Data Replication & “NoSQL”

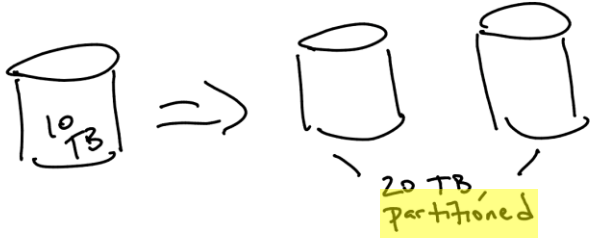
Berkeley
cs186

Outline

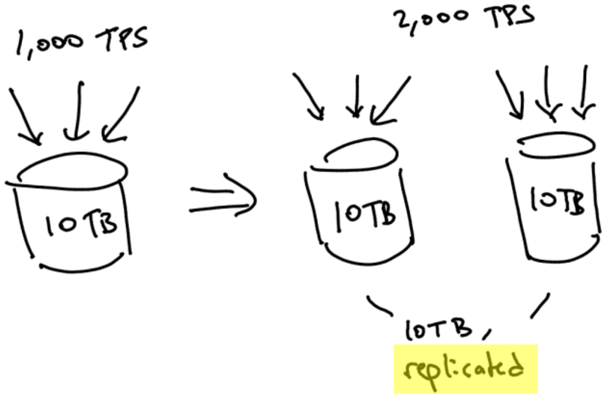
- Replicated Data
    - Transactional replication
- “Relaxing” transactions
    - Weak Isolation
    - Loose Consistency & NoSQL
        - Replica “consistency” & linearizability
        - Quorums
        - Eventual Consistency
- The Programmer’s View?

Why Replicate Data (1)? Scale.

- Does NOT help with data scaling.
  - That's what partitioning ("sharding") is for!



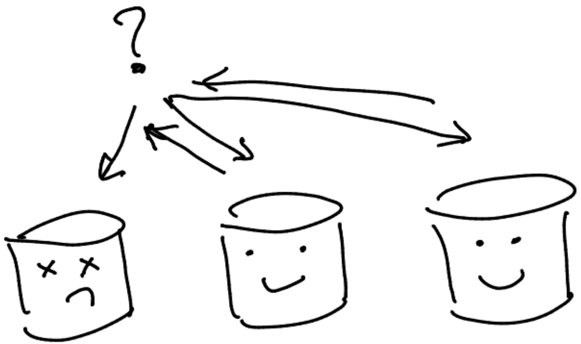
- DOES help with workload scaling!
  - Even on a small database
  - Load balancing



Why Replicate Data (2)? Availability

Replication increases availability

- If one replica can't answer, another can
- Tolerate one node's transient unavailability
  - Software crash, transient workload spike, JVM GC
- Survive catastrophic failures
  - Avoid correlation: place replicas on a different rack, different datacenter
- An "alternative" to logging
  - Actually, logging *is* a form of replication!
  - But full process replication recovers faster
    - Log-based recovery requires "interpreting" the log

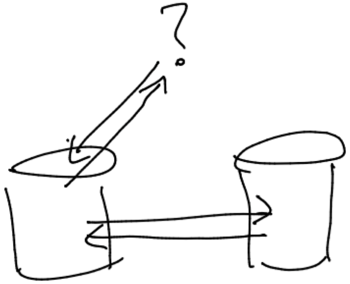


Why Replicate Data (3)? Locality

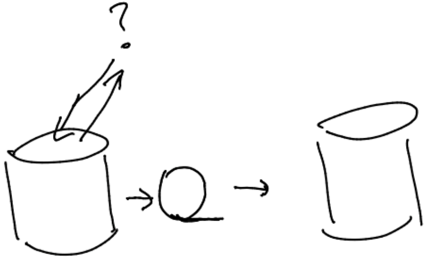
- Replication reduces latency
  - Choose a “nearby” server
    - Particularly for geo-distributed DBs
  - Ask many servers and take the 1st response

Traditional DB Replication Mechanisms 1: Single Master

- Single-Master replication
  - “Hot Standby”
  - Expensive: both sides handle full update volume
  - Expensive to get transactional guarantees
    - How do you ensure Durable commit?
    - Even single-site transactions require 2PC!

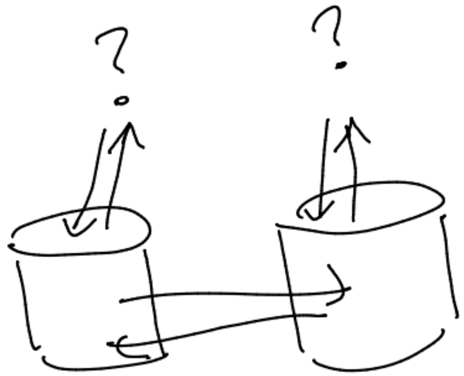


- Single-Master log-shipping
  - Cheap/free at the source node
  - Can be less expensive at the standby
  - Much lower bandwidth: diffs
  - Can be “hot” (via 2PC) or “warm” standby
    - Warm replica is a **valid transactional prefix**, but *stale*



Traditional DB Replication Mechanisms 2: Multi-Master

- Multi-Master
  - A.k.a. “Active-Active” replication
    - can do data-shipping or log-shipping
  - Writes happen anywhere
    - Low latency, more load balance
  - Can use 2PC to get copies to agree
    - But this causes high latency again



- What happens if we don’t do 2PC?
  - Writes may *conflict*
  - Need rules to resolve conflicting writes
    - More on this later!

Note: Replication Details

- Replication can be at various granularities:
  - DB, Table, Partition, Tuple

- Degree and choice of replication flexible
  - 3 is a good start
    - Odd, so under normal conditions, there is a “majority”
  - Choose locations in different “fault domains” for availability
    - Different racks, datacenters, continents
  - Replication factor could vary across objects
    - “hot objects” replicated to more nodes
    - improves read workload balancing
  - Some scheme needed to “route” queries to replicas
    - With load balance and fault tolerance
    - “Distributed Hash Tables”

Is 2PC really so expensive?

- How expensive is it, really, to hold a vote?
  - Raw latencies depend on your network
    - Geo-replication and speed of light
    - Vs. modern datacenter switches
  - But best-case behavior is not a good metric!

- Much depends on your machines' delay distribution
  - Latency to get responses from **all** participants?
    - The ***max*** latency is the worry, not the average
    - "Straggler" sensitive (so-called tail latency effects)
  - Hardware: NW switches, mag disk vs. Flash, etc.
  - Software: GC in the JVM, carefully-crafted event handlers...

Google does 2PC; it must be good?!

- Google Spanner uses 2PC and a host of other stuff
- But the latencies! Speed of light.
  - 7 times round the world per second

10 TPS!

operation	latency (ms)		count
	mean	std dev	
all reads	8.7	376.4	21.5B
single-site commit	72.3	112.8	31.2M
multi-site commit	103.0	52.2	32.1M

Outline

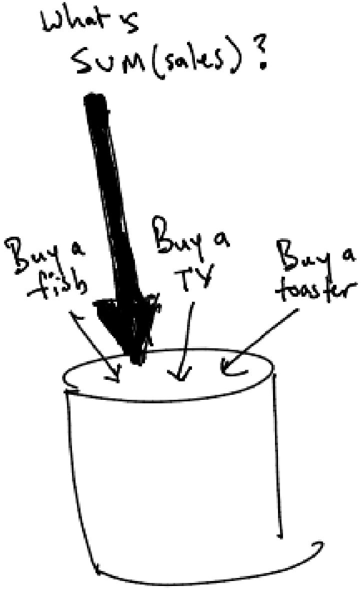
- Replicated Data
  - Transactional replication
- “Relaxing” transactions
  - Weak Isolation
  - Loose Consistency & NoSQL
    - Replica “consistency” & linearizability
    - Quorums
    - Eventual Consistency
- The Programmer’s View?

Weak Isolation

- I want transactions
- But serializability is not “concurrent enough”

- Can’t I have some “weaker” notion?
  - E.g. “short” read locks?
  - What might that “mean”

- Various (rather confusing) options commonly available in databases
  - See backup slides



Outline

- "Relaxing" transactions
    - Weak Isolation
    - Loose Consistency & NoSQL
        - Replica "consistency" & linearizability
        - Quorums
        - Eventual Consistency
- The Programmer's View

Hamilton Quote on Coordination

“The first principle of successful scalability is
to batter the consistency mechanisms down to a minimum,
move them off the critical path,
hide them in a rarely visited corner of the system, and then
make it as hard as possible for application developers
to get permission to use them”

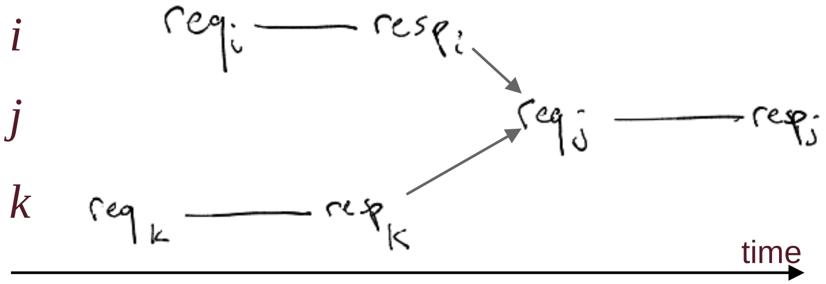
—James Hamilton (IBM, MS, Amazon)

A Definition: Replica Consistency

- Replica Consistency
  - Illusion: all copies of item X updated together, atomically
  - Really: readers *see* values of X that they would see without concurrency
    - **Linearizability** (a.k.a. “Atomic Consistency”)

- Notes
  - Not to be confused with the “C” in ACID!
    - This overloading has caused endless headaches
  - Not to be confused with serializability
    - Actions only; not transactions. Single-object writes
    - Deterministic ordering: writes on a given key are seen in the “real-time” order requested
      - Vs. “equivalent to *some* serial schedule”: non-deterministic
  - You can layer serializable concurrency control on top of linearizable replicated stores

Digging Deeper: Linearizability



- History:
  - Set of request/response pairs
- Linear History
  - A history with immediate responses to requests
- Linearizable Equivalence
  - History *h1* equiv to *h2* if:
    - Same set of requests/responses
    - **If response$_i$ precedes request$_j$ in h1, the same in h2**
- Linearizable History
  - Linearizably equivalent to Linear History
  - I.e. A fixed order of atomic request/response pairs

Quorum Consistency

- Assume writes are globally timestamped
  - <local-clock, node-ID> to break clock ties

- Want to ensure that all readers see the same (“consistent”) value
  - Even in the face of delayed replication
  - Again: this is *NOT* the C in ACID!! It’s linearizability.
- Idea 1: Write to All
  - Can read any 1 copy and get the “latest”

Quorum Consistency

- Assume writes are globally timestamped
  - <local-clock, node-ID>

- Want to ensure that all readers see the same (“consistent”) value
  - Even in the face of delayed replication
  - Again: this is *NOT* the C in ACID!! It’s linearizability.
- Idea 1: Write to All
  - Can read any 1 copy and get the “latest”
- Idea 2: Read from All
  - Can write “latest” to any copy

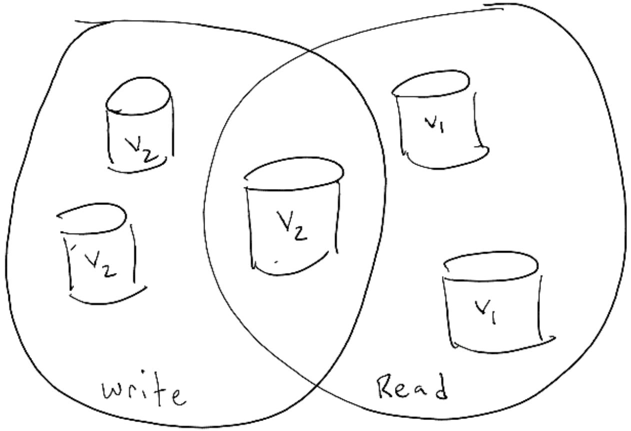
Quorum Consistency

- Assume writes are globally timestamped
  - <local-clock, node-ID>

- Want to ensure that all readers see the same (“consistent”) value
  - Even in the face of delayed replication
  - Again: this is *NOT* the C in ACID!! It’s linearizability.
- Idea 1: Write to All
  - Can read any 1 copy and get the “latest”
- Idea 2: Read from All
  - Can write “latest” to any copy
- Idea 3: Read & Write a majority
  - Read guaranteed to see “latest” on at least one node

Quorum Consistency

- More generally:
  - Write to a write-quorum of *w* nodes
  - Read from a read-quorum of *r* nodes
  - Ensure $r + w > N$ (#nodes)
    - Optional: ensure $w > N/2$ and you don't need write timestamps: no w/w race conditions
- Assumption (big!)
  - The set of nodes in the system (membership) is static
  - How do we handle failures? New nodes?
  - Take a distributed systems class!



Relaxing Replica Consistency

- With that background, here comes the Internet!

- And increasingly, the Cloud
  - Suddenly, we’re all using global-scale infrastructure. (??)

Update-heavy Global-Scale Services

- E.g. Amazon, Facebook, LinkedIn, Twitter
  - Shopping, Posting and Connecting

- Latency and Availability both paramount
  - Replication is critical
  - Favors **multi-master** solutions, with loose quora
  - Replica consistency becomes frustrating
    - See Hamilton quote

- Becomes quite natural to ditch replica Consistency!
  - Even linearizability is too expensive! (?)
    - All the more so serializability!!
  - The rise of NoSQL

NoSQL

Really two rejections:

1. **Say No! to schemas and declarative queries**
    - Not agile to have to define schemas in advance
    - SQL is a “hoop to jump through” from my “real” language
    - Instead, key-value stores
        - `put(key, val).  val = get(key)`
    - Sometimes the values are (JSON) documents
        - Notably MongoDB
2. **Say No! to serializable transactions**
    - Much easier to scale massively without them!
        - Replicate aggressively, with multi-master writes
        - No 2PC: Don’t worry about partial failure
    - Many use cases don’t need multi-object transactions
        - Because I’m just “sticking a stuff in the database”

NoSQL ambiguities

- Per-object ambiguities (no linearizability)
  - You may not read the latest version
  - Multi-master, Writers can conflict
    - Write the same key in different places
    - ‘Conflict Resolution’ or ‘Merge’ rules apply:
      - E.g. Last/First writer wins
        - But what clock do you use?
      - E.g. Semantic merge (e.g. Increment)
      - E.g. keep all values in a set (and let application decide)
- Across objects (no serializability)
  - Typically no guarantees
  - No notion of “trans”-actional semantics

“Eventual Consistency”

“if no new updates are made to the object, eventually all accesses will return the last updated value”

-- Werner Vogels, “Eventually Consistent”, CACM 2009

Which in practice means...??

Problems with E.C.

Two kinds of properties people discuss in distributed systems:

1. Safety: *nothing bad ever happens*
   - False! At any given time, the state of the DB may be “bad”
2. Liveness: *a good thing eventually happens*
   - Sort of! Only in a “quiescent” eventuality.
   - Arguably most EC systems service a collection of “sessions”, each of which quiesces in the real world

The State of NoSQL Today

1. No Schemas or Declarative Language
    - Schema flexibility adopted in relational DBMSs
    - Declarative SQL-like languages now common in NoSQL systems
        - E.g. Cassandra and MongoDB both have query languages
    - K/V and JSON support now common in RDBMSs
        - See PostgreSQL JSON support
    - RDBMS queries over schema-less files increasingly common as well
        - Redshift Spectrum, Postgres FDW, etc.

2. No Serializable Transactions
    - Still true in most NoSQL systems
    - Eventual Consistency models still evolving
        - More on this next

Outline

- Replicated Data
  - Transactional replication
- “Relaxing” transactions
  - Weak Isolation
  - Loose Consistency & NoSQL
    - Replica “consistency” & linearizability
    - Quorums
    - Eventual Consistency
- The Programmer’s View?

How do programmers deal with EC?

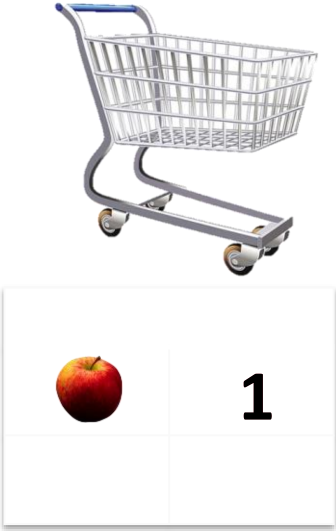
1. What, me worry?
2. Application-level coordination
   - The application needs to manage race conditions in its own (often parallel) setting
3. Provably consistent code: monotonicity
   - The application can be written in a language or framework where all messages are reorderable without affecting outcomes
   - We'll get to this shortly

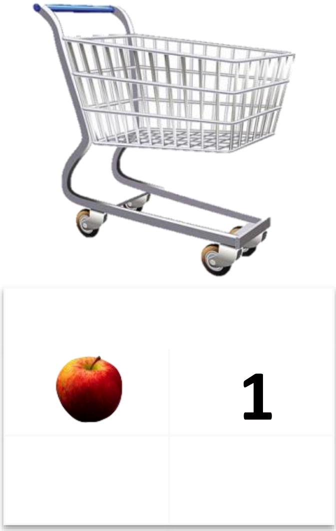
Case Study: The Shopping Cart

- Based on Amazon Dynamo
- With the global 2PC commit order “clock”


60
5
10
15
20
25
30
35
40
45
50
55
SHOCKRESISTANT


60
5
10
15
20
25
30
35
40
45
50
55
SHOCKRESISTANT


1


1


60
5
10
15
20
25
30
35
40
45
50
55
SHOCKRESISTANT

	1

	1

	1
	1

	1
	1

	1
	1

	1
	1


2
1
2
1
60
5
10
15
20
25
30
35
40
45
50
55
SHOCKRESISTANT

Eventually Consistent Version

- What could go wrong?

	01
	1

	21
	1

What, me worry?

- How often does inconsistency occur?
  - E.g. What are the odds of a “stale read”?
  - P.B.S. Study at Berkeley (pbs.cs.berkeley.edu)
    - Based on LinkedIn and Yammer traces
    - Stale reads are pretty rare
      - Esp in a single datacenter, using flash disks
    - But they happen!

- What’s a programmer to do?
  - In terms of understanding exceptions
    - What problems could they cause in the application layer?
  - In terms of exception detection/handling (“apologies”)

Application-level Reasoning

- The most interesting part of the Dynamo paper is its application-level smarts!
- Basic idea:
  - Don’t mutate values in a key-value store
  - Accumulate “action” logs at each node monotonically
    - Union up a set of actions (grows monotonically)
    - Union is commutative/associative!
  - Only coordinate for checkout


Proceed to
Checkout

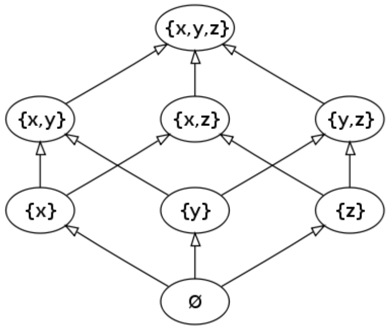

60
5
10
15
20
25
30
35
40
45
50
55
SHOCKRESISTANT

Monotonic Logic

- Merge things “upward”
  - sets grow bigger (merge: Union)
  - counters go up (merge: MAX)
  - booleans go from false to true (merge: OR)
- Also for the interesting merges:
  - Commutative
    - A merge B = B merge A
  - Associative
    - (A merge B) merge C = A merge (B merge C)
  - Idempotent
    - A merge A = A

General Monotonicity

- Merge things “upward”
  - sets grow bigger (merge: Union)
  - counters go up (merge: MAX)
  - booleans go from false to true (merge: OR)
- Can be partially ordered
  - E.g. sets growing
- Note why this works!
  - Associative, Commutative, Idempotent
- Core mathematical objects
  - Lattices & Monotonic logic
    - E.g. Select/project/join/union. But not set-difference, negation



CALM Theorem

- When can you have consistency without coordination?
  - Really.
  - I mean really. Like “complexity theory” really.
  - I.e. given application X, is there *any* way to code it up to run correctly without coordination?
- CALM: Consistency As Logical Monotonicity
  - Programs are eventually consistent (without coordination) iff they are expressible in monotonic logic.
    - Monotonicity => coordination can be avoided (somehow)
    - Non-monotonicity => coordination required

Example

- The fully monotone shopping cart

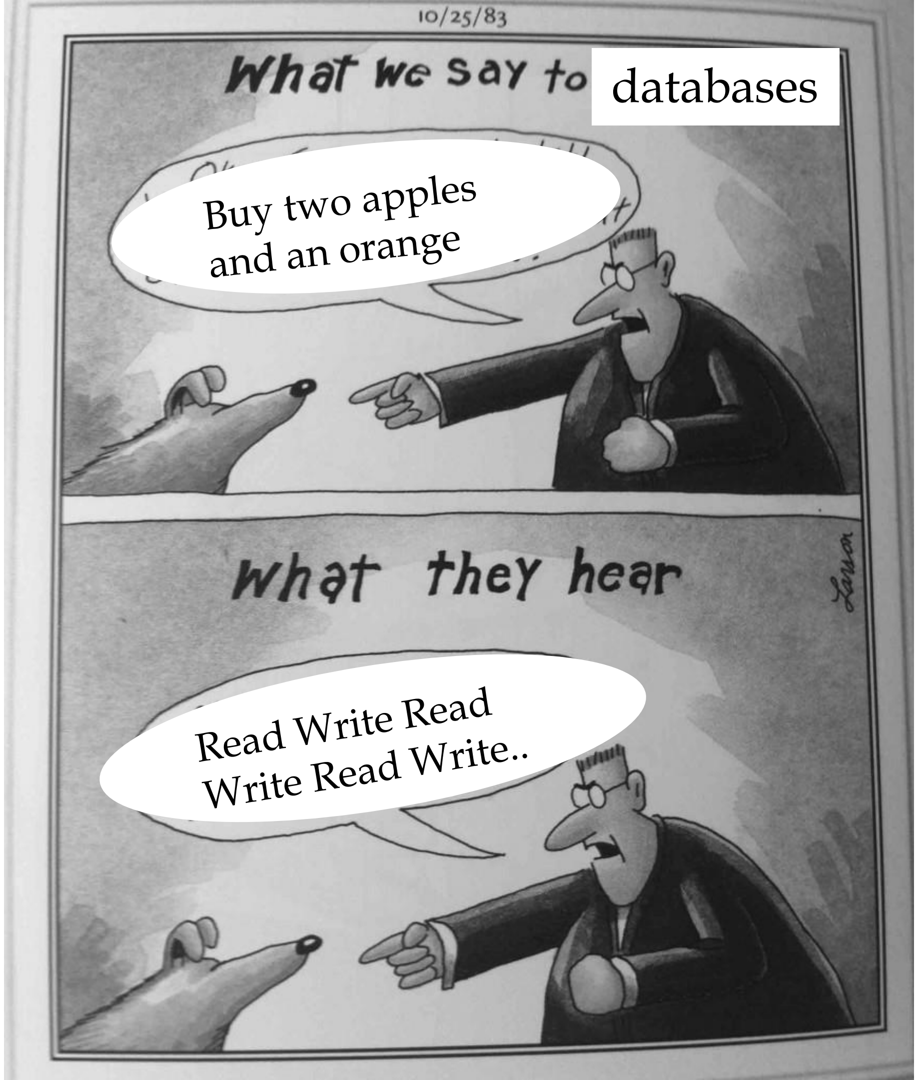
✔ ✔ A “seal” or “manifest”

The Burden on App Developers

- Given the tools you have
  - Java/Eclipse, C++/gdb, etc.
- Convince yourself your app is correct
  - No race conditions? Fault tolerant?
- Convince yourself your app will *remain* correct
  - Even after you are replaced
- Convince yourself the app plays well with others
  - Does your eventual consistency taint somebody else's serializable data?
- Wow. OK. Do you miss transactions yet?
  - Amazon reportedly replaced Dynamo
  - Transactions increasingly practical in a datacenter
  - But what about global, high-performance systems?

10/25/83


What we say to dogs
Okay, Ginger! I've had it! You stay out of the garbage! Understand, Ginger? Stay out of the garbage, or else!
What they hear
blah blah GINGER blah blah blah blah blah blah blah blah GINGER blah blah blah blah blah blah blah...
Larson



With thanks
to Peter Bailis

Shameless plug #1: <~ bloom

- If you know your application you can avoid coordination
- Really good programmers do this!
  - Maybe
- Everyone else needs a better programming model
  - One that encourages monotonicity, and analyzes for it.
  - E.g. `bloom` (http://bloom-lang.net)
    - + Confluence analysis (Blazes)
    - + Fault Tolerance analysis (Molly)
    - Etc.
  - See http://boom.cs.berkeley.edu

- We are still working on these ideas in my group

Popular NoSQL Systems

- MongoDB, Redis, Hbase, Cassandra, Couchbase, Voldemort, Riak, etc. etc.

- Why so many?
  - Not all that hard to build
  - Lack of standards, design alternatives

- Interesting to compare to Hadoop/Spark
  - Why is there no NoSQL core? Cause or effect?

Shameless Plug #2: Anna

- A system that scales across orders of magnitude
  - Because it’s fully shared-nothing, **coordination-free**
  - Even across threads!
  - No locks, no atomic instructions, no coordination protocols.
- On crazy performance benchmarks:
  - 800x faster than Intel’s “lock-free” main-memory hash table
  - 700x faster than the Masstree KVS from Harvard
- On more realistic Yahoo Benchmarks
  - 10x faster than Redis on a single node
  - 10x faster than Cassandra across the globe
- Many levels of consistency available
  - Uses lattice compositions to achieve many interesting consistency levels from research literature

Should you be using NoSQL?

- Data model and query support
  - Do you want/need the power of something like SQL?
    - And are your queries canned, or ad-hoc?
  - Do you want/need fixed or flexible schemas
    - Note that you can put flexible data into a SQL database
- Scale
  - Do you want/need massive scalability and high availability?
    - What’s your data volume? Update/query workload?
    - Are you committed to multi-master writes?
    - Will you need geo-replication
  - Are you willing to sacrifice replica consistency?
- Agility and growth
  - Are you building a service that could grow exponentially?
  - Optimizing for quick, simple coding?
    - Or maintainability?

Summing Up

- Partitioning provides Scale-Up
  - Can also partition lock tables and logs
- Replication provides workload scaling, availability
  - Can be done serializably, or linearizably
  - If not, new “consistency” challenges and compromises
- NoSQL is a playpen for exploring loosely consistent replication
  - Avoiding coordination scales up beautifully
  - Can you avoid coordination *and* get correct programs?

Backup material

Weak Isolation: Motivation

- Even on a single node,
  sometimes transactions seem too restrictive
  - The Chancellor requests the average GPA of all students
  - Various Profs want to make individual updates to grades
  - Can’t we all just get along?
- Sometimes transactions seem too expensive
  - E.g. 2PC requires computers to wait for each other

Must transactions be “all or nothing”?

Can’t we have “loose transactions” or “a little bit of” transactions

Short Answer (tl;dr):

- Yes, but the API for the programmer is hard to reason about.
- Still, many people adopt “don’t worry be happy” attitude

SQL Isolation Levels (Lock-based)

- Read Uncommitted
  - Idea: can read dirty data
  - Implementation: no locks on read
- Read Committed
  - Idea: only read committed items
  - Implementation: can unlock immediately after read
- Cursor Stability
  - Idea: ensure reads are consistent while app “thinks”
  - Implementation: unlock an object when moving to the next
- Repeatable Read
  - Idea: if you read an item twice in a transaction, you see the same committed version
  - Implementation: hold read locks until end of transaction
  - No phantom protection
- Serializable

Snapshot Isolation (SI)

1. All reads made in a transaction are from the same point in (transactional) time
   - Typically the time when the transaction starts
   - It’s like a “snapshot” from start time
2. Transaction aborts if its writes conflict with any writes since the snapshot.

When implemented on a MultiVersion system, this can run very efficiently! Oracle pioneered this. Postgres also implements it.

Fact 1: This is not equivalent to serializability.

Fact 2: Oracle calls this “serializable” mode.

SI Problem: Write Skew

- Checking (*C*) and Savings (*S* accounts)
- Constraint: $C_i + S_i >= 0$
- Begin: $C_i = S_i = 100$
  - T1: withdraw \$200 from C_i
  - T2: withdraw \$200 from S_i
- Serial schedules:
  - T1; T2. Outcome: ??
  - T2; T1. Outcome: ??
- SI schedule:

  ?? !!

Bad News

- The lock-based implementations don’t exactly match the SQL standards
  - They do uphold the ANSI standards
  - But the official SQL definitions are (unintentionally) somewhat more general

- Upshot:
  - It’s *very* hard to reason about the *meaning* of weak isolation
  - Usually people resort to thinking about the *implementation*
  - This provides little help for the app developer!

Worse News!

Database	Default	Maximum
Actian Ingres 10.0/10S	S	S
Aerospike	RC	RC
Akiban Persistit	SI	SI
Clustrix CLX 4100	RR	RR
Greenplum 4.1	RC	S
IBM DB2 10 for z/OS	CS	S
IBM Informix 11.50	Depends	S
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

RC: read committed, RR: repeatable read, SI: snapshot isolation, S: serializability, CS: cursor stability, CR: consistent read

Table 2: Default and maximum isolation levels for ACID and NewSQL databases as of January 2013 (from [9]).

Worse News!

Database	Default	Maximum
Actian Ingres 10.0/10S	S	S
Aerospike	RC	RC
Akiban Persistit	SI	SI
Clustrix CLX 4100	RR	RR
Greenplum 4.1	RC	S
IBM DB2 10 for z/OS	CS	S
IBM Informix 11.50	Depends	S
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

RC: read committed, RR: repeatable read, SI: snapshot isolation, S: serializability, CS: cursor stability, CR: consistent read

Table 2: Default and maximum isolation levels for ACID and NewSQL databases as of January 2013 (from [9]).

With thanks
to Peter Bailis