Big Data

Key-value store

KV-stores and Relational Tables

- KV-stores seem very simple. They can be viewed as two-column (key, value) tables with a single key column.
- But they can be used to implement more complicated relational tables:

State	ID	Population	Area	Senator_1
Alabama	1	4,822,023	52,419	Sessions
Alaska	2	731,449	663,267	Begich
Arizona	3	6,553,255	113,998	Boozman
Arkansas	4	2,949,131	53,178	Flake
California	5	38,041,430	163,695	Boxer
Colorado	6	5,187,582	104,094	Bennet
…	…			

Index

KV-stores and Relational Tables

The KV-version of the previous table includes one table indexed by the actual key, and others by an ID.

State	ID
Alabama	1
Alaska	2
Arizona	3
Arkansas	4
California	5
Colorado	6
…	…

ID	Population
1	4,822,023
2	731,449
3	6,553,255
4	2,949,131
5	38,041,430
6	5,187,582
…	…

ID	Area
1	52,419
2	663,267
3	113,998
4	53,178
5	163,695
6	104,094
…	…

ID	Senator_1
1	Sessions
2	Begich
3	Boozman
4	Flake
5	Boxer
6	Bennet
…	…

KV-stores and Relational Tables

We can add indices with new KV-tables:

Thus KV-tables are used for **column-based storage**, as opposed to row-based storage typical in older DBMS.

State	ID
Alabama	1
Alaska	2
Arizona	3
Arkansas	4
California	5
Colorado	6
…	…

Index

ID	Population
1	4,822,023
2	731,449
3	6,553,255
4	2,949,131
5	38,041,430
6	5,187,582
…	…

…

Senator_1	ID
Sessions	1
Begich	2
Boozman	3
Flake	4
Boxer	5
Bennet	6
…	…

Index_2

OR: the value field can contain complex data

Key-Values: Examples

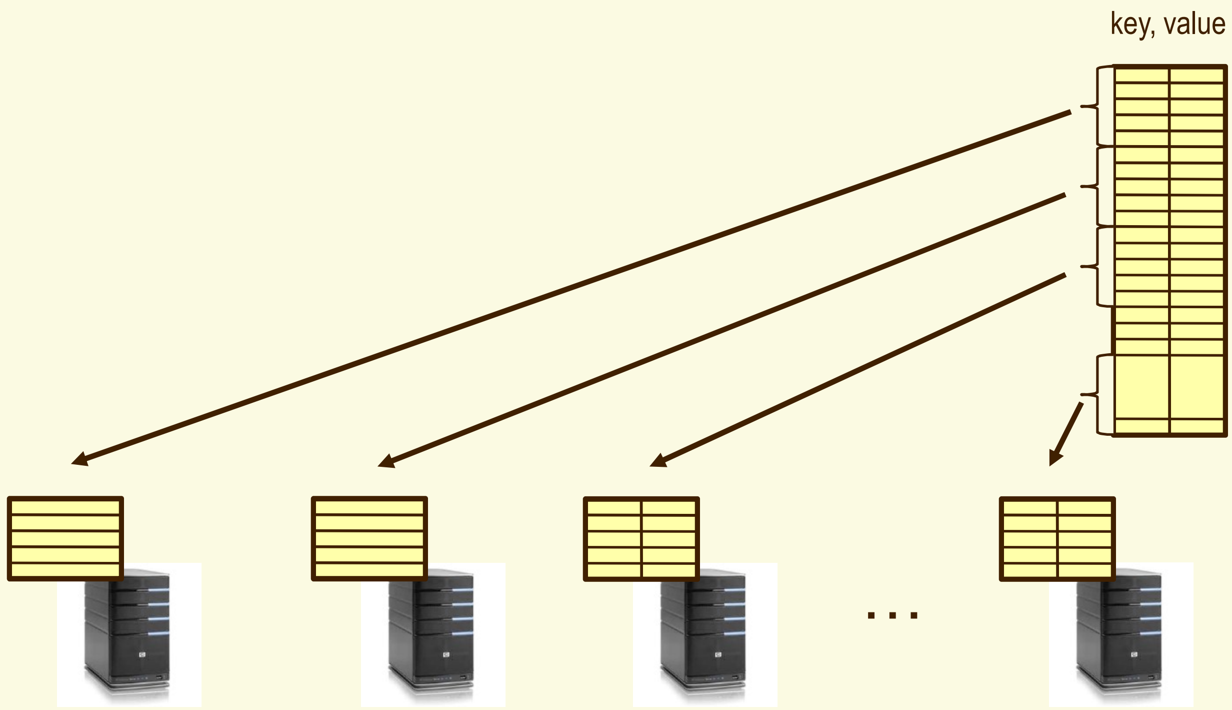
- ✓ Amazon:
  - Key: customerID
  - Value: customer profile (e.g., buying history, credit card, ..)
- ✓ Facebook, Twitter:
  - Key: UserID
  - Value: user profile (e.g., posting history, photos, friends, …)
- ✓ iCloud/iTunes:
  - Key: Movie/song name
  - Value: Movie, Song
- ✓ Distributed file systems
  - Key: Block ID
  - Value: Block

System Examples

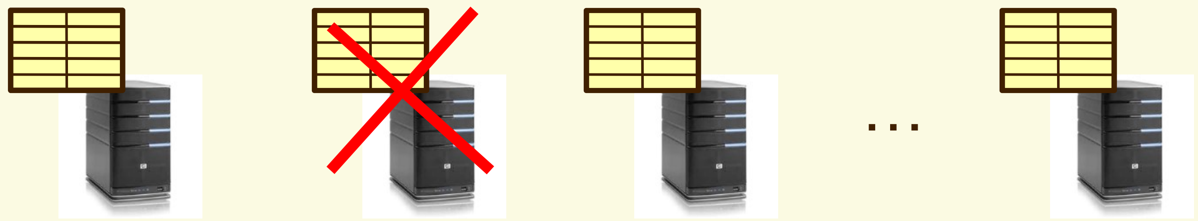
- **Google File System, Hadoop Dist. File Systems (HDFS)**
- **Amazon**
  - Dynamo: internal key value store used to power Amazon.com (shopping cart)
  - Simple Storage System (S3)
- **BigTable/HBase/Hypertable:** distributed, scalable data storage
- **Cassandra**: "distributed data management system" (Facebook)
- **Memcached:** in-memory key-value store for small chunks of arbitrary data (strings, objects)
- **eDonkey/eMule:** peer-to-peer sharing system

Key-Value Store

- Also called a Distributed Hash Table (DHT)
- Main idea: partition set of key-values across many machines



Challenges



- **Fault Tolerance:** handle machine failures without losing data and without degradation in performance
- **Scalability:**
  - Need to scale to thousands of machines
  - Need to allow easy addition of new machines
- **Consistency:** maintain data consistency in face of node failures and message losses
- **Heterogeneity** (if deployed as peer-to-peer systems):
  - Latency: 1ms to 1000ms
  - Bandwidth: 32Kb/s to several GB/s

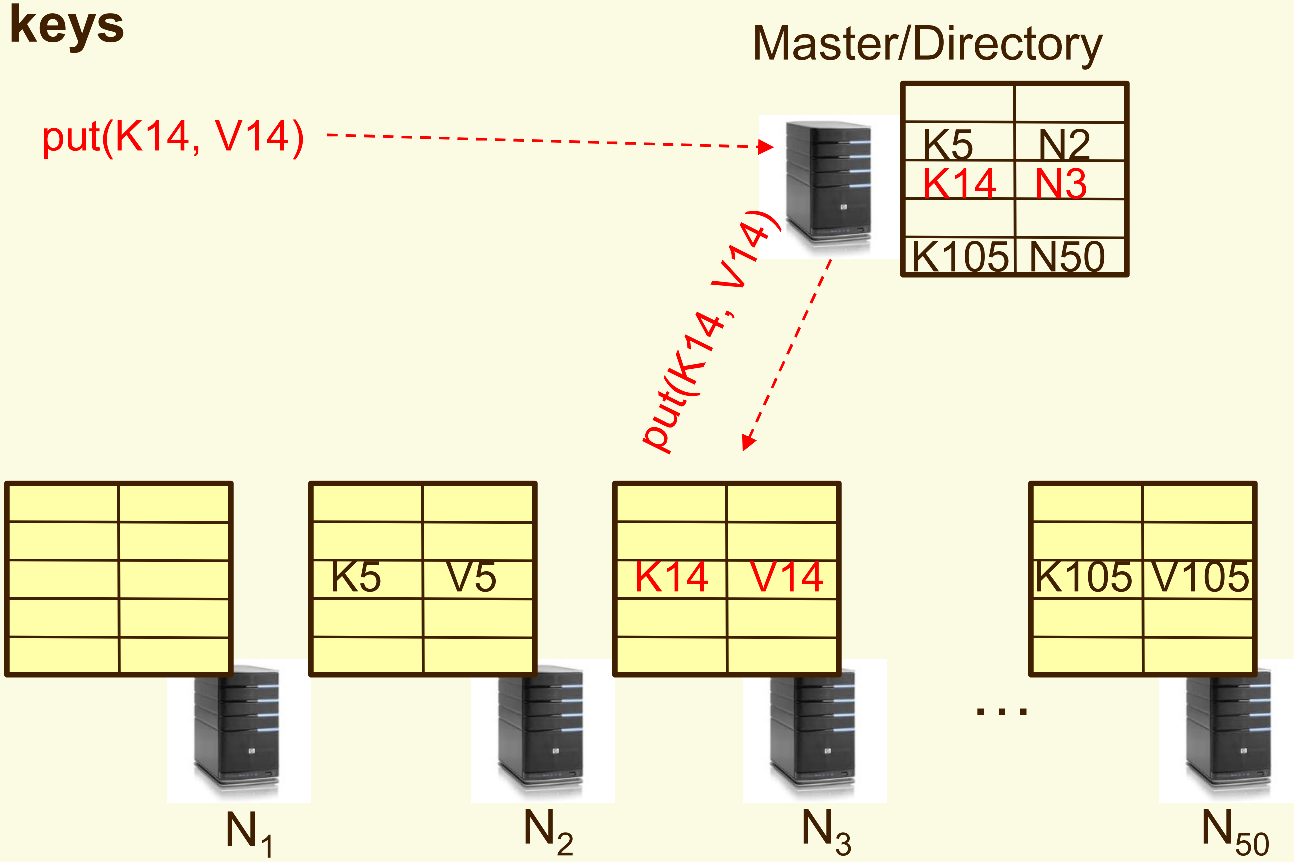
Key Operators

- ✓ put(key, value): where do you store a new (key, value) tuple?
- ✓ get(key): where is the value associated with a given "key" stored?

- ✓ And, do the above while providing
  - Fault Tolerance
  - Scalability
  - Consistency

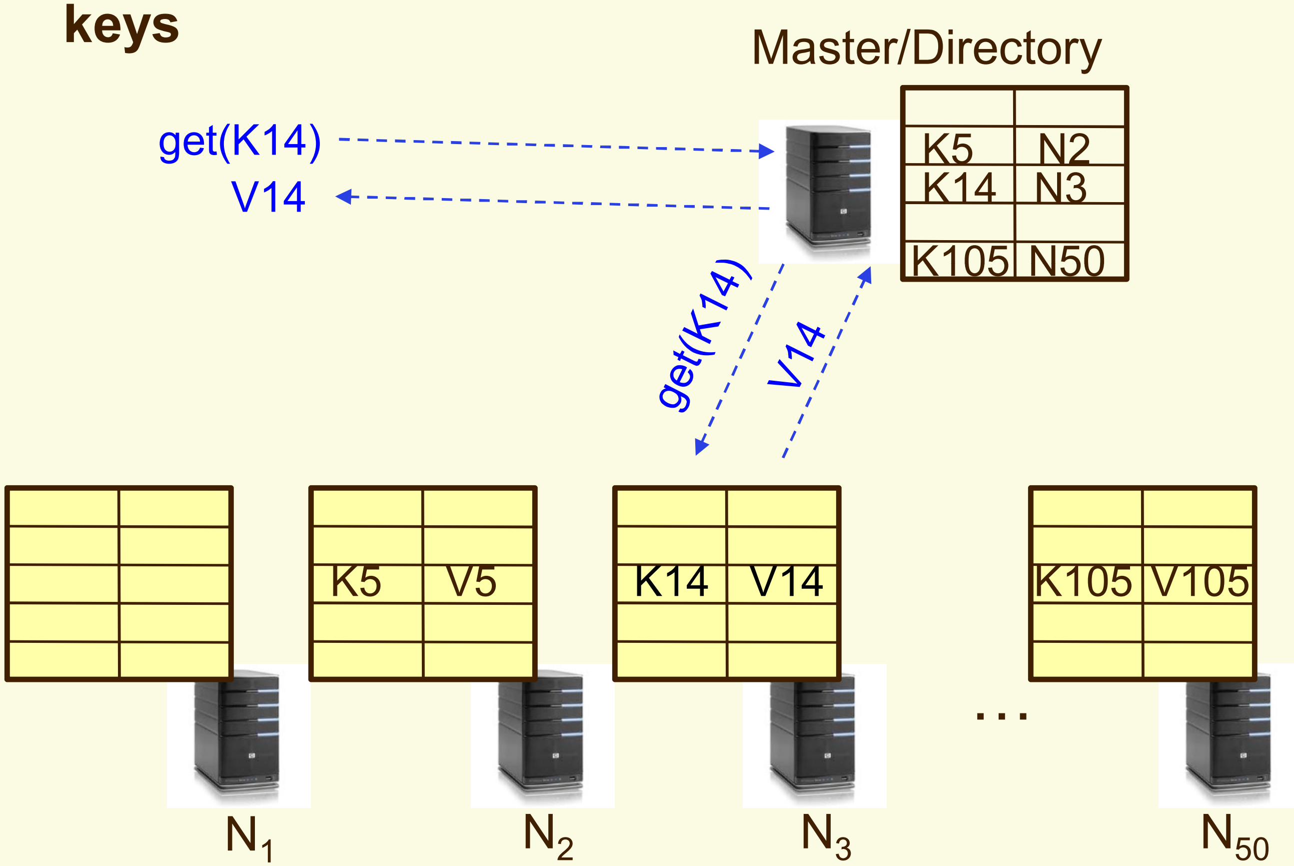
Directory-Based Architecture

- Have a node maintain the mapping between **keys** and the **machines (nodes)** that store the **values** associated with the **keys**



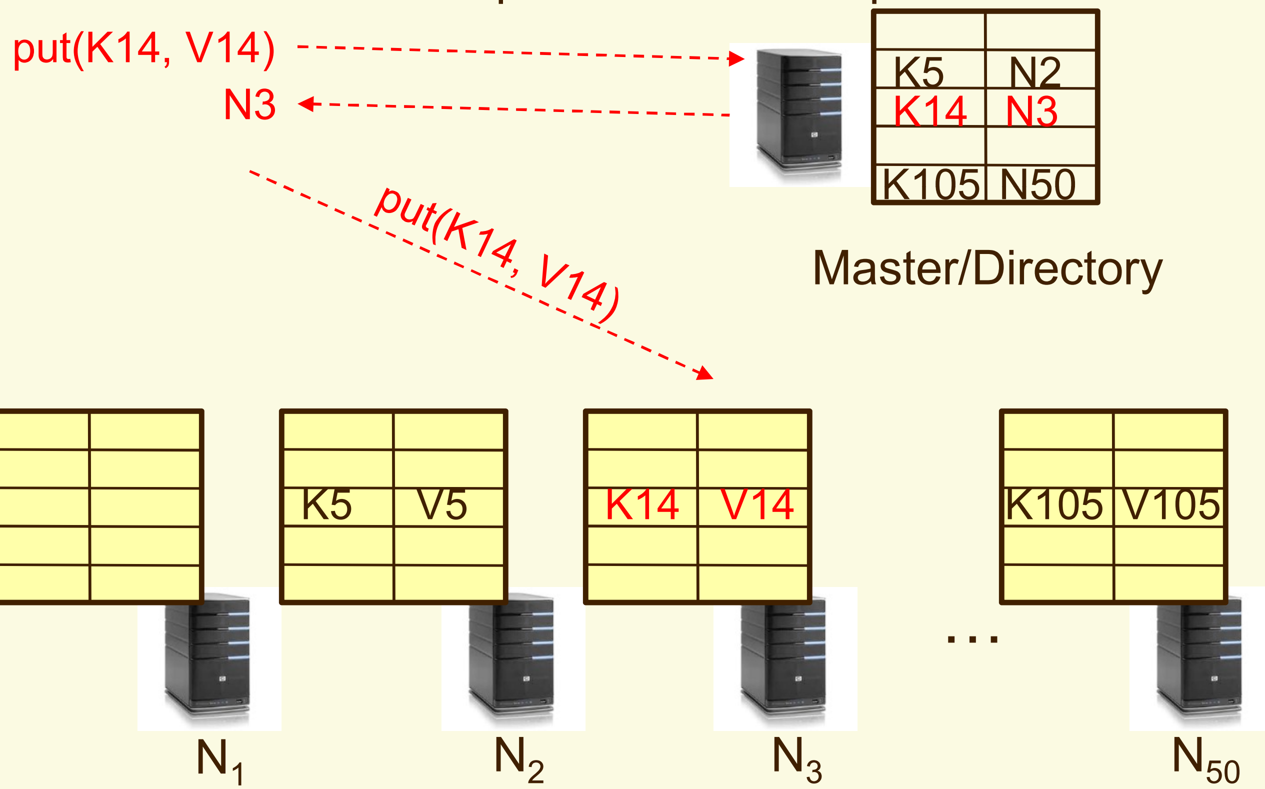
Directory-Based Architecture

- Have a node maintain the mapping between **keys** and the **machines (nodes)** that store the **values** associated with the **keys**



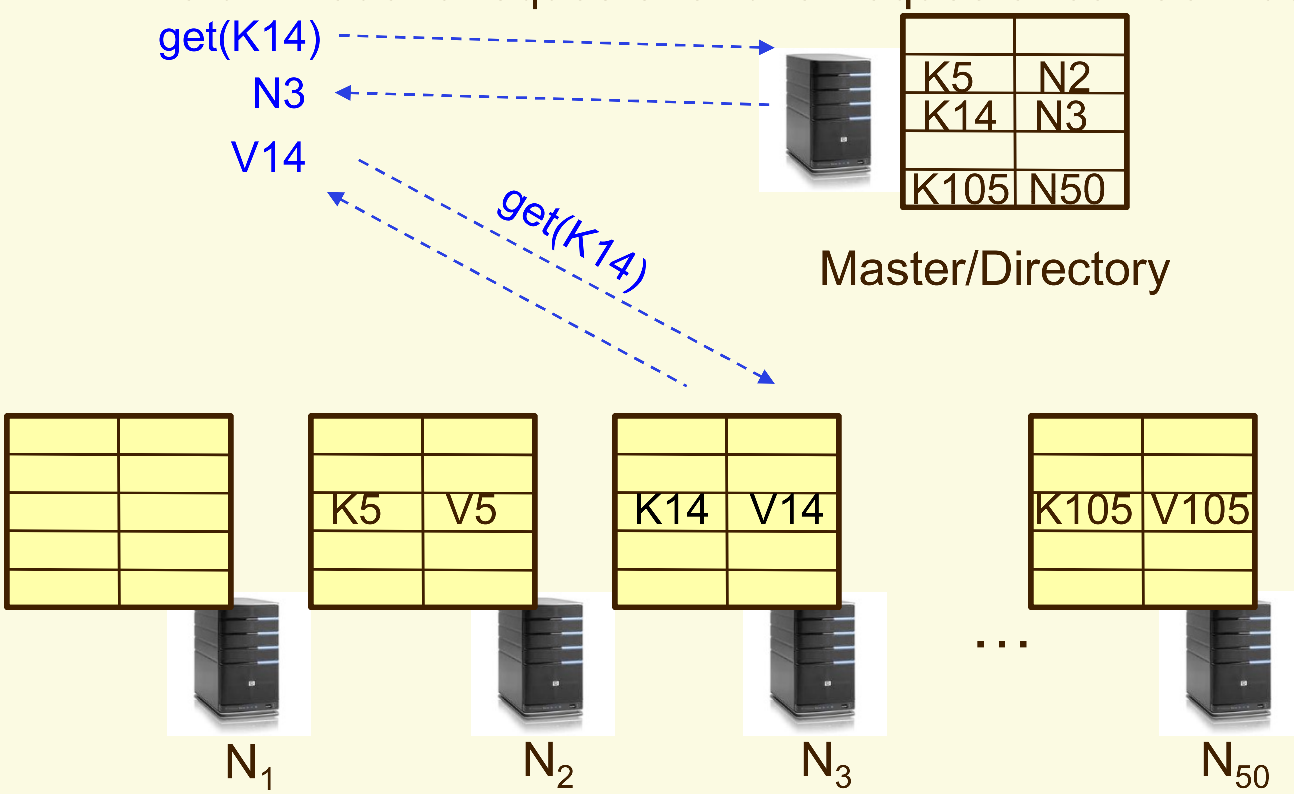
Directory-Based Architecture

- Having the master relay the requests → **recursive query**
- Another method: **iterative query** (this slide)
  - Return node to requester and let requester contact node

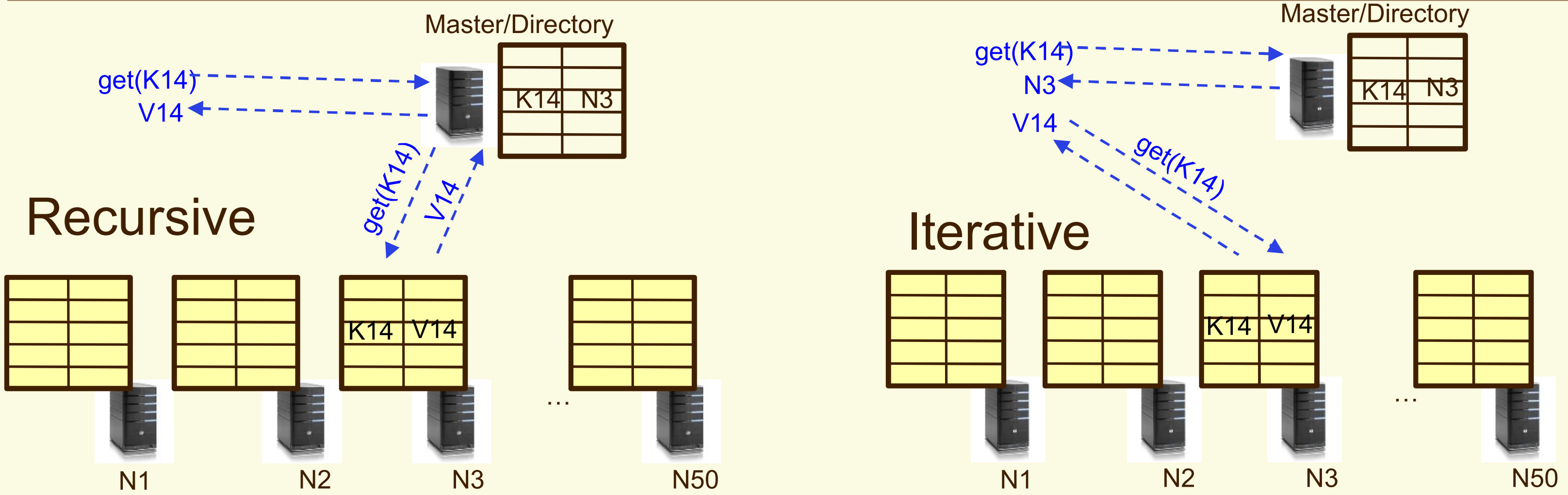


Directory-Based Architecture

- Having the master relay the requests → **recursive query**
- Another method: **iterative query**
  - Return node to requester and let requester contact node



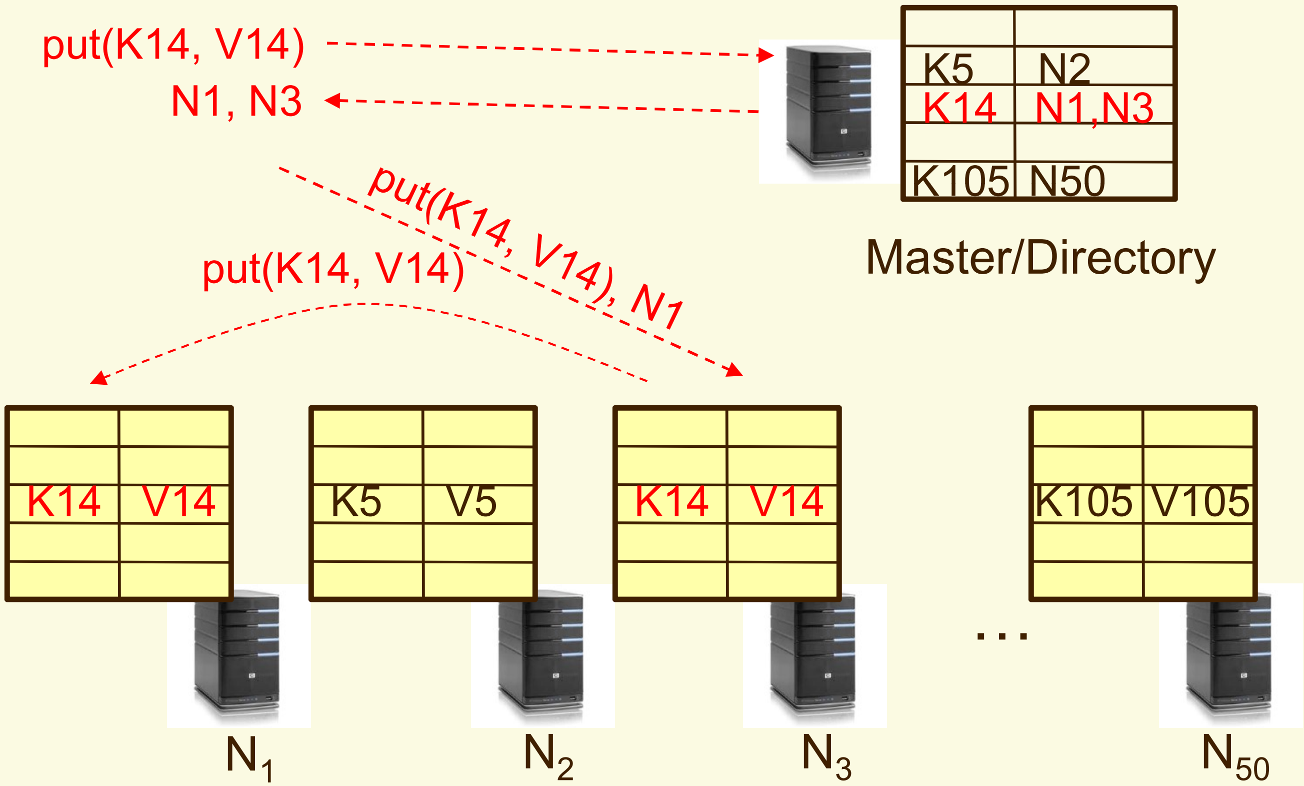
Iterative vs. Recursive Query



- ✓ Recursive Query:
  - Advantages:
    - Faster (latency), as typically master/directory closer to nodes
    - Easier to maintain consistency, as master/directory can serialize puts()/gets()
  - Disadvantages: scalability bottleneck, as all "Values" go through master/directory
- ✓ Iterative Query
  - Advantages: more scalable
  - Disadvantages: slower (latency), harder to enforce data consistency

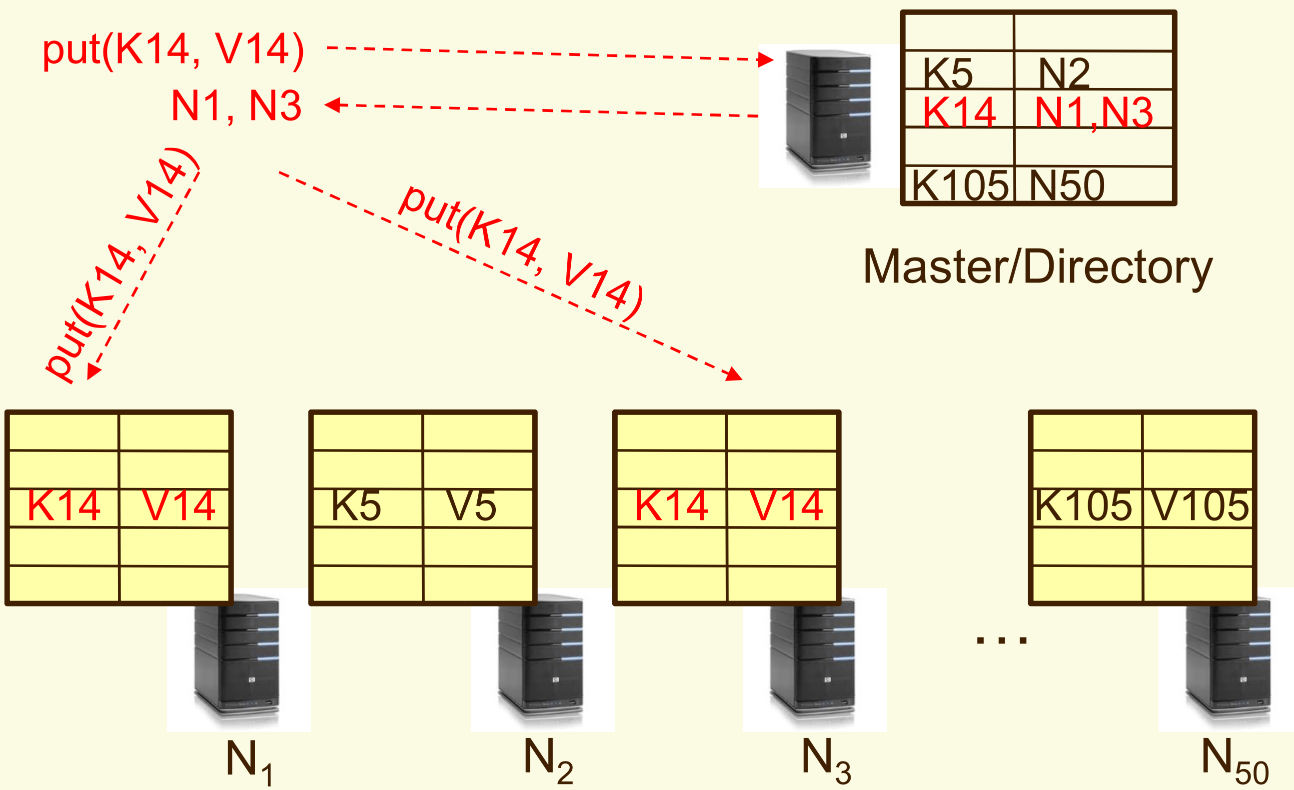
Fault Tolerance

- Replicate value on several nodes
- Usually, place replicas on different racks in a datacenter to guard against rack failures



Fault Tolerance

- Again, we can have
  - **Recursive** replication (previous slide)
  - **Iterative** replication (this slide)

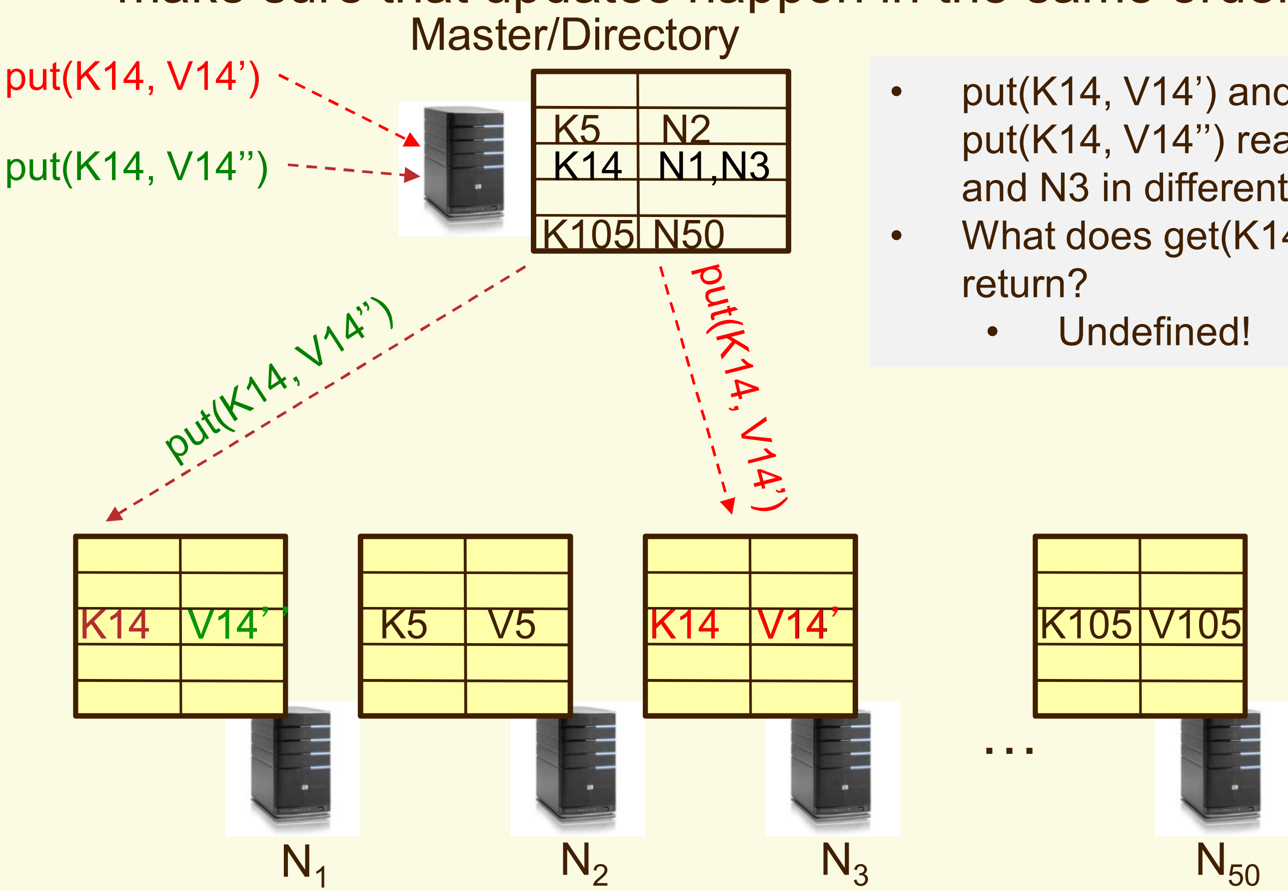


Consistency

- ✓ How close does a distributed system emulate a single machine in terms of read and write semantics?
- ✓ **Q:** Assume **put(K14, V14')** and **put(K14, V14'')** are concurrent, what value ends up being stored?
- ✓ **Q:** Assume a client calls **put(K14, V14)** and then **get(K14)**, what is the result returned by **get()**?
- ✓ Above semantics, not trivial to achieve in distributed systems

Concurrent Writes (Updates)

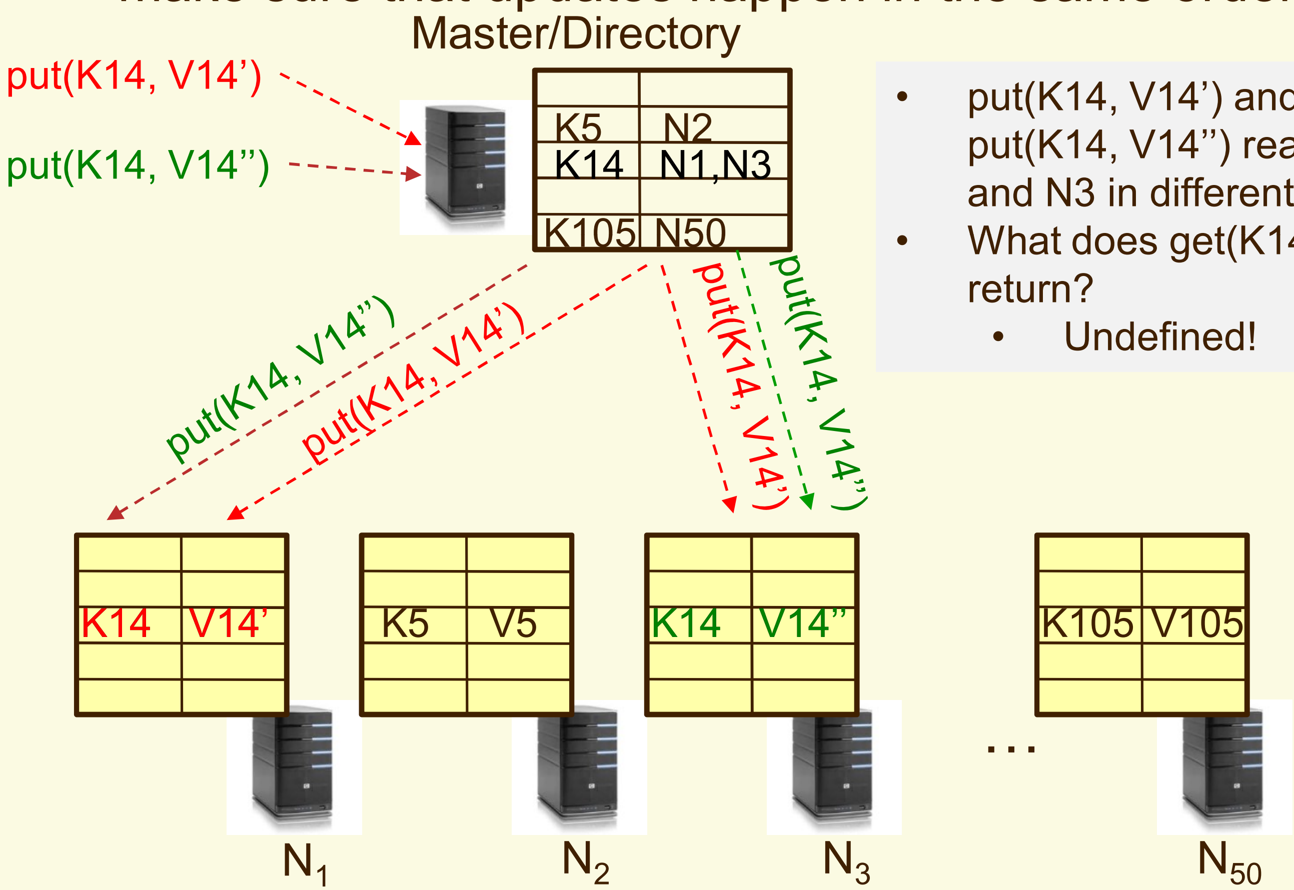
- ✓ If concurrent updates (i.e., puts to same key) may need to make sure that updates happen in the same order



- put(K14, V14') and put(K14, V14'') reach N1 and N3 in different order
- What does get(K14) return?
    - Undefined!

Concurrent Writes (Updates)

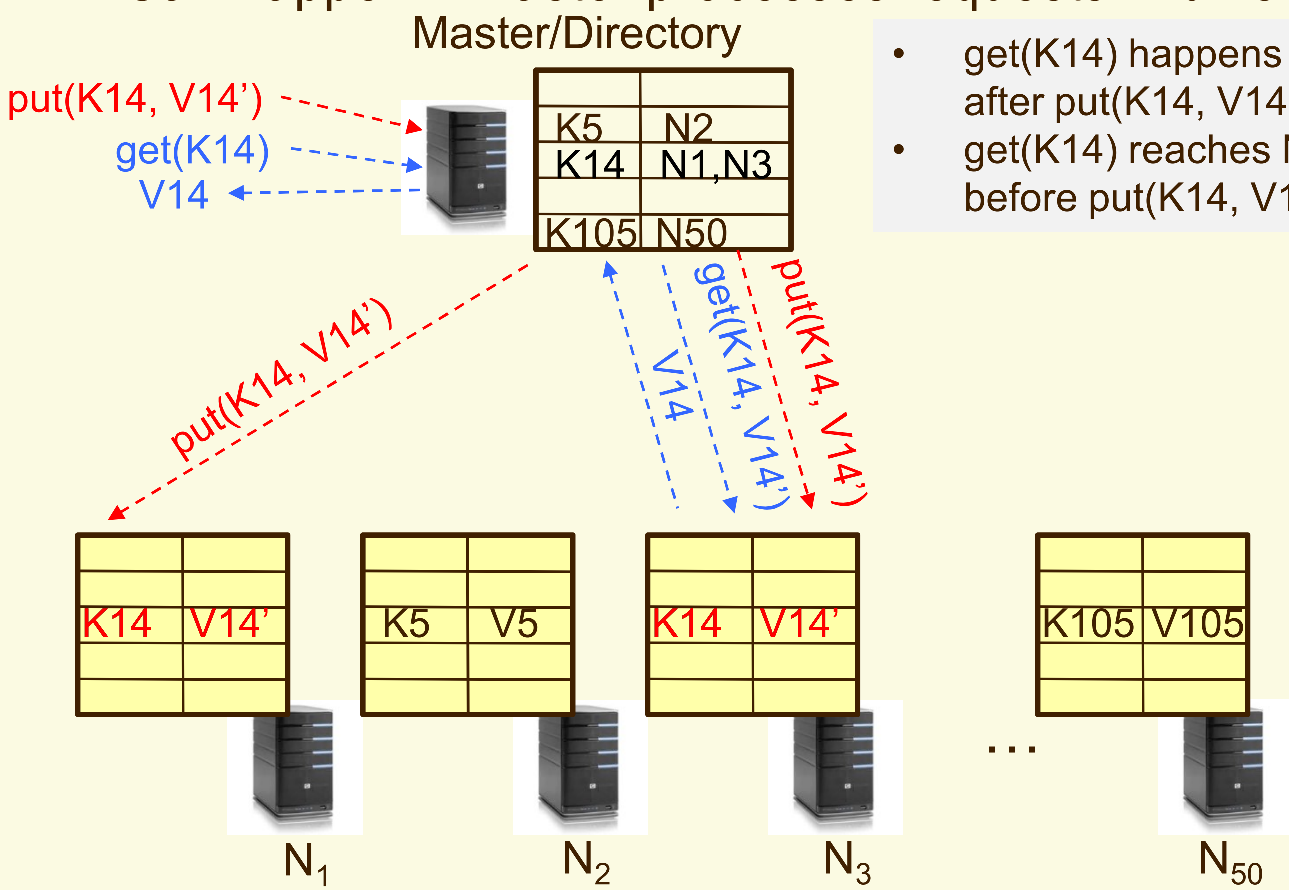
- ✓ If concurrent updates (i.e., puts to same key) may need to make sure that updates happen in the same order



- put(K14, V14') and put(K14, V14'') reach N1 and N3 in different order
- What does get(K14) return?
  - Undefined!

Read after Write

- ✓ Read not guaranteed to return value of latest write
  - Can happen if Master processes requests in different threads



- get(K14) happens right after put(K14, V14')
- get(K14) reaches N3 before put(K14, V14')!

Strong Consistency

- ✓ Assume Master serializes all operations
- ✓ Challenge: master becomes a bottleneck
- ✓ Still want to improve performance of reads/writes → quorum consensus

Quorum Consensus

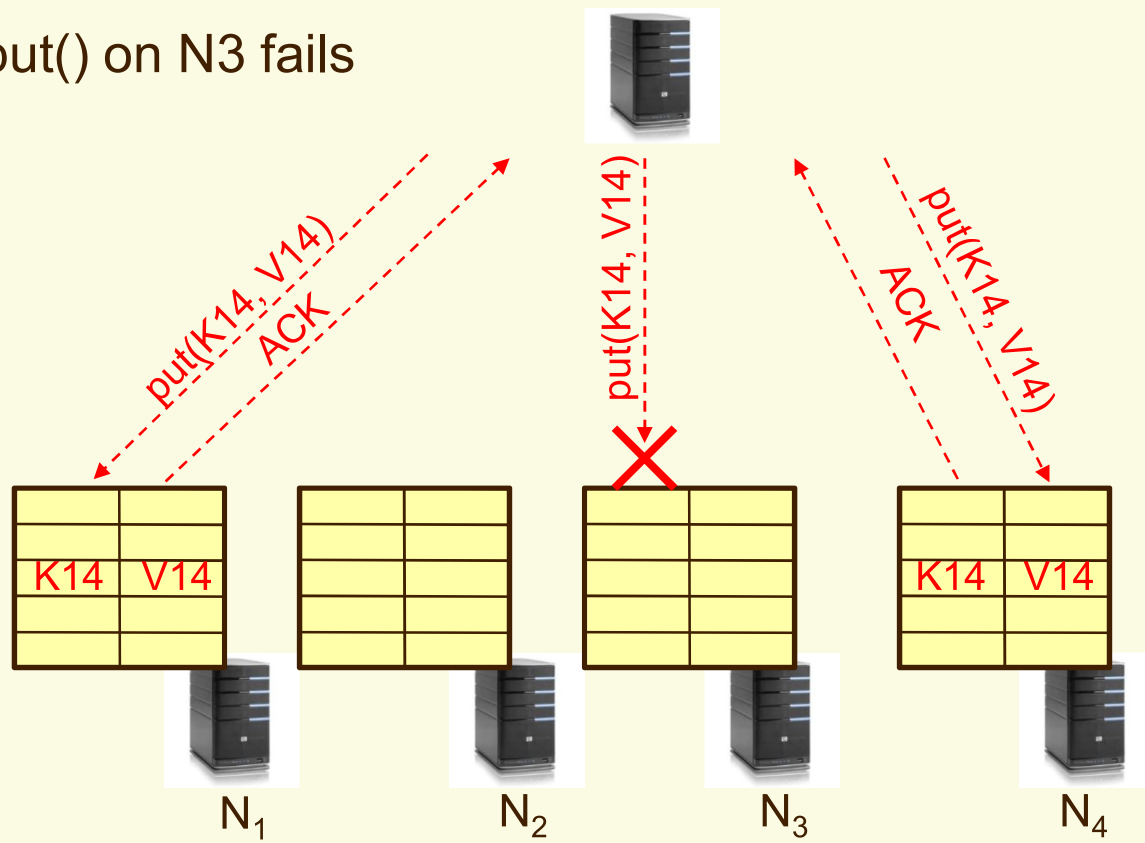
- Improve **put()** and **get()** operation performance

- Define a replica set of size N
- **put()** waits for acks from at least W replicas
- **get()** waits for responses from at least R replicas
- W+R > N

- Why does it work?
  - There is at least one node that contains the update

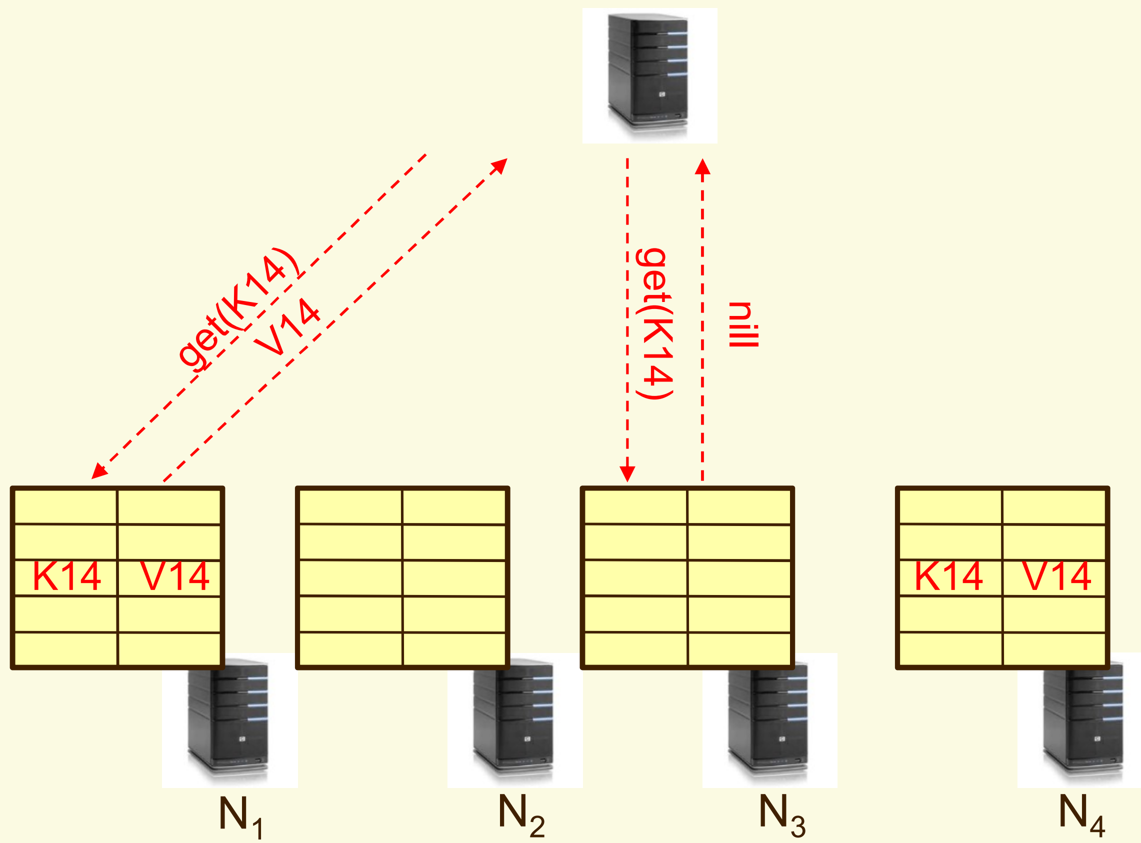
Quorum Consensus Example

- N=3, W=2, R=2
- Replica set for K14: {N1, N3, N4}
- Assume put() on N3 fails



Quorum Consensus Example

- Now, issuing get() to any two nodes out of three will return the answer



Scalability

- Storage: use more nodes
- Request Throughput:
  - Can serve requests from all nodes on which a value is stored in parallel
  - Large “values” can be broken into blocks (HDFS files are broken up this way)
  - Master can replicate a popular value on more nodes
- Master/directory scalability:
  - Replicate it
  - Partition it, so different keys are served by different masters/directories

Scalability: Load Balancing

- ✓ Directory keeps track of the storage availability at each node
  - Preferentially insert new values on nodes with more storage available
- ✓ What happens when a new node is added?
  - Move values from the heavy loaded nodes to the new node
- ✓ What happens when a node fails?
  - Need to replicate values from failed node to other nodes

Replication Challenges

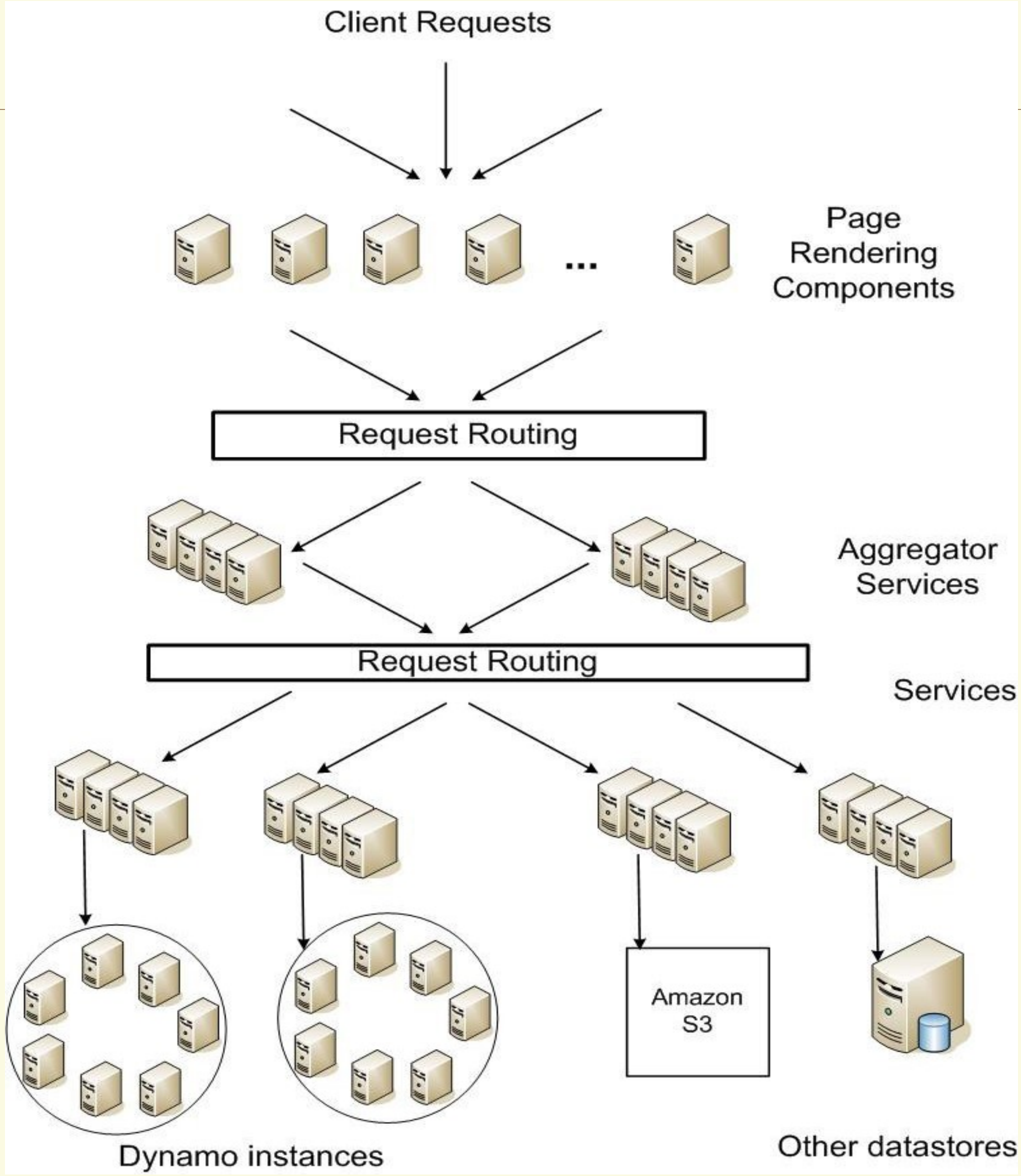
- ✓ Need to make sure that a value is replicated correctly
- ✓ How do you know a value has been replicated on every node?
  - Wait for acknowledgements from every node
- ✓ What happens if a node fails during replication?
  - Pick another node and try again
- ✓ What happens if a node is slow?
  - Slow down the entire put()? Pick another node
- ✓ In general, with multiple replicas
  - Slow puts and fast gets

Summary: Key-Value Store

- ✓ Very large scale storage systems
- ✓ Two operations
  - put(key, value)
  - value = get(key)
- ✓ Challenges
  - Fault Tolerance → replication
  - Scalability → serve get()'s in parallel; replicate/cache hot tuples
  - Consistency → quorum consensus to improve put/get performance
- ✓ System case study: Dynamo

Key-value store case study: Dynamo

Amazon's platform architecture



System Assumptions and Requirements

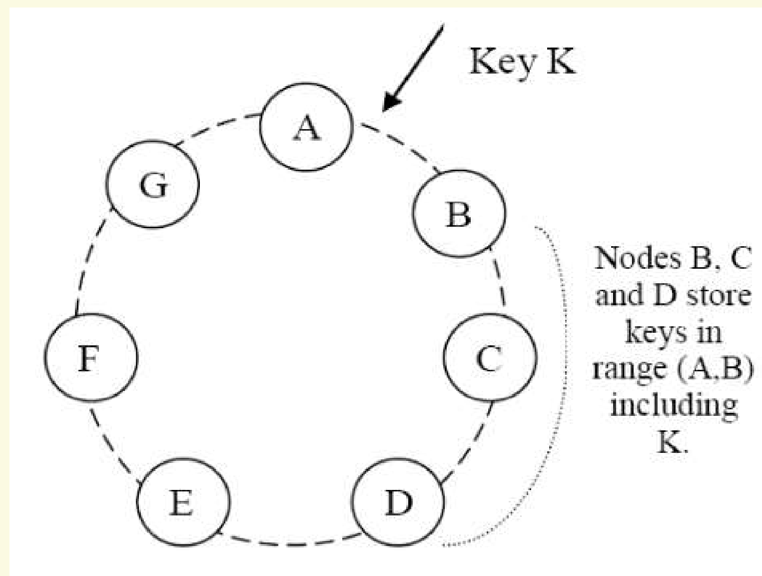
- simple read and write operations to a data item that is uniquely identified by a key.
- Most of Amazon's services can work with this simple query model and do not need any relational schema.
-
- targeted applications - store objects that are relatively small (usually less than 1 MB)
- Dynamo targets applications that operate with weaker consistency (the "C" in ACID) if this results in high availability.

System architecture

- Partitioning
- High Availability for writes
- Handling temporary failures
- Recovering from permanent failures
- Membership and failure detection

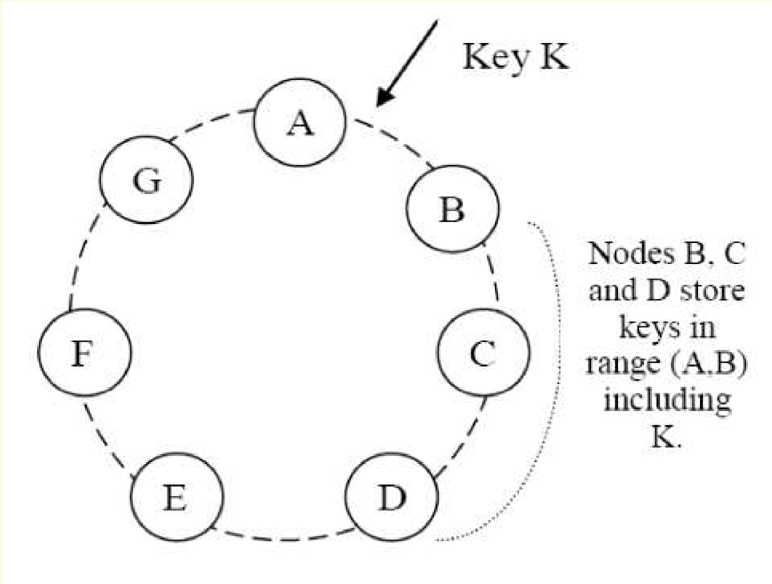
Partition Algorithm

- *Consistent hashing*: the output range of a hash function is treated as a fixed circular space or “ring”.
- *”Virtual Nodes”:* Each node can be responsible for more than one virtual node.



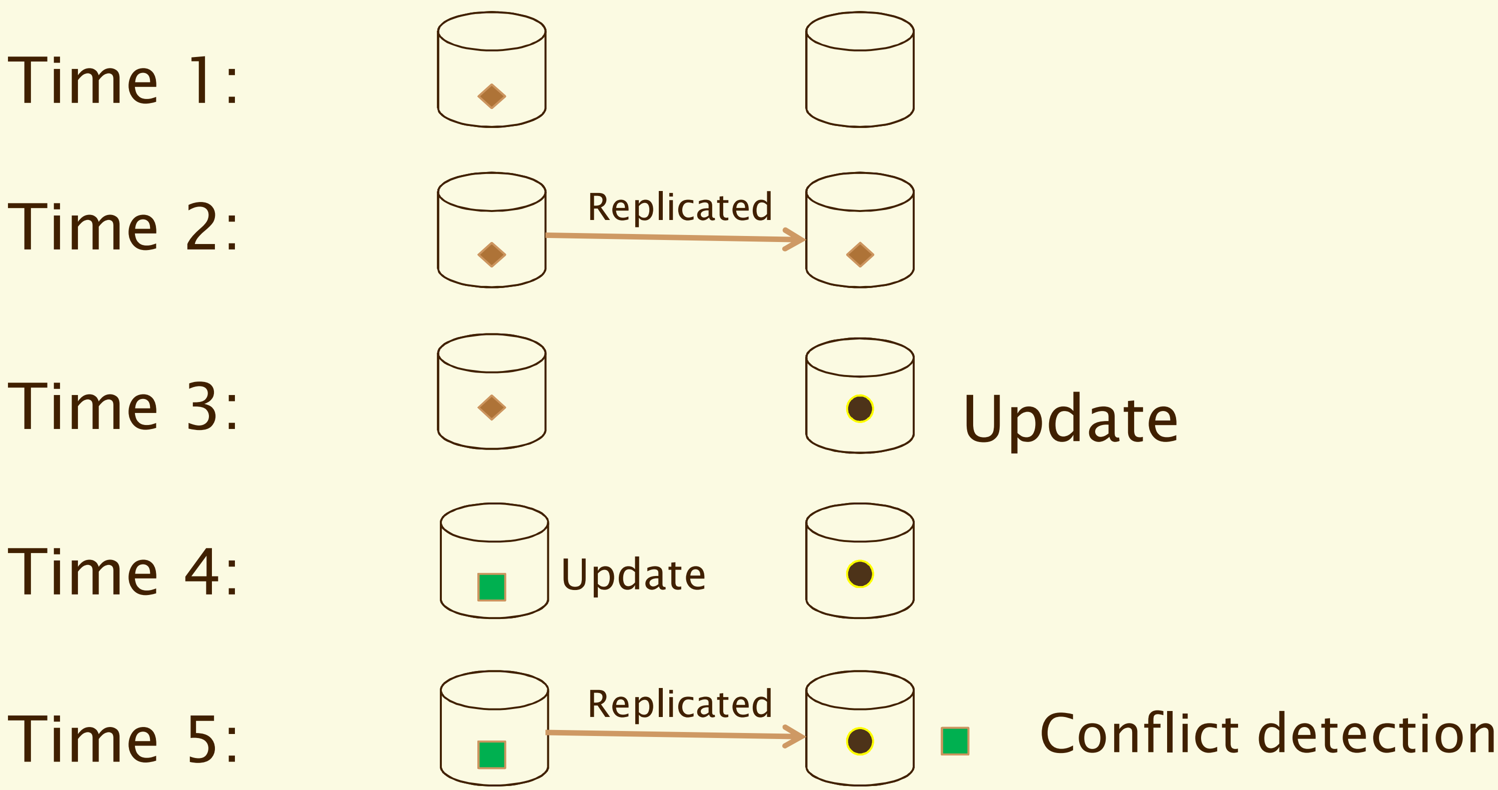
Replication

- Each data item is replicated at N hosts.
- "preference list": The list of nodes that is responsible for storing a particular key.



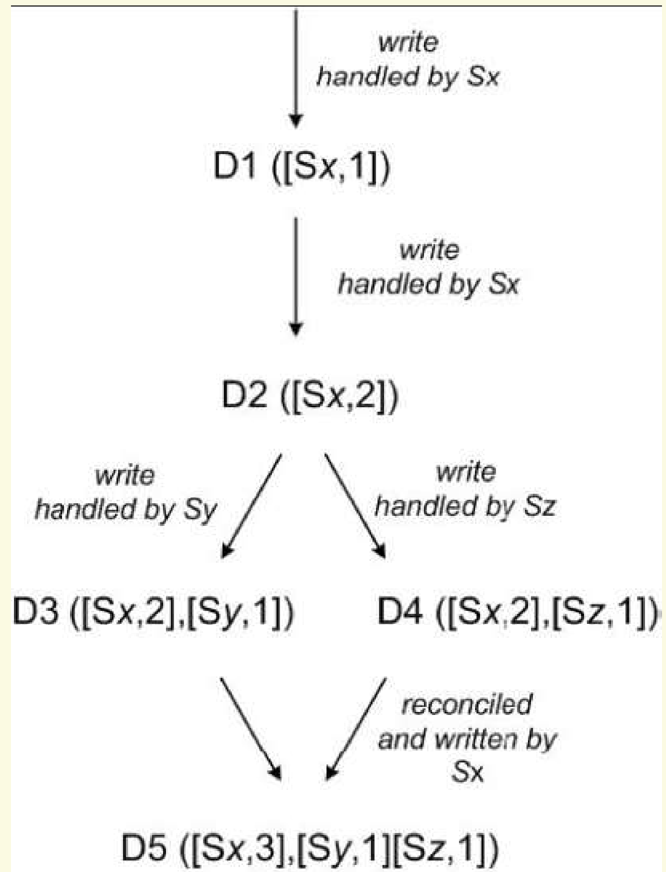
Vector Clocks

- ✓ Used for conflict **detection** of data.
- ✓ Timestamp based resolution of conflicts is not enough.



Vector Clock

- A vector clock is a list of (node, counter) pairs.
- Every version of every object is associated with one vector clock.
- *If the counters on the first object's clock are less-than-or-equal to all of the nodes in the second clock, then the first is an ancestor of the second and can be forgotten.*



Summary of techniques used in Dynamo and their advantages

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.