Dynamo DB - Implementation insight

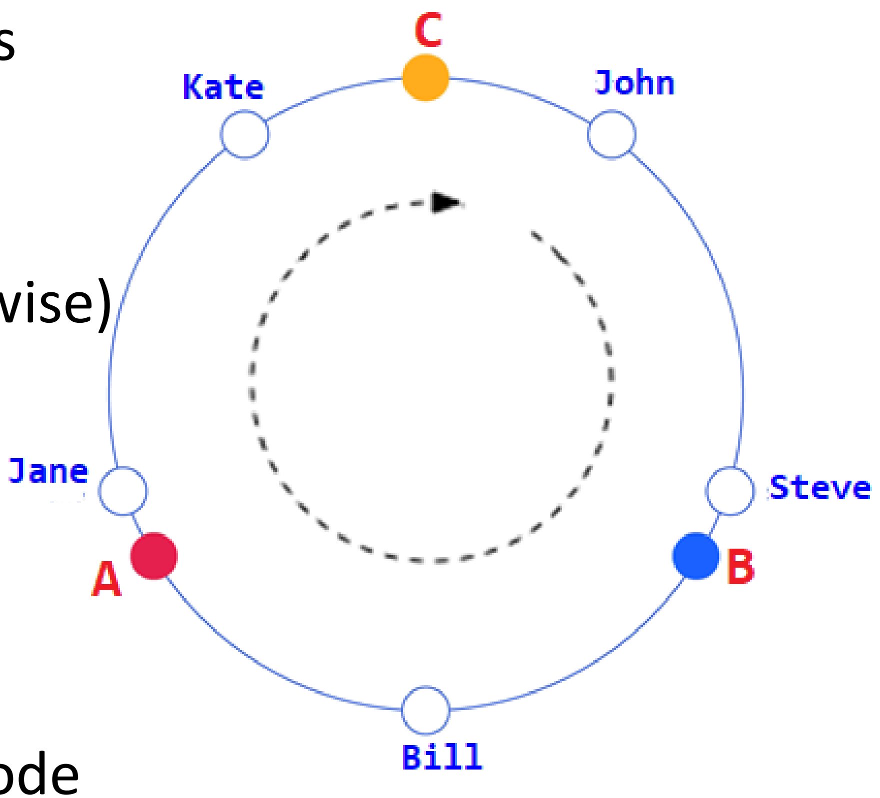
pm_jat @ daiict

Recap: Dynamo DB

- CAP theorem and Dynamo
- What stand Dynamo Takes in regard to CAP theorem
  - Availability: High
  - Partition Tolerance: High
  - Consistency: Low

Consistent Hashing

- Hash function applied on Key maps on ring
- Servers are attached at positions on ring!
- Data with values are placed on next node on the ring (clockwise)
- Virtual Nodes for
  - Uniform Distribution
  - Uniform sharing of load on adding and removing a node
  - Accounting for power of a Node

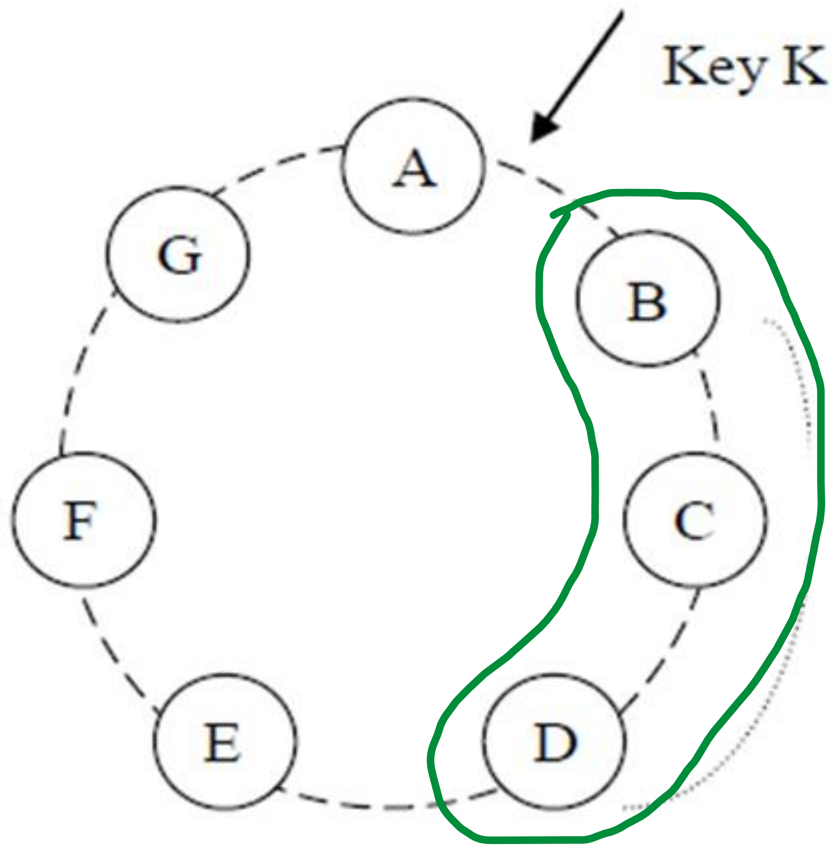


Replication

- Key Value K is **replicated** on B, C, D

  B is coordinator **N-1 replicas**

- **preference list** for k
  - N nodes
- **Extended preference list** to deal with failures > N
- **Distinct physical nodes** (skip virtual)



Today!

- Data Version in Dynamo (for consistency)
- Consistency supported in Dynamo DB
- reading/writing on “Replicated” data
- How failures are dealt!

Data Versioning – Dynamo DB[1]

- **Dynamo DB always write data as “new version”!**
- Dynamo DB uses concept of “Vector Clock” for “data versioning” purposes!
- Vector clock is a “time stamp” information for every version of data; it is a vector because, it holds vector of <nodeid, counter> pairs; where counter is local for every node!
- Every data version keeps record of its writes as “clock vectors”; for example, suppose xi is i-th version of x. A typical vector clock for x is: VC(xi)=([s1,3],[s2,1],[s3,2])

Data Versioning – Dynamo DB[3]

- A typical vector clock for x is: VC(xi)=([s1,3],[s2,1],[s3,2])
- This means x has write counter 3 on server s1, write counter 1 for s2, and 2 for server s3. How does it progresses and used for reconcile when data inconsistency is found?
- **Read S1, S2, S3 as Replica1, Replica2, and Replica3.**
- Dynamo is a peer-to-peer system, and write can be performed on any node;
  - **Write is always as a new version**
  - **Vector clock is saved along with every version**
- Let us try simulating it through an example!

1

<table>
<tr><th colspan="2">D1([S1,1]) | S1</th></tr>
<tr><th>Item</th><th>Qty</th></tr>
<tr><td>A</td><td>1</td></tr>
<tr><td>B</td><td>1</td></tr>
</table>

<table>
<tr><th colspan="2">D1([S1,1]) | S3</th></tr>
<tr><th>Item</th><th>Qty</th></tr>
<tr><td>A</td><td>1</td></tr>
<tr><td>B</td><td>1</td></tr>
</table>

S1 writes data item D;
say ver1 and replicated to all replicas (S2 and S3)

<table>
<tr><th colspan="2">D1([S1,1]) | S2</th></tr>
<tr><th>Item</th><th>Qty</th></tr>
<tr><td>A</td><td>1</td></tr>
<tr><td>B</td><td>1</td></tr>
</table>

- Say, S1 further writes D (ver2)
- Now it is time for replication
- For Replication, can one version replace the existing one?
- Vector clock is compared

  `VC(D1):([s1,1])`

  `VC(D2):([s1,2])`
- When `VC(D2) > VC(D1)`, then D2 can replace D1

  `VC(D2) >= VC(D1)`

  `iff VC(D2)[i] >= VC(D1)[i]`
- This is individually checked at each replica (before overwriting a version)
- Vector clock is also replaced.

2

D2([S1,2]) | S1	
Item	Qty
A	1
B	1
C	1

D1([S1,1]) | S3	
Item	Qty
A	1
B	1

D1([S1,1]) | S2	
Item	Qty
A	1
B	1

Thus, eventually all replica servers gets D2

3

D2([S1,2]) | S1	
Item	Qty
A	1
B	1
C	1

D2([S1,2]) | S3	
Item	Qty
A	1
B	1
C	1

D2([S1,2]) | S2	
Item	Qty
A	1
B	1
C	1

4

- Say S2 writes D ver3;
  its vector clocks becomes as : ([S1,2],[S2,1]), How?
- Can it be replicated (replace) D2 at S1 and S3 also?
- Vector clocks
  VC(D2): ([S1,2])
  VC(D3): ([S1,2],[S2,1])
- Is VC(D3)[i] > VC(D2)[i] ?
- Yes. <u>So D3 can overwrite D2</u>

D2([S1,2]) | S1	
Item	Qty
A	1
B	1
C	1

D2([S1,2]) | S3	
Item	Qty
A	1
B	1
C	1

D3([S1,2],[S2,1]) | S2	
Item	Qty
A	1
B	2
C	1

(1) D3 has been replicated to S1

5

D3([S1,2],[S2,1]) | S1	
Item	Qty
A	1
B	2
C	1

D4([S1,2],[s3,1]) | S3	
Item	Qty
A	1
C	2

D3([S1,2],[S2,1]) | S2	
Item	Qty
A	1
B	2
C	1

(2) Before D3 got a chance to replicate at S3; S3 has written D4?

Can D4 overwrite D3?
Can D3 overwrite D4?
VC(D3): ([S1,2],[S2,1])
VC(D4): ([S1,2],[S3,1])

(1) D3 has been replicated to S1

5

D3([S1,2],[S2,1]) | S1	
Item	Qty
A	1
B	2
C	1

D4([S1,2],[s3,1]) | S3	
Item	Qty
A	1
C	2

D3([S1,2],[S2,1]) | S2	
Item	Qty
A	1
B	2
C	1

(2) Before D3 got a chance to replicate at S3; S3 has written D4?

No. Neither D4 can overwrite D3 nor D3 can overwrite D4

How can all replicas be brought consistent?

Conflict Resolution is done read time!

- Version 3 and version 4 are parallel writes over version 2
- On finding conflict like this, new version is created with “merged data” and “combined vector clock”
- This conflict happened, because we allowed to write optimistically without checking for conflicts.
- Some thing that is never seen in conventional databases.
- This is consistency compromise for Availability!

6 Conflict Resolution!

- New version d5 is created that combines both versions.
- Combine vector clock. Note, how vector clock is combined?
- VC(D5) > VC(D3) and VC(D4) and therefore can replace D3 and D4, and eventually all replicas will have this D5 with VC(D5)!

D4([S1,2],[s3,1]) | S3	
Item	Qty
A	1
C	2

D3([S1,2],[S2,1]) | S2	
Item	Qty
A	1
B	2
C	1

D5([S1,2],[S2,2],[S3,1]) | S2	
Item	Qty
A	1
B	2
C	1
A	1
C	2

7 Conflict Resolution!

Eventually all replicas gets D5 with Vector Clock of D5

D5([S1,2],[S2,2],[S3,1]) | S1	
Item	Qty
A	1
B	2
C	1
A	1
C	2

D5([S1,2],[S2,2],[S3,1]) | S3	
Item	Qty
A	1
B	2
C	1
A	1
C	2

D5([S1,2],[S2,2],[S3,1]) | S2	
Item	Qty
A	1
B	2
C	1
A	1
C	2

Summarized: Vector Clock

- Dynamo DB uses concept of “Vector Clock” for “data versioning” purposes, and that is used for
  - (1) delivering appropriate version of data to the read requests!
  - (2) resolving the conflicts in data versions

Read/Write in Dynamo DB[1]

- In Dynamo DB, any node can receive get and put requests.
- A node handling a read or write operation is known as the *coordinator*.
- **Read and write operations involve the first N healthy nodes** from the **preference list**, skipping over those that are down or inaccessible.
- When all nodes are healthy, the top N nodes in a key's preference list are accessed.
- **When any node fails or network is unreachable, nodes that are lower ranked in the preference list are accessed**.

Read/Write in Dynamo DB[1]

- To maintain the consistency in replicas, Dynamo uses a “quorum-like” technique to decide whether an operation should be declared successful or not?
  - That is perform operation at multiple nodes and **return response where more than “one node” agrees** on!
- This approach allows specifying
  - minimum number of nodes to be consulted for Read, and
  - Minimum number of nodes where write needs to successful before a write can be called to be successful.

Read/Write in Dynamo DB[1]

- So, here we have two configurable parameters: R and `W`.
  - **R** is the minimum number of nodes that must participate in a successful read operation.
  - **`W`** is the minimum number of nodes that must participate in a successful write operation.
- Now, what value should be for **R** and **`W`** be appropriate?
- Consider example next:

What value for R and W?

D3([S1,2],[S2,1]) | S1	
Item	Qty
A	1
B	2
C	1

D2([S1,2]) | S3
Item
A
B
C

D4([S1,2],[s3,1]) | S3	
Item	Qty
A	1
C	2

D3([S1,2],[S2,1]) | S2	
Item	Qty
A	1
B	2
C	1

Suppose, A write request of D lands to S3.
Now
<u>Option one</u>: system writes it as D4 at S3 only, returns, and confirms that write has been successful.

<u>Other option</u>: before returning, system attempt replicating on few more replicas.

Which one has what advantage/disadvantage

What value for R and W?

- **System writing only on one server and returning is most prompt.** And thus, **most available**.
- While increase on W reduces the availability, it improves upon consistency.
- And, we can say that Consistency and Availability compete!
- So, what value for R and W?
- Setting R and W such that "**`R + W > N`**" yields some overlap of read and write would have "a quorum-like system", and should provide good tradeoff for "Consistency" and "Availability"?

Execute “PUT” Request

- Upon receiving a put() request for a key, the coordinator generates the vector clock for the **new version** and writes the new version locally.
- The coordinator then **sends** the new version (along with the new vector clock) to the N highest-ranked reachable nodes **for replication**
- If at least **W-1 nodes respond**, then the write is considered successful.

Execute “GET” Request

- For a GET request, the coordinator requests to N highest-ranked reachable nodes in the preference list for that key , and then
- **waits for R responses** before returning the result to the client.
- If the coordinator ends up gathering multiple versions of the data,
  - Sees if can be reconciled,
  - If not, then it returns all the versions

  As already discussed!

Consistency Options in Dynamo API

- Dynamo API does not allow user's controlling R and W but allows specifying the desired consistency level
- It can be one of following: "***eventually consistent read***" and "***strongly consistent* reads**"
- Hopefully, you already have understood what these are?
- **Eventually Consistent Reads:**
  - When we read data from a Dynamo DB table, the response might not reflect the results of a recently completed write operation.
  - The response might include some stale data.
  - If you repeat your read request after a short time, the response should return the latest data.

https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.ReadConsistency.html

Consistency Options in Dynamo API

- **Strongly Consistent Reads**
  - When you request a strongly consistent read, Dynamo DB **returns a response with the most up-to-date data**, reflecting the updates from all prior write operations that were successful.
- However, this consistency comes with some cost and disadvantages:
  - Strongly consistent reads may have higher latency than eventually consistent reads.
  - A strongly consistent read might not be available if there is a network delay or outage. In this case, Dynamo DB may return a server error (HTTP 500).
  - Strongly consistent reads are not supported on global secondary indexes.
  - Strongly consistent reads may use “more resources”

https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.ReadConsistency.html

Consistency Options in Dynamo API

- With get item request of API we can specify option for "consistency"
- When `--consisten-read` is set, then system attempts to provide "strong consistency" otherwise "eventual consistency"

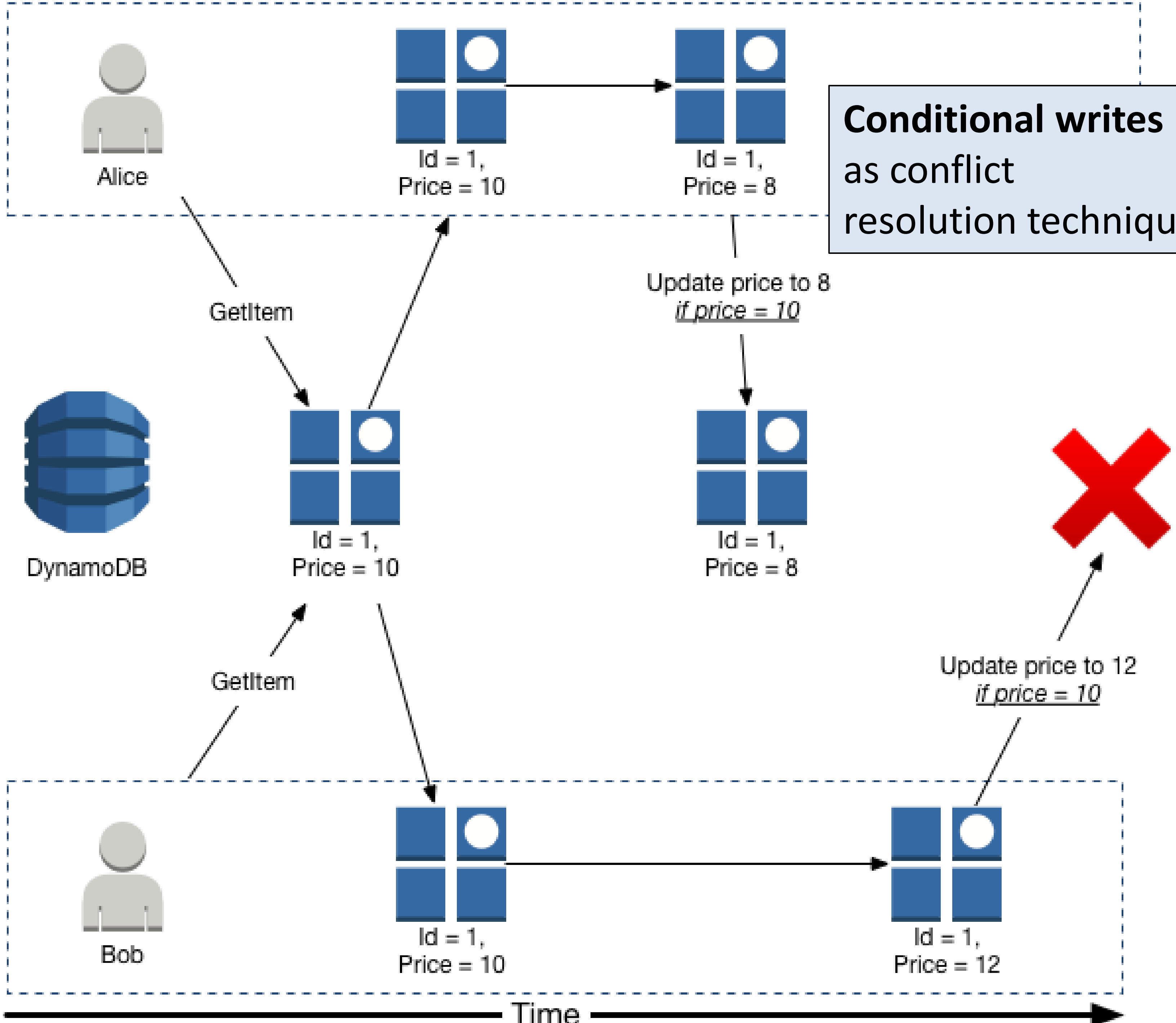
```
  get-item
--table-name <value>
--key <value>
[--attributes-to-get <value>]
[--consistent-read | --no-consistent-read]
[--return-consumed-capacity <value>]
[--projection-expression <value>]
[--expression-attribute-names <value>]
[--cli-input-json <value>]
[--generate-cli-skeleton <value>]
```


https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.ReadConsistency.html

Consistency Options in Dynamo API

- Current version of Dynamo also supports
  - "**conditional writes**" to avoid write-write conflicts
  - Can have better consistency by having "**Optimistic Locking with Version Number**"

Conditional writes as conflict resolution technique



Handling Failures: Hinted Handoff

- To deal with temporary failures, that is failures hopefully for a very short time, whatever could be reason.
- In this case what dynamo DB does is as following:
  - NEXT SLIDE

Temporary failures: hinted handoff

- Uses ‘temporary’ replicas
- Example:
  - B is out; E takes over
  - R/W for B are forwarded to E
  - E keeps a hint in the metadata (“meant for B”)
- replica transferred to B when recovers & cancelled from E

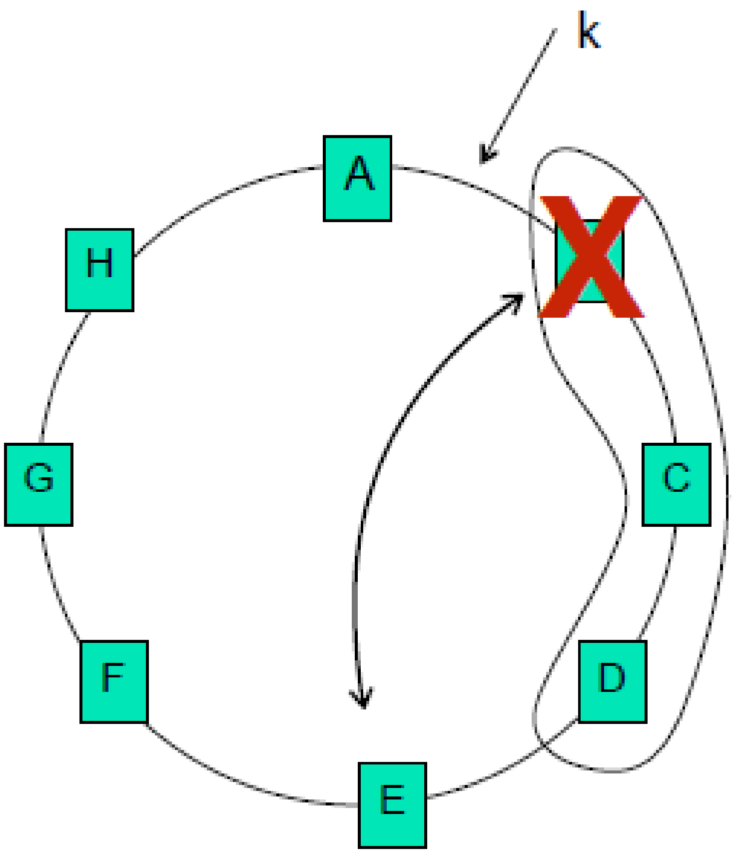


Fig source: http://wwwusers.di.uniroma1.it/~stefa/Distributed_Systems_18-19/Schedule_files/dynamo%202.pdf

Handling Failures: Hinted Handoff

- Using hinted handoff, Dynamo ensures that the read and write operations are not failed due to temporary node or network failures.
- Note that
  - Dynamo uses “sloppy quorum”, that is read write operations are performed on N healthy nodes from the preference list.
  - Due to failure, it can not have fixed set of nodes for “quorum” and that is what sloppy about it.

Handling permanent failures

- For permanent failures, say when failed node never returns back, a temporary may take the role of permanent replica.
- This **requires to ensure that replicas are in sync**.
- Dynamo implements “Anti Entropy” based Replica Synchronization protocol.
- Entropy often used as measure of homogeneity in many computing problems.
- Anti Entropy refers Anti Homogeneity, that is Heterogeneity. Here heterogeneity in “replica set” is used as measure of disagreement in replicas.
- Dynamo uses Merkle trees for measuring inconsistencies in replicas.

References/Further Readings

[1] DeCandia, Giuseppe, et al. "Dynamo: amazon's highly available key-value store." *ACM SIGOPS operating systems review* 41.6 (2007): 205-220. https://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf

[2] Karger, David, et al. "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web." *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*. 1997.

[3] http://wwwusers.di.uniroma1.it/~stefa/Distributed_Systems_18-19/Schedule_files/dynamo%202.pdf

[4] White, Tom: *Consistent Hashing*. November 2007. – Blog post http://tom-e-white.com/2007/11/consistent-hashing.html .

[5] Deshpande, Tanmay. *Mastering DynamoDB*. Packt Publishing Ltd, 2014.