Distributed File System

Diego Rivera Correa

Slides owned by: Mirek Riedewald

Key Learning Goals

- What does the distributed file system (DFS) have in common with traditional single-machine file systems? What is different?
- Why is a single DFS master node unlikely to become a bottleneck?
- Do Hadoop MapReduce and Spark require the ability to update existing files?

Introduction

- The file system is an essential component of the operating system (e.g., Windows, Linux, MacOS). It stores data permanently, organized into files and directories.
- We cover the Google File System (GFS), which inspired the popular Hadoop File System (HDFS).
  - Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google file system. In *Proc. of the nineteenth ACM Symposium on Operating Systems Principles* (SOSP '03), 29-43

What Does a File System Do?

What Does a File System Do?

- Create, rename, delete a file or directory.
- Open or close a file.
- Jump to a specific position (an *offset*) in a file.
- Read or write a certain number of bytes in the file, starting from a given offset or the current position.
- Append new data at the end of a file.
- Move a file from one directory to another.

These are everyday operations!

What Is Special about a Distributed File System (DFS)?

- All those operations we just listed work great on your laptop. But what happens when your file is 10 TB and your disk is only 2 TB?

- To avoid I/O bottlenecks, files should reside on many servers.

- File size should not be limited by the capacity of a single disk drive. To achieve this, a big file should be chunked up and distributed over multiple disks *automatically*.

What Is Special about a Distributed File System (DFS)?

Which of these operations become *harder* when the file is spread across 100 machines?

Intuition

- Think of a book split across 10 lockers in 10 different buildings on campus.
  - *Reading page 500* means you need to know which locker has that section — you need a directory (the master/namenode).
  - *Appending a new chapter* is straightforward — just put it in the next available locker.
  - *Editing a sentence on page 200* is hard — you need to find the right locker, update it, AND make sure the backup copies in other buildings also get updated.

What Is Special about a Distributed File System (DFS)?

- All operations should be as simple as in a regular file system, i.e., applications should not have to explicitly deal with file chunks and chunk locations.
  - E.g., a client should be able to "read 10 bytes at offset 100 in file X," instead of having to determine how X is partitioned and then contact the machine that stores the relevant chunk of X.

What Is Special about a Distributed File System (DFS)?

- The DFS must prevent concurrency bugs.
  - How are file-system accesses synchronized, e.g., when two processes are trying to create a file with the same name, or write to the same file?
  - Who coordinates the machines storing the file chunks?
- **Example:** Two students submit MapReduce jobs at the same time. Both jobs try to create /output/result.txt. Without coordination, both might succeed and you'd have corrupted or interleaved data.
  - The master uses locks to make file creation atomic — exactly one succeeds, the other gets an error.

Motivation for Use of a DFS

- The *abstraction* of a single global file system simplifies programming warehouse-scale computers.

- A client can access a file that is possibly distributed over many machines as if it was a regular file stored locally.

Motivation for Use of a DFS

- Imagine Google Photos with 10 billion images.
  - Without DFS: the programmer has to manually decide "images A-M go on server 1, N-Z on server 2," handle what happens when server 1 dies, manually replicate data, etc.
  - With DFS: the programmer just writes read("/photos/user123/vacation.jpg") and the system handles everything — chunking, replication, failure recovery, etc.
- **Concrete number for scale:** A single server might have 2 TB of disk. A cluster of 1,000 servers gives you 2 PB — but only if you have a system to manage it as one unified namespace.

Motivation for Use of a DFS

- This frees the programmer from worrying about messy details such as:
  - Into how many chunks should a large file be split and on which machines should these chunks be stored?
  - Keeping track of which chunk(s) to modify when the user tries to change a sequence of bytes somewhere in the file.
  - Management of chunk replicas: Replicas are needed to provision against failures. These replicas must be kept in sync with each other. What should be done when a machine fails? Which other machine should receive a new copy of a file chunk that was stored on the failed machine?
  - Coordinating concurrent file access: What if two processes are trying to update the same file chunk? What if the processes access different chunks of the same file? How do we make sure that file creation is atomic, i.e., no two processes can create the same file concurrently?

Main GFS Design Choices

- A single master service is used to ensure consistency and achieve consensus for concurrent file-system operations.
  - In HDFS this is called a *namenode*.
- To avoid a bottleneck at the master, it only performs lightweight tasks.
  - These are tasks related to metadata management and location of data chunks.
- Since the master is a single point of failure, it should be able to recover quickly.

Main GFS Design Choices

- Data management and data transfer is performed directly by chunkservers. This distributes the heavy tasks.
  - In HDFS these are called *datanodes*.
- HDFS further reduces problem complexity by not supporting any file modifications, other than concatenation. Its **write-once-read-many** approach guarantees high read throughput.
  - To “modify” a file, one must create a new copy with the updated content.

Other GFS Design Goals

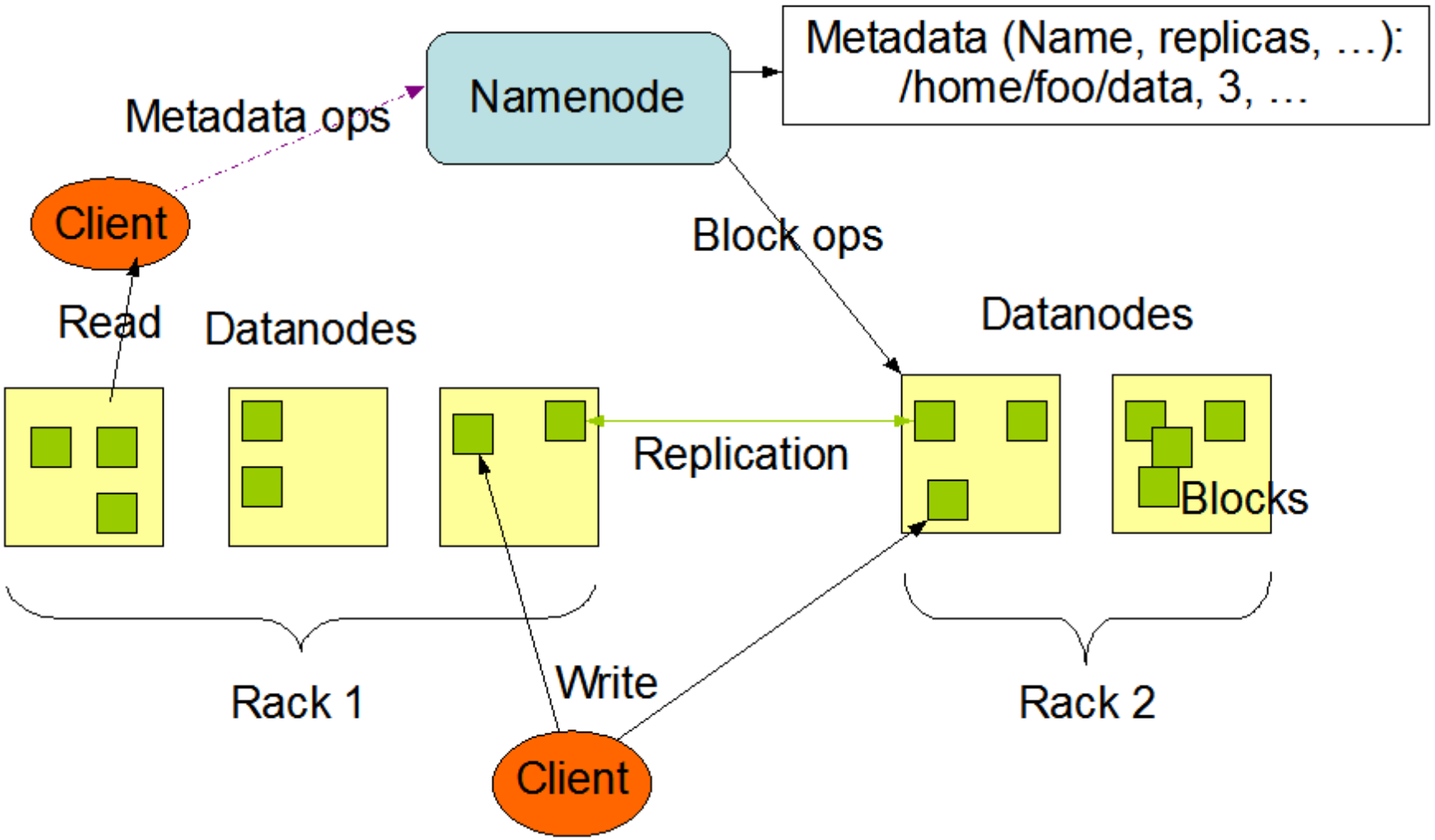
- The distributed file system should be able to handle a modest number of large files.

- Google aimed for a few million files, most of them 100 MB or larger in size. In particular, the Google File System (GFS) was designed to handle multi-gigabyte and larger files efficiently.

Other GFS Design Goals

- GFS assumes large sequential reads, not small random accesses.
  - **Sequential:** Read a large file -> a handful of master lookups (one per chunk), then stream continuously. You pay latency a few times, bandwidth does the heavy lifting.
  - **Random:** Read 1,000 scattered 1 KB records -> 1,000 master lookups + 1,000 network round trips. You pay latency **every single time**, and bandwidth barely matters because each read is tiny.
    - Recall: Transfer Time = Latency + Size/Bandwidth. Big reads -> bandwidth dominates (good). Tiny reads -> latency dominates (bad).
- **Standard file system interface** — create, delete, open, close, read, write. Same syntax as a local file system. No new API to learn.

HDFS Architecture



Source: https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html. Accessed in May 2026.

GFS and HDFS Architecture Details

- All machines are commodity Linux machines.

- Files are divided into fixed-size chunks—typically 256 MB or larger. These chunks are stored on the chunkservers' local disks as Linux files.
  - Chunks typically are replicated on multiple chunkservers, e.g., 3 times, for fault tolerance and faster read access.

- **Fun fact:** Yahoo's long-running HDFS clusters observed a node failure rate of 2–3 per 1,000 nodes per day. In a 10,000-node cluster, that's 20–30 machines failing EVERY DAY.

GFS and HDFS Architecture Details

- The master maintains all file system metadata such as namespace, access control information, the mapping from files to chunks, and chunk locations.
  - It uses locks to avoid concurrency-related conflicts, e.g., when creating a new file or directory.

A Note on Chunk Size

- The master keeps ALL metadata in memory for fast lookups.
  - Fewer chunks = less memory needed = more data the master can manage.
- How compact is it? Only ~64 bytes of metadata per chunk.
  - A master with 16 GB of RAM can track ~250 million chunks.
  - At 64 MB per chunk, that's **16 PB of data** — managed by ONE machine's memory.
- This means chunk size controls the master's capacity.

A Note on Chunk Size

- Advantages of a large chunk size:
  - Fewer interactions with the master. Recall that GFS is designed to support large sequential reads and writes. Reading a block of 128 MB would involve 2-3 chunks of size 64 MB, but 128-129 chunks of size 1 MB.
  - For the same file size, metadata is smaller. This reduces the amount of information managed by the master and allows metadata to be cached in the clients' memory.

A Note on Chunk Size

- Disadvantages of a large chunk size:
  - A small file fits entirely inside ONE chunk, which lives on ONE server. If many clients need it, they all queue up at that same server.
  - **Example:** A 10 MB config file that all 1,000 MapReduce tasks need to read.
    - With **2 MB chunks** -> 5 chunks on 5 different servers -> clients spread out, read in parallel.
    - With **64 MB chunks** -> 1 chunk on 1 server -> all 1,000 tasks hit the same machine.

A Note on Chunk Size

- **But wait — doesn't replication help?** Yes, partially. With 3 replicas, the load spreads across 3 servers instead of 1. But 1,000 tasks across 3 servers is still ~333 each. Increasing to 10 replicas helps more, but wastes storage.

- **Takeaway:** Large chunks are great for large files (fewer master lookups), but create bottlenecks for small popular files. It's a tradeoff.

A Note on Chunk-Replica Placement

- The goals of choosing machines to host the replicas of a chunk are scalability, reliability, and availability.

- Achieving this is difficult, because in a typical GFS setting there are hundreds of chunkservers spread across many racks, accessed from hundreds of clients running on machines in different racks.
  - Communication may cross multiple network switches.

A Note on Chunk-Replica Placement

- Spreading replicas across different racks is good for fault tolerance. Furthermore, read operations benefit from the aggregate bandwidth of multiple racks.

- On the other hand, it is bad for writes and file creation, as the data now flows through multiple racks.

A Note on Chunk-Replica Placement

- **Example: Assume our large dataset D (which fits in one chunk) is being replicated 3 times:**
  - Option A: All 3 on same rack -> fast writes, but one rack failure = all data gone
  - Option B (HDFS default): 2 on rack 1, 1 on rack 2 -> one cross-rack transfer, survives a full rack failure
  - Option C: Each on a different rack -> best fault tolerance, but 3 cross-rack writes

High-Level GFS Functionality

- The master controls all system-wide activities such as chunk lease management, garbage collection, and chunk migration.
- The master communicates with the chunkservers through HeartBeat messages to give instructions and collect their state.
- The client who tries to access a file gets metadata, incl. the locations of relevant file chunks, from the master, but accesses files directly through the chunkservers.

High-Level GFS Functionality

- There is no GFS-level file caching for several reasons.

- Caching helps when you read the *same* data repeatedly. But MapReduce/Spark workloads typically read a file once, process it, and move on.

- This avoids having to deal with cache-coherence issues in the distributed setting.

How Does the Client Know Which Chunk to Read?

- Every chunk holds exactly S megabytes, numbered starting at 0. So, finding the right chunk is just division.
- **Analogy:** A book where every chapter is exactly 100 pages.
  - Chapter 0 = pages [0, 99], Chapter 1 = pages [100, 199], Chapter 2 = pages [200, 299]
  - "Read page 150" -> 150 / 100 = 1.5 -> **Chapter 1**
  - "Read pages 150 to 250" -> starts in Chapter 1, ends in Chapter 2 -> **need both**

How Does the Client Know Which Chunk to Read?

- Same idea with chunks: replace "pages" with megabytes, "chapters" with chunks. **Example:** Chunk size = 64 MB. Client wants 10 MB starting at offset 150 MB.
  - Start chunk: ⌊150 / 64⌋ = **chunk 2**
  - End chunk: ⌊160 / 64⌋ = **chunk 2**
  - Same chunk means **1 master lookup**

- **Takeaway:** The client figures out which chunk(s) it needs with simple division, asks the master only for those chunk locations, then talks directly to the chunkservers.

Read Access Illustration

Application
(**GFS client**)

GFS master

File namespace:

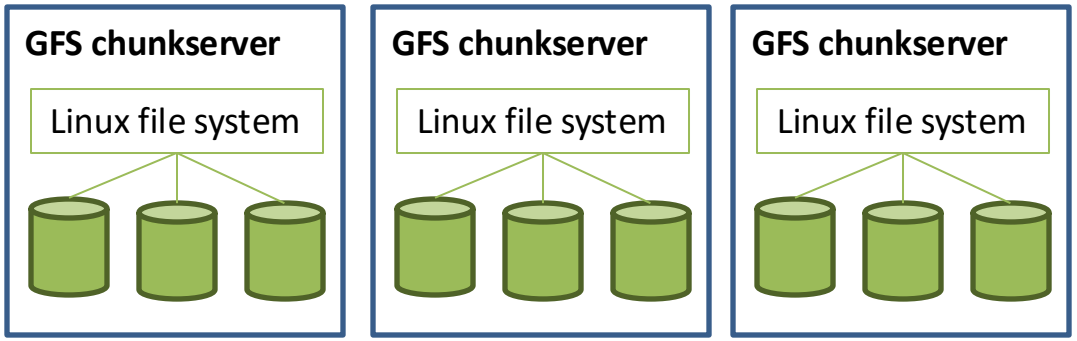
- Main directory
--- Subdirectory1
------ file1
------ file2
--- Subdirectory2
------file1

Chunk locations:

/Subdirectory1/file1
[list of chunk handles and their locations]

/Subdirectory1/file2
…

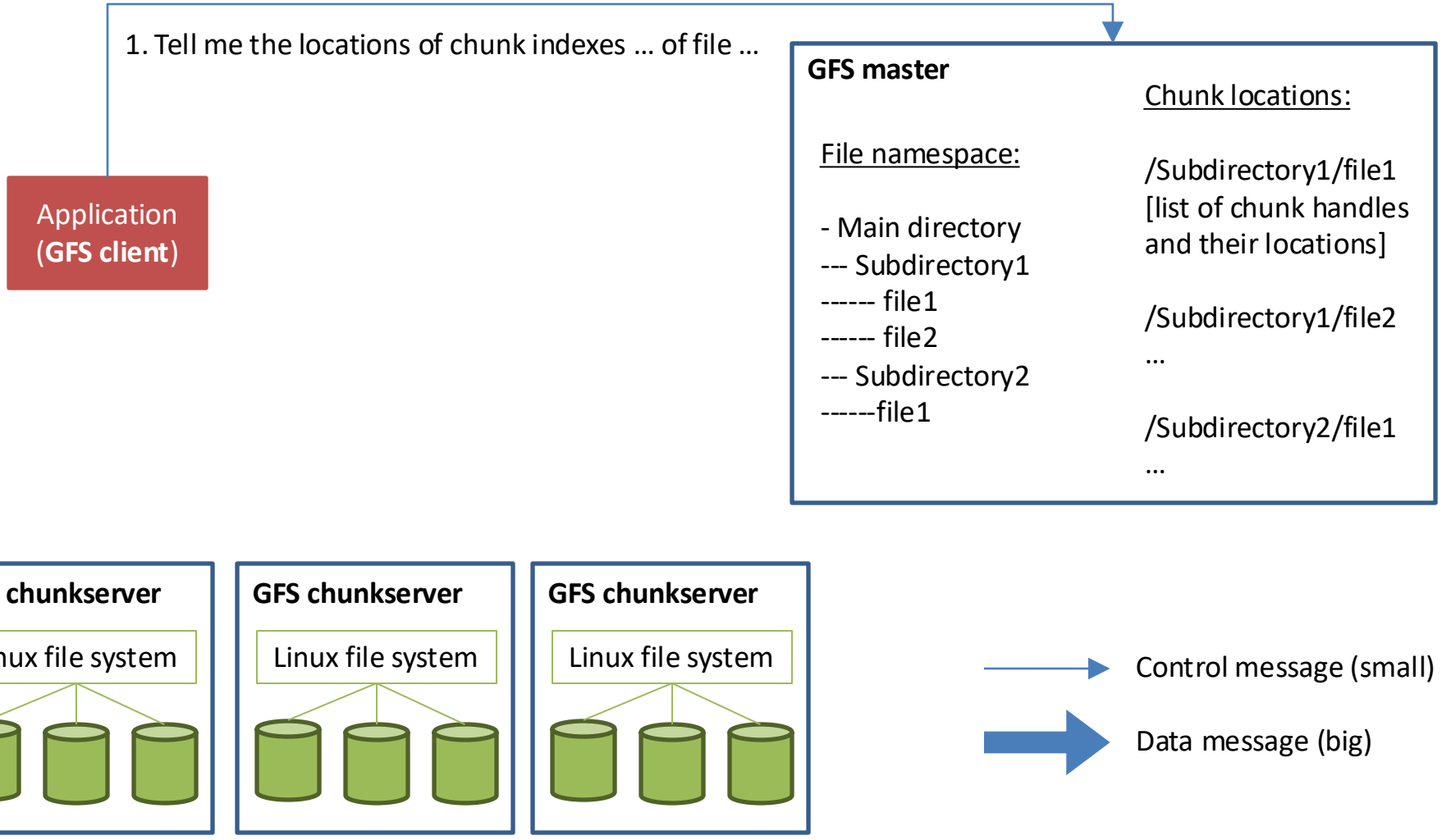
/Subdirectory2/file1
…



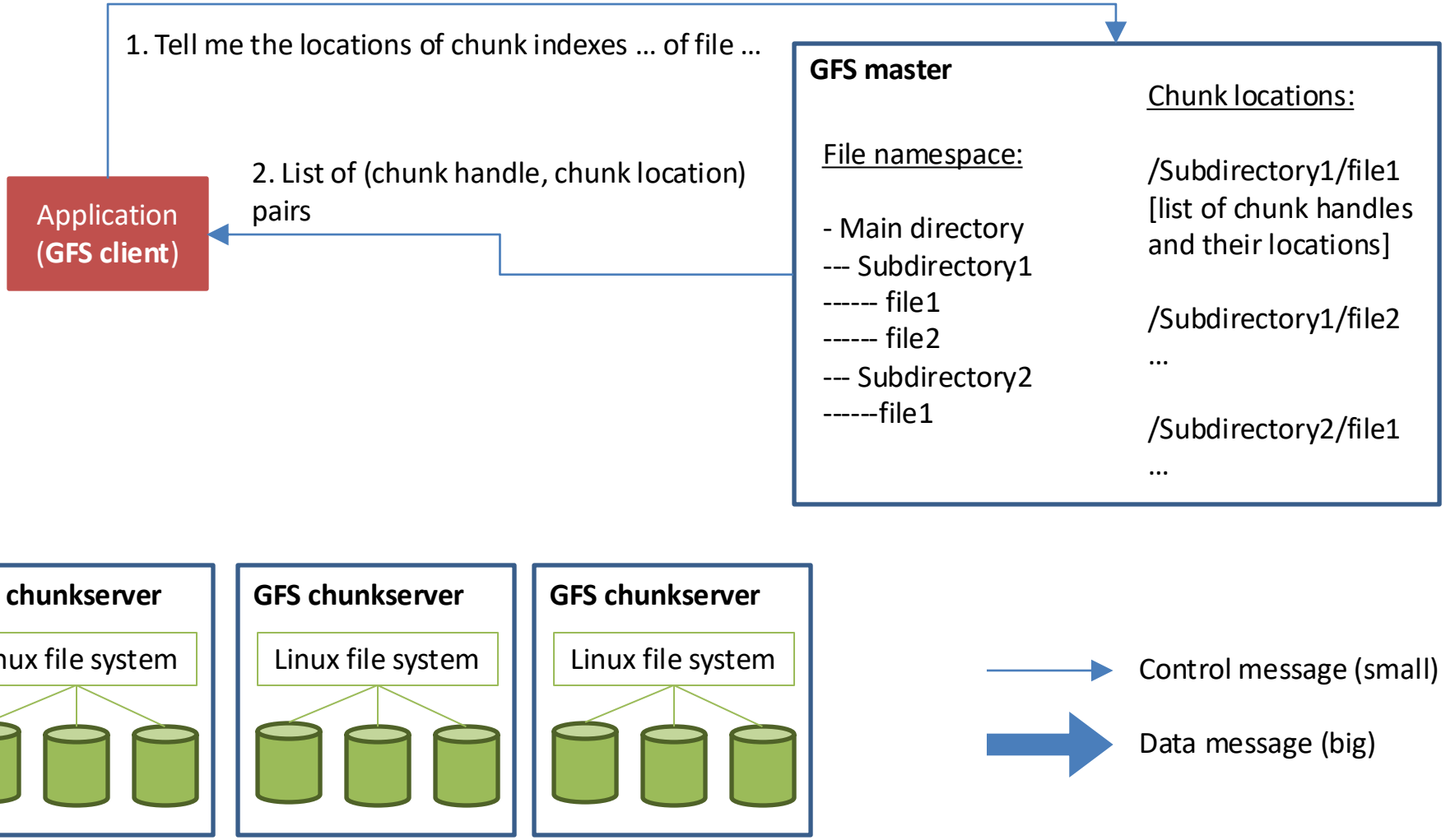
Control message (small)

Data message (big)

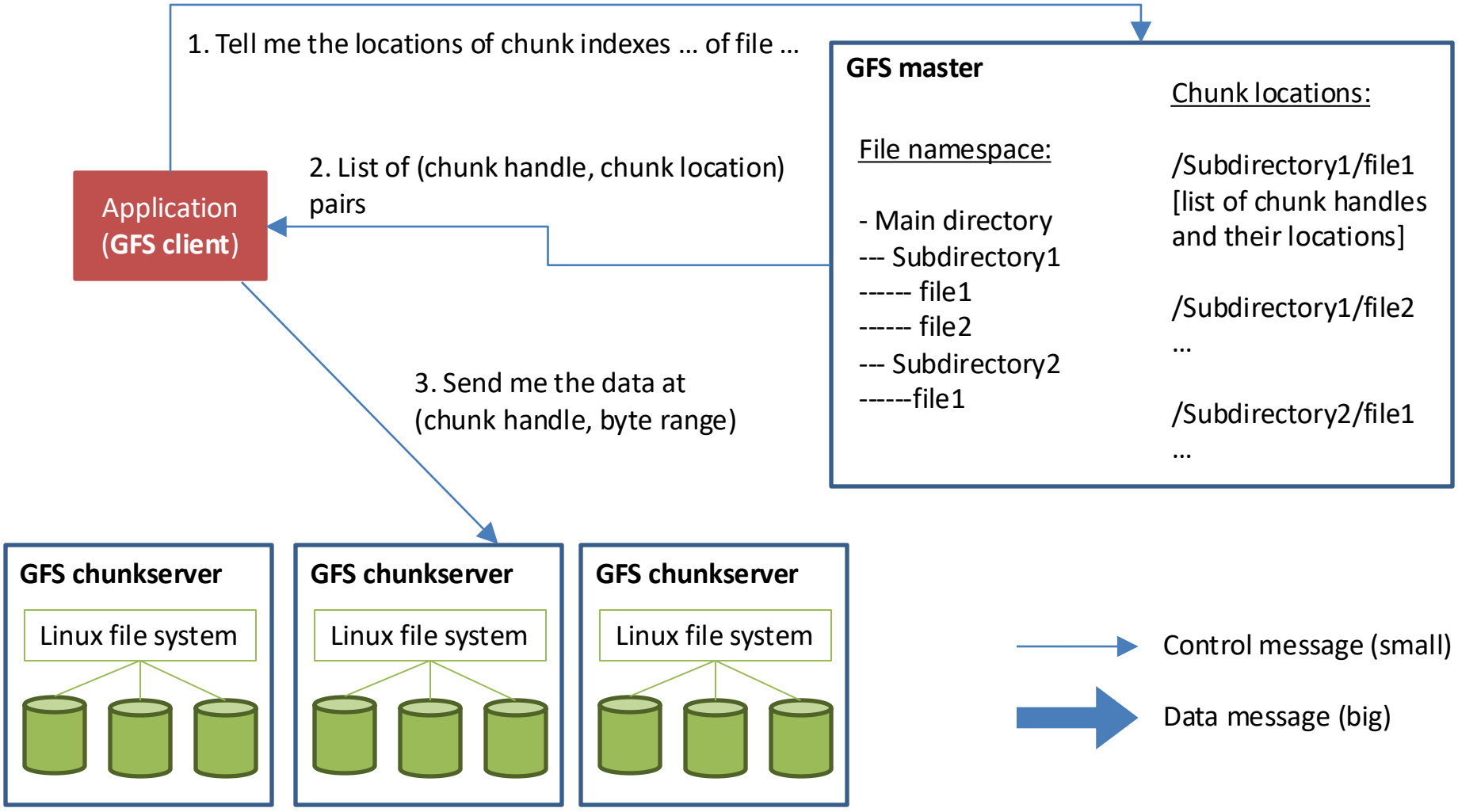
Read Access Illustration



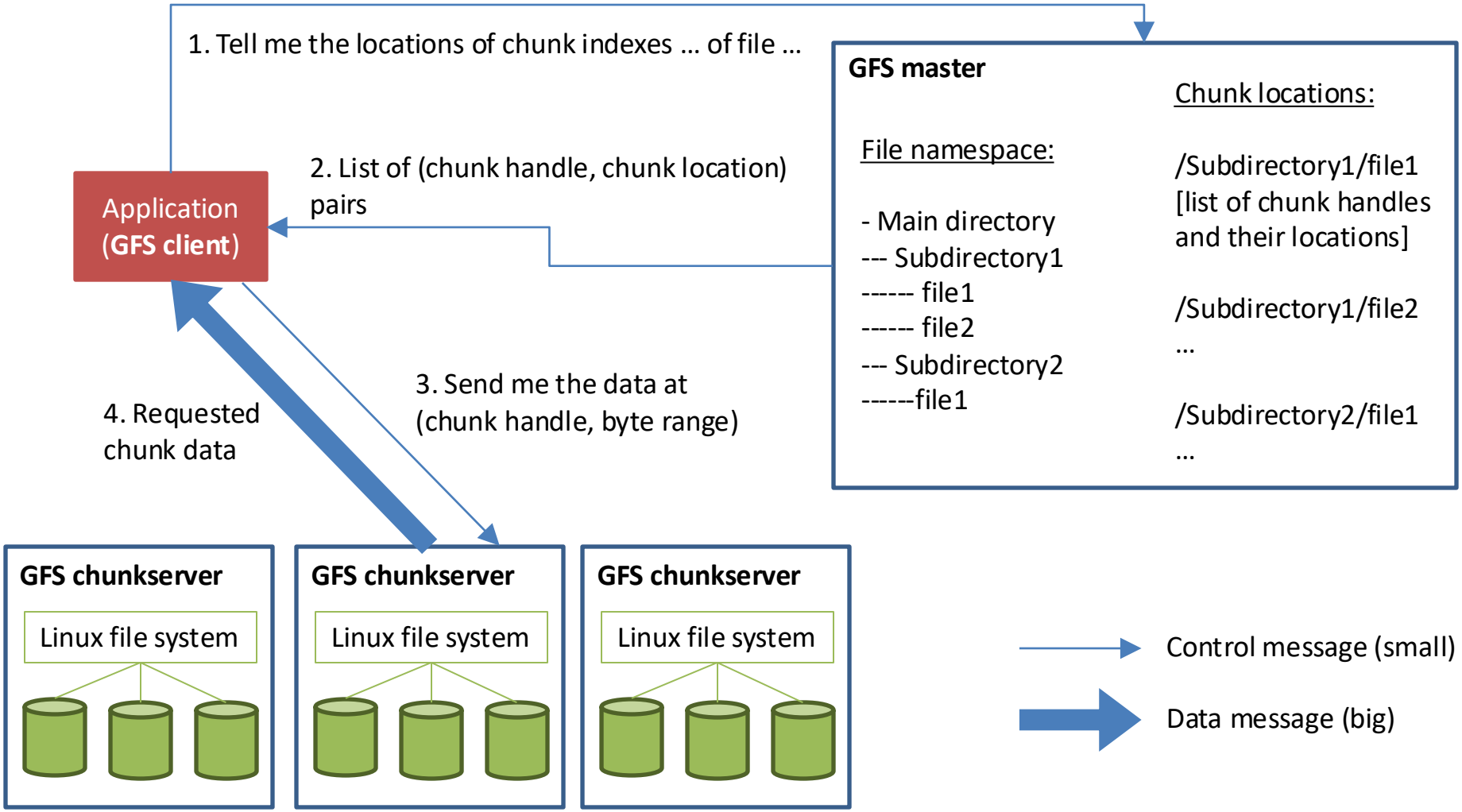
Read Access Illustration



Read Access Illustration



Read Access Illustration



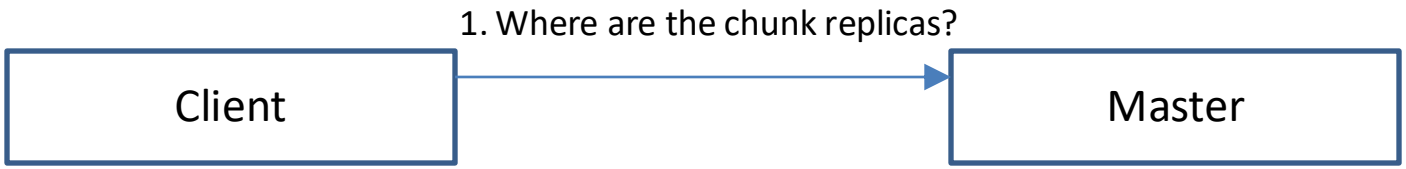
Updating a Chunk

Writing to a file is more challenging than reading, because all chunk replicas must be in sync. This means that they must apply the same updates in the same order.

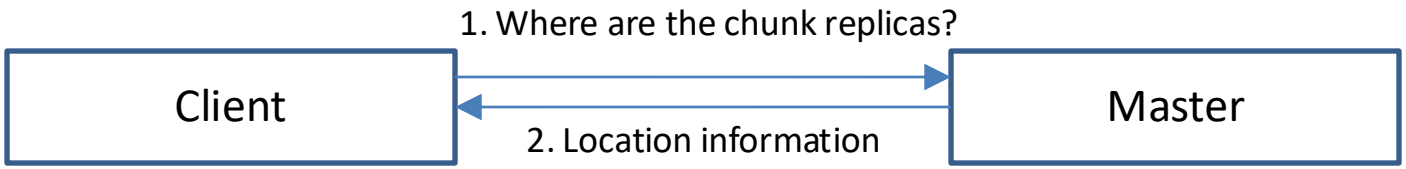
GFS achieves this by designating one replica of a chunk as the primary. It determines update order and notifies the secondaries to follow its lead.

Recall that updates are possible in GFS, not HDFS!!

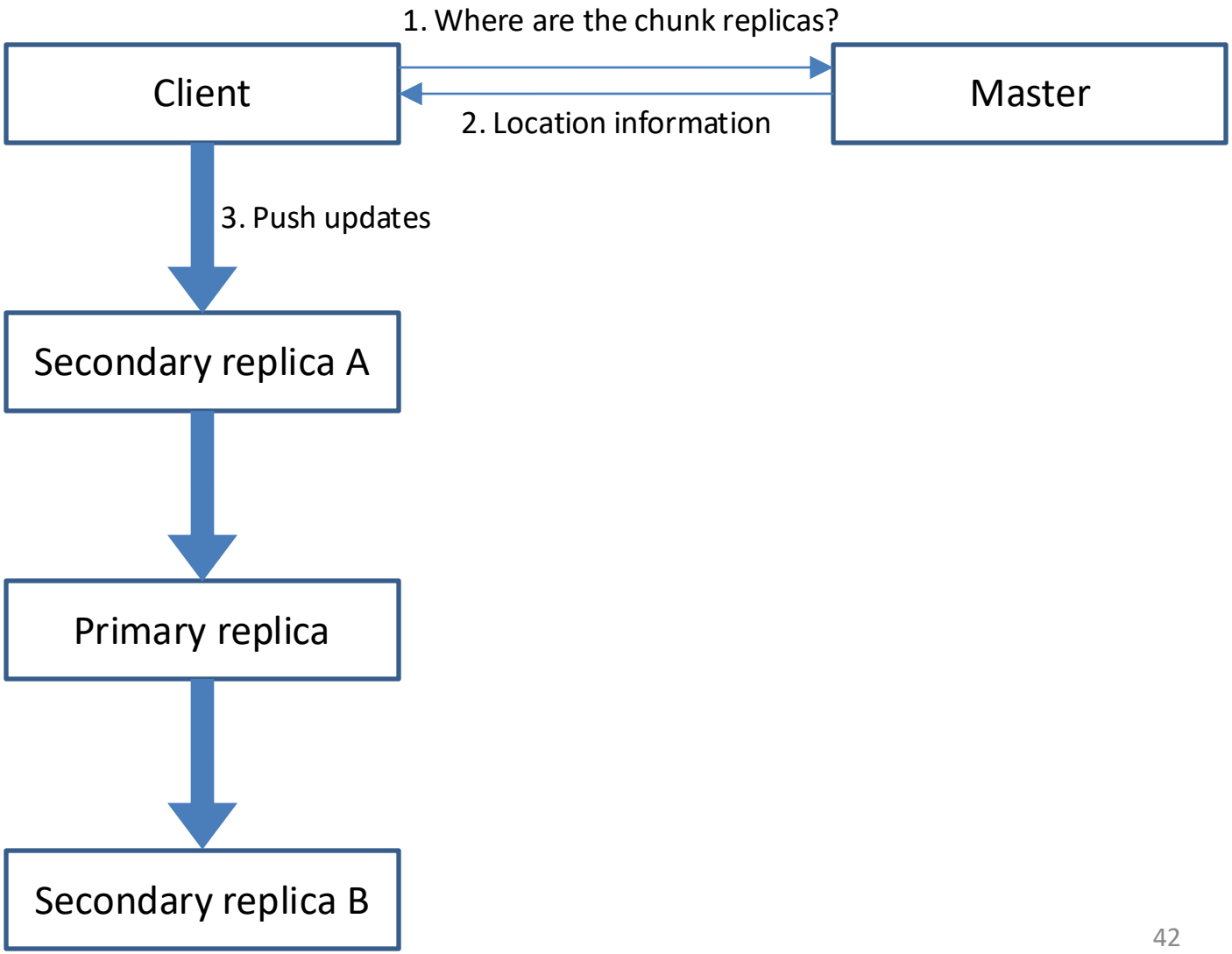
Update Process Illustration



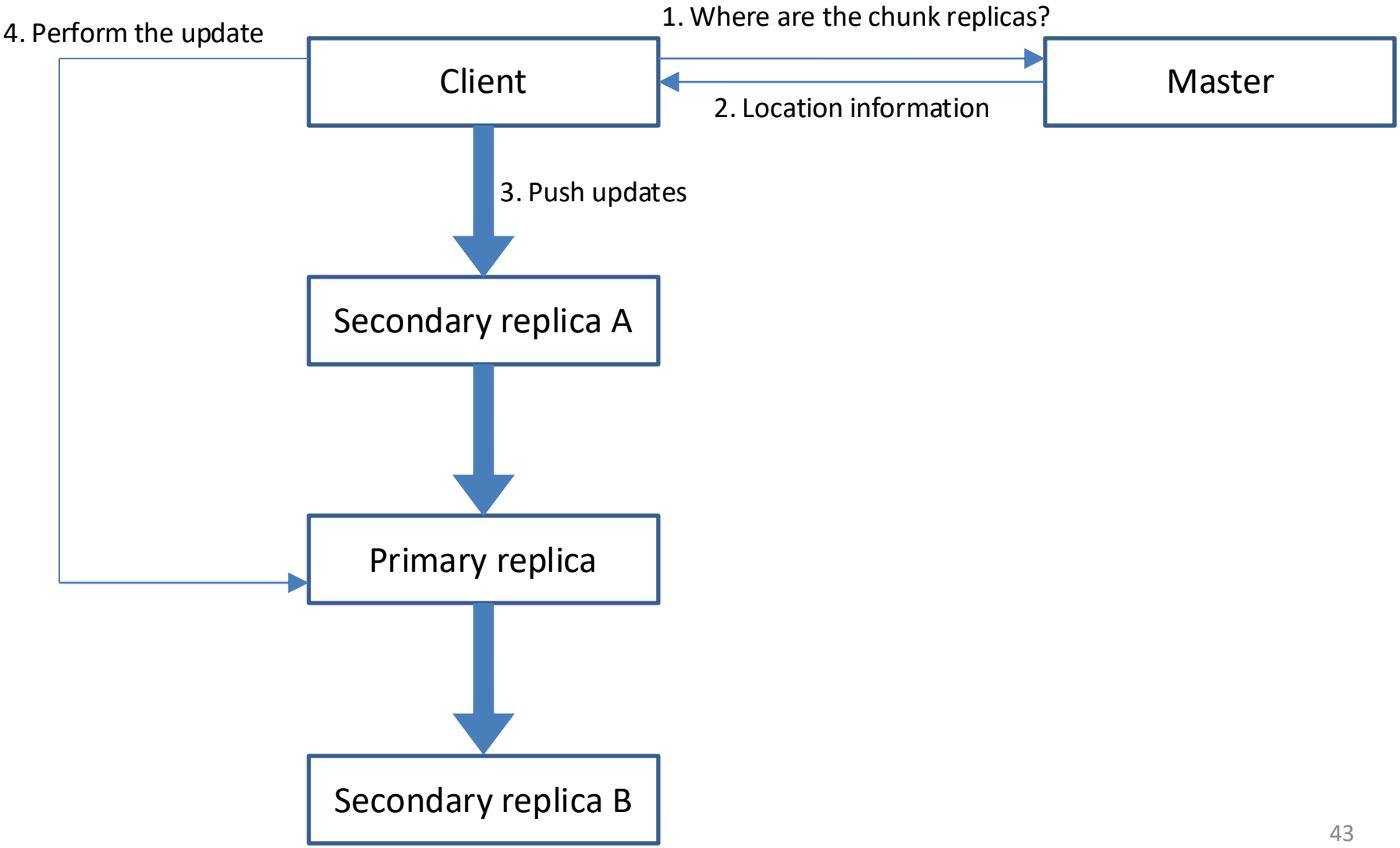
Update Process Illustration



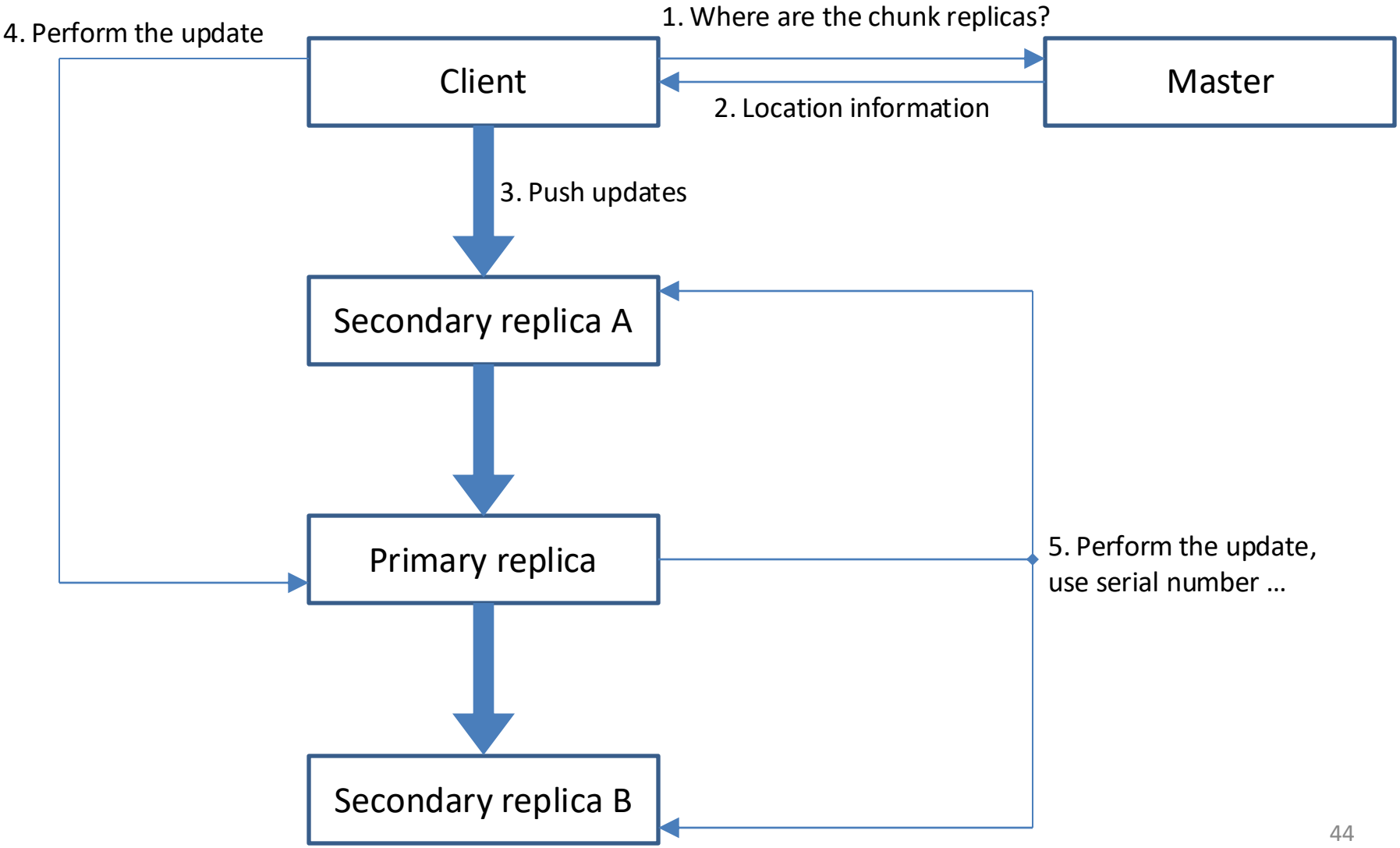
Update Process Illustration



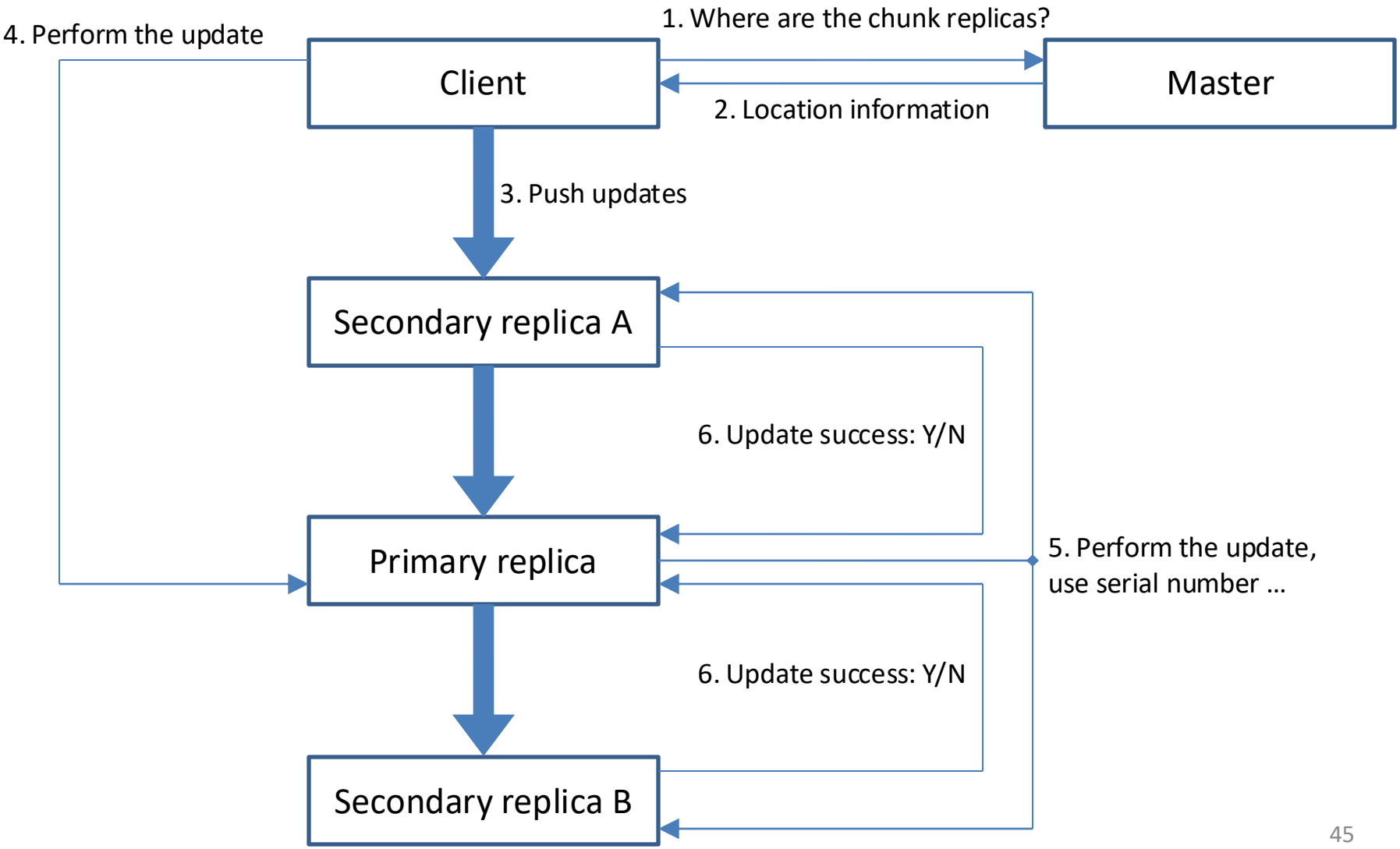
Update Process Illustration



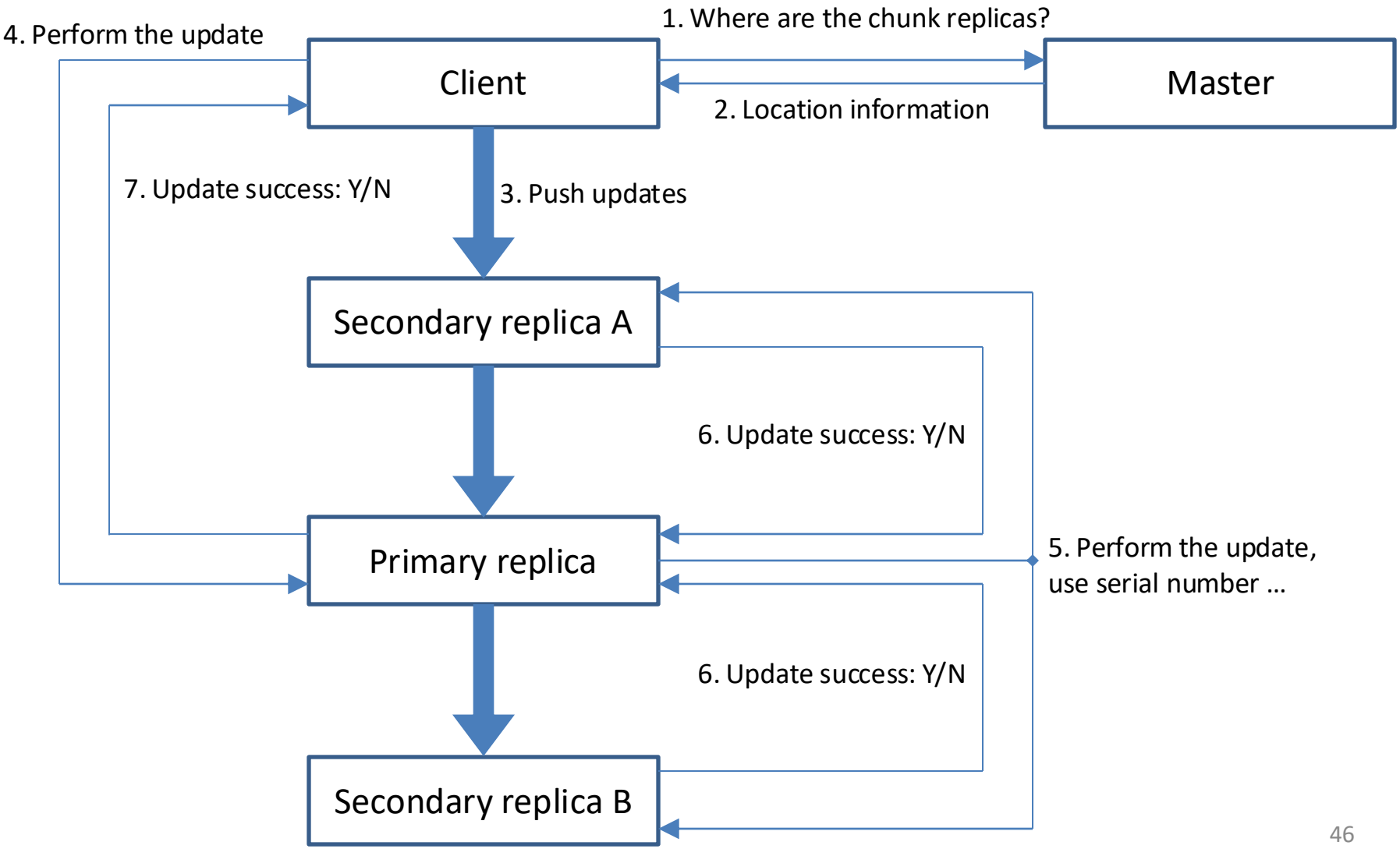
Update Process Illustration



Update Process Illustration



Update Process Illustration



Pushing Updates and New Files

- In GFS, data flow is decoupled from control flow for efficient network use. Instead of the client multi-casting the chunk update to each replica, the data is pipelined linearly along a chain of chunkservers. There are several reasons for this design decision…

Pushing Updates and New Files

- Pipelining makes use of the full outbound bandwidth for the fastest possible transfer, instead of dividing it in a non-linear topology.

- It avoids network bottlenecks by forwarding to the “next closest” destination machine.

Pushing Updates and New Files

- It minimizes latency:
  - Once a chunkserver receives data, it starts forwarding to the next one in the pipeline immediately.
  - Machines are typically connected by a switched network with full-duplex links.
  - This means that sending out data does not affect the bandwidth for receiving data (and vice versa).

Updates and Data Consistency

- It is challenging to ensure consistent update order across replicas when machines can fail or be slow to respond.
- GFS has mechanisms to deal with these problems, but for general updates (in contrast to append-only updates) it relies on a relaxed consistency model.
  - This means that under certain circumstances, e.g., when a client uses cached chunk location information, it might read from a stale data.

Updates and Data Consistency

- **Example**:
  - Client X writes "price = $50" to chunk 5
  - Primary replica applies it.
  - Primary tells secondaries A and B: "apply this write"
  - A succeeds, but B's network hiccups and the write fails.
  - Now: Primary and A say "price = $50", B still says "price = $30" (stale)

Updates and Data Consistency

Terrace: A Hierarchical Graph Container for Skewed Dynamic Graphs

Prashant Pandey
ppandey@berkeley.edu
Lawrence Berkeley National Lab and University of California Berkeley

Brian Wheatman
wheatman@cs.jhu.edu
Johns Hopkins University

Helen Xu
hjxu@mit.edu
Massachusetts Institute of Technology

Aydin Buluc
abuluc@lbl.gov
Lawrence Berkeley National Lab and University of California Berkeley

RapidStore: An Efficient Dynamic Graph Storage System for Concurrent Queries

Chiyu Hao[1], Jixian Su[1], Shixuan Sun[1], Hao Zhang[2], Sen Gao[1], Jianwen Zhao[2], Chenyi Zhang[2], Jieru Zhao[1], Chen Chen[1], Minyi Guo[1]
[1]Shanghai Jiao Tong University, China
[2]Huawei, China
{hcahoi11,sjx13623816973,sunshixuan,zhao-jieru,chen-chen,guo-my}@sjtu.edu.cn,{zhanghao687,zhaojianwen3,zhangchenyi2}@huawei.com,Gawssin@gmail.com

h storage systems are essential for real-time applica-
ocial networks and recommendation, where graph
usly evolves. However, they face significant chal-

and then operating on its neighbor set $N(u)$—these systems acquire locks on vertices rather than edges to coordinate access.

Despite these advancements, we observe significant performance issues due to their concurrency control methods. First, both read and

Dynamic Graph Databases with Out-of-order Updates

Angelos Christos Anadiotis
Oracle
Zurich, Switzerland
angelos.anadiotis@oracle.com

Muhammad Ghufran Khan
Inria & Institut Polytechnique de Paris
Palaiseau, France
muhammad.khan@inria.fr

Ioana Manolescu
Inria & Institut Polytechnique de Paris
Palaiseau, France
ioana.manolescu@inria.fr

ABSTRACT

Several real-time applications rely on dynamic graphs to model and store data arriving from multiple streams. Providing both high ingestion rate and efficient analytics with transactional guarantees is challenging, even more so when updates may be received *out-of-order* at the database. In this work, we propose HAL, a novel in-memory dynamic graph database design, addressing these chal-

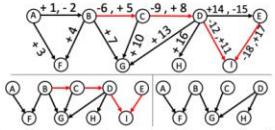


Figure 1: Sample dynamic graph with out-of-order updates.

GTX: A Write-Optimized Latch-free Graph Data System with Transactional Support

LIBIN ZHOU, Purdue University, USA
LU XING, Purdue University, USA
YEASIR RAYHAN, Purdue University, USA
WALID G. AREF, Purdue University, USA

This paper introduces GTX, a standalone main-memory write-optimized graph data system that specializes in

ered
$_T$).

ACID
action
same
otocol

Sortledton: a Universal Graph Data Structure

Per Fuchs
TU Munich
per.fuchs@cs.tum.edu

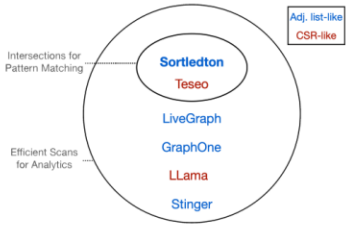
Domagoj Margan
Imperial College London
d.margan15@imperial.ac.uk

Jana Giceva
TU Munich // MDSI
jana.giceva@in.tum.de

ABSTRACT

Despite the wide adoption of graph processing across many different application domains, there is no underlying data structure that can serve a variety of graph workloads (analytics, traversals, and pattern matching) on dynamic graphs with single edge updates updates.

In this paper, we present Sortledton, a universal graph data structure that addresses the open problem by carefully optimizing for the most relevant data access patterns used by graph computation kernels. It can support millions of updates per second, while providing competitive performance (1.22x on average) for the most common graph work-



LiveGraph: A Transactional Graph Storage System with Purely Sequential Adjacency List Scans

Xiaowei Zhu[1], Guanyu Feng[1], Marco Serafini[2], Xiaosong Ma[3], Jiping Yu[1], Lei Xie[1], Ashraf Aboulnaga[3], and Wenguang Chen* [1,4]

[1]Tsinghua University, [2]University of Massachusetts Amherst, [3]Qatar Computing Research Institute, [4]Beijing National Research Center for Information Science and Technology,
[1]{zhuxiaowei,cwg}@tsinghua.edu.cn; {fgy18,yjp19,xie-l18}@mails.tsinghua.edu.cn;
[2]marco@cs.umass.edu;
[3]{xma,aaboulnaga}@hbku.edu.qa

Abstract

The specific characteristics of graph workloads make it hard to design a one-size-fits-all graph storage system. Systems that support transactional updates use data structures with poor data locality, which limits the efficiency of analytical workloads or even simple edge scans. Other systems run graph analytics workloads efficiently, but cannot properly support transactions.

This paper presents LiveGraph, a graph storage

and adjacency lists.[1] Facebook, for example, stores posts, friendship relationships, comments, and other critical data in a graph format [12, 20]. Write transactions incrementally update the graph, while read transactions are localized to edges, vertices, or the neighborhood of single vertices. These applications require a graph storage system to have very low latency and high throughput, be it a key-value store, a relational database management system, or a specialized graph database system. The system must also have classic transactional features: concurrency con-

Updates and Data Consistency

- Life is easier without updates. There would be no need to keep replicas in sync and no risk of reading stale replicas.
  - This is why HDFS does not support updates. Hadoop MapReduce and Spark do not need update functionality. They can simply write to new output files.
  - For instance, each Reduce task in a MapReduce job creates a temp file in the local file system. It then atomically copies and renames it to a file in the global file system.

Achieving High Availability

- **Chunkservers die?** No problem. Data is replicated on other servers.

- **Master dies temporarily?** It restarts and rebuilds its state from the operation log (a journal of every decision it made) + the latest checkpoint. Takes seconds because metadata is small.

- **Master dies permanently?** A new master starts on a different machine, loads the same replicated operation log.
  - Clients get redirected automatically to the new master.

Distributed File System Summary

- Distributed file systems support large-scale data processing workloads on commodity hardware.
- Component failures are treated as the norm, not the exception. GFS deals with failures through multiple mechanisms:
  - Constant monitoring of all participants.
  - Replication of crucial data.
  - A relaxed consistency model for updates.
  - Fast, automatic recovery.

Distributed File System Summary

- GFS is optimized for huge files, append writes, and large sequential reads.

- It achieves high aggregate throughput for concurrent readers and writers through separation of file system control (through master) from data transfer (directly between chunkservers and clients).

File Management in the Cloud

- In the cloud, you don't need GFS or HDFS. Instead, computation and storage can live on separate systems. Two reasons:

- Reason 1: For the user: you only pay for compute while your job is running. When it finishes, you shut down the compute machines and stop paying — but your data stays safely in the storage service.
  - With HDFS, your data lives on the same machines as your compute, so you can't shut them down without losing your data.

File Management in the Cloud

- Reason 2: For the cloud provider: storage and compute have different hardware needs (disks vs. CPUs), different scaling patterns, and different failure modes.
  - Managing them as separate specialized services is simpler than bundling everything together.

File Management in the Cloud

- A typical AWS setup: data lives permanently in S3 (cheap, durable storage). When you run a job, you spin up an EMR cluster (compute machines), which runs MapReduce or Spark. Two options for how the job accesses data:
  - Option 1: read directly from S3 during the job. Simpler, but every data access goes over the network to S3.
  - Option 2: copy data from S3 into HDFS on the EMR cluster at the start, use HDFS for all intermediate results (faster, local access), then write the final output back to S3. When the job is done, shut down the cluster.

- Side note: Spark and MapReduce can read from any file system, including your local one. The standalone single-machine mode (useful for debugging) uses local files by default.

More Information about Amazon

- Source: https://docs.aws.amazon.com/emr/latest/ManagementGuide/emr-plan-file-systems.html

File system (prefix)	Desription
HDFS (hdfs:// or no prefix)	An advantage of HDFS is data awareness between the Hadoop cluster nodes managing the clusters and the Hadoop cluster nodes managing the individual steps. HDFS is used by the master and core nodes. One advantage is that it's fast; a disadvantage is that it's ephemeral storage which is reclaimed when the cluster ends. It's best used for caching the results produced by intermediate job-flow steps.
EMRFS (s3://)	EMRFS is an implementation of the Hadoop file system used for reading and writing regular files from Amazon EMR directly to Amazon S3. EMRFS provides the convenience of storing persistent data in Amazon S3 for use with Hadoop while also providing features like Amazon S3 server-side encryption, read-after-write consistency, and list consistency.
local file system	When a Hadoop cluster is created, each node is created from an EC2 instance that comes with a preconfigured block of preattached disk storage called an *instance store*. Data on instance store volumes persists only during the life of its EC2 instance. Instance store volumes are ideal for storing temporary data that is continually changing, such as buffers, caches, scratch data, and other temporary content.
Amazon S3 block file system (s3bfs://)	The Amazon S3 block file system is a legacy file storage system. We strongly discourage the use of this system.

References

- Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google file system. In *Proc. of the nineteenth ACM Symposium on Operating Systems Principles* (SOSP '03), 29-43
  - https://scholar.google.com/scholar?cluster=9821092550821837 1&hl=en&as_sdt=0,22
- Amazon EMR documentation: Management guide
  - https://docs.aws.amazon.com/emr/latest/ManagementGuide/emr-what-is-emr.html