CSE6331: Cloud Computing

Leonidas Fegaras

University of Texas at Arlington

Data Storage

BigTable: A Distributed Storage System for Structured Data

Based on a presentation by

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber @ Google

Introduction

- BigTable is a distributed storage system for managing structured data.
- Designed to scale to a very large size
  - Petabytes of data across thousands of servers
- Used for many Google projects
  - Web indexing, Personalized Search, Google Earth, Google Analytics, Google Finance, …
- Flexible, high-performance solution for all of Google's products

Motivation

- Lots of (semi-)structured data at Google
  - URLs:
    - Contents, crawl metadata, links, anchors, pagerank, …
  - Per-user data:
    - User preference settings, recent queries/search results, …
  - Geographic locations:
    - Physical entities (shops, restaurants, etc.), roads, satellite image data, user annotations, …
- Scale is large
  - Billions of URLs, many versions/page (~20K/version)
  - Hundreds of millions of users, thousands or queries/sec
  - 100TB+ of satellite image data

Why not just use commercial DB?

- Scale is too large for most commercial databases
- Even if it weren't, cost would be very high
  - Building internally means system can be applied across many projects for low incremental cost
- Low-level storage optimizations help performance significantly
  - Much harder to do when running on top of a database layer

Goals

- Want asynchronous processes to be continuously updating different pieces of data
  - Want access to most current data at any time
- Need to support:
  - Very high read/write rates (millions of ops per second)
  - Efficient scans over all or interesting subsets of data
  - Efficient joins of large one-to-one and one-to-many datasets
- Often want to examine data changes over time
  - e.g. contents of a web page over multiple crawls

BigTable

- Distributed multi-level map
- Fault-tolerant, persistent
- Scalable
  - Thousands of servers
  - Terabytes of in-memory data
  - Petabyte of disk-based data
  - Millions of reads/writes per second, efficient scans
- Self-managing
  - Servers can be added/removed dynamically
  - Servers adjust to load imbalance

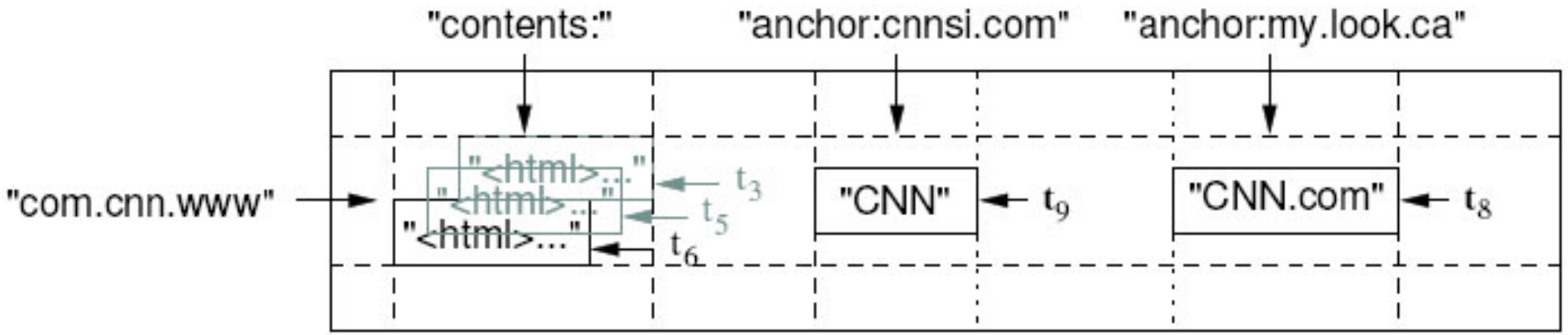
Building Blocks

- Building blocks:
  - Google File System (GFS): Raw storage
  - Scheduler: schedules jobs onto machines
  - Lock service: distributed lock manager
  - MapReduce: simplified large-scale data processing
- BigTable uses of building blocks:
  - GFS: stores persistent data (SSTable file format for storage of data)
  - Scheduler: schedules jobs involved in BigTable serving
  - Lock service: master election, location bootstrapping
  - Map Reduce: often used to read/write BigTable data

Basic Data Model

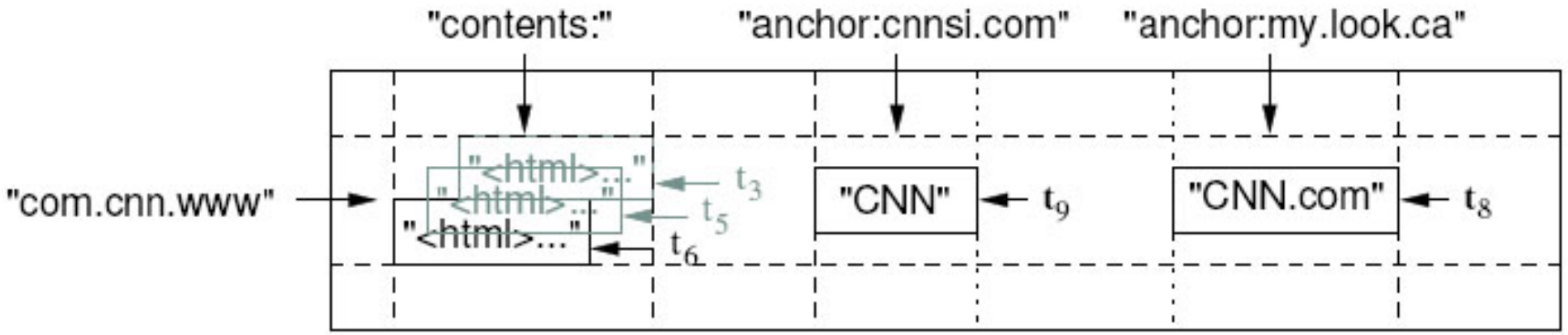
- A BigTable is a sparse, distributed, persistent, multi-dimensional sorted map

  (row, column, timestamp) -> cell contents



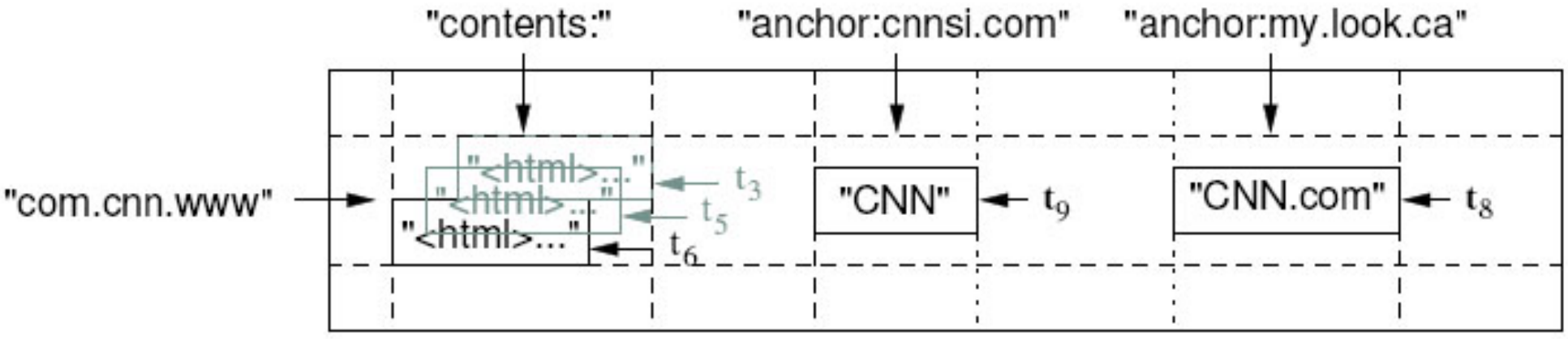
- Three dimensions: row: string, column: string, time: int64
- Good match for most Google applications

WebTable Example



- Want to keep copy of a large collection of web pages and related information
- Use URLs as row keys
- Various aspects of web page as column names
- Store contents of web pages in the `contents:` column under the timestamps when they were fetched.

Rows



- Name is an arbitrary string
  - Access to data in a row is atomic
  - Row creation is implicit upon storing data
- Rows with consecutive keys are grouped into tablets
  - Tablets are the unit of distribution and load balancing
- Rows ordered lexicographically
  - Rows close together lexicographically usually on one or a small number of machines

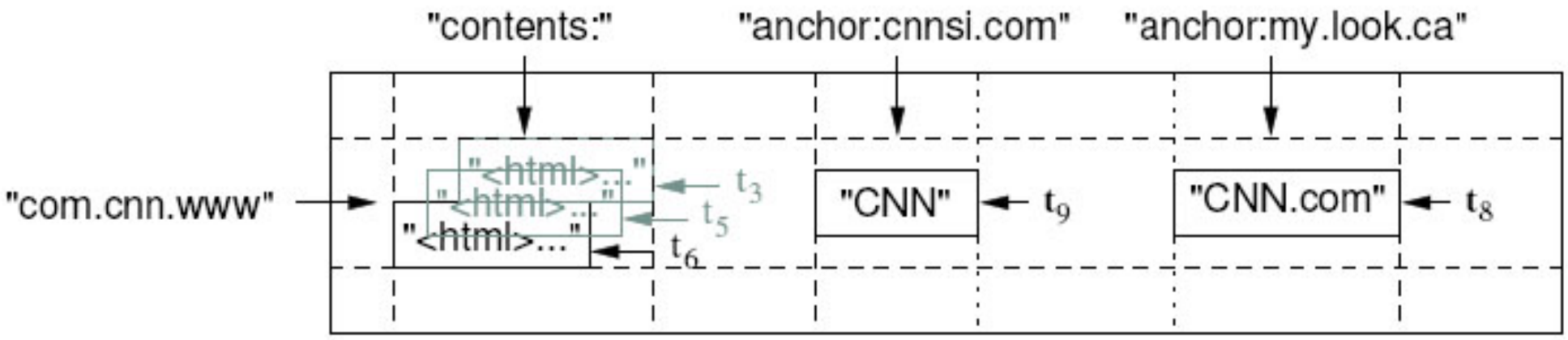
Rows (cont.)

Reads of short row ranges are efficient and typically require communication with a small number of machines.

- Can exploit this property by selecting row keys so they get good locality for data access.
- Example:

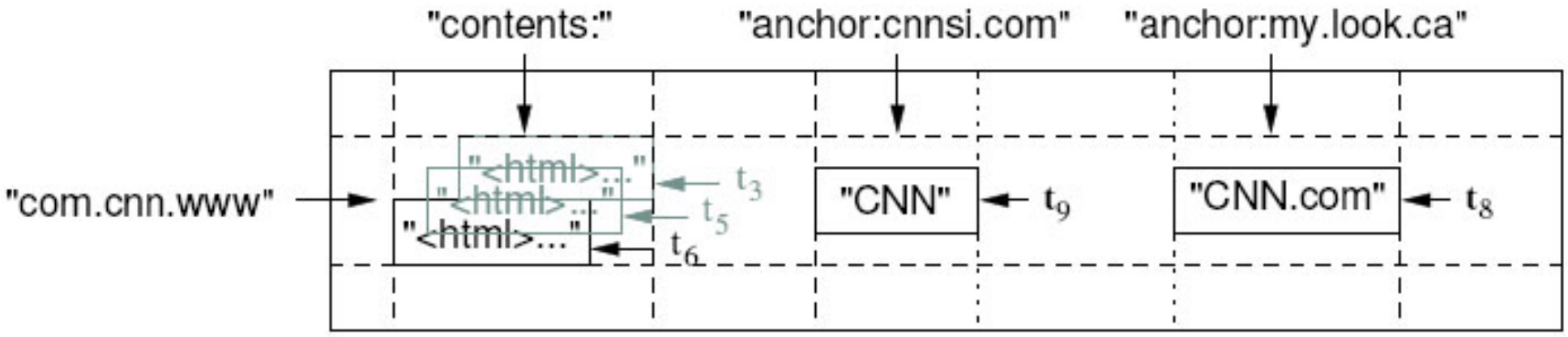
```
math.gatech.edu, math.uga.edu, phys.gatech.edu, phys.uga.edu
VS
edu.gatech.math, edu.gatech.phys, edu.uga.math, edu.uga.phys
```

Columns



- Columns have two-level name structure:
  - family:optional_qualifier
- Column family
  - Unit of access control
  - Has associated type information
  - Must be created explicitly
- Qualifier gives unbounded columns
  - Additional levels of indexing, if desired

Timestamps



- Used to store different versions of data in a cell
  - New writes default to current time, but timestamps for writes can also be set explicitly by clients
- Lookup options:
  - *"Return most recent K values"*
  - *"Return all values in timestamp range (or all values)"*
- Column families can be marked w/ attributes:
  - *"Only retain most recent K values in a cell"*
  - *"Keep values until they are older than K seconds"*

Implementation – Three Major Components

- Library linked into every client
- One master server
  - Responsible for:
    - Assigning tablets to tablet servers
    - Detecting addition and expiration of tablet servers
    - Balancing tablet-server load
    - Garbage collection
- Many tablet servers
  - Tablet servers handle read and write requests to its table
  - Splits tablets that have grown too large

Implementation (cont.)

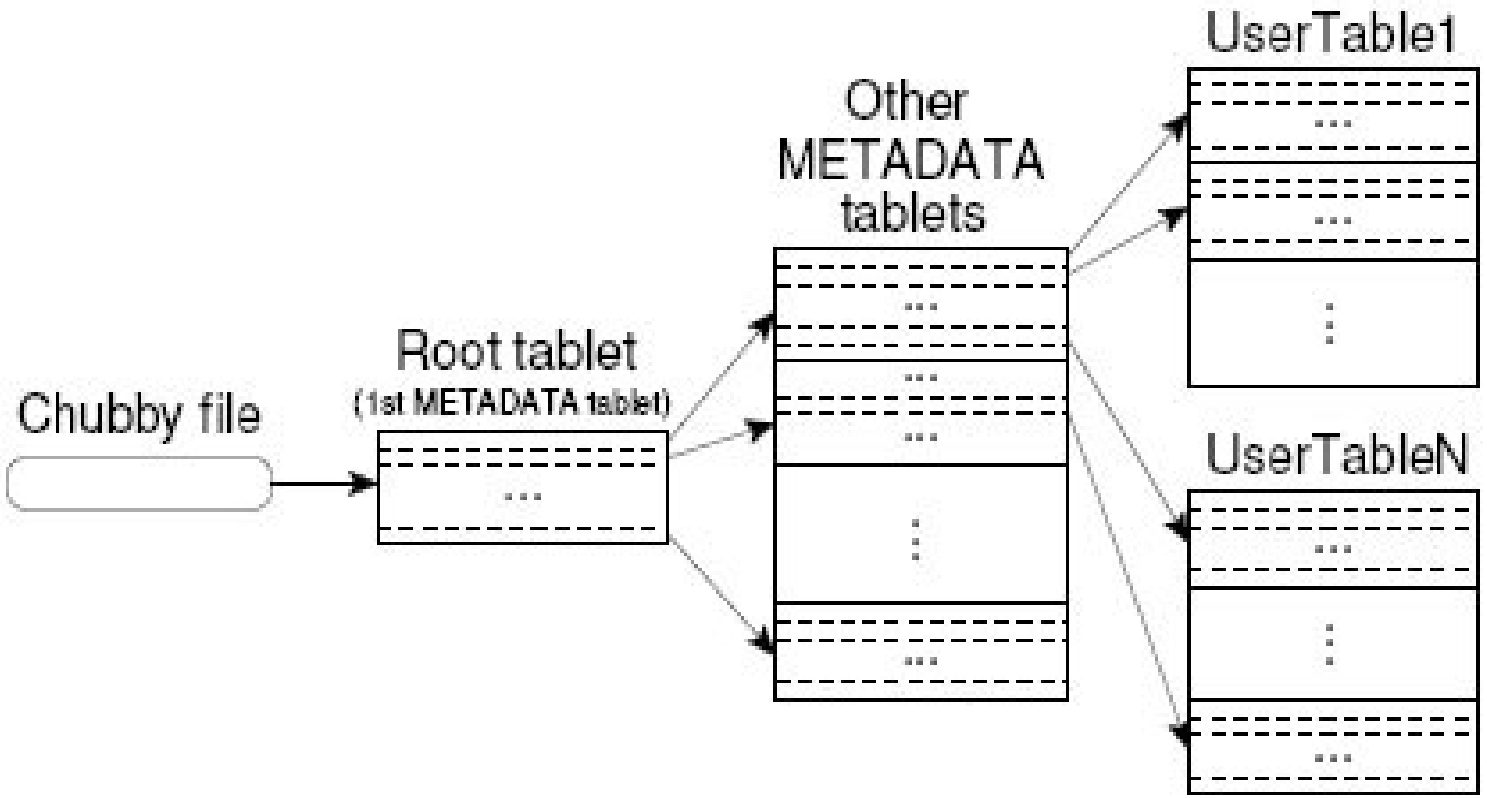
- Client data doesn't move through master server. Clients communicate directly with tablet servers for reads and writes.
- Most clients never communicate with the master server, leaving it lightly loaded in practice.

Tablets

- Large tables broken into tablets at row boundaries
  - Tablet holds contiguous range of rows
    - Clients can often choose row keys to achieve locality
  - Aim for ~100MB to 200MB of data per tablet
- Serving machine responsible for ~100 tablets
  - Fast recovery:
    - 100 machines each pick up 1 tablet for failed machine
  - Fine-grained load balancing:
    - Migrate tablets away from overloaded machine
    - Master makes load-balancing decisions

Tablet Location

- Since tablets move around from server to server, given a row, how do clients find the right machine?
  - Need to find tablet whose row range covers the target row



Tablet Assignment

- Each tablet is assigned to one tablet server at a time.
- Master server keeps track of the set of live tablet servers and current assignments of tablets to servers. Also keeps track of unassigned tablets.
- When a tablet is unassigned, master assigns the tablet to an tablet server with sufficient room.

API

- Metadata operations
  - Create/delete tables, column families, change metadata
- Writes (atomic)
  - Set(): write cells in a row
  - DeleteCells(): delete cells in a row
  - DeleteRow(): delete all cells in a row
- Reads
  - Scanner: read arbitrary cells in a bigtable
    - Each row read is atomic
    - Can restrict returned rows to a particular range
    - Can ask for just data from 1 row, all rows, etc.
    - Can ask for all columns, just certain column families, or specific columns

Updating

```
// Open the table
Table *T = OpenOrDie("/bigtable/web/webtable");

// Write a new anchor and delete an old anchor
RowMutation r1(T, "com.cnn.www");
r1.Set("anchor:www.c-span.org", "CNN");
r1.Delete("anchor:www.abc.com");
Operation op;
Apply(&op, &r1);
```

Scanning

```
Scanner scanner(T);
ScanStream *stream;
stream = scanner.FetchColumnFamily("anchor");
stream->SetReturnAllVersions();
scanner.Lookup("com.cnn.www");
for (; !stream->Done(); stream->Next()) {
  printf("%s %s %lld %s\n",
         scanner.RowName(),
         stream->ColumnName(),
         stream->MicroTimestamp(),
         stream->Value());
}
```

Amazon Dynamo

Based on “Data Models; Key/Value: Amazon Dynamo,” by Kristin Tufte at Portland State University

Motivation

- Lessons learned at Amazon
  - lack of reliability and scalability has significant financial consequences
  - reliability and scalability depend on how application state is managed
  - key/value data model is sufficient for many applications: bestseller lists, shopping carts, customer preferences, session management, etc.
- RDBMS are not an ideal solution
  - most features are not used
  - scales up, not out
  - availability limitations due to transactional processing
- Consistency vs. availability
  - high availability is very important
  - user-perceived consistency is very important
  - trade off strong consistency in favor of higher availability

System Assumptions and Requirements

- Query Model
  - simple read and write operations only
  - small objects (BLOB) are uniquely identified by their key
  - no operations span multiple data items
- ACID Properties
  - trade off weaker consistency for higher availability
  - no isolation guarantees and single key updates only
- Efficiency
  - latency requirements measured at the 99.9^{th} percentile of distribution
  - configurable to consistently achieving required latency and throughput
  - trade off performance, cost efficiency, availability, and durability

System Assumptions and Requirements

- Scalability
  - each service (application) uses a distinct Dynamo instance

    What does this say about consistency across instances?
  - requires scale up to hundreds of nodes, not thousands of nodes
- Security
  - operation environment is assumed to be non-hostile
  - no security requirements (e.g., authentication and authorization)

Amazon's Platform Architecture

- Decentralized, loosely-coupled, service-oriented architecture
  - **page rendering components** generate dynamic web content and query many other services
  - (stateless) **aggregator services** use other services to produce composite response
  - **stateful services** own and manage their own state using different data stores, which are only accessible within its service boundaries
- Availability is paramount
- Large scale (and growing)

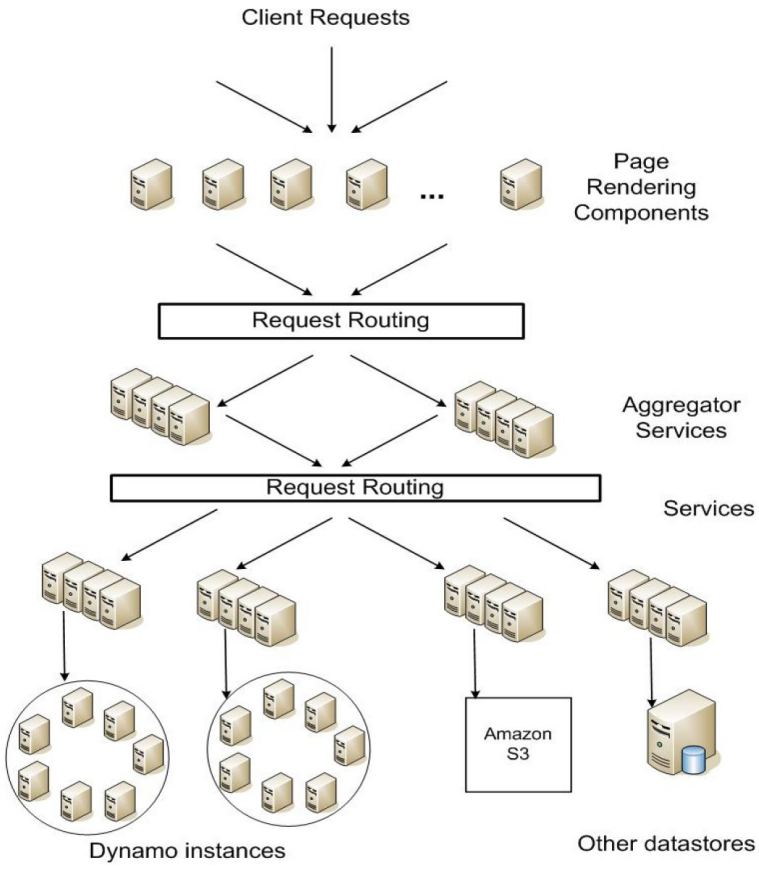


Figure Credit: G. DeCandia et al.: Dynamo: Amazon's Highly Available Key-Value Store, Proc. SOSP, 2007.

Service Level Agreements

- Service Level Agreement (SLA)
  - formal contract between service and client
  - agreement on several system-related characteristics
  - *"This service guarantees to provide a response within 300ms for 99.9% of its requests for a peak client load of 500 requests per second."*
- SLA play an important role in Amazon's decentralized loosely-coupled service-oriented infrastructure
  - up to 150 services need to be contacted to process a typical request on one of Amazon's e-commerce sites
  - services can have dependencies to other services leading to a call graph with more than one level
  - all services within the call chain must obey their (tighter) contracts to ensure that the overall system can maintain a clear bound on operation

Design Considerations

- Replication for high availability and durability
  - replication technique: synchronous or asynchronous?
  - conflict resolution: when and who?
- Dynamo's goal is to be "always writable"
  - rejecting writes may result in poor Amazon customer experience
  - data store needs to be highly available for writes, e.g., accept writes during failures and allow write conversations without prior context
- Design choices
  - optimistic (asynchronous) replication for non-blocking writes
  - conflict resolution on read operation for high write throughput
  - conflict resolution by client for user-perceived consistency

Key Design Principles

- Incremental Scalability
  - scale out (and in) one node at a time
  - minimal impact on both operators of the systems and the system itself
- Symmetry
  - every node should have the same set of responsibilities
  - no distinguished nodes or nodes that take on a special role
  - symmetry simplifies system provisioning and maintenance
- Decentralization
  - favor decentralized peer-to-peer techniques
  - achieve a simpler, more scalable, and more available system
- Heterogeneity
  - exploit heterogeneity in the underlying infrastructure
  - load distribution proportional to capabilities of individual servers
  - adding new nodes with higher capacity without upgrading all nodes

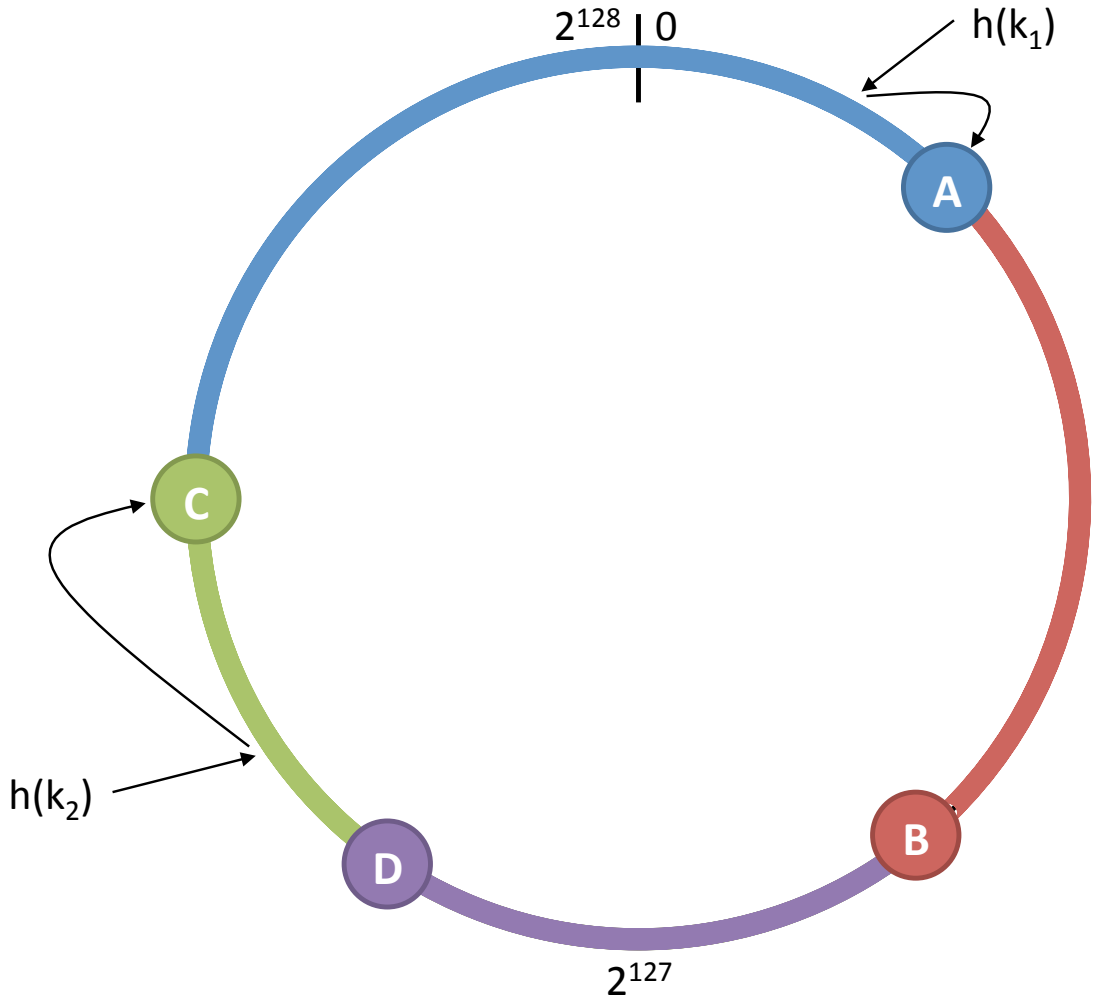
System Interface

- Dynamo stores objects identified by a key *k* through a simple interface that exposes two operations
- Get operation
  - get(*k*) → [object(s), context]
  - locates object replicas associated with the key *k* in the storage system
  - returns a single object or a list of objects with conflicting versions along with a context
- Put operation
  - put(*k*, context, object)
  - determines where the replicas of the object should be placed based on its key *k* and writes the replicas to disk
- Context
  - encodes system metadata about the object that is opaque to the caller
  - includes information such as the version of the object (vector clock)

Partitioning

- Partitioning based on **consistent hashing**
  - output range of hash function treated as a fixed circular space or **ring**, i.e., the largest hash value wraps around to the smallest hash value
  - each node in the system is assigned a random value within this space, which represents its **position** on the ring
  - each data item identified by a key *k* is assigned to a node by
    1. hashing the data item's key to find its position on the ring
    2. walking the ring clockwise to the first node with a position larger than the data item's position
- Each node is responsible for the region in the ring between itself and its predecessor node on the ring
- Arrival and departure of nodes only affects its immediate neighbors and other nodes remain unaffected

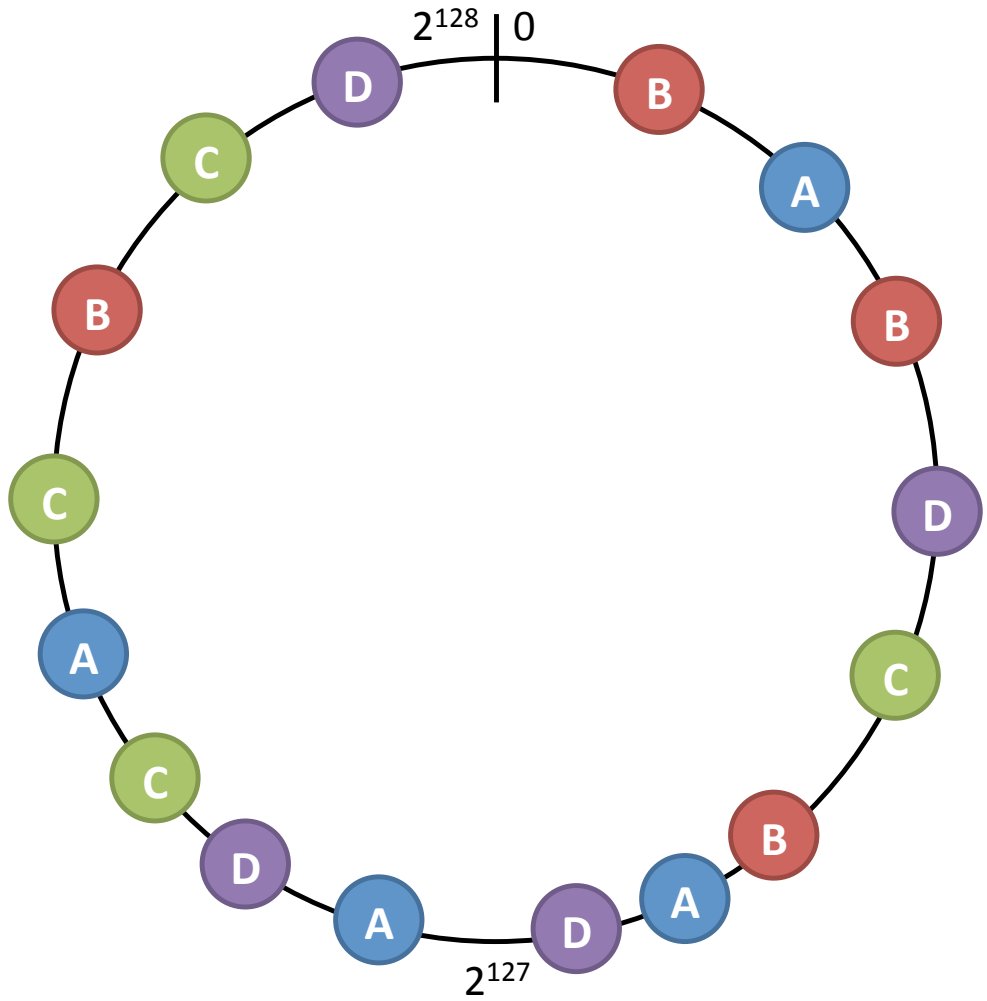
Partitioning



Load Balancing

- Problems with basic partitioning algorithm
  - assigning nodes a random position on the node can lead to non-uniform data and load distribution
  - algorithm does not consider heterogeneity in the performance of nodes
- Virtual nodes
  - each **physical node** gets assigned multiple positions (tokens) in the ring
  - a **virtual node** behaves like a single node in the system
- Advantages
  - if a node becomes **unavailable**, its load can be evenly distributed onto the remaining available nodes
  - if a node becomes **available**, it gets a roughly equivalent share of the load from each of the other available nodes
  - number of virtual nodes that a physical node is responsible for can be decided based on its capacity, accounting for heterogeneity in the physical infrastructure

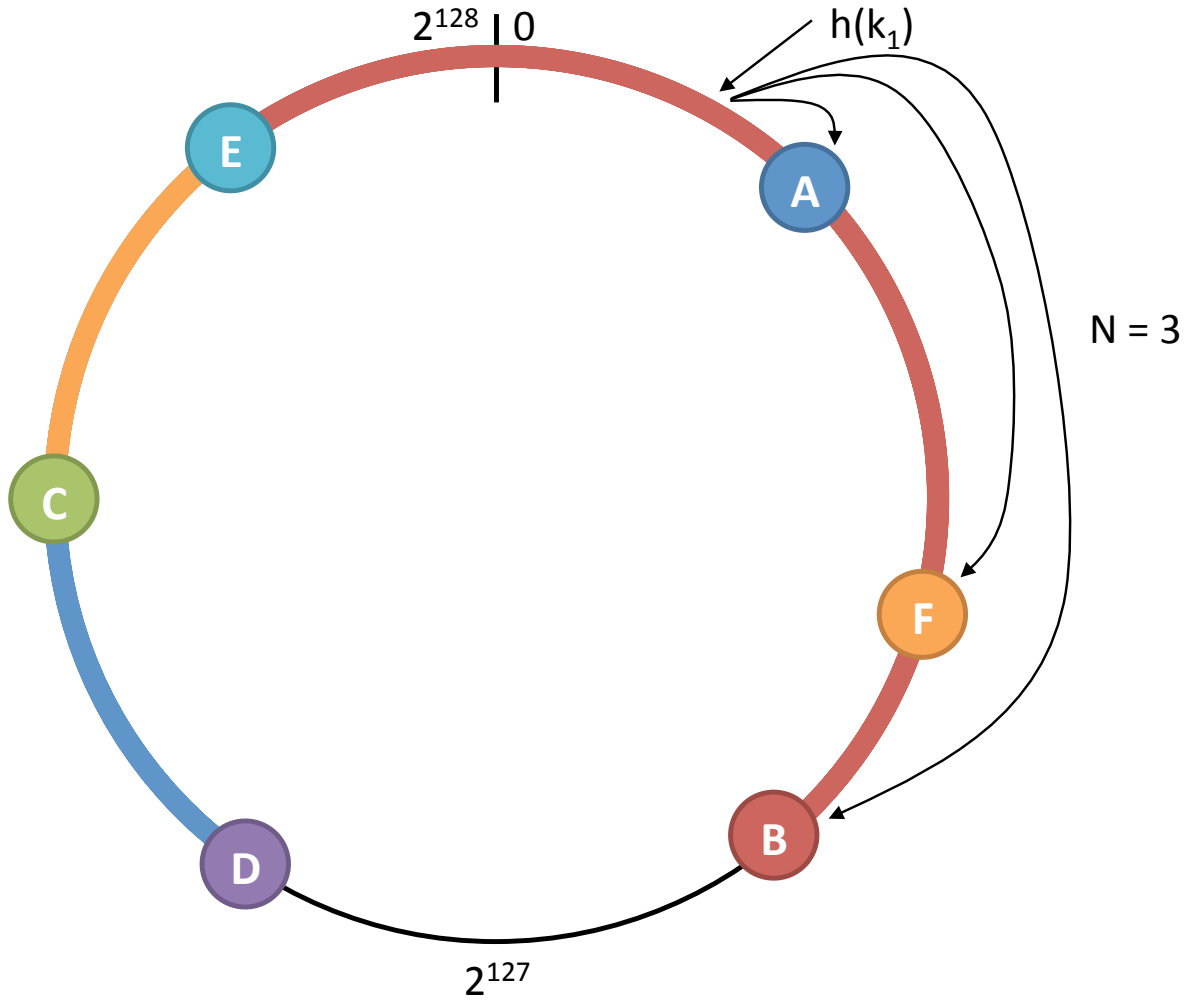
Load Balancing



Replication

- Replication is used to achieve high availability and durability
- Every data item is replicated at *N* hosts
  - *N* is configured "per-instance" of Dynamo
  - each key *k* is assigned a **coordinator** host that handles write requests for *k*
  - coordinator is also in charge of replication of data items within its range
- Algorithm
  1. coordinator stores data item with key *k* locally
  2. coordinator stores data item at *N*-1 clockwise successors nodes
- Every node is responsible for the region of the ring between itself and its N^{th} predecessor
- Preference list
  - enumerates nodes that are responsible for storing a key *k*
  - contains more than *N* nodes to account for node failures

Replication



Data Versioning

- Eventual consistency by asynchronous updates of replicas
  - put() may return to caller before all replicas have been updated
  - subsequent get() may return objects that have not been updated yet
- "Always writable" design
  - result of each modification is a new and immutable version of the data
  - multiple versions may be exist in the system at the same time
  - vector clocks capture causality between different versions of an object
- Version reconciliation
  - **system-based**: most versions simply subsume the previous version
  - **client-based**: if failures combined with concurrent updates lead to branches, the client needs to collapse multiple branches into one
- Context passed between get and put operations contains vector clock information

Data Versioning Evolution

1. Client *A*
   - writes new object D
   - → node *Sx* writes version D1
2. Client *A*
   - updates object D
   - → node *Sx* writes version D2
3. Client *A*
   - updates object D
   - → node *Sy* writes version D3
4. Client *B*
   - reads and updates object D
   - → node *Sz* writes version D4
5. Client *C*
   - reads object D (i.e., D3 and D4)
   - client performs reconciliation
   - → node *Sx* write version D5

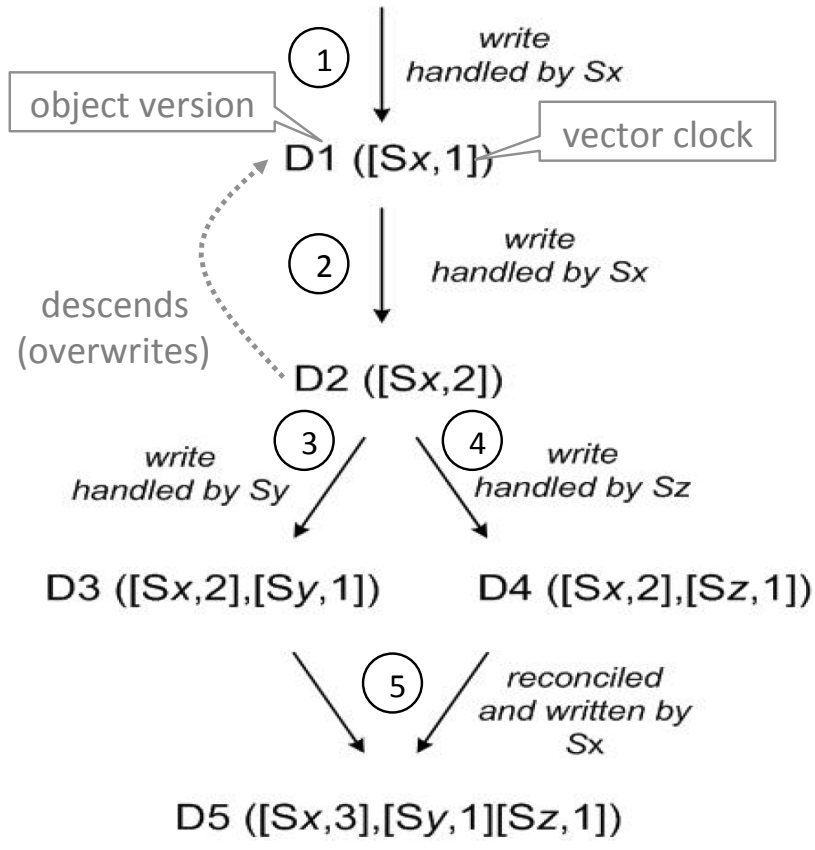


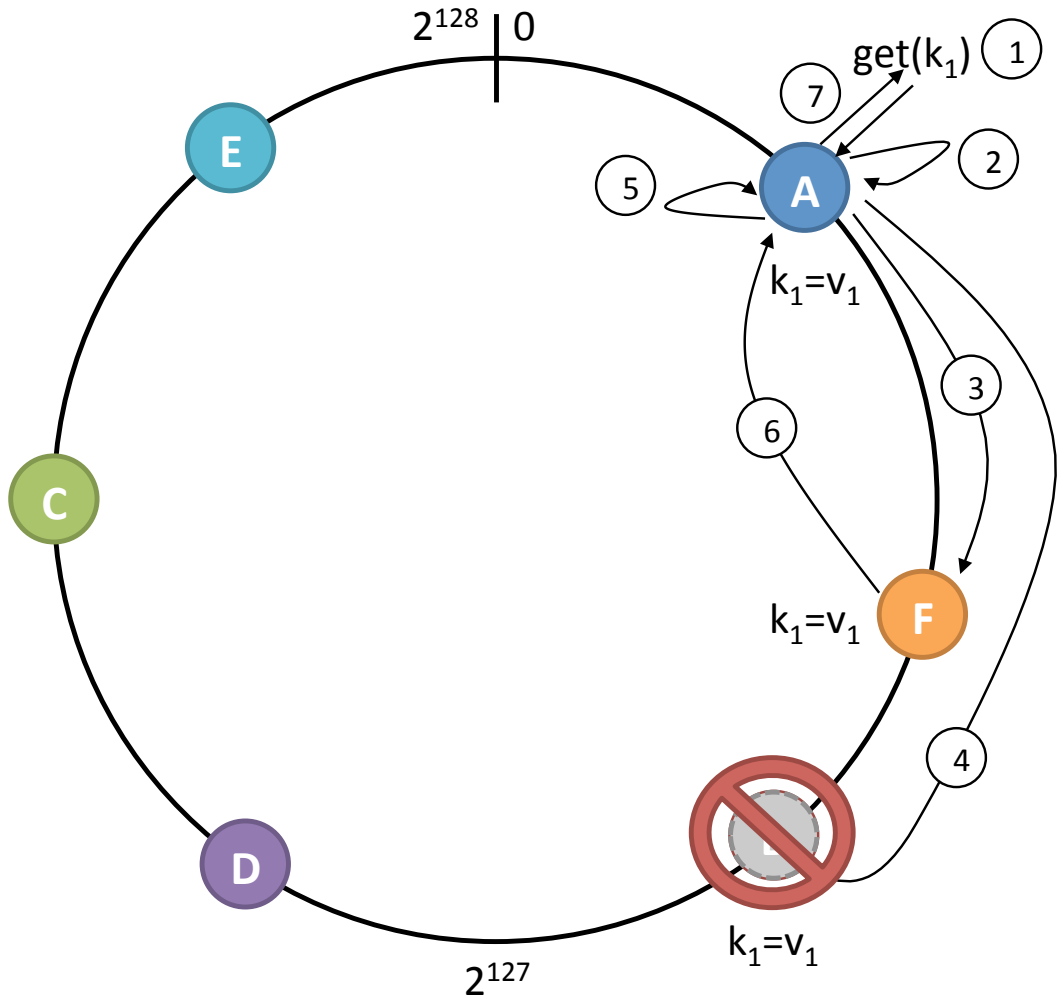
Figure Credit: G. DeCandia et al.: Dynamo: Amazon's Highly Available Key-Value Store, Proc. SOSP, 2007.

Execution of Get and Put Operations

- **Coordinator**: node that handles read or write operation
  - typically the first of the top *N* nodes of the preference list for a write
  - requests that hit a node that is not in the top *N* of the requested key's preference list are forwarded to the appropriate coordinator
- Quorum-like Protocol
  - R: minimum number of nodes that must participate in successful read
  - W: minimum number of nodes that must participate in successful write
- "Sloppy Quorum"
  - read and write operations are performed on the first *N* **healthy** nodes
  - not guaranteed be the first *N* nodes encountered while walking the ring
- Hinted Handoff
  - if a node is unreachable, a **hinted** replica is sent to next healthy node
  - nodes receive hinted replicas keep them in a separate database
  - hinted replica is delivered to original node when it recovers

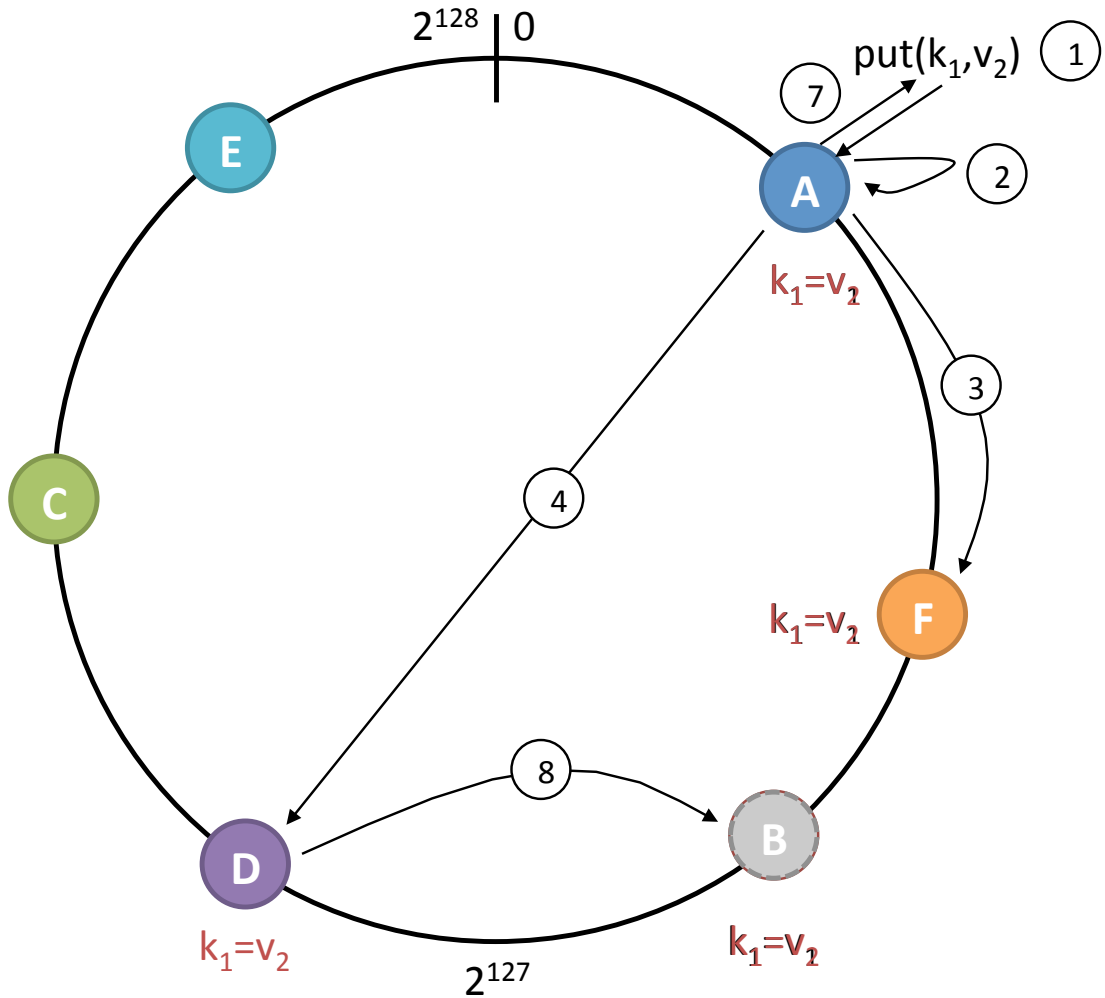
Execution of Get Operation

N = 3
R = 2
W = 2



Execution of Put Operation

N = 3
R = 2
W = 2

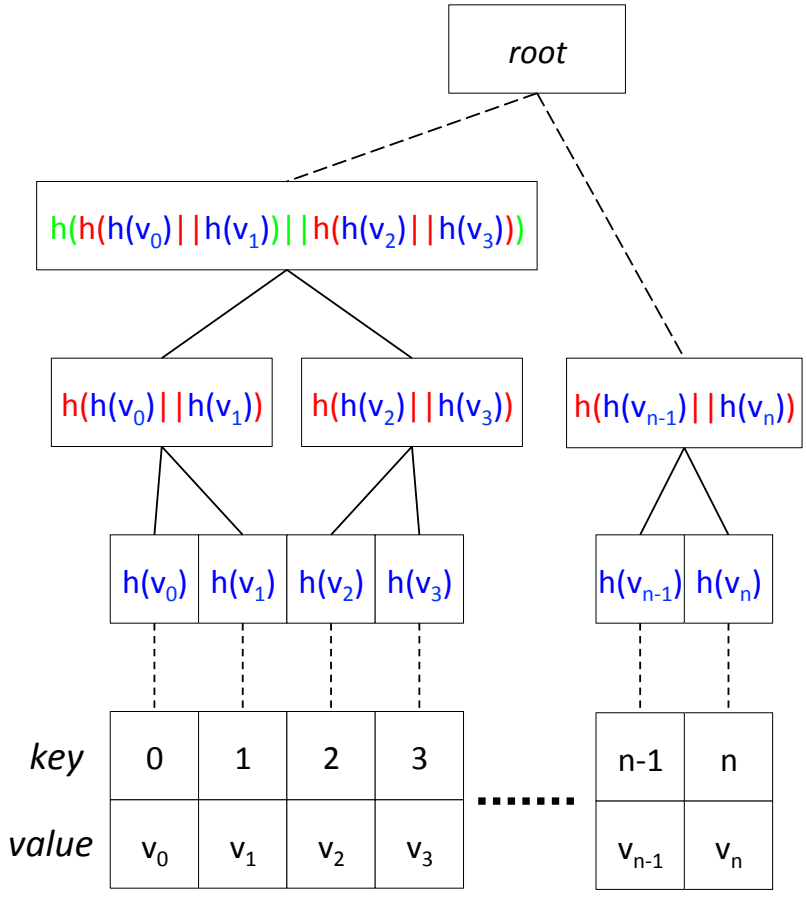


Replica Synchronization

- Scenarios exist where hinted replicas become unavailable before being returned to the original node
  - Dynamo implements an anti-entropy (replica synchronization) protocol
  - Dynamo uses Merkle trees to detect inconsistencies between replicas
- Algorithm
  - every physical node maintains a separate Merkle tree for each hosted key range, i.e., the set of keys covered by a virtual host
  - to check if their key ranges are up-to-date, two nodes exchange the roots of the Merkle tree of the key ranges they have in common
  - if the value of the roots are equal, the key ranges are up-to-date
  - if not, they recursively exchange the values of the children until they reach the leaves of the Merkle tree
  - at that point, the inconsistent keys are identified and the corresponding values can be exchanged

Merkle Tree

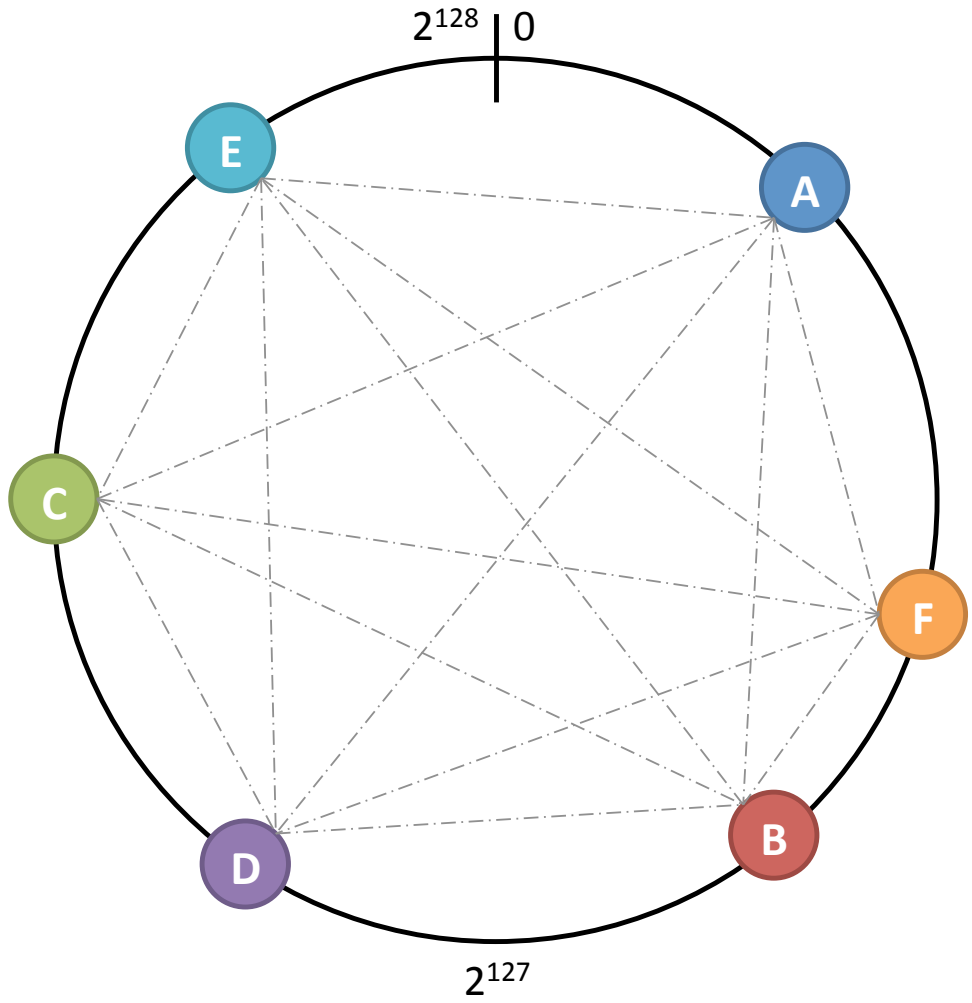
- Hash tree
  - **leaves** are hashes of the values of individual keys
  - **non-leaves** are hashes of their respective children nodes
- Properties
  - efficient verification of large data structures
  - every branch can be processed independently
- Applications
  - ZFS file systems
  - Git revision control system
  - NoSQL systems (e.g., Dynamo, Cassandra, and Riak)



Membership and Failure Detection

- Explicit mechanism to initiate addition and removal of nodes
  - outage or failure rarely signifies permanent departure of node
  - manual error could result in unintentional startup of node
  - these events should not result in rebalancing of partition assignment or repair of unreachable replica
- Full membership model
  - administrative command issued to join/remove a node to/from ring
  - node that handles request updates its persistent membership table
  - gossip-based protocol propagates membership changes
- Failure detection
  - communication failures avoided based on a purely local failure notion
  - globally consistent view on failure state is not required thanks to explicit join/remove commands

Full Membership Model



Implementation

Three main software components

1. Local persistence engine
   - plug-in architecture supports different storage engines
   - storage engine chosen based on application's object size distribution
   - *BerkeleyDB*: objects in the order of tens of kilobytes
   - *MySQL*: larger objects
2. Request coordination
   - built on top of an event-driven messaging infrastructure
   - communication implemented using Java NIO channels
   - client requests create a state machine on node that received request
   - each state machine handles exactly one client request
3. Membership and failure detection

Configurability

N	R	W	Application
3	2	2	Consistent, durable, interactive user state (typical configuration)
n	1	*n*	High-performance read engine
1	1	1	Distributed web cache

Figure Credit: Peter Vosshall, Amazon.com, 2007.

Typical Replication Patterns

- Business logic-specific reconciliation
  - each data object is replicated across multiple nodes
  - client applications performs reconciliation in case of divergent versions
  - **example**: merging different versions of a customer's shopping cart
- Timestamp-based reconciliation
  - Dynamo performs simple timestamp-based reconciliation
  - "last write wins", i.e., object with largest physical times is selected
  - **example**: maintaining a customer's session information
- High-performance read engine
  - R = 1, W = N
  - high read request rate with only a small number of updates
  - **example**: management of product catalog and promotional items