Data Storage in the Cloud

KHALID ELGAZZAR
GOODWIN 531
ELGAZZAR@CS.QUEENSU.CA

Outline

1. Distributed File Systems
    - 1.1. Google File System (GFS)
2. NoSQL Data Store
    - 2.1. BigTable

References

The Google File System

S. Ghemawat, H. Gobioff, and S. Leung

SOSP 2003.

Bigtable: A Distributed Storage System for Structured Data

F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber.

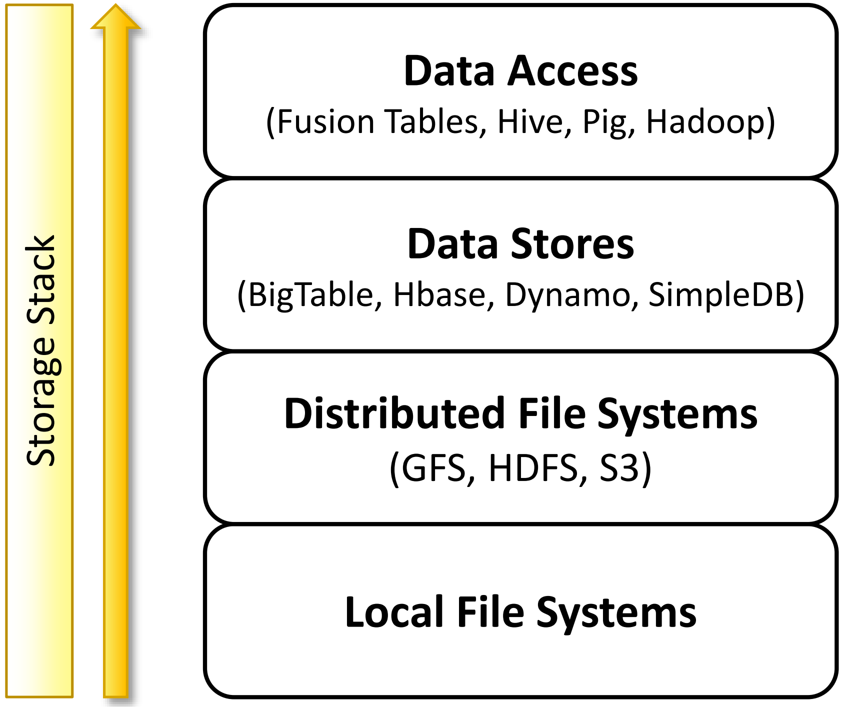
OSDI 2006.

Scalable SQL and NoSQL Data Stores

R. Cattell

SIGMOD Record 39(4).

Storage Stack



Storage Stack
Data Access
(Fusion Tables, Hive, Pig, Hadoop)
Data Stores
(BigTable, Hbase, Dynamo, SimpleDB)
Distributed File Systems
(GFS, HDFS, S3)
Local File Systems

1. Distributed File Systems

DFS Functionality

- Persistent stored data sets
- Hierarchical name space visible to all processes
- API that provides:
  - access and update operations on persistently stored data sets
  - Sequential access model (with additional random facilities)
- Sharing of data between users, with access control
- Concurrent access:
  - certainly for read-only access
  - what about updates?
- Mountable file stores

DFS Design Considerations

- Transparency
- Concurrency
- Replication
- Heterogeneity
- Fault Tolerance
- Consistency
- Efficiency

DFS Design Considerations (cont.)

- Transparency
  - *Access*
    - Programs are unaware of distribution of files)
  - *Location*
    - **Location transparency** – file name does not reveal the file’s physical storage location.
    - **Location independence** – file name does not need to be changed when the file’s physical storage location changes.
  - *Mobility*
    - Automatic relocation of files is possible (neither client programs nor system admin tables in client nodes need to be changed when files are moved).
  - *Performance*
    - Satisfactory performance across a specified range of system loads
  - *Scaling*
    - Service can be expanded to meet additional loads or growth.

DFS Design Considerations (cont.)

Concurrency

- Changes to a file by one client should not interfere with the operation of other clients simultaneously accessing or changing the same file.
- Concurrency properties
  - Isolation
  - File-level or record-level locking
  - Other forms of concurrency control to minimise contention

DFS Design Considerations (cont.)

Replication

- File service maintains multiple identical copies of files
- Load-sharing between servers makes service more scalable
- Local access has better response (lower latency)
- Fault tolerance
  - Full replication is difficult to implement.
  - Caching (of all or part of a file) gives most of the benefits (except fault tolerance)

DFS Design Considerations (cont.)

Heterogeneity

- Service can be accessed by clients running on (almost) any OS or hardware platform.
- Design must be compatible with the file systems of different OS's
- Service interfaces must be *open* - precise specifications of APIs are published.

DFS Design Considerations (cont.)

Fault Tolerance

- Service must continue to operate even when clients make errors or crash.
- at-most-once semantics
- at-least-once semantics
  - requires idempotent operations
- Service must resume after a server machine crashes.
- If the service is replicated, it can continue to operate even during a server crash.

DFS Design Considerations (cont.)

Consistency

- Unix offers one-copy update semantics for operations on local files - caching is completely transparent.
- Difficult to achieve the same for distributed file systems while maintaining good performance and scalability
- Replication – eager or lazy
- Cached copies need to be checked or given time –to-live

DFS Design Considerations (cont.)

Security

- Must maintain access control and privacy as for local files.
  - based on identity of user making request
  - identities of remote users must be authenticated
  - privacy requires secure communication
- Service interfaces are open to all processes not excluded by a firewall.
  - vulnerable to impersonation and other attacks

DFS Design Considerations (cont.)

Efficiency

- Goal for distributed file systems is usually performance comparable to local file system.

1.1. Google File System (GFS)

Google Disk Farm

Early days…

…today

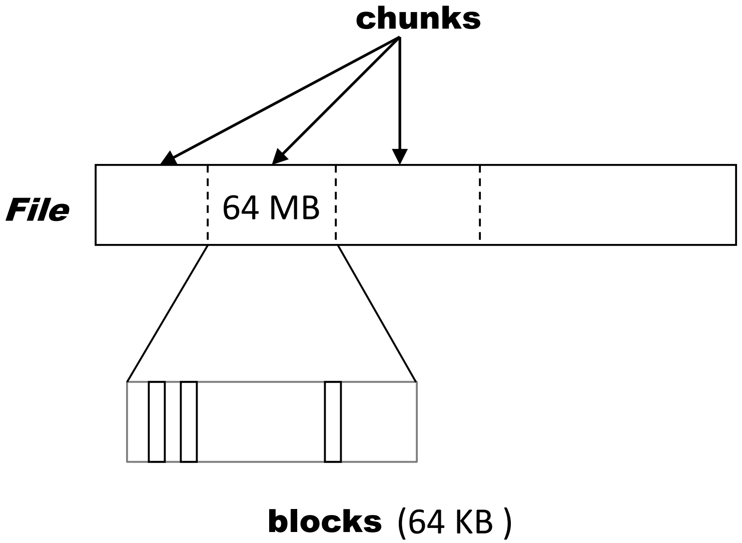
Motivation

- Google needed a good distributed file system
  - Redundant storage of massive amounts of data on cheap and unreliable computers

- Why not use an existing file system?
  - Google's problems are different from anyone else's
    - Different workload and design considerations
  - GFS is designed for Google apps and workloads
  - Google apps are designed for GFS

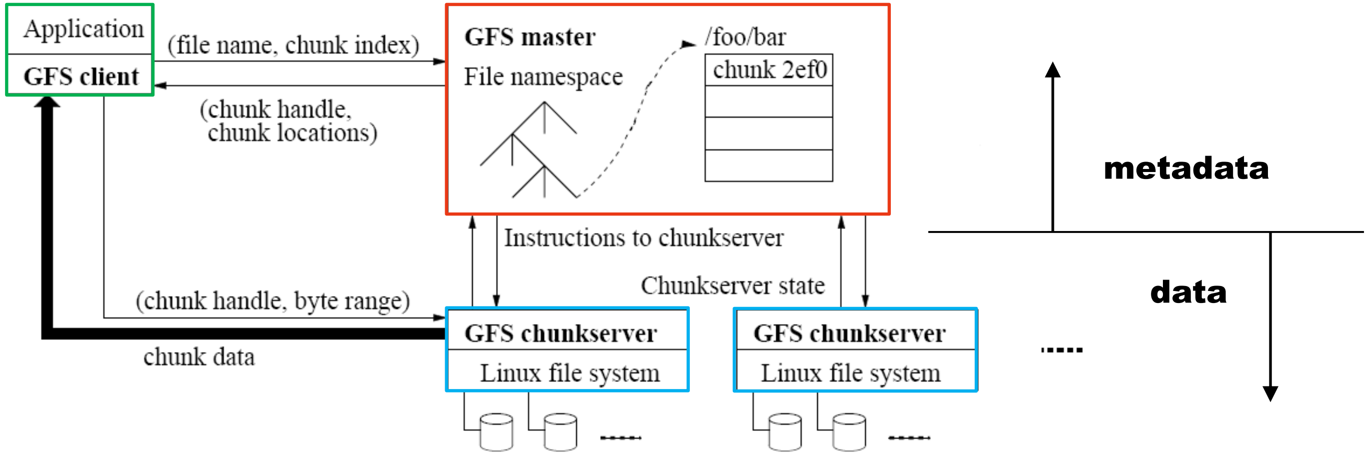
Design Considerations

- **Design factors**
  - Failures are common (built from inexpensive commodity components)
  - Files
    - large (extremely large, multi-GB)
    - mutation principally via appending new data
    - low-overhead atomicity essential
  - Co-design applications and file system API
  - Sustained bandwidth more critical than low latency
- **File structure**
  - Divided into chunks, 64 MB each
  - Chunks identified by 64-bit handle
  - Chunks replicated (3+ replicas)
  - Chunks divided blocks, 64KB each
  - Each block has a 32-bit checksum



GFS Architecture

- **Master**
  - Manages namespace/metadata
    - File and chunk namespaces
    - Mapping from files to chunks
    - Locations of each chunk's replicas
  - Manages chunk creation, replication, placement
  - Performs snapshot operation to create duplicate of file or directory tree
  - Performs checkpointing and logging of changes to metadata



- **Chunkservers**
  - Stores chunk data and checksum for each block
  - On startup/failure recovery, reports chunks to master
  - Periodically reports sub-set of chunks to master
    (to detect no longer needed chunks)

Single Master!

From distributed systems we know this is a:

- Single point of failure
- Scalability bottleneck

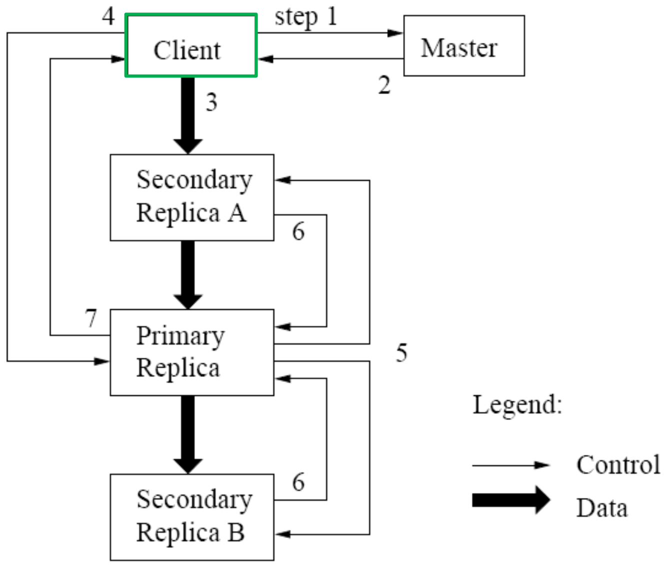
Google solution:

- Shadow masters
- Minimize master involvement
  - never move data through it, use only for metadata
    - and cache metadata at clients
  - large chunk size
  - master delegates authority to primary replicas in data mutations (chunk leases)

Simple, and good enough!

Mutation Operations

- Mutation = write or append
- Must be done for all replicas
- Goal: minimize master involvement
- Lease mechanism:
- master picks one replica as primary; gives it a "lease" for mutations
- primary defines a serial order of mutations
- all replicas follow this order
- Data flow decoupled from control flow



Mutation Operations

- **Primary replica**
  - Holds lease assigned by master (60 sec. default)
  - Assigns serial order for all mutation operations performed on replicas

- **Write operation**

1-2. client obtains replica locations and identity of primary replica

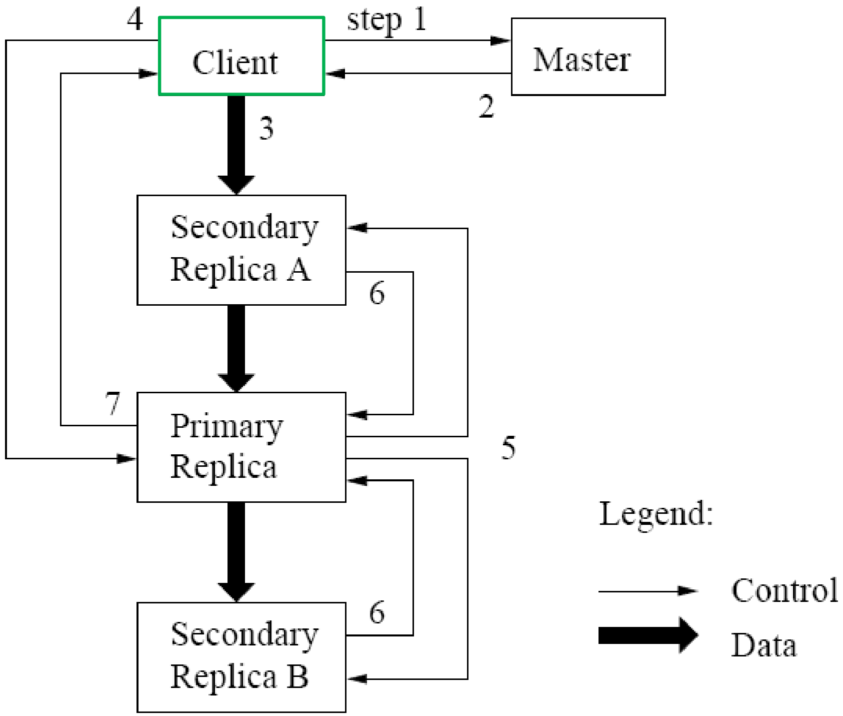
3. client pushes data to replicas (stored in LRU buffer by chunk servers holding replicas)

4. client issues update request to primary

5. primary forwards/performs write request

6. primary receives replies from replica

7. primary replies to client

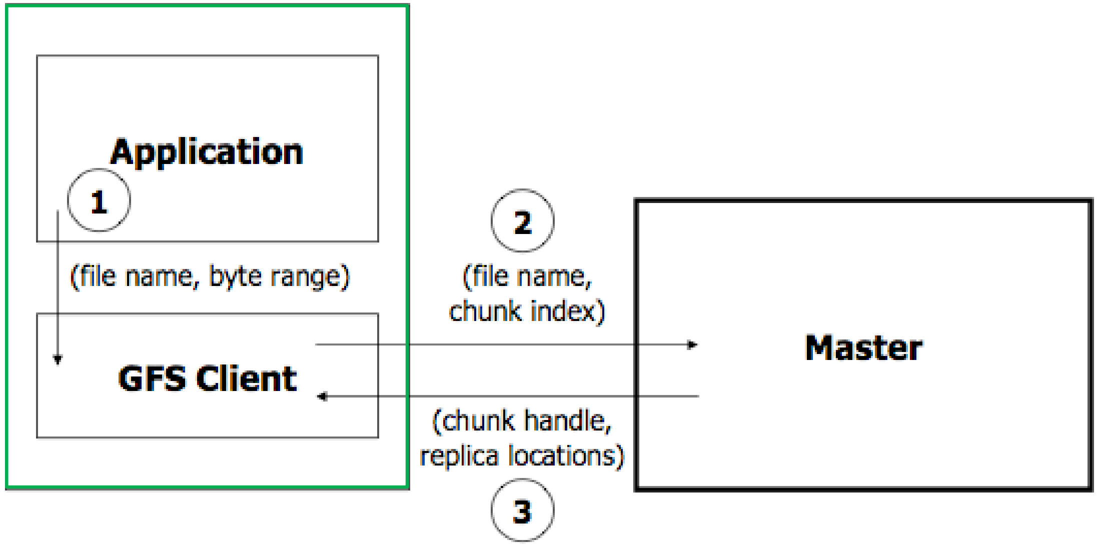


Read Algorithm

1. Application originates the read request

2. GFS client translates the request form (filename, byte range) -> (filename, chunk index), and sends it to master

3. Master responds with chunk handle and replica locations (i.e. chunkservers where the replicas are stored)

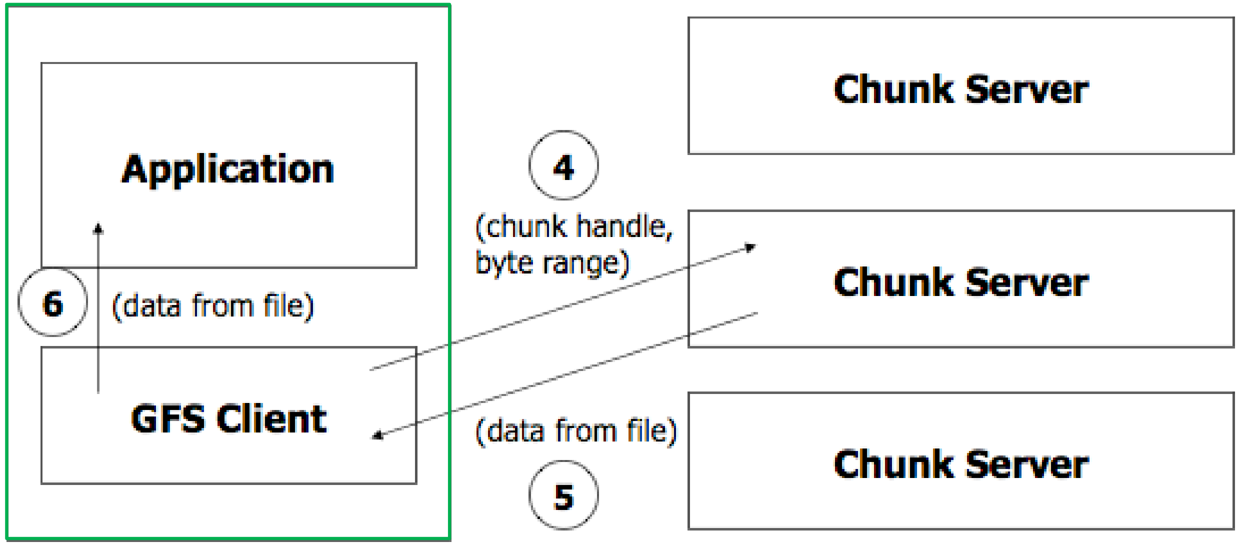


Read Algorithm (cont.)

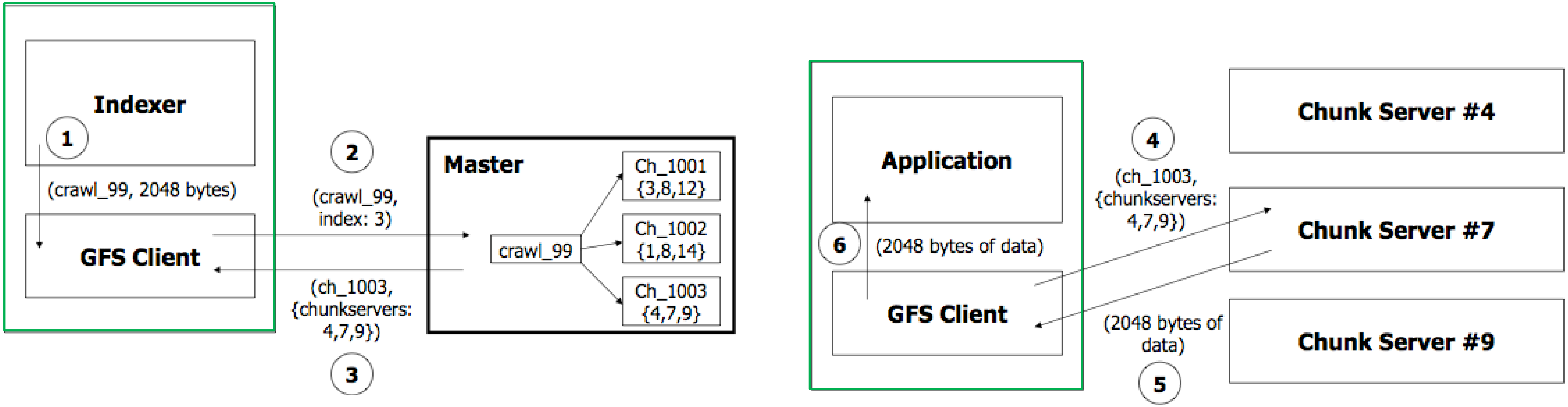
4. Client picks a location and sends the (chunk handle, byte range) request to the location

5. Chunkserver sends requested data to the client

6. Client forwards the data to the application



Read Example

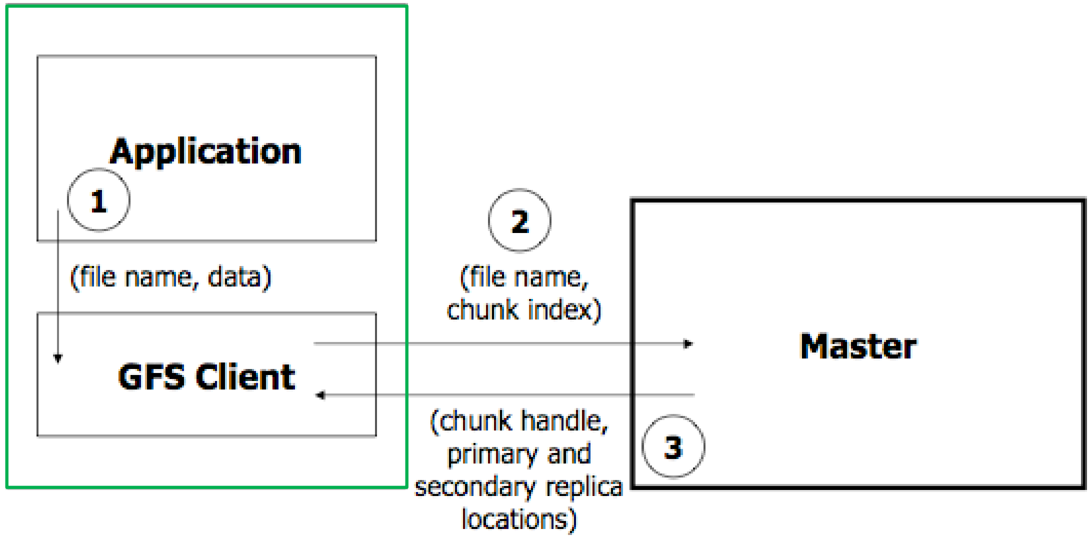


Write Algorithm

1. Application originates the request

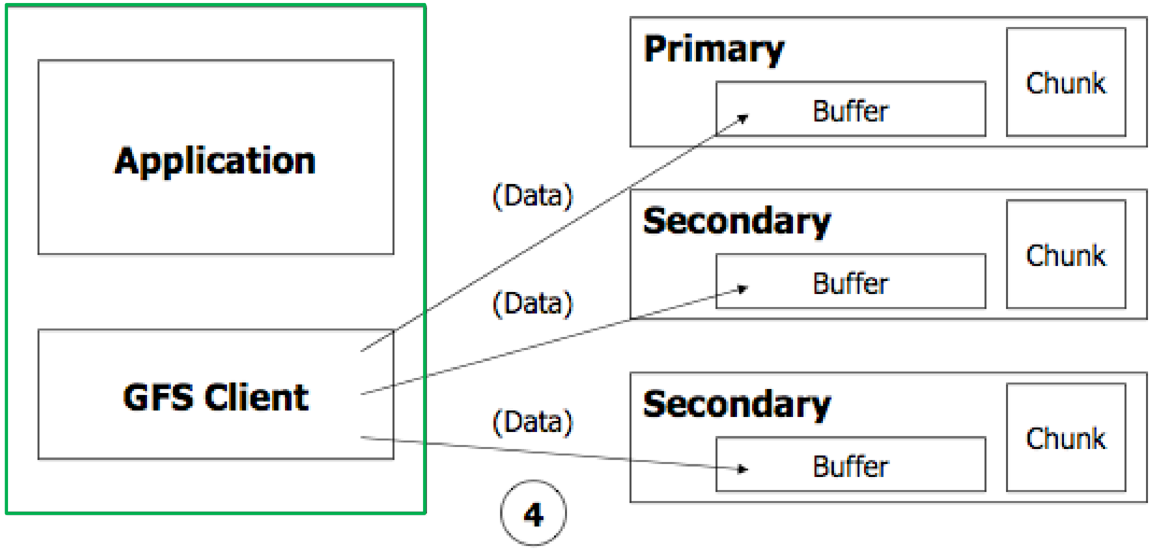
2. GFS client translates request from (filename, data) -> (filename, chunk index), and sends it to master

3. Master responds with chunk handle and (primary + secondary) replica locations



Write Algorithm (cont.)

4. Client pushes write data to all locations. Data is stored in chunkservers' internal buffers

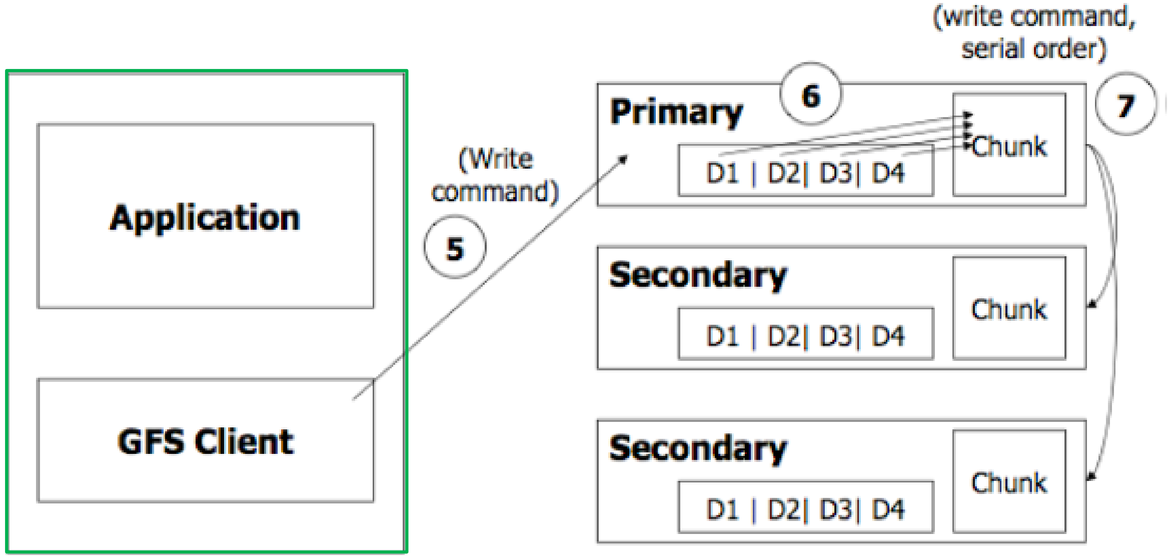


Write Algorithm (cont.)

5. Client sends write command to primary

6. Primary determines serial order for data instances stored in its buffer and writes the instances in that order to the chunk

7. Primary send the serial order to the secondaries and tell them to perform the write

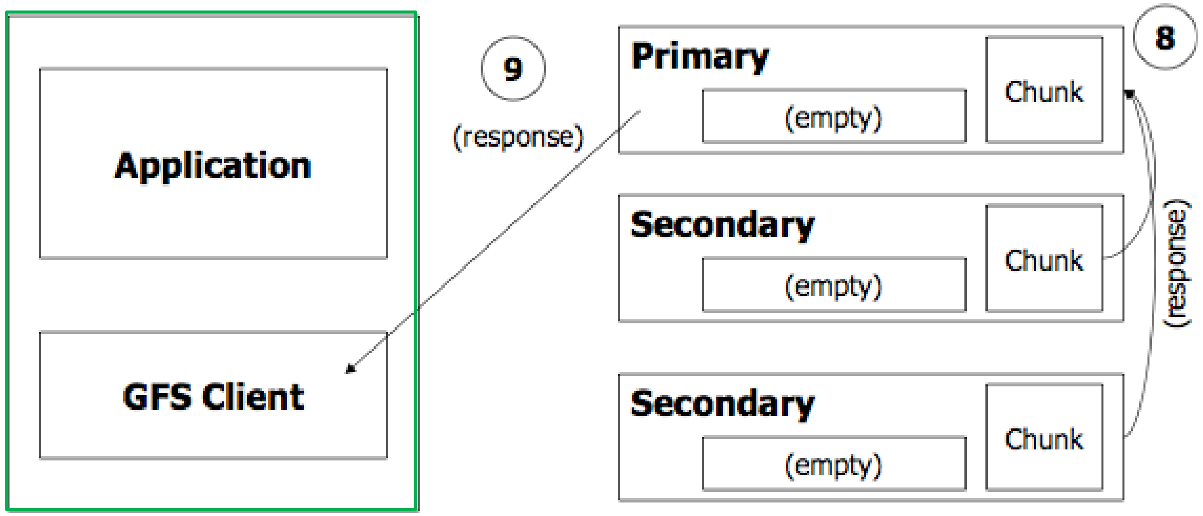


Write Algorithm (cont.)

8. Secondaries respond to the primary

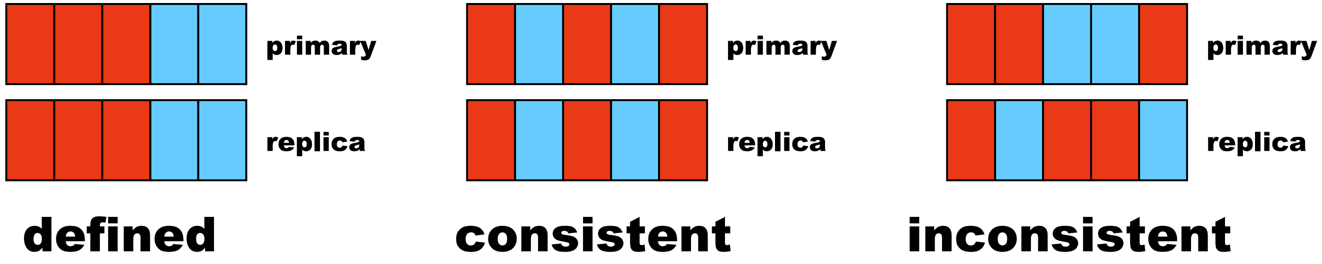
9. Primary respond back to the client

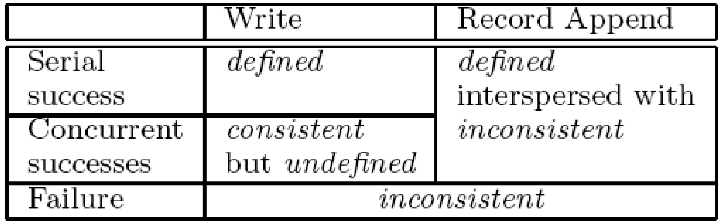
Note: If write fails at one of chunkservers, client is informed and retries the write



Consistency Guarantees

- Write
  - Concurrent writes may be consistent but undefined
  - Write operations that are large or cross chunk boundaries are subdivided by client into individual writes
  - Concurrent writes may become interleaved
- Record append
  - Atomically, at-least-once semantics
  - Client retries failed operation
  - After successful retry, replicas are defined in region of append but may have intervening undefined regions
- Application safeguards
  - Use record append rather than write
  - Insert checksums in record headers to detect fragments
  - Insert sequence numbers to detect duplicates





	Write	Record Append
Serial success	*defined*	*defined* interspersed with *inconsistent*
Concurrent successes	*consistent* but *undefined*	
Failure	*inconsistent*	

Metadata Management

- Namespace
  - Logically a mapping from pathname to chunk list
  - Allows concurrent file creation in same directory
  - Read/write locks prevent conflicting operations
  - File deletion by renaming to a hidden name; removed during regular scan
- Operation log
  - Historical record of metadata changes
  - Kept on multiple remote machines
  - Checkpoint created when log exceeds threshold
  - When checkpointing, switch to new log and create checkpoint in separate thread
  - Recovery made from most recent checkpoint and subsequent log
- Snapshot
  - Revokes leases on chunks in file/directory
  - Log operation
  - Duplicate metadata (not the chunks!) for the source
  - On first client write to chunk:
    - Required for client to gain access to chunk
    - Reference count > 1 indicates a duplicated chunk
    - Create a new chunk and update chunk list for duplicate

Logical structure

pathname	lock	chunk list
/home	read	Chunk4400488,…
/save		Chunk8ffe07783,…
/home/user/foo	write	Chunk6254ee0,…
/home/user	read	Chunk88f703,…

Chunk/replica management

- Placement
  - On chunkservers with below-average disk space utilization
  - Limit number of “recent” creations on a chunkserver (since access traffic will follow)
  - Spread replicas across racks (for reliability)

- Reclamation
  - Chunk become garbage when file of which they are a part is deleted
  - Lazy strategy (garbage college) is used since no attempt is made to reclaim chunks at time of deletion
  - In periodic “HeartBeat” message chunkserver reports to the master a subset of its current chunks
  - Master identifies which reported chunks are no longer accessible (i.e., are garbage)
  - Chunkserver reclaims garbage chunks

- Stale replica detection
  - Master assigns a version number to each chunk/replica
  - Version number incremented each time a lease is granted
  - Replicas on failed chunkservers will not have the current version number
  - Stale replicas removed as part of garbage collection

Fault Tolerance

- High availability
  - fast recovery
    - master and chunkservers are restartable in a few seconds
  - chunk replication
    - default: 3 replicas.
  - shadow masters
- Data integrity
  - checksum every 64KB block in each chunk

Deployment in Google

- Many GFS clusters
- Hundreds/thousands of storage nodes each managing petabytes of data
- GFS is under BigTable, etc.

Summary

GFS demonstrates how to support large-scale processing workloads on commodity hardware

Lessons to learn:

- design to tolerate frequent component failures
- optimize for huge files that are mostly appended and read
- feel free to relax and extend FS interface as required
- go for simple solutions (e.g., single master)

2. NoSQL Data Stores

What is NoSQL Data Stores?

New class of system designed to provide good horizontal scalability for simple database operations

- Relational DBMSs don’t scale well to 100’s or 1000’s of nodes.

NoSQL stands for

- “Not Only SQL”
- “Not Relational”

NoSQL Data Stores (Cont)

Key features (Cattell)

- Ability to horizontally scale to many servers
- Ability to replicate and distribute data over many servers
- Simple call level interface or protocol
- Transaction model weaker than ACID
- Efficient use of distributed indexes and RAM
- Ability to dynamically add new attributes to data records

Shared Nothing

NoSQL data stores use *shared nothing* horizontal scaling

- Nodes in the data store do not share any resources (i.e. CPU, disk, memory)

Allows data to be replicated and partitioned over many servers

Supports large number of simple read/write operations per second

BASE vs ACID

BASE – Basically Available, Soft state, Eventually consistent

- Updates are eventually propagated to replicas
- Limited guarantees on consistency of reads
- Tradeoff ACID properties for higher performance and scalability

BASE vs ACID (Cont)

CAP Theorem [Gilbert and Lynch]

- A shared data system can have at most 2 of the following properties:
  - Consistency – all readers see the same version of the data
  - Availability – system continues to respond to requests despite failures of components
  - Partition tolerance – system continues to operate despite network partitions
- NoSQL data stores generally give up consistency!

Replicated Data Consistency

D. Terry. Replicated Data Consistency Explained through Baseball. Communications of the ACM, Vol 56, No 12, December 2013.

Key Insights

- Although replicated cloud services generally offer strong or eventual consistency, intermediate consistency guarantees may better meet an application’s needs.
- Consistency guarantees can be defined in an implementation-independent manner and chosen for each read operation.
- Dealing with relaxed consistency need not place an excessive burden on application developers.

Replicated Data Consistency (cont)

Questions:

- Are different consistencies useful in practice?
- Can application developers cope with eventual consistency?
- Should cloud storage systems offer an even greater choice of consistency than the consistent and eventually consistent reads offered by some of today’s services?

Replicated Data Consistency (cont.)

Six consistency guarantees.

- *Strong Consistency:* See all previous writes.
- *Eventual Consistency:* See subset of previous writes.
- *Consistent Prefix:* See initial sequence of writes.
- *Bounded Staleness:* See all “old” writes.
- *Monotonic reads:* See increasing subset of writes.
- *Read My Writes:* See all writes performed by reader.

Types of Data Stores

Key-value stores

- Data values (of a single type) associated with a single key
- Keys used with distributed hashing to locate data
- Client interface provides insert, delete and index lookup
- Eg Voldemort, Membase

Types of Data Stores (cont.)

Document stores

- Values can be nested documents, lists or scalars
- Documents are self-describing, i.e. no schema
- Generally support secondary indexes and multiple types of documents
- Eg SimpleDB, MongoDB

Types of Data Stores (cont.)

Extensible record stores

- Hybrid records – has a schema to define a set of attributes but new attributes can be added on a record basis
- Records can be partitioned horizontally and vertically across nodes
- Eg. BigTable, Hbase, Cassandra

Types of Data Stores (cont.)

Relational Databases

- New RDBMSs that support horizontal scaling
- Scaling is comparable to what other types provide
  - Use small-scope operations (no joins over multiple nodes)
  - Use small-scope transactions (no 2PC across many nodes)
- Support ACID properties
- Eg. MySQL Cluster, VoltDB

2.1. Bigtable

Motivation

Distributed storage system for managing data at Google

- Reliably scales to petabytes (10^{15}) of data and thousands of machines
- Used for a wide variety of Google applications – App Engine, Analytics, News, Earth , Personalized search, etc.

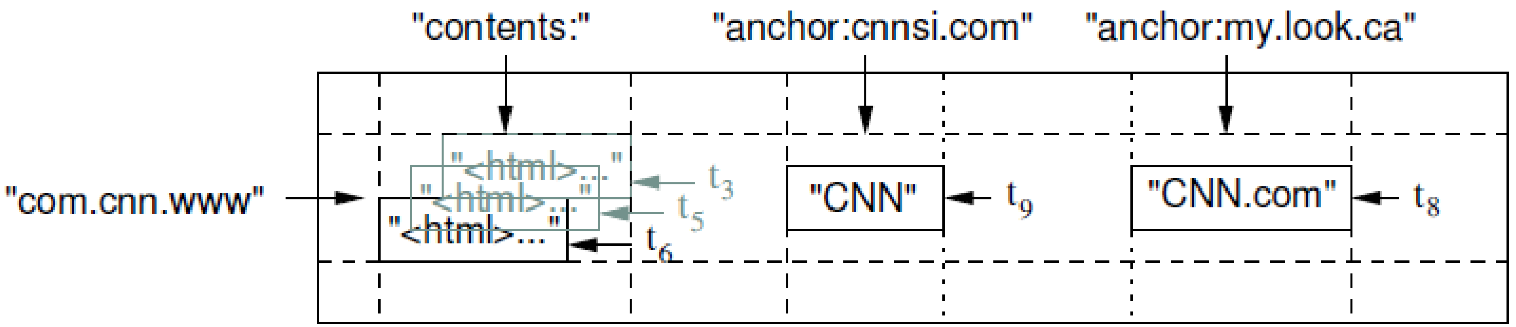
Like a DBMS but

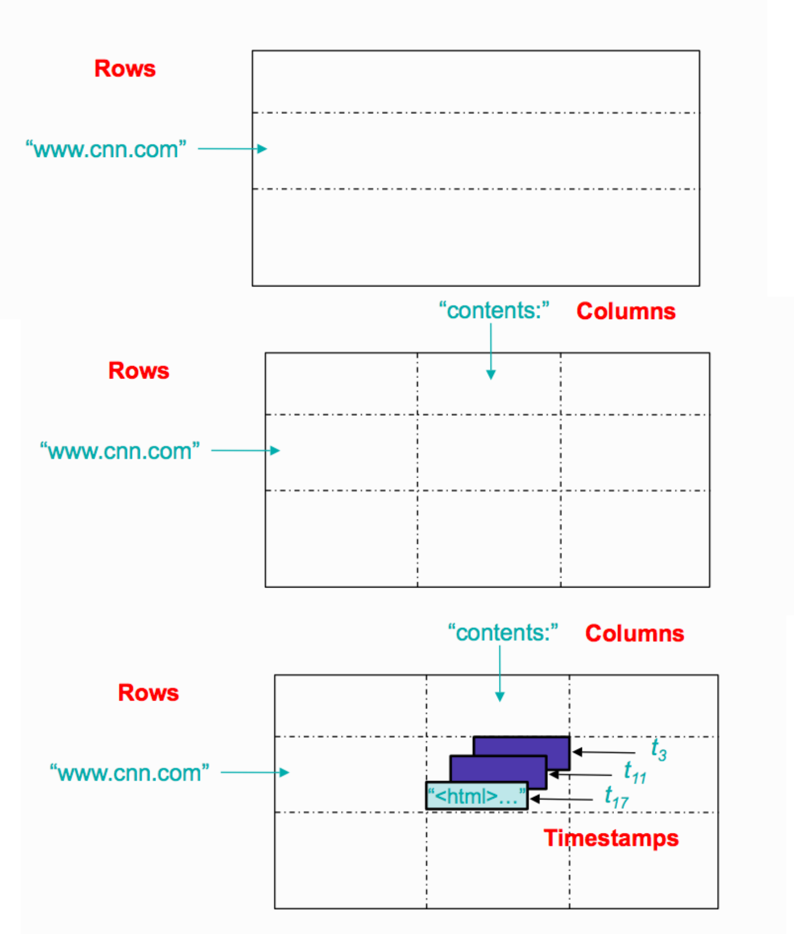
- Simple model and API
- Dynamic control over data layout, format, locality
- Data treated as uninterrupted strings

Data Model

Bigtable is a “sparse, distributed, persistent multidimensional sorted map”

Values in map indexed by <*row, column, time*>





Data Model (cont.)

Rows

- Row keys are arbitrary strings
- Data stored in lexicographic order by row keys
- Data partitioned into row ranges called *tablets*

Column families (f*amily:qualifier*)

- Data of same type
- Unit for access control, compression

Timestamps

- Each cell can contain multiple versions indexed by a timestamp

Building Blocks

Google File System

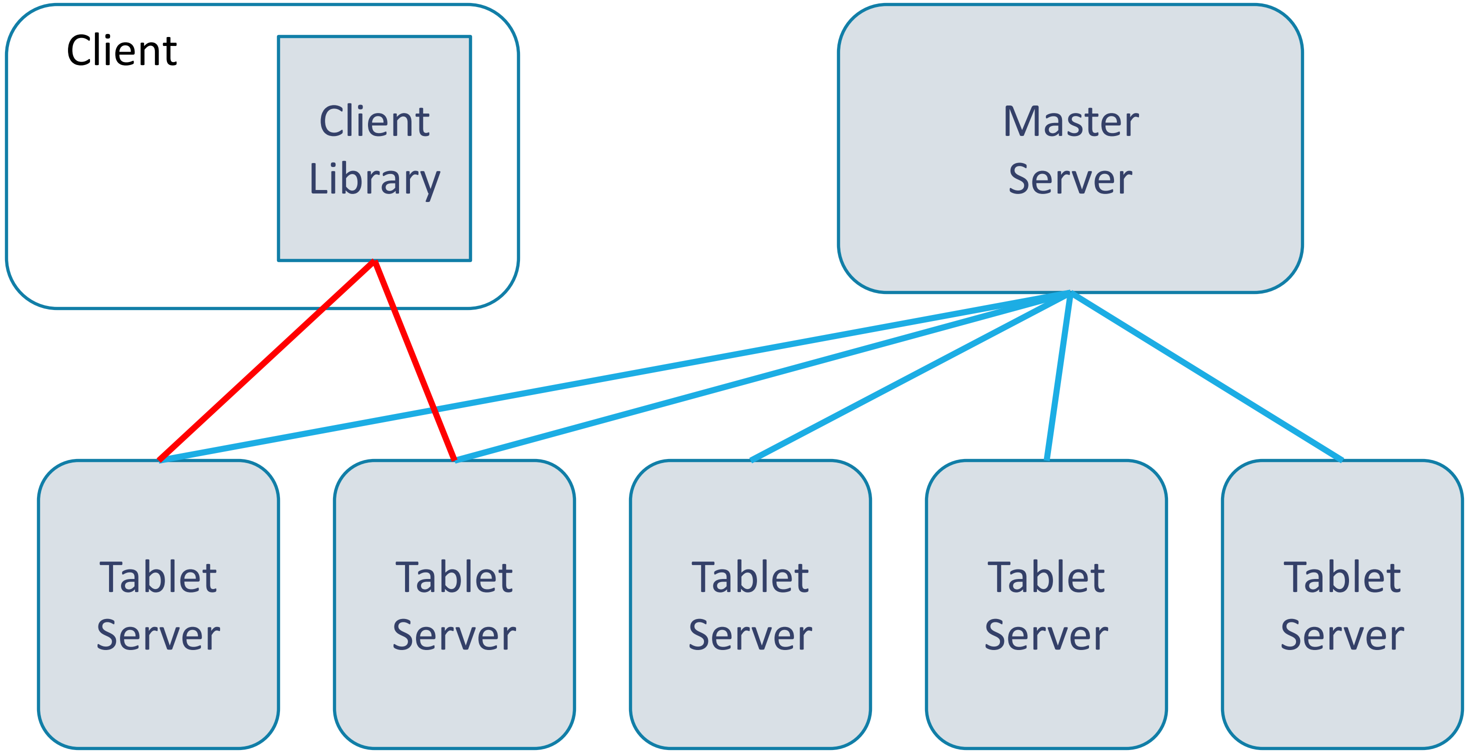
SSTable file format

- Each SSTable contains an number of blocks and a block index

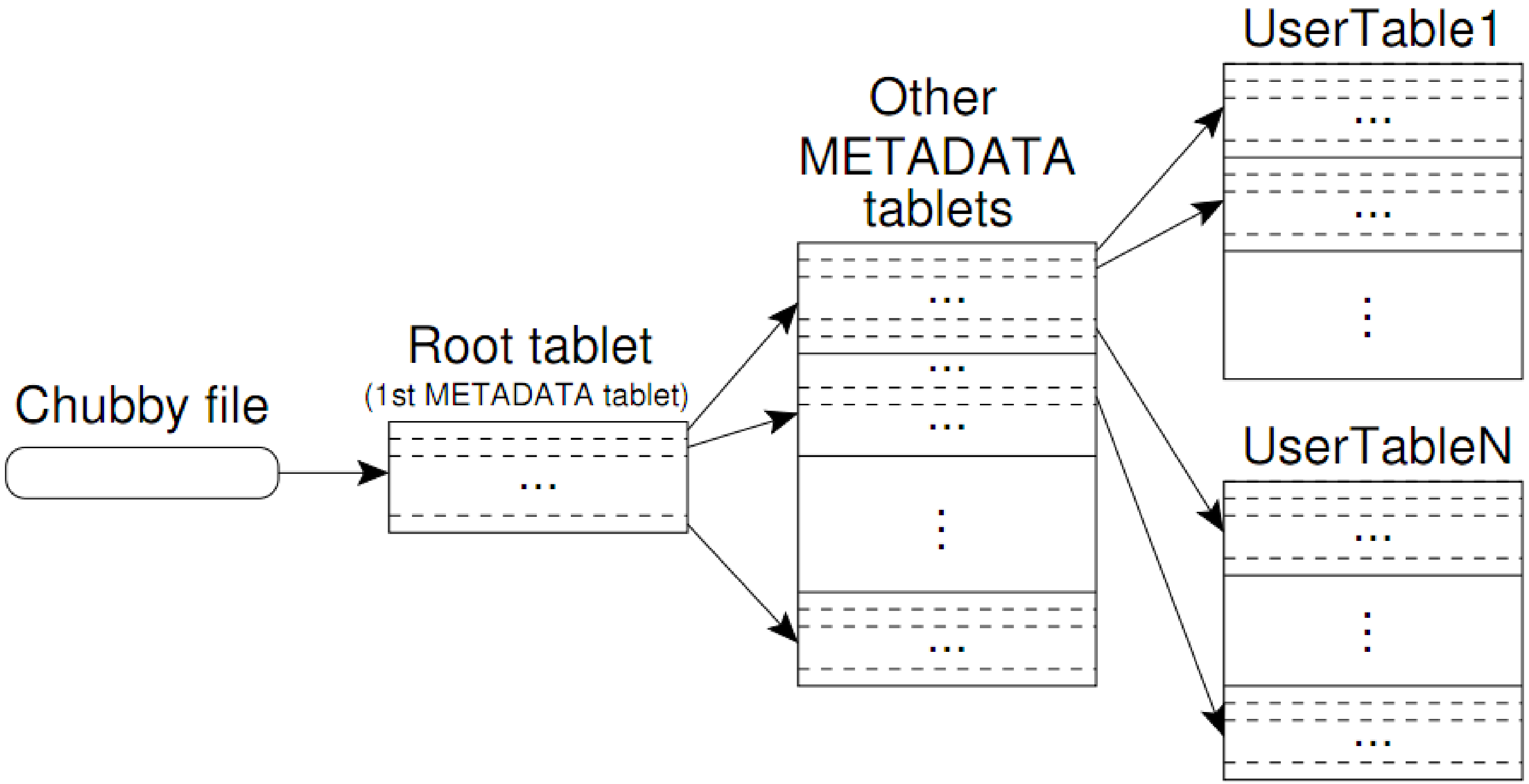
Chubby lock service

- Maintains a namespace of directories and files where each file can be used as a lock
- Used for variety of tasks by Bigtable
  - Ensure 1 active master; store bootstrap location; discover tablet servers, store schema and access control info

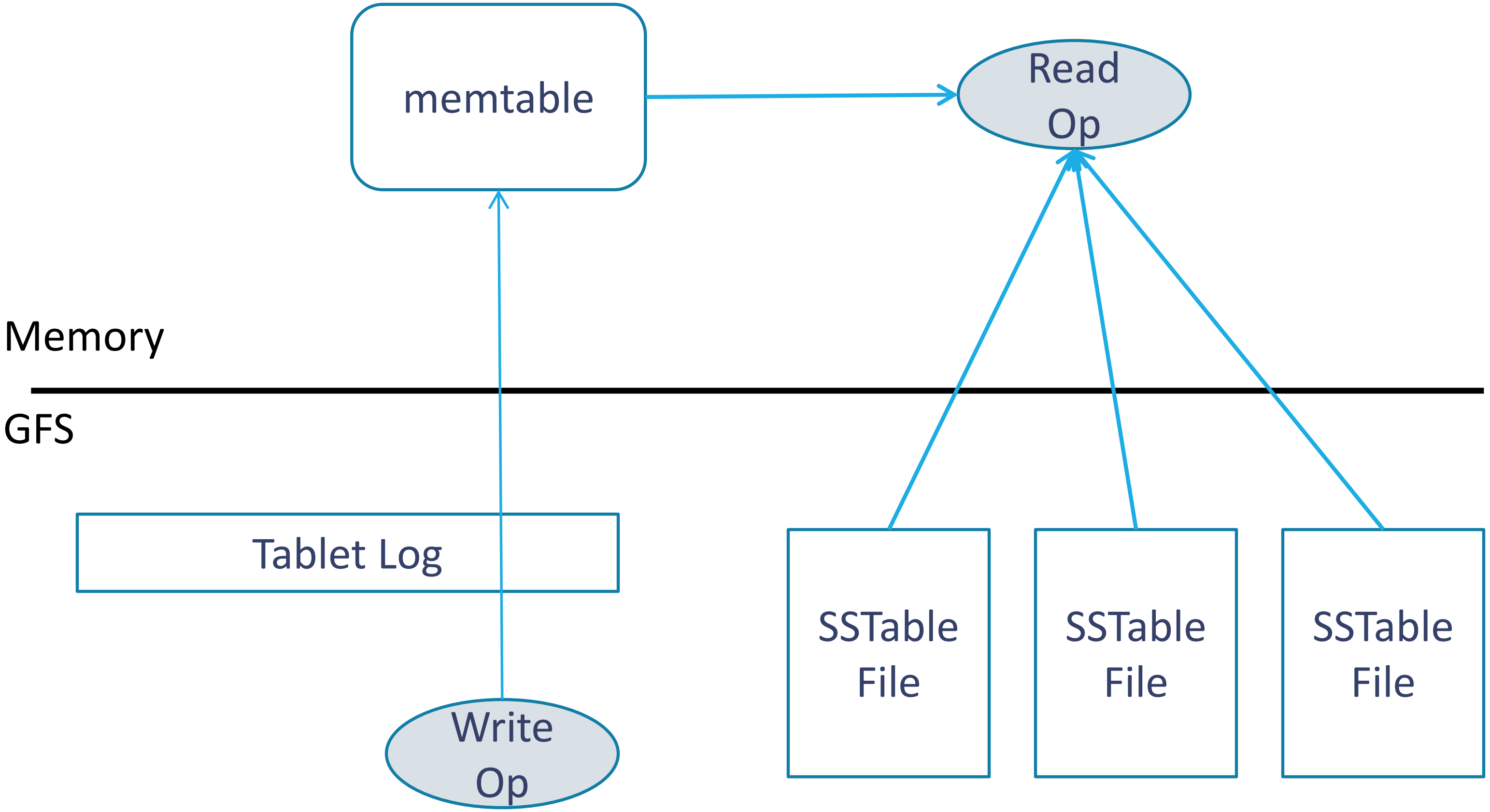
Implementation



Finding a Tablet



Tablet Implementation



Compactions

Minor compaction – convert the memtable into an SSTable

- Reduce memory usage
- Reduce log traffic on restart

Merging compaction

- Reduce number of SSTables
- Good place to apply policy “keep only N versions”

Major compaction

- Merging compaction that results in only one SSTable
- No deletion records, only live data

Locality Groups

Group column families together into an SSTable

- Avoid mingling data, ie page contents and page metadata
- Can keep some groups all in memory

Can compress locality groups

Bloom Filters on locality groups – avoid searching SSTable

Introduction to MapReduce

Reference

J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters, Proc of OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, December, 2004.
http://research.google.com/archive/mapreduce.html

Some slides based on
J. Zhao and J. Pjesivac-Grbovic, MapReduce: The programming Model and Practice, *SIGMETRICS Performance 2009 tutorial*.

What is MapReduce?

A programming model for large-scale distributed data processing

- Simple, elegant concept
- Restricted, yet powerful programming construct
- Building block for other parallel programming tools
- Extensible for different applications

Also an implementation of a system to execute such programs

- Take advantage of parallelism
- Tolerate failures and jitters
- Hide messy internals from users
- Provide tuning knobs for different applications

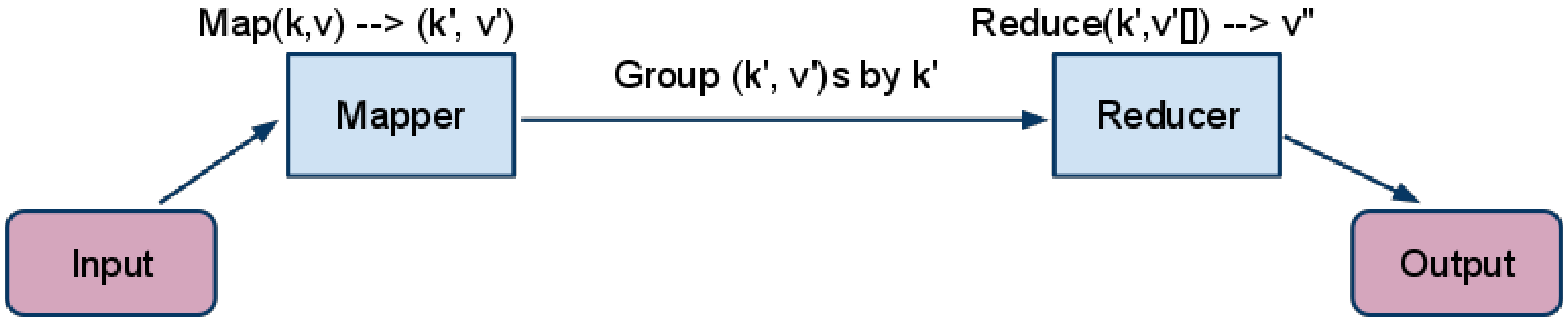
Population of Canada (a MapReduce task)

Programming Model

Inspired by Map/Reduce in functional programming languages

Computation takes set of input key/value pairs and produces set of output key/value pairs

Map and reduce functions written by user



Example – Count Word Occurrences

```
map(String key, String value):
        // key: document name,
        // value: document contents
        for each word w in value:
        EmitIntermediate(w, "1");

reduce(String key, Iterator values):
        // key: a word
        // values: a list of counts
        int word_count = 0;
        for each v in values:
        word_count += ParseInt(v);
        Emit(key, AsString(word_count));
```

Implementation

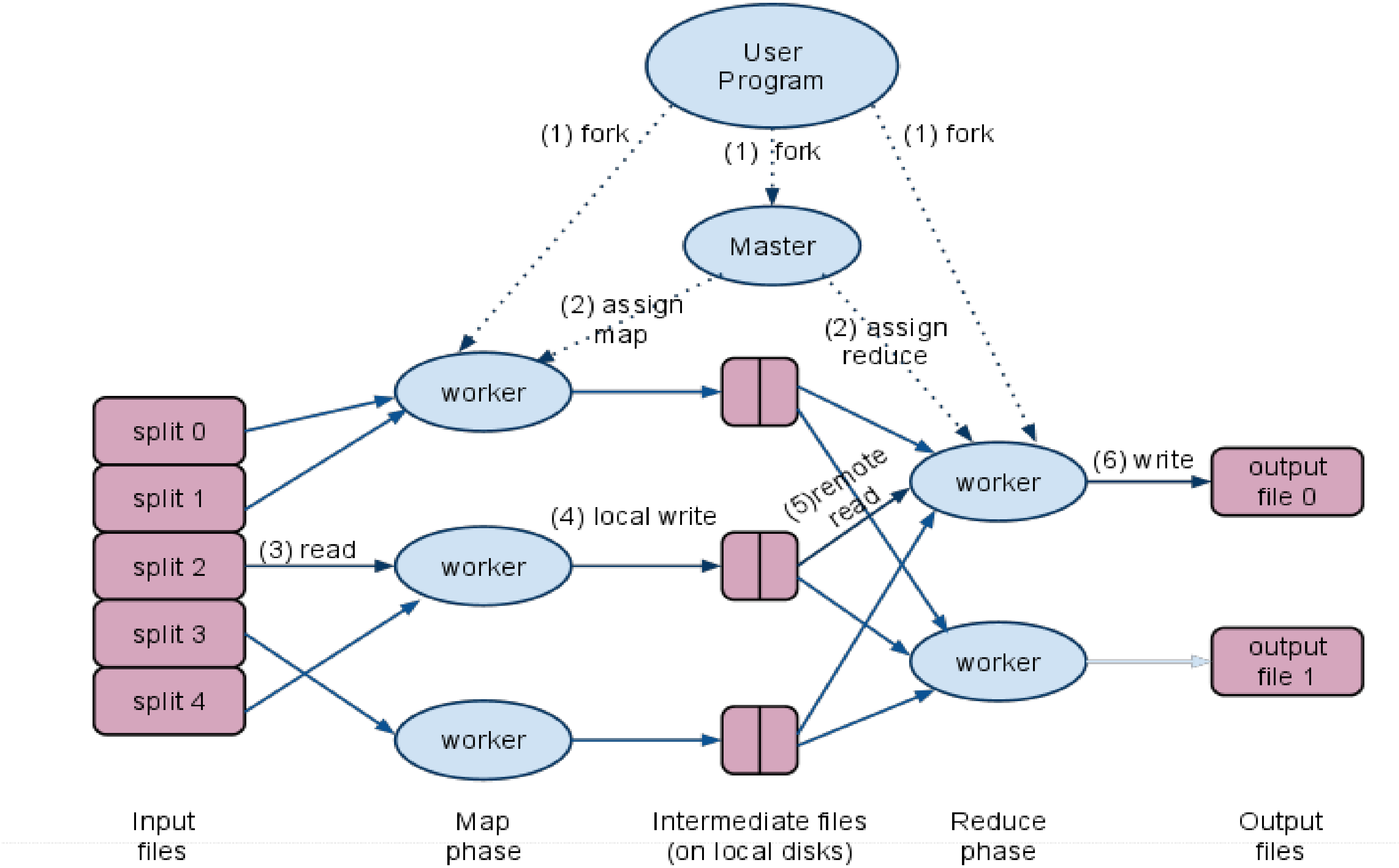
Google implementation targeted for large clusters of commodity PCs connected by switched Gigabit Ethernet

Cluster contains 1,000s of machines (so failure common!)

Storage is IDE disks connected to the machines

Google File System (GFS) used to manage storage

Implementation



Fault Tolerance

Worker failures

- Master pings workers – no response then failed
- Completed map tasks by failed worker are rescheduled
- Current map and reduce tasks are rescheduled

Master failures

- Single point of failure
- Could be restarted using checkpointing but currently just fails

Fault Tolerance (cont.)

Failure semantics

- Map and reduce functions deterministic functions of their input
- Atomic commits of map and reduce tasks

Locality

GFS divides each file into 64MB blocks and stores copies on different machines

Master attempts to schedule a map task on (or near) a machine that contains the input data

Task Granularity

Map and Reduce phases divided into ***M*** and ***R*** pieces, respectively

- ***M*** and ***R*** > > number of worker machines
  - Improves dynamic load balancing
  - Speeds up recovery when worker fails

In practice

- Choose ***M*** so that each task has input 16 – 64 MB (achieve locality)
- Choose ***R*** small multiple of number of worker machines (limits # of output files)

Handling Stragglers

Stragglers are a common cause of longer execution times

When close to completion, master schedules backup executions of remaining in-progress tasks and results taken from first one that finishes

MapReduce Inside Google

Large-scale web search indexing

Clustering problems for Google News

Produce reports for popular queries, e.g. Google Trend

Processing of satellite imagery data

Language model processing for statistical machine

translation

Large-scale machine learning problems

Tool to reliably spawn large number of tasks, e.g. parallel data backup and restore