ST446 Distributed Computing for Big Data

Lecture 2

Distributed file systems and key-value stores

Milan Vojnovic

https://github.com/lse-st446/lectures2021

Goals of this lecture

- Learn basic principles of distributed file systems

- Learn basic principles of key-value stores

- Focus on some of the main principles of computer systems architecture by considering some specific, but canonical existing systems

Outline

- Distributed file systems: introduction
  - Google File System (GFS) example

- Distributed file systems: architecture
  - GFS architecture
  - Hadoop Distributed File System (HDFS)
  - Windows Azure Storage

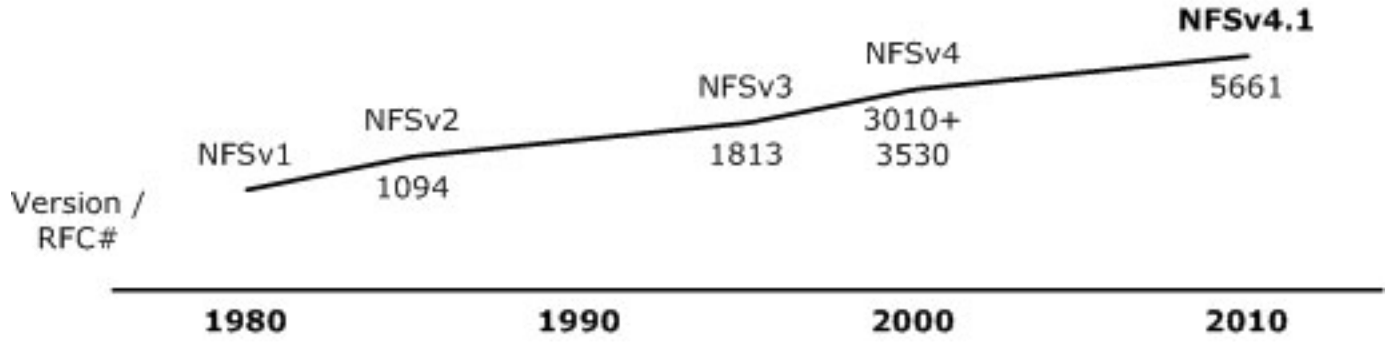
- Distributed key-value stores: introduction
  - Amazon Dynamo example

- Distributed key-value stores continued
  - Google Bigtable example

Distributed file systems: introduction

Network file systems

- Network file system: a network abstraction of a file system
  - Allows a remote client to access it over a network in a similar way to a local file system
  - Origin: File Access Listener, DEC,1976
- NFS: the first modern network file system
  - Developed by Sun Microsystems in early 1980s
  - 1995: NFS v3, standard RFC 1813: more scalable, supporting large files (> 2GB), async writes, using TCP (Transport Protocol Protocol)
  - 2000: NFS v4, RFC 3010, enterprise applications
  - 2003: NFS v4 revision, RFC 3530
  - 2010: NFS v4.1, RFC 5661, added protocol support for parallel access across distributed servers



Source: M. Jones, Network file systems and Linux, IBM developerWorks, 2010

POSIX semantics

- POSIX: Portable Operating System Interface
  - A family of standards specified by the IEEE Computer Society for maintaining compatibility between different operating systems
  - Defines API, command line shells and utility interfaces, for software compatibility with variants of Unix and other operating systems
- Original standard developed by IEEE Std 1003.1-1988
- POSIX standard sections:
  - POSIX.1: core services (includes IO port interface and control)
  - POSIX.1b: real-time extensions (IEEE Std 1003.1b-1993)
  - POSIX.1c: threads extensions (IEEE Std 1003.1c-1995)
  - POSIX.2: shell and utilities (IEEE Std 1003.2-1992)
- Modern distributed file systems: some requirements traded for high throughput
  - B. J. Layton, POSIX IO Must Die!, Linux Magazine, 2010

Distributed file systems

- Several distributed file systems used in practice:
  - Google: Google File System (GFS), new version Colossus
  - Apache: Hadoop Distributed File System (HDFS)
  - Microsoft: Windows Azure Streams

- Common main design elements
  - Different designs influenced one another, ex. HDFS influenced by GFS

- Our focus is on GFS, one of early systems
  - Other systems have similar design properties
  - We will highlight some of the innovations introduced by other file systems

Google File System

- A scalable distributed file system for large distributed data-intensive applications
  - The architecture described in a SOSP 2003 paper
- Design objectives:
  - Fault tolerance: running on inexpensive commodity hardware
  - Batch data processing on large files: original use case, web search index
- Non goals:
  - Support for latency sensitive applications
- Main architecture concepts:
  - Big files partitioned into chunks (originally 65MB chunk size)
  - Master node for metadata and chunk management
  - Chunks are replicated for reliability (typically 3x)

Web search index use case

- *Search engine indexing*: collects, parses, and stores data to facilitate fast and accurate information retrieval
- Hundreds of web crawling clients
- Periodic batch analytics jobs like MapReduce (covered in the next lecture)
  - Later done through incremental processing (as new data arrives)
- Large data volume requires *scaling out*: ex. 100s of TB, 1000s of machines
- More on search engine index: [wikipedia]

GFS design rationale

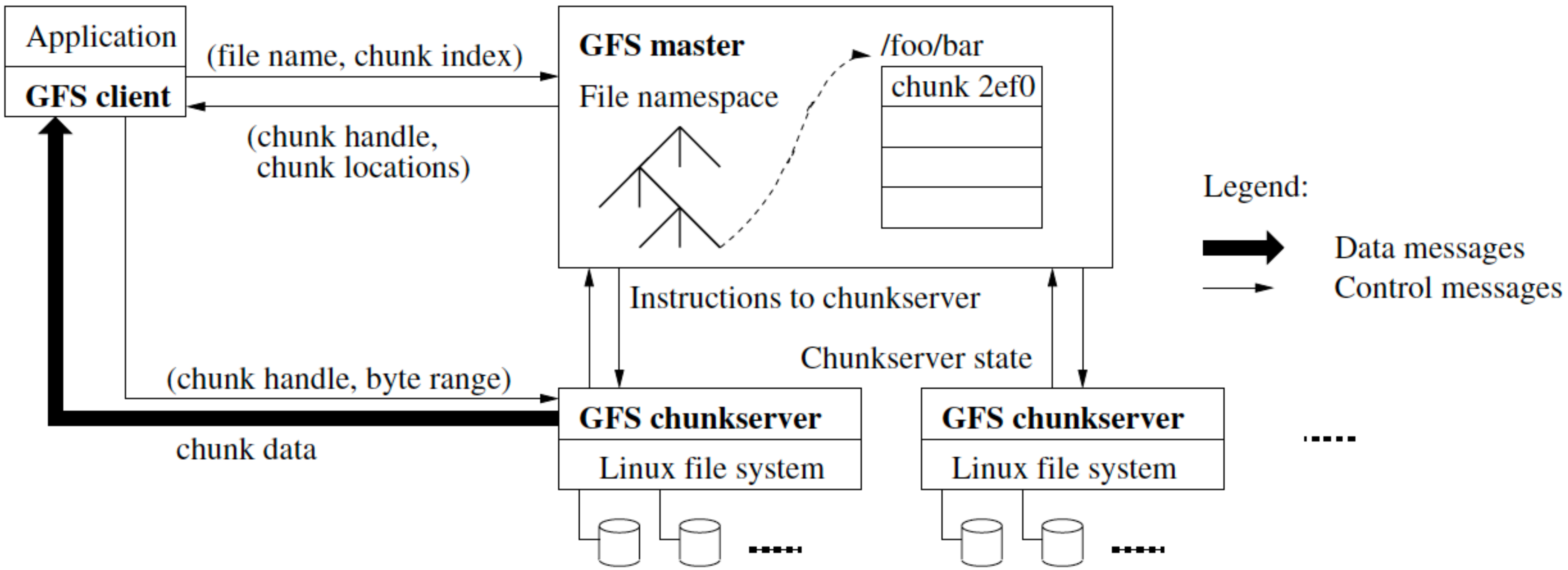
- Designed to satisfy the needs of batch data processing workloads:
  - Reads: small reads and large *streaming reads*
  - Writes: many files *written once*, other files *appended to*
- No support for *random writes*
  - Expensive and not needed for intended use cases

Distributed file systems: system architecture

GFS system architecture

- Main components: client, master, chunk servers
- Single master architecture
    - Use of shadow masters for replication
- Master stores metadata (in memory):
  (filename, list of chunk ids) and (chunk id, list of chunk servers)
- Master *does not* store file content
- Interface:
    - Application-level library, not a POSIX file system
    - File system operations: create, delete, open, close, read, write
    - Concurrent writes not guaranteed to be consistent (trading consistency for speed)
    - Record *append* guaranteed to be atomic

Read operations



Read operation steps

1. Client translates a file name and byte offset specified by the application into a chunk index within the file
2. Client sends a request to master containing the file name and chunk index
3. Master returns to client with the chunk handle and locations of the replicas
4. Client caches this information using the file name and chunk index as the key
5. Client sends a request to one of the replicas, most likely the closest one: the request specifies the chunk handle and a byte range within that chunk

- Notes:
  - Subsequent reads of the same chunk require no client-master interaction until the cached information expires or the file is reopened
  - Client typically asks for multiple chunks in the same request

Write operations

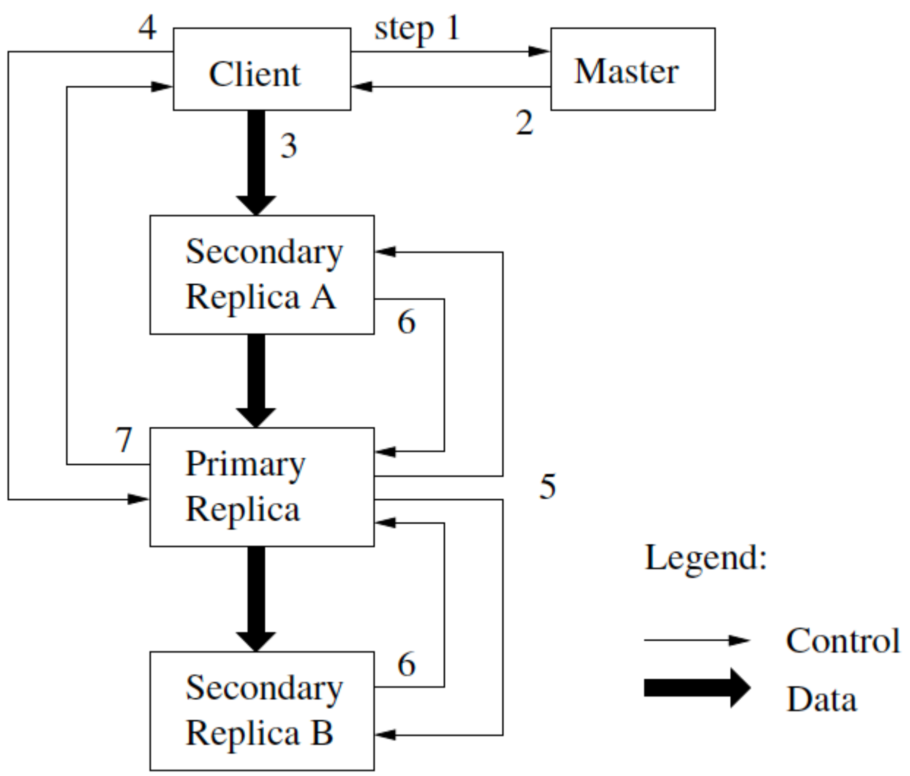
- Mutation is either write or append
  - No support for modify

- Lease mechanism:
  - Master picks one replica as primary; gives it a lease
  - Primary defines a serial order of mutations

- Important: data flow decoupled from control flow
  - Minimal involvement of the master for scalability reasons

Write operation steps

1. Application sends a request to master to return chunk indices for input (file name, data)
2. Master returns chunk handle replica locations (primary + secondary)
3. Client pushes write data to all locations
4. Client sends write command to primary
5. Primary determines a serial order for data instance stored in its buffer and writes the instances in that order to the chunk; primary sends serial order to the secondaries and tells them to perform the write
6. Secondaries respond to the primary
7. Primary responds back to the client



GFS upgrades

- Original system scaled to order 50M files and 10PB total data size in production
- Typical system size was in the order of 1,000 servers, now order 10,000 servers
- Workload changes:
  - 100PB total data size
  - Not everything is batch updates to small files
- Around 2010, moved to incremental index updates, instead of periodic rebuilding of the index with MapReduce
- Needed a new file system design: Colossus

GFS upgrades: Colossus

- A next-generation cluster-level file system
- Scalability of a single master improved by automatically sharded metadata layer
- Added *data coding* for fault tolerance (previously used only *data replication*)
- Support 100M files per master and smaller chunk sizes (1MB instead of 64MB)
- Client-driven replication, encoding and replication
- Metadata space has enabled availability analyses

Coding in storage systems

- Original approach to fault tolerance:
  - 3-way replication (3 copies of each data chunk stored)
    - 2x overhead, 0.33 storage efficiency, 2 failures tolerance, easy recovery
    - Def overhead = S / D, storage efficiency = D / (S + D), D = data size, S = stored size

- Coding now widely-deployed as an alternative in different distributed file systems:

  - Google Colossus: (6,3) Reed-Solomon code
    - 1.5x overhead, 0.67 storage efficiency, 3 failures tolerance
  - Facebook HDFS: (10,4) Reed-Solomon code
    - 1.4x overhead, 0.71 storage efficiency, 4 failures tolerance, expensive recovery
  - Microsoft Azure: (12,4) Local Reconstruction Code (LRC)
    - 1.33x overhead, 0.75 storage efficiency, 4 failures tolerance, same recovery cost as Colossus

- Note: (m,k) Reed-Solomon code
  m data symbols per block out of which k are check (parity) symbols

To probe further: https://blog.cloudera.com/introduction-to-hdfs-erasure-coding-in-apache-hadoop/

Basic facts about coding

- Read-Solomon code
  - Error-correcting code introduced by Reed and Solomon in 1960
  - Applied to consumer technologies such as *storage* (CDs, DVDs, Blu-ray discs, RAID 6) and *data transmission* (DSL, WiMAX, DVB)
  - To probe further Wikipedia

- Local Reconstruction Code (LRC)
  - LRC is an erasure code introduced by Huang et al, Erasure Coding in Windows Azure Storage, USENIX 2012, for distributed file systems
  - Erasure code: a forward error correction code assuming bit erasures (not bit errors)
  - An erasure code transforms a massage of s data symbols into a message of $c \geq s$ codword symbols
  - Each original data message can be recovered from any subset of s codeword symbols
  - Erasure code has code rate $r = s/c$

Hadoop Distributed File System (HDFS)

- Designed to reliably store very large files across machines in a large cluster

- The system design inspired by GFS

- Main system properties:
  - Handling hardware failures
  - Streaming data access: design for batch processing rather than interactive use
  - Large datasets (typical files from GB to TB in size)
  - Simple coherency model: write-once-read-many access model
  - Moving computation considered cheaper than moving data
  - Portability across heterogeneous hardware and software platforms

HDFS system architecture

- Similar to GFS: master : namenode, chunk server : datanode, chunk : block
- Default block size: 128MB
- Namenode resilience to failure
    - Backup files for persisting the state of the filesystem metadata
        - Ex. write to local disk or a remote NFS mount
    - Secondary namenode periodically merging the namespace image with the edit log
        - To prevent the edit log from becoming too large
- Scaling the namenode
    - HDFS federation: multiple namenodes each managing a portion of the filesystem namespace
        - Ex. \`/user\` and \`/share\`
- HDFS high availability
    - Use of a pair of namenodes in an active standby configuration

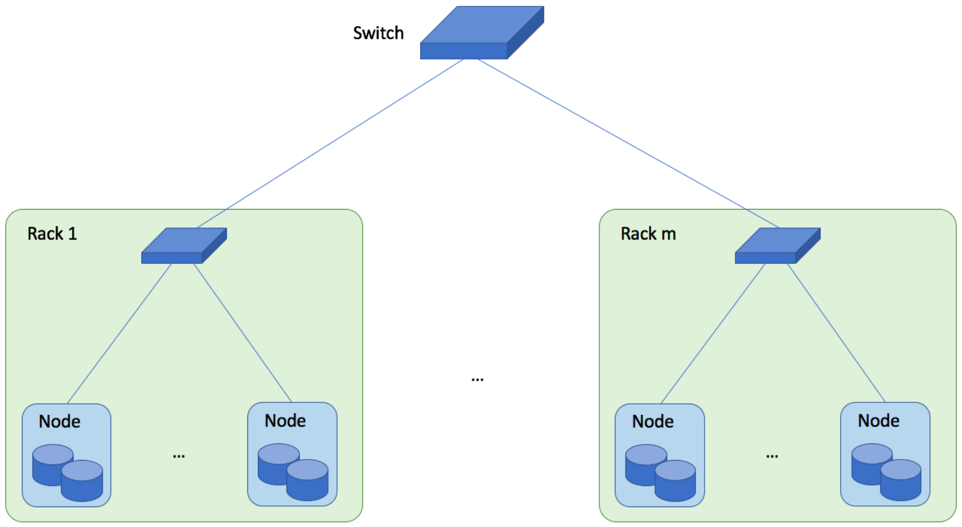
Hadoop filesystems

- Hadoop abstraction: filesystems
  - HDFS is just one implementation in this abstraction
  - The client interface to a file system is represented by a Java abstract class
    - Java abstract class: `org.apache.hadoop.fs.FileSystem`
- Example Hadoop filesystems:

Filesystem	URI	Description
local	file	A file system for a locally connected disk
HDFS	hdfs	HDFS desiged to work efficiently in conjuction with Mapreduce
WebHDFS	webhfds	Authenticated read/write access to HDFS over HTTP
FTP	ftp	A filesystem backed by an FTP server
S3	s3a	Amazon S3 filesystem
Azure	wasb	Microsoft Azure filesystem
Swift	swift	OpenStack file system

Data block location optimization

- A typical Hadoop cluster architecture consists of a two-level network topology



Default block replica placement:

- First replica: placed on the same node as the client
    - If the client is running outside the cluster, a node is chosen at random
- Second replica: placed on a different rack from the first (off-rack)
- Third replica: placed on the same rack as the second, but on a different node chosen randomly
- Any further replicas: placed on random nodes in the cluster

Distributed key-value stores: introduction

Distributed key-value stores

- Simple data model: (key, value) records

- Key design objectives:
  - Fast reads and writes
  - Scalability

- Example systems:
  - Amazon Dynamo
  - Voldemort Linkedin (inspired by Dynamo)
  - Github, Digg, Blizzard: Redis
  - Facebook, Twitter, Zynga: Memcached
  - Apache Cassandra: a schema-based distributed key-value store

Example: Amazon Dynamo

- Highly available key-value storage system
- Used by Amazon's core services to provide an "always-on" experience
  - Reliability is one of the most important requirements
  - Outages cost money and impact customer trust
- Key design objectives:
  - High availability: achieved by
    - trading-off consistency under certain failure scenarios
    - using object versioning
    - application-assisted conflict resolution
  - Scalability: highly scalable to support a continuous growth
- Superseded original use of a relational database to increase scale and availability

Example use cases in Amazon example

- Shopping carts
- Best seller lists
- Customer preferences
- Session management
- Sales ranks
- Product catalog
- ...

Key architecture concepts

- Consistent hashing: mapping data keys to machines
  - Partitioning and replicating (key, value) over machines
- Object versioning: data object versioning for consistency
- Simple quorum method for consistency of data replicas
- Decentralized replication and synchronization protocol
- Gossip protocol for distributed failure detection and membership

Application Programming Interface (API)

- Basic data access operations:
  - get(key)
    - locates the object replicas associated with the key in the storage system
    - returns a single object and a context (or a list of objects with conflicting versions)
  - put(key, context, object)
    - determines where the object replicas should be placed based on the key
    - writes the replicas to disk

- Context: encodes system metadata about the object that is opaque to the caller and includes information such as the object's version

- Hashing: MD5 hash applied on the key to generate a 128-bit identifier used to determine the storage nodes that are responsible for serving the key
  - MD5 hash: 128-bit hash value (Ronald Rivest, 1991) more wikipedia
  - Both the key and the object treated as an array of bytes

MD5 hashes

```
LSE021353:~ vojnovic$ man md5


MD5(1)                    BSD General Commands Manual                   MD5(1)

NAME
     md5 -- calculate a message-digest fingerprint (checksum) for a file

SYNOPSIS
     md5 [-pqrtx] [-s string] [file ...]

DESCRIPTION
     The md5 utility takes as input a message of arbitrary length and produces
     as output a ``fingerprint'' or ``message digest'' of the input.  It is
     conjectured that it is computationally infeasible to produce two messages
     having the same message digest, or to produce any message having a given
     prespecified target message digest.  The MD5 algorithm is intended for
     digital signature applications, where a large file must be ``compressed''
     in a secure manner before being encrypted with a private (secret) key
     under a public-key cryptosystem such as RSA.

     MD5's designer Ron Rivest has stated "md5 and sha1 are both clearly bro-
     ken (in terms of collision-resistance)".  So MD5 should be avoided when
     creating new protocols, or implementing protocols with better options.
     SHA256 and SHA512 are better options as they have been more resilient to
     attacks (as of 2009).


 ⋮

LSE021353:Documents vojnovic$ md5 a.txt
MD5 (a.txt) = d41d8cd98f00b204e9800998ecf8427e
```

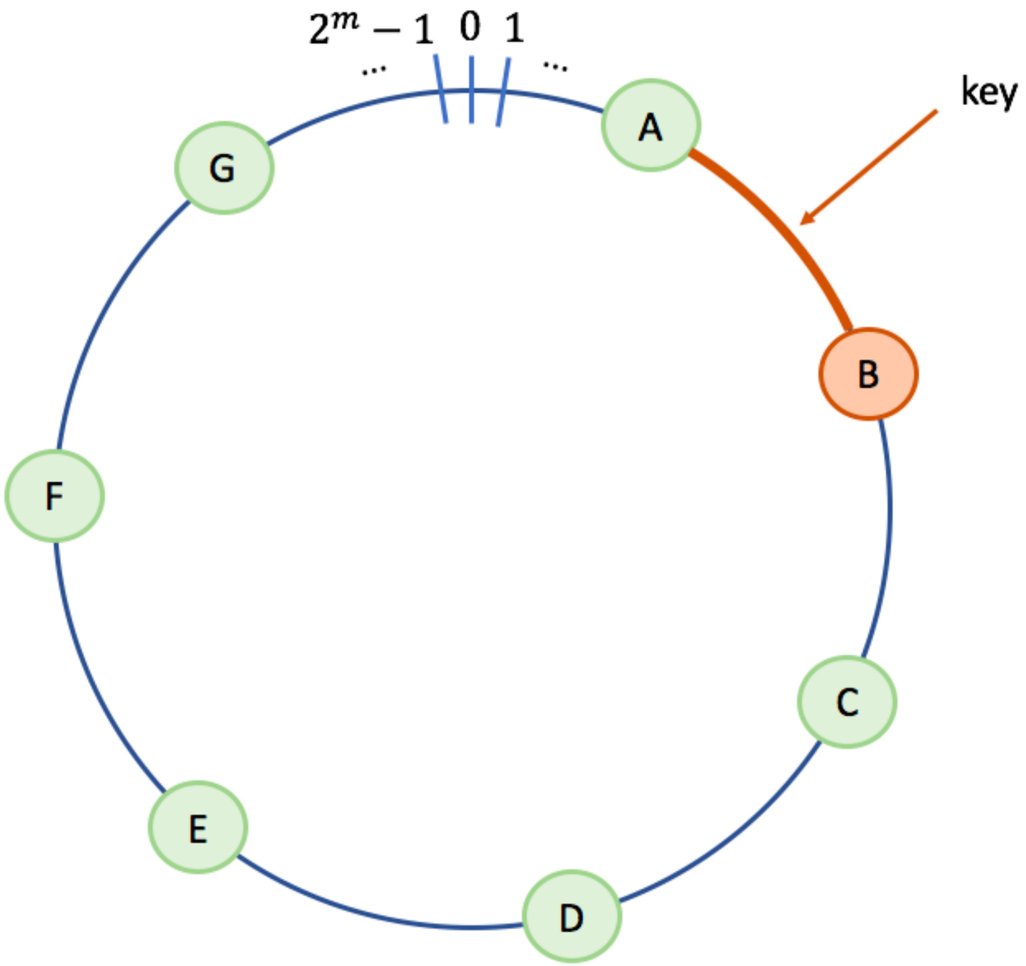
Data partitioning

- System required to scale incrementally
  - Achieved by dynamically partitioning the data over a set of nodes (storage hosts)
  - Data distributed across multiple storage hosts using consistent hashing

- Consistent hashing:
  - The output range of a hash function treated as a fixed circular space or ring
  - Each node assigned a random value in the ring, which represents its position
  - Each data item identified by a key assigned to a node by hashing the data item's key
  - Ring traversed clockwise to find the first node with position larger than the item's position
  - Each node responsible for the region between it and its predecessor node

- Note: consistent hashing used in peer-to-peer systems
  - Ex. distributed hash tables (DHTs) such as Chord

Consistent hashing ring



- Circles denote nodes
- m = number of bits output by the hash function

Data partitioning (cont'd)

- Consistent hashing advantage
  - Departure or arrival of a node only affects its immediate neighbors, other nodes unaffected

- Replication: each node gets assigned to multiple points in the ring for better load balancing and dealing with node performance heterogeneity

- Virtual node: looks like a single node in the system
  - Each node can be responsible for one or more virtual nodes

- Virtual node advantages:
  - If a node becomes unavailable, its load is evenly spread across the available nodes
  - When a node becomes available again, or a new node is added to the system, the newly available node accepts an equivalent amount of load from each of the other available nodes
  - The number of virtual nodes that a node is responsible can be decided based on its capacity, accounting for heterogeneity of the physical infrastructure and other factors

Data replication

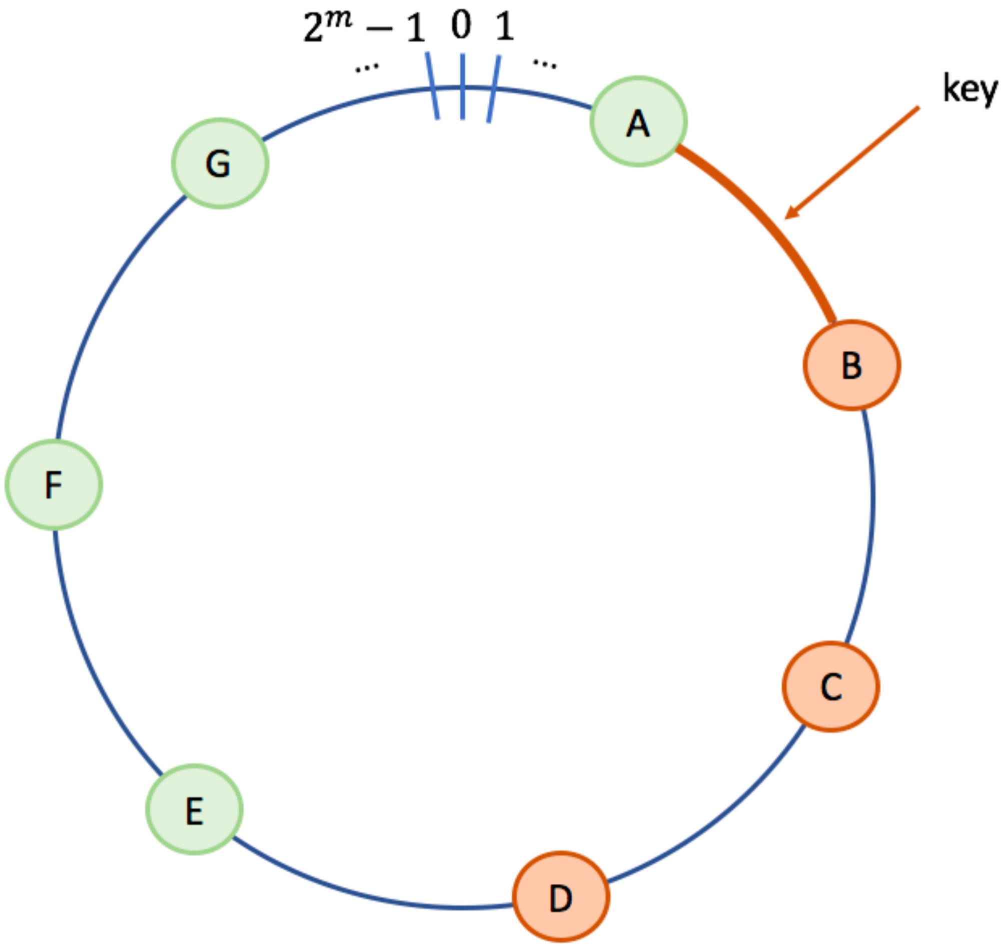
- Data replicated across multiple nodes for high availability and durability
  - Each data item is replicated on k nodes

- Coordinator node: responsible for replication of data items falling within its range

- Coordinator node's responsibilities:
  - Locally stores each key within its range
  - Replicates these keys at k-1 clockwise successor nodes in the ring (each node is responsible for the region of the ring between it and its k-th predecessor)

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

Consistent hashing ring with replication



- Nodes B, C and D store data items in the range (A,B]

Consistency

- Coordinator node handles read and write operations
  - Default: the first node in the top k nodes in the preference list

- Read and write operations involve the first k available nodes in the preference list

- Consistency protocol used similar to a quorum system
  - Two configurable parameters: r and w
  - r = min number of nodes that must participate in a successful read operation
  - w = min number of nodes that must participate in a successful write operation

- r and w set such that r + w > k

Distributed key-value stores continued (bigtable)

Bigtable

- Distributed storage system for managing structured data
- Architecture described in an OSDI 2006 paper
- Designed to scale to very large data sizes
  - Petabytes of data across thousands of commodity servers
- Use cases: web indexing, Google Earth, Google Finance, ...
- Data model: multidimensional array
  - Clients have control over data layout and format
- Offered as a service by Google Cloud Platform
  - Open source cousin: Apache HBase

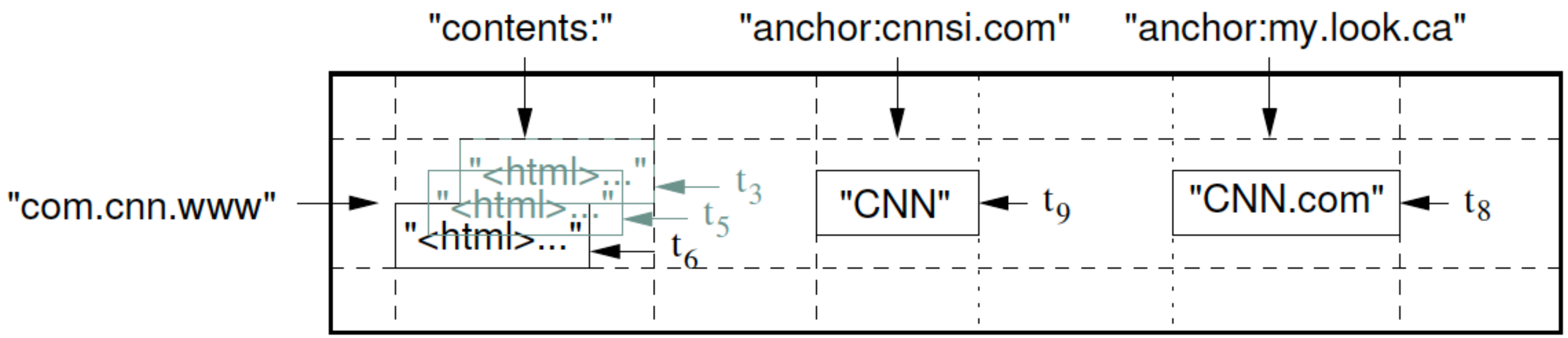
Data model

- Data model: sparse, distributed, persistent multidimensional sorted map

- Map indexed by a row key, a column key and a timestamp, byte array values

  (row:string, column:string, time:int64) -> string

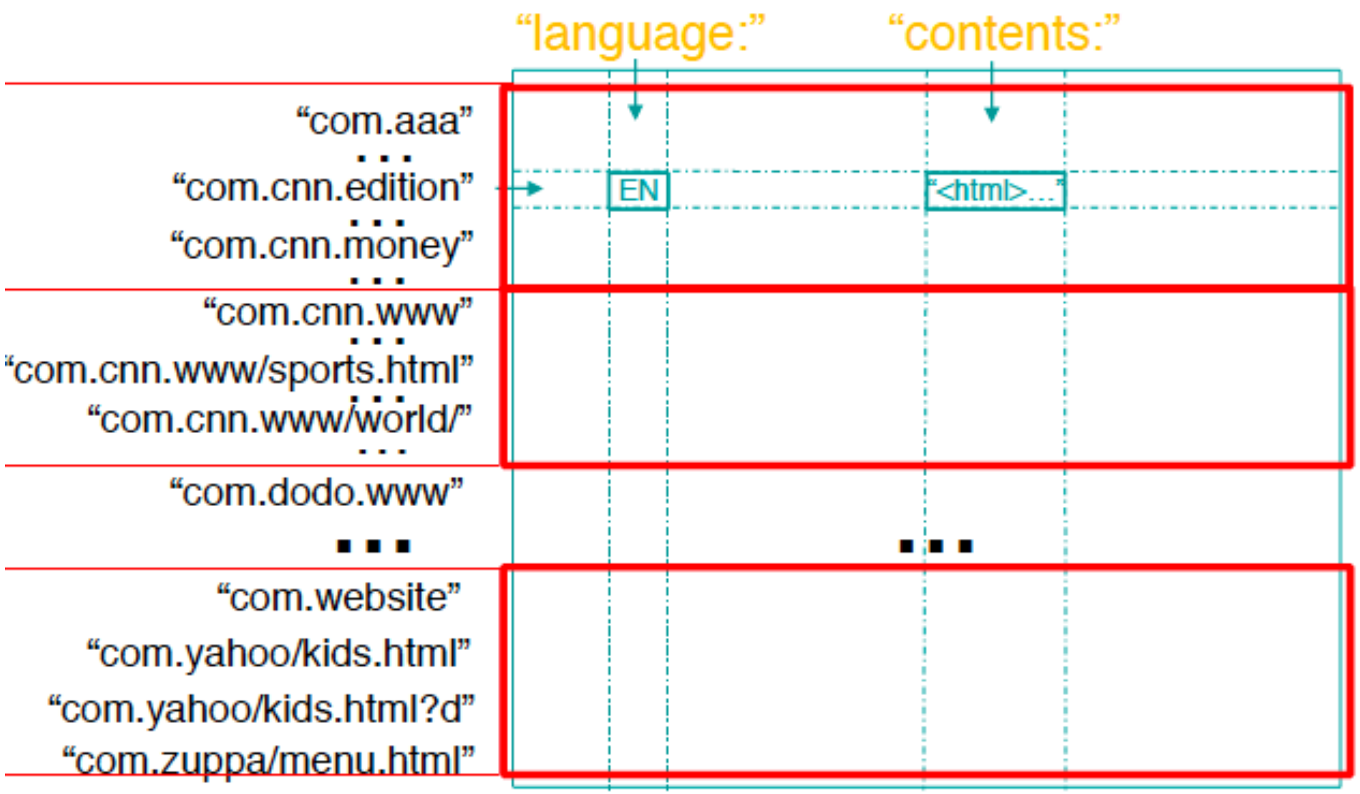
- Example: store a large collection of web pages and related information
  - Row key: URL
  - Column name: aspect of the web page (e.g. anchor)



Rows and tablets

- Row keys: arbitrary strings
  - Originally 64KB with 10-100 bytes being a typical size
- Atomic updates: atomic reads or writes under a single row key
  - Makes it easier for clients to reason about system behavior for concurrent updates to the same row
- Data order: lexicographic order by row key
  - Allows for dynamic partitioning into row ranges
- Tablet: a range of table rows
  - Basic unit used for data distribution and load balancing
  - Efficient read of short row ranges as they typically require communication with only a small number of machines
  - Clients can exploit this by selecting their row keys so that they get good locality for their data accesses

Tablets



Tablet:
Start: com.aaa
End: com.cnn.www

Tablet:
Start: com.cnn.www
End: com.dodo.www

Columns and column families

- Column family: a group of column keys
  - Basic unit for access control

- Column family data type: all data stored in a column family usually of the same type
  - Data in the same column family is compressed together

- Creation of column families: each column family must be created before data can be stored under any column key in that family

- Design assumptions:
  - Number of distinct column families in a table is small (in the order of 100s)
  - Column families rarely change during operation
  - A table may have a virtually unbounded number of columns

Columns and column families (cont'd)

- Column key naming syntax: family:qualifier
- Column family names must be printable, but qualifiers may be arbitrary strings
  - Ex. family = html anchor
- Access control, disk and memory accounting performed at column-family level
- Timestamps: each cell of a table may contain multiple versions of the data
  - Different versions are indexed by timestamps (64-bit integers)
  - Timestamps are either assigned by the system (microseconds) or explicitly assigned by client applications
  - Automatic garbage collection: client can specify either that only the last n versions of a cell are kept, or that only recent enough versions are kept (e.g. only keep values that were written in the last seven days)

Schema design

- Key idea: de-normalize your database
- Replicate, cluster data if you can
- In contrast to RDBMS that aim to normalize the data

Bigtable API

- Basic functions:
  - Creating, deleting tables and column families
  - Changing cluster, table, and column family metadata (ex access control rights)
- Client applications can
  - write or delete values in a table
  - look up values for individual rows
  - iterate over a subset of rows in a table
- Transactions: support for single-row transactions
  - Can be used to perform atomic read-modify-write sequences on data stored under a single row key
- Original design did not support general transactions across row keys

Bigtable API example: connect

- Python source: [GCP]

Importing bigtable module

```
from google.cloud import bigtable
```

user input data: project_id, instance_id, table_id

Connecting to bigtable:

```
client = bigtable.Client(project=project_id, admin=True)
instance = client.instance(instance_id)
```

Bigtable API example: create and delete

Creating a table:

```
table = instance.table(table_id)
table.create()
```

Creating a column family:

```
column_family_id = 'cf1'
cf1 = table.column_family(column_family_id)
cf1.create()
```

Deleting a table:

```
table.delete()
```

Bigtable API example: writing rows

Writing rows example:

```python
column_id = 'greeting'.encode('utf-8')
greetings = [
    'Hello World!',
    'Hello Cloud Bigtable!',
    'Hello Python!',
]

for i, value in enumerate(greetings):
    row_key = 'greeting{}'.format(i)
    row = table.row(row_key)
    row.set_cell(
        column_family_id,
        column_id,
        value.encode('utf-8'))
    row.commit()
```

Bigtable API example: reading rows

Getting a row by key:

```
key = 'greeting0'
row = table.read_row(key.encode('utf-8'))
value = row.cells[column_family_id][column_id][0].value
print('\t{}: {}'.format(key, value.decode('utf-8')))
```

Scanning all rows:

```
partial_rows = table.read_rows()
partial_rows.consume_all()

for row_key, row in partial_rows.rows.items():
    key = row_key.decode('utf-8')
    cell = row.cells[column_family_id][column_id][0]
    value = cell.value.decode('utf-8')
    print('\t{}: {}'.format(key, value))
```

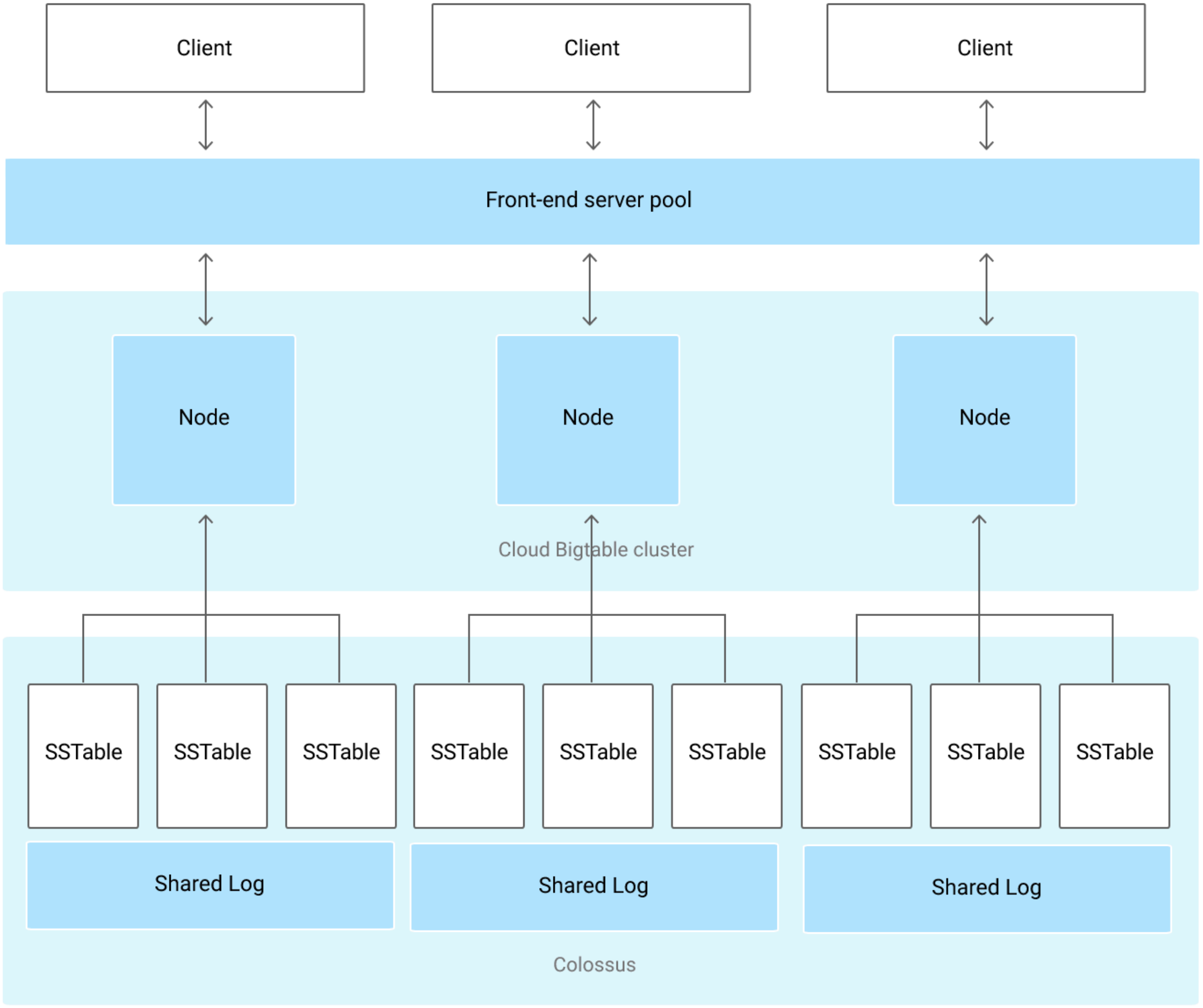
System architecture components

- Storing log and data files in a distributed file system
  - Colossus
- Persistent storage: storing ordered immutable key-value pairs
  - Google SSTable file format
- Highly-available and persistent distributed lock service Chuby
  - Five active replicas, one of which is elected to be master and actively serve requests
  - Chuby uses Paxos algorithm to keep its replicas consistent in case of failures
  - Paxos: distributed consensus protocol (more wikipedia)

SSTable (sorted string table)

- A simple abstraction for storing immutable key/value string pairs, sorted by keys
  - Persistent, ordered immutable map from keys to values, where both keys and values are arbitrary byte strings
  - Operations to look up value for a specified key, and to iterate over key/value pairs in a specified key range
- Internal design:
  - Each SSTable contains a sequence of blocks
    - Typical block size is 64KB, but this is configurable)
  - A block index (stored at the end of the SSTable) is used to locate blocks; the index is loaded into memory when the SSTable is opened
  - Efficient look-ups: a lookup can be performed with a single disk seek: first the appropriate block is found by performing a binary search in the in-memory index, and then reading the appropriate block from disk
  - Optionally, an SSTable can be completely mapped into memory, which allows to perform lookups and scans without touching the disk

Bigtable: system architecture



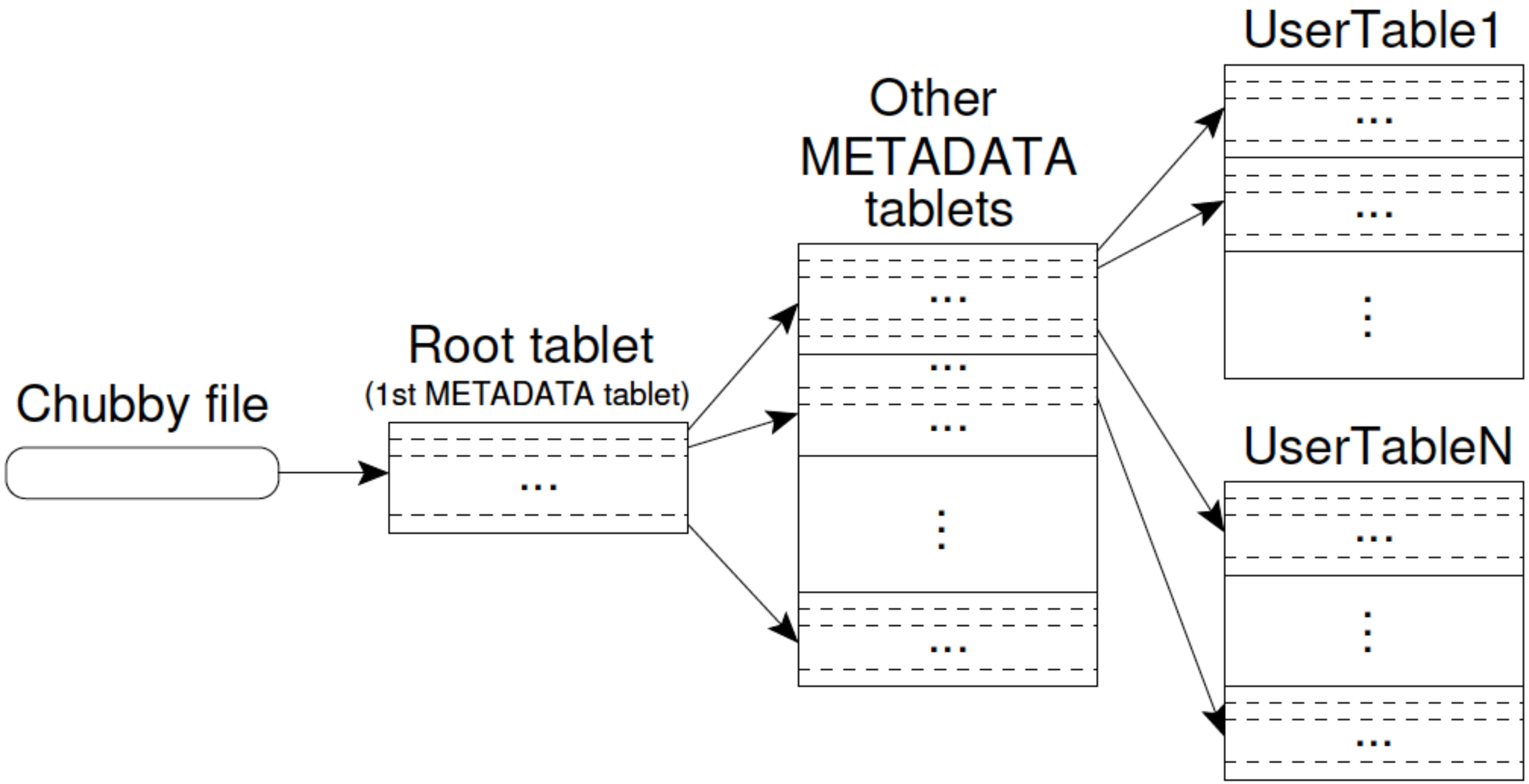
System components

- Client library
  - Library linked into every client
- Master server: single master
  - Assigning tablets to tablet servers
  - Detecting addition and expiration of tablet servers
  - Garbage collection of files in the file system
  - Handling schema changes such as table and column family creation
- Tablet servers: multiple servers
  - Managing a set of tablets (typically 10-1000 tablets)
  - Handling read and write requests to the tablets that it has loaded
  - Splitting tablets that have grown to large
- Scalability: separation of data and control flow
  - Clients communicate directly with tablet servers for data reads and writes

Tablet locations

- Three-level data structure used to store tablet location information
  - 1: a file that contains the location of the root tablet
  - 2: the root tablet contains the location of all tablets in a METADATA table
  - 3: each METADATA tablet contains the location of a set of user tablets
- Caching tablet locations: the client library caches tablet locations
  - If the client does not know the location of a tablet, or it discovers that cached location information is incorrect, then it recursively moves up the tablet location hierarchy

Tablet locations (cont'd)



Refinements

- Locality groups: clients can group multiple column families into a locality group
  - A separate SSTable is generated for each locality group in each tablet
  - More efficient reads by segregating column families that are infrequently accessed together into separate locality groups
  - A locality group can be declared to be in-memory

- Compression: clients can control whether SSTables in a locality group are compressed, and if so, which compression format is used
  - Common to use a two-pass custom compression scheme: first pass compressing long common strings across a large window, second pass using a fast compression algorithm for repetitions in a small window of data

- Caching for read performance: tablets use two levels of caching
  - High-level cache: caches key-value pairs returned by the SSTable interface to the tablet server code (useful for applications that tend to read the same data repeatedly)
  - Low-level cache: caches SSTables blocks that were read from the file system (useful for applications that tend to read data that is close to the data they recently read)

Refinements (cont'd)

- Bloom filters: a read operation must read from all SSTables that make a tablet
- If these SSTables are not in memory, may end up doing many disk accesses
- Reduced by allowing clients to specify that Bloom filters should be created for SSTables in a locality group
- Allows to ask whether an SSTable *might* contain any data for a specified row/column pair
- Commit log implementation: a single commit log per tablet server is used

Bloom filter

- Bloom filter: a space-efficient probabilistic data structure for

  testing whether an element is in a set

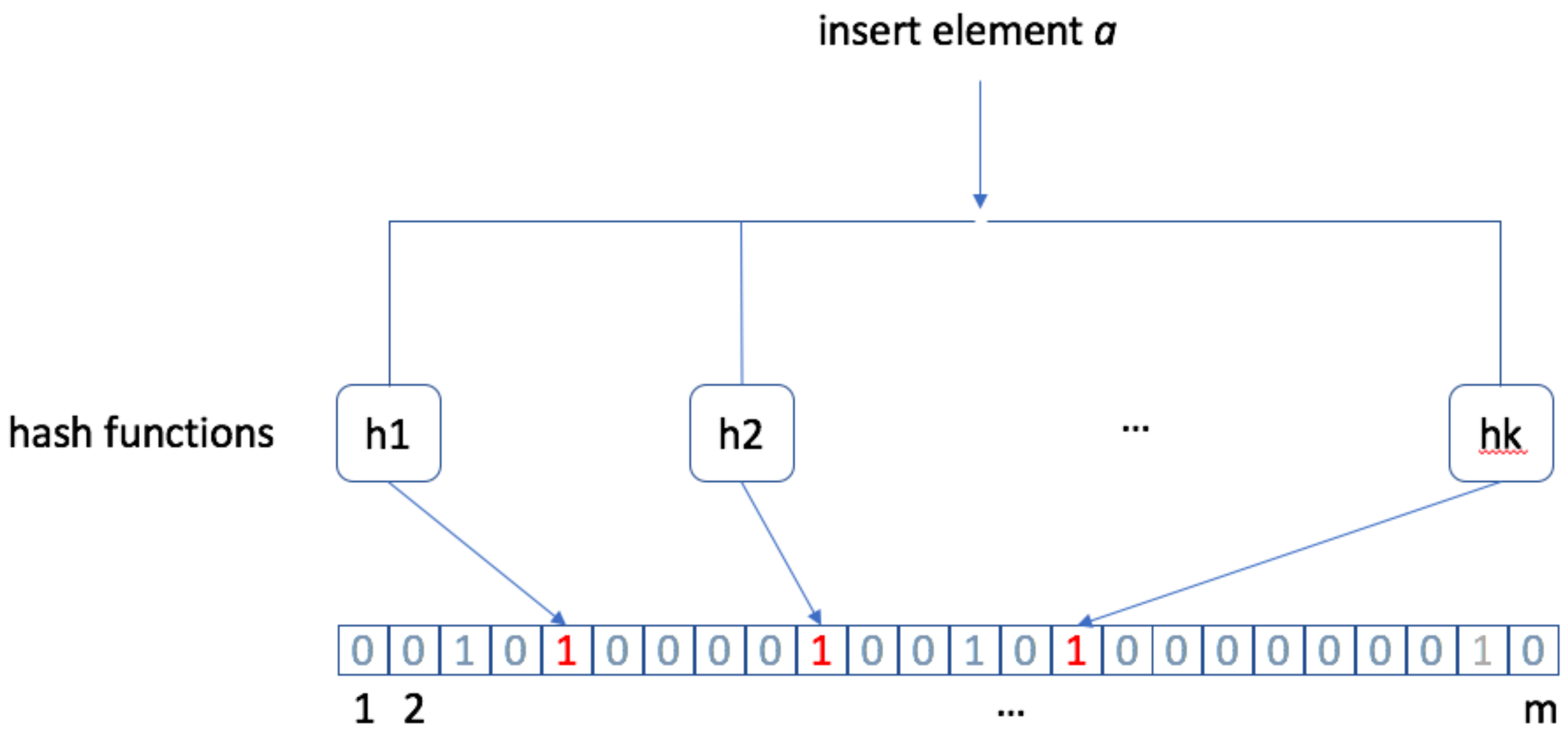
  - Conceived by Burton Howard Bloom in 1970

- Key properties:
  - False positive answers are possible
  - False negative answers are *not* possible
  - A query returns either "possibly in set" or "definitely not in set"
  - Elements can be added to the set, but not removed !
  - The more elements are added, the larger the probability of false positives

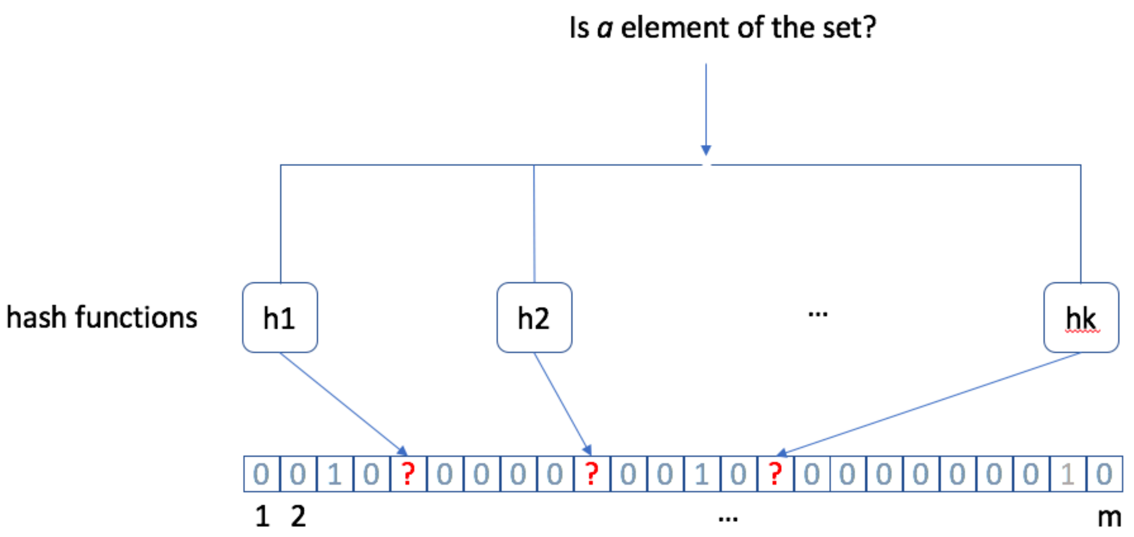
- Data structure: array of m bits and k hash functions
  - Each hash function maps an input element to one of m bits

Bloom filter: insert element

- Input element mapped to k array positions by hash functions, which are all set to 1



Bloom filter: query



Return	Maybe	if all hash functions point to bit value 1
	No	otherwise

- Query: check the values of bits at k array positions corresponding to the queried element

- If all are 1, then return "possibly in set"

  (either the element is in the set, or the bits have by chance been set to 1 during the insertion of some other element, resulting in a false positive)

- Else if any of the bits at these positions is 0, then return "not in the set"

  (the element is not in the set)

Seminar class 2: HDFS and Bigtable

- Before class: get started with Hadoop
- HDFS: basic HDFS commands, admin commands, run a mapreduce job
- Bigtable: using Bigtable on GCP from a Jupyter notebook on your laptop

https://github.com/lse-st446/lectures2020/blob/master/Week02/class/README.md

References

- Google file system:
  - Ghemawat, S., Gobioff, H. and Leung S.-T., The Google file system, SOSP 2003
  - GFS: Evolution on Fast-Forward, ACM Queue, Vol 7, No 7, August, 2009

- Apache Hadoop HDFS:
  - Shvachko, K., Kuang, H., Radia, S., and Chansler, R., The Hadoop Distributed File System, IEEE MSST 2010; see also html
  - Apache Hadoop docs: HDFS Architecture

- Windows Azure storage:
  - Calder, B., et al, Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency, SOSP 2011
  - Huang, C., Simitci, H., Xu, Y., Ogus, A., Caider, B., Gopalan, P., Li, J. and Yekhanin, S., Erasure Coding in Windows Azure Storage, USENIX 2012

References (cont'd)

- Amazon Dynamo:
  - DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Voshall, P. and Vogels, W., Dynamo: Amazon's Highly Available Key-value Store, SOSP 2007
  - Amazon Web Services, Amazon DynamoDB
  - Amazon Web Services, Amazon DynamoDB documentation

- Bigtable:
  - Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E., Bigtable: A distributed storage system for structured data, OSDI 2006
  - Google Cloud, Python Client Library Overview
  - Google Cloud, Bigtable Native API
  - Google Cloud, Cloud Bigtable Documentation

References cont'd

- Some key-value stores:
  - Lakshman, A. and Malik, K., A Decentralized Structured Storage System, LADIS 2009
  - Ellis, J., Facebook's Cassandra paper
  - Voldemort
  - Nishtala, R. et al, Scaling Memcache at Facebook, NSDI 2013
  - Fitzpatrick, B., Distributed Caching with Memcached, Linux Journal, 2004
- Cloud storage services:
  - Amazon Web Services, AWS Storage Services Overview: A Look at Storage Services Offered by AWS, November 2015
  - Microsoft Azure, Introduction to Microsoft Azure Storage
  - Google Cloud Platform, Google Cloud Storage Options

Books

- White, T., Hadoop: The Definitive Guide, 4th Edition, O'Reilly, 2015
    - Ch 3: The Hadoop distributed file system
    - Ch 5: Hadoop I/O
    - Ch 10: Setting Up a Hadoop Cluster, Network Topology (p. 286-288)
    - Ch 20: Hbase
- Carpenter, J. and Hewitt, E., Cassandra: The Definitive Guide, 2nd Edition, O'Reilly, 2016