Dynamo: Amazon's Highly-Available Key-Value Store

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels

Amazon.com

Presented by Jeff Avery

CS 848: Modern Database Systems

January 27, 2015

Outline

- Background
- System Requirements
- Design Principles
- System Architecture
- Implementation & Lessons
- Conclusions

Werner Vogels
CTO – Amazon.com

Outline

- Background
  - Motivation
  - Challenges at Scale
- System Requirements
- Design Principles
- System Architecture
- Implementation & Lessons
- Conclusions

Motivation

"**Reliability at massive scale is one of the biggest challenges we face** at Amazon.com, one of the largest e-commerce operations in the world; even the slightest outage has significant financial consequences and impacts customer trust. "

"**Customers should be able to view and add items to their shopping cart even if disks are failing, network routes are flapping, or data centers are being destroyed by tornados.**"

"There are always a small but significant number of server and network components that are failing at any given time. As such Amazon's software systems need to be constructed in a manner that treats **failure handling as the normal case without impacting availability or performance**."

- DeCandia, pg. 205

Challenges at Scale

"Amazon.com started 10 years ago as a monolithic application, running on a Web server, talking to a database on the back end. This application, dubbed Obidos, evolved to hold all the business logic, all the display logic, and all the functionality… **scaling efforts at Amazon were focused on making the back-end databases scale to hold more items, more customers, more orders, and to support multiple international sites. This went on until 2001 when it became clear that the front-end application couldn't scale anymore.**"

- ACM interview with Werner Vogels

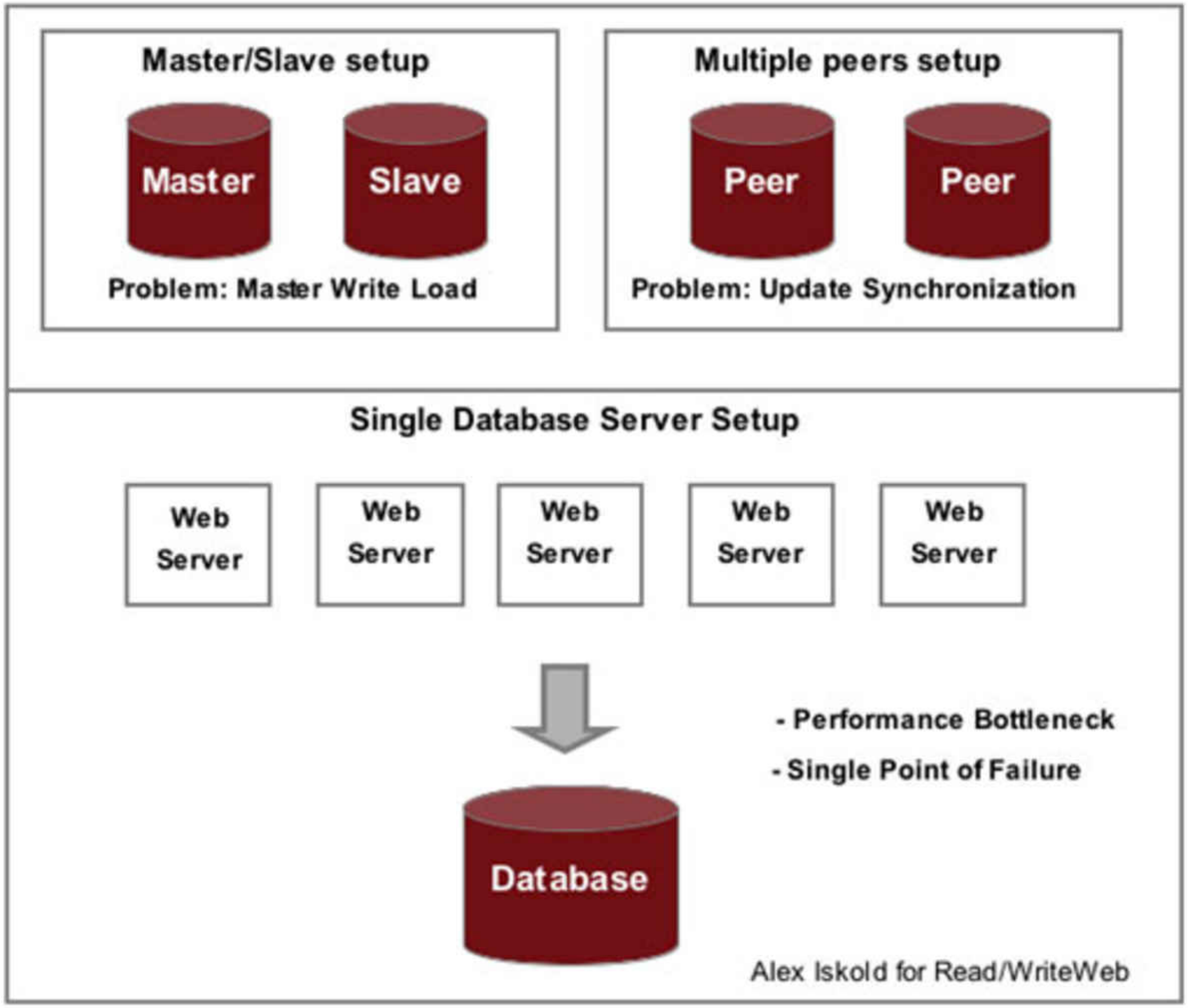
"**RDBMS replication** technologies are limited and typically **choose consistency over availability.** "

- DeCandia, pg. 207.

RDBMS Replication

"It is difficult to create redundancy and parallelism with relational databases, so they become a single point of failure. **In particular, replication is not trivial**."

http://readwrite.com/2007/10/30/amazon_dynamo



Outline

- Background
- System Requirements
  - Approach
  - System Requirements
  - Service-Level Agreements
- Design Principles
- System Architecture
- Implementation & Lessons
- Conclusions

Approach

Challenge

- Designing a highly-available system that can scale to millions of users, while meeting service-level SLA.

Problem

- **Traditional systems** perform synchronous replica coordination in order to **provide strongly consistent data, at the cost of availability**.
- Network and hardware failures mean that **strong consistency and high data availability cannot be achieved simultaneously**.

Solution

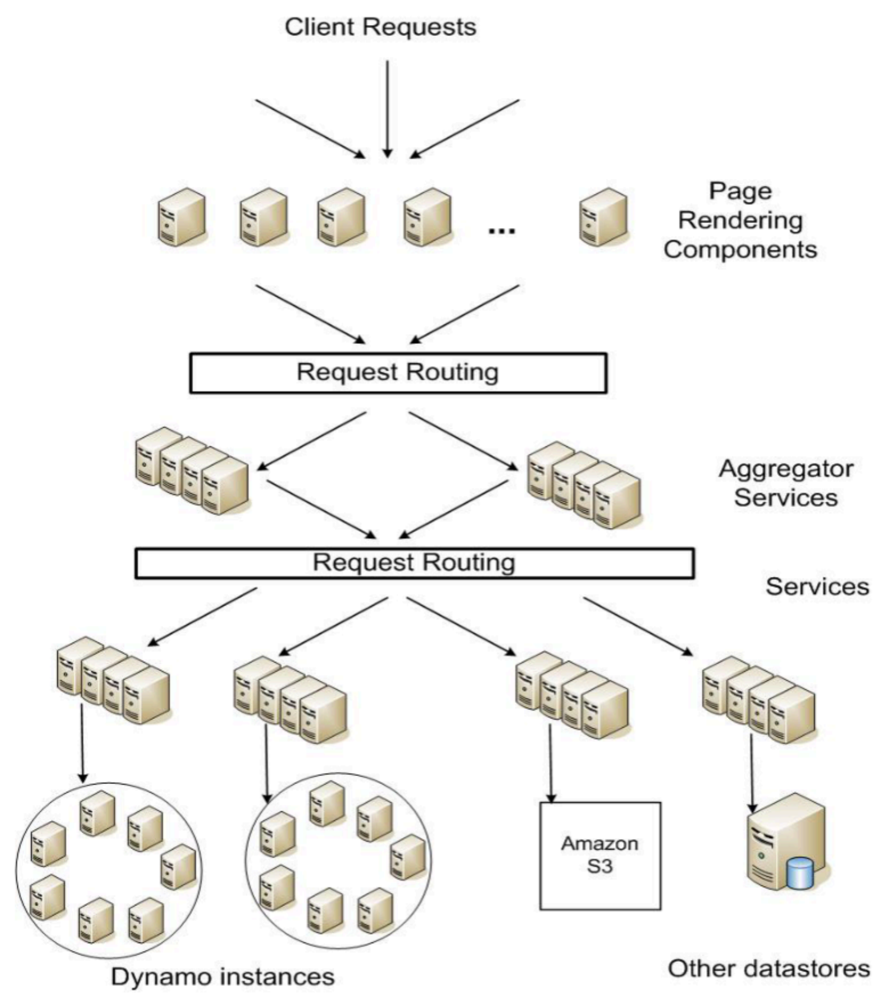
- **Sacrifice strong consistency for high availability**.
- Give users the "ability to trade-off cost, consistency, durability and performance, while maintaining high-availability."

System Requirements

Property	**Definition**	**Rationale**
ACID	ACID (Atomicity, Consistency, Isolation, Durability) is a set of properties that guarantee that database transactions are processed reliably.	Dynamo targets applications that operate with weaker consistency (the "C" in ACID) if this results in high availability.
Query Model	Simple read and write operations to a data item that is uniquely identified by a key.	No operations span multiple data items and there is no need for relational schema.
Efficiency	The system needs to function on a commodity hardware infrastructure.	Services must be able to configure Dynamo to consistently achieve latency and throughput requirements.
Facing	Dynamo is used only by Amazon's internal services.	No security related requirements

Service Level Agreements (SLA)

- Contract where a client and a service agree on system-related characteristics. Promises **bounded time for a response.**
- Every dependency in the platform needs to deliver with even *tighter* bounds.
- At Amazon, **SLAs** are expressed and measured at the **99.9th percentile of the distribution** (i.e. the edge-cases can represent critical customers).



Outline

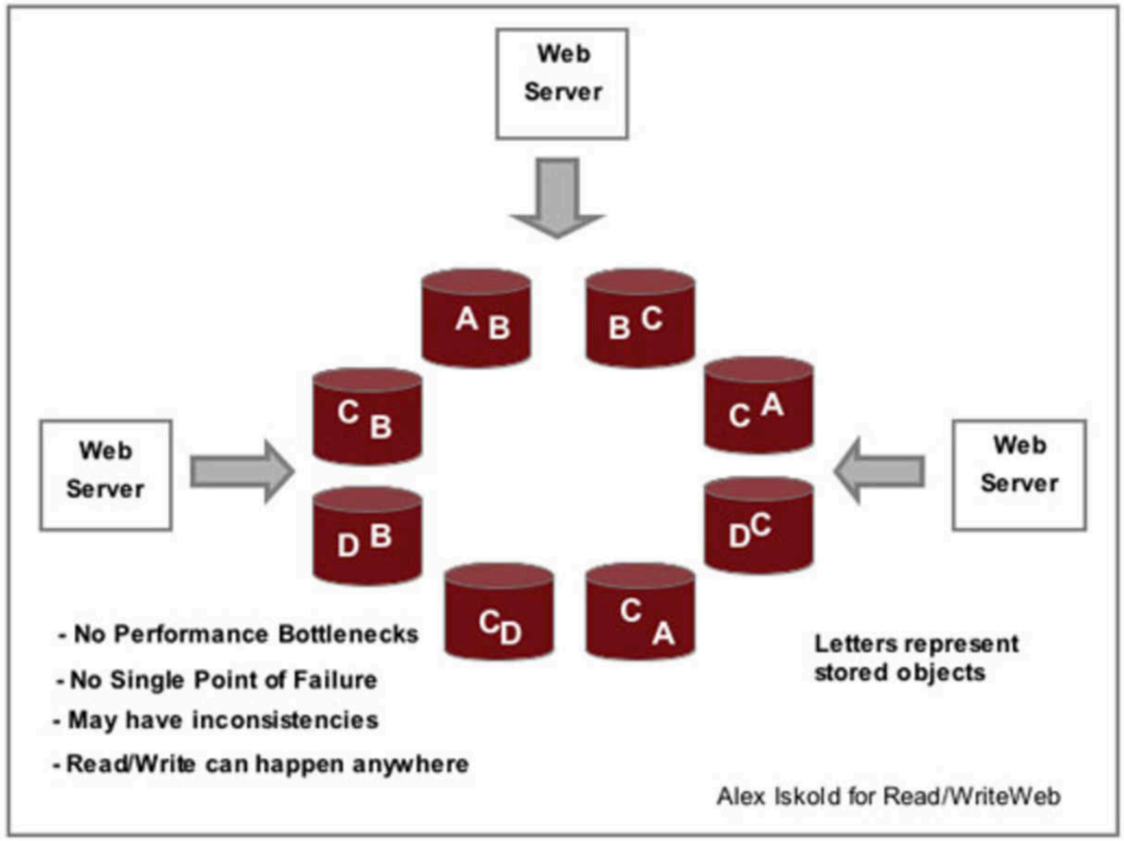
- Background
- System Requirements
- Design Principles
  - Distributed Storage
  - Key Design Principles
- System Architecture
- Implementation & Lessons
- Conclusions

Distributed Storage

Remove the database as the bottleneck.

Distributed **storage nodes** share the burden.

Requests are routed to the storage node holding the data.



http://readwrite.com/2007/10/30/amazon_dynamo

Optimistic Replication

Optimistic replication allows changes to propagate asynchronously. Availability is increased, but the **risk is that you have multiple, conflicting versions of data** in the system.

Conflicts aren't prevented, but resolved.

- Notion of an "eventually consistent data store" and delaying reconciliation.
- **When** to resolve: resolving conflicts during reads, not writes (e.g. shopping cart example, cannot reject writes).
- **Who** to resolve: tradeoff between system and application level resolution.

Key Design Principles

- **Incremental scalability**: Dynamo should be able to scale out one storage host (henceforth, referred to as "*node*") at a time
- **Symmetry**: Every node in Dynamo should have the same set of responsibilities as its peers; there should be no distinguished node or nodes that take special roles or extra set of responsibilities.
- **Decentralization**: An extension of symmetry, the design should favor decentralized peer-to-peer techniques
- **Heterogeneity**: The system needs to be able to exploit heterogeneity (i.e. work allocated is proportional to the characteristics of the hardware).

Outline

- Background
- System Requirements
- Design Principles
- System Architecture
  - System Interface
  - Partitioning & Virtual Nodes
  - Replication
  - Data Versioning
  - Execution of get() and put()
  - Failures & Data Consistency
  - Failures & Hinted Handoff
  - Membership & Failure Detection
- Implementation & Lessons
- Conclusions

Dynamo Techniques

Problem	Technique
Partitioning and replication	Consistent hashing (notions of “eventually consistent” and delayed reconciliation).
Consistency	Object versioning. Quorum-like techniques used to maintain consistency.
Recovering from failures	Merkle trees used to quickly detect inconsistencies.
Membership	Gossip-based membership protocol, also used for failure detection.
Decentralized	Minimal need for manual administration (i.e. no manual partitioning required)

System Interface

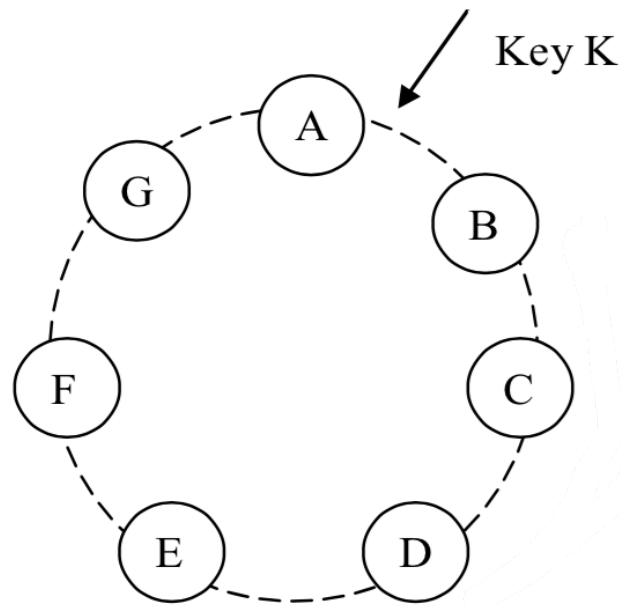
- Focus on *simple* query model.
- Key-value storage of objects:
  - get() : for a given key, returns a single object or list of objects with their *context* (metadata including version).
  - put() : writes replicas (versions) to disk.

Partitioning

Scale incrementally by dynamic partitioning across all available nodes.

Consistent hashing: the output range of the hash function returns is a bounded, circular region.

- Newly added nodes are randomly assigned a key / position.
- Nodes are responsible for the values ranging from the previous node to themselves.



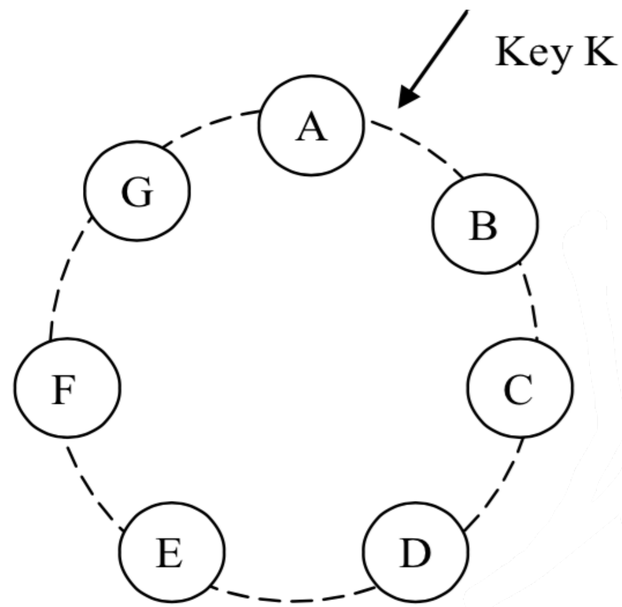
Partitioning: Virtual Nodes

The Problem with Consistent Hashing:

- Random positioning leads to non-uniform load distribution.

Virtual nodes improve reliability:

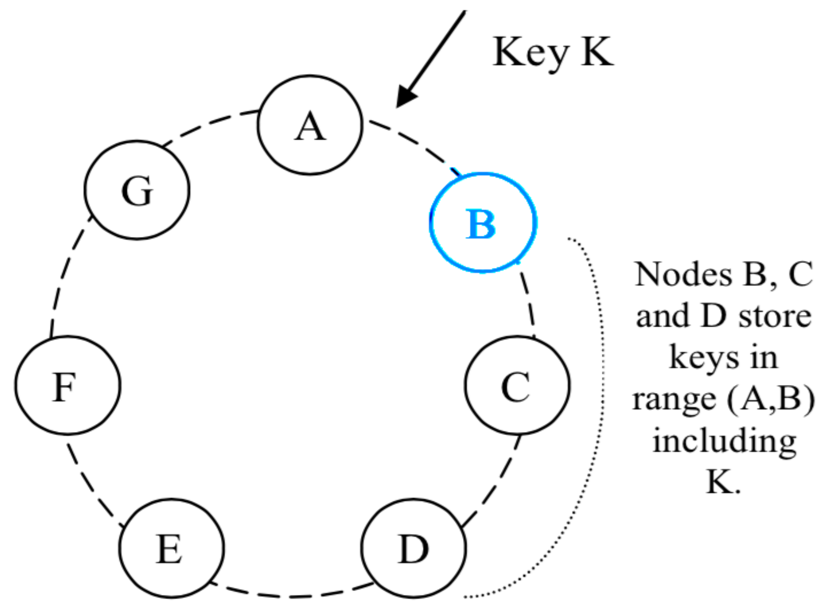
- Each node gets assigned to multiple “virtual” positions.
- This allows for failover when a node is available, or load rebalancing in extreme cases.
- Virtual nodes are easier to reallocate!



Partitioning: Replication

To achieve high-reliability, Dynamo replicates data across nodes.

- Each key is assigned a "coordinator" node that is responsible for handling replication.
- The coordinator, in turn, handles replication for all items that fall within its range.
- The coordinator replicates these keys at the N-1 clockwise successor nodes in the ring.
- This list of key-owners is called the *preference list* (and is circulated around the system).



Consistency: Data Versioning

Dynamo guarantees “eventual consistency”. **Updates are propagated asynchronously**, so there’s no guarantee that replicas are always in-sync.

Multiple versions of a particular key:value pair may exist at any given time (e.g. one node returns a value before a replica has propagated an update).

To handle this:

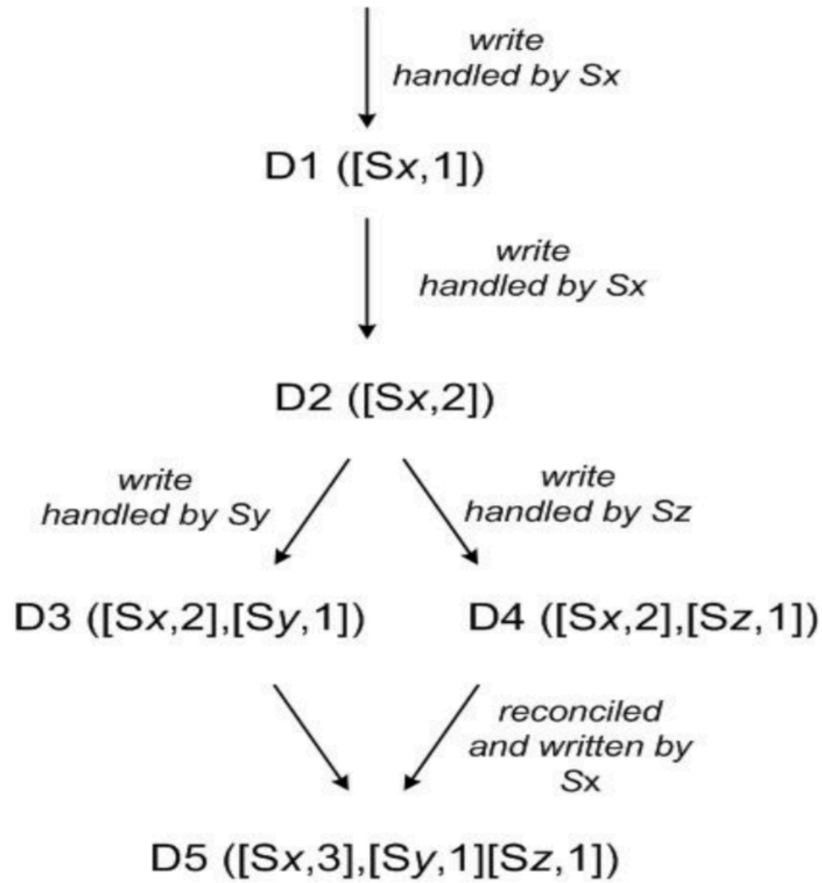
- Each modification results in a new version being created.
- Most of the time, new versions replace old versions.
- When versions cannot be reconciled automatically, vector clocks are used to order the versions and attempt to reconcile version histories.

Consistency: Vector Clocks

Client updates must specify a version (from the context that it obtained during the initial get() operation).

Any updates to data will result in the node creating a new version (with a new vector clock timestamp).

If a node is responsible for both read/write, and may reconcile and collapse versions. If not, both versions need to be passed back to the application for reconciliation.



Handling Failures: Base Case

Normal operations look like this:

- get() and put() operations from the application need to be sent to the appropriate node.
- Two strategies to locate this node:
  - Allow a load-balancer to select the appropriate node
  - Use a partition-aware library that knows which nodes are responsible for a given key. This pushes responsibility for node selection onto the application.
- A node handling a read or write operation is known as the ***coordinator***. Typically, this is the first among the top N nodes in the preference list.

Handling Failures: Data Consistency

What happens when storage nodes disagree?

Dynamo uses a quorum-based consistency protocol. This means that a number of storage nodes must "agree" on result.

For a series of N nodes, there are two configurable values for a given request:

- **R** : the minimum number of nodes that must participate in a **read** request
- **W** : the minimum number of nodes that must participate in a **write** request

Use:

- $R + W > N$: Quorum
- $R + W < N$: Not-Quorum (but better latency!)

Handling Failures: Quorum Use

Examples

- put(): The coordinator node will
  - Create a new version.
  - Send the date to N healthy nodes.
  - If W-1 respond, treat the write as successful.
- get(): The coordinator node will
  - Request versions of that data from N highest-ranked reachable nodes in the preference list.
  - Waits for R responses before returning the result to the client. If it receives multiple versions, it will reconcile and write-back a new version (superseding the previous ones).

Handling Failures: Hinted Handoff

Dynamo uses a "sloppy quorum", where the first *available, healthy* nodes are used to determine quorum. This increases durability for writes.

- **Hinted handoff** occurs when a node is unavailable, and the next replica handles the request. This node tracks the node that was unavailable, and when it comes back online, delivers the replica.
- Value of W determines durability:
  - W=1: allows any writes as long as a single node is available to process it.
  - W>1: in practice, we usually aim slightly higher than 1.

Handling Failures: Replica Sync

- To detect inconsistencies between replicas faster, and decrease time required, Dynamo uses **Merkle trees**.
  - A Merkle tree is a hash tree where leaves are hashes of the individual leaves. Parent nodes higher in the tree are hashes of their children.
  - Each branch can be checked for membership without traversing the entire tree. Less data needs to be passed around when checking status.
- Each node maintains a Merkle tree for each key range it hosts. To compare values with another node, they just exchange root nodes for that tree.

Membership & Failure Detection

"A node outage rarely signifies a permanent departure and therefore should not result in rebalancing of the partition assignment or repair of the unreachable replicas. "

- Changes to membership (adding or removing nodes) is done *manually* to avoid thrashing.
- **Gossip-based protocol** propagates changes automatically:
  - Each node contacts another random node and they exchange membership information.
  - Newly added nodes are assigned virtual nodes, and gossip.
  - Seed nodes prevent new nodes from becoming isolated.
- Node-mappings are also propagated though gossip!
- Unresponsive nodes are also flagged and gossiped-about.

Outline

- Background
- System Requirements
- Design Principles
- System Architecture
- Implementation & Lessons
  - Usage Patterns
  - Performance Tradeoffs
  - Load Distribution
  - Divergent Versions
- Conclusions

Storage Node Overview

Built in Java.

Each storage node has three main components:

- Request coordination
  - Multi-stage messaging pipeline, built using Java NIO channels.
  - Each client requests results in the creation of a single state machine to handle that request.
- Membership and failure detection
- Local persistence engine
  - Pluggable, supports many different engines (incl. Berkeley Database (BDB) Transactional Data Store, MySQL, in-memory).
  - Most use BDB Transactional Data Store.

Usage Patterns

Dynamo is used by a number of services with drastically different usage patterns:

- **Business logic specific reconciliation**: Many-node replication, with client handling reconciliation. e.g. shopping cart logic.
- **Timestamp based reconciliation**: "Last write wins". e.g. customer session service.
- **High-performance read engine**: Services with a high read-request rate, small number of updates. e.g. product catalogs.

Value in allowing applications to tune R and W (affecting consistency, durability, availability)

- Common: (N:3, R:2, W:2)
- High-performance read: (N:3, R:1, W:3)
- High-performance write: (N:3, R:3, W:1) – dangerous

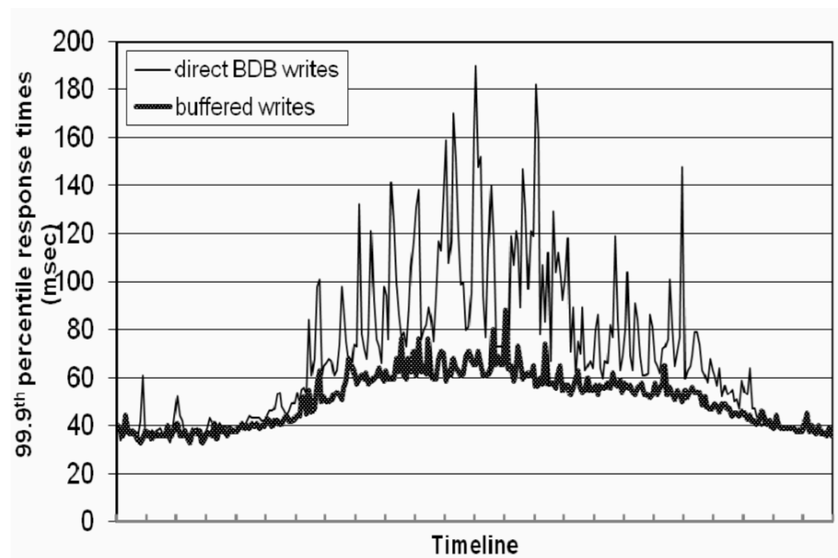
Performance Tradeoffs

“A typical SLA required of services that use Dynamo is that 99.9% of the read and write requests execute within 300ms. “

- Commodity hardware makes I/O challenging.
- Multiple storage nodes constrains performance to the slowest node.

To achieve higher-performance on writes, an *optional* writer thread can be used to buffer writes.

- Improves latencies at the risk of losing data (i.e. server crash).
- Can mitigate by having a node assigned to “durable writes”.



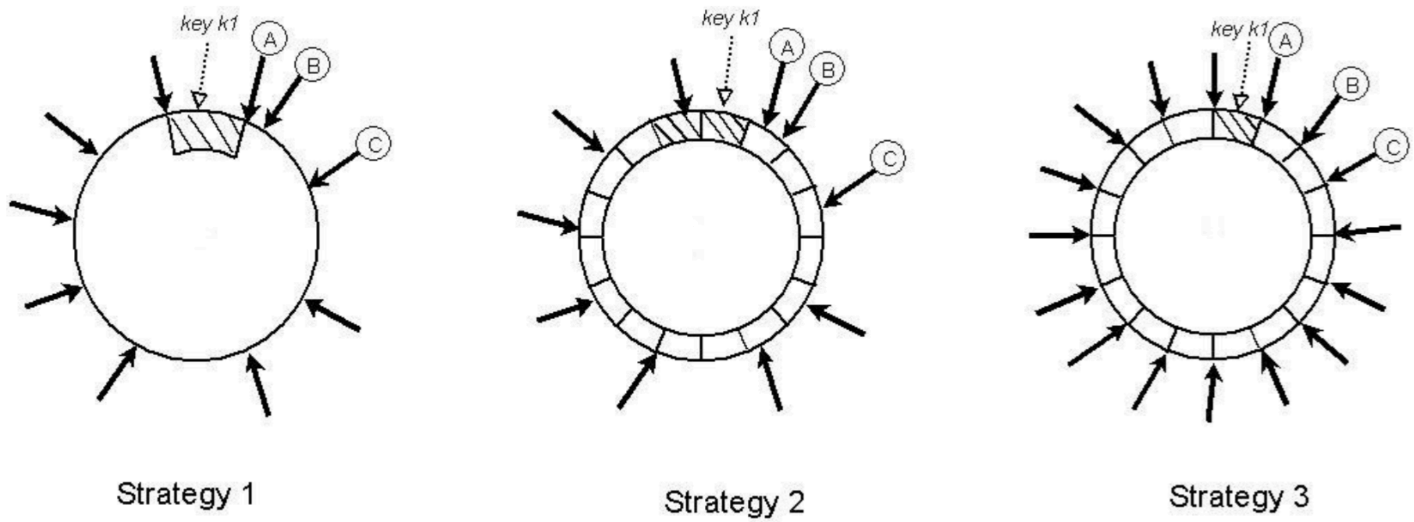
Load Distribution

Dynamo uses consistent hashing to partition it's key space across all of the available nodes.

- This results in a semi-uniform distribution.
- The assumption is that there are enough nodes at either end of the distribution to handle any skewed access patterns (i.e. "popular" requests at peak times).

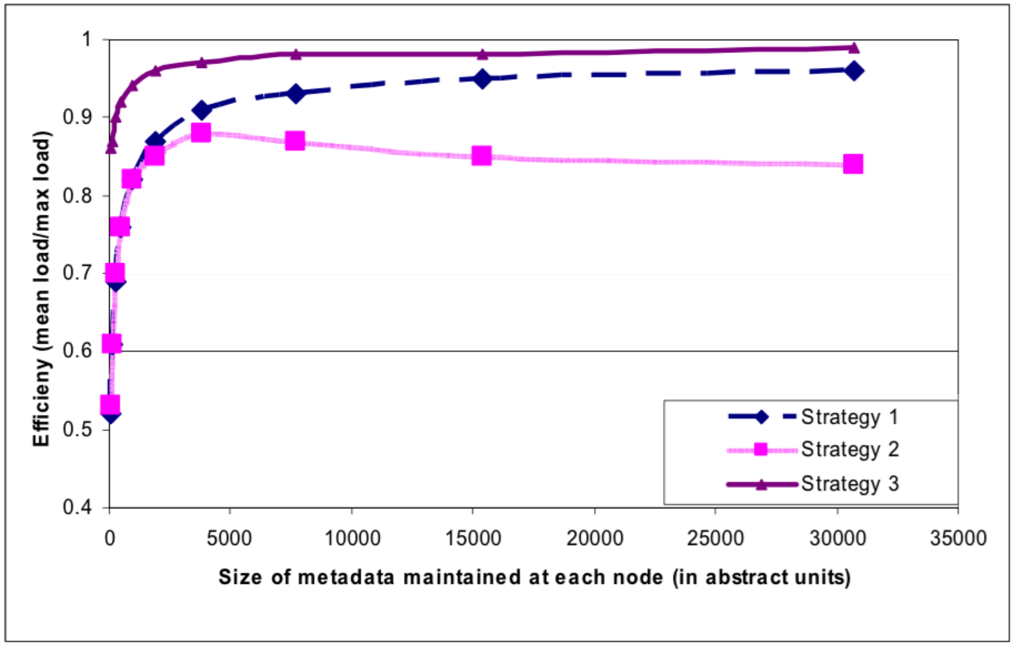
Is there a better way of partitioning keys?

Load Distribution Strategies



T random tokens per node and **partition by token value**	T random tokens per node and **equal sized partitions**	Q/S tokens per node, **equal-sized partitions**
• New nodes need to "steal" key ranges from exiting nodes. • Changing key ranges invalidates Merkle trees. • Difficult to archive.	• Decoupling of partitioning and partition placement. • Enables the possibility of changing the placement scheme at runtime.	• When a node leaves the system, its tokens are randomly distributed to the remaining nodes. • When a node joins the system it "steals" tokens from other nodes.

Load Balancing Efficiency



- Strategy 2 is the worst, Strategy 3 is the best.
- Compared to Strategy 1, Strategy 3 achieves better efficiency and reduces the size of membership information maintained at each node by three orders of magnitude.
- Strategy 3 is faster to bootstrap (fixed partitions) and archive.

Divergent Versions

Divergent versions of a data item occur when

- Failures are happening, such node or data center failures.
- A large number of writes against the same data are happening, and multiple nodes are handling the updates.

The number of versions returned to the shopping cart service was profiled for a period of 24 hours.

1 version	99.94% of requests
2 versions	0.00057% of requests
3 versions	0.00047% of requests
4 versions	0.00009% of requests

Coordination

There are two ways of locating a node to service a request:

- A load-balancer can determine the node for a given key/request. The burden is on the load-balancer/system.
- The client can periodically sample a random node (every 10 seconds), grab its membership state and use that to query nodes directly.

	99.9th percentile read latency (ms)	99.9th percentile write latency (ms)	Average read latency (ms)	Average write latency (ms)
Server-driven	68.9	68.5	3.9	4.02
Client-driven	30.4	30.4	1.55	1.9

Outline

- Background
- System Requirements
- Design Principles
- System Architecture
- Implementation & Lessons
- Conclusions

Conclusions

Dynamo is a highly available and scalable data store, that **provides customers with the ability to customize their storage system** to meet their desired performance, durability and consistency SLAs.

It has been proven as a durable, robust and scalable solution to delivering at massive-scale. Its success demonstrates that "**an eventual- consistent storage system can be a building block for highly- available applications**".

Discussion

Q: Partitioning. Strategies assume a fairly uniform access pattern. How could we account for skewed access patterns (i.e. popular query items)?

Q. Performance. What are the tradeoffs for different R and W values? Would you realistically use R=1 or W=1, or would you expect to just set R=2, W=2?

Q. Scale. Where might you see bottlenecks as it scale out?

Q. Java? I would have expected some performance issues (e.g. large heap sizes). Is this really a concern?

Q. Features. Customers may, in theory, be exposed to un-reconciled versions (e.g. deleted items reappearing in a shopping cart). Was this a reasonable choice?

References

Amazon

- Werner Vogels, Blog Post, 02 Oct 2007. http://www.allthingsdistributed.com/2007/10/amazons_dynamo.html
- Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swami Sivasubramanian, Peter Vosshall and Werner Vogels, "Dynamo: Amazon's Highly Available Key-Value Store", in the *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, Stevenson, WA, October 2007.

Press

- A Conversation with Werner Vogels: http://queue.acm.org/detail.cfm?id=1142065
- Consistency and availability in Amazon's Dynamo: http://the-paper-trail.org/blog/consistency-and-availability-in-amazons-dynamo/
- Amazon Dynamo: The Next Generation Of Virtual Distributed Storage: http://readwrite.com/2007/10/30/amazon_dynamo
- Ars Technica: Dynamo: http://arstechnica.com/uncategorized/2007/10/amazon-reveals-its-distributed-storage-dynamo/