IF3230 – Sistem Terdistribusi
Google File Systems

Achmad Imam Kistijantoro (imam@staff.stei.itb.ac.id)
Judhi Santoso (judhi@ staff.stei.itb.ac.id)
Anggrahita Bayu Sasmita (bayu.anggrahita@ staff.stei.itb.ac.id)

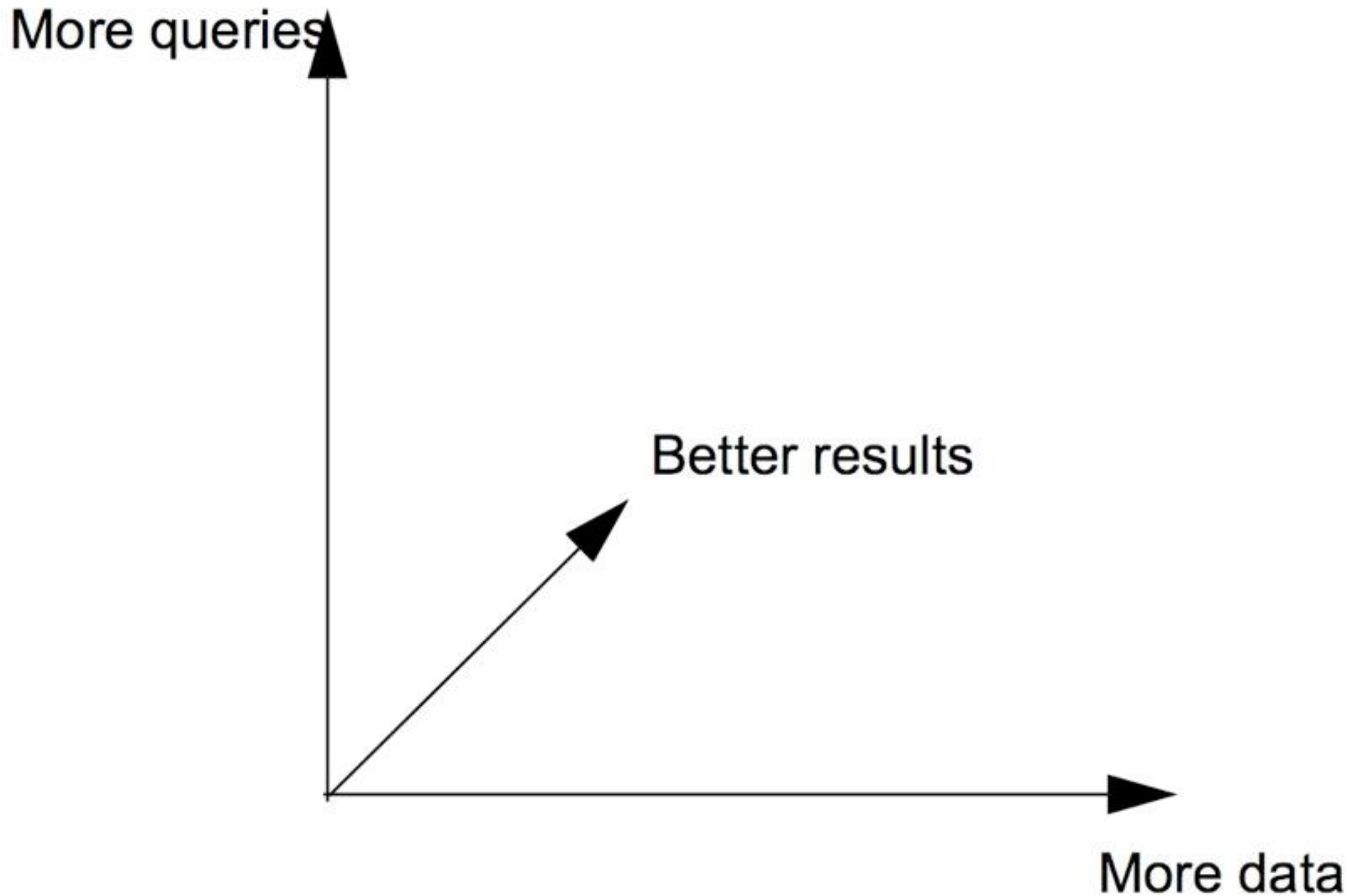
Goal

- Redundant => tolerate failures
- massive amount of data
- on cheap and unreliables computers
- assumption:
  - high component failure rates
  - huge files, each 100 MBs or larger
  - write once, mostly appended to
  - large streaming reads
  - high sustained throughput favored over low latency

Requirements of Google File System (GFS)

- Run reliably with failures.
- Solve problems that Google needs solved – Not a massive number of files but massively large files are common.
- Access is dominated by sequential reads and appends.
- Think of very large files each holding a very large number of HTML documents scanned from the web. These need read and analyzed.
- This is not your everyday use of a distributed file system (NFS and AFS).

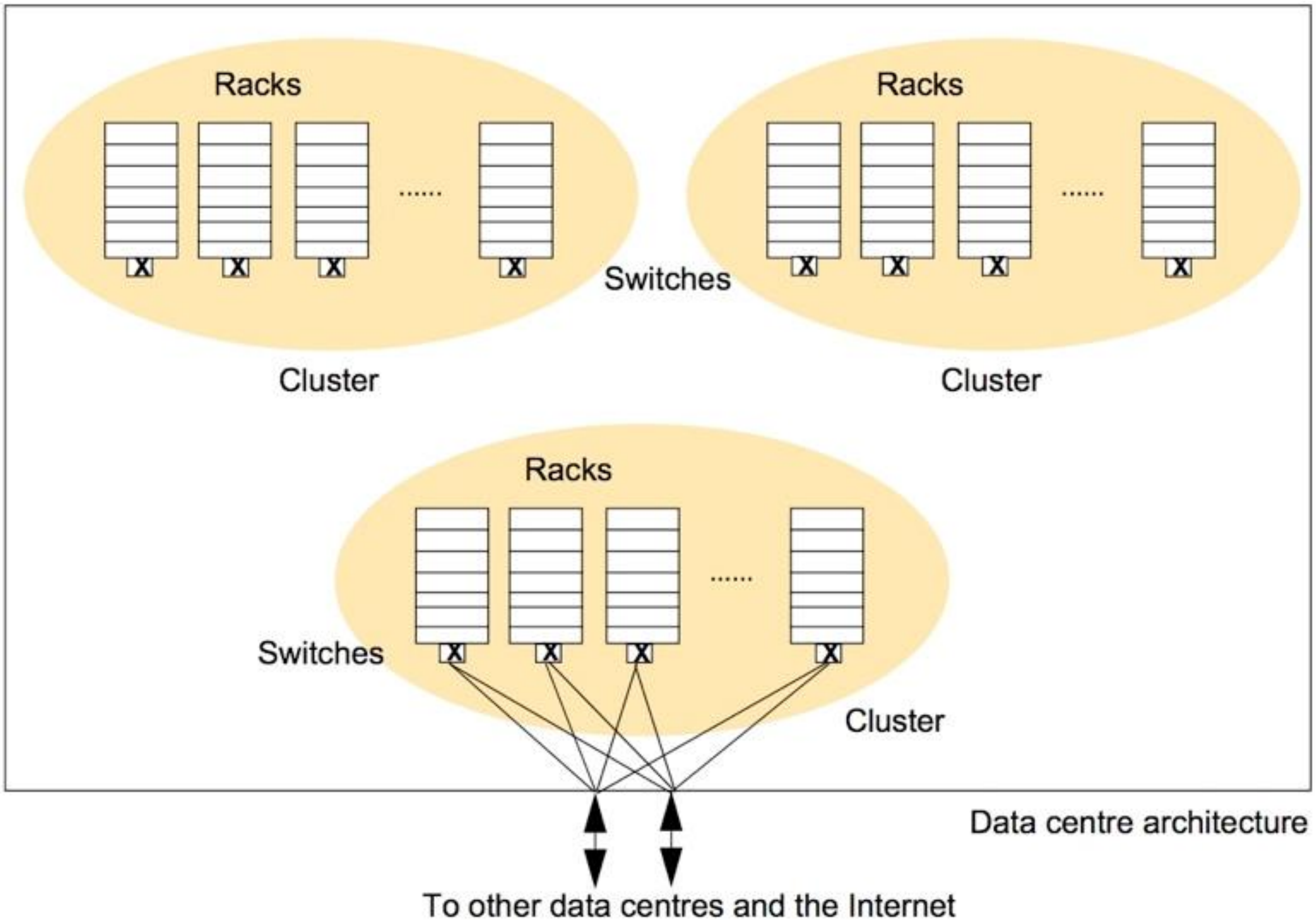
Scalability problem in Google



Simple Analysis

- Web consists of 20 billion web @ 20KB each
  - Total size: 400 terabytes
  - To crawl: with 1 computer, 30 MB/s => 4 months
  - With 1000 computers => less than 3 hours
  - Other jobs: indexing, ranking, searching

Organization of the Google physical infrastructure



(To avoid clutter the Ethernet connections are shown from only one of the clusters to the external links)

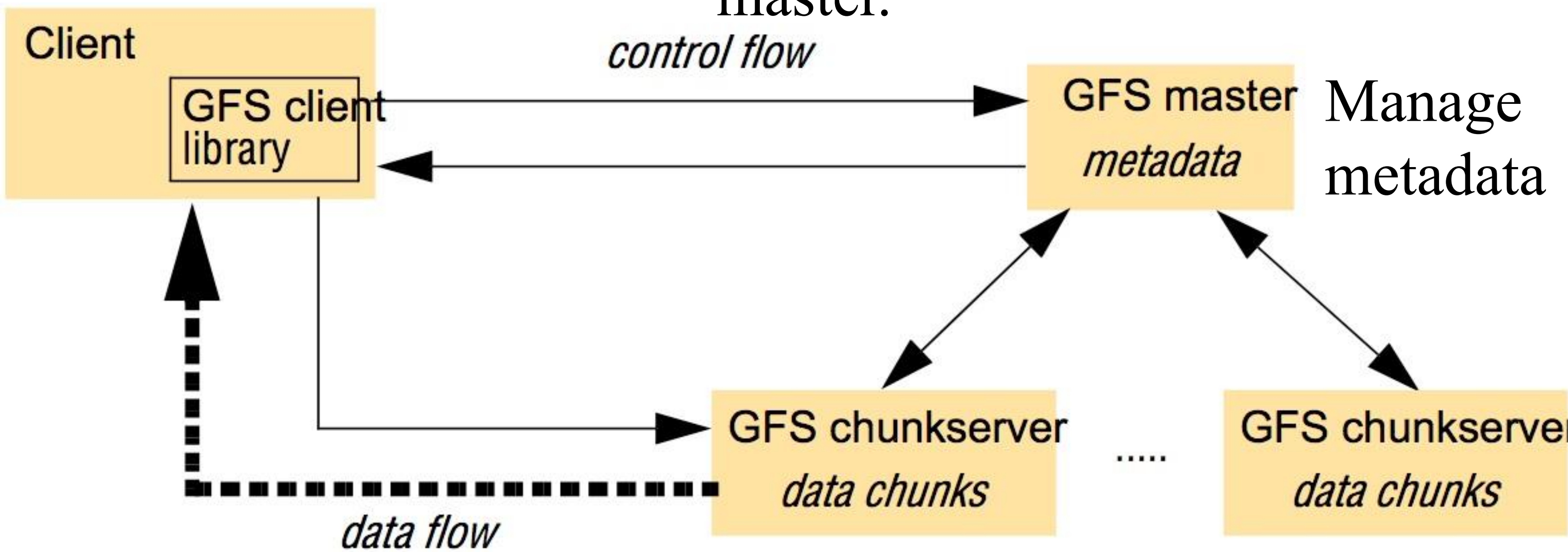
Commodity PC's which are assumed to fail. 40-80 PC's per rack. Racks are organized into clusters. Each cluster >30 racks. Each PC has >2 terabytes. 30 racks is about 4.8 petabytes. All of Google > 1 exabyte (10^18 bytes).

GFS Design decision

- files stored as chunks
  - fixed size (64MB)
- reliability through replication
  - each chunk replicated across 3+ chunk servers
- single master to coordinate access, keep metadata
  - simple centralized management
- no data caching
- customized API
  - add snapshot and record append operations

Overall architecture of GFS

Each GFS cluster has a single master.



Manage metadata

Hundreds of chunkservers

Data is replicated on three independent chunkservers.
Locations known by master.
With log files, the master is restorable after failure.

single master

- problem
  - single point of failure
  - scalability bottleneck
- GFS 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, good enough for google purpose

metadata

- global metadata is stored on the master
  - file and chunk namespaces
  - mapping from files to chunks
  - location of each chunk's replicas
- all in memory (64 bytes/chunk)
  - fast
  - easily accessible

metadata

- master has an operation log for persistent logging of critical metadata updates
  - persistent on local disk
  - replicated
  - checkpoints for faster recovery

master responsibility

- metadata storage
- namespace management/locking
- periodic communication with chunk servers
  - give instructions, collect state, track cluster's health
- chunk creation, re-replication, rebalancing
  - balance space utilization and access speed
  - spread replicas across racks to reduce correlated failures
  - re-replicated data if redundancy falls below threshold
  - rebalance data to smooth out storage and request load

master responsibility

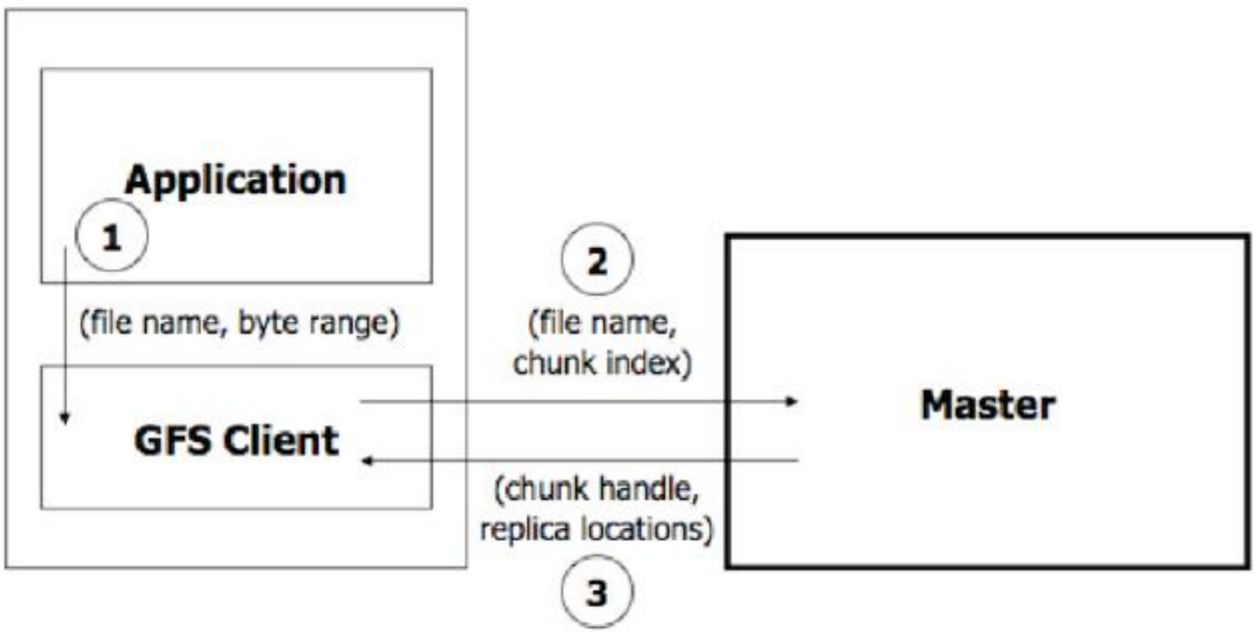
- garbage collection
  - simpler, more reliable than traditional file delete
  - master logs deletion, renames the file to a hidden name
  - lazily garbage collects hidden files
- stale replica deletion
  - detects stale replicas using chunk version numbers

mutations

- mutation = write or record append
  - must be done on all replicas
- goal: minimize master involvement
- lease mechanism:
  - master picks one replica as primary; gives it a lease for mutations
- data flow decoupled from control flow

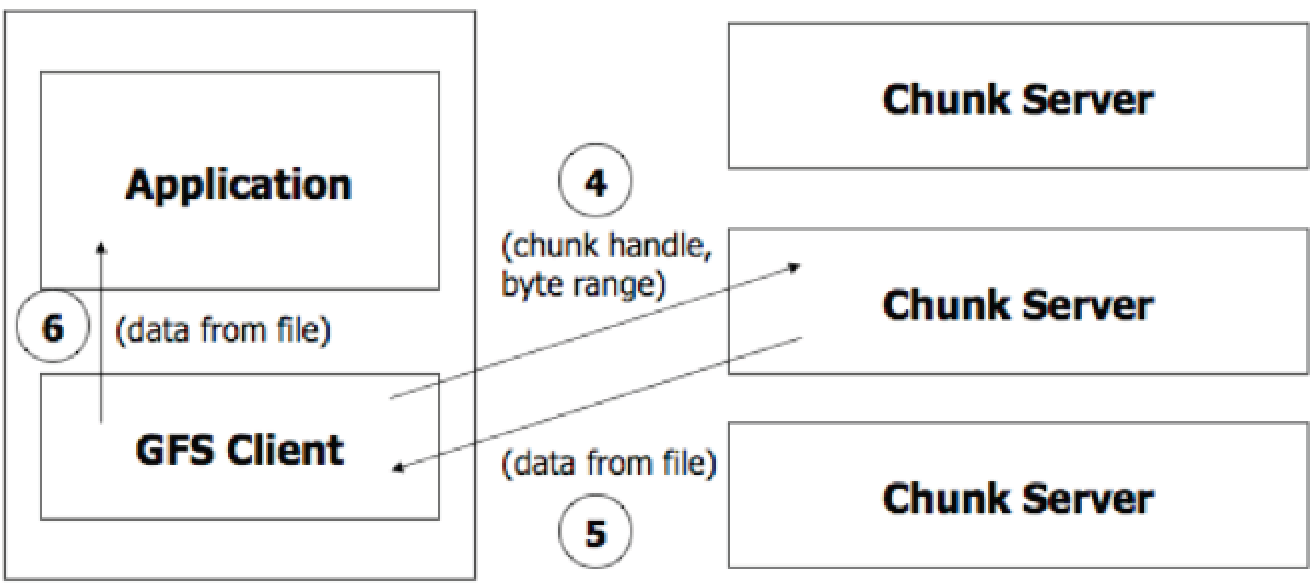
Read algorithm

- application originates the read request
- GFS client translates request and sends it to master
- master responds with chunk handle and replica locations



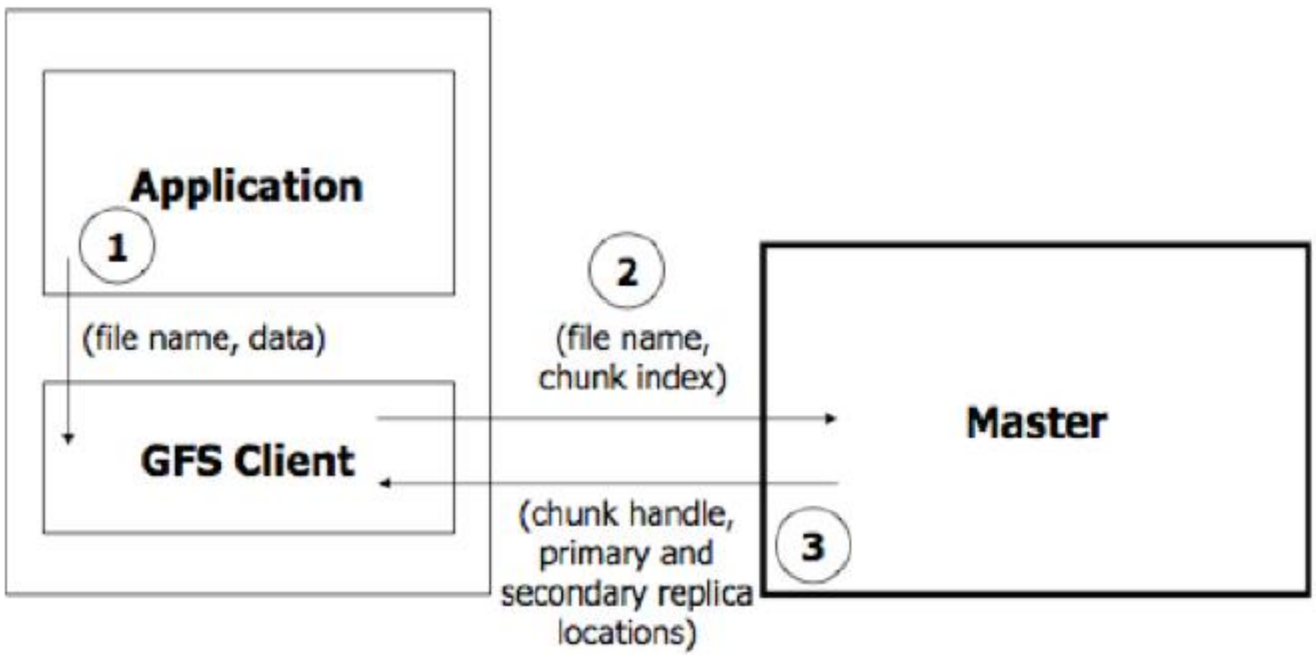
Read algorithm

- client picks a location and sends the request
- chunkserver sends requested data to the client
- client forwards the data to the application



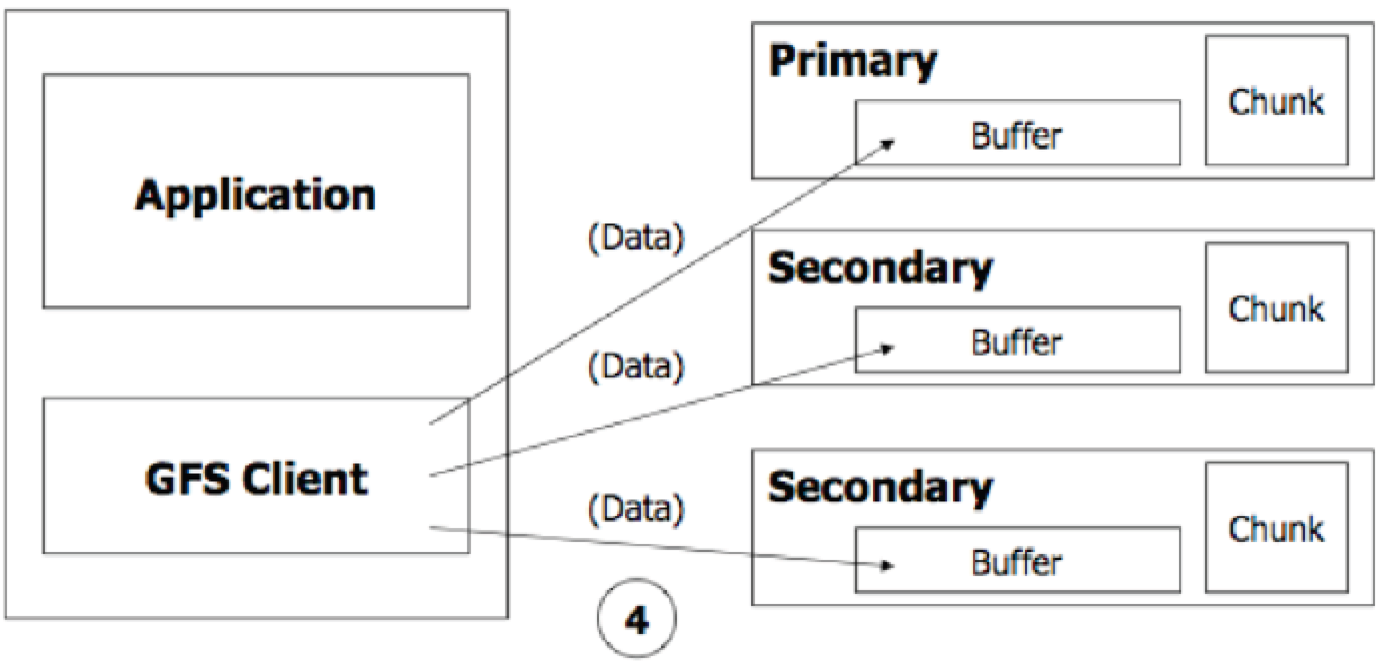
Write algorithm

- application originates the request
- GFS client translates the request and sends it to master
- Master responds with chunk handle and replica locations



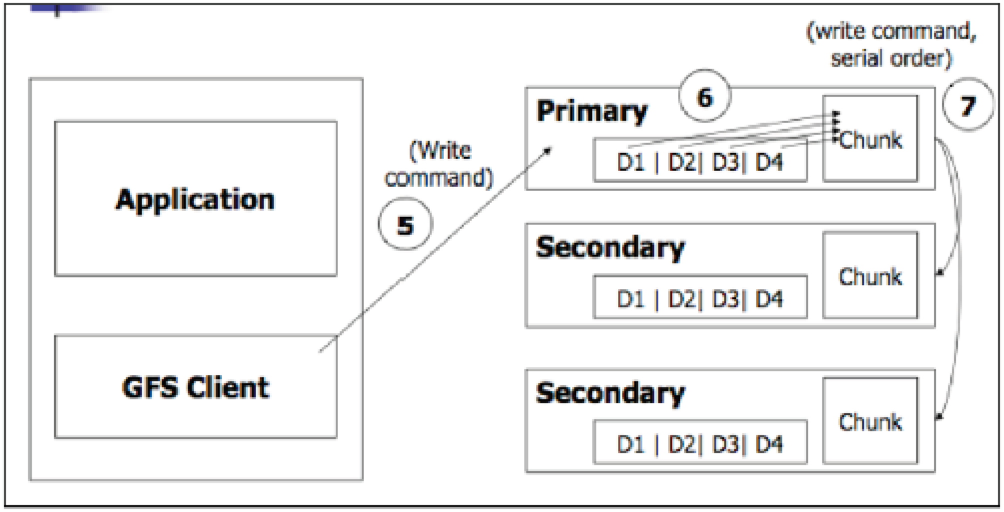
Write algorithm

- Client pushes write data to all locations. Data is stored in chunkservers' internal buffer



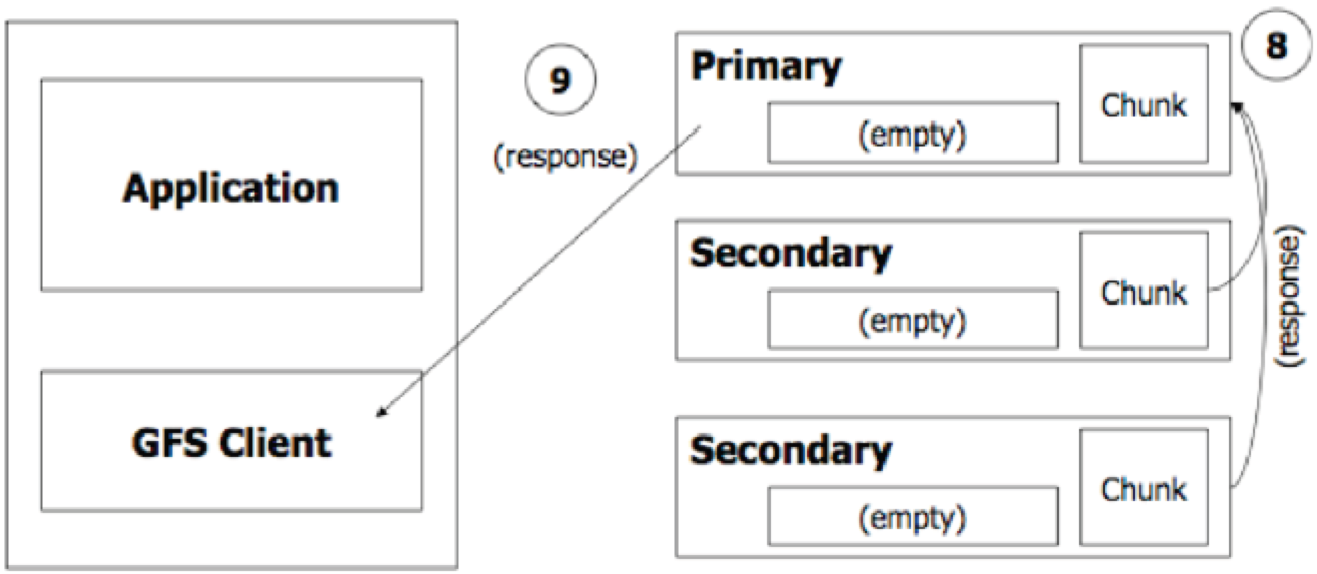
Write algorithm

- Client sends write command to primary
- primary determines serial order for data instances in its buffer and writes the instances in that order to the chunk
- primary sends the serial order to the secondaries and tells them to perform the write



Write algorithm

- Secondaries respond back to primary
- primary responds back to the client



Atomic record append

- GFS appends it to the file atomically at least once
  - GFS picks the offset
  - works for concurrent writers

Atomic record append

- Same as write, but no offset and
  - client pushes write data to all locations
  - primary checks if record fits in specified chunk
  - if the record doesn't fit:
    - pads the chunk
    - tells secondary to do the same
    - inform client and has the client retry
  - if the record fits, then the primary
    - appends the record
    - tells secondaries to do the same
    - receives responses and responds to the client

Relaxed consistency model

- Consistent = all replicas have the same value
- Defined = replica reflects the mutation consistent
- Some properties:
  - concurrent writes leave region consistent, but possibly undefined
  - failed writes leave the region inconsistent
- some work has moved into the applications:
  - e.g. self validating, self identifying records
- simple, efficient

Fault tolerance

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

Hadoop File System

- design assumptions
  - single machines tend to fail
    - hard disk, power supply
  - more machines = increased failure probability
  - data doesn't fit on a single node
  - desired: commodity hardware
  - built-in backup and failover

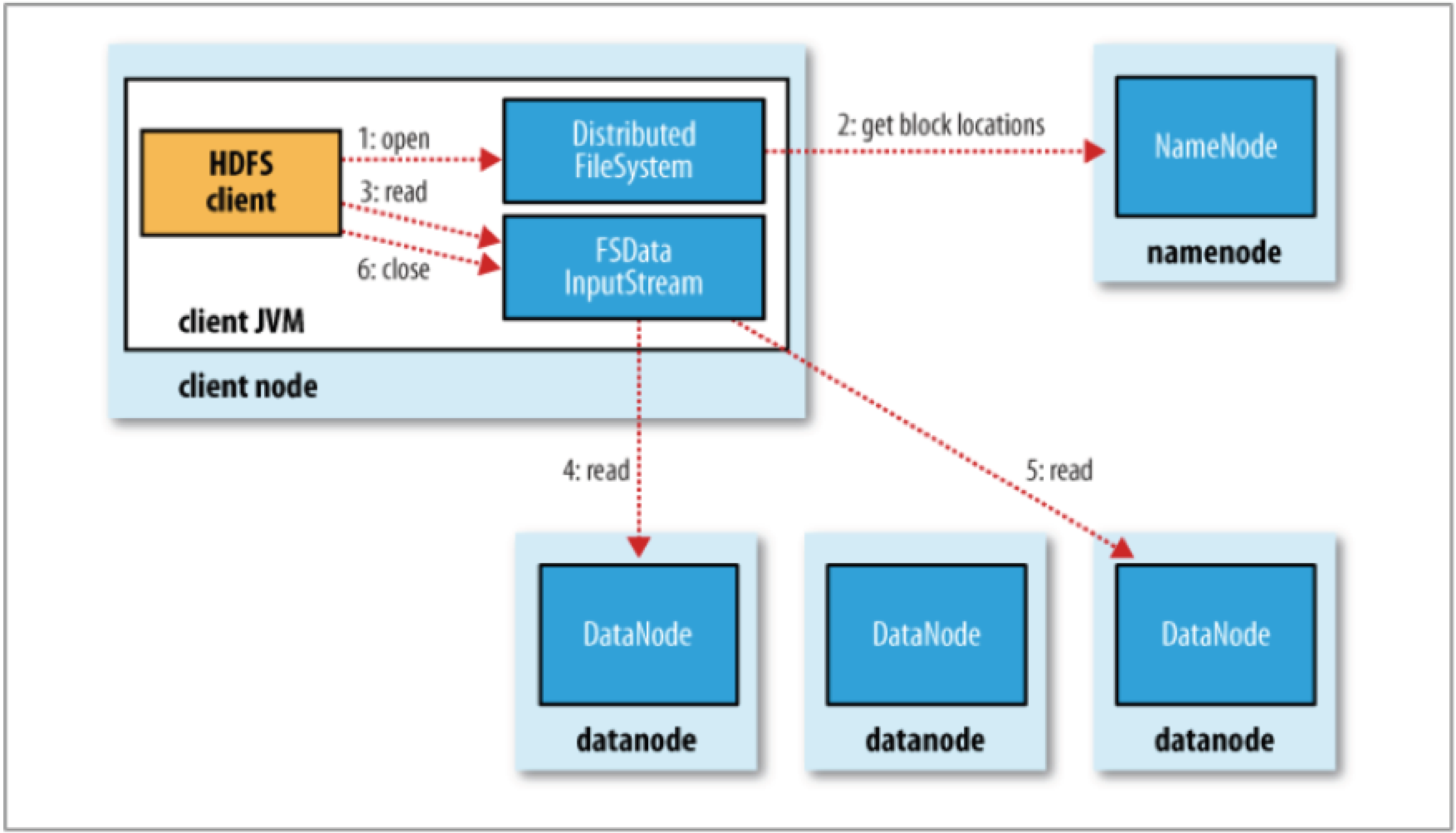
Namenode and datanodes

- Namenode (Master)
  - metadata
    - where file blocks are stored (namespace image)
    - edit (operation) log
  - secondary namenode (shadow master)
- Datanode (Chunkserver)
  - stores and retrieves blocks
  - reports to namenode with list of blocks they are storing

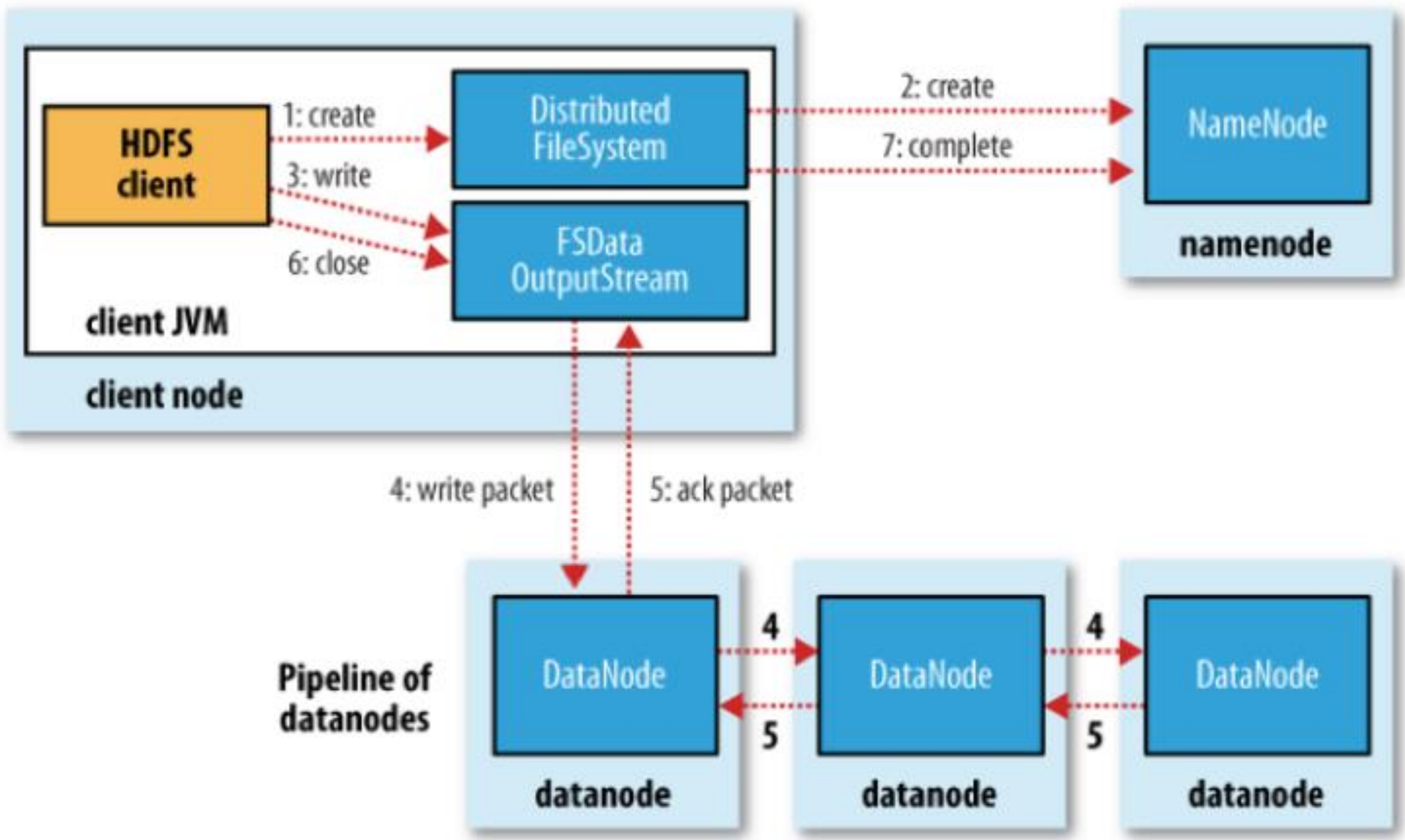
Noticeable differences from GFS

- Only single-writers per file
- no record append operation
- open source
  - provides many interfaces and libraries for different file systems

Anatomy of a file read



Anatomy of a file write



4

```java
// Print the contents of the HDFS file pathName to stdout.
public static void PrintHDFSFile(Context context, String pathName)
        throws IOException {

    // Load the HDFS library.
    Configuration conf = context.getConfiguration();
    FileSystem fs = FileSystem.get(conf);

    // Load the file input stream.
    Path hdfsPath = new Path(pathName);
    FSDataInputStream in = fs.open(hdfsPath);

    String line = null;
    while ((line = in.readUTF()) != null) {
        System.out.println(line);
    }

    in.close();
}
```

Examples of the use of MapReduce

Function	*Initial step*	*Map phase*	*Intermediate step*	*Reduce phase*
Word count		For each occurrence of word in data partition, emit <*word*, 1>		For each word in the intermediary set, count the number of 1s
Grep		Output a line if it matches a given pattern		Null
Sort *N.B. This relies heavily on the intermediate step*	*Partition data into fixed-size chunks for processing*	For each entry in the input data, output the key-value pairs to be sorted	*Merge/sort all key-value keys according to their intermediary key*	Null
Inverted index		Parse the associated documents and output a <*word*, document ID> pair wherever that word exists		For each word, produce a list of (sorted) document IDs

The overall execution of a MapReduce program