Storage Systems for the Cloud

Introduction

1. Cloud is all about scale, availability, reliability, and performance
    - Reliability and performance are often at odds with each other
    - NoSQL storage systems have taken over cloud computing

Storage Systems for the Cloud

1. Requirements
    - Scale, scale, scale…
    - High throughput
    - Support for concurrent access without stifling performance
    - Availability and reliability in the presence of failures and continuous evolution of the data center resources
    - Horizontal scalability
    - "One size fits all" thinking of DB systems not appropriate for cloud storage
        - Read dominant request pattern
        - Databases are more concerned with ACID properties and concurrent writes
    - Performance vs reliability conundrum
2. NoSQL Storage
    - "Not only" SQL
    - Simplicity in design and performance scalability
        - <key, value> organization
    - Weaker consistency semantics
        - Eventual consistency
            - Reads could return stale (but not incorrect) data
            - Given "enough time" reads will return the "last write" to the data
    - Use of commodity hardware components
    - Availability -> geographical replication
    - Easier evolution in the presence of failures and upgrades
    - Evolution of NoSQL vs relational databases is analogous to GPU vs CPU in computer architecture

Amazon Dynamo

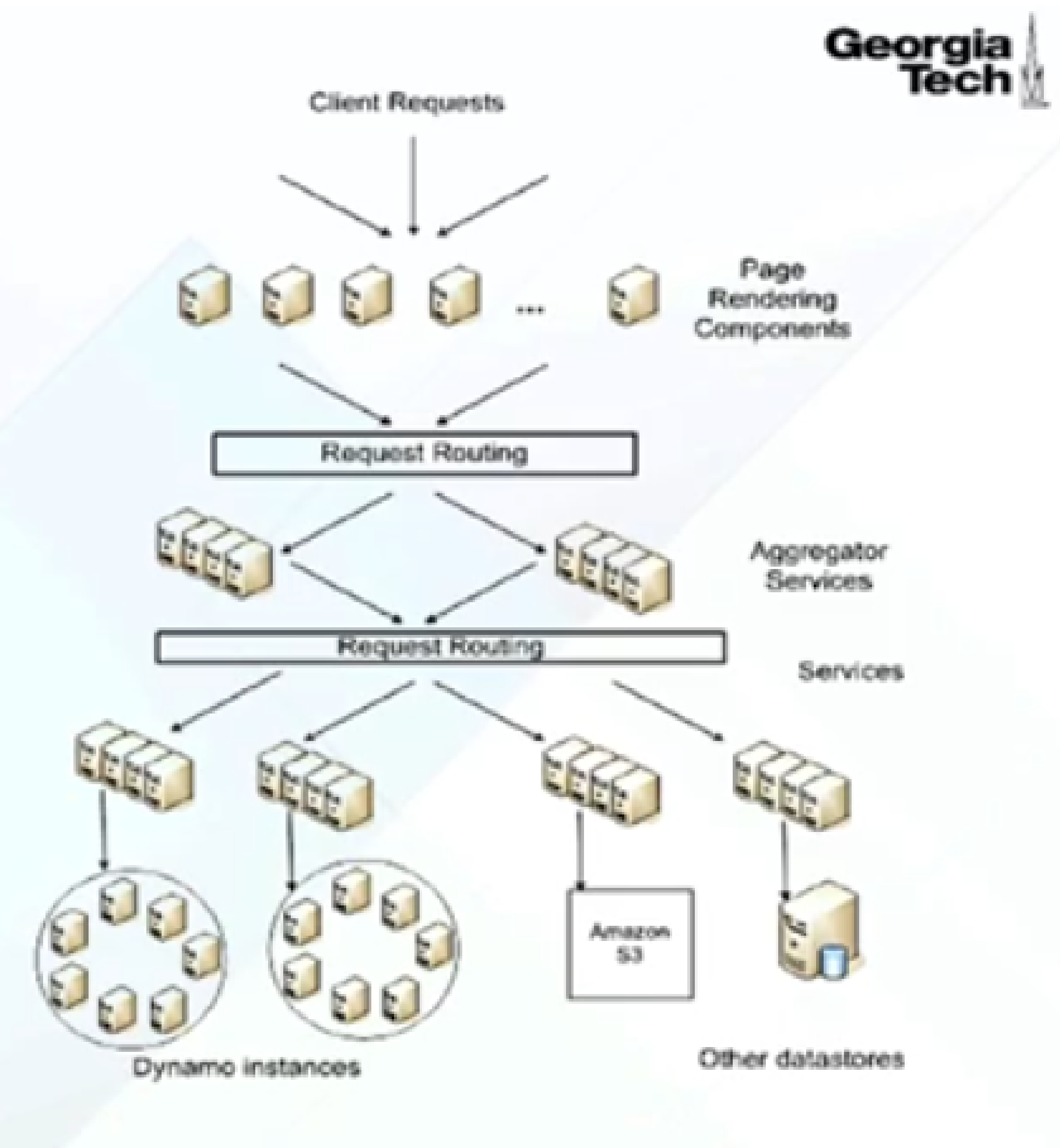
1. Assumptions
    - Target services to support
        - Best seller lists, shopping carts, customer preferences, session management, sales rank, and product catalog
    - Query model
        - Simple read/write operations to data items identified by a key
        - Operations do not span multiple data items -> no need for relational schema
    - Trusted operational environment
2. Design goals
    - High reliability requirements and need tight control over the tradeoffs between availability, consistency, cost-effectiveness, and performance
        - Reliability: Can I trust the results?
        - Avaialability: Even if failures occur, can I retrieve my data?
    - Support SLAs: 99th percentile rather than median or averages
    - Availability in the presence of failures

Amazon Dynamo Design Principles

1. Principles

- Decentralized system with minimal need for manual administration
- Synthesis of well-known techniques
  - Data partitioning and replication
  - Object versioning using vector clocks for consistency
  - Replica consistency via quorum-like techniques
  - Gossip-based failure detection and membership maintenance
- Query processing may touch several sources
  - Services make their own tradeoffs between functionality, performance, and cost-effectiveness
  - Honor SLAs
- Scholarship of discovery: Finding new techniques
- Scholarship of integration: Taking well-known techniques and applying them

Amazon Dynamo System Architecture

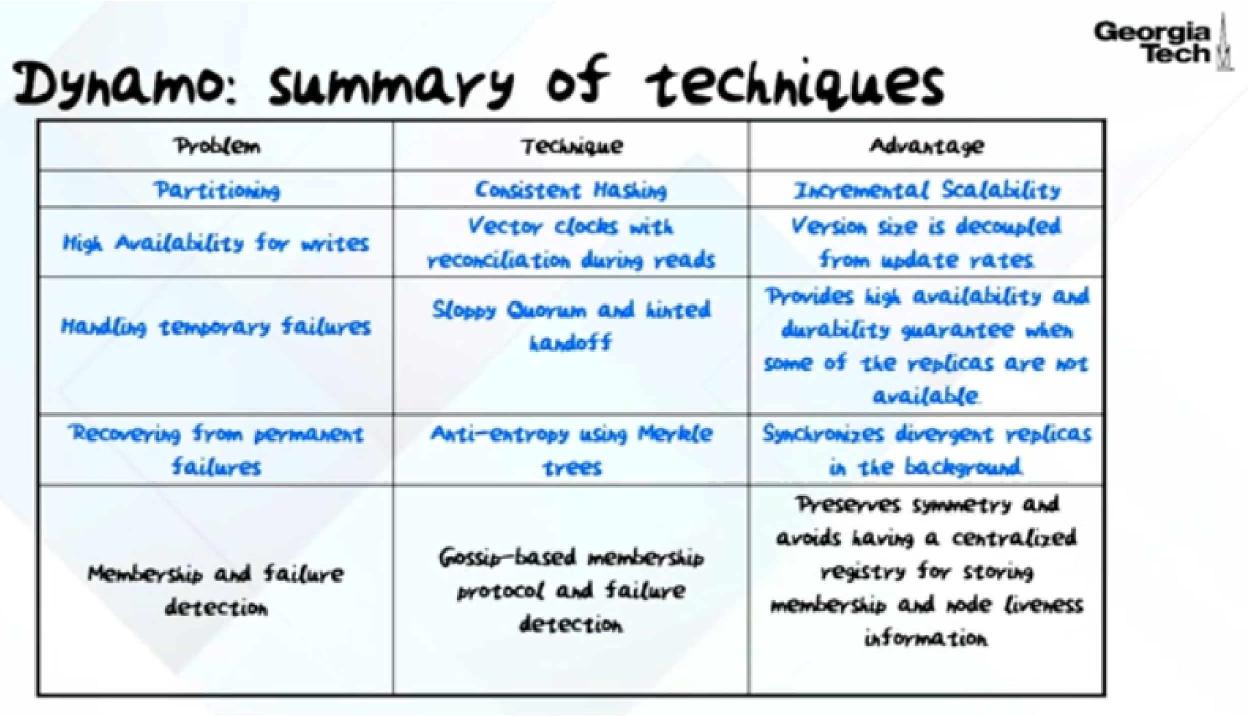


Amazon Dynamo

Amazon Dynamo Design Details

1. Data items “always writable”
   - Conflict resolution at the time of “read”
   - Merge conflict resolution is a responsibility of the application, not the system
2. Data partitioning
   - Consistent hashing -> incremental stability
3. Replication
   - Overlaid on consistent hashing to increase availability
4. Eventual consistency
   - Versioning and vector clocks to achieve consistency

Amazon Dynamo Summary



Dynamo: summary of techniques

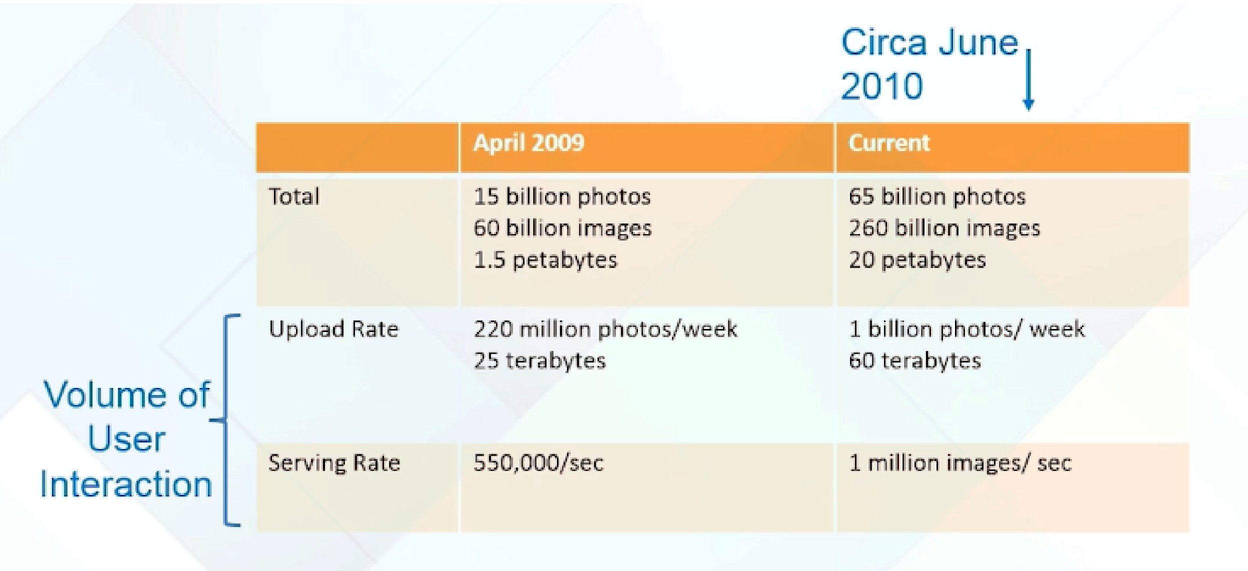
Problem	Technique	Advantage
Partitioning	Consistent Hashing	Incremental Scalability
High Availability for writes	Vector clocks with reconciliation during reads	Version size is decoupled from update rates
Handling temporary failures	Sloppy Quorum and hinted handoff	Provides high availability and durability 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

Amazon Dynamo Summary

1. Dynamo does a form of garbage collection of old values once it is certain that the value is no longer used

Facebook Haystack

1. How to build a scalable/available photostore?
    - Good example of a problem-oriented design
    - Design point not for generality as in Amazon Dynamo by customization



Circa June 2010

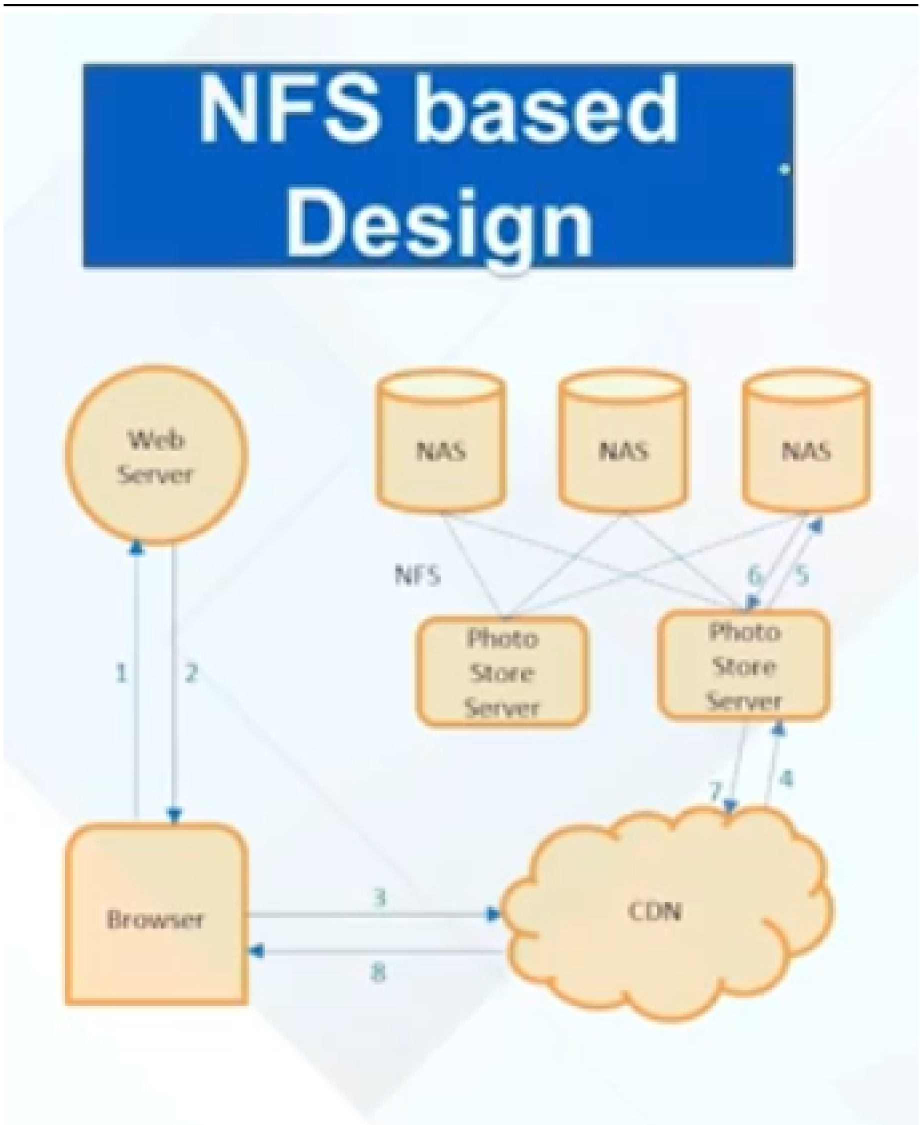
	April 2009	Current
Total	15 billion photos 60 billion images 1.5 petabytes	65 billion photos 260 billion images 20 petabytes
Upload Rate	220 million photos/week 25 terabytes	1 billion photos/ week 60 terabytes
Serving Rate	550,000/sec	1 million images/ sec

Volume of User Interaction

Facebook Volume of Photos

Haystack NFS-based Design

1. Long-tail distribution: Popular photos are served by the CDN and less popular photos are serviced by the NFS
   - Long latency for less popular photos
   - CDN serves as a cache for NFS store
   - Metadata bottleneck: Can't store data for all photos in CDN, too big



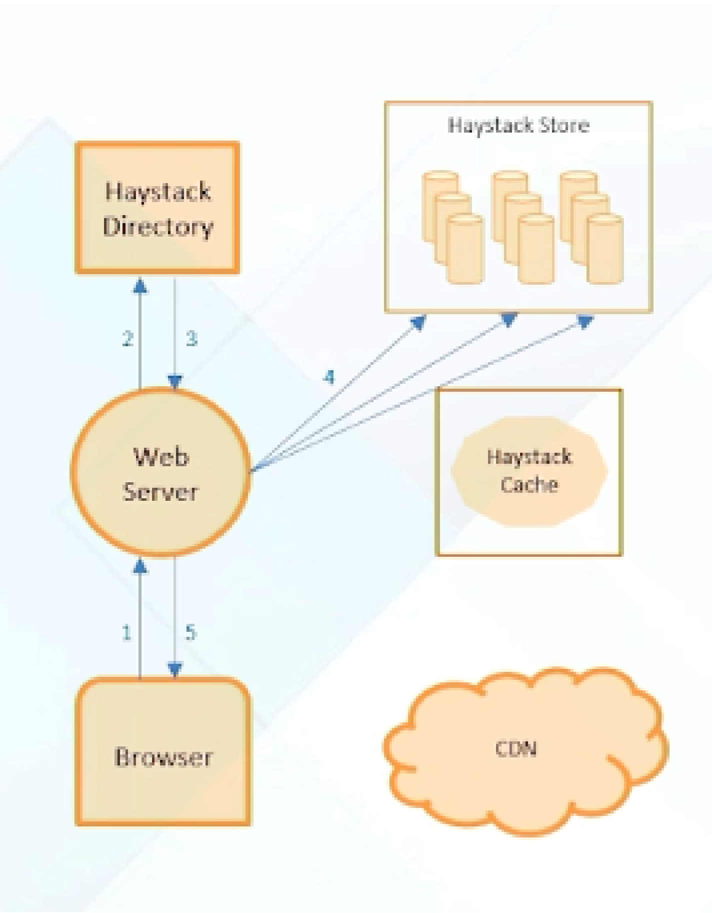
NFS based Design

Haystack NFS

Haystack Haystack-based Design

1. Store multiple photos in one file
    - Single inode is a repository for many photos
    - Log-structured file system: Instead of storing filepaths and updating files in place, changes are stored in a log file
        - Reduces number of small writes in the system
2. Replaces photo server and NAS
3. Upshot
    - Metadata for "long-tail" photo will be in the Haystack cache
    - Photo itself may be in Haystack-store
4. Photo Upload in Haystack
    - Request photo upload
    - Find volume to write
    - Return load-balanced writable volume
    - Write to replicated physical volumes for availability

    - Append to end of volume
5. Photo Download
    - Request photo download
    - Find logical volume
    - Assign load-balanced physical volume to read
    - Browser gets the photo handle (needle)
    - Directory decides the read path depending on long-tail or not
        - Direct to Haystack cache and then store, OR
        - Via CDN
            - Miss in CDN means don't bother caching the result



Haystack

Haystack Summary

1. Haystack directory has the mapping given a photo URL
    - Haystack organized into logical volumes
    - Each photo has a marker called "needle" in the haystack
    - URL to needle mapping in the directory
    - Directory tells the browser whether to go to the CDN or the cache
2. Haystack cache responds to <haystack, needle> requests
    - Direct requests from the browser
        - Metadata is likely in the cache; if not, bring from disk and cache
    - Requests from the CDN -> do not cache

3. Haystack store
    - Log structured file system

Google Bigtable

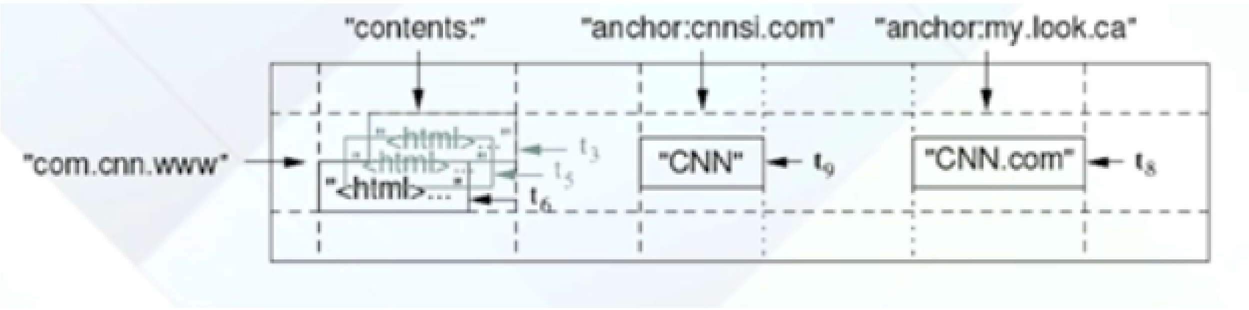
1. What is the right abstraction for dealing with the data produced and consumed by a variety of Google projects
    - Web crawling/indexing, Google earth, Google finance, Google news, Google analytics
        - All of them generate lots of structured data
        - Interactive (human in the loop) and batch
2. Why a new abstraction?
    - Relational data model is too general
        - Does not lend itself to efficient application-level locality management and hence performance
    - <key, value> stores are too limited for multi-dimensional indexing into semi-structured data repositories
    - A new abstraction allows incremental evolution of features and implemenation choices (e.g., in memory, disk, etc)
3. Data model of Bigtable
    - Sparse, distributed, persistent, multi-dimensional, sorted data structure
    - Index by {row key, column key, timestamp}
    - "value" in a cell of the table is an un-interpreted array of bytes

Google Bigtable Rows

1. Row creation is automatic on write to the table
    - Row name is an arbitrary string (up to 64 KB in length)
    - Lexicographically ordered to aid locality management -e.g. reverse URL (hierarchically descending from DNS name resolution perspective)
2. Row access is atomic
3. Tablet
    - A range of rows, unit of distribution of load-balancing
    - Locality of communication for access to a set of rows

Google Bigtable Columns Timestamps API

1. Columns
    - A row may have any number of columns
    - Column family has to be created before populating the columns with data
        - Two level name for column key
        - Once family created, any column key within the family can be used
    - Examples
        - "Contents" stores web pages (so no additional qualifier)
        - "Anchor" is a family with 2 members in this table; "value" stored is the link text



Bigtable

2. Timestamps
    - Associated with multiple versions of the same data in a cell
        - Writes default to current time unless explicitly set by clients
    - Lookup options:
        - "Return most recent k values"
        - "Return all values in timestamp range (or all values)"
    - Column families can be marked with attributes
        - "Only retain most recent k values in a cell"
3. Bigtable API
    - Read operation
    - Write operation for rows
    - Write operations for Tables and Column families
    - Administrative operations
    - Server side code execution
    - Map-reduce operations

Facebook Cassandra

1. Decentralized structured storage system
2. Data model similar to Bigtable
    - Rows and column families with timestamps
3. Dynamo's ideas for distribution and replication
    - Uses the row key and consistent hashing (a la Dynamo) to distribute and replicate for load balancing and availability
4. Facebook implemented this for messaging

Google Spanner

1. Focus on managing cross-datacenter replicated data
2. Automatic failover among replicas
    - Fault tolerance and disaster recovery
3. Evolved from Bigtable to support applications that need stronger semantics in the presence of wide-area multi-site replication
4. Globally distributed multiversion database
    - Replication can be controlled at a fine grain
    - Externally consistent reads and writes (given a timestamp) despite global distribution
    - Globally consistent reads across the database
5. Features
    - Schematized semi-relational data model
    - Query language and general-purpose transactions (ACID)
    - Interval-based global time

Conclusion

1. Covered storage technologies employed by tech giants
2. Pillar of reliable storage technologies (relational databases) that were pioneered by IBM and used by the industry for several decades is inappropriate for meeting the requirements of horizontal scaling and high throughput in the context of cloud computing
    - There are global scale applications that still need stronger semantics