Eventual Consistency & Amazon Dynamo

“Dynamo Machine”, Natalia Sergeevna Goncharova, 1913

“electric dynamo in the style of early 20th century art”, Stable Diffusion, November 14, 2022
2PC Recap

• Remember this?

• If Coord + 1 Worker fail, no way to recover
  • Coord may have told failed Worker about outcome, it may have exposed results
Failure Cases

Coordinator

1. PREPARE(T)

2. VOTE(T, YES/NO)

3. FW(COMMIT/ABORT)

Worker

4. FW(COMMIT/ABORT)

5. COMMIT/ABORT(T)

6. FW(COMMIT/ABORT)

7. ACK

8. W(DONE), once all W's ACK
Failure Cases

What happens if Coordinator transmits COMMIT to some workers, then dies forever?

Some workers think COMMIT took place, others can never obtain an outcome.
Amazon Operational DB Desiderata

- “Always Available” shopping cart
  - Should not go down even if a datacenter fails
  - No centralized point of failure
- Very low latency
  - Lots of orders being processed
  - Many lookups required to render a page
- No need for complex analytics
- Incrementally scalable
Enter Dynamo

- “Always Available” shopping cart
  - Data replicated across multiple nodes
  - Favor availability over consistency
- Very low latency
- No need for complex analytics
- Incrementally scalable
  - Key value store
  - CRUD semantics
  - Keys partitioned across workers using consistent hashing
Versus RDBMS

• “Always Available” shopping cart
  - Data replicated across multiple nodes
  - Favor availability over consistency
• Very low latency
• No need for complex analytics
• Incrementally scalable
  - Key value store
  - CRUD semantics
  - Keys partitioned across workers using consistent hashing

  Favor consistency above all else
  Complex SQL queries can be slow
  Can add new nodes in shared nothing but shuffle joins may not scale incrementally
Replication Primer

• Replicating data helps with fault tolerance and performance
• Reads:
  • On a fault, reads can be directed to replica
  • Also, reads can be handled by local replica

• What about writes?
  • Slower? (More nodes to write)
  • Less available? (Have to write all nodes, what if some nodes crash?)
Availability

• Availability: can the system process requests?
• In large systems, even w/ very reliable nodes, failures happen!
• Replication clearly provides read availability
• What about writes?
Write Availability Tradeoff

• If we write to all replicas, availability is worse!
• If we only write some replicas, availability is better, but replicas can be stale
• Availability and consistency are a spectrum:

• Many models of consistency
No Free Lunch

• Pick one of availability or consistency
• CAP Theorem
  • Eric Brewer at PODC 02; system can have 2 of 3 properties
    Consistency
    Availability
    Partition Tolerance

• CAP proof on systems with async communication
Options:
1. Wait for partition to heal (Consistent)
2. Forge ahead: n1 and n2 process write, somehow make n3 aware later? (Available)

If data is partitioned must choose either consistent or available!
NoSQL

Class of systems like Dynamo that generally offer:

- Key/value storage (not SQL!)
- Partitioned and replicated by key
- Favoring availability over consistency
Early 2010’s saw MANY such systems, with slightly different data models and semantics.

Source: https://www.slideshare.net/danglbl/schemaless-databases/7
Dynamo Query Interface

- Key / Value store
- All keys and values are arbitrary byte arrays
  - md5 on key to generate ID
- get(key)
- put(key, context, value)
  - Context is a sequence number done by coordinator of write
  - More later
- single-key atomicity
  - i.e., each read/write is atomic, but only with respect to key
Dynamo Data Partitioning and Replication

- All data replicated on N nodes
- Each node has an address on a “ring” representing space of hash values from say, $0 \rightarrow 2^{128}$
- Data stored on ring as well

“Overlay network”: Nodes are not actually in a physical ring, but are just machines on the Internet

Each node occupies one (or multiple) random locations on ring

A key hashed to location $k$ is stored on N successors in ring
Consistent Hashing

• Data and nodes mapped to ring
• Data assigned to nearest successor(s)
• When a node joins, it takes over only keys in range it joins
• No need to rehash all values!
Joining the Ring

- Administrators explicitly add / remove nodes
- When a node joins, it contacts a list of “seed nodes”
  - Other nodes periodically “gossip” to learn about ring structure
- When a node i learns about new node j, i sends j any keys j is responsible for

Seed nodes are nodes clients and other nodes can ask for current mapping

External discovery service
Seed node list

Each node has mapping of all other nodes. This is small, even for thousands of nodes

<table>
<thead>
<tr>
<th>Node</th>
<th>Loc</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>X</td>
</tr>
<tr>
<td>C</td>
<td>Y</td>
</tr>
<tr>
<td>F</td>
<td>W</td>
</tr>
</tbody>
</table>

B Table

Keys in range [X,W]

<table>
<thead>
<tr>
<th>Node</th>
<th>Loc</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>X</td>
</tr>
<tr>
<td>C</td>
<td>Y</td>
</tr>
<tr>
<td>F</td>
<td>W</td>
</tr>
</tbody>
</table>

C Table
Handling Reads

• Each item is replicated on N nodes
• To read: hash key, send request to one replica
  • Client either uses Amazon front end or reads mapping table from seeds

Client selects one node to be *coordinator*
Handling Writes

- Route as in reads
- Back to our availability conundrum
  - Do we write all replicas? What if one has failed / isn’t available?
  - Do we write just one replica? How do we ensure that our read will be visible to other nodes?

Client selects one node to be coordinator
Dynamo Consistency

• “Quorum Writes”
• R+W > N
  • N = number of replicas of each data item
  • R = number of replicas each read must be heard from
  • W = number of replicas each write must be sent to
• E.g., R = 2, W = 2, N = 3
  
<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>v1</td>
<td>v1</td>
<td>v1</td>
<td></td>
</tr>
<tr>
<td>v2</td>
<td>v1</td>
<td>v2</td>
<td></td>
</tr>
</tbody>
</table>

  ➔ write to 2 out of 3

Any read of 2 will see v2!
Dynamo Consistency

• “Quorum Writes”
• \( R+W > N \)
  • \( N \) = number of replicas of each data item
  • \( R \) = number of replicas each read must be heard from
  • \( W \) = number of replicas each write must be sent to

• Need some way to ensure that if fewer than \( N \) nodes written to, write eventually propagates
  • If a reader sees that a replica has a stale version, it writes back
Sloppy Quorum

• Quorums still favor consistency too heavily, because:
  • Decreased durability (want to write all data at least N times)
  • Decreased availability in the case of partitioning.

• Solution: Sloppy Quorum
Sloppy Quorum & Hinted Handoff

- If fewer than $N$ writes succeed, continue around ring, past successors

**Diagram:**
- $N=3$
- $K=x$
- Hint: Owner=$E$
- "Hinted Handoff": $B$ will keep trying to read $E$ to let it know about the value of $K$

2 out of 3 writes succeed
Continue around ring, write to $B$
Sloppy Quorum ➔ Divergence

• If network is partitioned, hinted handoff can lead to divergent replicas
• E.g., suppose N=3, W=2, R=2, Partitioned

Diagram:
- Nodes A, B, C, D, E
- Client 1
- Dotted line partitioning
- Nodes A, B, C on one side, D, E on the other
- Node A marked as (sloppy)
Sloppy Quorum ➔ Divergence

- If network is partitioned, hinted handoff can lead to divergent replicas
- E.g., suppose N=3, W=2, R=2, Partitioned

Two different versions of key k, k1 and k2 now exist
Vector Clocks

- Each node keeps a monotonic version counter that increments for every write it coordinates.
- Each data item has a clock, consisting of a list of the most recent version it includes from each coordinator.
Vector Clocks

- Each node keeps a monotonic version counter that increments for every write it coordinates.
- Each data item has a *clock*, consisting of a list of the most recent version it includes from each coordinator.

Client 1
Create k → C
C writes [C,1] to C, D, E

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>1 [C,1]</td>
<td>1 [C,1]</td>
<td>1 [C,1]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>[C,1]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

[C,1]: Contains first version from C as coordinator.
Vector Clocks

- Each node keeps a monotonic version counter that increments for every write it coordinates.
- Each data item has a clock, consisting of a list of the most recent version it includes from each coordinator.

<table>
<thead>
<tr>
<th>Client 1</th>
<th>Create k → C</th>
</tr>
</thead>
<tbody>
<tr>
<td>C writes [C,1] to C, D, E</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1 [C,1]</td>
<td>1 [C,1]</td>
<td>1 [C,1]</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Client 1</th>
<th>Read k → C</th>
</tr>
</thead>
<tbody>
<tr>
<td>C reads C, D, E</td>
<td></td>
</tr>
<tr>
<td>C returns [C,1]</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Write k [C, 1] → C</th>
</tr>
</thead>
<tbody>
<tr>
<td>C writes [C,2] → C, A, B</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Vector Clocks

- Each node keeps a monotonic version counter that increments for every write it coordinates.
- Each data item has a clock, consisting of a list of the most recent version it includes from each coordinator.

**Client 2**
Read k → D
D reads D,E,F
D returns [C,1]
Write k [C,1] → D
D writes [C,1][D,1] to D, E, F

**Table**

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>[C,1]</td>
<td>[C,1]</td>
<td>[C,1]</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>[C,2]</td>
<td>[C,2]</td>
<td>[C,2]</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>[C,1][D,1]</td>
<td>[C,1][D,1]</td>
<td>[C,1][D,1]</td>
<td>[C,1][D,1]</td>
</tr>
</tbody>
</table>

Incomparable (can’t totally order)
Read Repair

• Possible for a client to read 2 incomparable versions

• Need *reconciliation*; options:
  • Latest writer wins
  • Application specific reconciliation (e.g., shopping cart union)

• After reconciliation, perform *write back*, so replicas know about new state
Study Break

Suppose there are three replicas, $R_1$, $R_2$, and $R_3$. Three writes are performed to key $K$, resulting in three version clocks:

$$V_1 = \langle R_1:0, R_2:3, R_3:2 \rangle$$
$$V_2 = \langle R_1:1, R_2:3, R_3:2 \rangle$$
$$V_3 = \langle R_1:0, R_2:0, R_3:3 \rangle$$

Which of the following are true statements?
Study Break

V1 =<R1:0,R2:3,R3:2>  
V2 =<R1:1,R2:3,R3:2>  
V3 =<R1:0,R2:0,R3:3>  

1. The writer that produced V1 observed V2. ✗
2. The writer that produced V2 observed V1. ✓
3. The writer that produced V3 observed V1. ✗

V2 was coordinated by R1, saw same versions as V1
V3 was coordinated by R3, did not see R2 1, 2, or 3, and happened concurrently with V2
Anti-entropy

• Once a partition heals, or a node recovers, need a way to patch up
• Could rely on gossip & hinted handoff
• Dynamo also compares nodes responsible for each key range
  • Comparison done via hashing, using a technique called Merkle trees

If $N=3$, A responsible for keys in dashed range

Here, for EA range, B and C are also responsible
Merkle Trees

Suppose EA range has keys u,v,w,x,y,z, A and B are comparing

A
\[ H(u,v,w,x,y,z) = h1 \]

B
\[ H(u,v,w,x,y,z) = h2 \]

If N=3, A responsible for keys in dashed range

Here, for EA range, B and C are also responsible
Merkle Trees

Suppose EA range has keys u,v,w,x,y,z, A and B are comparing

A

H(u,v,w,x,y,z) = h1

H(x,y,z) = h3

H(u,v) = h4

B

H(u,v,w,x,y,z) = h2

H(x,y,z) = h3

H(u,v,w) = h5

If N=3, A responsible for keys in dashed range

Here, for EA range, B and C are also responsible
Merkle Trees

Suppose EA range has keys u,v,w,x,y,z, A and B are comparing

If N=3, A responsible for keys in dashed range

Here, for EA range, B and C are also responsible
Merkle Trees

Suppose EA range has keys u,v,w,x,y,z, A and B are comparing

If N=3, A responsible for keys in dashed range

Here, for EA range, B and C are also responsible
Merkle Trees

Suppose EA range has keys $u,v,w,x,y,z$, A and B are comparing

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.

Here, for EA range, B and C are also responsible
## Summary

<table>
<thead>
<tr>
<th>Problem</th>
<th>Technique</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partitioning</td>
<td>Consistent hashing</td>
<td>Incremental scalability</td>
</tr>
</tbody>
</table>
## Summary

<table>
<thead>
<tr>
<th>Problem</th>
<th>Technique</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partitioning</td>
<td>Consistent hashing</td>
<td>Incremental scalability</td>
</tr>
<tr>
<td>Highly available for writes</td>
<td>Vector clocks with read repair</td>
<td>Version size decoupled from update rate</td>
</tr>
</tbody>
</table>
### Summary

<table>
<thead>
<tr>
<th>Problem</th>
<th>Technique</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partitioning</td>
<td>Consistent hashing</td>
<td>Incremental scalability</td>
</tr>
<tr>
<td>Highly available for writes</td>
<td>Vector clocks with read repair</td>
<td>Version size decoupled from update rate</td>
</tr>
<tr>
<td>Handle temporary failures</td>
<td>Sloppy quorum and hinted handoff</td>
<td>HA with some durability</td>
</tr>
</tbody>
</table>
## Summary

<table>
<thead>
<tr>
<th>Problem</th>
<th>Technique</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partitioning</td>
<td>Consistent hashing</td>
<td>Incremental scalability</td>
</tr>
<tr>
<td>Highly available for writes</td>
<td>Vector clocks with read repair</td>
<td>Version size decoupled from update rate</td>
</tr>
<tr>
<td>Handle temporary failures</td>
<td>Sloppy quorum and hinted handoff</td>
<td>HA with some durability</td>
</tr>
<tr>
<td>Recovery from permanent failures</td>
<td>Anti-entropy</td>
<td>Sync replicas w/ Merkle Trees</td>
</tr>
</tbody>
</table>
## Summary

<table>
<thead>
<tr>
<th>Problem</th>
<th>Technique</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partitioning</td>
<td>Consistent hashing</td>
<td>Incremental scalability</td>
</tr>
<tr>
<td>Highly available for writes</td>
<td>Vector clocks with read repair</td>
<td>Version size decoupled from update rate</td>
</tr>
<tr>
<td>Handle temporary failures</td>
<td>Sloppy quorum and hinted handoff</td>
<td>HA with some durability</td>
</tr>
<tr>
<td>Recovery from permanent failures</td>
<td>Anti-entropy</td>
<td>Sync replicas w/ Merkle Trees</td>
</tr>
<tr>
<td>Membership / failure detection</td>
<td>Gossip based membership</td>
<td>Symmetry and no centralized coordination</td>
</tr>
</tbody>
</table>