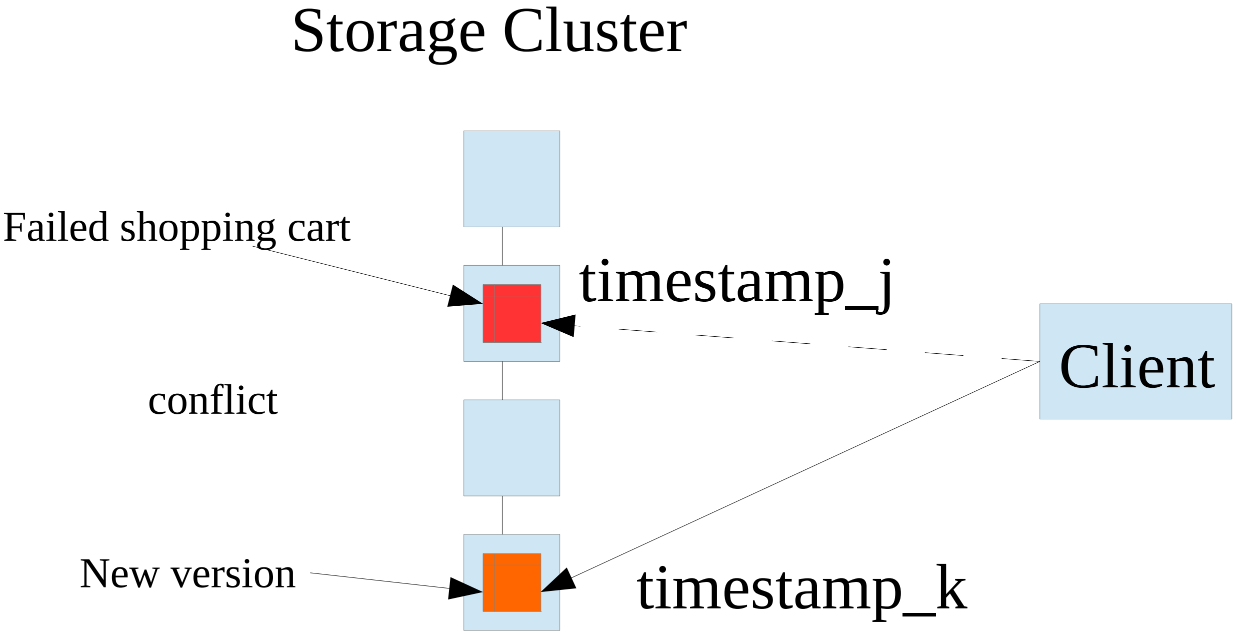
DynamoDB

A.Debrie, SQL, NoSQL, and Scale: How DynamoDB scales where relational databases don't. https://www.alexdebrie.com/posts/dynamodb-no-bad-queries/

Dynamo: Always Writeable AP Key/Value Store

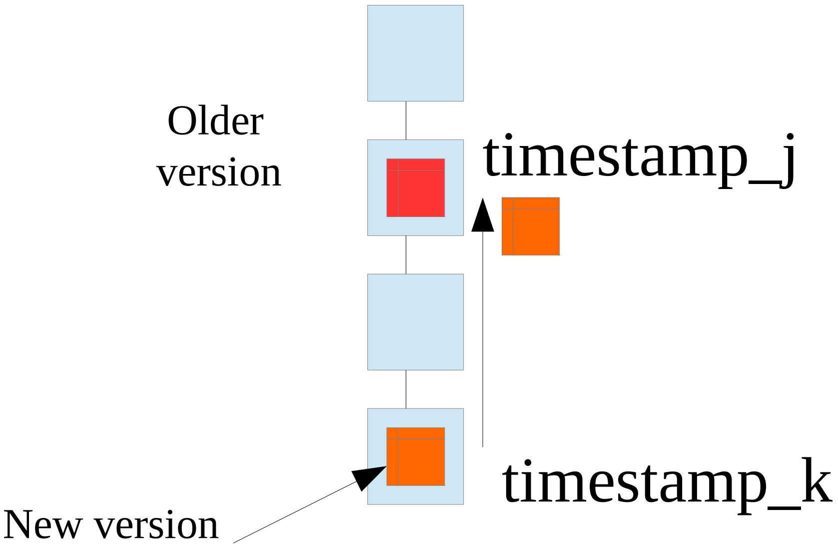
Must see: on Dynamo Design Patterns: Rick Houlihan
https://www.youtube.com/watch?time_continue=9&v=HaEPXoXVf2k

Dynamo Application Area: Shopping Cart Example



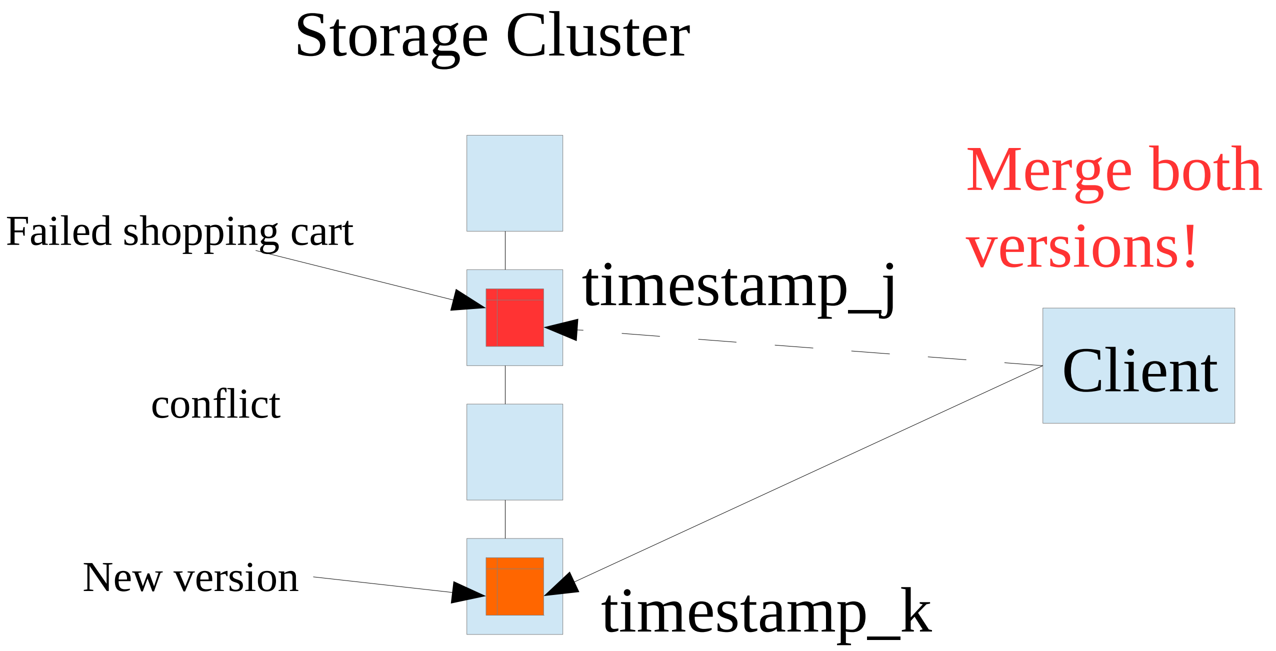
After:Giuseppe DeCandia et.al., Dynamo: Amazon’s Highly Available Key-value Store. The Dynamo design principles need to be known by clients. Not every use-case can be mapped to an eventually consistent store.

Automatic - Reconciliation Approach



“last write wins” strategy. After:G. DeCandia et.al. Dynamo can automatically try to reconcile different versions across replicas.

Client/Business – Reconciliation Approach



Storage Cluster

Failed shopping cart

timestamp_j

Merge both versions!

Client

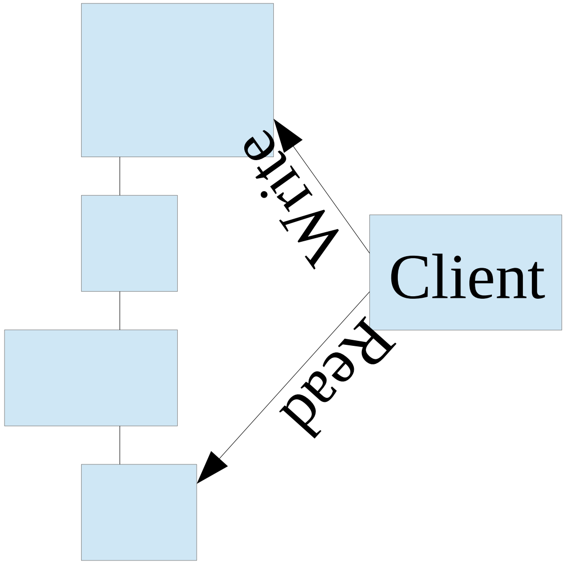
conflict

New version

timestamp_k

After:Giuseppe DeCandia et.al., Dynamo: Amazon’s Highly Available Key-value Store. The client gets all available versions to choose or merge. This is a business-driven strategy. For an idea on the complexity behind such “compensation” schemes, see: P.Bailis, A.Ghodsi, Eventual Consistency Today: Limitations, Extensions, and Beyond. How can applications be built on eventually consistent infrastructure given no guarantee of safety? In 11/3 acmqueue

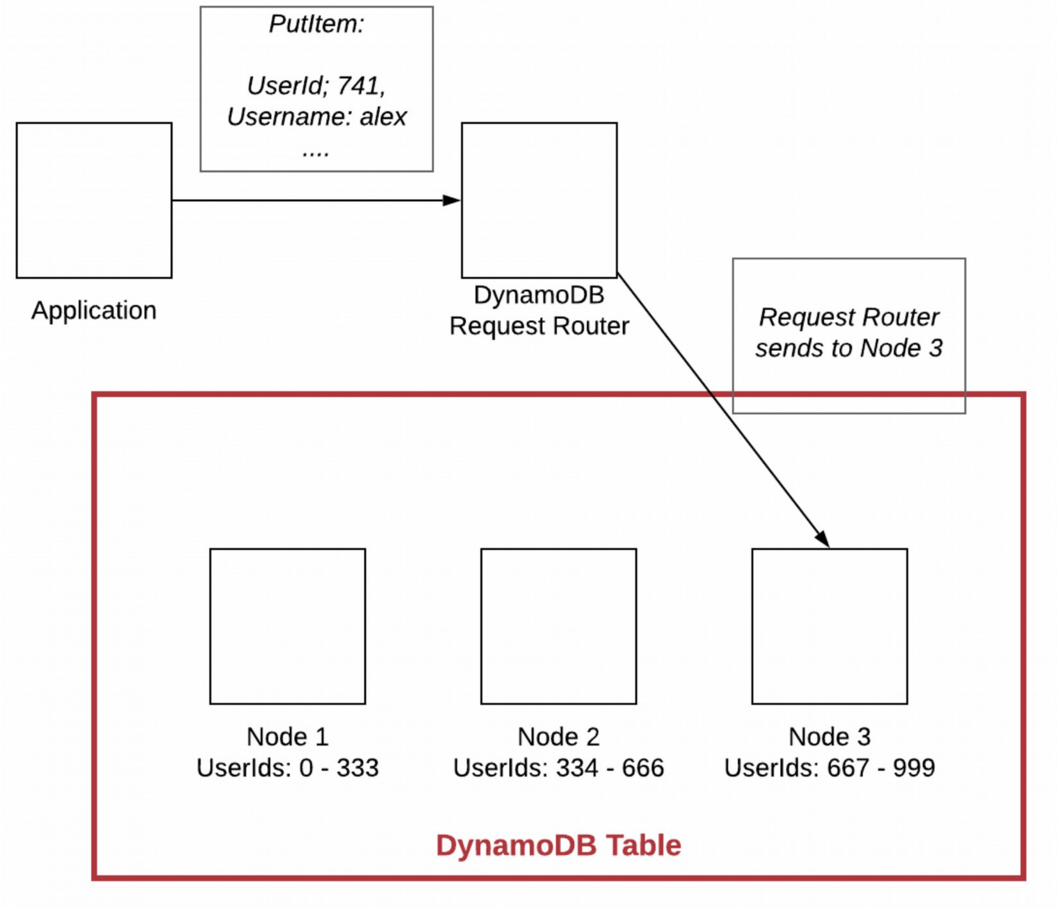
Dynamo Design Principles



- No master (decentralized)
- heterogeneous hardware
- symmetric peers
- incrementally scalable
- eventually consistent
- trusted environment needed
- replication support (conf.)
- “always write” enabled (conflict resolution during read)
- multi-version store with conflict resolution policies

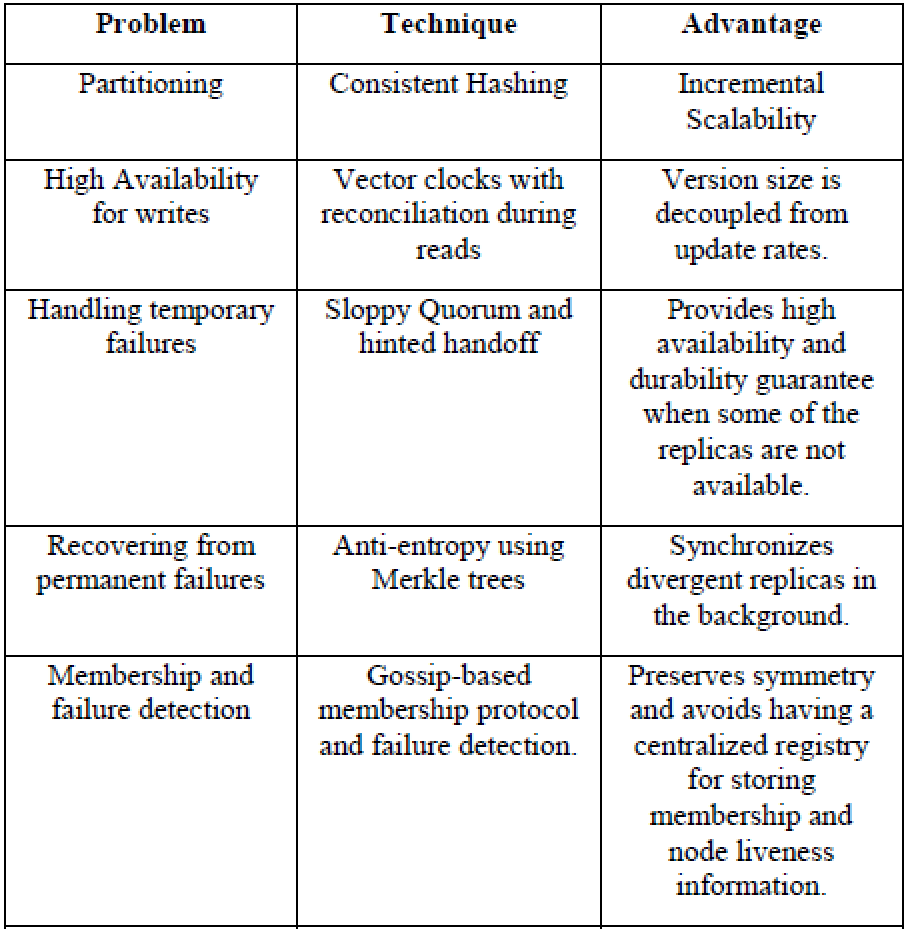
After:Giuseppe DeCandia et.al., Dynamo: Amazon’s Highly Available Key-value Store. Support for heterogeneous hardware requires the concept of “virtual” nodes. Key spaces are assigned to virtual nodes.

Horizontal Scaling



A.Debrie, SQL, NoSQL, and Scale: How DynamoDB scales where relational databases don't. https://www.alexdebrie.com/posts/dynamodb-no-bad-queries/

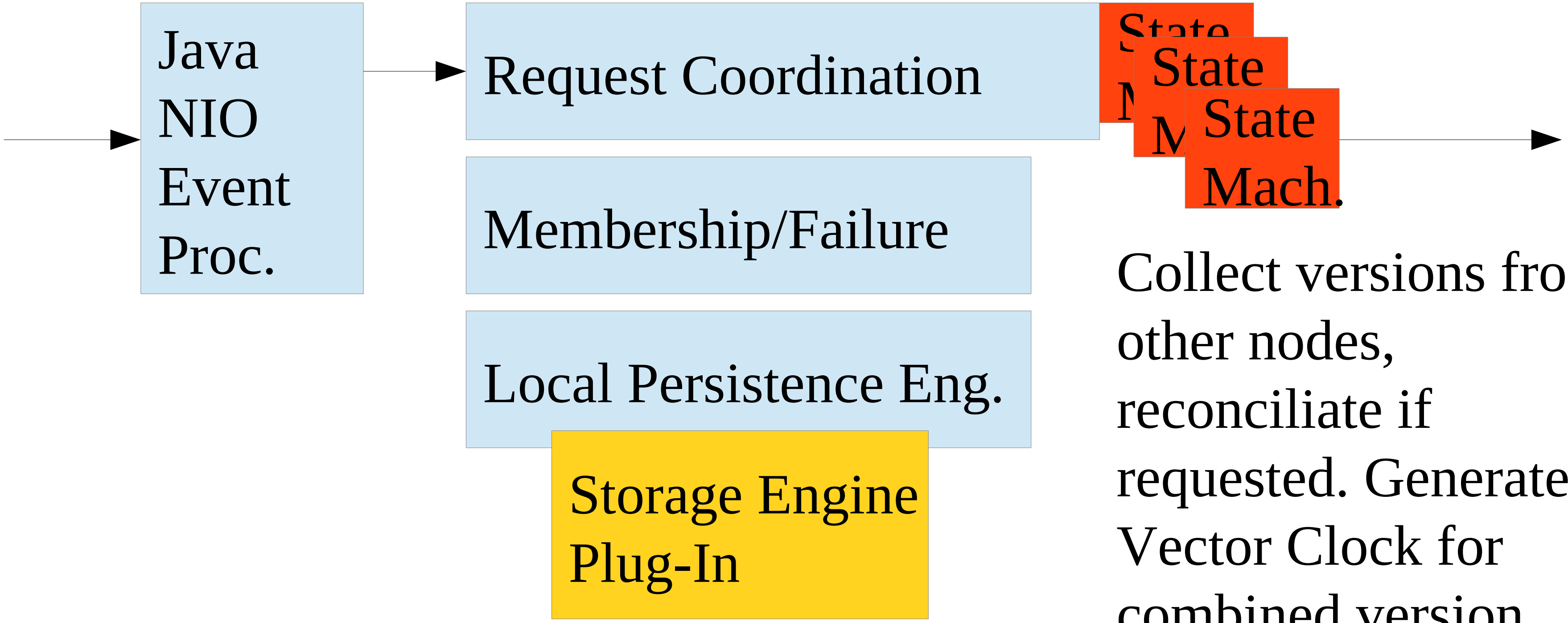
Dynamo-Technology



Problem	Technique	Advantage
Partitioning	Consistent Hashing	Incremental Scalability
High Availability for writes	Vector clocks with reconciliation during reads	Version size is decoupled from update rates.
Handling temporary failures	Sloppy Quorum and hinted handoff	Provides high availability and durability guarantee when some of the replicas are not available.
Recovering from permanent failures	Anti-entropy using Merkle trees	Synchronizes divergent replicas in the background.
Membership and failure detection	Gossip-based membership protocol and failure detection.	Preserves symmetry and avoids having a centralized registry for storing membership and node liveness information.

After:G. DeCandia et.al. Dynamo allows clients to define the number of replicas (n).

Dynamo Software Architecture



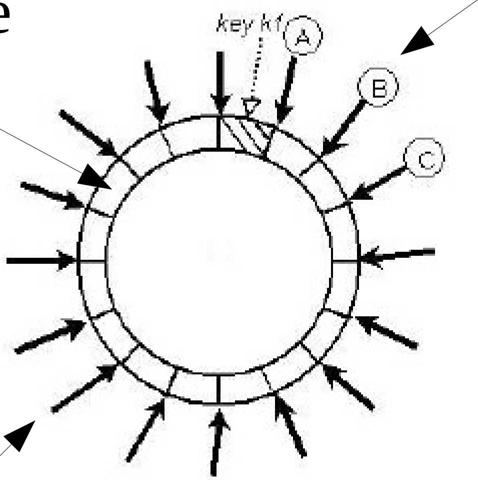
Collect versions from other nodes, reconciliate if requested. Generate Vector Clock for combined version. Perform read-repair on outdated nodes.

After:Giuseppe DeCandia et.al., Dynamo: Amazon’s Highly Available Key-value Store. For writes, a coordinator node is picked from the preference list for this key space. Usually the fastest node that answered the last read request is chosen. This makes “read-your-writes consistency more likely and causes fewer SLA violations.

Dynamo-DHT: Placement vs. Partitioning

Virtual nodes resp. for
key space and replicas

Key spaces of equal size



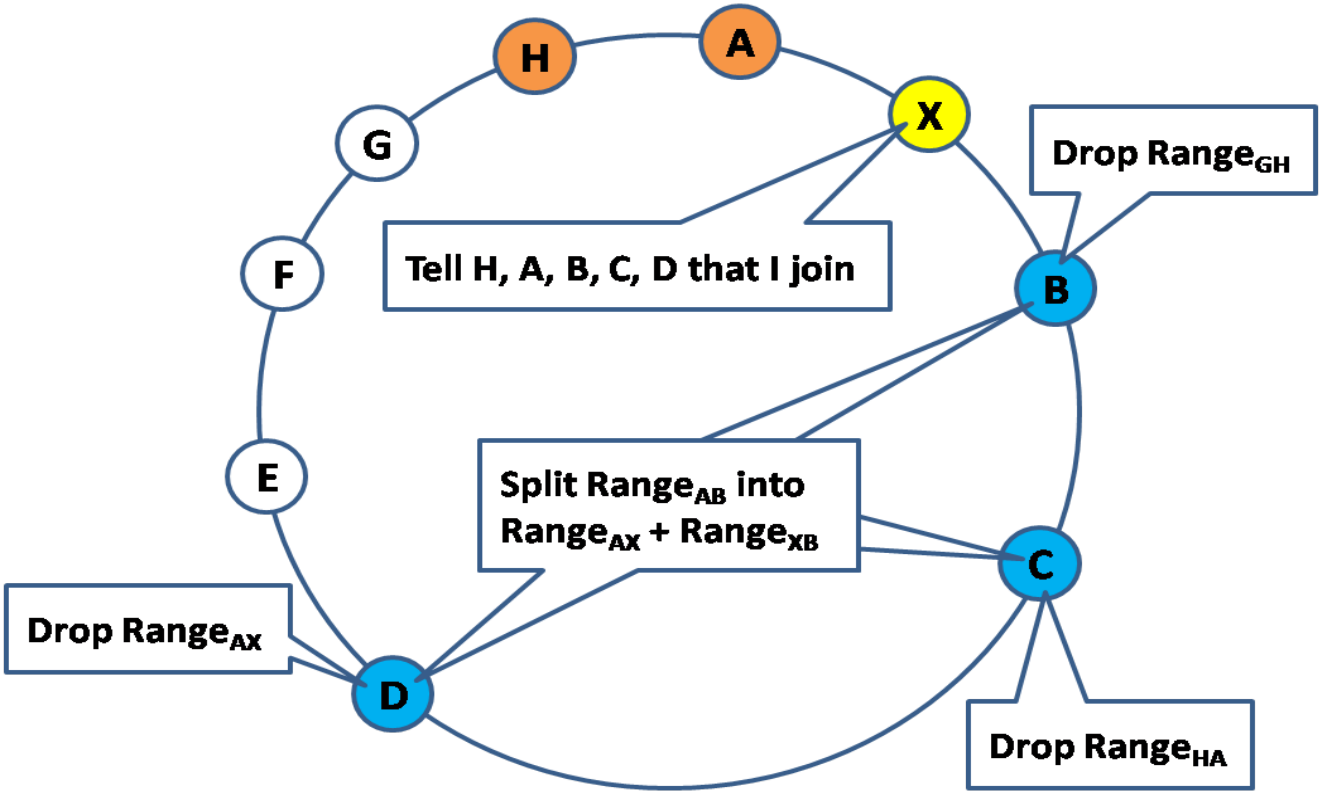
Strategy 3

Positions in the ring, assigned to virtual hosts

After:G. DeCandia et.al. Dynamo uses an enhanced consistent hashing algorithm to balance load between nodes and to minimize changes when nodes join or leave the ring. Equally sized key spaces allow efficient copying of spaces across machines. A gossip protocol distributes key space and node information across the ring. (An alternative would be to use a DHT approach directly to locate a machine that "knows" where to find the required information)

DHT: Membership-Change (Join)

H, A, X, B, C, D will update the membership synchronously
And then asynchronously propagate the membership changes to other nodes



Tell H, A, B, C, D that I join

Drop Range$_{GH}$

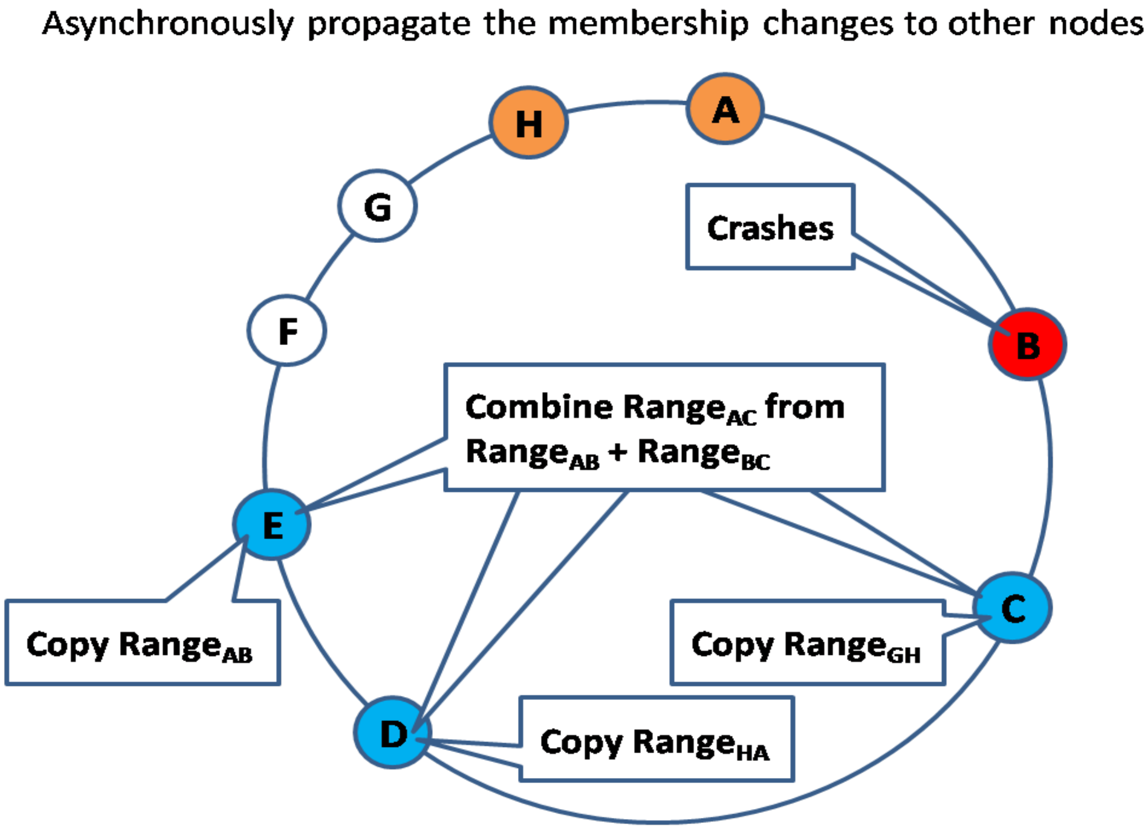
Split Range$_{AB}$ into Range$_{AX}$ + Range$_{XB}$

Drop Range$_{AX}$

Drop Range$_{HA}$

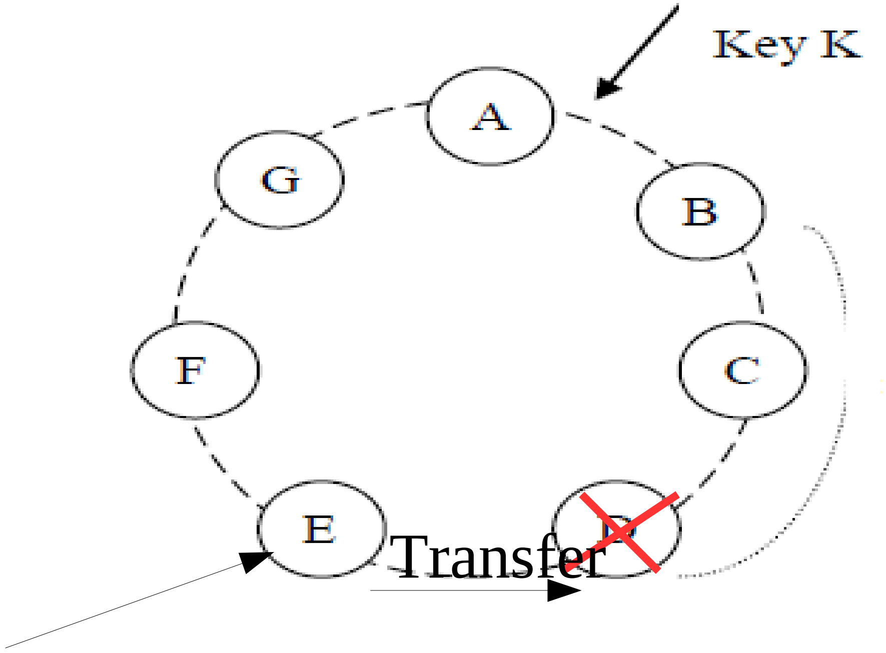
After:Ricky Ho, NOSQL Patterns.

DHT: Membership-Change (Leave)



After:Ricky Ho, NOSQL Patterns. Interestingly, the leaving of a real node – probably due to a crash – should not be considered a permanent thing, causing re-balancing of the ring. Most likely, the node will be replaced shortly. Dynamo uses an explicit error handling protocol to add and remove nodes to avoid unnecessary overhead.

Dynamo “Sloppy Quorums”

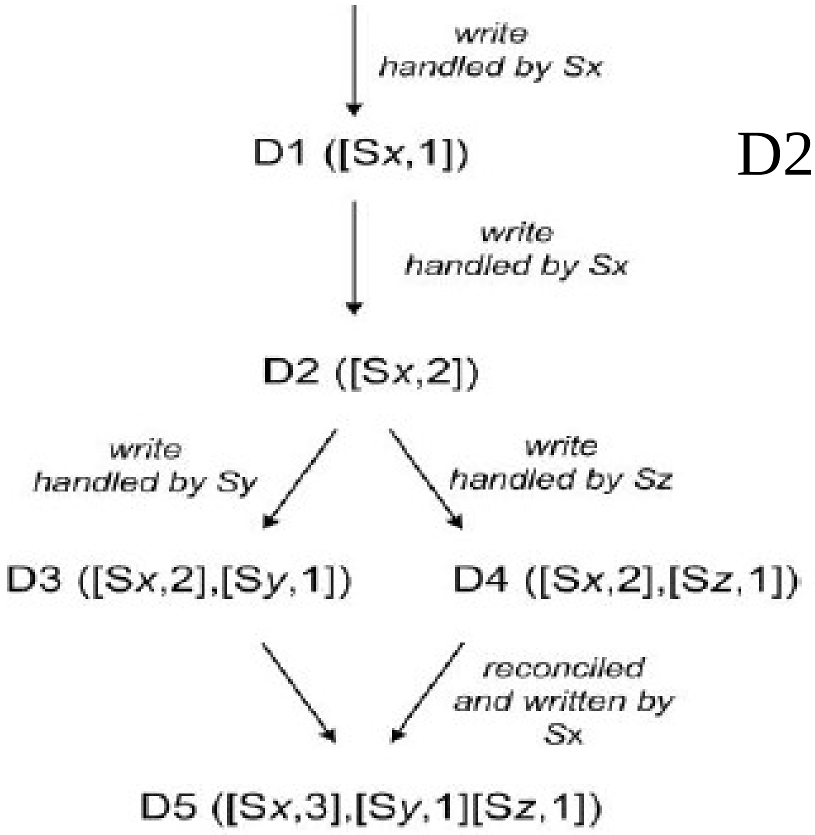


Temp.
Replica for
D's keys

Transfer

After:G. DeCandia et.al. “All read and write operations are performed on the first N healthy nodes from the preference list, which may not always be the first N nodes encountered while walking the consistent hashing ring.” Node E will temporarily store replicas for D and later transfer those back to D.

Dynamo Versioning and Reconciliation

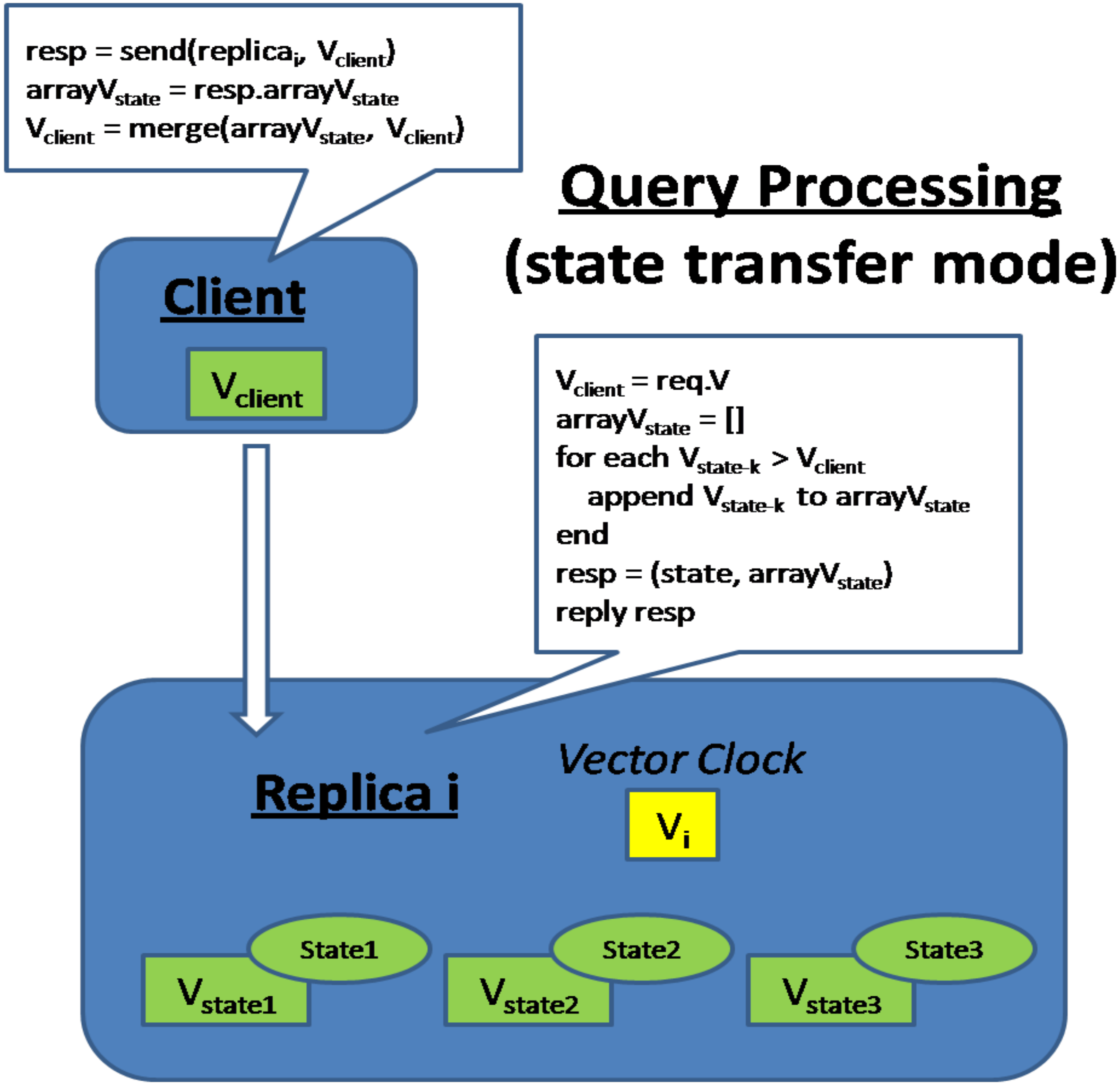


D2 is descendant of D1

D3 and D4 are causally unrelated versions (conflict)

After:G. DeCandia et.al. A client reading D will get D3 and D4 versions and needs to reconcile them into a new version (D5). Dynamo will then distribute D5 as the new version and discard older D's.

Read Processing with VC



```
$resp = send(replica_i, V_{client})$
$arrayV_{state} = resp.arrayV_{state}$
$V_{client} = merge(arrayV_{state}, V_{client})$
```

Query Processing

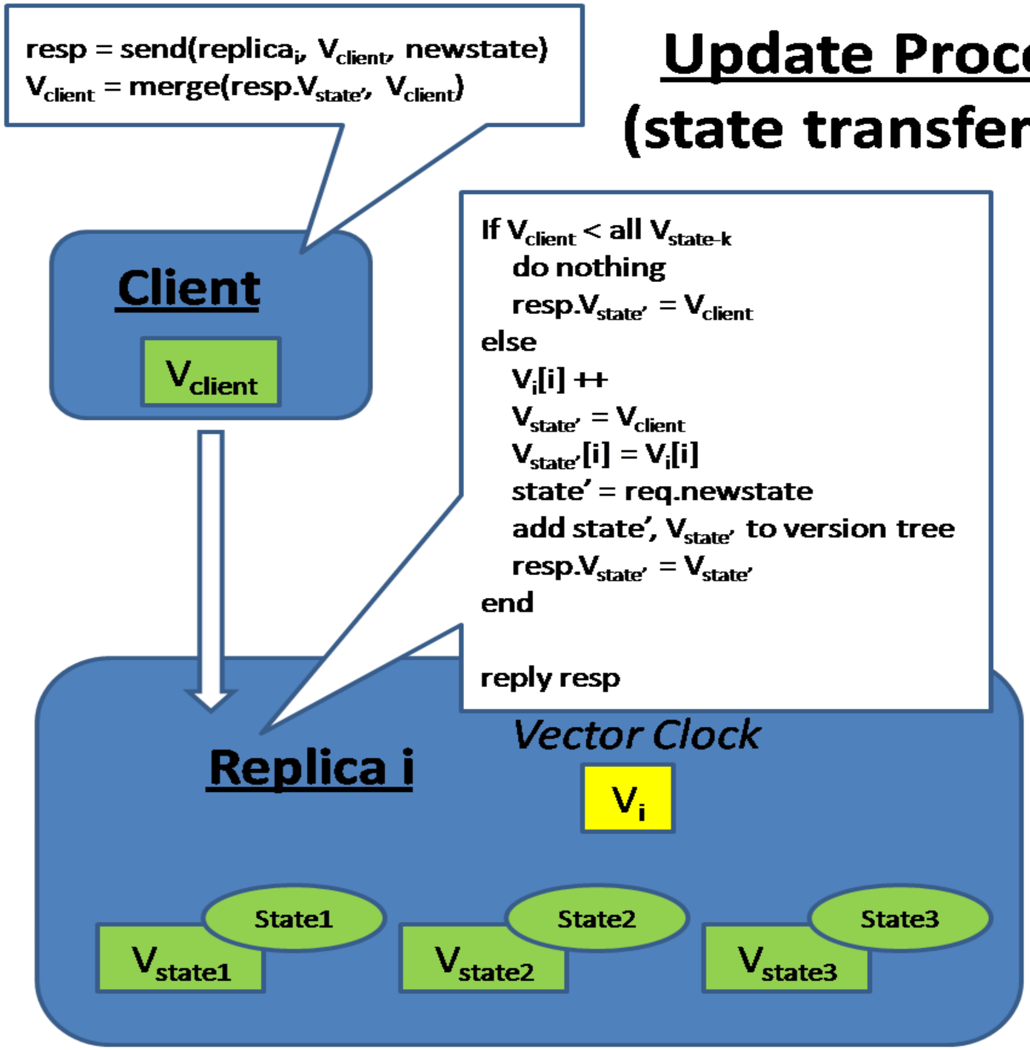
(state transfer mode)

```
$V_{client} = req.V$
$arrayV_{state} = []$
for each $V_{state-k} > V_{client}$
    append $V_{state-k}$ to $arrayV_{state}$
end
$resp = (state, arrayV_{state})$
reply resp
```

After:Ricky Ho, NOSQL Patterns. A replica keeps a counter for every key and also the version state from other replicas (conflicts)

Update Processing with VC

Update Processing (state transfer mode)

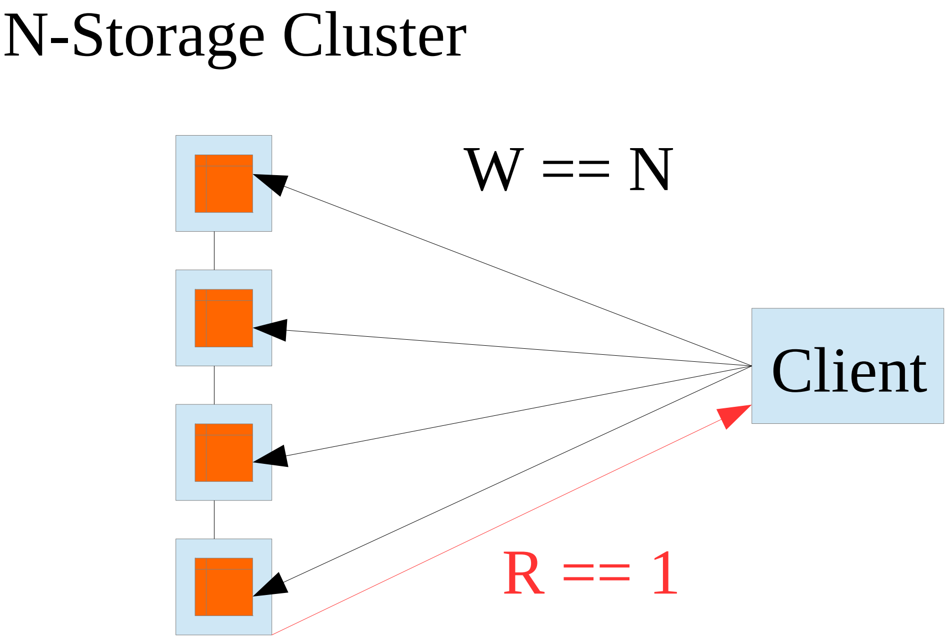


```
resp = send(replica$_i$, V$_{client}$, newstate)
V$_{client}$ = merge(resp.V$_{state'}$, V$_{client}$)
```

```
If V$_{client}$ < all V$_{state-k}$
    do nothing
    resp.V$_{state'}$ = V$_{client}$
else
    V$_i$[i] ++
    V$_{state'}$ = V$_{client}$
    V$_{state'}$[i] = V$_i$[i]
    state’ = req.newstate
    add state’, V$_{state'}$ to version tree
    resp.V$_{state'}$ = V$_{state'}$
end

reply resp
```

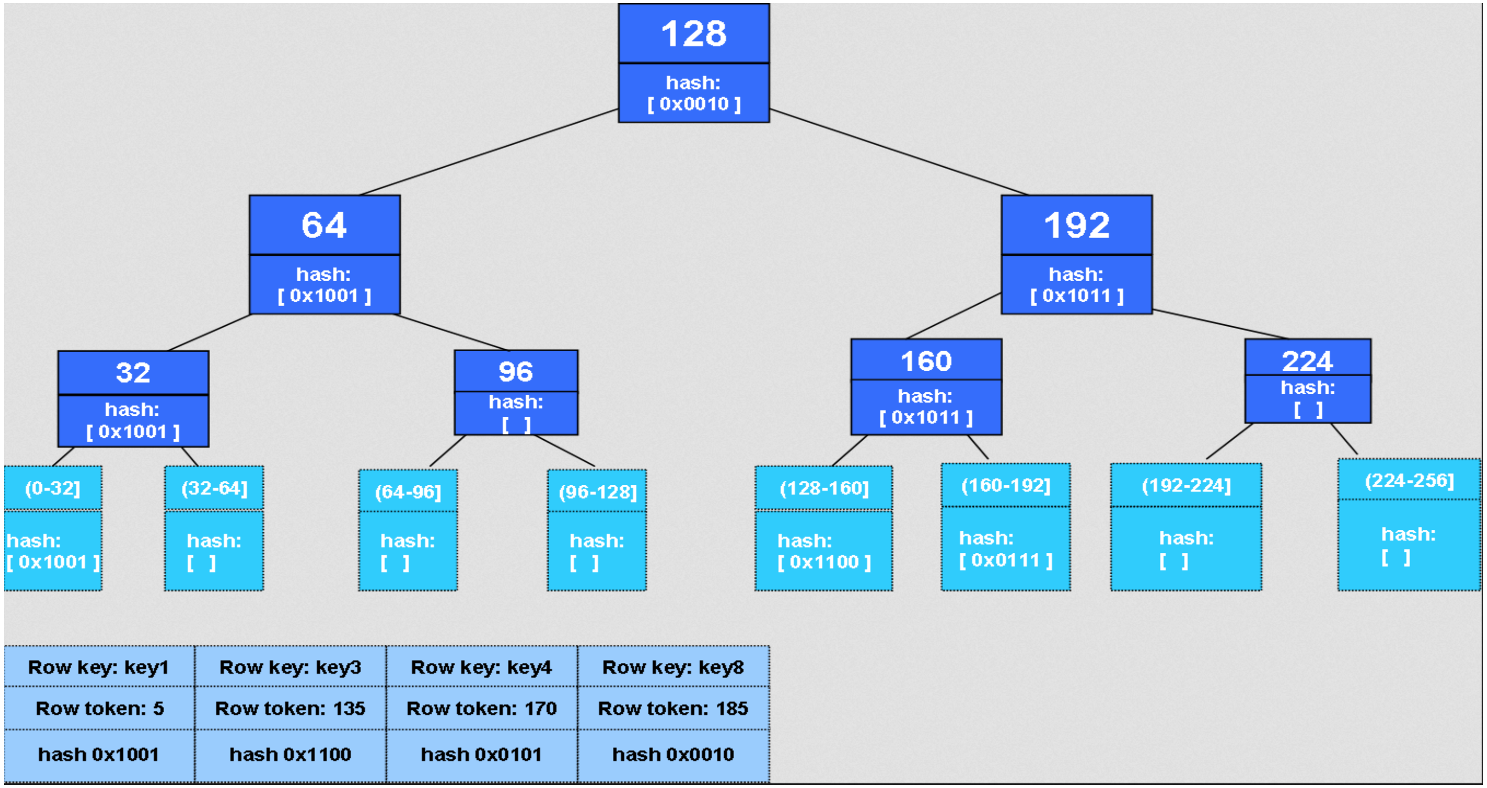
After:Ricky Ho, NOSQL Patterns. An update reconciles different versions into a new one – or gets thrown away, if it is based on an outdated vector clock.

High-Performance Read Engine



After:Giuseppe DeCandia et.al., Dynamo: Amazon’s Highly Available Key-value Store. In case of much more reads than writes, it is OK that writes are slow.

Background Anti-Entropy



Row key: key1	Row key: key3	Row key: key4	Row key: key8
Row token: 5	Row token: 135	Row token: 170	Row token: 185
hash 0x1001	hash 0x1100	hash 0x0101	hash 0x0010

After: Bharatendra Boddu, Using Merkle trees to detect inconsistencies in data (http://distributeddatastore.blogspot.de/2013/07/cassandra-using-merkle-trees-to-detect.html). Two replicas exchange Merkle hash-trees instead of whole volumes (gossip protocol). Hash-trees allow fast detection of changes in branches by only comparing hashes. In case of differences, replicas perform bulk-updates. Note that anti-entropy needs to work asynchronously in the background!

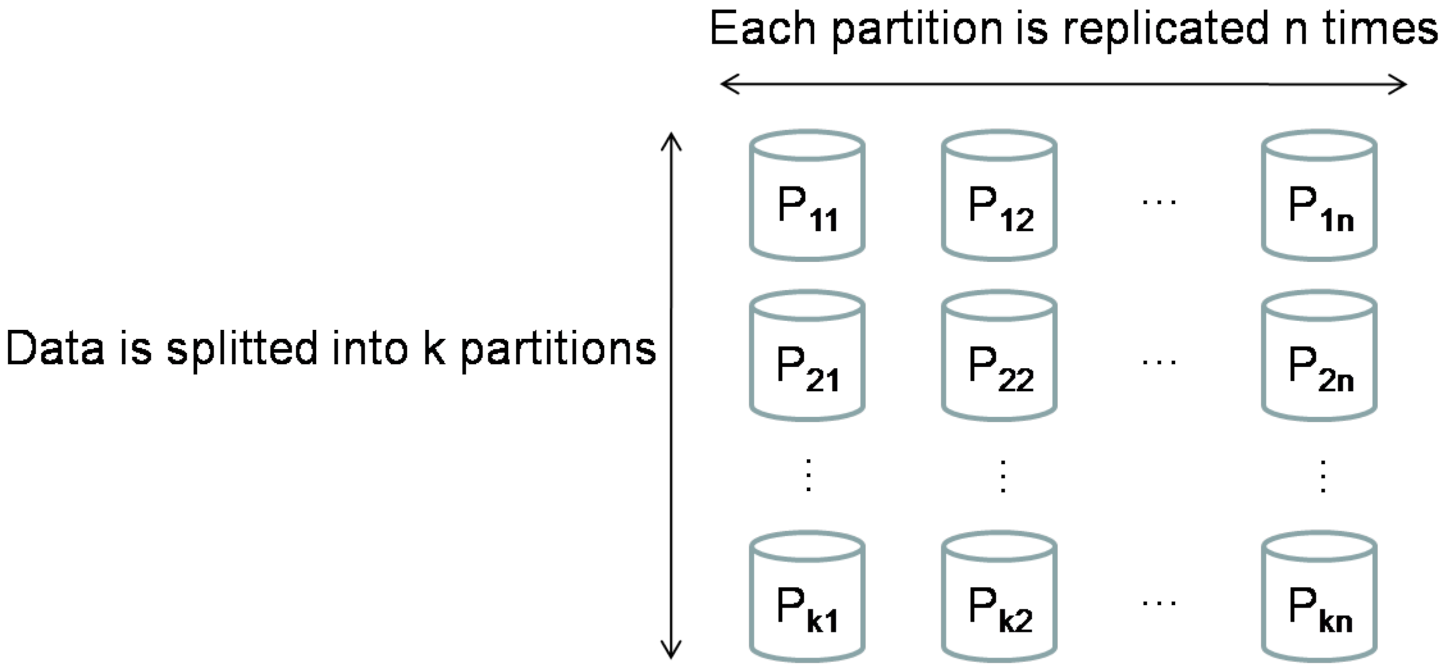
Dynamo Configuration

R: Number of replicas for a read operation
W: Number of replicas for an update operation
N: Number of replicas wanted

After Werner Vogels, http://www.allthingsdistributed.com/2007/10/amazons_dynamo.html

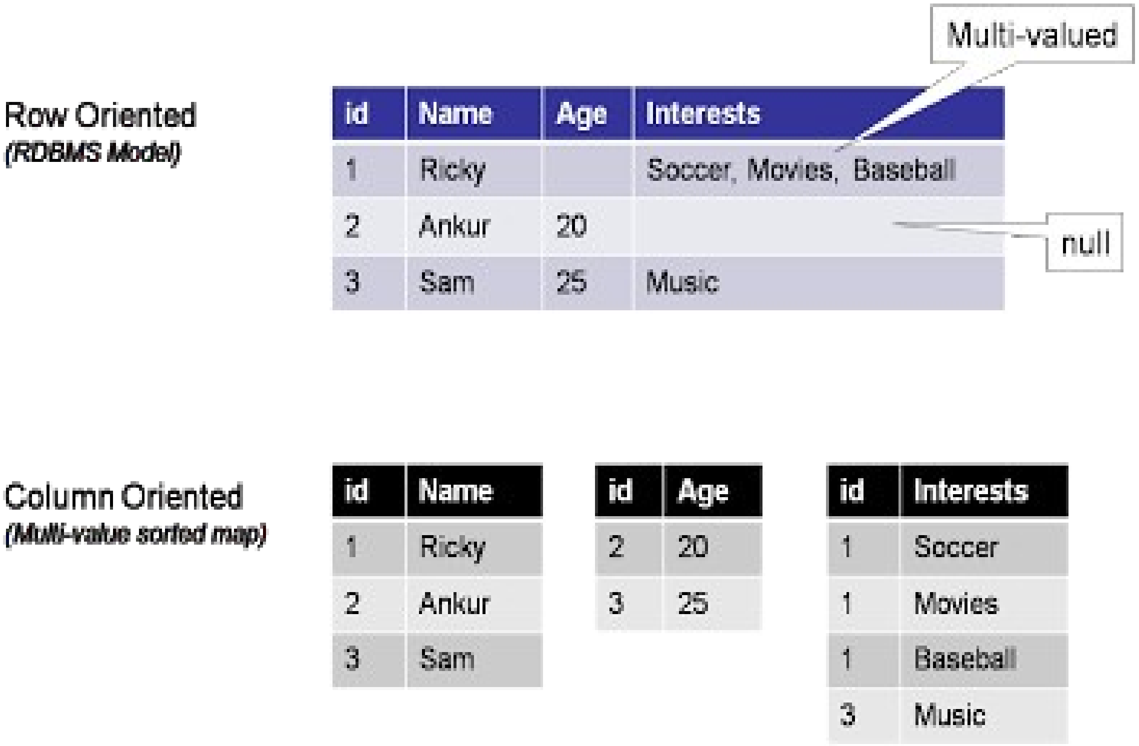
AP-Column-Store: Cassandra

BigTable Cluster Architecture



After: Ricky Ho, BigTable Model with Cassandra and Hbase. This Architecture can be realized e.g. with a DHT storage layer. Also search engines use the option to scale both data size and request numbers independently.

BigTable Column Store Concept



Row Oriented
(RDBMS Model)

id	Name	Age	Interests
1	Ricky		Soccer, Movies, Baseball
2	Ankur	20	
3	Sam	25	Music

Column Oriented
(Multi-value sorted map)

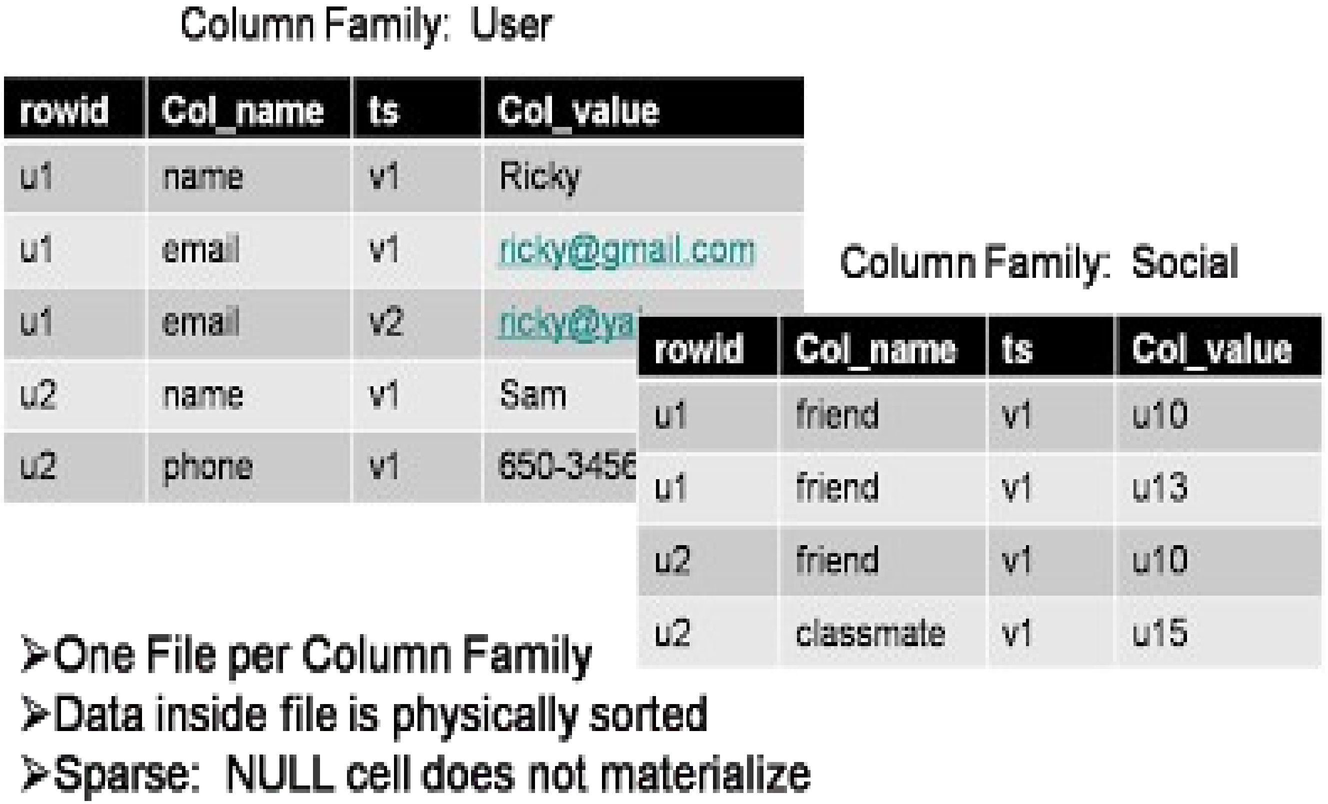
id	Name
1	Ricky
2	Ankur
3	Sam

id	Age
2	20
3	25

id	Interests
1	Soccer
1	Movies
1	Baseball
3	Music

After: Ricky Ho, BigTable Model with Cassandra and Hbase. Columns allow sequential processing at high speed. They can easily deal with empty fields (a user could have 0 or millions of followers).

BigTable Column Families Concept



Column Family: User

rowid	Col_name	ts	Col_value
u1	name	v1	Ricky
u1	email	v1	ricky@gmail.com
u1	email	v2	ricky@ya
u2	name	v1	Sam
u2	phone	v1	650-3456

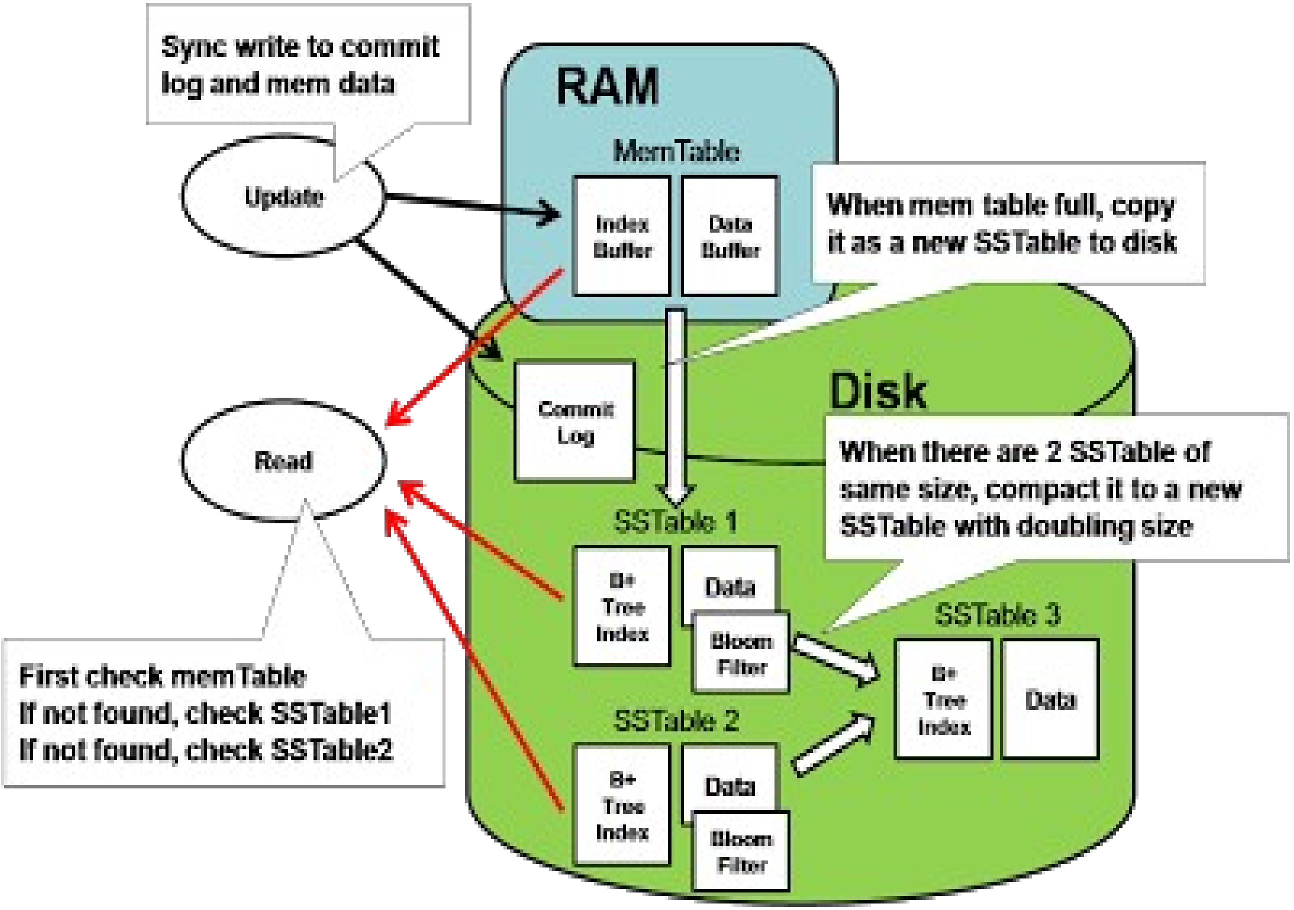
Column Family: Social

rowid	Col_name	ts	Col_value
u1	friend	v1	u10
u1	friend	v1	u13
u2	friend	v1	u10
u2	classmate	v1	u15

- One File per Column Family
- Data inside file is physically sorted
- Sparse: NULL cell does not materialize

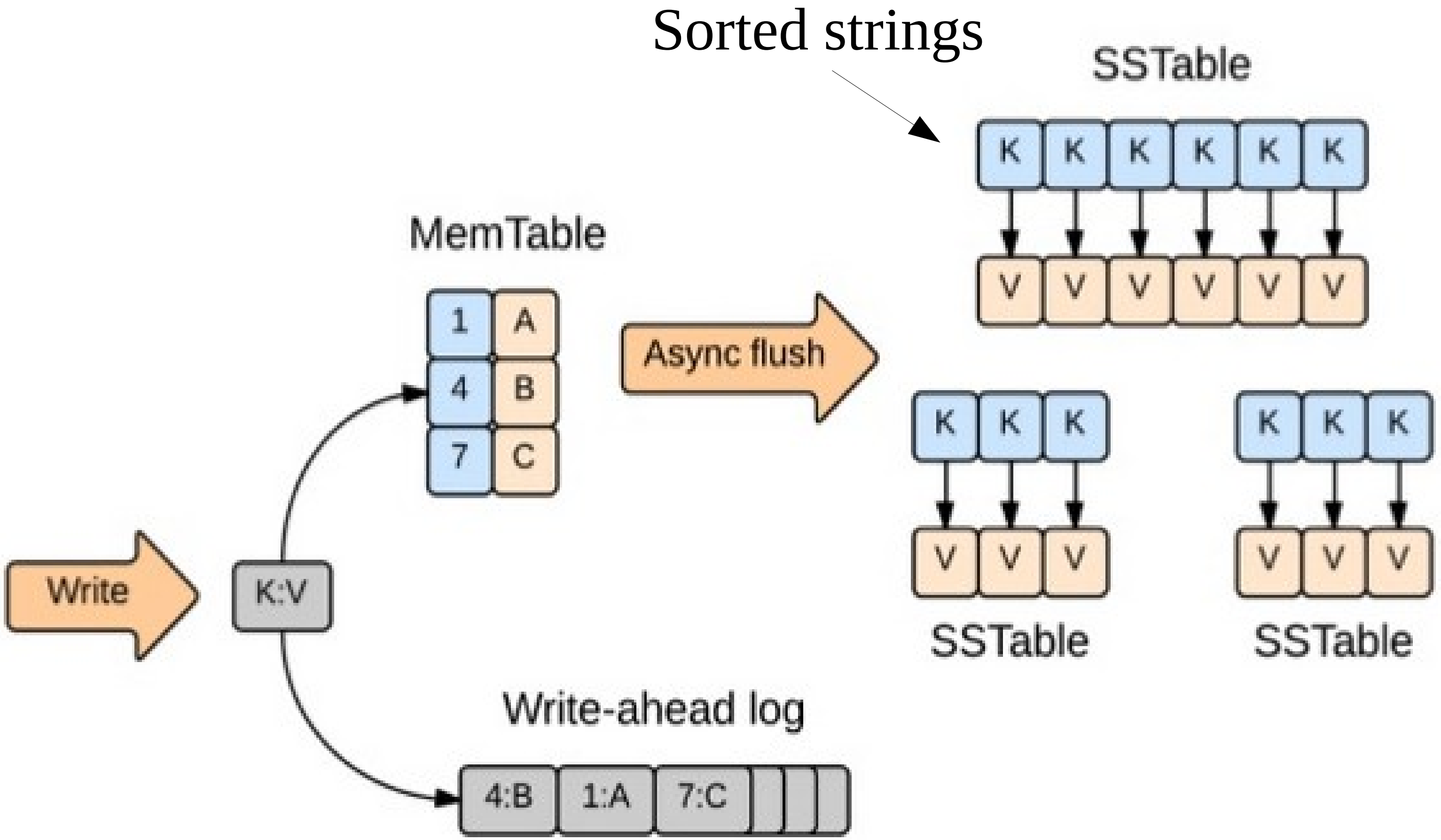
After: Ricky Ho, BigTable Model with Cassandra and Hbase. RowIds for different column families can have different types. Applications use this to build an index. In the above case, the user names could be a rowId for a ColumnFamily that has u[1-n] as a key and an empty value.

BigTable Column-Oriented Store



After: Ricky Ho, BigTable Model with Cassandra and Hbase. A key concept for processing large numbers of writes is to use sequential, append only writes. On disk, data is stored using Sorted String Tables (SSTable). These tables are never overwritten and algorithms can rely on that! Periodically they are collected and recombined into a new table using a simple sorted merge. The commit log serves as a backup, if there is a problem with in-memory tables. The concept comes originally from Google BigTable.

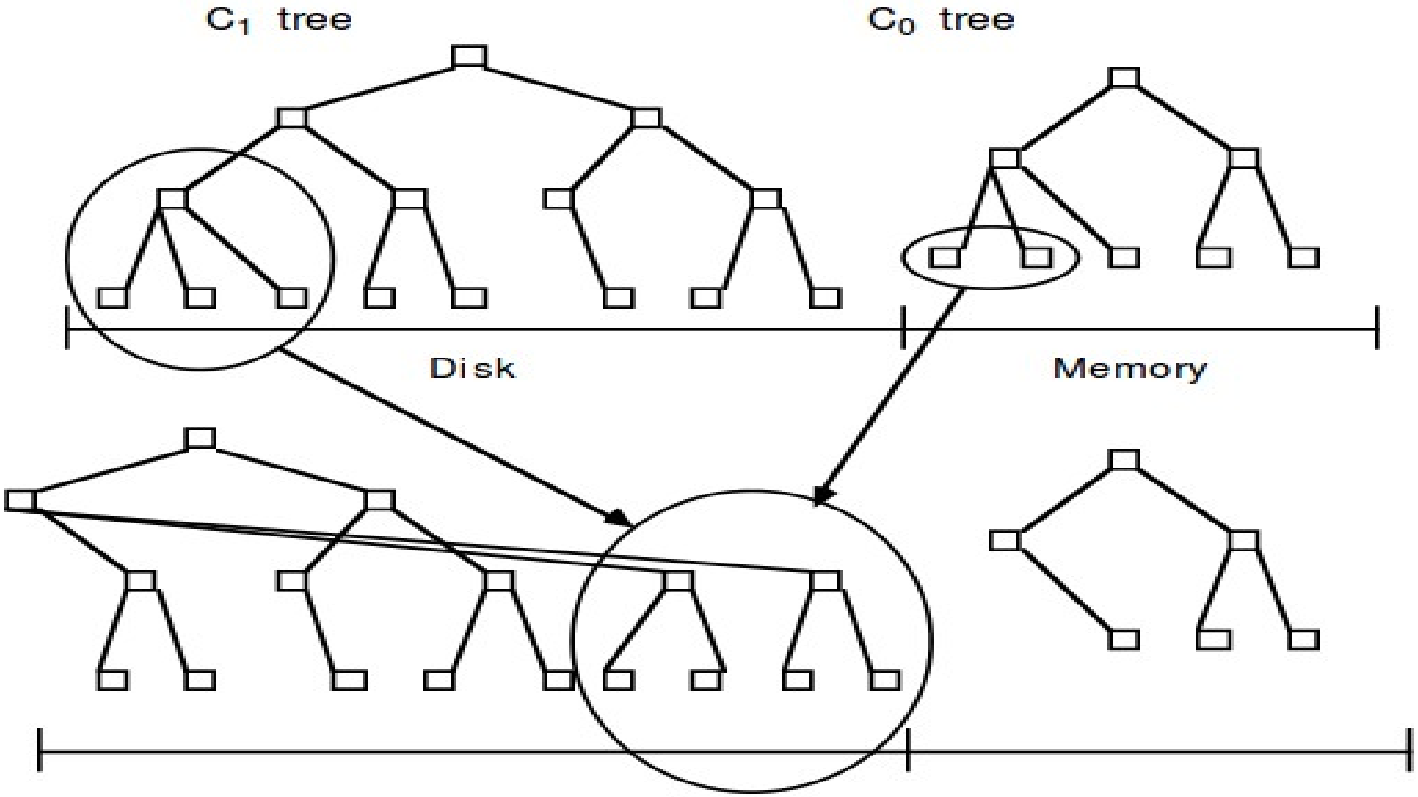
Log Structured Merge Trees (LSM)



After:Dmitri Babaev, Cassandra vs. Hbase. The diagram shows memory and disk parts as well as the WAL (write ahead log) to secure persistence and consistence in case of a crash. http://de.slideshare.net/DmitriBabaev1/cassnadra-vs-hbase

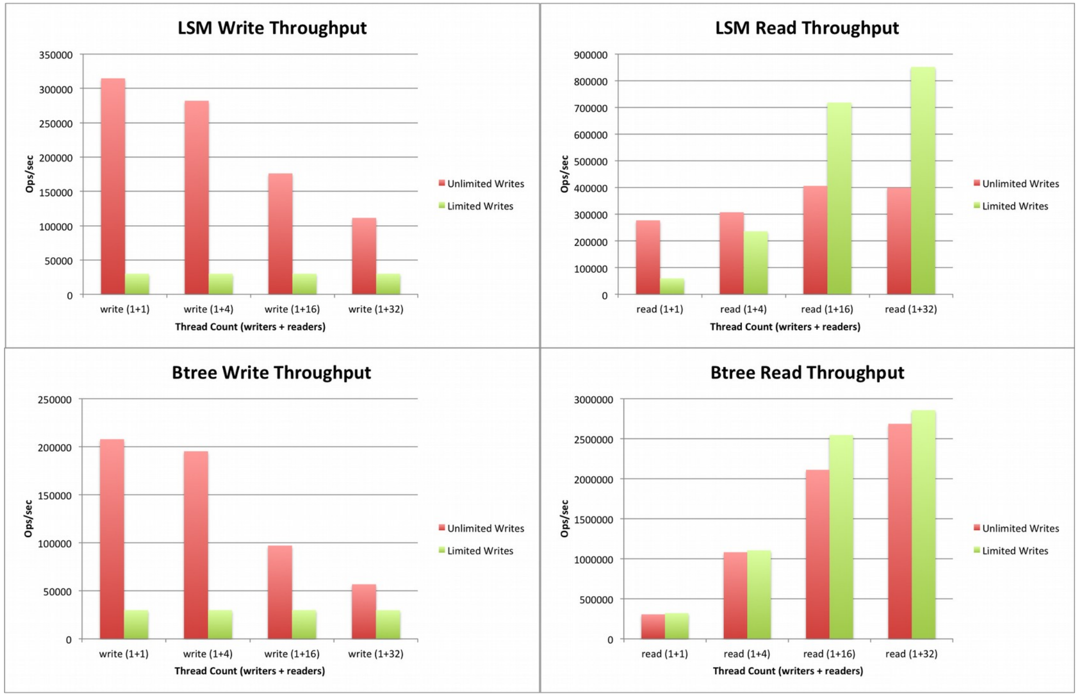
See also: Chris Lohfink, LSM Trees – A high level overview of read/write paths (with examples of updates)

Log Structured Merge Trees (LSM)



After: O'Neil, Cheng, Gawlick, O'Neil, The Log-Structured Merge-Tree (LSM-Tree). When a key space in memory becomes too big, it gets merged with on-disk content. Nothing is overwritten. Lit: Pat Helland, Immutability changes everything! (an overview of techniques based on immutable data) http://www.cidrdb.org/cidr2015/Papers/CIDR15_Paper16.pdf C0 and C1 parts are merged with a "rolling merge" technique (like merge sort).

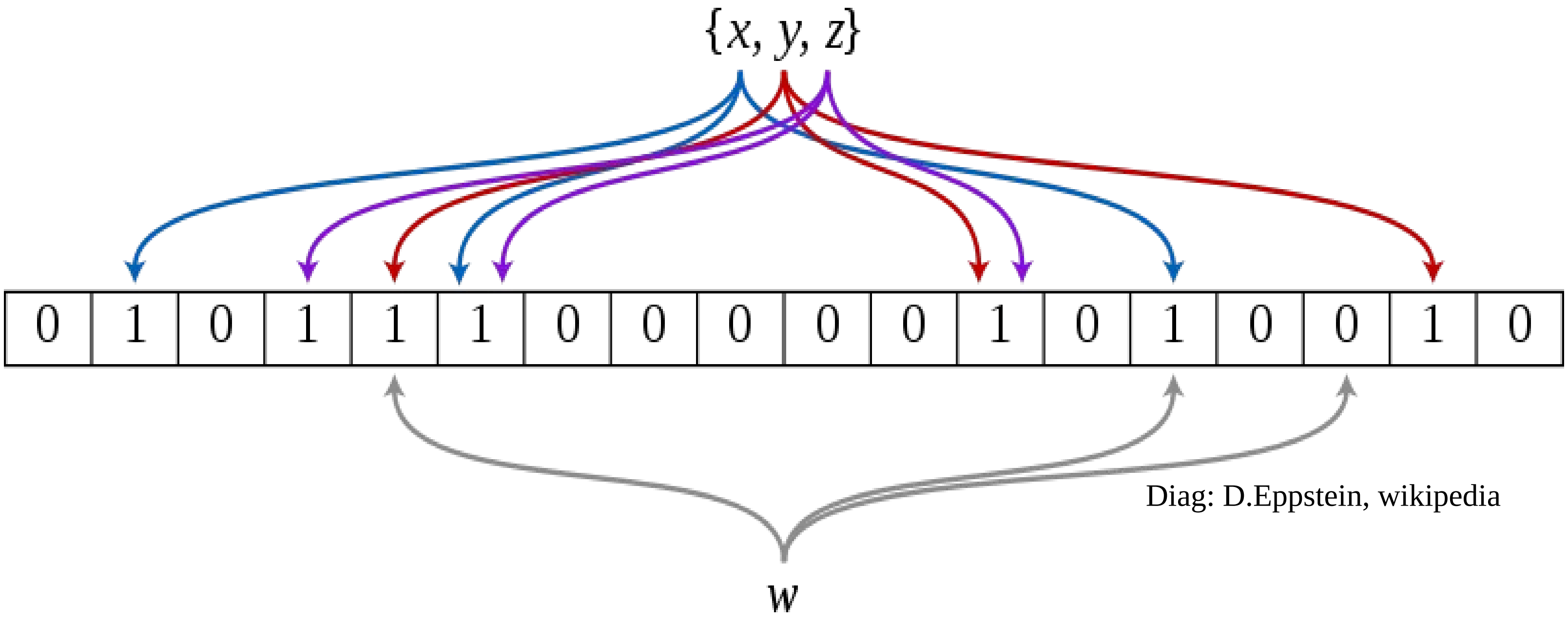
LSM vs. Btree



Write amplification: db-writes/storage writes
Read amplification: disk reads/query
Storage amplification: db-size/storage size

After: **https://tikv.github.io/deep-dive-tikv/key-value-engine/B-Tree-vs-Log-Structured-Merge-Tree.html**

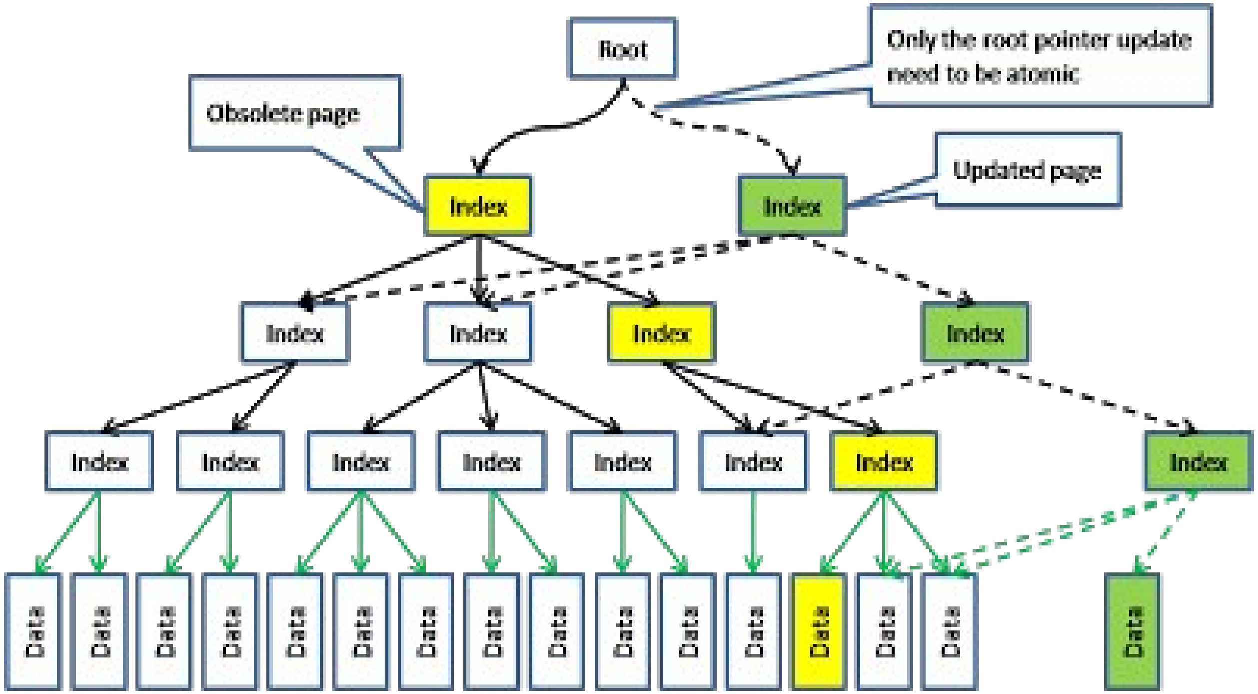
Bloom-Filter to save Disk-Access



Diag: D.Eppstein, wikipedia

After: Ricky Ho, BigTable Model with Cassandra and Hbase. The bloom filter checks, whether a key is in a SSTable. This is pretty fast and the lookup algorithm knows, that SSTables do not change later! Only if the bloom filter comes back with a negative, an expensive SSTable seek is performed. Bloom filters allow false positives though and you cannot remove an element later (which is anyway no issue here). See: http://spyced.blogspot.com/2009/01/all-you-ever-wanted-to-know-about.html by Jonathan Ellis. The filter works like this: A key is run through k different hash functions and the results are marked in a memory array. The hashes from the 3 elements X,Y,Z have been inserted into the array. A fourth one, W, is not contained in the array because one hash position is not marked (0). There are no false negatives – which is quite obvious, because inserting means marking the array!

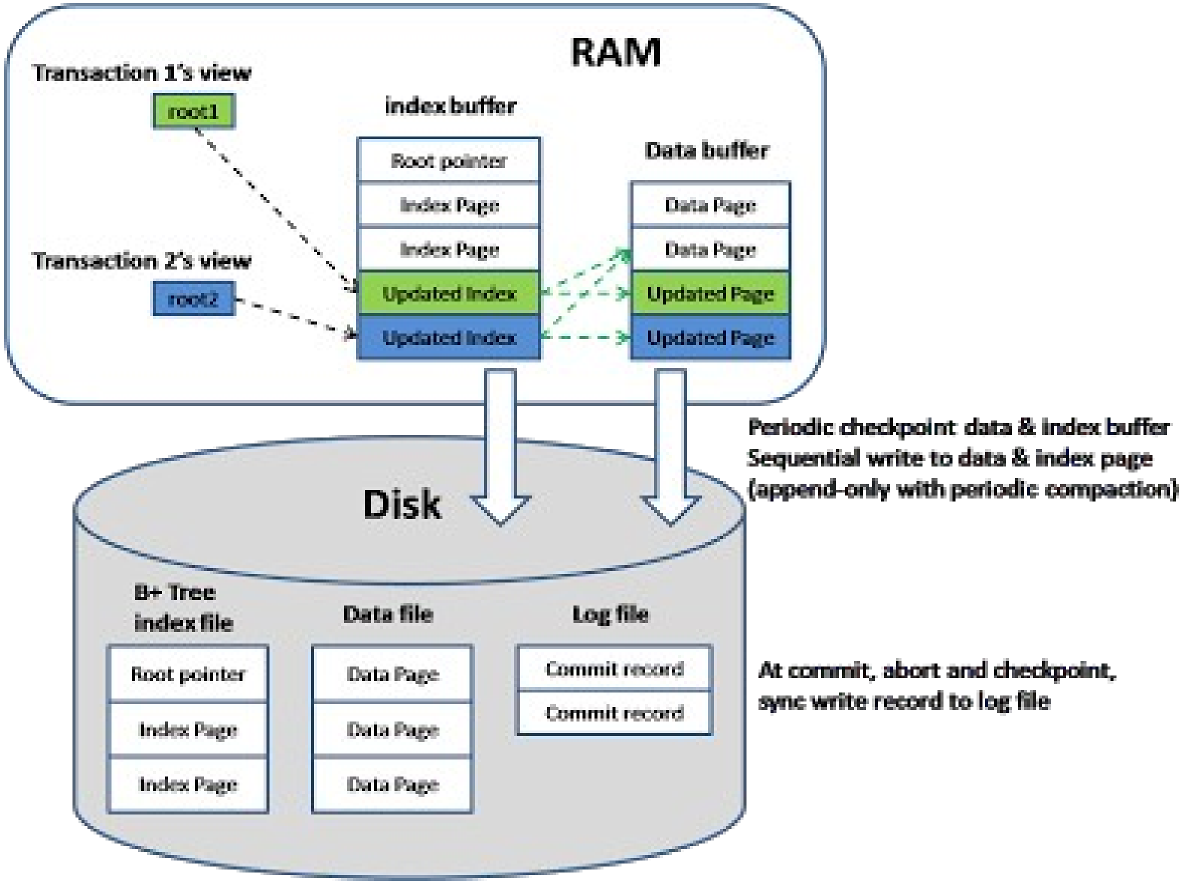
Another Example of Write-Once Logic



Copy on modify. Everyone sees his own copy of update
Finally the root pointer is swapped and everyone's view is updated
Yellow page becomes garbage over time.
File will be compacted periodically by copying to a different file.

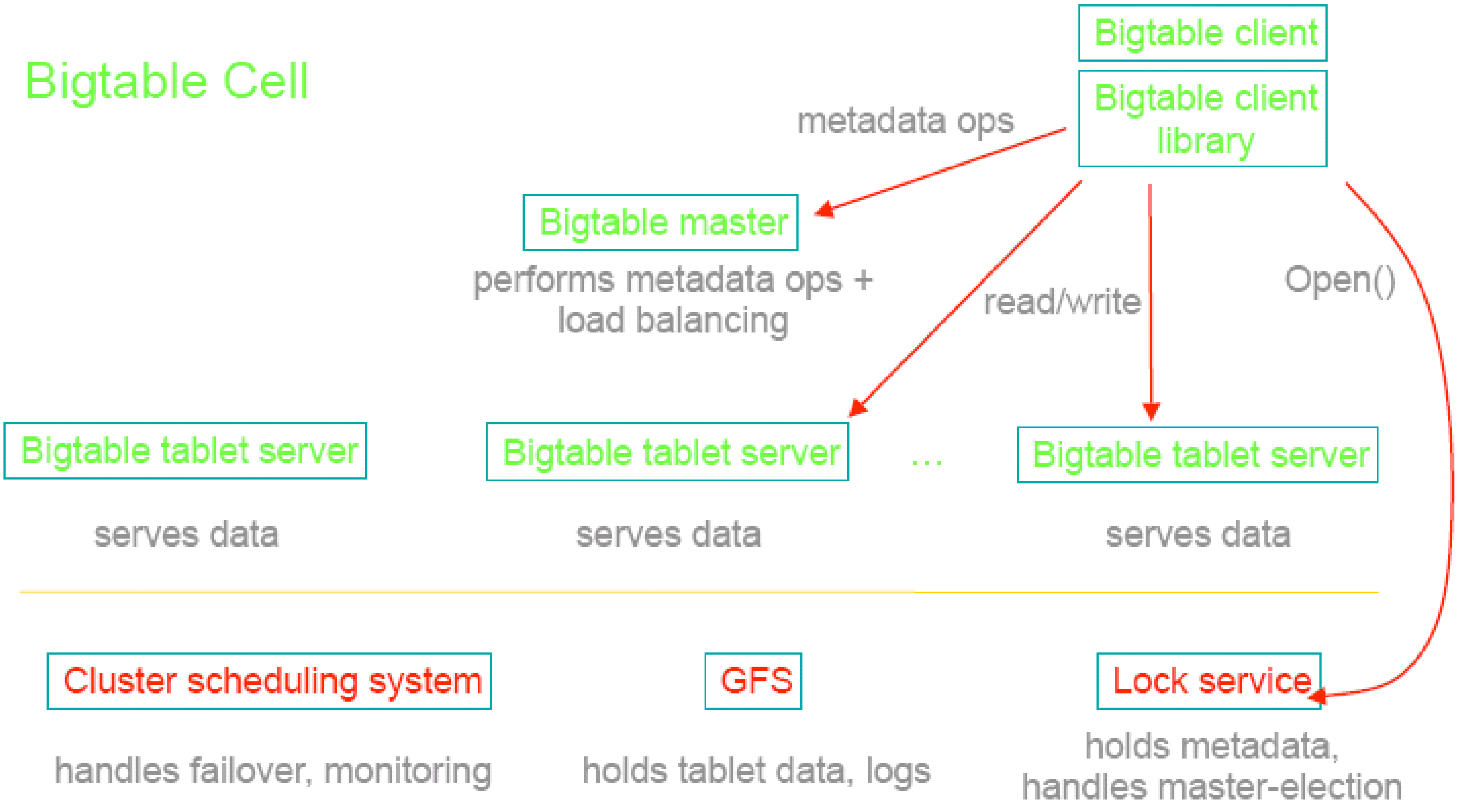
After: Ricky Ho,NOSQL patterns. This is an example form the CouchDB architecture. Just one atomic test-and-swap mechanism can consistently update a store. Write-once data structures are also easy to cache and algorithms do not need to re-read data. Also: Pat Helland, Immutability Changes Everything CDIR15,

CouchDB Storage Structure



After: Ricky Ho,NOSQL patterns. This is an example form the CouchDB architecture. Log-file management is very similar to BigTable approaches.

BigTable System Structure



Bigtable Cell

Bigtable client

Bigtable client library

metadata ops

Bigtable master

performs metadata ops + load balancing

read/write

Open()

Bigtable tablet server

serves data

Bigtable tablet server

serves data

…

Bigtable tablet server

serves data

Cluster scheduling system

handles failover, monitoring

GFS

holds tablet data, logs

Lock service

holds metadata, handles master-election

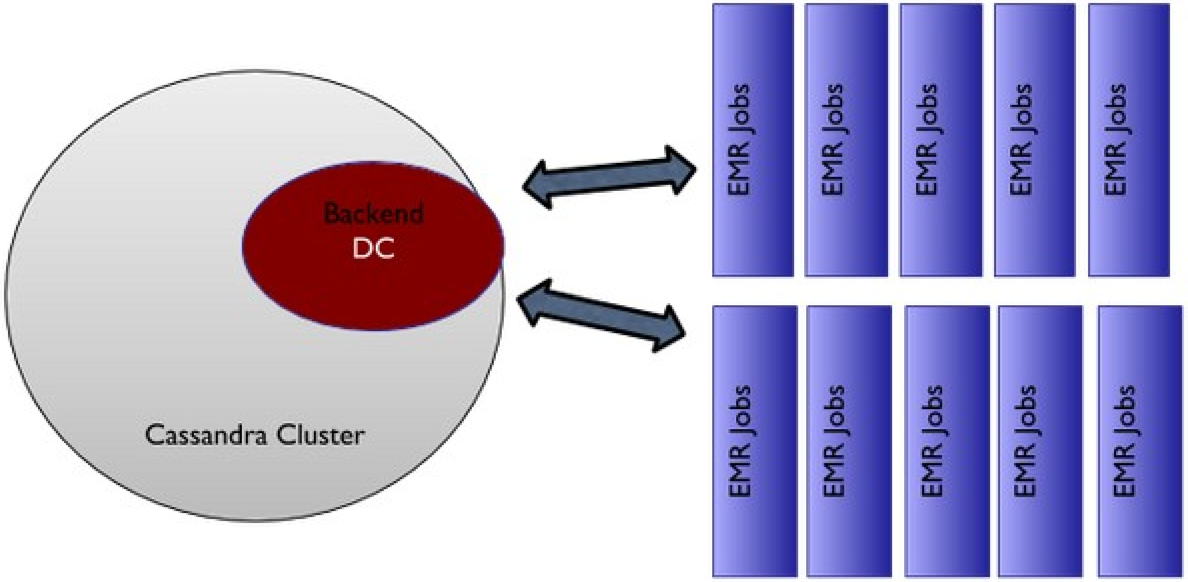
Google

Customizations and Guarantees

- R/W/N selection decides about consistency levels
- Only atomic update of a row.
- No multi-row transactions
- Indexes for reverse lookup by applications
- Danger of overwriting intermediate changes (lost update)
- Failed updates (write-quorum not reached) leave some replicas updated. Through anti-entropy copying, those can distributed through the store. Clients need to repeat failed updates and deal with duplicates.

After: Ricky Ho, BigTable Model with Cassandra and Hbase. These trade-offs are quite typical for eventually consistent stores. Surprisingly many applications can live successfully with those restrictions.

Cassandra on AWS



After: Jorge Rodriguez, Global Cassandra on AWS EC2 at BloomReach. The problem shown is the combination of an elastic resource (EMR) with a fixed cluster. BloomReach finally used a request throttling strategy to ease cluster load. They describe more optimizations in their paper, e.g. elastic cassandra.

NewSQL: The Comeback of SQL?

- SQL Overlays for query processing
- Unified DB storage types (key/value, doc, columns, tables)
- Support for advanced isolation and consistency models
- Example: CockroachDB two excellent papers:
  https://www.cockroachlabs.com/blog/consistency-model/

https://www.cockroachlabs.com/blog/cockroachdb-on-rocksd/

The first paper tries to give a unified view on serializable and linearizablle features in NoSQL/NewSQL Dbs. The second tells the difference between higher level DB and storage engines.

Dealing with eventual consistency and maintaining several different NoSQL databases within one application becomes very cumbersome...

Beyond Relaxed Consistency...

Beyond Relaxed Consistency...

- Order insensitive (CALM) processing. EC programs that follow monotonic logic principles

- State-based CRDTs (Converging replicated Data Types)
- Operation-based CRDTs

If distributed consensus is too expensive, and relaxing consistency not good enough, a look at algorithms and data structures which are insensitive to ordering might pay off. Lit: M.Shapiro: A comprehensive study of CRDTs

The CALM Principle

> “the tight relationship between Consistency And Logical Monotonicity. Monotonic programs guarantee eventual consistency under any interleaving of delivery and computation. By contrast, non-monotonicity—the property that adding an element to an input set may revoke a previously valid element of an output set—requires coordination schemes that “wait” until inputs can be guaranteed to be complete.”

Peter Alvaro, Neil Conway, Joseph M. Hellerstein, William R. Marczak, Consistency Analysis in Bloom: a CALM and Collected Approach. Monotonic program parts are safe under eventual consistency (P.Bailis)

CALM Operations

Logically monotonic:

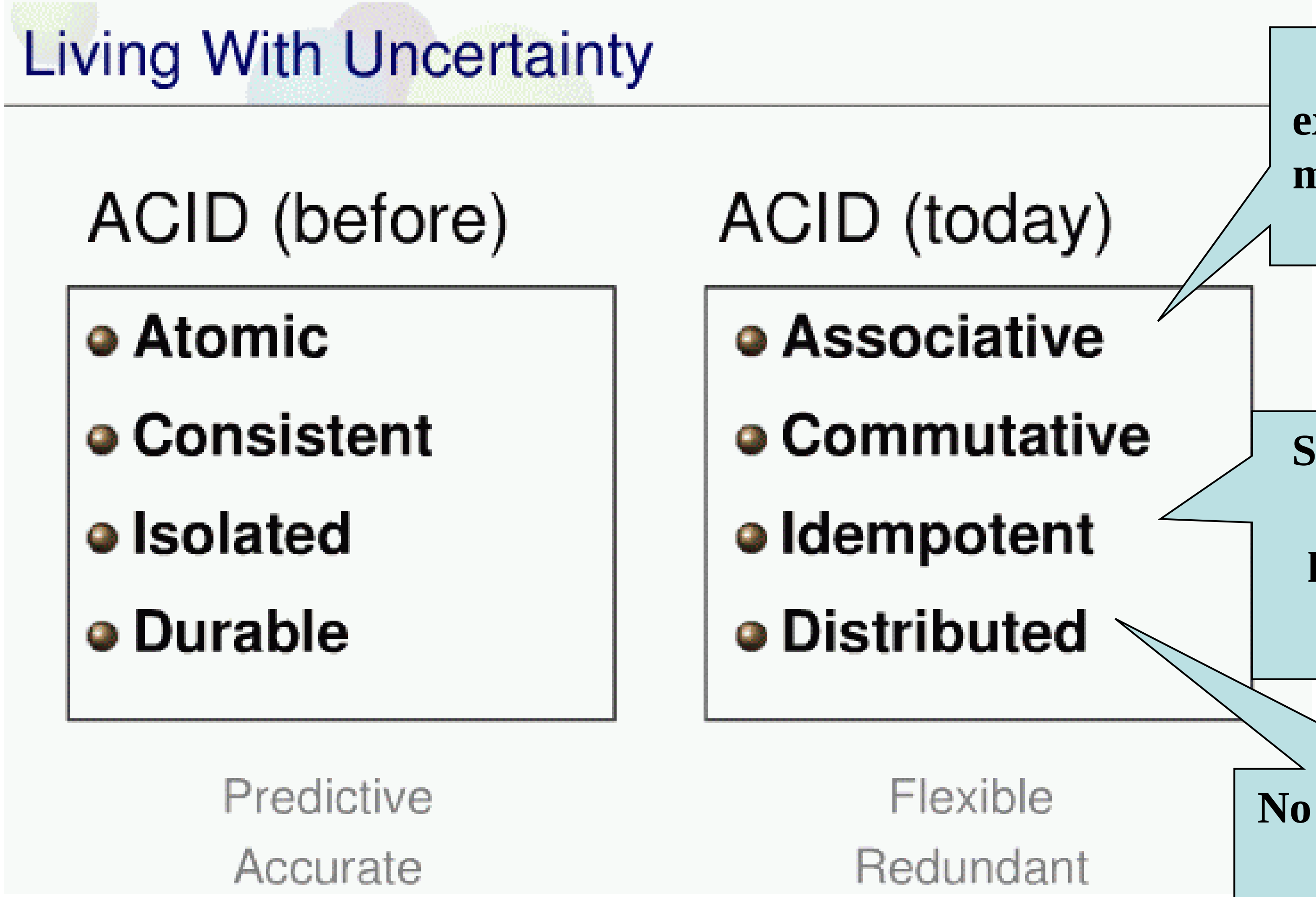
- initializing variables,
- accumulating set members,
- testing a threshold condition

non-monotonic:

- overwriting variables,
- set deletion,
- resetting counter
- negation

P.Bailis, A.Ghodsi, Eventual Consistency Today: Limitations, Extensions, and Beyond

CALM Design Patterns



Living With Uncertainty

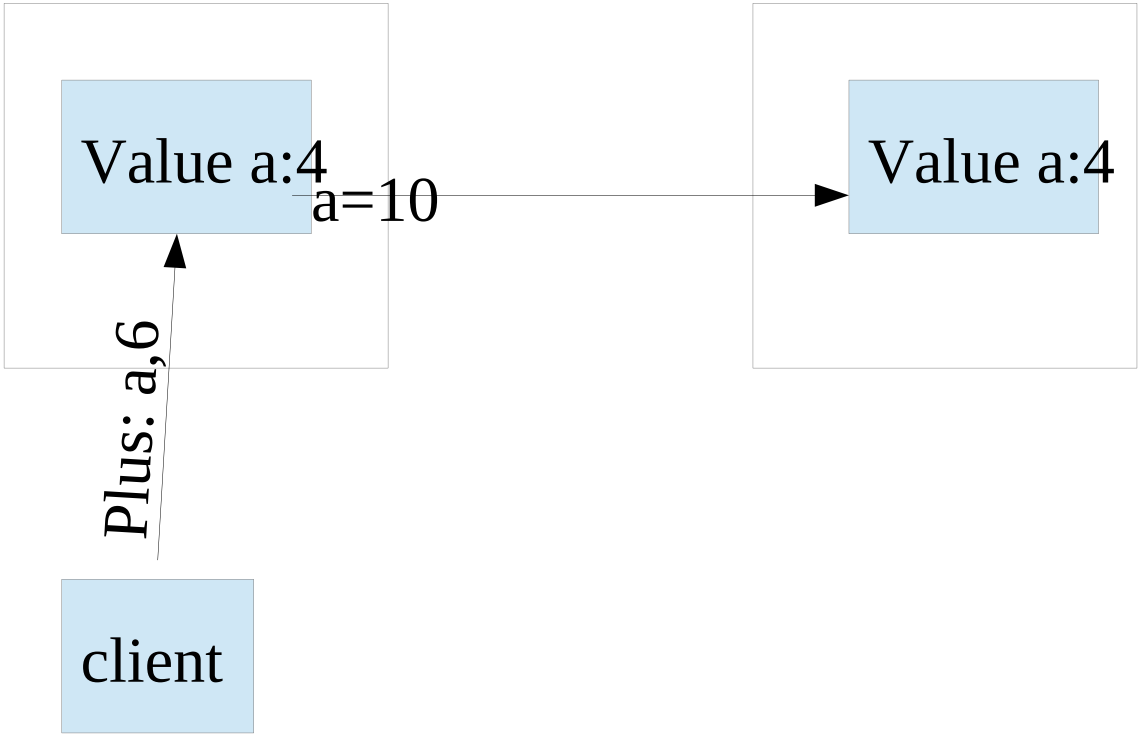
ACID (before)	ACID (today)
Atomic	Associative
Consistent	Commutative
Isolated	Idempotent
Durable	Distributed
Predictive Accurate	Flexible Redundant

Order of execution/arguments does not matter!

Service is either a natural or our protool needs to achieve it!

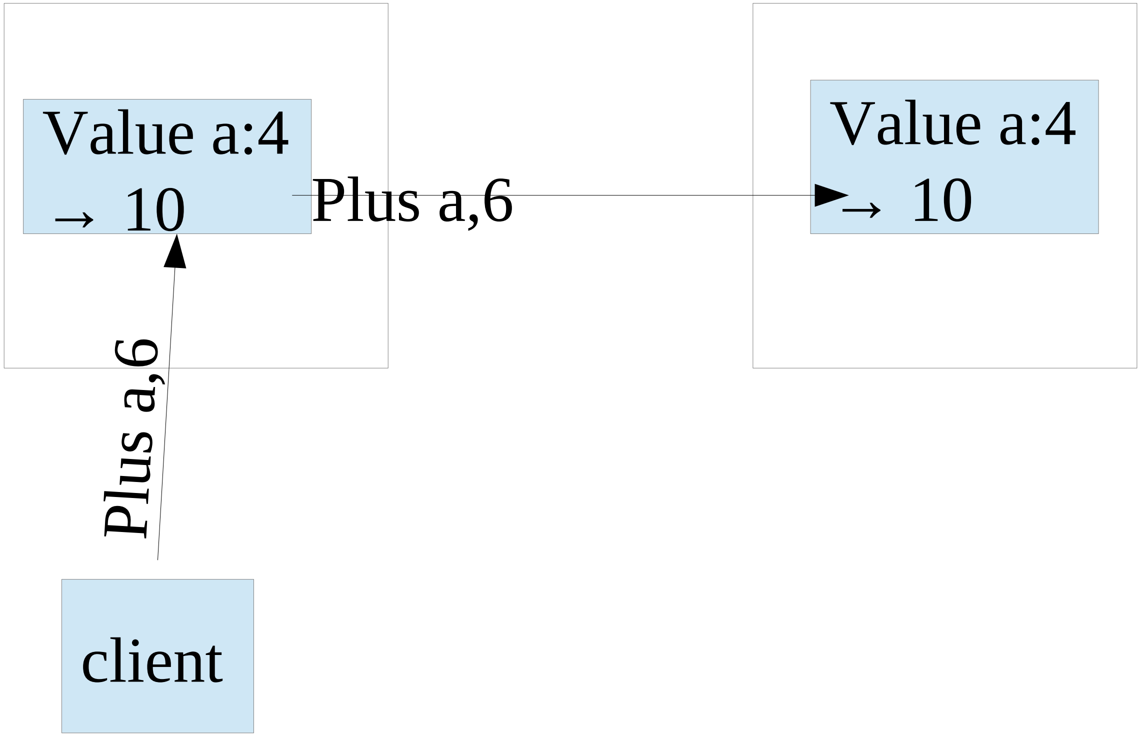
No synchronization Needed!

State-based CRDTs



State-based CRDTs calculate the new result at one node and then propagate the result to replicas. The data structure needs to be commutative, associative and idempotent. This is e.g. true for sets.
See: Arnout Engelen, CRDTs illustrated, Strangeloop 2015

Operation-based CRDTs



Operation-based CRDTs send the requested operation to each replica and the results are calculated locally. The operations need to be commutative with “exactly once” semantics (idempotent) and in fifo order. Those delivery guarantees are rather hard to guarantee and therefore state-based CRDTs are currently more popular.
See: Arnout Engelen, CRDTs illustrated, Strangeloop 2015

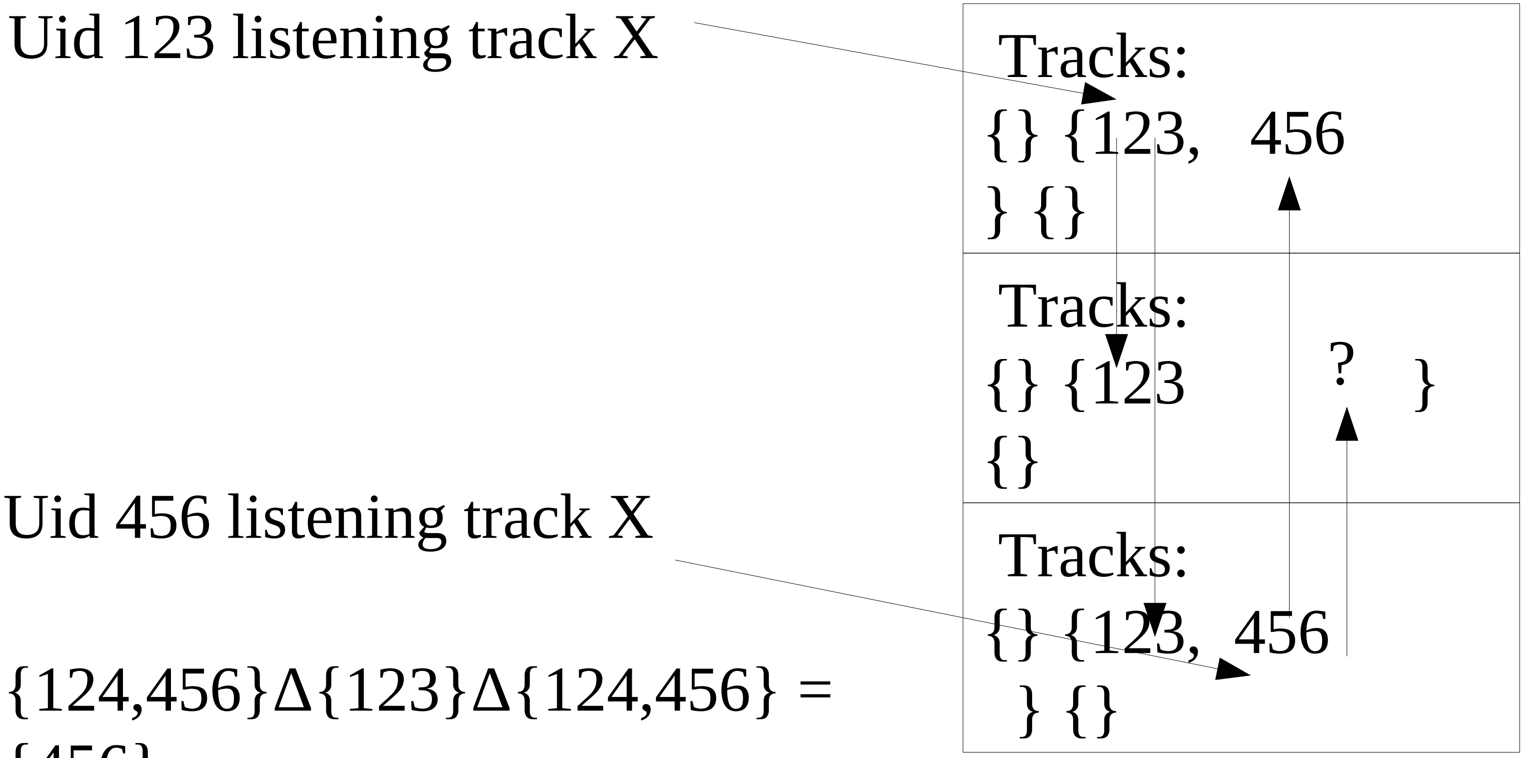
“Bending the Problem”

”A key property of these advances is that they separate data store and application-level consistency concerns. While the underlying store may return inconsistent data at the level of reads and writes, CALM, ACID 2.0 and CRDT appeal to higher-level consistency criteria, typically in the form of application-level invariants that the application maintains.

Instead of requiring that every read and write to and from the data store is strongly consistent, the application simply has to ensure a semantic guarantee (such as "the counter is strictly increasing")—granting considerable leeway in how reads and writes are processed.”

(P.Bailis et.al.)

“Bending the Problem”: Counting Track-Views



Uid 123 listening track X

Tracks:
{} {123, 456
} {}

Tracks:
{} {123 ? }
{}

Uid 456 listening track X

Tracks:
{} {123, 456
} {}

$\{124,456\}\Delta\{123\}\Delta\{124,456\} = \{456\}$

Peter Bouton, Soundcloud, Consistency without consensus in production systems, Strangeloop 2015. Symmetric difference allows to find missing elements. Fixing is idempotent.

Examples of CRDTs

Counters:

Grow-only counter (merge = max(values); payload = single integer)

Positive-negative counter (consists of two grow counters, one for increments and another for decrements)

Registers:

Last Write Wins -register (timestamps or version numbers;

merge = max(ts); payload = blob)

Multi-valued -register (vector clocks; merge = take both)

Sets:

Grow-only set (merge = union(items); payload = set; no removal)

Two-phase set (consists of two sets, one for adding, and another for removing; elements can be added once and removed once)

Unique set (an optimized version of the two-phase set)

Last write wins set (merge = max(ts); payload = set)

Positive-negative set (consists of one PN-counter per set item)

Observed-remove set

From: “Distributed Systems for fun and profit”

Distributed Configuration and Orchestration

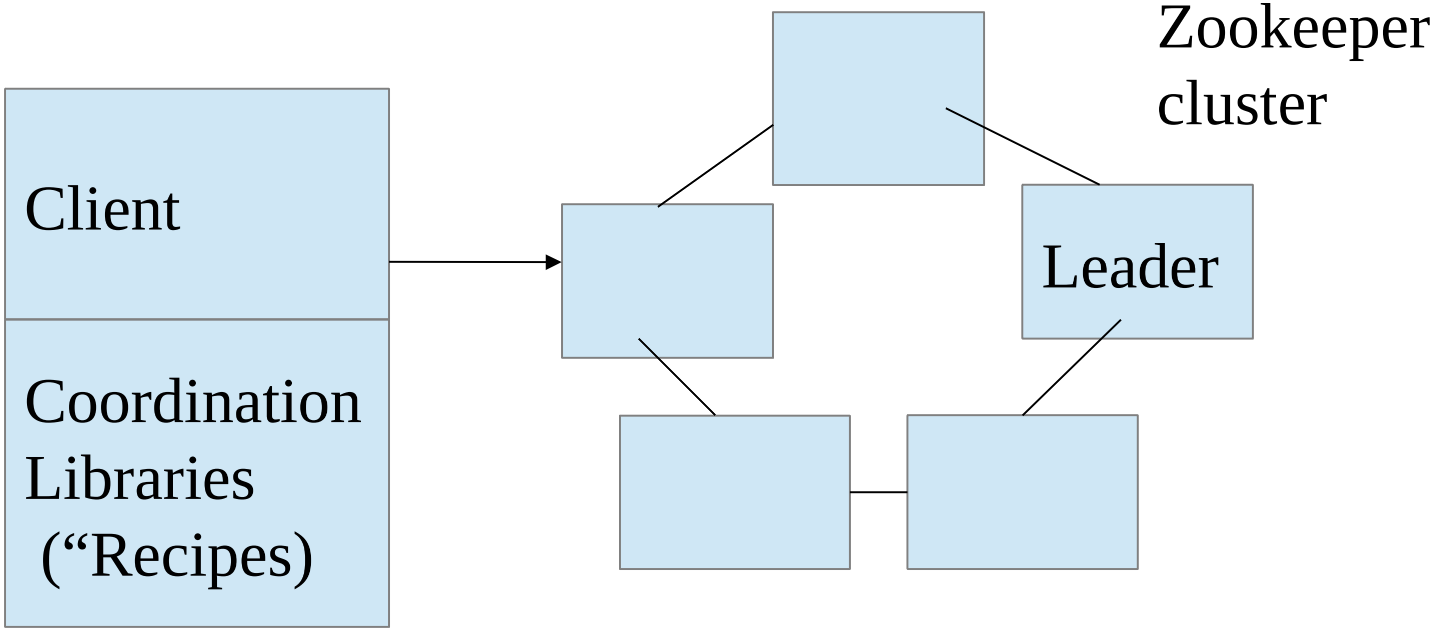
When the power in a warehouse computing center is turned on....

“An Oracle is needed”...

- Configuration changes and notifications
- Update of failed machines
- Dynamically integrate new machines/deconfiguration
- Elastic configuration with partial failures
- API for watches, callbacks, automatic file removal, triggers
- Simple data model (directory tree model)
- High performance, highly available in-memory cluster solution
- No locks for updates but total ordering of requests for all cluster replicas
- All replicas answer reads
- wait-free implementation of coordination service with client API performing locks, leader-selection etc.

Benjamin Reed, Zookeeper, the making of.

Becomes Distributed Coordination...



Zookeeper
cluster

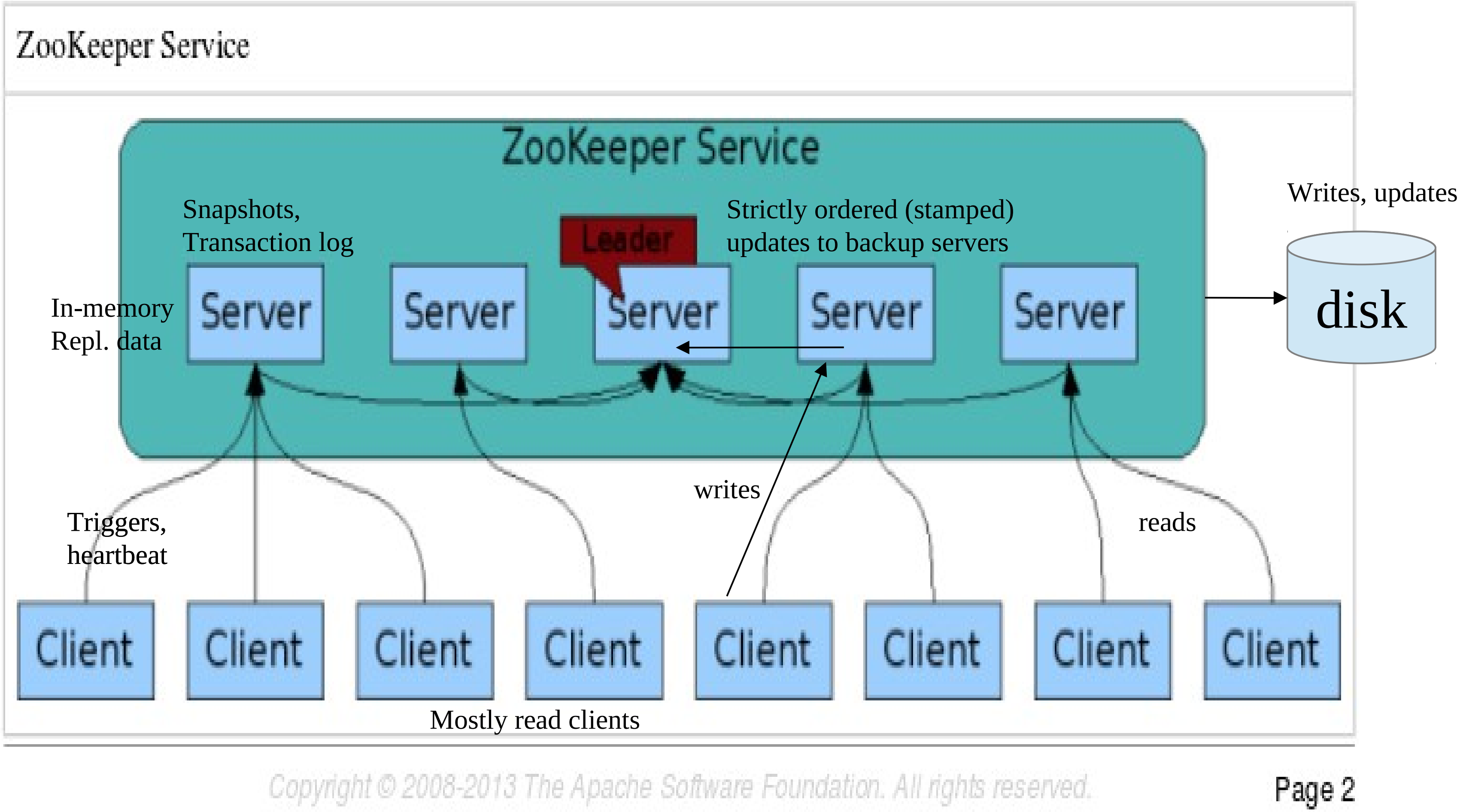
Client

Coordination
Libraries
(“Recipes)

Leader

wait-free implementation (request ordering) of coordination service with client API implementing locks queues, barriers, leader-selection, group membership etc. From: Benjamin Reed, Zookeeper, the making of.

Zookeeper



Snapshots,
Transaction log

Strictly ordered (stamped)
updates to backup servers

Writes, updates

In-memory
Repl. data

writes

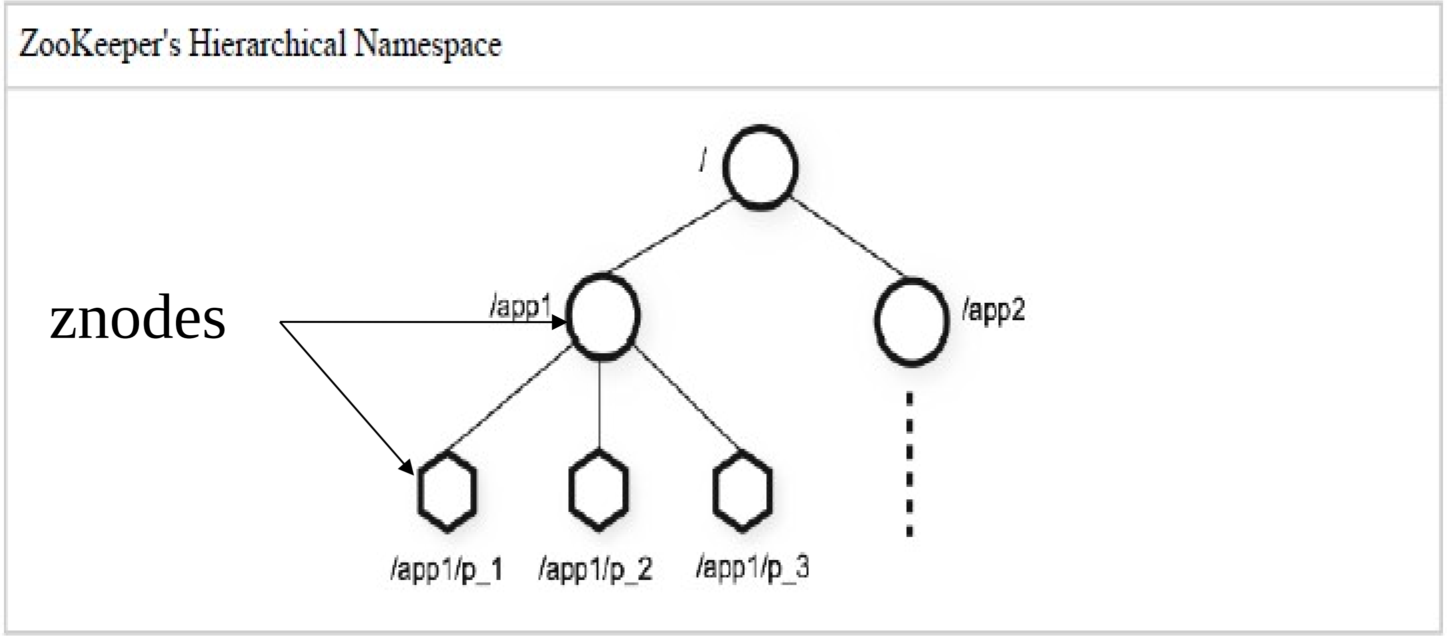
Triggers,
heartbeat

reads

Mostly read clients

TA-log, snapshot,

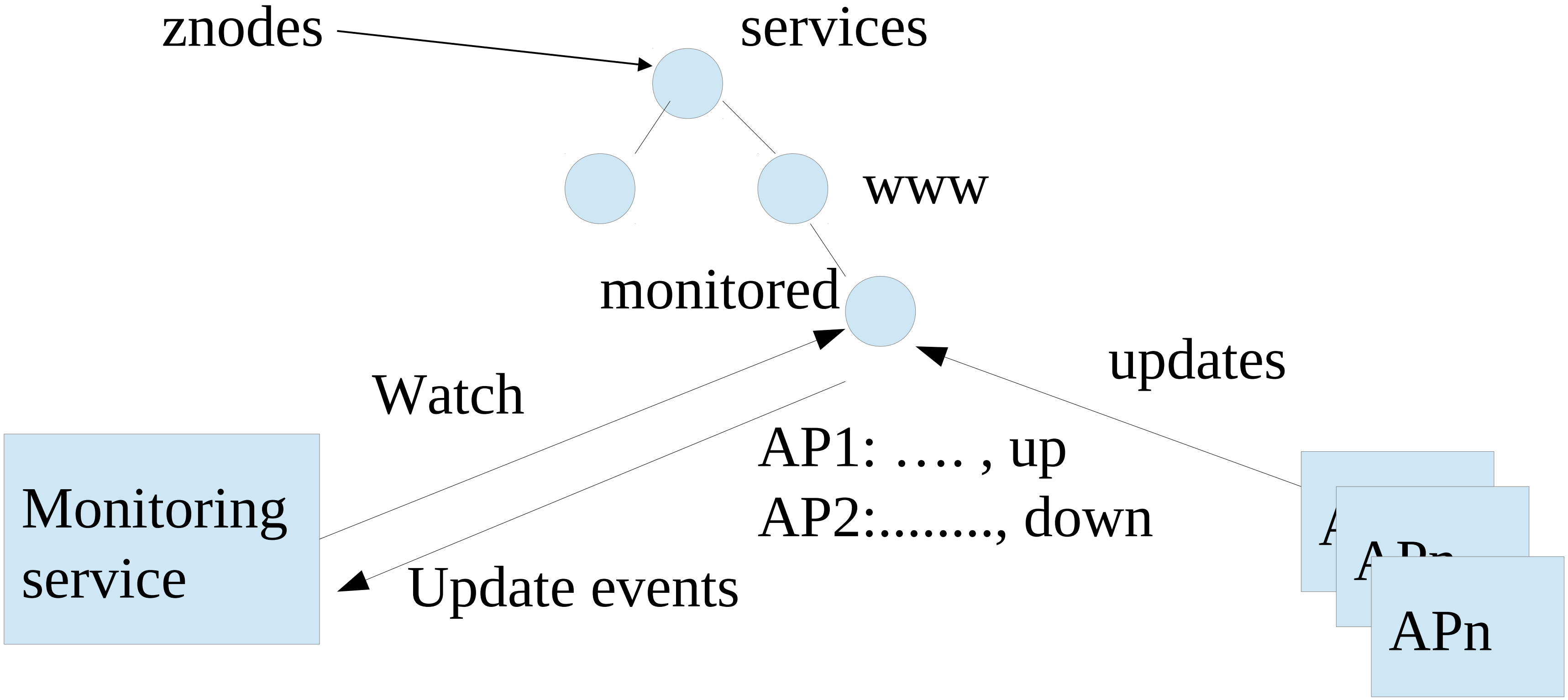
Directory-like Namespace



ZooKeeper's Hierarchical Namespace

Znodes are like files which can be directories as well. They can be updated atomically. Znodes are versioned and changes can be "watched" by clients.

Use Case 1: Service Monitoring



znodes

services

www

monitored

updates

Watch

AP1: …. , up

AP2:........, down

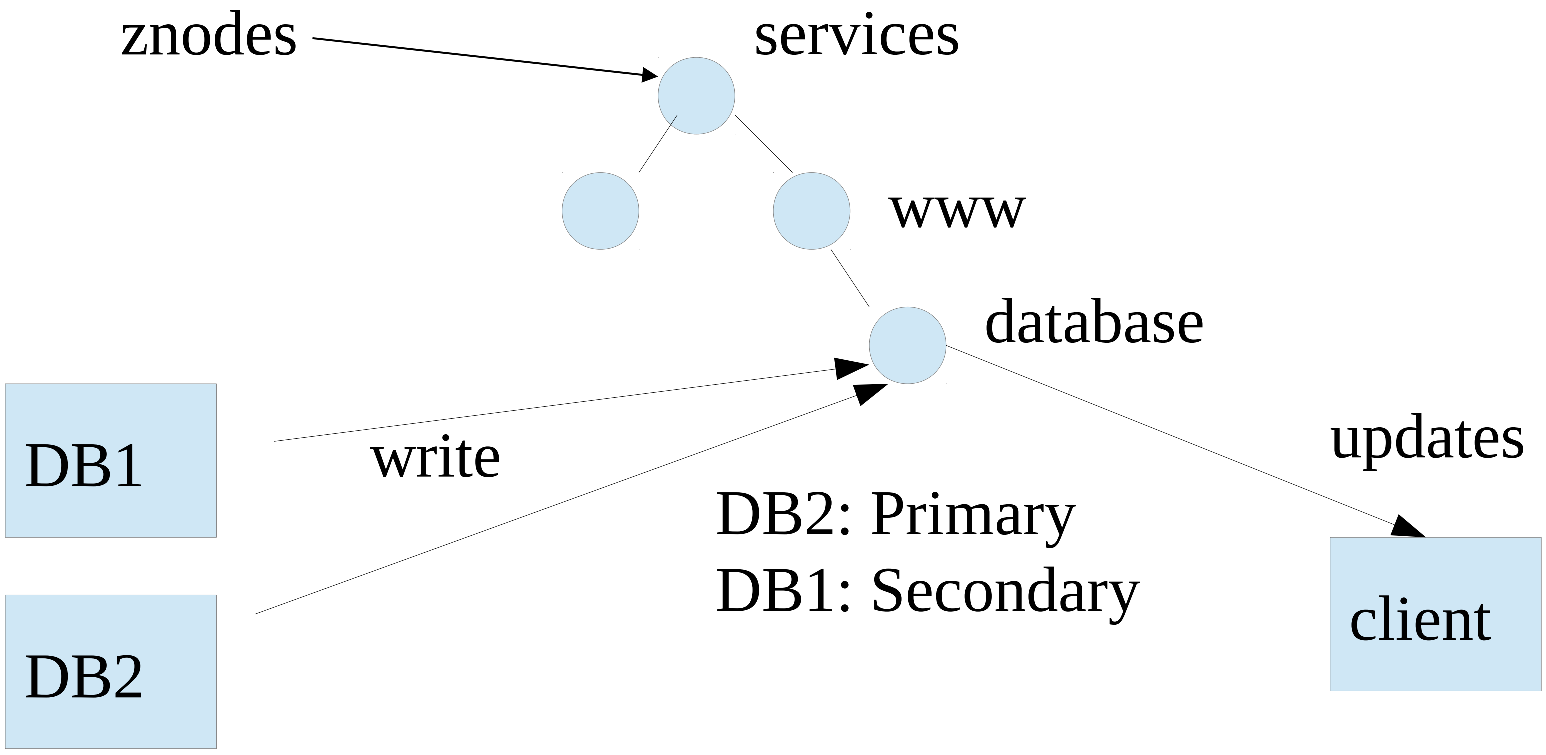
Monitoring service

Update events

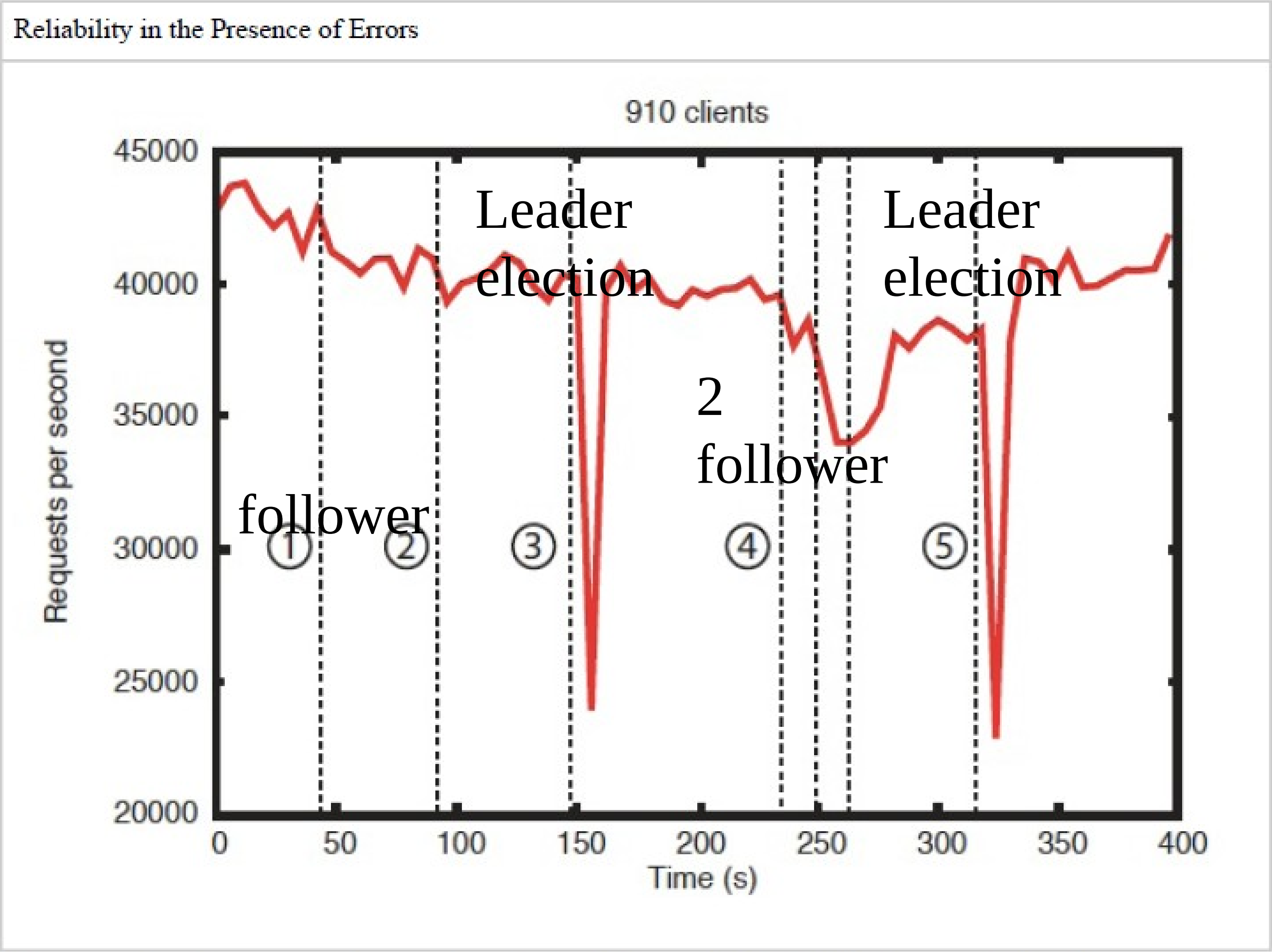
Updates are atomic. Events delivered by server order.

The coordination service keeps state in a replicated DB.

Use Case 1: Self-Organized Boot



Additional protocols allow leader election, service location etc. Locking must be supported too. Configuration files do not work in a cluster environment



Reliability in the Presence of Errors

follower

Leader election

2 follower

Leader election

Liveness and Correctness

- Sequential Consistency - Updates from a client will be applied in the order that they were sent.

- Atomicity - Updates either succeed or fail. No partial results.

- Single System Image - A client will see the same view of the service regardless of the server that it connects to.

- Reliability - Once an update has been applied, it will persist from that time forward until a client overwrites the update.

- Timeliness - The clients view of the system is guaranteed to be up-to-date within a certain time bound.

Zookeeper API

create: creates a node at a location in the tree

delete: deletes a node

exists: tests if a node exists at a location

get data: reads the data from a node

set data: writes data to a node

get children: retrieves a list of children of a node

sync: waits for data to be propagated

Primary-Order Atomic Broadcast with Zab

- A primary sends non-commutative, incremental state changes to backup units
- The order of incremental changes is kept even in case of a primary crash
- Multiple outstanding requests are possible
- An identification scheme prevents re-ordering of updates
- A synchronization phase prevents new updates from being stored before old updates are delivered.

Flavio P. Junqueira, Benjamin C. Reed, and Marco Serafini, Zab: High-performance broadcast for primary-backup systems

Consistency Requirements for ABCast

Validity: If a correct process broadcasts a message, then all correct processes
will eventually deliver it.

No gaps!

Uniform Agreement: If a process delivers a message, then all correct processes
eventually deliver that message.

Uniform Integrity: For any message m, every process delivers m at most once,
and only if m was previously broadcast by the sender of m.

Same order!

Uniform Total Order: If processes p and q both deliver messages m and m0,
then p delivers m before m0 if and only if q delivers m before m0.

Primary Order

FiFo order!

Local primary order: If a primary broadcasts (v, z) before it broadcasts (v'; z'), then a process that delivers (v,z) must have delivered (v',z') before (v,z).

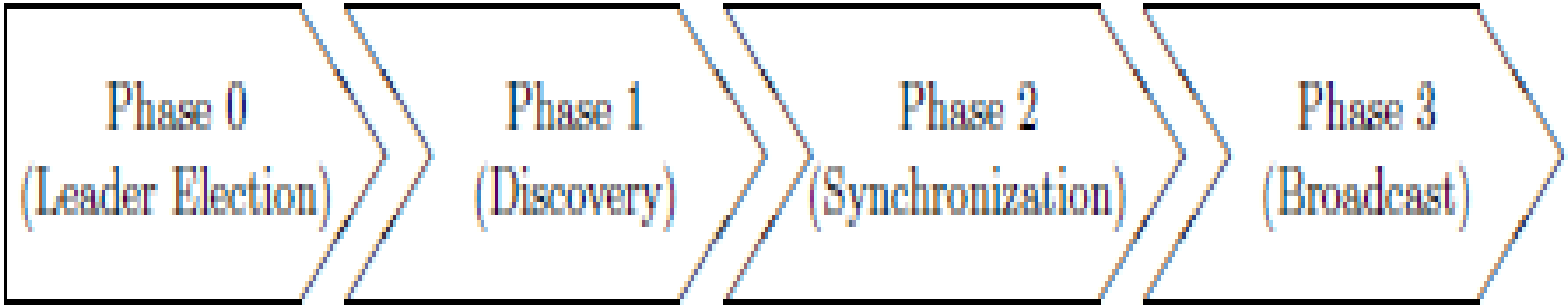
Global primary order: Suppose a primary Pi broadcasts (v,z), and a primary Pj > Pi broadcasts (v',z'). If a process delivers both (v,z) and (v',z'), then it must deliver (v,z) before (v',z').

No gaps!

Primary integrity: If a primary Pe broadcasts (v,z) and some process delivers (v',z') which was broadcast by Pe' < Pe, then Pe must have delivered (v',z') before broadcasting (v,z).

After: ZooKeeper's atomic broadcast protocol:Theory and practice, Andre Medeiros

Zab Protocol



Peers try to find a leader, store votes in vol.mem.

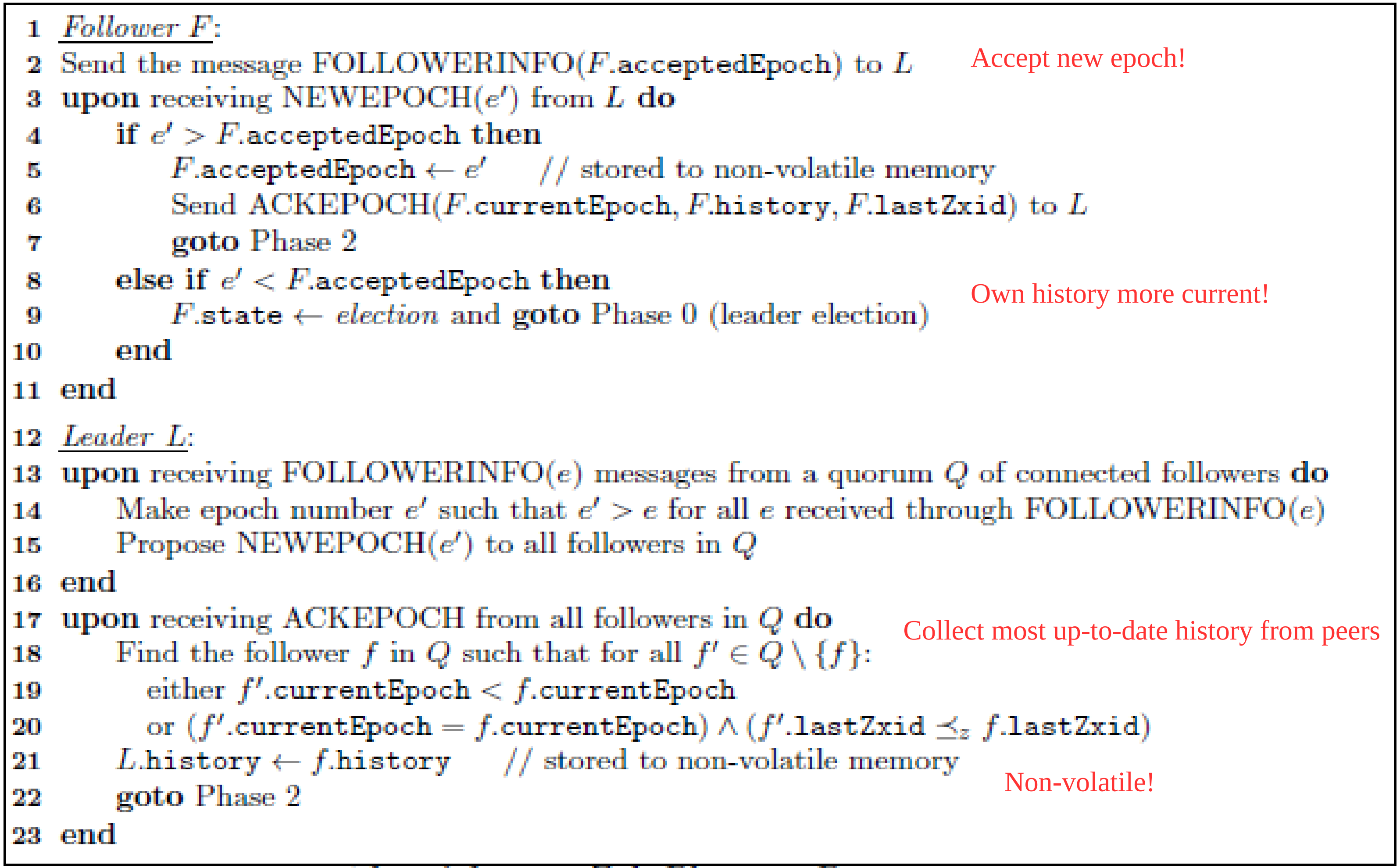
Leader tries to find the most up-to-date sequence of TA's in a quorum. New epoch defined

Leader suggests TA's to followers who miss some. Quorum acceptance establishes leader

Broadcast layer is ready to perform new state changes under the new leader.

After: ZooKeeper's atomic broadcast protocol:Theory and practice, Andre Medeiros.

Zab Protocol Phase 1: Discovery



```
Follower F:
Send the message FOLLOWERINFO($F$.acceptedEpoch) to $L$
upon receiving NEWEPOCH($e'$) from $L$ do
    if $e' > F$.acceptedEpoch then
        $F$.acceptedEpoch $\leftarrow e'$    // stored to non-volatile memory
        Send ACKEPOCH($F$.currentEpoch, $F$.history, $F$.lastZxid) to $L$
        goto Phase 2
    else if $e' < F$.acceptedEpoch then
        $F$.state $\leftarrow$ election and goto Phase 0 (leader election)
    end
end
Leader L:
upon receiving FOLLOWERINFO($e$) messages from a quorum $Q$ of connected followers do
    Make epoch number $e'$ such that $e' > e$ for all $e$ received through FOLLOWERINFO($e$)
    Propose NEWEPOCH($e'$) to all followers in $Q$
end
upon receiving ACKEPOCH from all followers in $Q$ do
    Find the follower $f$ in $Q$ such that for all $f' \in Q \setminus \{f\}$:
      either $f'$.currentEpoch $<$ $f$.currentEpoch
      or ($f'$.currentEpoch $=$ $f$.currentEpoch) $\wedge$ ($f'$.lastZxid $\preceq_z$ $f$.lastZxid)
    $L$.history $\leftarrow$ $f$.history    // stored to non-volatile memory
    goto Phase 2
end
```

Accept new epoch!

Own history more current!

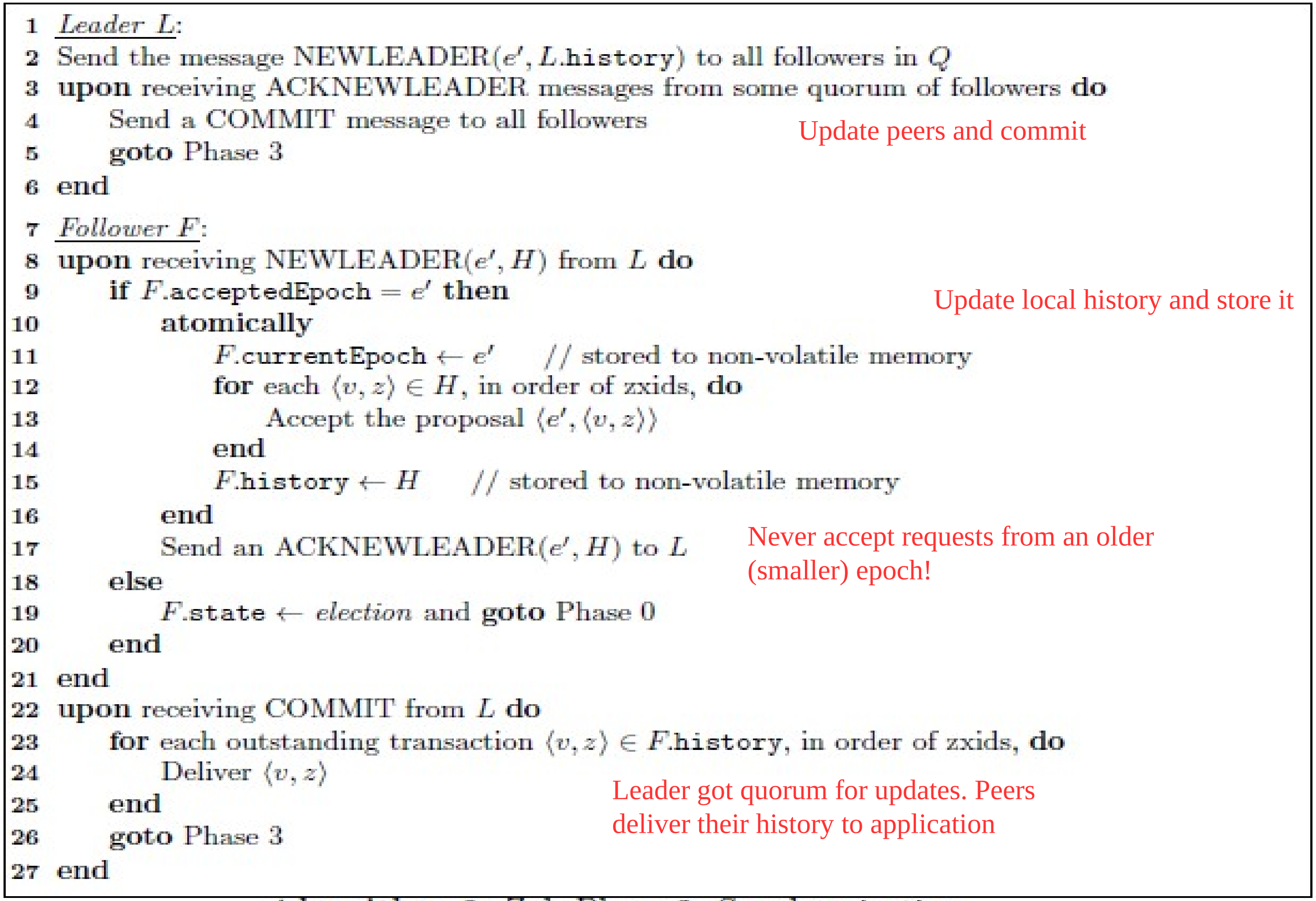
Collect most up-to-date history from peers

Non-volatile!

Algorithm 1: Zab Phase 1: Discovery.

After: ZooKeeper's atomic broadcast protocol:Theory and practice, Andre Medeiros. Peers try to find a leader, store votes in vol.mem.

Zab Protocol Phase 2: Synchronization



```
Leader L:
Send the message NEWLEADER($e'$, L.history) to all followers in Q
upon receiving ACKNEWLEADER messages from some quorum of followers do
    Send a COMMIT message to all followers
    goto Phase 3
end
Follower F:
upon receiving NEWLEADER($e'$, H) from L do
    if F.acceptedEpoch = $e'$ then
        atomically
            F.currentEpoch ← $e'$      // stored to non-volatile memory
            for each $\langle v, z \rangle \in H$, in order of zxids, do
                Accept the proposal $\langle e', \langle v, z \rangle \rangle$
            end
            F.history ← H      // stored to non-volatile memory
        end
        Send an ACKNEWLEADER($e'$, H) to L
    else
        F.state ← election and goto Phase 0
    end
end
upon receiving COMMIT from L do
    for each outstanding transaction $\langle v, z \rangle \in$ F.history, in order of zxids, do
        Deliver $\langle v, z \rangle$
    end
    goto Phase 3
end
```

Update peers and commit

Update local history and store it

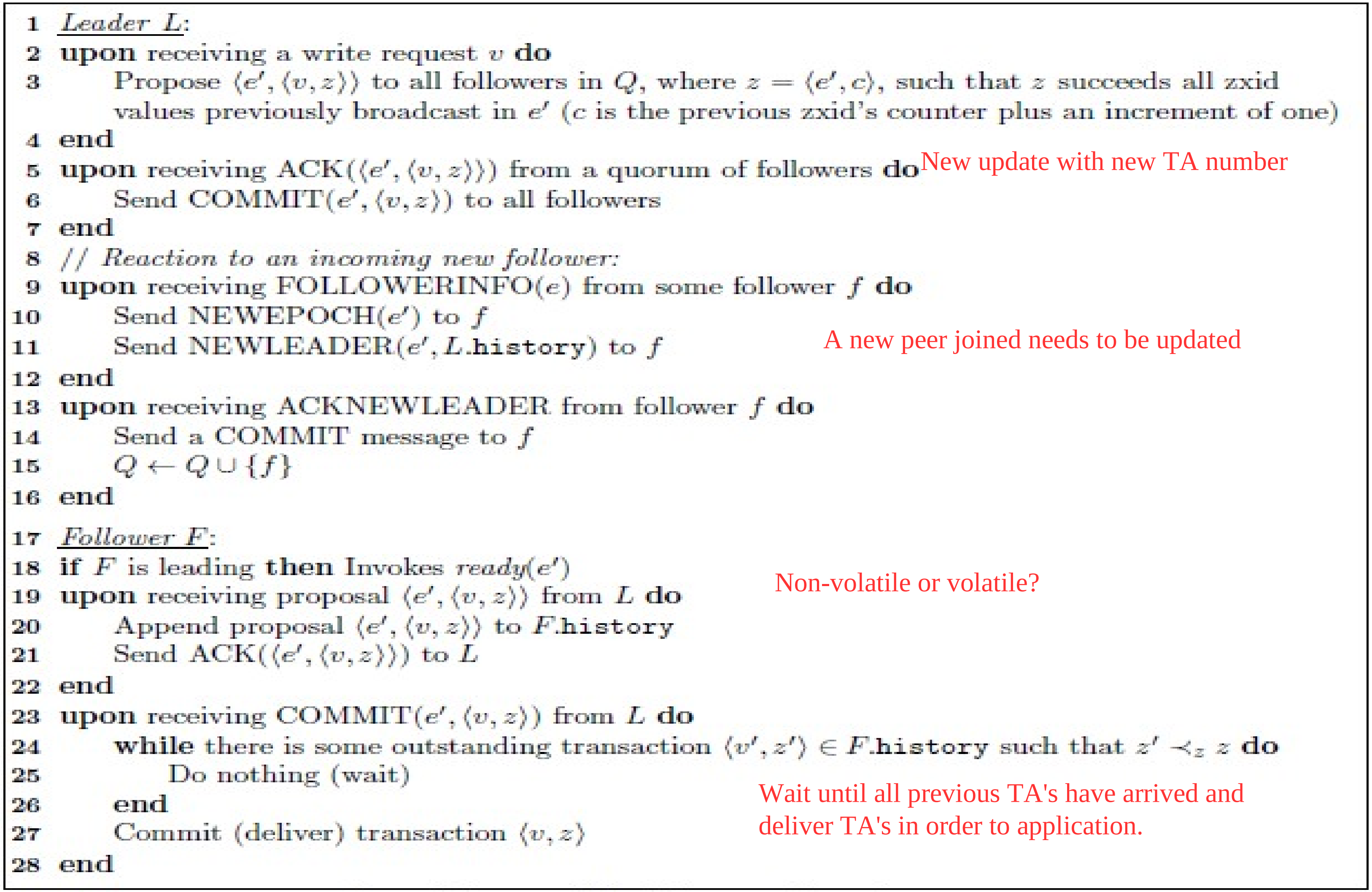
Never accept requests from an older (smaller) epoch!

Leader got quorum for updates. Peers deliver their history to application

Algorithm 2: Zab Phase 2: Synchronization.

After: ZooKeeper's atomic broadcast protocol:Theory and practice, Andre Medeiros. Leader suggests TA's to followers who miss some. Quorum acceptance establishes leader

Zab Protocol Phase 3: Broadcast



```
Leader L:
upon receiving a write request $v$ do
    Propose $\langle e', \langle v, z \rangle \rangle$ to all followers in $Q$, where $z = \langle e', c \rangle$, such that $z$ succeeds all zxid
    values previously broadcast in $e'$ ($c$ is the previous zxid's counter plus an increment of one)
end
upon receiving ACK($\langle e', \langle v, z \rangle \rangle$) from a quorum of followers do
    Send COMMIT($e', \langle v, z \rangle$) to all followers
end
// Reaction to an incoming new follower:
upon receiving FOLLOWERINFO($e$) from some follower $f$ do
    Send NEWEPOCH($e'$) to $f$
    Send NEWLEADER($e'$, $L$.history) to $f$
end
upon receiving ACKNEWLEADER from follower $f$ do
    Send a COMMIT message to $f$
    $Q \leftarrow Q \cup \{f\}$
end

Follower F:
if $F$ is leading then Invokes ready($e'$)
upon receiving proposal $\langle e', \langle v, z \rangle \rangle$ from $L$ do
    Append proposal $\langle e', \langle v, z \rangle \rangle$ to $F$.history
    Send ACK($\langle e', \langle v, z \rangle \rangle$) to $L$
end
upon receiving COMMIT($e', \langle v, z \rangle$) from $L$ do
    while there is some outstanding transaction $\langle v', z' \rangle \in F$.history such that $z' \prec_z z$ do
        Do nothing (wait)
    end
    Commit (deliver) transaction $\langle v, z \rangle$
end
```

New update with new TA number

A new peer joined needs to be updated

Non-volatile or volatile?

Wait until all previous TA's have arrived and deliver TA's in order to application.

Algorithm 3: Zab Phase 3: Broadcast.

After: ZooKeeper's atomic broadcast protocol:Theory and practice, Andre Medeiros. Broadcast layer is ready to perform new state changes under the new leader.

ABCast Implementations

- Implementing the theoretical invariants of such protocols is hard
- Non-volatile stores hurt performance and throughput
- Error detection is needed to recover from async. Protocol
- Frequent leader changes hurt throughput
- Consistency sometimes lowered for performance reasons
- Watch out for Byzantine errors like disk failures

One of the best reads about implementation: T. D. Chandra, R. Griesemer, and J. Redstone, “Paxos made live: An engineering perspective,” in PODC ’07: Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing. ACM, 2007, pp. 398–407. Learn how fault-tolerance can mask errors etc.

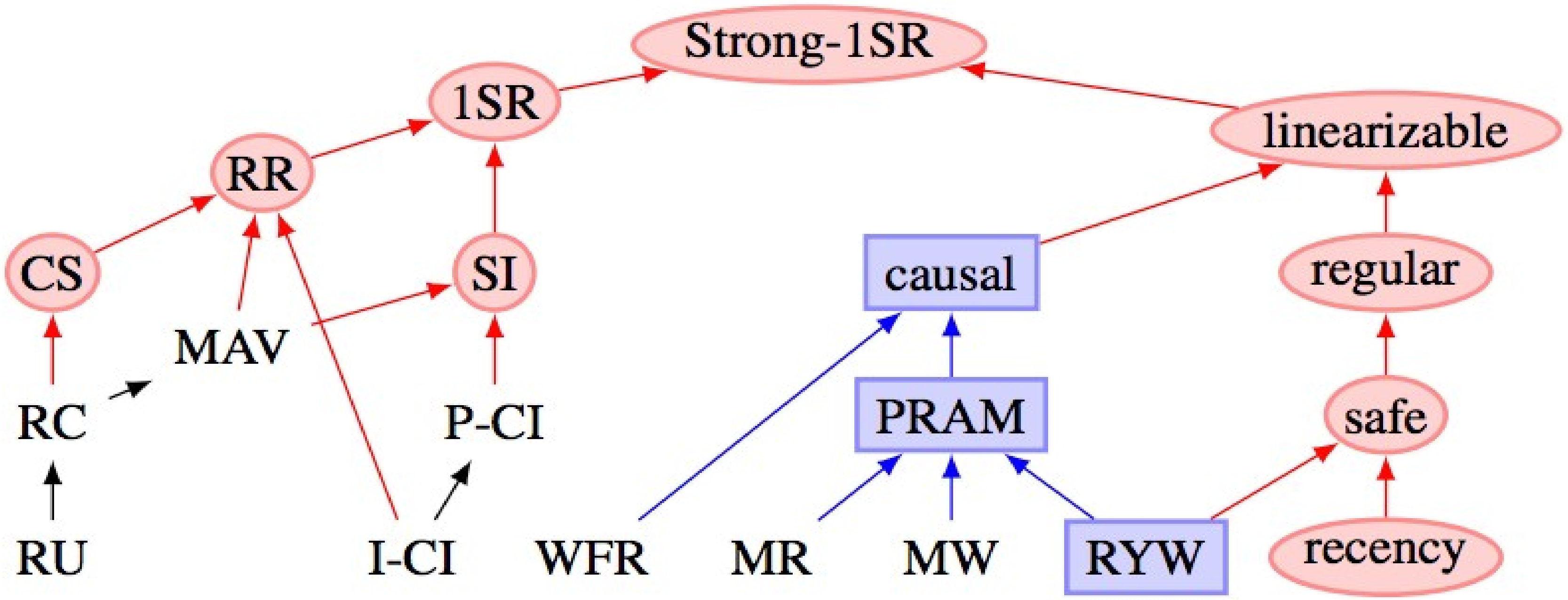
Highly-Available Transactions

HA Transactions

- Transactional guarantees that do not suffer unavailability during system partitions or incur high network latency. (Non-failing Replica MUST respond)
- Not CAP: linearizability as being able to read the most recent write from a replica
- Not: Serializability, Snapshot Isolation and Repeatable Read isolation are not HAT-compliant
- Read Committed isolation, transactional atomicity, etc. are possible with algorithms that rely on multi-versioning and limited client-side caching.
- causal consistency with phantom prevention and ANSI Repeatable Read need affinity with at least one server (sticky sessions)
- HA systems are fundamentally unable to prevent concurrent updates to shared data items and cannot provide recency guarantees for reads

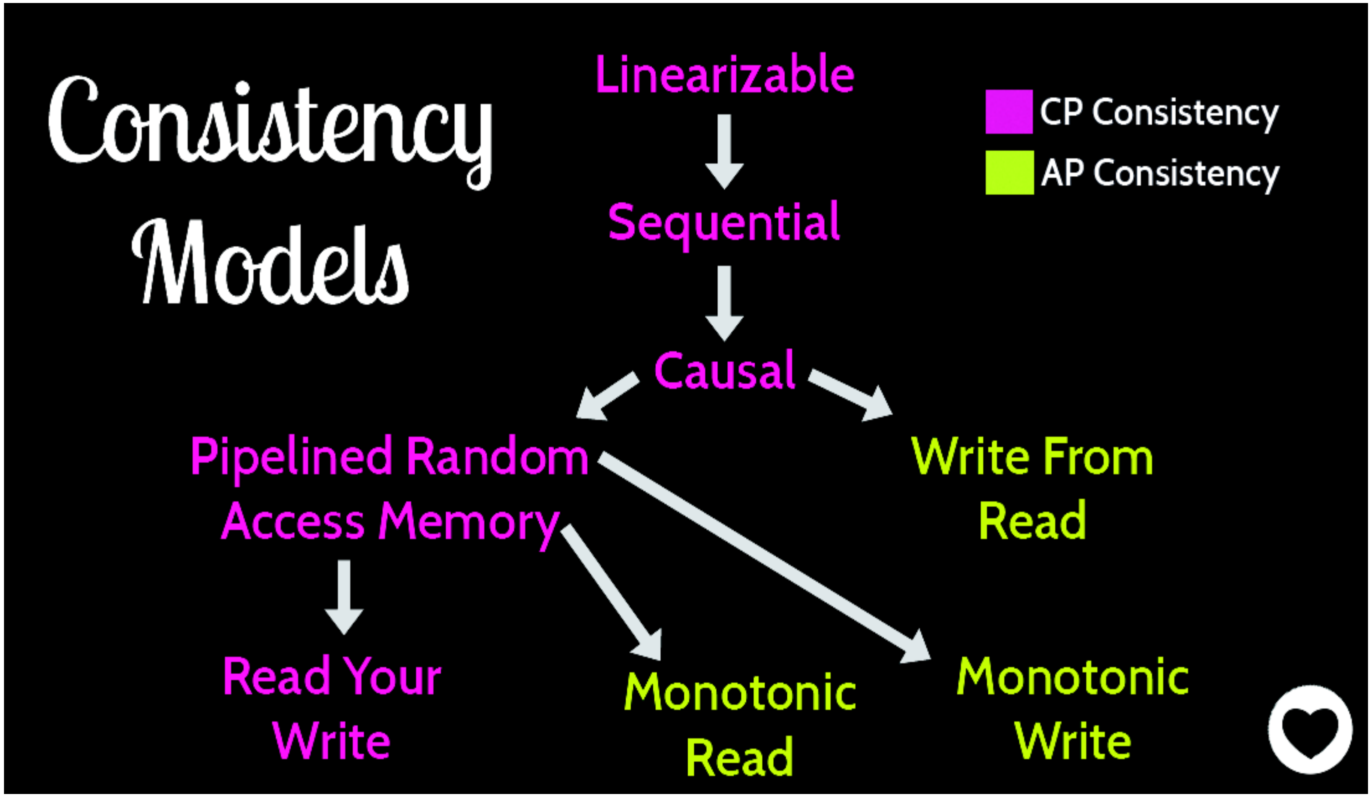
P.Bailies et.al., HA Transactions, Virtues and Limitations. HATs offer a one to three order of magnitude latency decrease compared to traditional distributed serializability protocols, and they can provide acceptable semantics for a wide range of programs, especially those with monotonic logic and commutative updates

HA-Transactions



Highly Available Transactions: Virtues and Limitations (Extended Version), Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, Ion Stoica, http://arxiv.org/pdf/1302.0309.pdf

Consistency Models



Consistency Models

🟪 CP Consistency
🟩 AP Consistency

Linearizable

Sequential

Causal

Pipelined Random Access Memory

Write From Read

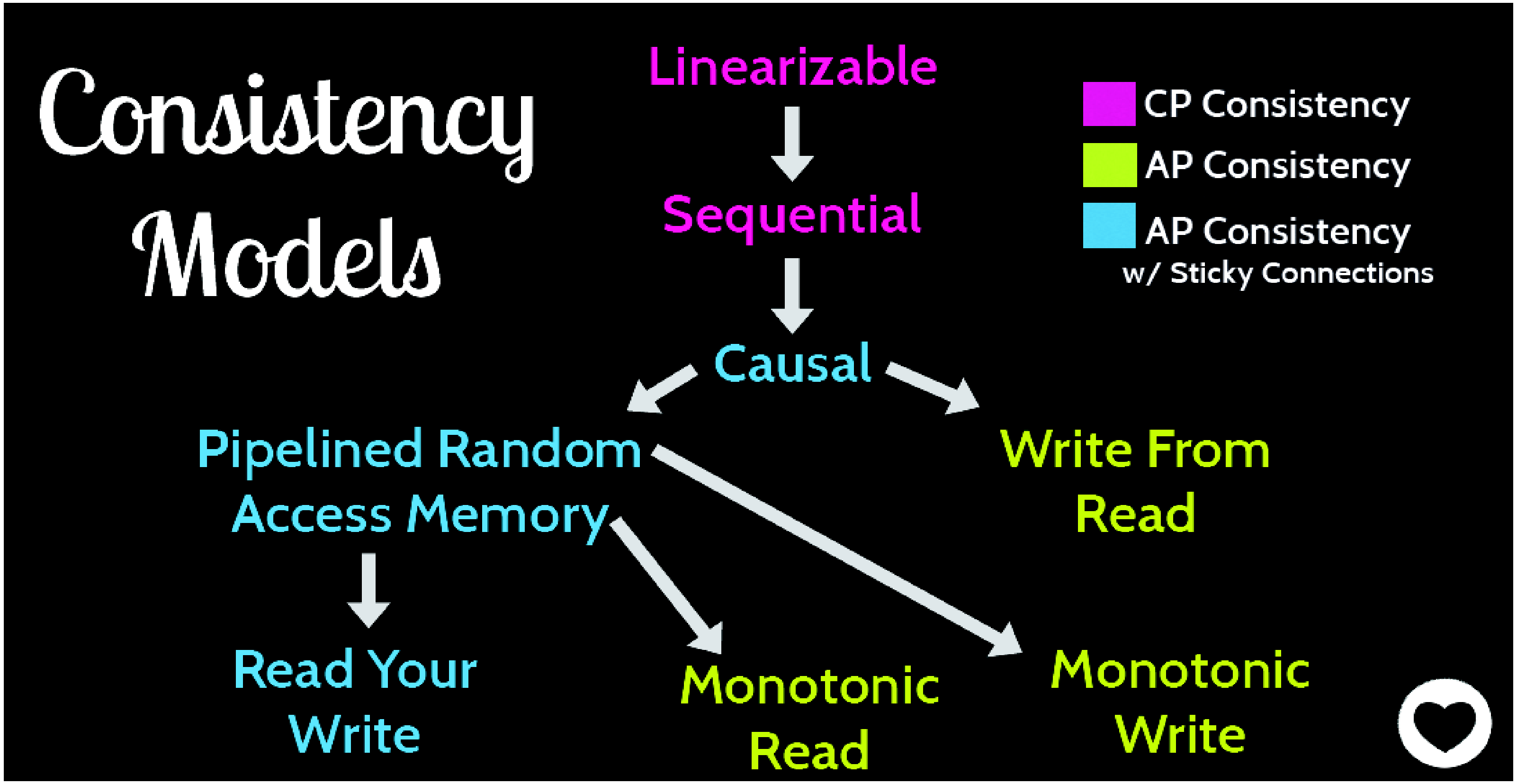
Read Your Write

Monotonic Read

Monotonic Write

From: Caitie McCaffrey, Building Scalable Stateful Services, Strangeloop 2015

Consistency with Sticky Sessions



Consistency Models

Linearizable

Sequential

Causal

Pipelined Random Access Memory

Write From Read

Read Your Write

Monotonic Read

Monotonic Write

🟪 CP Consistency

🟩 AP Consistency

🟦 AP Consistency w/ Sticky Connections

From: Caitie McCaffrey, Building Scalable Stateful Services, Strangeloop 2015

The World's Worst Distributed DB...

Uses approximately the same amount of electricity as could power an average American household for a day per transaction.
Supports 3 transactions / second across a global network with millions of CPUs/purpose-built ASICs.
Takes over 10 minutes to “commit” a transaction
Doesn’t acknowledge accepted writes: requires you read your writes, but at any given time you may be on a blockchain fork, meaning your write might not actually make it into the “winning” fork of the blockchain (and no, just making it into the mempool doesn’t count). In other words: “blockchain technology” cannot by definition tell you if a given write is ever accepted/committed except by reading it out of the blockchain itself (and even then)
Can only be used as a transaction ledger denominated in a single currency, or to store/timestamp a maximum of 80 bytes per transaction

But it is auditable and completely decentralized!

Toni Arcieri, On the dangers of a blockchain monoculture,
https://tonyarcieri.com/on-the-dangers-of-a-blockchain-monoculture
Maurice Herlihy, Blockchains From a Distributed Computing Perspective,
ommunications of the ACM, February 2019, Vol. 62 No. 2, Pages 78-85
10.1145/3209623

Resources (1)

Daniel Abadi, Problems with CAP and Yahoo's little known NoSQL system, http://dbmsmusings.blogspot.de/2010/04/problems-with-cap-and-yahoos-little.html

Java Data Objects Version 1.0 (www.java.sun.com)

Concurrency Control and Recovery in Database Systems, Philip A. Bernstein, Vassos Hadzilacos, Nathan Goodman

http://research.microsoft.com/en-us/people/philbe/ccontrol.aspx (free book)

Multi-Version-Concurrency-Control (MVCC), http://research.microsoft.com/en-us/people/philbe/chapter4.pdf

Davidson, Garcia-Molina, Skeen, Consistency in Partitioned Networks, http://www.cs.cornell.edu/courses/CS614/2004sp/papers/DGS85.pdf

Making Snapshot Isolation Serializable, Fekete, Liarokapis, O'Neil, O'Neil, Shasha, http://www.cse.iitb.ac.in/infolab/Data/Courses/CS632/2009/Papers/p492-fekete.pdf

Fekete, Goldrei, Asenjo, Quantifying Isolation Anomalies, http://www.vldb.org/pvldb/2/vldb09-185.pdf

Alvaro, Conway, Hellerstein, Marczak, Consistency Analysis in BLOOM: A CALM and Collected Approach, http://www.cidrdb.org/cidr2011/Papers/CIDR11_Paper35.pdf

Arjun Narajan, https://ristret.com/s/f643zk/history_transaction_histories (perfect intro to Tas and serialization)

Atul Adya, PhD Thesis, Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions

Resources (2)

- Colouris et.al., Chapters 12 an 13
- Ken Birman, Building secure and reliable network applications, Chapter 21 (Transactional Systems).
- Grey/Reuters, Transaction Processing (The bible of TA's)
- The Postgres manual (for isolation levels)
- Don Chamberlain, Universal Database (even though it's on DB2 and UDB he knows how to explain the database stuff perfectly – easy to read as well!)
- Meet the experts: Gang Chen on Transactions. Details of Websphere TA processing for J2EE architecture. With further links. http://www-128.ibm.com/developerworks/websphere/library/techarticles/0502_chen/0502_chen.html

Resources (3)

Java Communicating Sequential Processes. Middleware that implements Hoares CSP in Java. Excellent introduction by Abhijit Belapurkar on http://www.developers.net/node/view/849 (three parts with many links, e.g. on Pi-calculus for mobility, model checker for parallel process networks

Serializability Theory for replicated data, http://research.microsoft.com/en- us/people/philbe/chapter8.pdf

Analysis of Replication and Replication Algorithms in. Distributed System. Nikhil Chaturvedi. Prof. Dinesh Chandra Jain. http://www.ijarcsse.com/docs/papers/May2012/Volum2_issue5/V2I500414.pdf

Benjamin Reed, Zookeeper, the making of. https://developer.yahoo.com/blogs/hadoop/apache-zookeeper-making-417.html

Zookeeper Overview, Apache, https://zookeeper.apache.org/doc/trunk/zookeeperOver.pdf

Marco Serafini, Zab vs. Paxos (primary-backup vs. state-machine-replication) https://cwiki.apache.org/confluence/display/ZOOKEEPER/Zab+vs.+Paxos

Flavio P. Junqueira, Benjamin C. Reed, and Marco Serani. Zab: High-

performance broadcast for primary-backup systems. In DSN, pages 245{256. IEEE,

2011. ISBN 978-1-4244-9233-6 (crash-recovery model).

Call-me-maybe: MariaDB Galera Cluster, https://aphyr.com/posts/327-call-me-maybe-mariadb-galera-cluster (Kyle Kingsbury)

“Jepsenblog Series” by Kyle Kingsbury on Distributed Systems Correctness: aphyr.com/posts/jepsen

ZooKeeper's atomic broadcast protocol:Theory and practice, Andre Medeiros

T. D. Chandra, R. Griesemer, and J. Redstone, “Paxos made live: An engineering perspective,” in PODC ’07: Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing. ACM, 2007, pp. 398–407

G. V. Chockler, I. Keidar, and R. Vitenberg, “Group communication specifications: a comprehensive study,” ACM Comput. Surv., vol. 33, pp. 427–469, December 2001.

Tyler Treat, https://bravenewgeek.com/building-a-distributed-log-from-scratch-part-1-storage-mechanics/ (parts 1 to 5)

Resources (4)

Peter Bailis, When ist “ACID” ACID? Rarely! http://www.bailis.org/blog/when-is-acid-acid-rarely/

Peter Bailis, HAT, not CAP: Introducing Highly Available Transactions, Feb. 2013, http://www.bailis.org/blog/hat-not-cap-introducing-highly-available-transactions/

Peter Bailis et.al., Highly Available Transactions: Virtues and Limitations, (Extended Version)

Marc Shapiro, A comprehensive study of Convergent and Commutative Replicated Data Types, Shapiro et al., 2011

Pat Helland, Immutability changes everything! (an overview of techniques based on immutable data) http://www.cidrdb.org/cidr2015/Papers/CIDR15_Paper16.pdf

Adrian Colyer, Bolt on Causal Consistency, http://blog.acolyer.org/2015/09/01/bolt-on-causal-consistency/, morning paper on Bailis et.al, http://www.bailis.org/papers/bolton-sigmod2013.pdf

A.Colyer, ‘Cause I’m Strong Enough: Reasoning About Consistency Choices in Distributed Systems, February 3, 2016,

http://blog.acolyer.org/2016/02/03/the-rule/

Understandable RAFT visualization: http://thesecretlivesofdata.com/raft/