ISEngineering

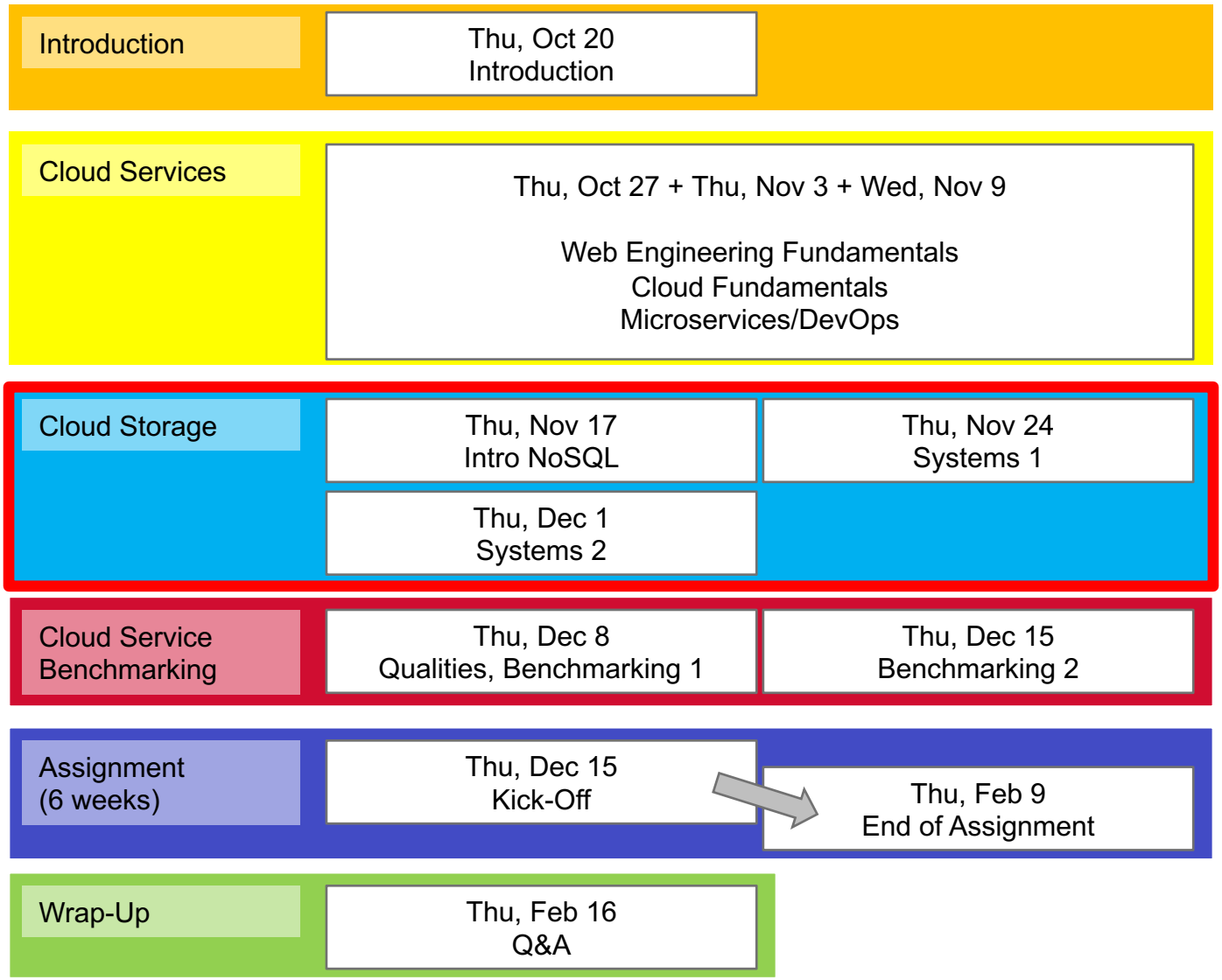
Technische Universität Berlin

Enterprise Computing – Cloud Storage

Stefan Tai, Marco Peise

ISEngineering
Wirtschaftsinformatik –
Information Systems Engineering

Nov 17, Nov 24, Dec 1



Agenda

1. **Intro Cloud Storage**

2. Systems in detail
   a) Dynamo
   b) GFS, BigTable

3. Other systems
   c) Cassandra
   d) MongoDB

Recap: ACID transactions with traditional relational DBMS

A transaction is a sequence of operations that is guaranteed to be atomic in the presence of concurrent clients and failures

- Free from interference by operations being performed on behalf of other concurrent clients
- Either all of the operations must be completed successfully, or, they must have no effect at all even in the presence of (service, server, network) failures

Recap: ACID transactions with traditional relational DBMS (cont.)

In traditional relational DBMS, transactions are known to have ACID properties:

- Atomicity: The “All or Nothing” behavior
- Consistency: Each transaction, if executed by itself, transforms the database in a ‘consistent’ manner (regarding schema and integrity constraints)
- Isolation (Serializability): Concurrent transaction execution should be equivalent (in effect) to a serialized execution
- Durability: Once a transaction is done, it stays done

…but the world is changing

There is no longer a database capable of handling the needs of today‘s web applications

- Yahoo!: „hundred of thousands of requests per sec” (2009)
- Youtube: > 11,500 views per second (2009)
- Twitter: > 2300 tweets per second (2011)
- Facebook: 4.75 bn. posts, 10 bn. messages per day (2013)

Downtime visible to end users is not an option (outages cost $millions)

Huge variability in load

Sources:
- *highscalability.com*
- *blog.twitter.com*
- *Ramakrishnan et al., Yahoo!, 2009*

Infographic source: smartinsights.com

Application needs

Web serving applications need:

- Scalability! Preferably *elastic*
- Flexible schemas
- Geographic distribution
- High availability
- Reliable storage

Web serving applications typically can do without:

- Complicated queries
- Strong transactions
  - but some form of consistency is sometimes still desirable

Source: Ramakrishnan et al., Yahoo!, 2009

Very Large Scale Distributed Data Stores

Technische Universität Berlin

Must partition data across machines

- How are partitions determined?
- Can partitions be changed easily? (affects elasticity)
- How are read/update requests routed?
- Range selections? Can requests span machines?

Availability:

What failures are handled?

- With what semantic guarantees on data access?

(How) Is data replicated?

- Sync or async?
- Consistency model?
- Local or geo?

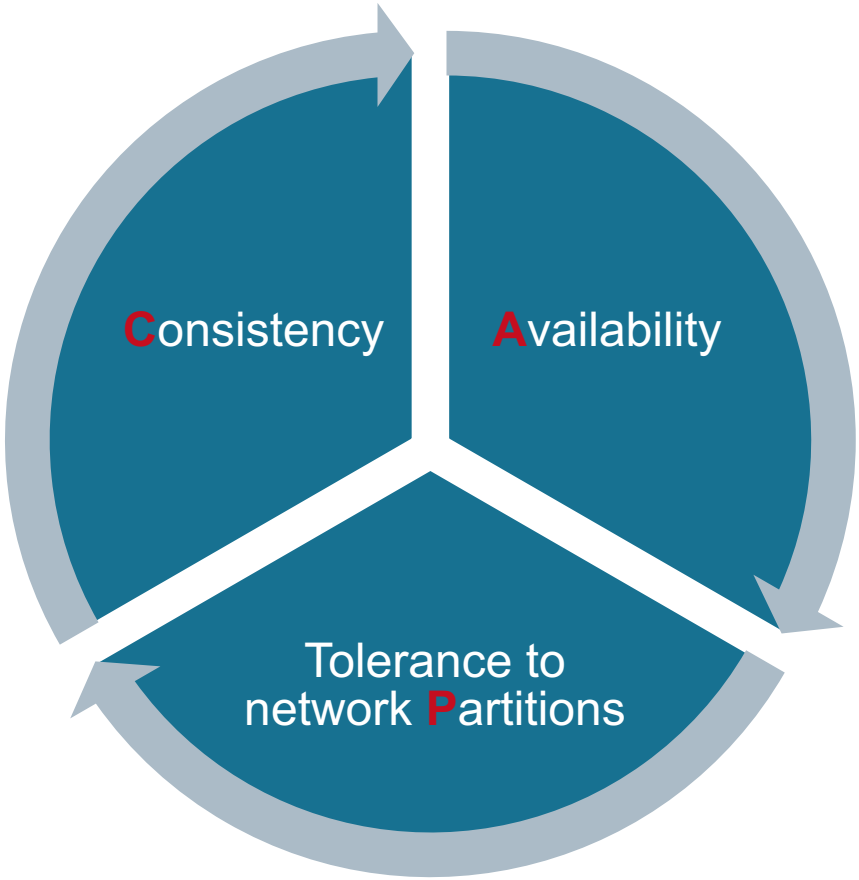
How are updates made durable?

How is data stored on a single machine?

Source: Ramakrishnan et al., Yahoo!, 2009

Brewer‘s CAP Theorem

You can have at most two of these properties for any shared-data system:



CAP Theorem

Technische Universität Berlin

Consistency of data

Serializability / “a total order on all operations exists such that each operation looks as if it were completed at a single instant” [2]

Availability

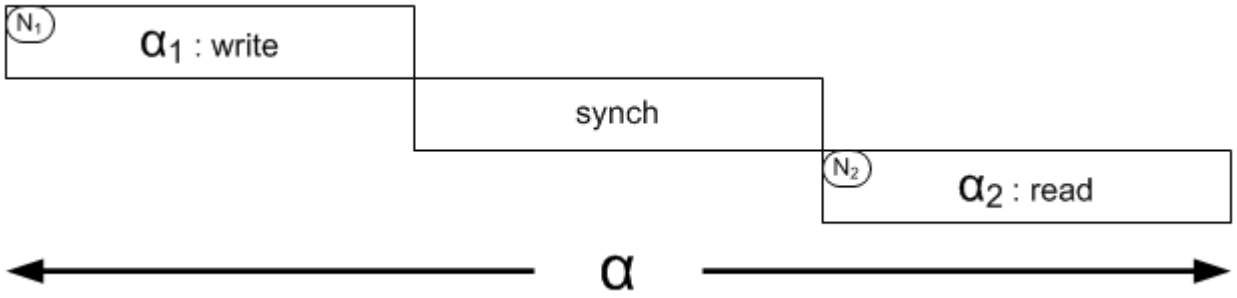
Every request must result in a response

Partition tolerance

Live nodes should not be blocked by partitions / “no set of failures less than total network failure is allowed to cause the system to respond incorrectly” [2]

[1] Brewer, PODC 2000
[2] Gilbert/Lynch, SIGACT News 2002

CAP Theorem Illustrated



Consider transaction α (i.e. unit of work based around the persistent data item V) with α1 being a write operation and α2 could being a read operation.

On a local system this would be easily be handled by a database with some simple locking, isolating any attempt to read in α2 until α1 completes safely.

In the distributed model though, with nodes N1 and N2, the intermediate synchronizing message also has to complete. Unless we can control when α2 happens, we can never guarantee it will see the same data values α1 writes. All methods to add control (blocking, isolation, centralized management, etc) will impact either partition tolerance or the availability of α1 (A) and/or α2 (B).

Source: http://www.julianbrowne.com/article/viewer/brewers-cap-theorem

CAP Theorem – Example

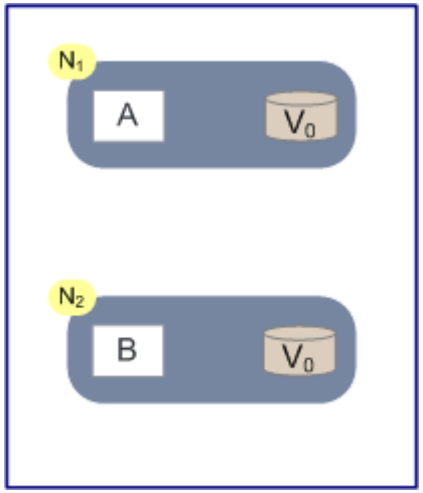


Figure source: http://www.julianbrowne.com/article/viewer/brewers-cap-theorem

CAP Theorem – Example cont.

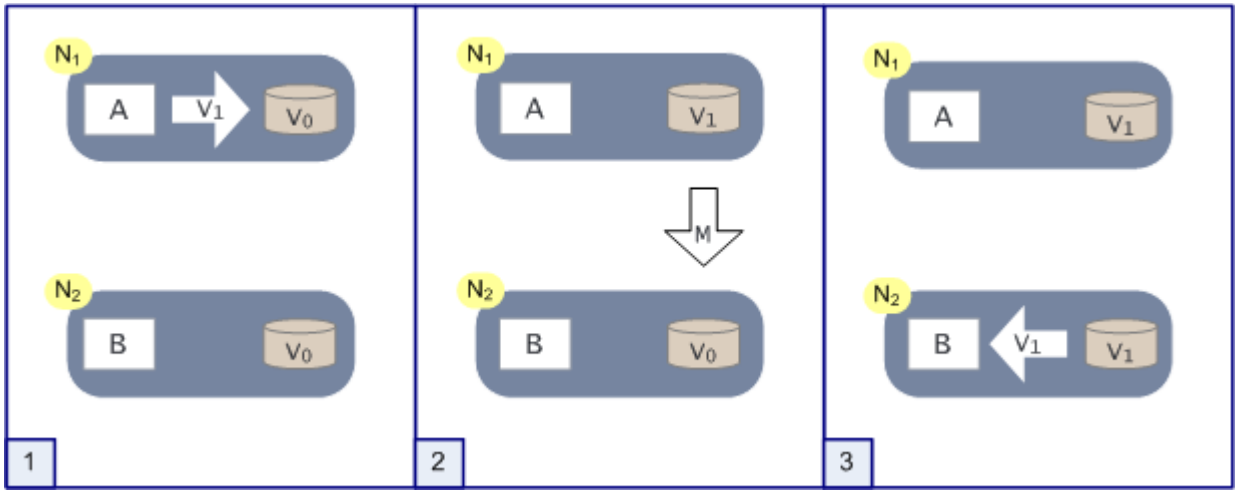


Figure source: http://www.julianbrowne.com/article/viewer/brewers-cap-theorem

CAP Theorem – Example cont.

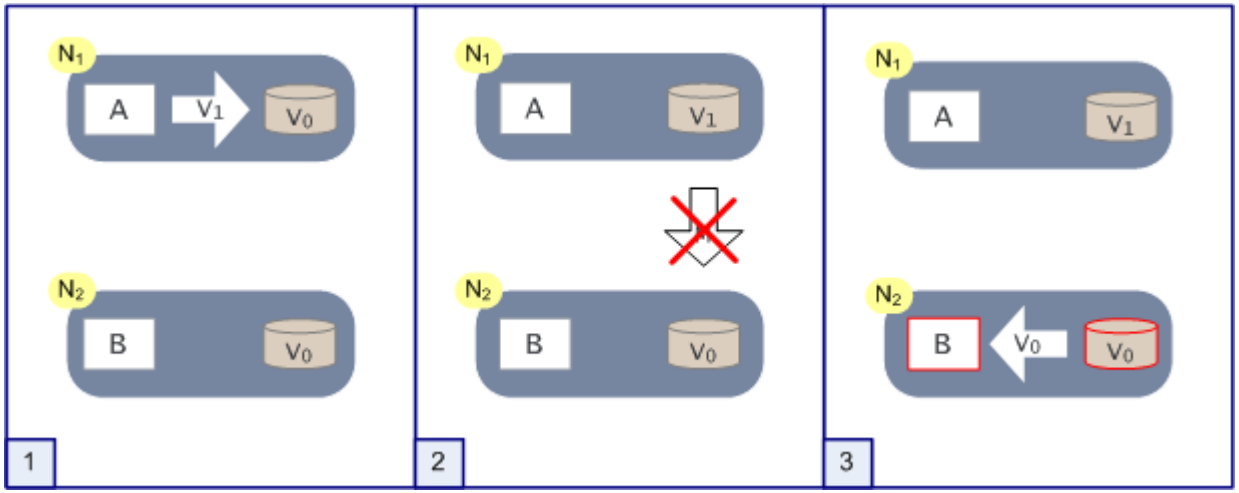


Figure source: http://www.julianbrowne.com/article/viewer/brewers-cap-theorem

Approaches to CAP

1. *Drop Partition Tolerance*
   "put everything (related to the transaction) on one machine (or in one atomically-failing unit like a rack)."
   - There are, of course, significant scaling limits to this.

2. *Drop Availability*
   "the flip side of the drop-partition-tolerance coin. On encountering a partition event, affected services simply wait until data is consistent and therefore remain unavailable during that time"

3. *Drop Consistency*
   "accept that things will become *eventually* consistent"

Approaches to CAP (cont.)

4. *Do BASE: Basically Available, Soft-state, Eventually consistent*
   "the logical opposite of ACID, though it would be quite wrong to imply that any architecture should (or could) be based wholly on one or the other"

5. *Design around it*

Source: http://www.julianbrowne.com/article/viewer/brewers-cap-theorem

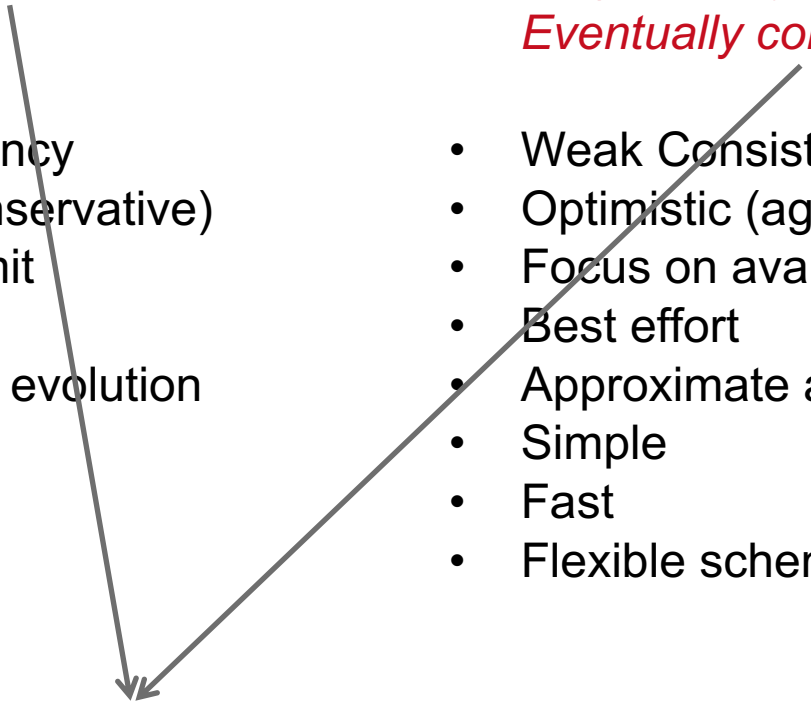
ACID versus BASE

ACID (Atomic, Consistent, Isolated, Durable)*

- Strong Consistency
- Pessimistic (conservative)
- Focus on Commit
- Isolation
- Difficult schema evolution

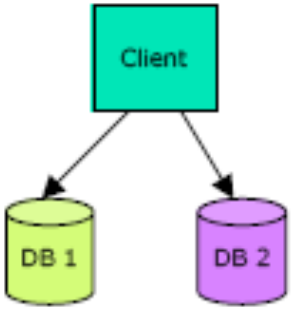
BASE (Basically Available, Soft-State, Eventually consistent)

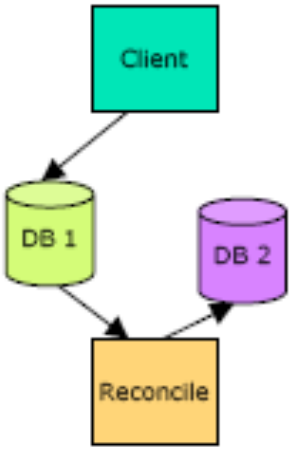
- Weak Consistency
- Optimistic (aggressive)
- Focus on availability
- Best effort
- Approximate answers ok
- Simple
- Fast
- Flexible schema evolution



These are two different “Consistency“ definitions

ACID versus BASE





ACID:

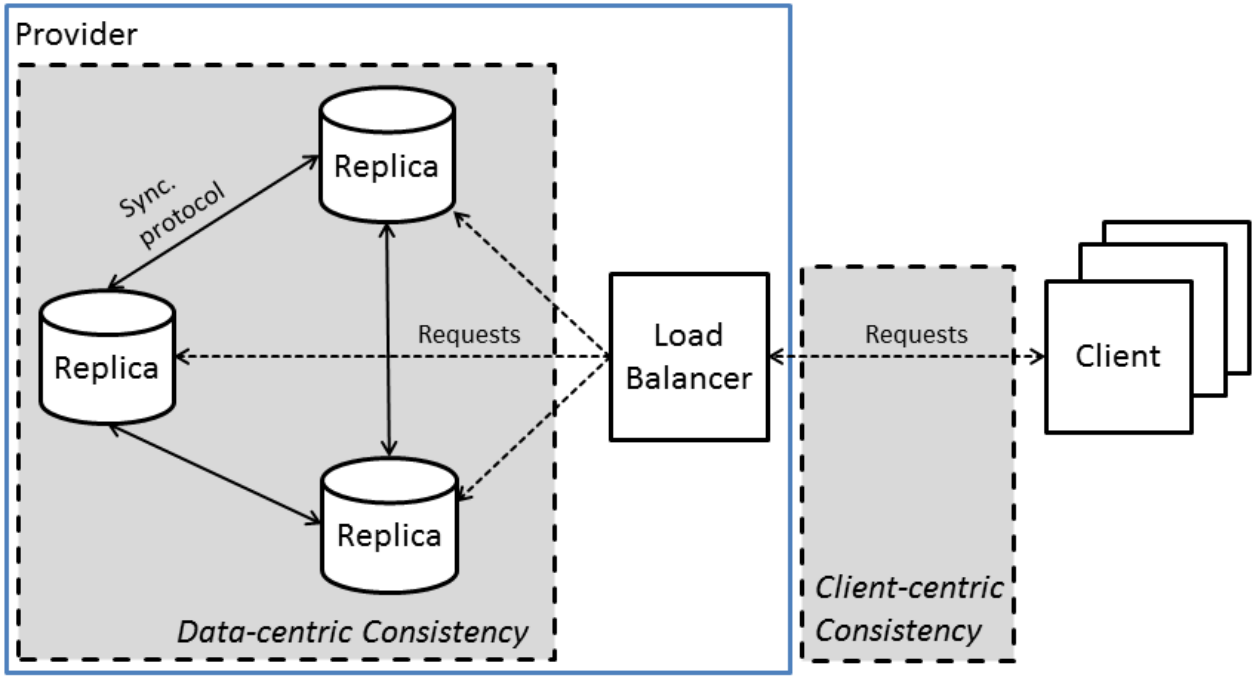
- 2PC commit to DB1 and DB2
- Client availability coupled to both
- Latency on both paths critical

BASE:

- Single commit to one database; reconcile later
- Client only dependent on one database
- Decoupled availability
- Latency tolerant

Consistency Perspectives

Two perspectives:
Data-centric guarantees take a storage provider perspective
Client-centric guarantees take an application perspective



Consistency Dimensions

Two dimensions:

Ordering describes
- The execution order of requests on different replicas (data-centric)
- The ordering of requests visible to clients (client-centric)

Staleness describes
- The difference in time between the commit of an update* and reaching the last replica (data-centric)
- The difference in time between the commit of an update* and the last point in time when the previous value was still returned (client-centric)

* If dirty reads are possible this changes to the start of an update

Data-centric Consistency Models

- Weak Consistency
  No guarantees at all
- Eventual Consistency
  + Replica convergence
- Causal Consistency
  + Identical ordering for causally related requests
- Sequential Consistency
  + Identical ordering for all requests
- Linearizability
  + No staleness

Client-centric Consistency Models

- Monotonic Read Consistency

  Monotonically increasing versions in reads *per client*

- Monotonic Write Consistency

  Updates by the same client serialized in correct order

- Read Your Writes Consistency

  A client will always read his own (or newer) data

- Write Follows Read Consistency

  Updates following a read execute only on replicas that are at least as new as the seen value

MRC + MWC + RYWC + WFRC = Causal Consistency

Eventual Consistency

Technische Universität Berlin

Updates are made on any replica of an object

All updates to an object will *eventually* be applied, but potentially in different orders at different replicas

=> In the absence of updates and failures the system eventually converges towards a consistent state

For many Web applications, eventual consistency is sufficient (thus emphasizing high availability)

Eventual Consistency is easy to implement and easy to scale!

Eventual Consistency, however, may be too weak and inadequate for some Web applications

Technische Universität Berlin

Example: A photo sharing application that allows users to post photos and to control access

- A user‘s record contains a list of photos, and the set of people allowed to view those photos

What if a user wishes to do a sequence of 2 updates to his record:

- Update1: Remove mother from the list of people who can view photos
- Update2: Post spring-break photos

Under the eventual consistency model, Update1 can go to Replica1, while Update2 might go to Replica2. Eventually, the final state of both replicas are guaranteed to be the same. However, the replicas apply the updates in opposite order, which breaks the application‘s contract with the user.

➢ *In Web applications, if is often acceptable to read slightly stale data, but occasionally stronger consistency guarantees than eventual consistency are required by applications*

Source: [PNUTS]

NoSQL Systems

NoSQL = Not Only SQL

Typical properties

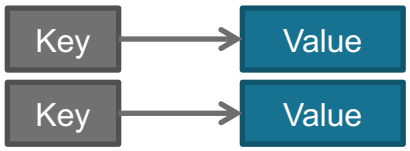
- Focus on Scalability, low Latency, high Availability
- Eventual Consistency
- Different interfaces, data structures and query languages (no SQL!)
- Popular for web applications
- Datastore application co-design
- Distributed deployment

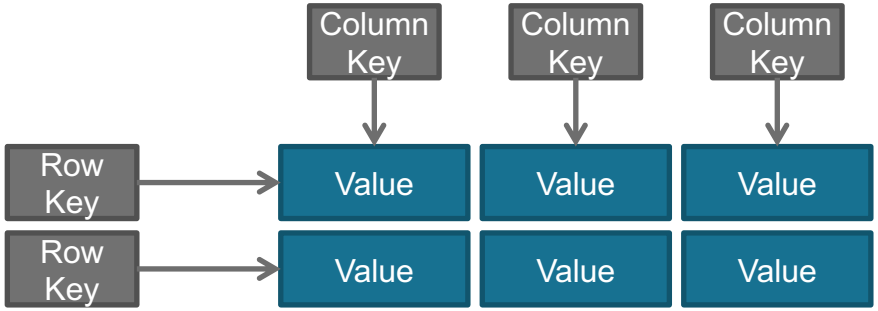
Often open source reimplementations of proprietary systems

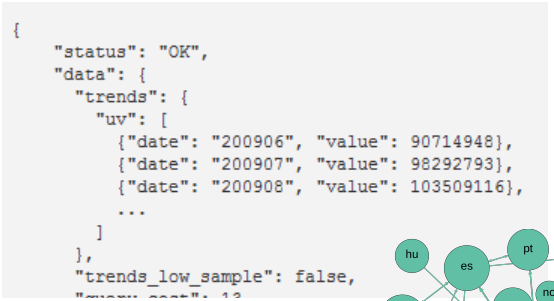
Examples of NoSQL Systems

Technische Universität Berlin

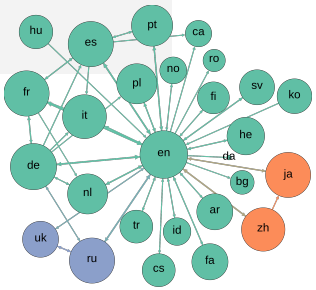
- Key-Value Stores
  - Riak
  - Redis
  - Voldemort
  - …
- Column Stores
  - Cassandra
  - HBase
  - …
- Document Stores
  - MongoDB
  - CouchDB
  - …
- Graph Stores
  - Neo4J







```
{
    "status": "OK",
    "data": {
      "trends": {
        "uv": [
          {"date": "200906", "value": 90714948},
          {"date": "200907", "value": 98292793},
          {"date": "200908", "value": 103509116},
          ...
        ]
      },
      "trends_low_sample": false,
```



ISEngineering
Wirtschaftsinformatik – Information Systems Engineering

NoSQL Classifications

	Performance	Scalability	Flexibility	Complexity	Functionality
Key-Value	high	high	high	none	Variable (none)
Column	high	high	moderate	low	minimal
Document	high	variable (high)	high	low	variable (low)
Graph	variable	variable	high	high	graph theory

Source: Ben Scofield, 2010, @CodeMash http://www.slideshare.net/bscofield/nosql-codemash-2010

Summary

Modern Web applications present unprecedented data management challenges

Scalability, performance (response time), and high availability (fault tolerance) are foremost requirements

Tradeoffs must be balanced; relaxed consistency guarantees are frequently tolerated (cf. CAP and PACELC)

Novel NoSQL systems offer a plethora of interfaces, data structures and query languages

Agenda

1. Intro Cloud Storage

2. Systems in detail
   a) **Dynamo**
   b) GFS, BigTable

3. Other systems
   c) Cassandra
   d) MongoDB

Context: Amazon.com's e-commerce platform

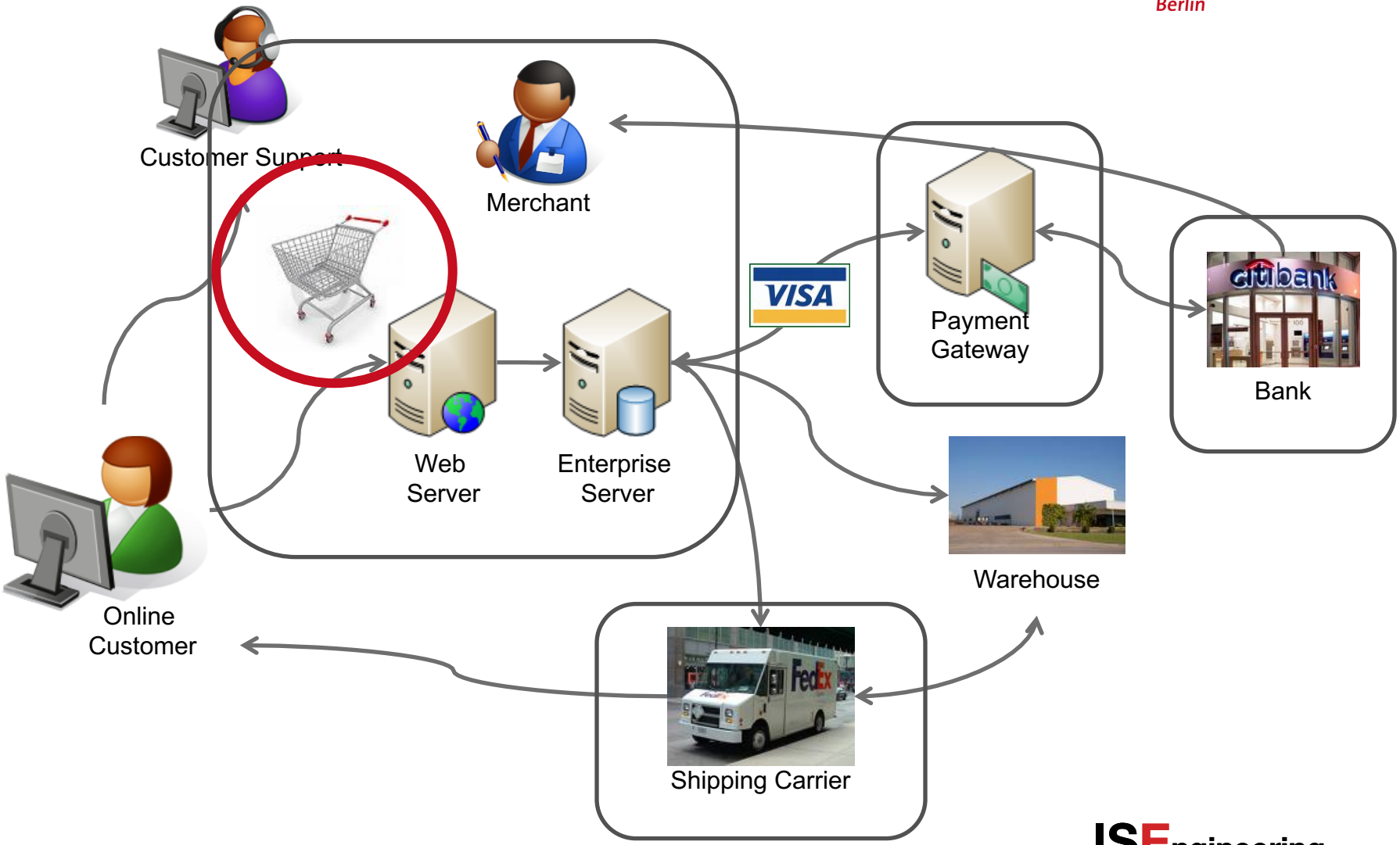
Amazon.com

- Platform providing hundreds of services for many web sites worldwide
- Implemented on top of an infrastructure of tens of thousands of servers and network components
- Tens of millions customers at peak times using tens of thousands of servers located in many data centers around the world; continuous growth
- Dynamo is the underlying storage technology for a number of core services in Amazon's platform

Dealing with failures is the standard mode of operation

- A small but significant number of servers and network components fail at any given time
- Failure handling must be treated as the normal case, without impacting availability or performance

A typical e-commerce scenario



Shopping cart service

Operational requirements include:

- 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“
- Must be able to scale to extreme peak loads efficiently without any downtime, for example, during the busy holiday shopping season

Amazon needs…

- Incremental scalability
- Reliability and the highest possible availability, especially for writes
- Ability to handle huge numbers of concurrent requests
- Ability to run on commodity hardware
- Ability to finetune the system for each service
- Key-value storage for small data items (<1MB)

Here, amazon does not need…

- Complicated queries or complex schemas
- Transactions spanning more than one data item
- ACID guarantees
- Complex security setups (Dynamo is used internally)

Learning objectives

Technische Universität Berlin

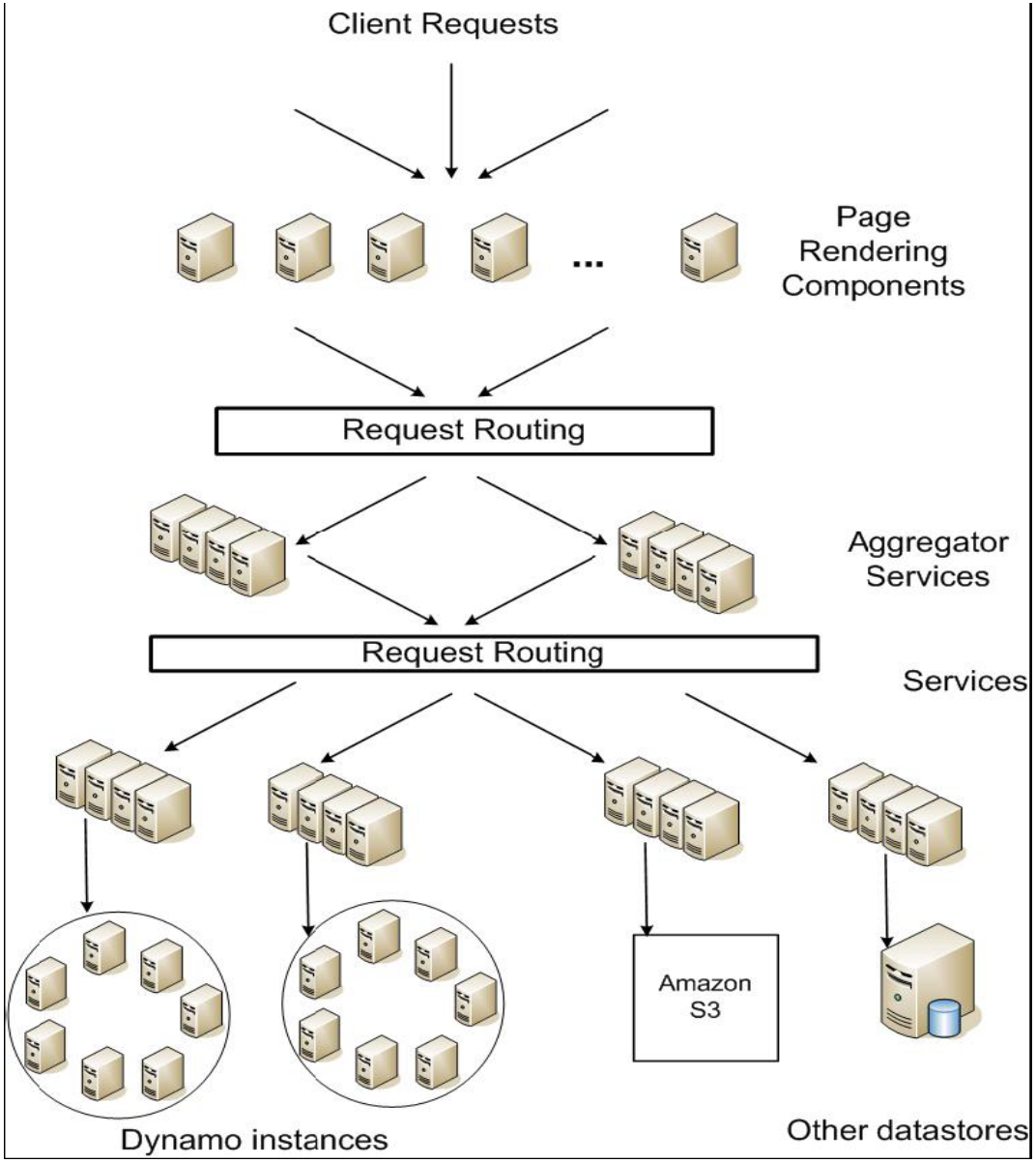
Reliability and scalability of a system depend on how its application state is managed

There are tradeoffs between availability, consistency, cost-effectiveness and performance

A synthesis of well-known techniques can be used to achieve scalability and availability; different techniques are combined to implement an eventually consistent storage system

Reference:
G. DeCandia et.al.
“Dynamo: Amazon’s Highly Available Key-value Store”, in: Procs. SOSP’07

SOA


Client Requests
Page Rendering Components
Request Routing
Aggregator Services
Request Routing
Services
Amazon S3
Dynamo instances
Other datastores

Data replication

Technische Universität Berlin

Strong consistency and high data availability cannot be achieved simultaneously

Optimistic replication techniques can be used to increase availability: changes are allowed to propagate to replicas in the background, and concurrent, disconnected work is tolerated

Requires conflict resolution, inducing two problems: when to resolve conflicts, who resolves conflicts

Conflict resolution

An important design consideration is when to perform the process of resolving update conflicts: during reads or writes

- Many traditional systems reject writes if not all replicas can be reached at a given time
- Dynamo implements an „always writable“ design store (cf. example of shopping cart service) – thus, the complexity of conflict resolution here is pushed to the reads

Who performs the process of conflict resolution: the data store or the application?

- In case of data store, simple policies are used (such as „last write wins“)
- Otherwise, application-specific conflict resolution methods can be implemented (cf. once more the shopping cart example)

Further Design Considerations in Dynamo

Technische Universität Berlin

Incremental scalability

- System should be able to scale-out one storage host (=node) at a time, with minimal impact on the system itself and its operators

Symmetry and Decentralization

- Decentralized peer-to-peer techniques are favored over centralized control
- Every node has the same set of responsibilities as its peers
- Symmetry simplifies the process of system provisioning and maintenance, and leads to a more scalable and available system

Heterogeneity

- Work distribution must be proportional to the capabilities of the individual servers, assuming that the infrastructure always is heterogeneous

Techniques used in Dynamo

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.

Techniques used in Dynamo

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.

Dynamo System Interface

Stored objects are associated with a key

A simple interface exposing get() and put() operations is used to access the object

- get(key) locates all object replicas associated with the key and returns a single object or a list of objects with conflicting versions along with a context
- put(key, context, object) determines where replicas of the object should be placed based on the associated key, and writes the replicas to disk
- The context encodes system metadata, such as the version of the object

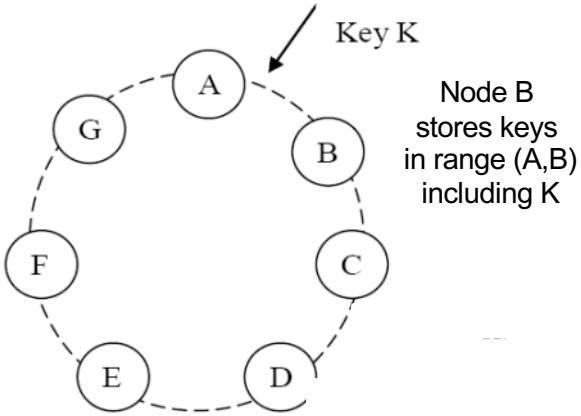
Both the key and the object are treated as an opaque array of bytes. A MD5 hash is applied on the key to generate a 128-bit identifier, which is used to determine the storage nodes that are responsible for serving the key.

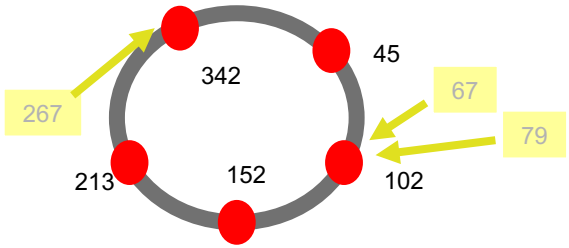
Consistent hashing

In consistent hashing, the output range of a hash function is treated as a fixed circular space or “ring” (i.e. the largest hash value wraps around to the smallest)

Each node in the ring is assigned a random value which represents its “position” on the ring

Each object identified by a key is assigned to a node by hashing the object’s key to yield its position on the ring, and then walking the ring clockwise to find the first node with a position larger than the object’s position





Consistent hashing cont.

Technische Universität Berlin

Pros/Cons

- Departure or arrival of a node only affects its immediate neighbors; other nodes remain unaffected
- Random position assignment leads to non-uniform load distribution
- Heterogeneity in the performance of nodes is not considered

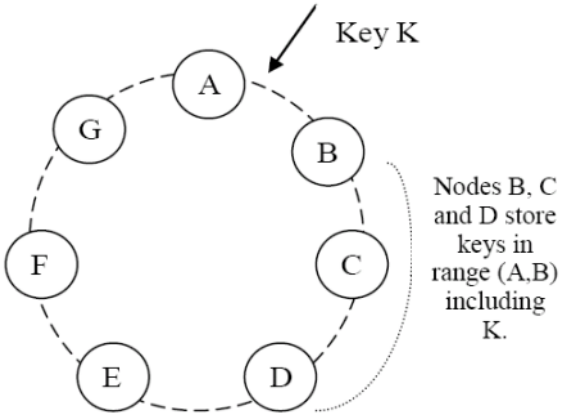
Variant used in Dynamo

- Each node gets assigned to multiple points in the ring; concept of „virtual nodes“
- A virtual node looks like a single node
- Each node can be responsible for more than one virtual node
- When a new node is added to the system, it is assigned multiple positions („tokens“) in the ring

Replication

Data is replicated an N hosts (N is configured per Dynamo instance)

Each key is assigned to one coordinator node; the coordinator node is responsible for replicating the data items within its range at the N-1 clockwise successor nodes in the ring
For example, node B as coordinator node replicates the key K at nodes C and D



The list of nodes responsible for storing a particular key is called the preference list

Techniques used in Dynamo

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.

Data Versioning

With eventual consistency, updates are allowed to propagate to all replicas asynchronously

- Under certain failure scenarios, updates may not arrive at all replicas for an extended period of time
- A put() call may return to its caller before the update has been applied at all the replicas
- A get() call may return many versions of the same object

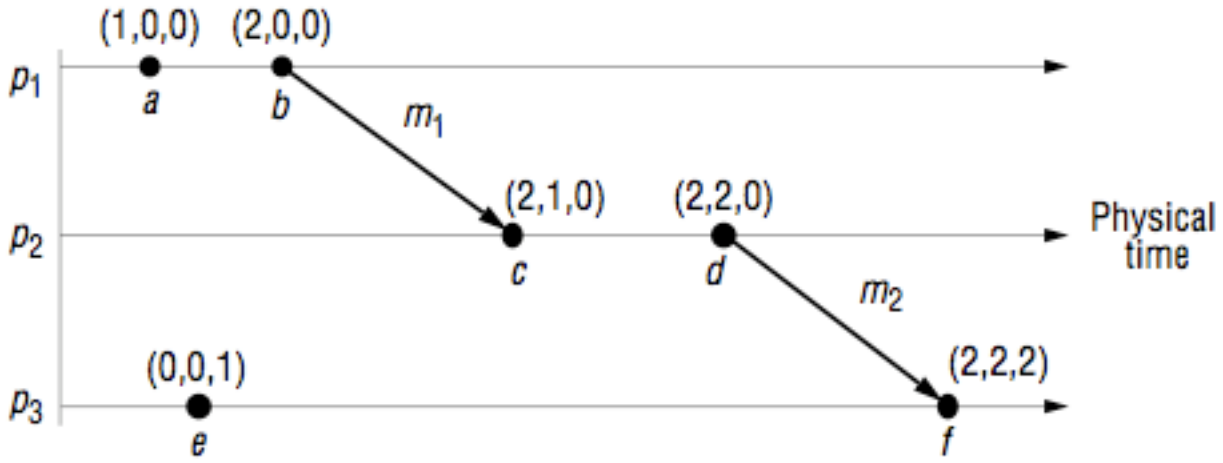
Some applications can tolerate such inconsistencies

- For example: “add to cart” operation in the shopping cart example
- Note that “add to cart” and “delete item from cart” are both put() requests in Dynamo

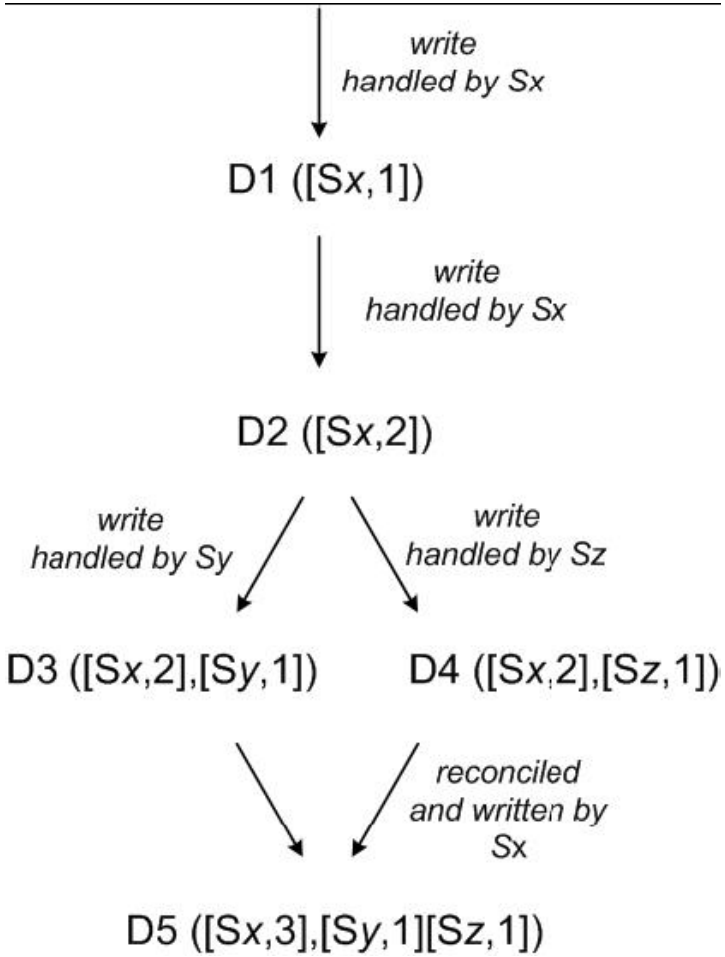
Therefore, *different object versions may exist, which the system may need to reconcile*

Vector clocks

A vector clock is a list of (node, counter) pairs
One vector clock is associated with every version of every object
Using vector clocks, one can determine whether two versions of an object are on parallel branches or have a causal ordering



Vector clock example



Techniques used in Dynamo

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.

Classical Quorum configuration

(N,R,W)

- N replicas
- R: mininum number of nodes that must participate in a successful read operation
- W: mininum number of nodes that must participate in a successful write operation
- In Dynamo, a typical (N,R,W) setting is (3, 2, 2)

R+W > N and W > N/2 yields a quorum system

Not required in Dynamo!

Sloppy Quorum and Hinted Handoffs

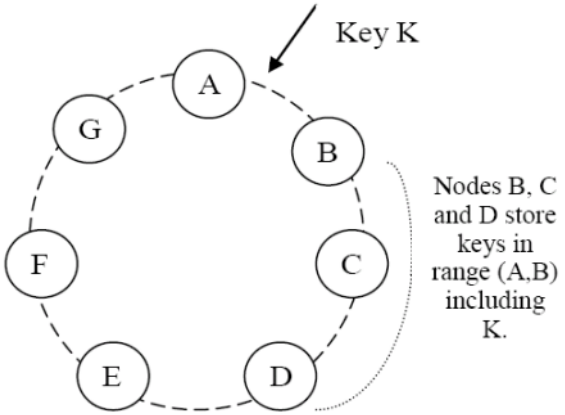
Instead of storing on the first N nodes, only the first N healthy nodes from the preference list are considered: „sloppy quorum“

If a node is temporarily not available the data is stored on a substitute node and a „hinted handoff“ is created

Example: Hinted Handoffs

Assume N = 3. When A is temporarily down or unreachable during a write operation, send replica to D.
D will have a hint in the replica's metadata that the intended recipient was A

Nodes that receive hinted replicas will keep them in a separate local database that is scanned periodically
When A is recovered, the replica will be delivered to A



Techniques used in Dynamo

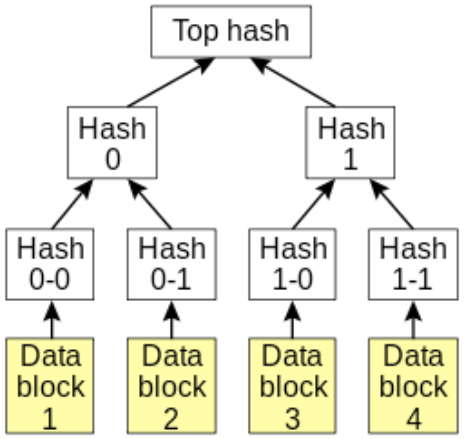
Problem	Technique	Advantage
Partitioning	Consistent Hashing	Incremental Scalability
High Availability for writes	Vector clocks with reconciliation during reads	Version size is decoupled from update rates.
Handling temporary failures	Sloppy Quorum and hinted handoff	Provides high availability and durability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

Handling permanent failures: Replica synchronization

Problem: Hinted replicas may become unavailable before they can be returned to the original replica node

Solution: Anti-entropy protocol using Merkle trees

- Each node maintains a separate Merkle tree for each key range that it hosts
- Merkle trees are hash trees where leaves are hashes of keys, and interior nodes are the secure hashes of their children;
  Merkle trees are an efficient summary technique
- Only the root of the Merkle tree is exchanged and compared first to detect inconsistencies among replicas;
  tree traversal follows only when needed



Requires refined partitioning scheme for uniform load distribution – see reference paper for details

Techniques used in Dynamo

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.

Ring membership

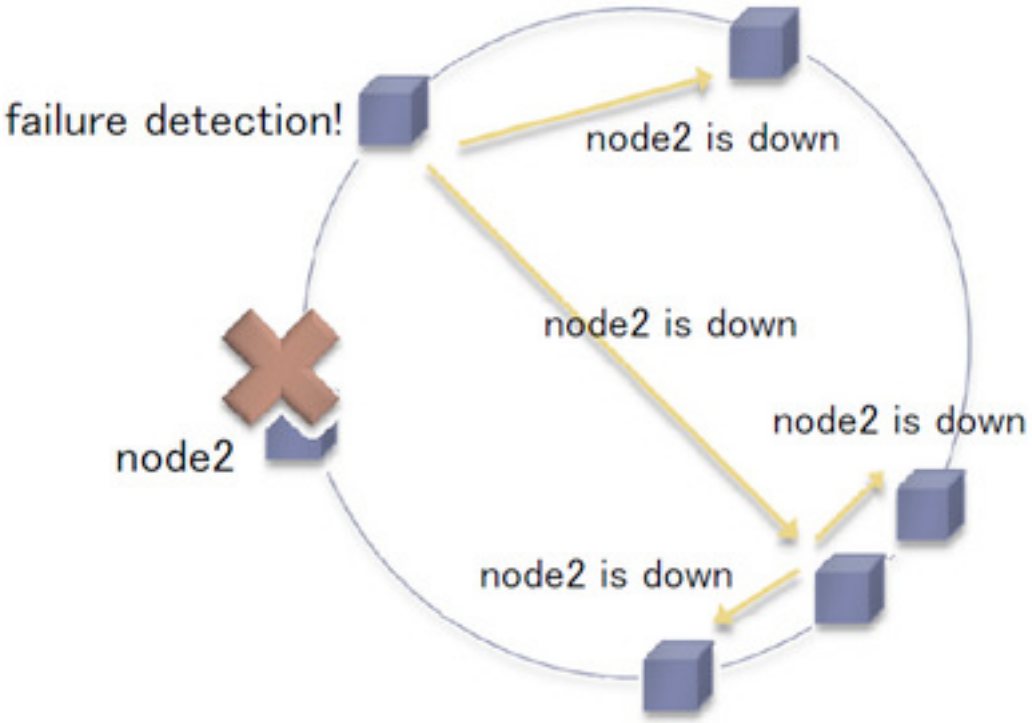
Explicit mechanism (command line tool) to initiate addition or removal of nodes to/from a ring

The node that serves the request writes the membership change and its time to persistent store

A gossip-based protocol propagates membership changes and maintains an eventually consistent view of membership

- Each node contacts a peer chosen at random every second and the nodes reconcile their persisted membership change histories
- Partitioning and placement information also propagate via the gossip-based protocol and each node is aware of the token ranges handled by its peers

Gossip protocols illustrated



Summary: Amazon‘s Dynamo

Technische Universität Berlin

Reliability at massive scale is a key challenge for service providers

Reliability and scalability of a system depend on how its application state is managed

There are tradeoffs between availability, consistency, cost-effectiveness and performance

Dynamo is an example of a decentralized, incrementally scalable, highly available architecture that can handle server and network failures

Dynamo uses a synthesis of well-known distributed computing techniques to implement an eventually consistent storage system

- Consistent hashing and vector clocks
- Sloppy quorum, hinted handoffs, and replica synchronisation
- Gossip-based membership and failure detection

Agenda

1. Intro Cloud Storage

2. Systems in detail
   a) Dynamo
   b) GFS, BigTable

3. Other systems
   c) Cassandra
   d) MongoDB

The Google File System (GFS)

Massive distributed file system

GFS design is driven by key observations of Google's application workload and technological environment

- Re-examination of traditional file system solutions and in parts radically different design
- *Designing for a specific use case* avoids many of the difficulties traditionally associated with replication

Reference:
S. Ghemawat, H. Gobioff, S.-T. Leung:
The Google File System, Procs. SOSP'03,
ACM, 2003

GFS requirements and assumptions

Technische Universität Berlin

As in Dynamo, the GFS considers failures to be the norm rather than the exception

GFS consists of hundreds of storage machines built from inexpensive commodity parts

Common failures: application bugs, OS bugs, human errors, failures of disks, memory, connectors, networking, and power supply

Therefore, constant monitoring, error detection, fault tolerance, and automatic recovery must be integral to the system

GFS requirements and assumptions

Large files, millions of files

- Each file typically contains many app objects such as web documents
- Multi-GB files are common

Most files are mutated by *appending new data rather than overwriting existing data*

- Random writes practically non-existent
- Once written, the files are only read, often only sequentially
- Examples: large repositories for data analysis, continuously generated data streams, archival data, intermediate program results, etc

GFS requirements and assumptions

Applications are written for GFS: Co-design of applications and file benefits overall system by increasing flexibility

- Relaxed GFS consistency model
- Atomic append operation: concurrent file append by multiple clients with minimal synchronization overhead

Specifics of the Google Workload

Technische Universität Berlin

Workload primarily consists of two kinds of reads:

- Large streaming reads: individual operations read hundreds of KBs; successive operations from the same client often read through a contiguous region of a file
- Small random reads: reads a few KBs at some arbitrary offset

Workloads have many large, sequential writes that append data to files

Files are often used as producer-consumer queues for many-way merging:
Hundred of client (machines) concurrently append to a file
High sustained bandwidth (processing data) more important than low latency (response time)

GFS Interface

Files are organized hierarchically in directories and identified by pathnames

Usual operations: create, delete, open, close, read, write
Moreover:

- Snapshot: creates a copy of a file of directory tree
- Record append: append data to a file (atomic, concurrent appends supported)

GFS Architecture

A GFS cluster consists of

- A single master
- Multiple chunkservers

and is accessed by multiple clients

Each is a commodity Linux machine running a user-level server process

Files are divided into fixed-size chunks

- Each chunk has a handle (an ID assigned by the master)
- Chunkservers store chunks on local disks as Linux files
- Each chunk is replicated on multiple chunkservers (three machines by default)

GFS Architecture

Technische Universität Berlin

Master maintains all FS metadata: namespace, access control information, mapping from files to chunks, current locations of chunks

Also controls system-wide activities such as chunk lease management, garbage collection of orphaned chunks, and chunk migration between servers

Master periodically communicates with each chunkserver in HeartBeat messages to give it instructions and to collect its state

GFS Architecture

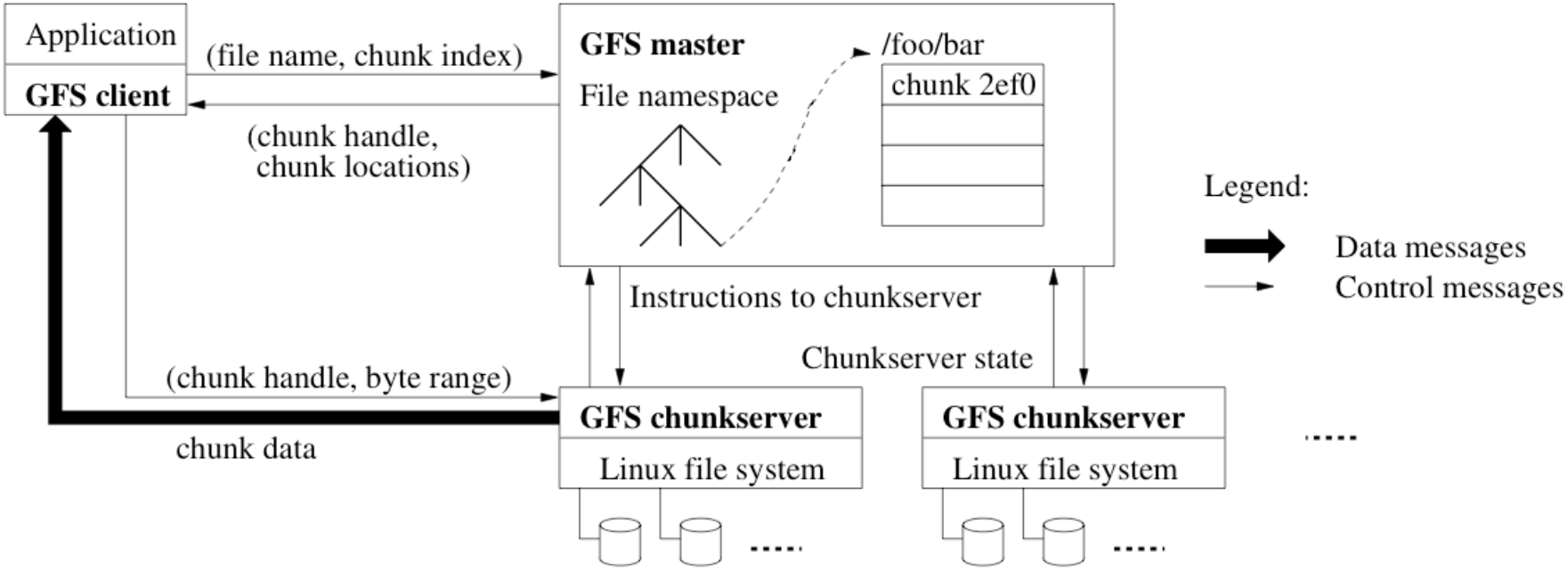


Figure 1: GFS Architecture

Client/GFS Interaction

Client:

- Translates file name and byte offset specified by the application into a chunk index within the file
- Sends request to master, containing file name and chunk index

Master:

- Replies with the corresponding chunk handle and locations of the replicas

Client:

- Caches this information
- Contacts one of the replicas for data

Single Master

Clients interact with master for metadata operations, but with chunkservers for data operations

- Single master simplifies design, centralizes global knowledge for chunk placement and replication decisions
- Master involvement in reads and writes however must be minimized so that it does not become a bottleneck

Neither client nor chunkserver caches file data (simplifies client and overall system by eliminating cache coherence issues)

- Client caches chunk handle and locations of chunkservers

Chunk Size

Chunk size is 64 MB (much larger than typical FL block sizes)

- Large chunk size reduces number of interactions between client and master
- Reduces network overhead by keeping a persistent TCP connection to the chunkserver over an extended period of time
- Reduces size of metadata stored on the master
- Potential disadvantage: chunks for small files may become hot spots (requires higher replication factor)

Each chunk replica is stored as a plain Linux file and is extended as needed

Metadata

Metadata – data about the data:

- File and chunk namespaces
- Mapping from files (file names) to chunks (handles/IDs)
- Locations of each chunk's replicas

All metadata is kept in the master's memory

< 64 Byte per 64 MB chunk

Namespaces and file-to-chunk mappings are also kept persistent in an *operation log* stored on the master's local disk (+ replication on remote machines)

Chunk locations are *kept in memory* only

- They will be lost during a crash
- The master asks chunk servers about their chunks at startup – builds a table of chunk locations

Why Keep Metadata In Memory?

To keep master operations fast

Master can easily and efficiently periodically scan its entire state in the background, for purposes of:

- Chunk garbage collection
- Re-replication (in case of chunkserver failures)
- Chunk migration (for load balancing)

Capacity of the system (number of chunks) is limited by how much memory the master has

- Has not been a problem for Google in practice
- Master maintains <64bytes of metadata for each 64MB chunk

“The cost of adding extra memory to the master is a small price to pay for the simplicity, reliability, performance, and flexibility gained”

Why Not Keep Chunk Locations Persistent?

Reason: Simplicity

- Eliminates the need to keep master and chunkservers synchronized
- Synchronization would be needed when chunkservers:
  - Join and leave the cluster
  - Change names
  - Fail and restart chunk location – which chunk server has a replica of a given chunk

Master polls chunkservers for chunk locations on startup
Thereafter, master keeps itself up-to-date by

- Controlling all initial chunk placement, migration and re-replication
- Monitoring chunkserver status with regular HeartBeat messages

Operation Log

Contains a historical record of critical metadata changes
Serves as logical time line that defines the order of concurrent operations

- Files and chunks (incl. versions) are uniquely identified by the logical times at which they were created

Log is used for recovery – the master replays it in the event of failures

- Checkpoint is a compact B-tree data structure that can be loaded into memory
- Used for namespace lookup without extra parsing

Checkpointing (switching to a new log file) can be done on the background (takes a minute for a cluster of a few million files)
Operation log is replicated on multiple remote machines

System Interactions: Data Mutations

A mutation is an operation that changes the contents or metadata of a chunk
Each mutation is performed at all the chunk‘s replicas

- Writes: causes data to be written at an application-specified file offfset
- Record appends: causes data (the „record“) to be appended atomically at least once even in the presence of concurrent mutations, at an offset of GFS‘s choosing
  - The offset is returned to the client and marks the beginning of a defined region that contains the record

GFS Consistency Model

State of a file region after a data mutation depends on:

- Type of mutation
- Sucess or failure of mutation
- Concurrent mutations

A consistent file region: all clients always see the same data, regardless of replica

A defined (consistent) file region: all clients can see what the mutation has written in its entirety

An undefined (consistent) file region: all clients see the same data, but it may not reflect what any one mutation has written

An inconsistent file region: different clients may see different data at different times

Implications of Loose Data Consistency for Applications

Technische Universität Berlin

Replicas are not guaranteed to be identical (!)

In fact, GFS does not even guarantee eventual consistency – data can stay inconsistent for eternity (!)

Applications may see inconsistent data (data is different on different replicas) and may see data from partially completed writes (undefined file region)

Applications must be designed to handle loose data consistency

- Rely on appends rather than overwrites
- Use checkpointing
- Write self-validating, self-identifying records (e.g., check sums)

Leases and Mutation Order

Technische Universität Berlin

Leases are used to maintain a consistent mutation order across replicas

Master grants a chunk lease to one replica, called the primary (initial timeout 60 sec.)

Primary picks a serial order for all mutations to the chunk (assigning consecutive serial numbers); all other replicas will later on follow this order

Leases can be renewed/extended using periodic HeartBeat messages between the master and the chunkservers

Leases and Mutation Order

1. Client asks master for primary and secondary replicas for a chunk
2. Master replies with the identity of primary and the locations of the secondary replicas
3. Client pushes data to all replicas (in any order)
4. Once all replicas acknowledged receipt of data, the client sends a write request to the primary; the primary then applies the mutation in serial number order
5. Primary forwards write request to all secondary replicas
6. All secondaries reply to the primary indicating completion
7. Primary replies to the client (may report errors at secondaries, which the client must handle)

Write Control and Data Flow

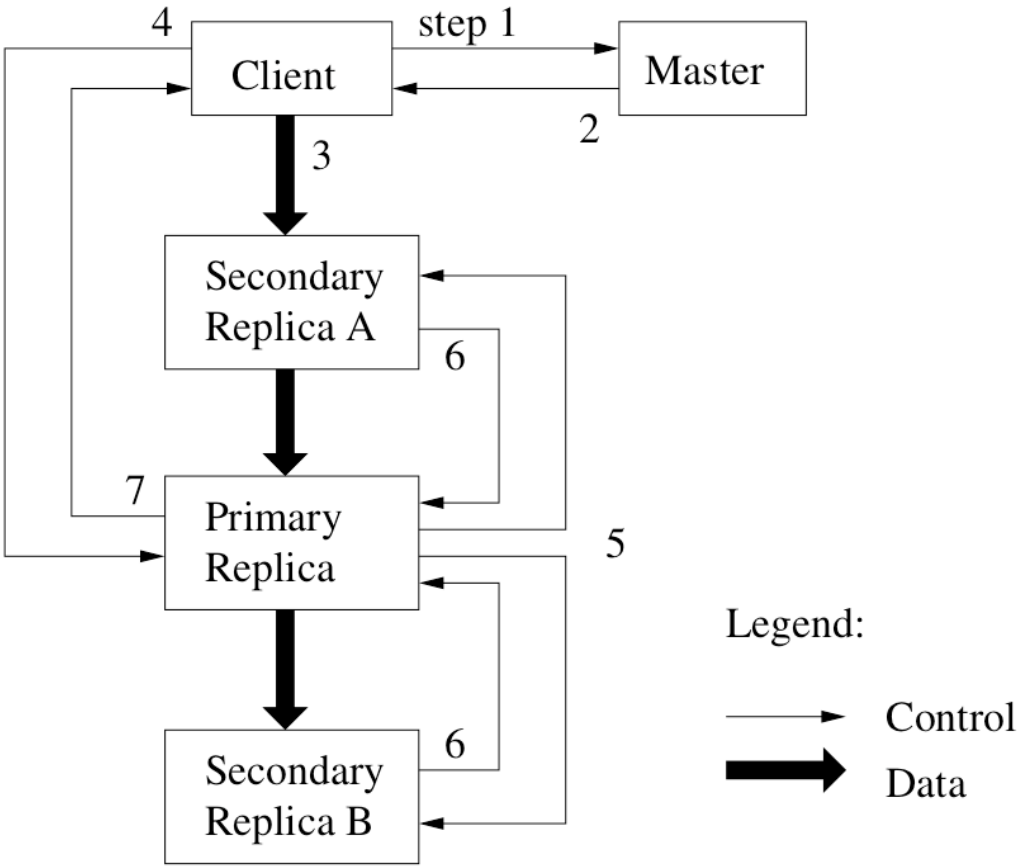


Figure 2: Write Control and Data Flow

Decoupling data and control flow

Technische Universität Berlin

Notice: Control flows from client to primary to secondaries, data is pushed linearly along a chain of chunkservers in a pipelined fashion

Goal is to fully utilize each machine‘s network bandwidth, avoid network bottlenecks, and minimize latency

Atomic Record Append

Traditional, concurrent writes to the same region are not serializable
In a record append, however, the client only specifies the data

Record appends follows the control flow as for writes, except:

- The client pushes the data to all replicas of the last chunk of the file
- The client sends its request to the primary
- Primary checks if append to current chunk exceeds maximum size (64MB)
  - If so, chunk is padded to max. size, tells secondaries to do the same, and the client to retry the append on the next chunk
  - If the request fits in current last chunk (common case):
    - The primary appends the data to replica,
    - tells secondaries to do same at same byte offset,
    - replies success to the client

Failure Handling During Updates

If a write fails at the primary:

- The primary may report failure to the client – the client will retry
- If the primary does not respond, the client retries from Step 1 by contacting the master

If a write succeeds at the primary, but fails at several replicas

- The client retries several times (Steps 3-7)

Non-Identical Replicas

Technische Universität Berlin

Because of failed and retried record appends, replicas may be non-identical

- Some replicas may have duplicate records (because of failed and retried appends)
- Some replicas may have padded file space (empty space filled with junk) – if the master chooses record offset higher than the first available offset at a replica

Clients must deal with it: they write self-identifying records so they can distinguish valid data from junk

If clients cannot tolerate duplicates, they must insert version numbers in records

GFS pushes complexity to the client; without this, complex failure recovery scheme would need to be in place

Further Issues

- Load Balancing: Maximize data availability and reliability; maximize network bandwidth utilization
- Creation (initial replica placement)
  - Select chunkservers with low disk space utilization
  - Limit the number of recent creations on each chunkserver – recent creations mean heavy write traffic
  - Spread replicas across racks
- Re-replication
  - When the number of replicas falls below the replication target
  - When a chunkserver becomes unavailable
  - When a replica becomes corrupted
  - A new replica is copied directly from an existing one

Further Issues: Fault Tolerance

Two simple, yet highly effective strategies:
fast recovery, and replication

Fast recovery

- No distinction between normal and abnormal termination
- Servers are routinely shut down by killing a server process
- Servers are designed for fast recovery – all state can be recovered from the log

Chunk replication

- As discussed

Master replication

- See next slide

Master Replication

Master state is replicated on multiple machines
What is replicated: operation logs and checkpoints
A modification is considered successful only after it has been logged on all master replicas

A single master is in charge; if it fails, it restarts almost instantaneously
If a machine fails and the master cannot restart itself, a failure detector outside GFS starts a new master with a replicated operation log (no master election)
"Shadow" masters provide read-only access to the FS even when the primary master is down

- Shadows, not mirrors, in that they may lag the primary slightly

Data Integrity

Failures happen and they may cause data corruption
How to detect corrupt replicas?

- Comparing replicas across chunkservers does not work in GFS – divergent replicas are legal
- Solution: Each chunkserver verifies its own replicas using checksums (each chunk is broken up into 64KB blocks; each block has a 32bit checksum)

Checksums are kept in memory and stored persistently in the log

- Small effect on read performance
- Write performance: checksum computation optimized for appends

If corruption is detected, the master creates new replicas using data from correct chunks
During idle periods chunkservers scan inactive chunks for corruption

Diagnostic Tools: Detecting Stale Replicas

A replica may become stale if it misses a modification while the chunkserver was down

Each chunk has a version number; version numbers are used to detect stale replicas

A stale replica will never be given to the client as a chunk location, and will never participate in mutation

A client may read from a stale replica (because the client cached the location)

- But this window is limited, because cache entries time out

GFS servers generate diagnostic logs that record significant events (such as chunkservers going up and down) and all requests and replies (except for the file data)

GFS Summary

File system specifically designed for a specific use case (Google applications)
  – and vice versa

Simplicity is a key objective in the design

GFS is actively used e.g. to support Google's search service
- Availability and recoverability
- High throughput by decoupling control and data
- Supports massive data sets and concurrent appends

Semantics not transparent to apps
  – Must verify file contents to avoid inconsistent regions, repeated appends (at-least-once semantics)

Performance not good for all apps
  – Assumes read-once, write-once workload (with no client caching)

Exercises

1. Can you name similarities between GFS and Dynamo?
2. Can you name differences?
3. Design GFS over Dynamo
   - A system layer that presents the GFS interface on top of Dynamo's key-value store
   - How would you write a GFS-over-Dynamo application?
4. Design Dynamo over GFS
   - as above, the other way round
5. Discussion
   - Does 3) respective 4) make sense?
   - What system properties make this a fundamentally good/bad idea?

Agenda

1. Intro Cloud Storage

2. Systems in detail
   a) Dynamo
   b) GFS, **BigTable**

3. Other systems
   c) Cassandra
   d) MongoDB

BigTable – Motivation

GFS only provides means to store unstructured data – however, some applications require a certain amount of structure

GFS was made for large files – some applications need to store small data items

All the positive aspects (scalability, high availability, high performance etc.) should be kept

Reference:
F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, R. Gruber: Bigtable: A distributed storage system for structured data, OSDI, 2006.

What is Bigtable?

Technische Universität Berlin

Building on GFS Bigtable provides additional structure, query semantics and support for small files

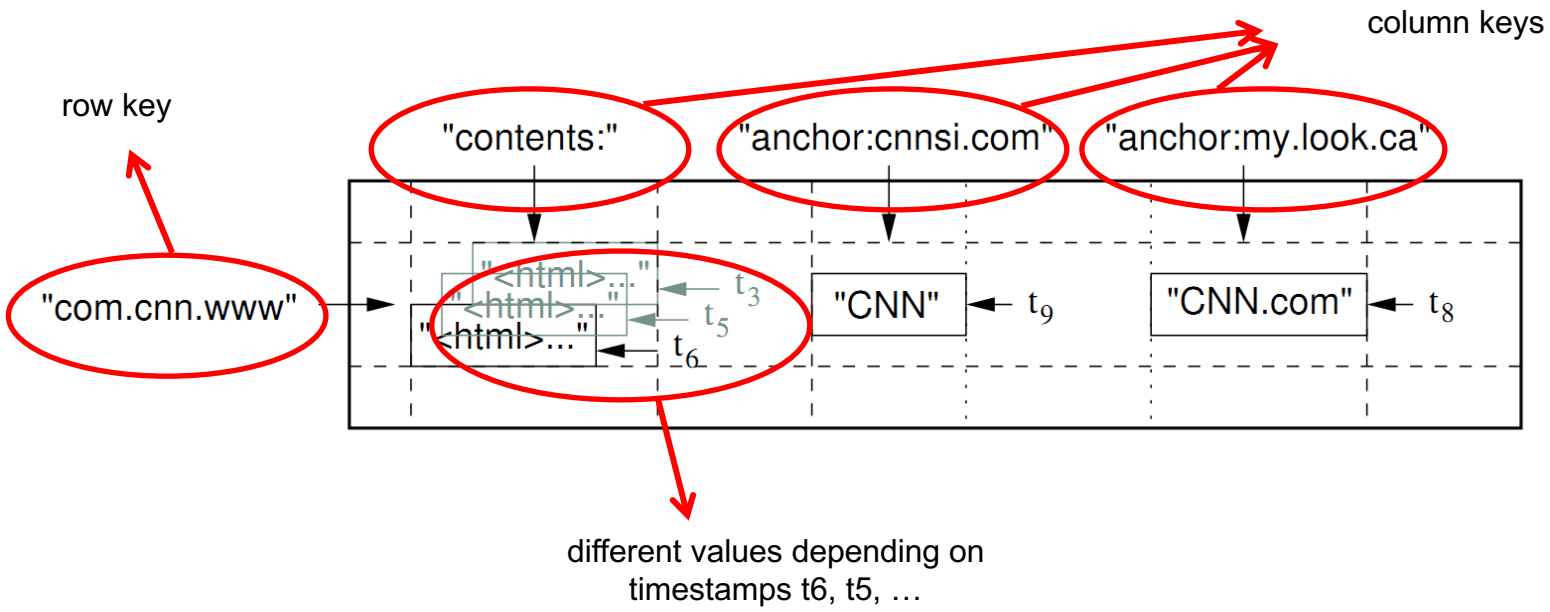
Chang et al. phrase it as:

> A Bigtable is a sparse, distributed, persistent multi-dimensional sorted map. The map is indexed by a row key, column key, and a timestamp; each value in the map is an uninterpreted array of bytes.

Used by more than 60 Google projects already in 2006, including MapReduce, Earth, Analytics and Orkut

Data Model

(row:string, column:string, time:int64) → value:string



Structure and Query model

A table contains a number of rows and is broken down into tablets which each contain a subset of the rows

Tablets are the unit of distribution and load balancing

Rows are sorted lexicographically by row key

- → subsequent row keys are within a tablet
- → Allows efficient range queries for small numbers of rows

Operations: Read, write and delete items + batch writes and Sawzall scripts running on (!) tablet server

Supports single-row transactions

Elements

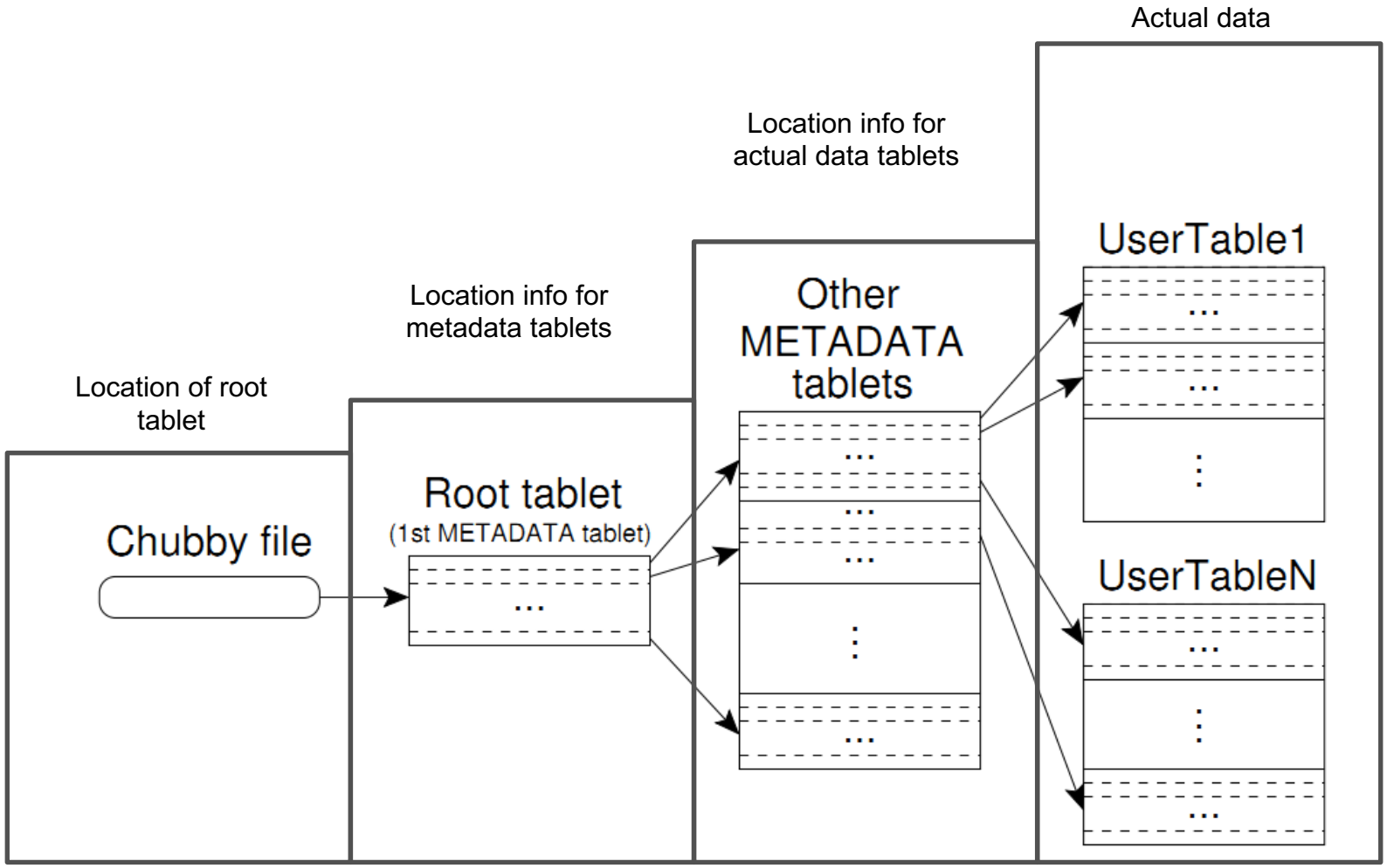
A client-side library for access to Bigtable
A single master server

- Handles tablet locations, load balancing, schema changes, garbage collection, …

A number of tablet servers

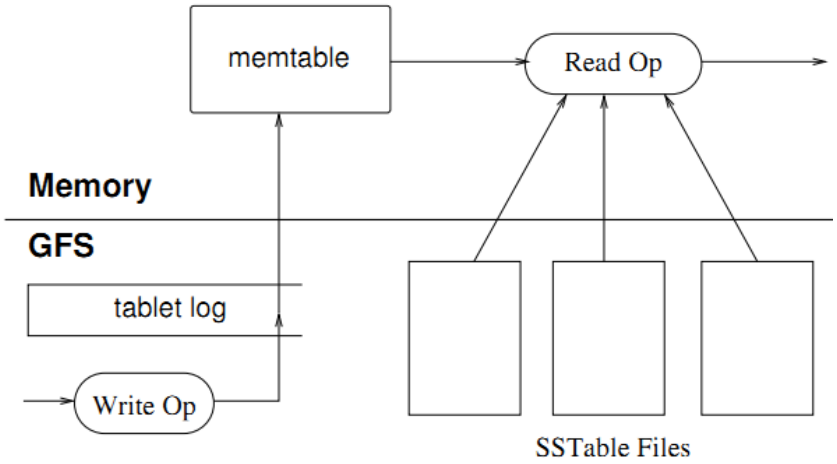
- Each server stores a number of tablets (10-1000) at approx. 100-200MB each
- Handles read and write requests
- Splits tablets that have grown too large

Tablet location



Local persistence

Tablet servers store updates in commit logs written in so-called SSTables
Fresh updates are kept in memory (memtable), old updates are stored in GFS
Minor compactions flush memtable into new SSTable
Major compaction merge SSTables into just one new SSTable



Summary

Bigtable enhances GFS with a structured data model and more sophisticated query functionality

It adds support for small files while still maintaining system properties like

- High availability
- High throughput
- Low latency
- Slightly sub-linear scalability

It abstracts the user from GFS hassles with asserting that data is consistent and defined

Agenda

1. Intro Cloud Storage

2. Systems in detail
   a) Dynamo
   b) GFS, BigTable

3. Other systems
   c) Cassandra
   d) MongoDB

Cassandra

Technische Universität Berlin

TOPICS

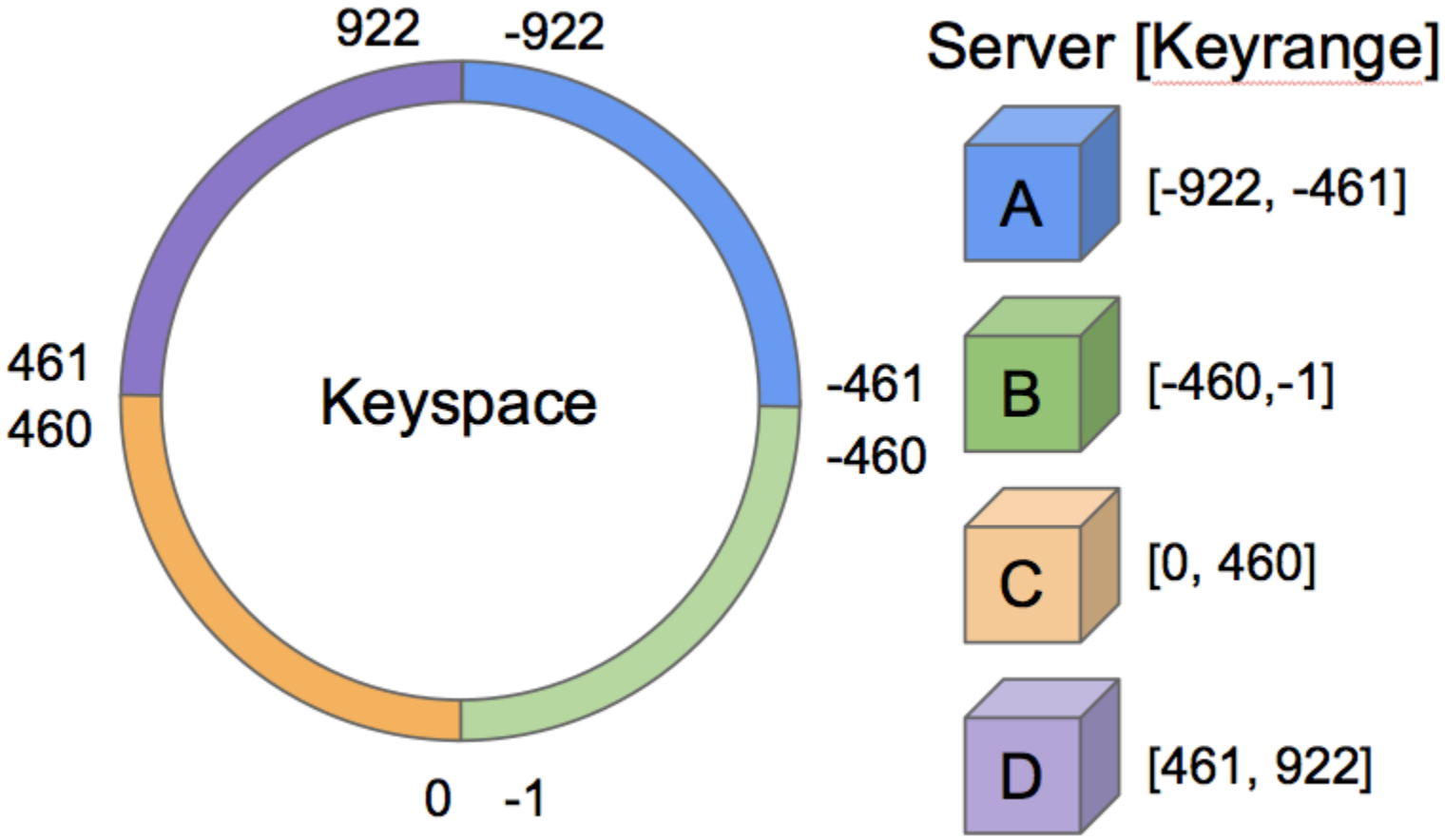
1. Cassandra Architecture (lecture & exercise)
2. Cassandra Data Model (exercise)
3. Cassandra Query Language (exercise)

Reference:
Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. SIGOPS Oper. Syst. Rev. 44, 2 (April 2010), 35-40.

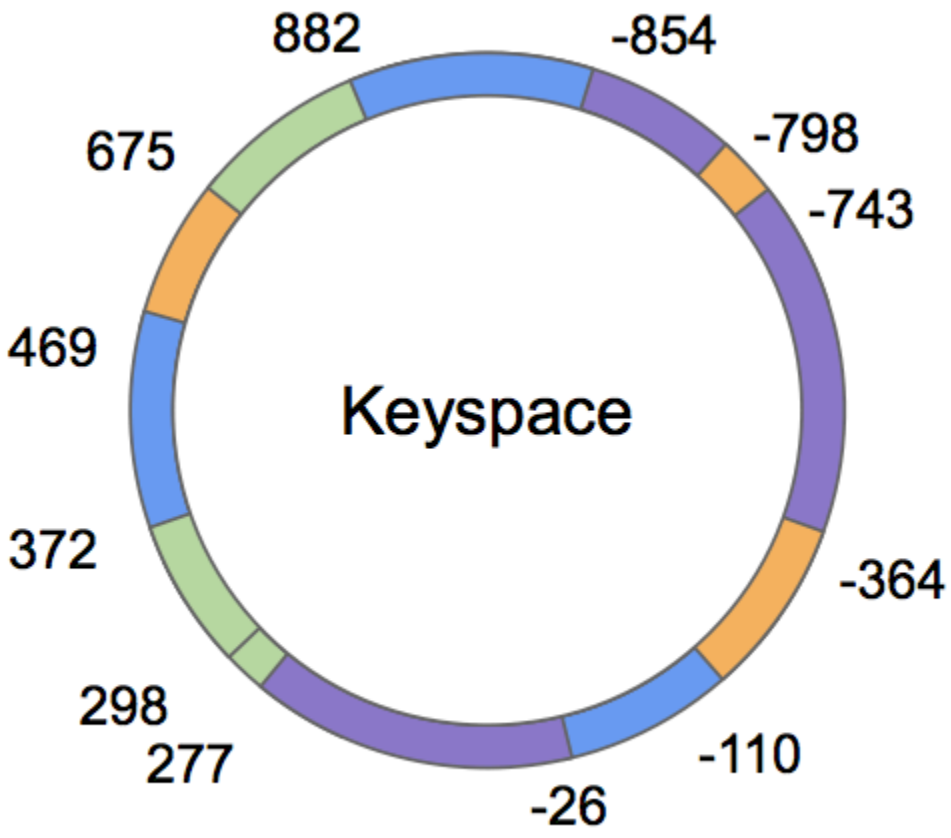
Cassandra Architecture

- Data Partitioning & Distribution
  - Partitioners
  - Virtual Nodes
- Data Replication
- Network Toplogy (Snitches)
- Server-to-Server Communication (Gossip)
  - Membership
  - Failure detection
- Scaling a Cluster
- Client-Server Communication
- Local Persistence

Cassandra Architecture: Data Partitioning & Distribution



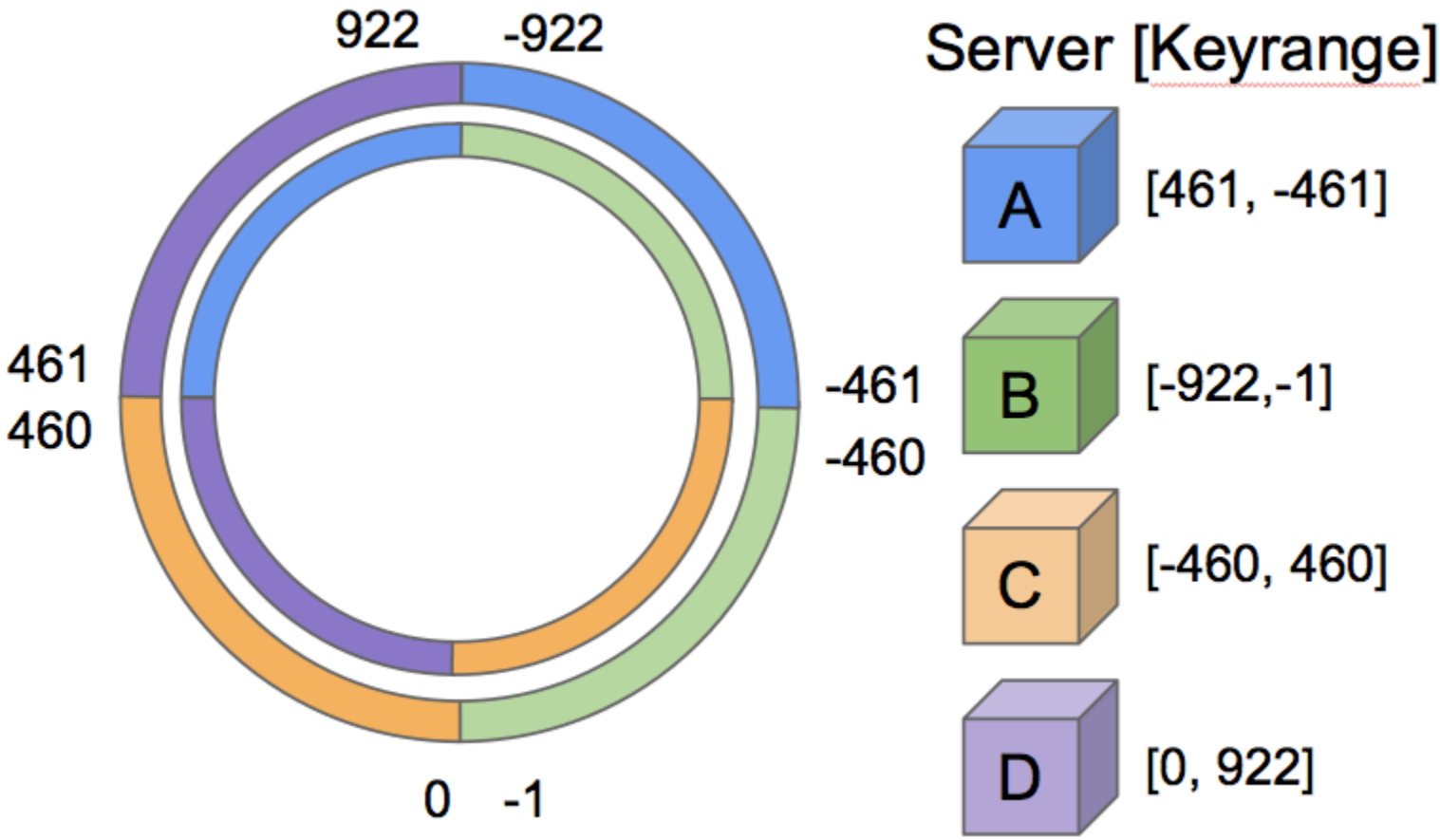
Cassandra Architecture: Virtual Nodes



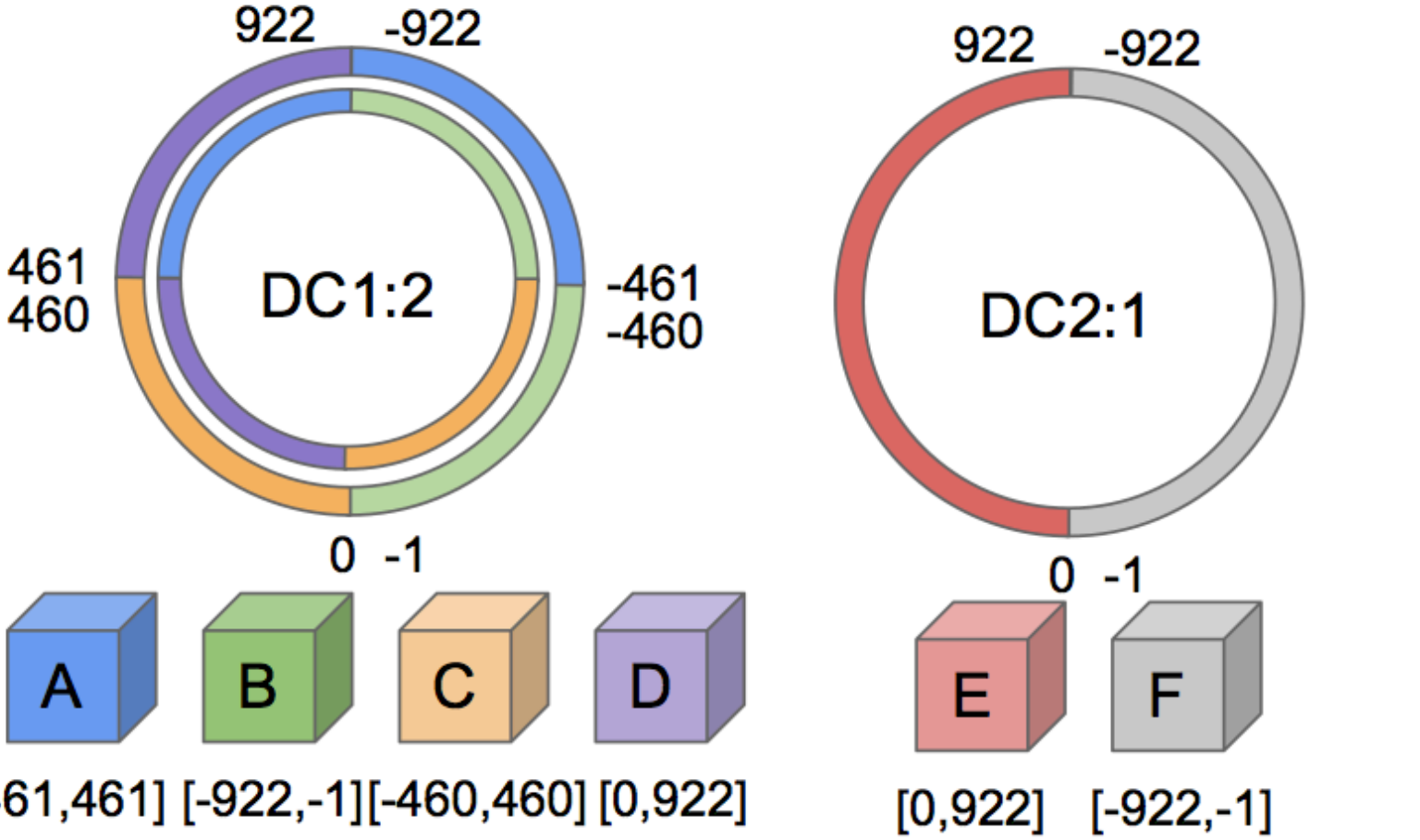
Server [Keyrange]

A	[882, -854] [-110, --26] [372, 469]
B	[675, 882] [227, 298] [298, 372]
C	[-798, -743] [-364, -110] [469, -675]
D	[-854, -798] [-743, -364] [-26, 277]

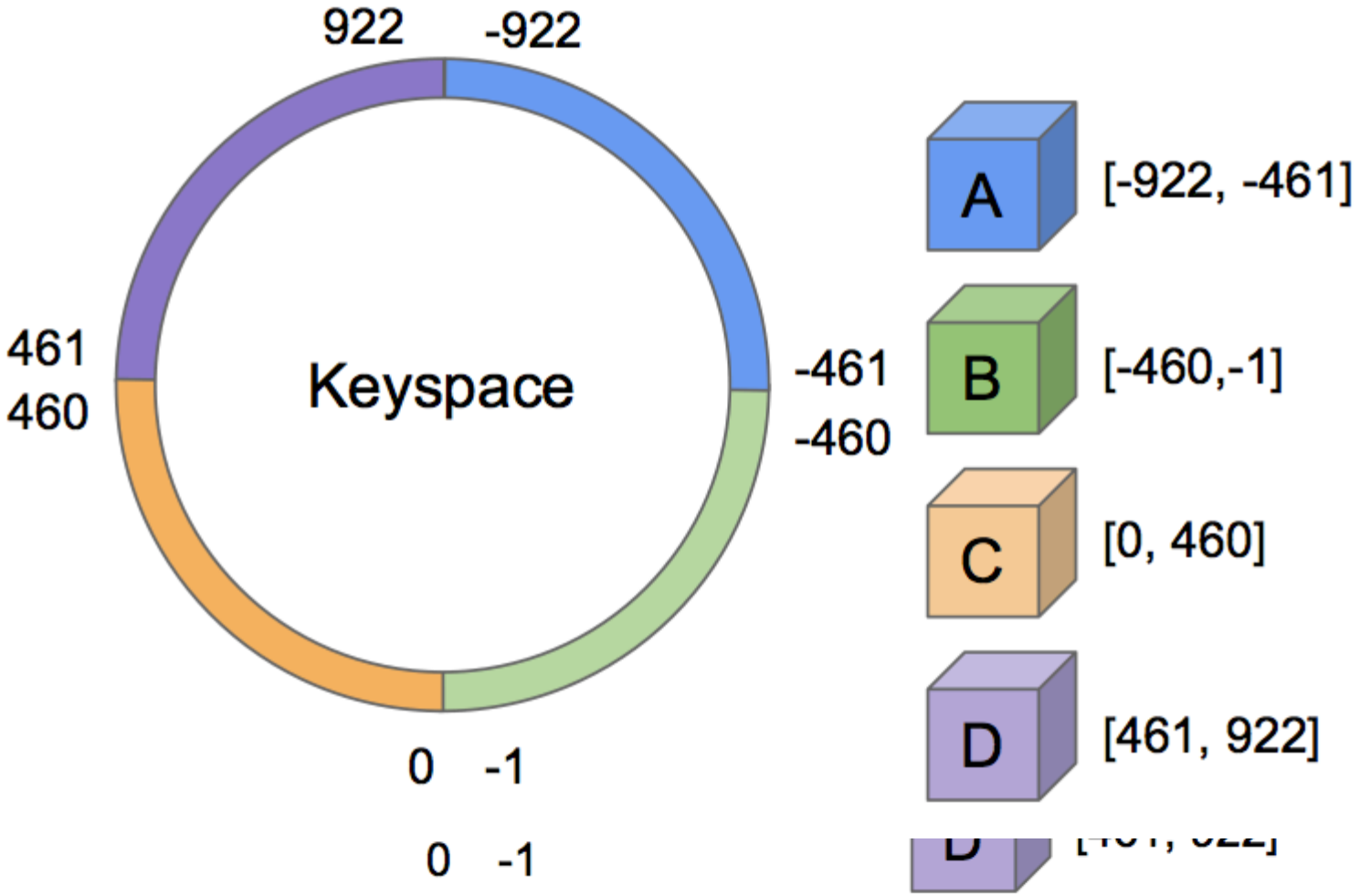
Cassandra Architecture: Data Replication



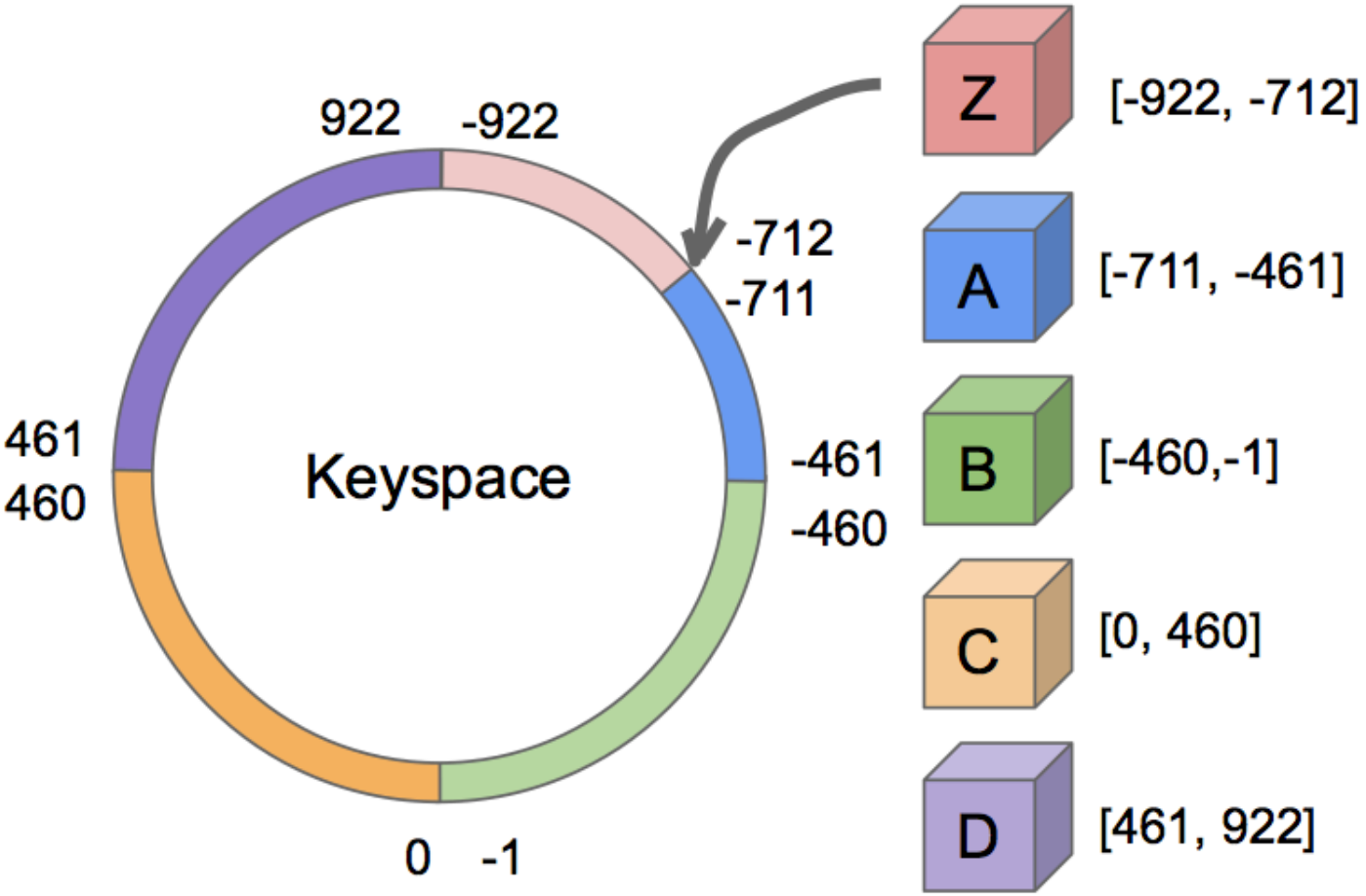
Cassandra Architecture: Multi-DC Data Replication



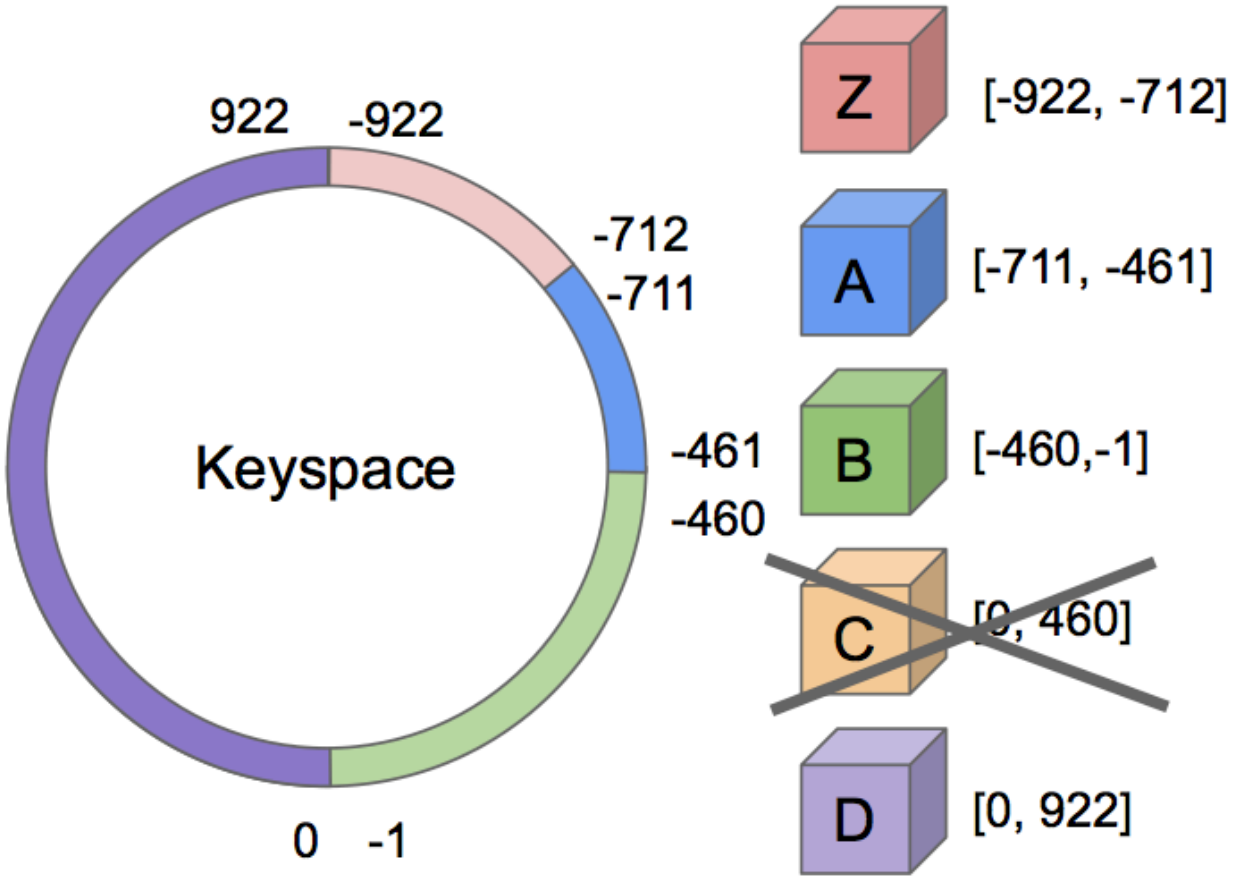
Cassandra Architecture: Scaling



Cassandra Architecture: Scaling



Cassandra Architecture: Scaling



Cassandra Architecture: Local Persistence

Write

- Append to commit log for durability (recoverability)
- Update of in-memory, per-column-family Memtable

If Memtable crosses a threshold

- Sequential write to disk (SSTable).
- Merge SSTables from time to time (compactions)

Cassandra Architecture: Local Persistence

Technische Universität Berlin

Read

- Query in-memory Memtable
- Check in-memory bloom filter
  - Used to prevent unnecessary disk access.
  - A bloom filter summarizes the keys in a file.
  - False Positives are possible
- Check column index to jump to the columns on disk as fast as possible.
  - Index for every 256K chunk.

Summary

Cassandra implements a scalable and highly available replication architecture very similar to Amazon Dynamo and a write-optimized storage engine very similar to BigTable.

Cassandra has evolved since 2010 from an open source project into a robust database solution.

- Main contributor: http://www.datastax.com
- New features (more than Dynamo + BigTable)
  - Cassandra Query Language (CQL), “lightweight transactions”, etc.
  - Architecture improvements (Leveled Compactions, etc.)
  - Integration with other systems, such as Hadoop

Agenda

1. **Intro Cloud Storage**

2. Systems in detail
   a) Dynamo
   b) GFS, BigTable

3. Other systems
   c) Cassandra
   d) MongoDB

Document Store Example: MongoDB
Learning objectives

Technische Universität Berlin

Cross-platform document-oriented database which supports ad hoc queries by field, range & regular expression.

High availability with replica sets.

Horizontal scaling through sharding.

Reference:
Kristina Chodorow
“MongoDB: The Definitive Guide, Second Edition”, O’Reilly, 2013

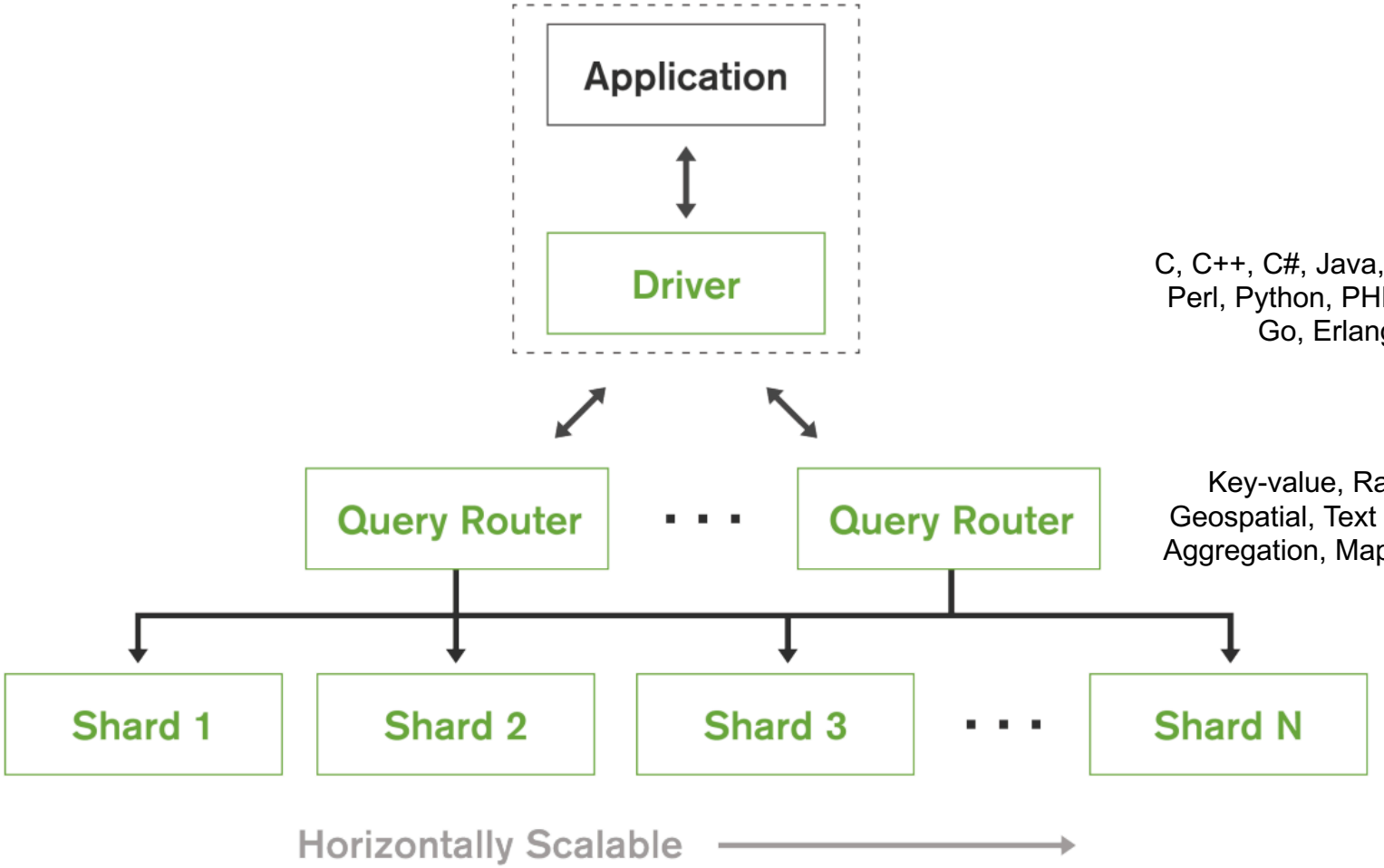
MongoDB

- document oriented & schema less
- written in C++ & developed in an open-source project (2007)
- supports:
  - Indexes & Secondary Indexes (unique, compound, array, ttl, geospatial, text ...)
  - Aggregation (count, distinct, group)
  - Special collection types
  - File storage
  - Deep query-ability
  - MapReduce
- does not support Joins & Transactions ($atomic flag)

best to use:

- deeply nested, complex data structures
- applications in javascript - stores in JSON-like format

MongoDB: Architecture



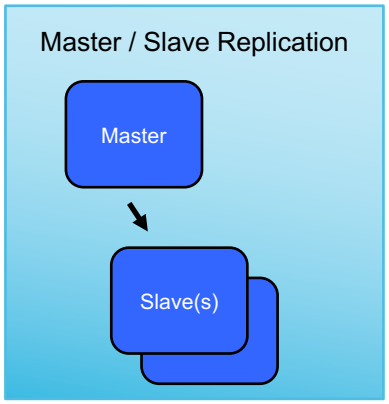
C, C++, C#, Java, Node.js, Perl, Python, PHP, Ruby, Go, Erlang

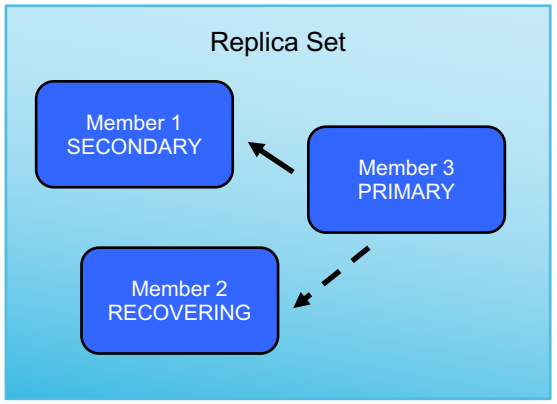
Key-value, Range, Geospatial, Text Search, Aggregation, MapReduce

Source: 2015 http://s3.amazonaws.com/info-mongodb-com/MongoDB_Architecture_Guide.pdf

MongoDB: Distribution Aspects

- Partitions called „Shard‘s“
  - Range-based
  - Hash-based
  - Location-aware





- asynchronous replication with primary server/node (+ hidden/delayed)
- failover causes elections