Lecture 19 Replication, Consistency, NoSQL, and Dynamo
5/3/2021

Based on notes from Peter Bailis and Aaron Elmore

Replication refresher: we have many choices with replication protocols including primary copy vs multi-master and async vs sync.

Replication is often used to address fault-tolerance and performance.

How does it address performance?
Fast Reads and Locality.

What about writes? Less clear - slower, and maybe less available if have to write all nodes?

What do we mean by availability? (Can the system process requests)
Why do are we worried about this in large systems?
==> Fact of life in large scale systems: **nodes will fail**.

Even with long mean time to failure (MTTF) with enough nodes we we can expect regular failures.

Replication clearly helps with read availability, but how about write-availability?

Tradeoff: if we write to all replicas, availability is worse; if we right to only some replicas, availability maybe better, but may see stale data
Consistency -- or how we reason about replicated state. Many notions of consistency:

Eventual Consistency	Strong Consistency
(replicas will eventually converge if updates/reads stop)	Act as though not replciated 1-copy serilizability

Many models of consistency (admissibility criteria).
Examples:
- read your writes
- monotonicity (always see a view of data moving forward in time)

No free lunch: decision between availability and consistency.

CAP Theorem
Eric Brewer at PODC 02 stated system can have 2 of 3 properties
-Consistency
-Availability
-Partition Tolerance

(CAP) proof on systems with async communication
With DBs that scale we will need to tolerate partitions

Why is the decision between A & C?

Consider
3 nodes n1, n2, and n3 storing value of x as 4.

```
n1      n2      |   n3
x=4     x=4     |  x=4
                |
```

Update x=5 arrives at n1 while n3 is partitioned.
A read arrives at n3... What do we do?

Options:
- Wait for n3 to recover
- Forge ahead -- n1 and n2 process write, somehow make n3 aware in the future?

If we forge ahead, can we ensure that we don't ever read old state of n3?

Partitioning data makes updates hard.

Enter NoSQL

Web scale companies had issue scaling databases, especially in the face of partitioning. Homebrew new DBMSs to address scale-out issues, often favoring availability over transactional consistency.

Common Attributes:
- Partition data on key
- Single-key atomicity
- Drop expensive ops (txns, joins, secondary indexes)
- Avoid single points of failure

Dynamo (open source projects Voldemort, Cassandra [sort of])

Amazon wanted always available DB (i.e. add to shopping cart should never fail)
A page render can use up to 150 services, so stringent SLA requirements.

Partitions or temporary failures happen.

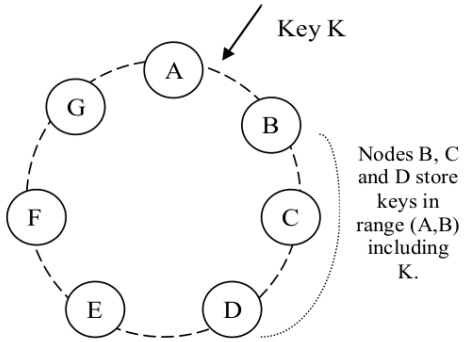
Simple query language/data model:
- key:value assume both are byte array, md5 on key to generate ID - get(key)
- put(key,context,value)
- get(key)

- single-key atomicity
Other key design principles:
- Incremental scalability (add nodes)
- Symmetry/ decentralizations (each node does same thing, no central control)
- Heterogeneity in nodes (servers will change)

Challenges faced:
- partitioning
- highly available for writes
- handle temporary failures recovery separately from permanent failures membership / failure detection

Partitioning
Consistent hashing: h(k) -> 1...2^128, mapped on to a ring (wraps for larger values). Each node randomly assigned a point on the ring.

One hop routing: The node(s) as we move clockwise from a point K along ring are responsible for a key with h(k) = K



Dynamo uses a variation with *virtual nodes*, where each node is responsible for multiple points on the ring. Has the following benefits:

- Node unavailability distributes load requirements
- New nodes or node re-availability accepts equivalent amount of from each of other available noes
- Allows for heterogeneity in nodes with different virtual node allocations

<u>How do nodes join the ring?</u>

Administrator explicitly adds or removes nodes. Nodes learn about other nodes (and part of space they are responsible for) by contacting on of a small number of *seed nodes,* which are statically configured. Nodes then *gossip (randomly talk with other peers)* with each other to exchange membership and partitioning information.

When a node is added it chooses a set of random tokens (hash values); mapping is persisted and reconciled with other nodes. This creates the mapping of keys to nodes,

which allows one hop lookup for reads/writes.
When a new node joins, it takes ownership of keys in the range of data it is responsible for. Data is reallocated on joins.

Replication

N replicas — each write goes to next N successors on the ring, it's "preference list"

Reads and writes

Client contacts coordinator (one of the N top nodes, not always first to load balance)

Reads and writes are driven by N, R, W
R+W > N for ensures that if there is no partitioning any subsequent read will see a write.
This is called a *quorum write*.

Example:

N = 3, R = 1, W = 1

key x

A	B	C
v1	v1	v1
v2	v1	v1

Read (x) --> B, gets 1

N = 3, R = 2, W = 2

A	B	C
v1	v1	v1
v2	v2	v1

Any read of 2 replicas will see at least one copy of version 2.

Sloppy Quorum (healthy N)

Dynamo authors don't think quorums are sufficient, for 2 reasons:
- Decreased durability (want to write all data at least 3 times)
- Decreased availability in the case of partitioning.

Example
N = 4, W = 3, R = 2 Can't write if A & B are partitioned from C & D

E	A	B	C	D
	v1	v1	v1	v1

In sloppy quorum, writes & reads just keeps going around the ring until it has written to / read from N nodes.

If a node not in the top N gets a write request for a key, the write will include a "hint" that has the original target for the node.

In example E gets request intended for D if D is unavailable. E will check to see if it can reach D periodically, and if so, will send the update to it (or reconcile). "Hinted handoff"

What's the problem with sloppy quorum? Can get divergent versions -- i.e., doesn't actually ensure we will always read the most recent version, even with R + W > N

Example
N = 4, W = 3, R = 2

E	A	B	C	D	F
	v1	v1	v1	v1	
v2	v2	v2	v2'	v2'	v2'

A reader may contact C & B, see two different writes!

Hinted handoff works best with low churn and transient failure — with long permanent failures will lead to lots of inconsistent reads.

How to detect conflicts? Vector clocks to capture *causality* between versions. A vector clock is just a list of (node,counter) pair attached to each data item. Also called the write *context*.

Example -- data item x
N = 4, W = 3, R = 2

E	A	B	C	D	F	Can't write if A & B are partitioned from C & D
	1	1	1	1		
	[A,1]	[A,1]	[A,1]	[A,1]		==>[A,1] is the vector clock attached to this data item because A was the last coordinator for the write
	2	2	2	1		
[A,2]....				[A,1]		==> At this point, [A,1] << [A,2], so these are causally related (D saw a subset of everyone else's writes
3	3	3	3'	3'	3'	
[A,3]	...		[A,2],[C,1]			==> [A,3] is incomparable to [A,2],[C,1] Nodes on two sides of partition saw different writes

At this point, a read sent to B and C will get both versions, can tell they are incomparable, so must reconcile via r*ead repair*. Can either use *latest writer*, or do something application specific (.e.g, for shopping cart union items in cart — won't lose

items, but deleted items can re-appear)

After reconciliation, write back. (Also do a write back if did a read and some node had an older version that was comparable.)

Partition heals -- B write back

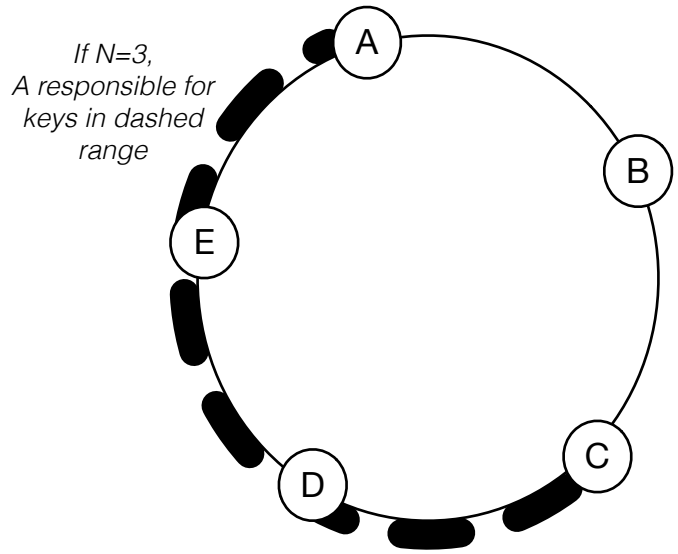
4	4	4	4	4	4	
[A,3]	[A,3],[C,1],[B,1]		....		[A,2],[C,1]	==> A,B,C,D have new value Note that E & F don't get written to so still have some old version!

Although write back on read can help synchronize versions, want even more *anti-entropy* measures to ensure replicas stay in sync.

Could just compare the vector clocks of all data items between all nodes as a part of gossip, but that'd be a lot of data transfer. Instead, use a cute trick called a Merkle tree.

Idea is for each key range a node is responsible for it computes a hash tree, and compares that with other nodes also responsible for key range.

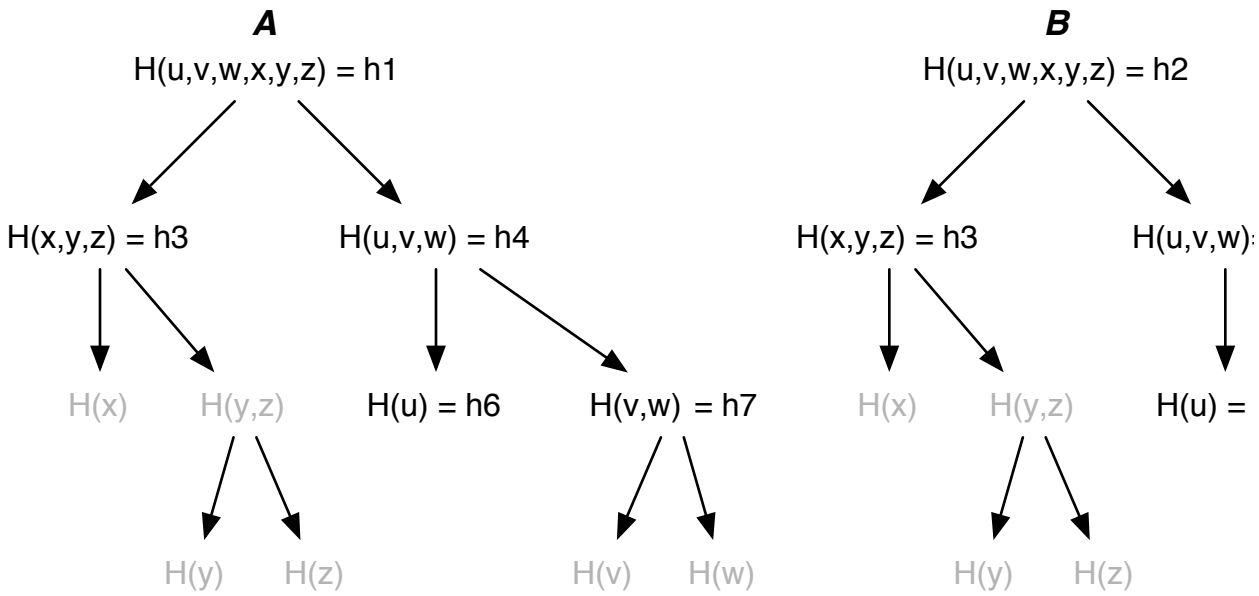
Which key ranges is a node responsible for?



Ex for EA range, B and C are also responsible

What is a Merkle tree?

Suppose EA range has keys u,v,w,x,y,z



This whole tree is as big as data, but only need to exchange parts of it that are different, i.e., no need to send light gray nodes in diagram, since parent hashes are all equal

Summary:

Problem	Technique	Advantage
partitioning	consistent hashing	incremental scalability
highly available for writes	vector clocks with read repair	version size decoupled from update rate
handle temporary failures	sloppy quorum and hinted handoff	HA with some durability
recovery from permanent failures	anti-entropy with merkle trees	sync replicas
membership / failure detection	gossip based membership	symmetry and no centralized repo

Thoughts on this design?