RAID

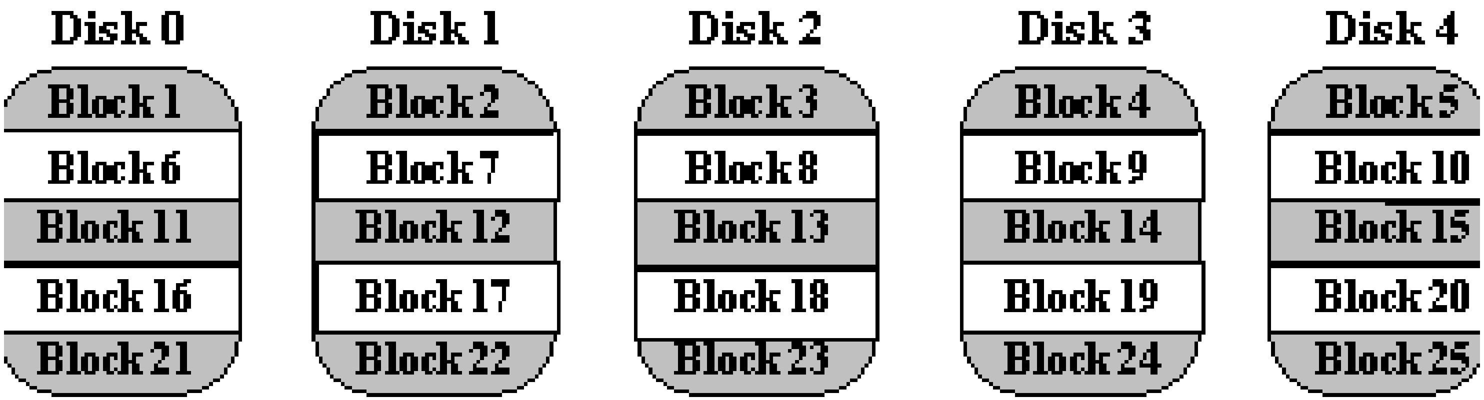
The performance of different RAID levels
read/write/reliability (fault-tolerant)/overhead

Tiffany Yu-Han Chen

(These slides modified from Hao-Hua Chu – National Taiwan University)

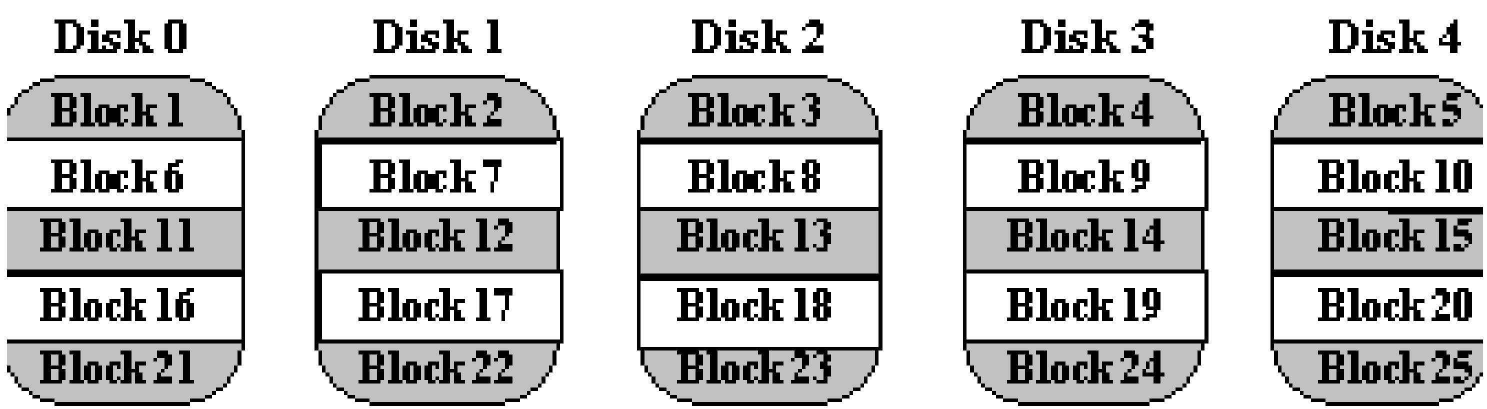
RAID 0 - Striping

- Strip data across all drives (minimum 2 drives)
  - Sequential blocks of data are written across multiple disks in stripes.



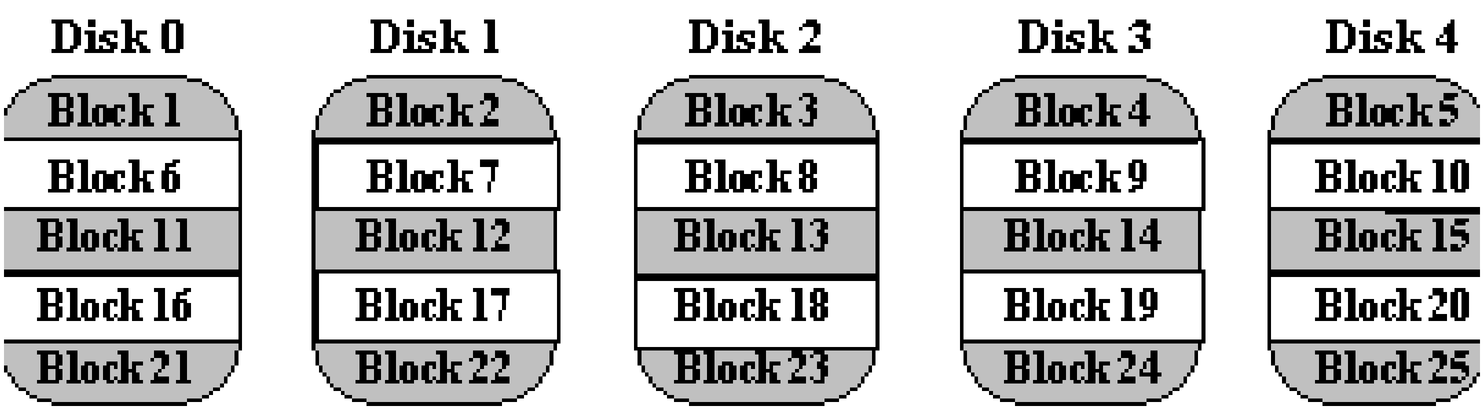
RAID 0 - Striping

- Strip data across all drives (minimum 2 drives)
  - Sequential blocks of data are written across multiple disks in stripes.
- Read/write?
- Reliability? Failure recovery?
- Space redundancy overhead?



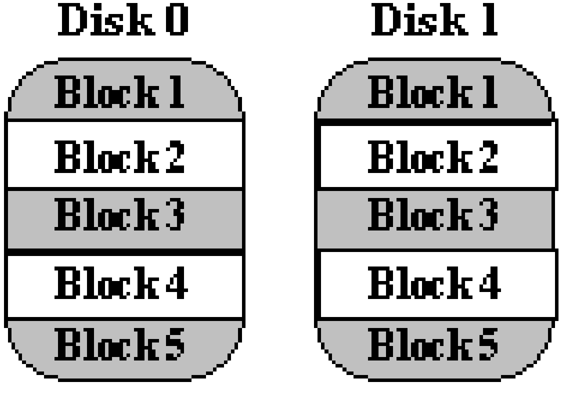
RAID 0 - Striping

- Strip data across all drives (minimum 2 drives)
  - Sequential blocks of data are written across multiple disks in stripes.
- Read/write? **Concurrent read/write**
- Reliability? Failure recovery? **No/No**
- Space redundancy overhead? **0%**



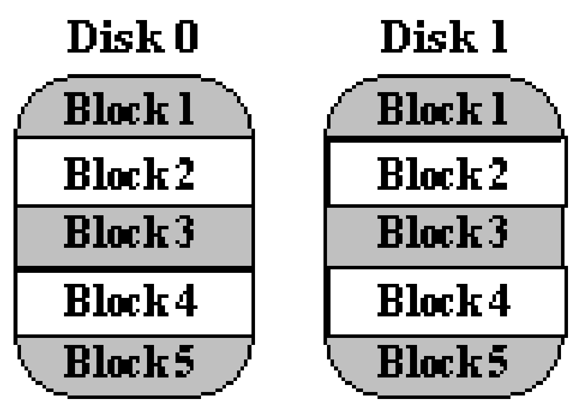
RAID 1 - Mirrored

- Redundancy by duplicating data on multiple disks
  - Copy each block to both disks



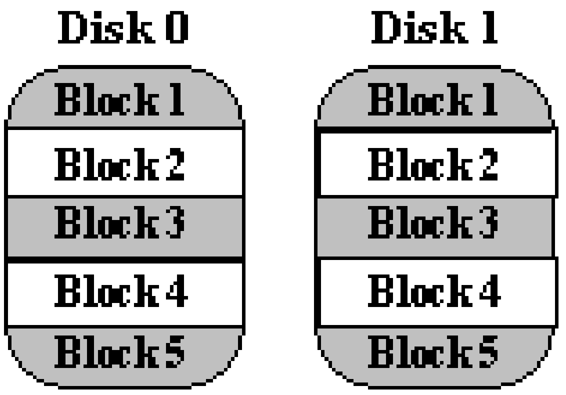
RAID 1 - Mirrored

- Redundancy by duplicating data on different disks
  - Copy each block to both disks
- Read?
- Write?
- Reliability? Failure recovery?
- Space redundancy overhead?



RAID 1 - Mirrored

- Redundancy by duplicating data on different disks
  - Copy each block to both disks
- Read? **Concurrent read**
- Write? **Need to write to both disks**
- Reliability? Failure recovery? **Yes**
- Space redundancy overhead? **50%**

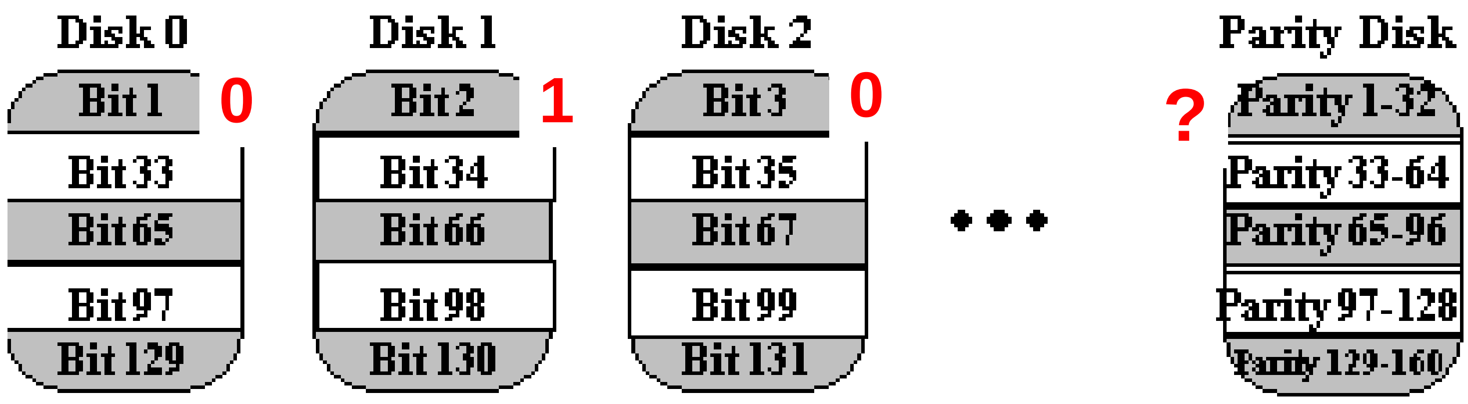


RAID 3 – RAID 5
Parity Disk for *Error Correction*

- Disk controllers can detect incorrect disk blocks while reading it
  - Check disk does not need to do error detection, error correction would be sufficient

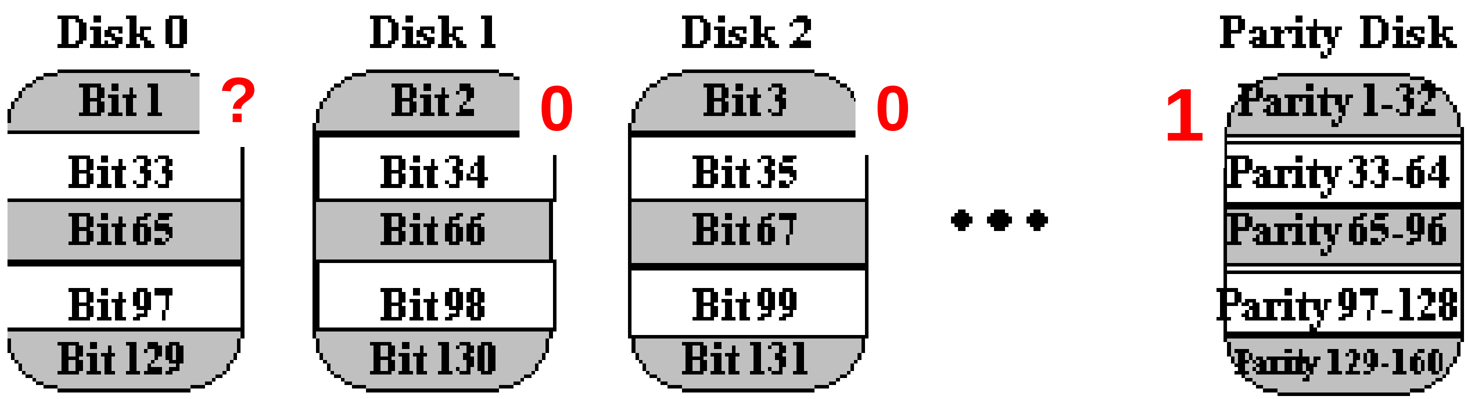
RAID 3 – RAID 5
Parity Disk for *Error Correction*

- One parity disk for error correction
  - Parity bit value = XOR across all data bit values



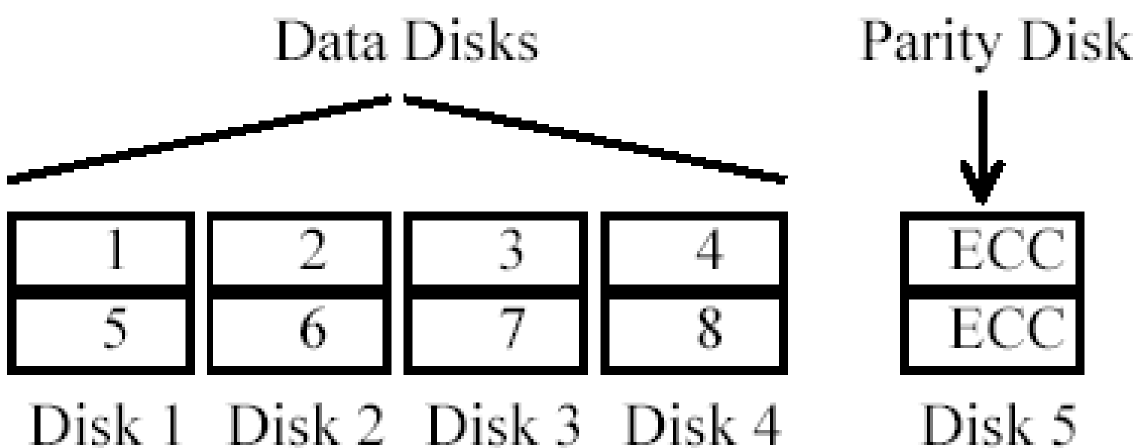
RAID 3 – RAID 5
Parity Disk for *Error Correction*

- One parity disk for error correction
  - Parity bit value = XOR across all data bit values
- If one disk fails, recover the lost data
  - XOR across all good data bit values and parity bit value



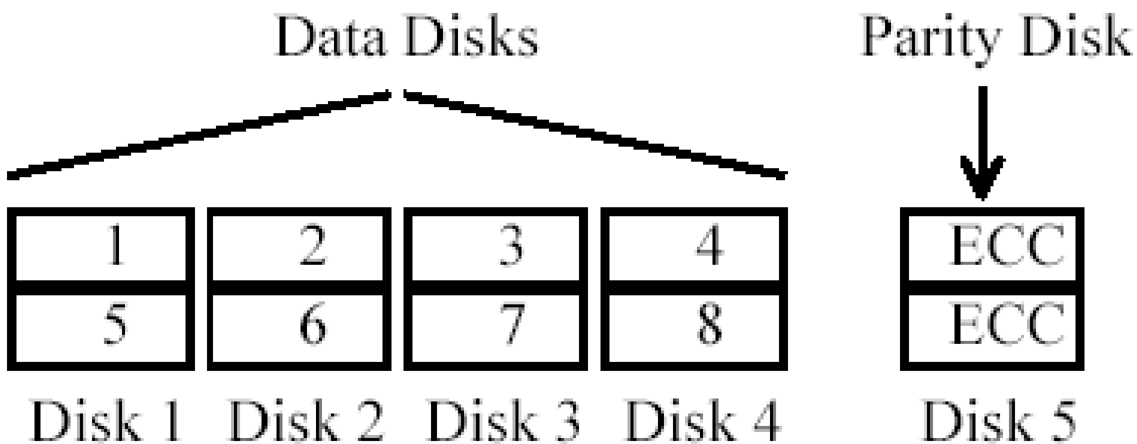
RAID 4 - Block-Interleaved Parity

- Coarse-grained striping at the block level
- One parity disk



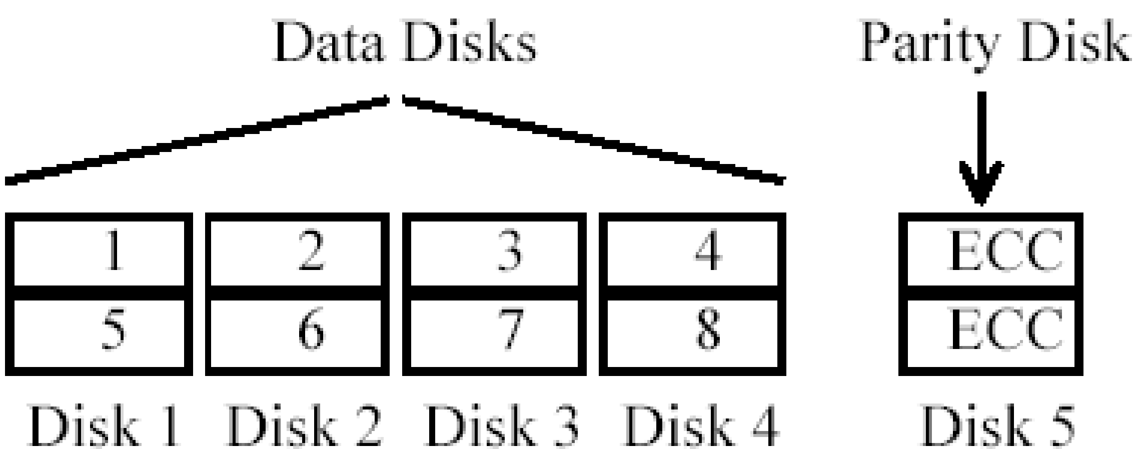
RAID 4 - Block-Interleaved Parity

- Coarse-grained striping at the block level
- One parity disk
- If one disk fails, # of disk access?



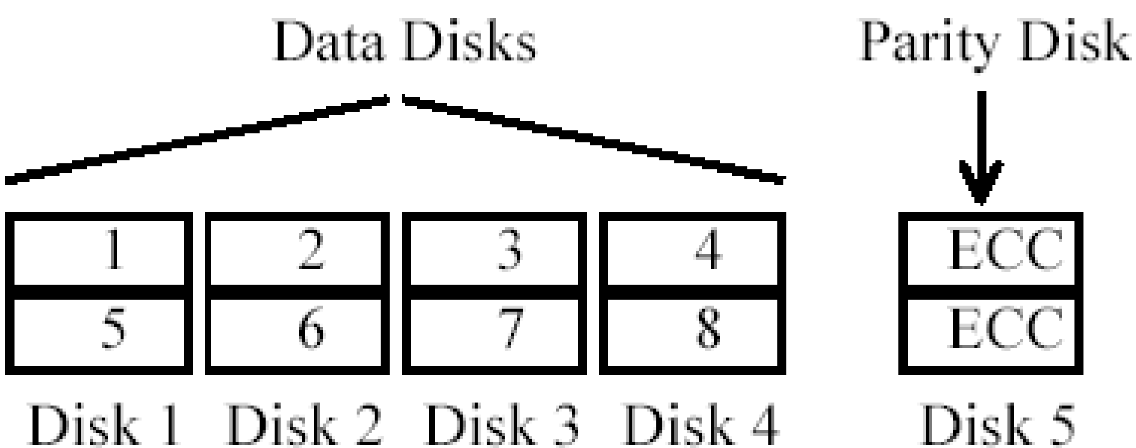
RAID 4 - Block-Interleaved Parity

- Coarse-grained striping at the block level
- One parity disk
- If one disk fails, # of disk access:
  - Read from all good disks (including parity disk) to reconstruct the lost block.



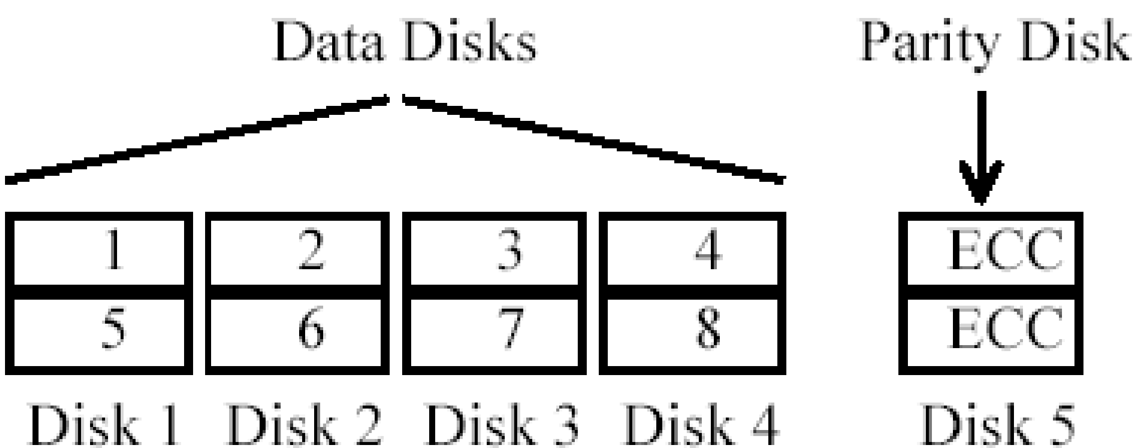
RAID 4 - Block-Interleaved Parity

- Read?
  - 1-block read, disk access?
- Reliability? Failure recovery?
- Space redundancy overhead?



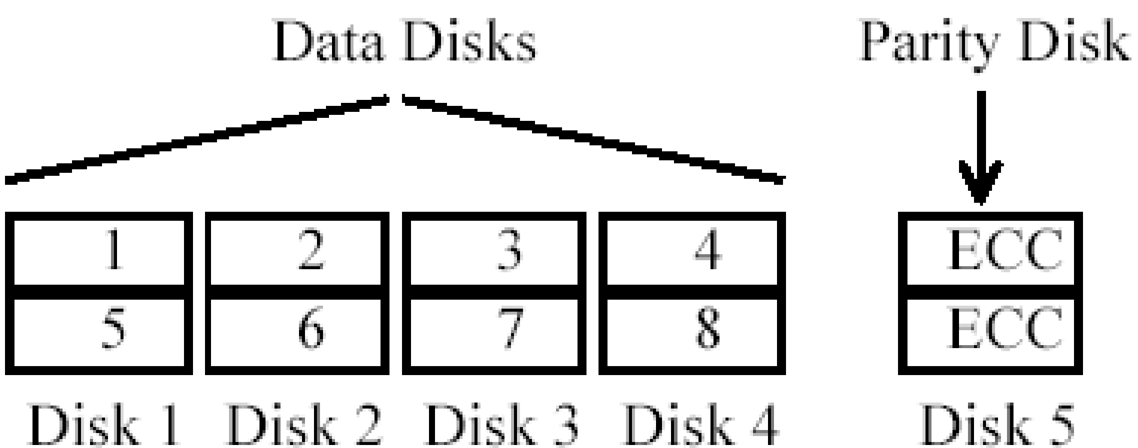
RAID 4 - Block-Interleaved Parity

- Read? **Concurrent read**
- Reliability? Failure recovery? **Can tolerate 1 disk failure**
- Space redundancy overhead? **1 parity disk**



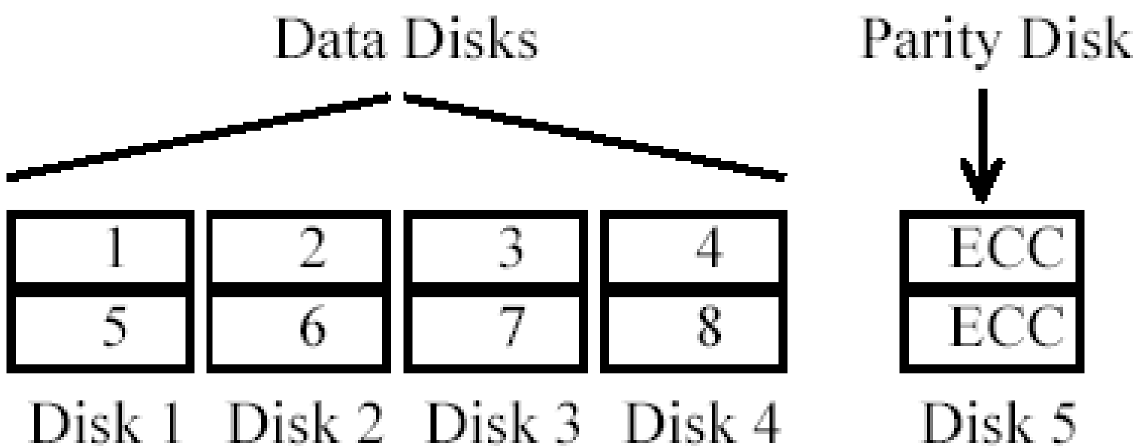
RAID 4 - Block-Interleaved Parity

- Write
  - Write also needs to update the parity disk.
  - ***no need to read other disks!***



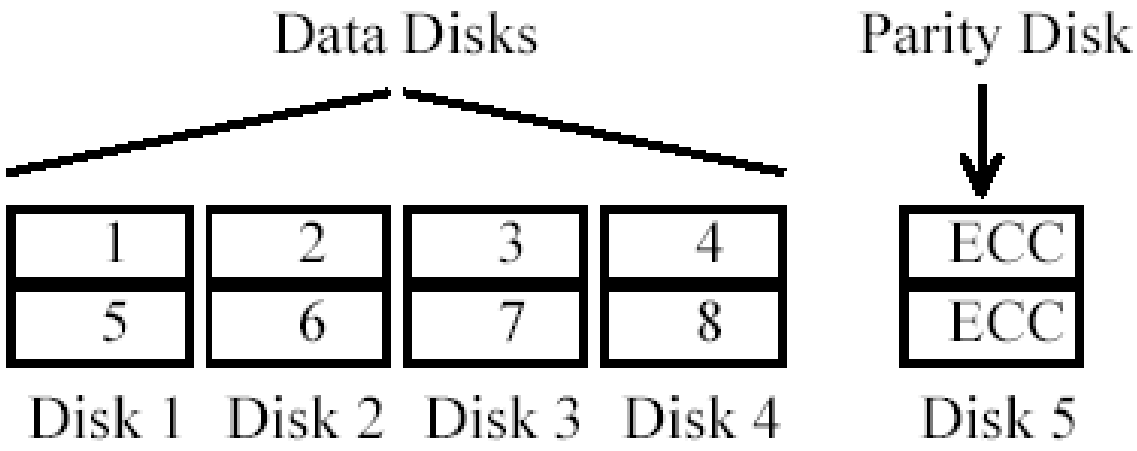
RAID 4 - Block-Interleaved Parity

- Write
  - Write also needs to update the parity block.
  - ***no need to read other disks!***
    - Can compute new parity based on old data, new data, and old parity
    - New parity = (old data XOR new data) XOR old parity
  - 1-block write, # of disk access?



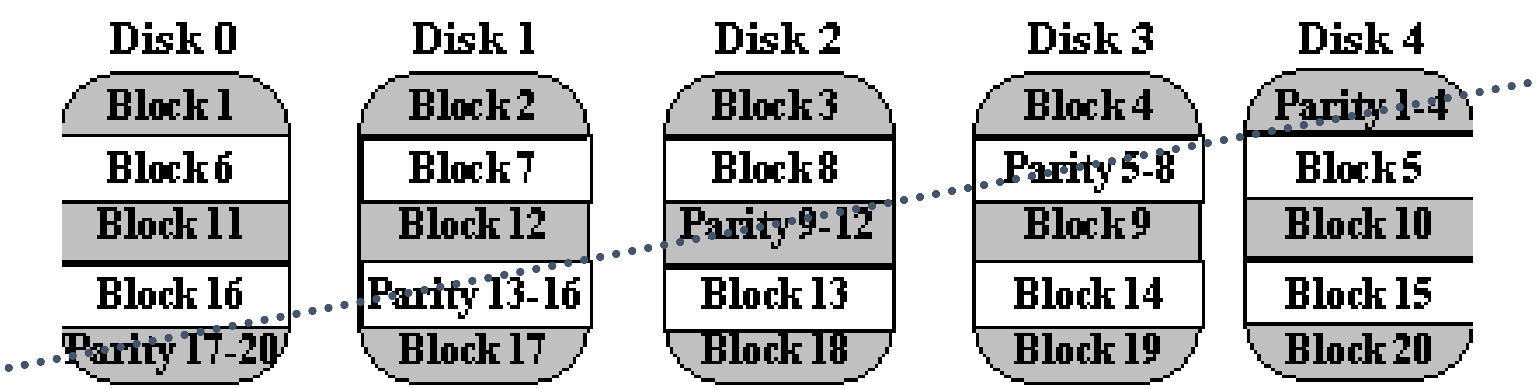
RAID 4 - Block-Interleaved Parity

- Write
  - Write also needs to update the parity block.
  - ***no need to read other disks!***
    - Can compute new parity based on old data, new data, and old parity
    - New parity = (old data XOR new data) XOR old parity
  - 1-block write, # of disk access?
  - Result in bottleneck on the parity disk! (can do only one write at a time)



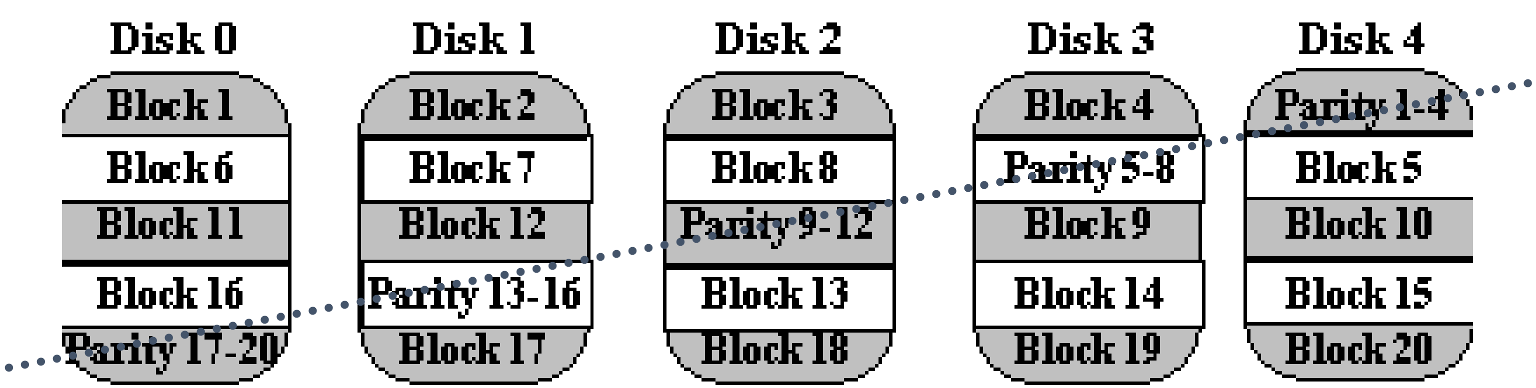
RAID 5- Block-Interleaved Distributed-Parity

- Remove the parity disk bottleneck in RAID 4 by distributing the parity uniformly over all of the disks.
- Comparing to RAID 4
  - Read?
  - Write?
  - Reliability, space redundancy?



RAID 5- Block-Interleaved Distributed-Parity

- Remove the parity disk bottleneck in RAID 4 by distributing the parity uniformly over all of the disks.
- Comparing to RAID 4
  - Read? **Similar**
  - Write? **Better**
  - Reliability, space redundancy? **Similar**



Questions?

Google File System (GFS)

\# Observations -> design decisions

(These slides modified from Alex Moshchuk, University of Washington)

Assumptions

- GFS built with commodity hardware
  - **High** component failure rates
- GFS stores a modest number of **huge files**
  - A few million files
  - Each is 100MB or larger

Workloads

- Read: Large streaming reads (> 1MB); small random reads (a few KBs)

Workloads

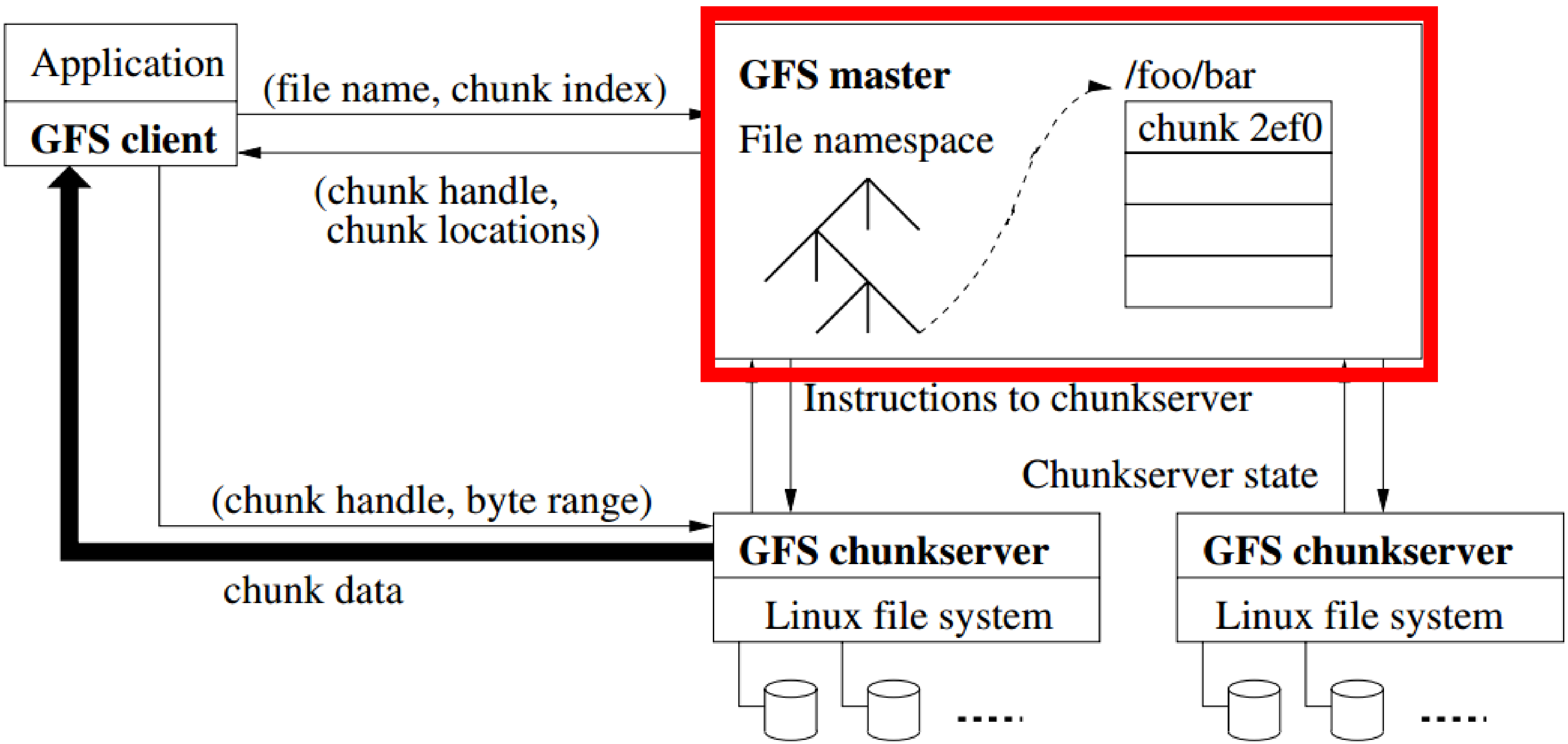
- Read: Large streaming reads (> 1MB); small random reads (a few KBs)
- Write: Files are write-once, mostly append to
  - File is **concurrently** written by multiple clients (map-reduce)

GFS Design Decisions

- Files stored as/divided into chunks
  - Chunk size: 64MB
- Reliability through replication
  - Each chunk replicated across 3+ chunkservers
- *Single* master to coordinate access, keep metadata
  - Simple
- Add record append operations
  - Support concurrent appends

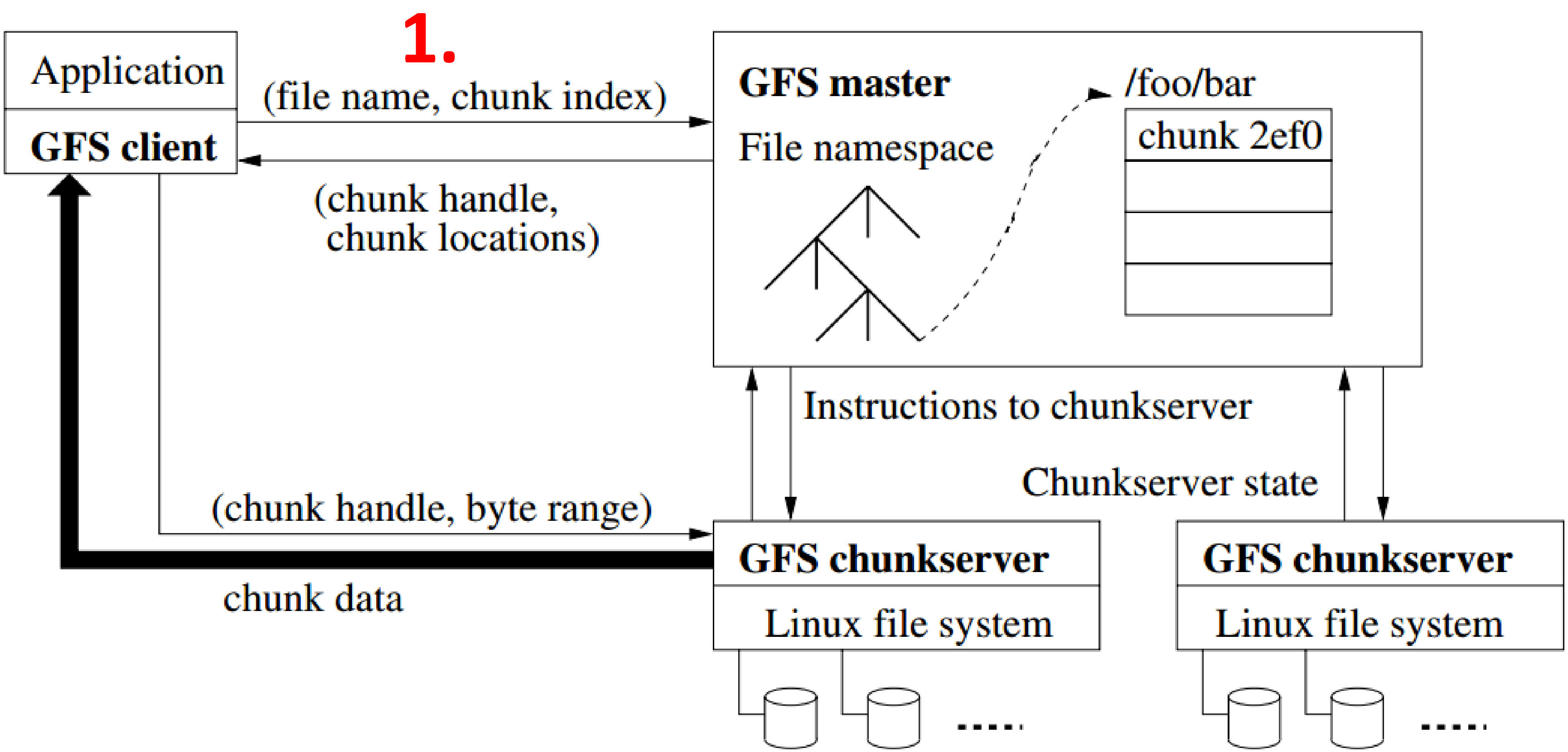
Architecture

Single master
Multiple chunkservers



Architecture

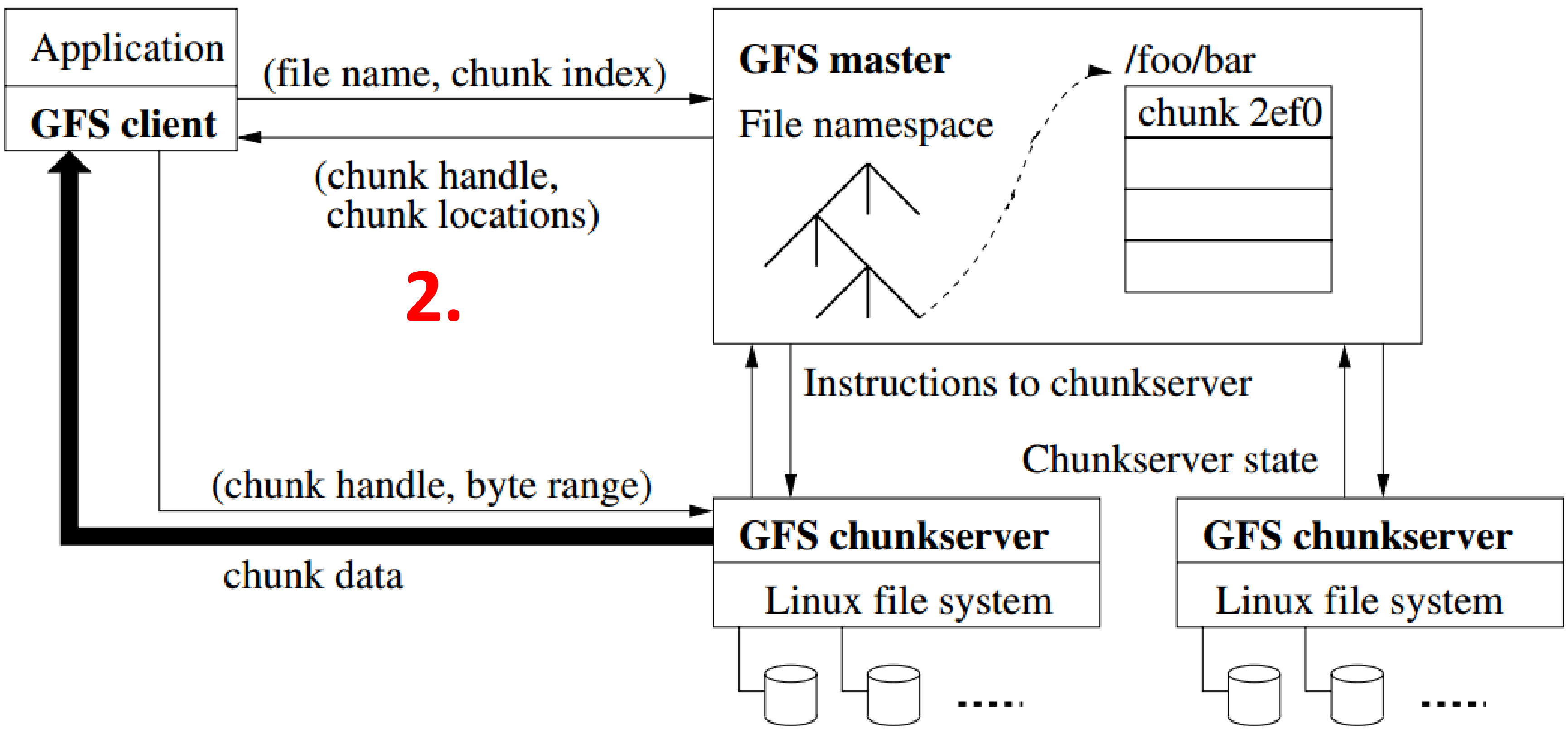
Single master
Multiple chunkservers



Architecture

Single master

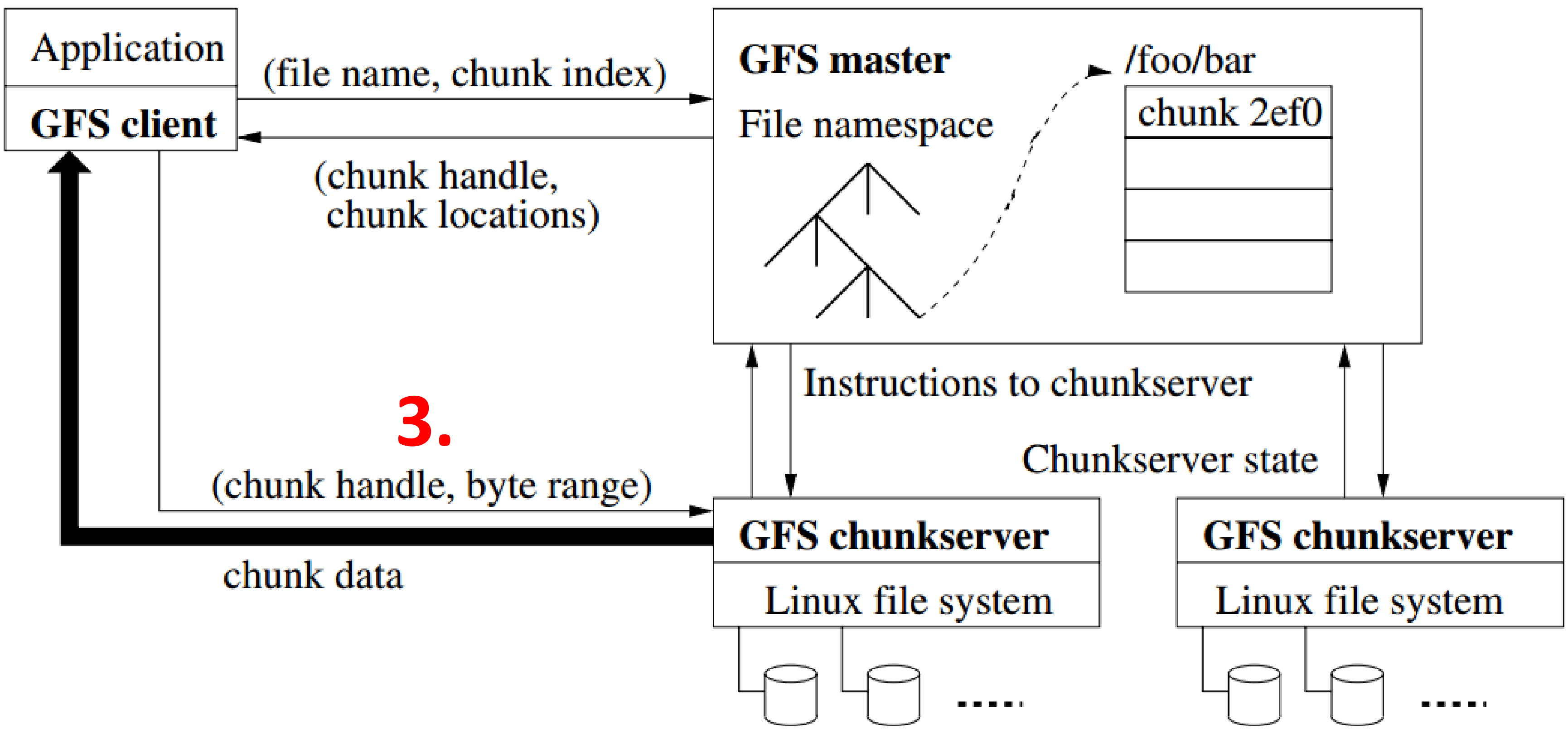
Multiple chunkservers



Architecture

Single master

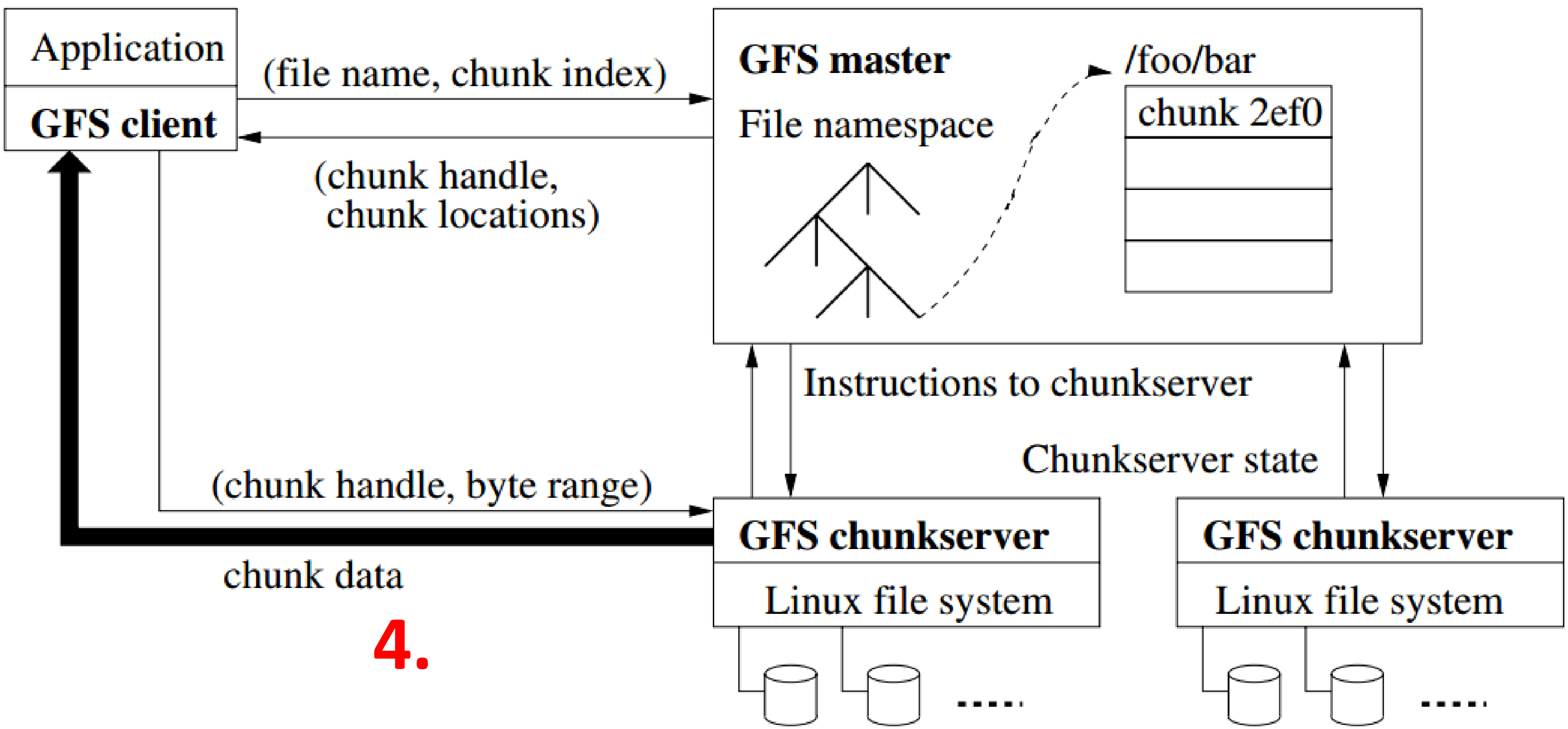
Multiple chunkservers



Architecture

Single master

Multiple chunkservers



Single Master - Simple

- Problem:
  - Single point of failure
  - Scalability bottleneck

Single Master - Simple

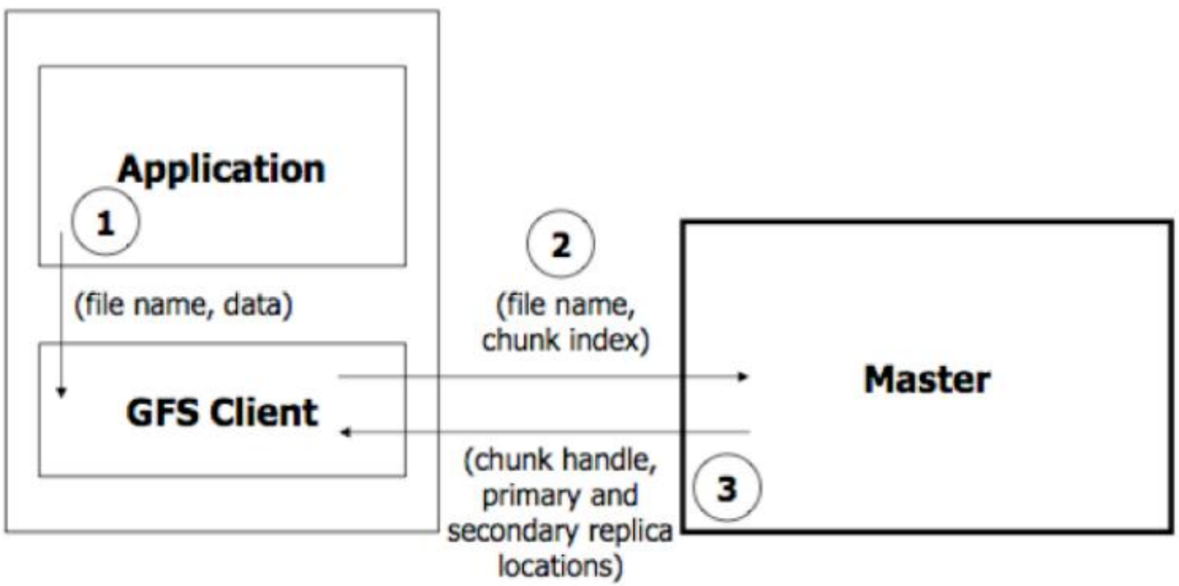
- Problem:
  - Single point of failure
  - Scalability bottleneck
- Solutions:
  - *Shadow* masters

Single Master - Simple

- Problem:
  - Single point of failure
  - Scalability bottleneck
- Solutions:
  - *Shadow* masters
  - Minimize master involvement
    - **Chunk leases:** master delegates authority to primary replicas in data mutations
    - never move data through it, use only for metadata

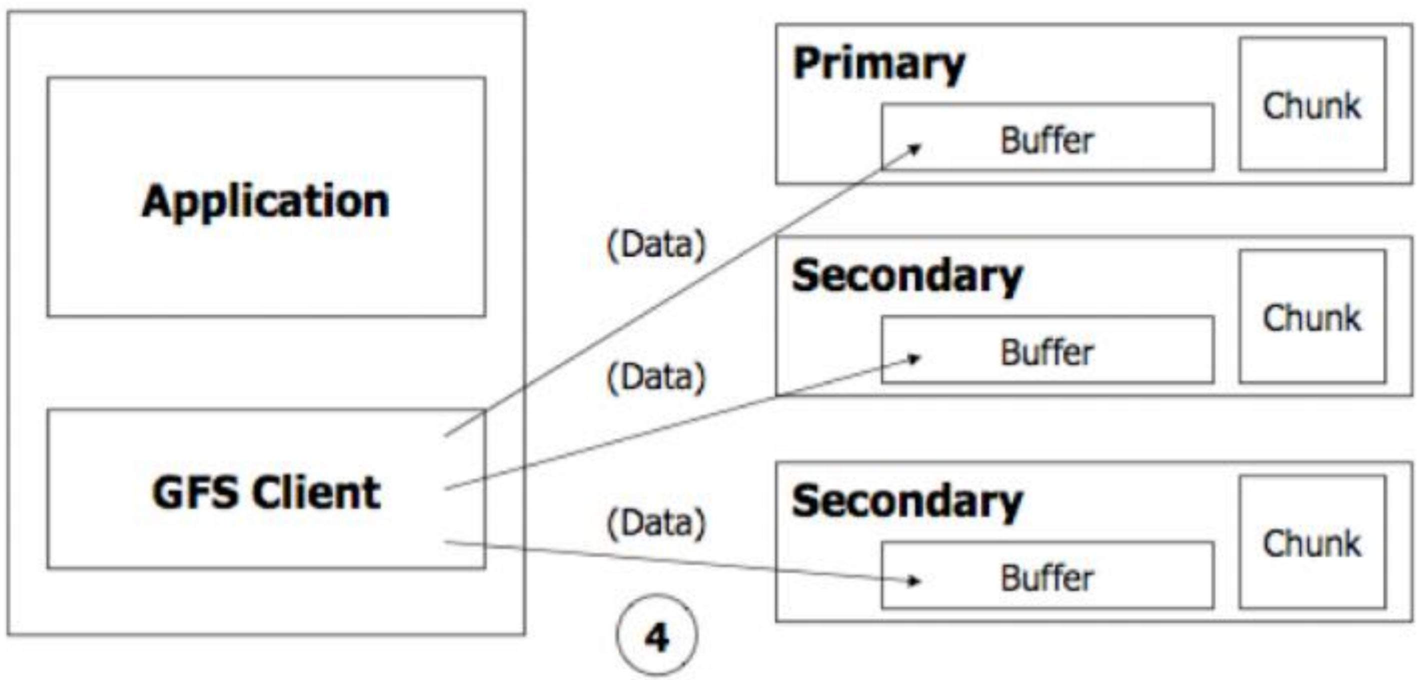
Write Algorithm

1. Application originates the request
2. GFS client translates request and sends it to master
3. Master responds with chunk handle and replica locations



Write Algorithm

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

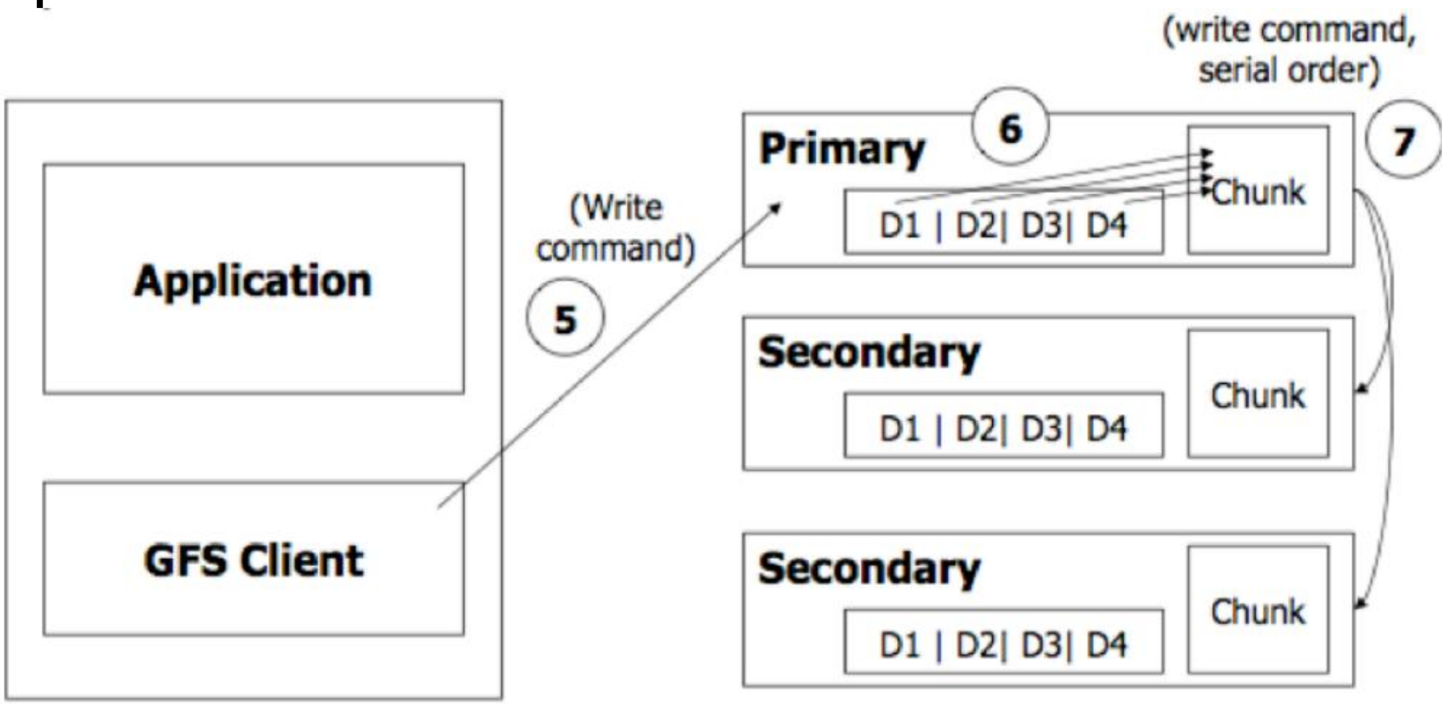


Write Algorithm

5. Client sends write command to primary

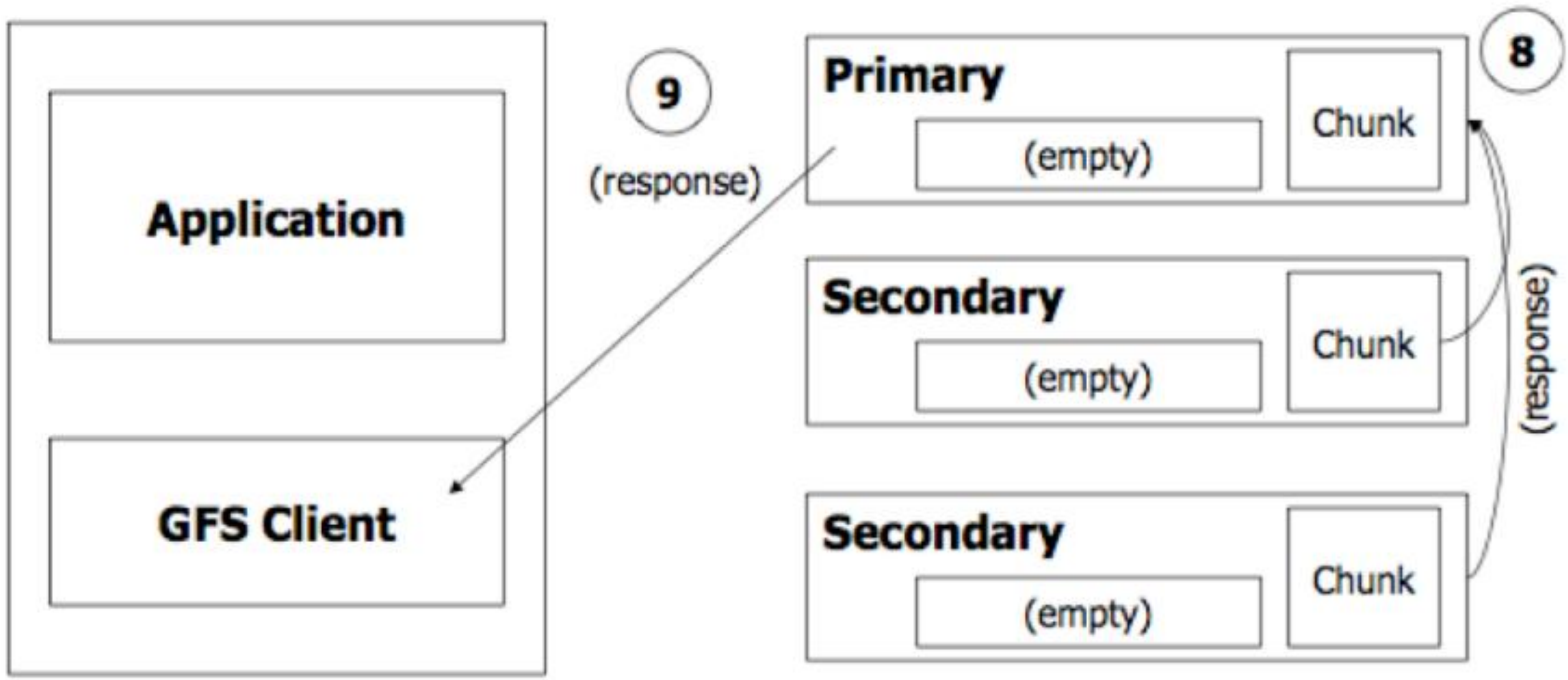
6. **Primary determines serial order for data mutations** in its buffer and writes the mutations in that order to the chunk

7. Primary sends the serial order to the secondaries and tells them to perform the write



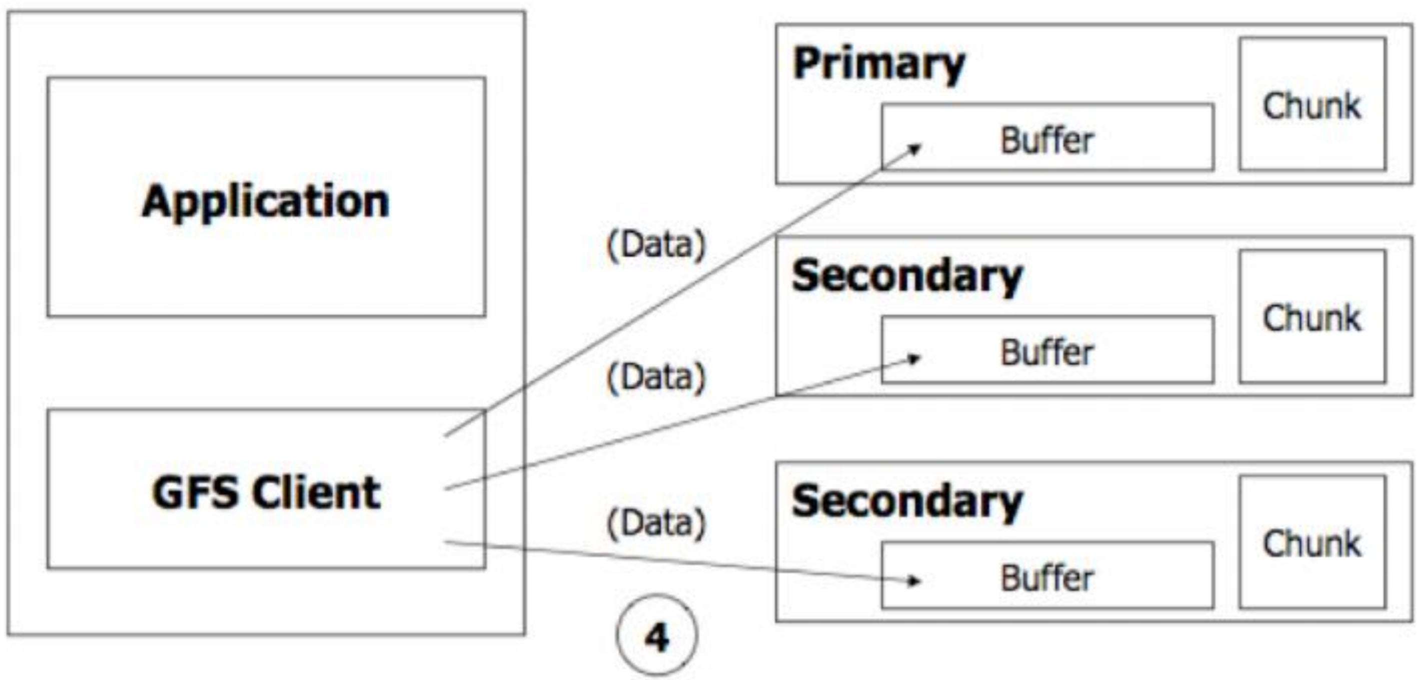
Write Algorithm

8. Secondaries respond back to primary
9. Primary responds back to the client



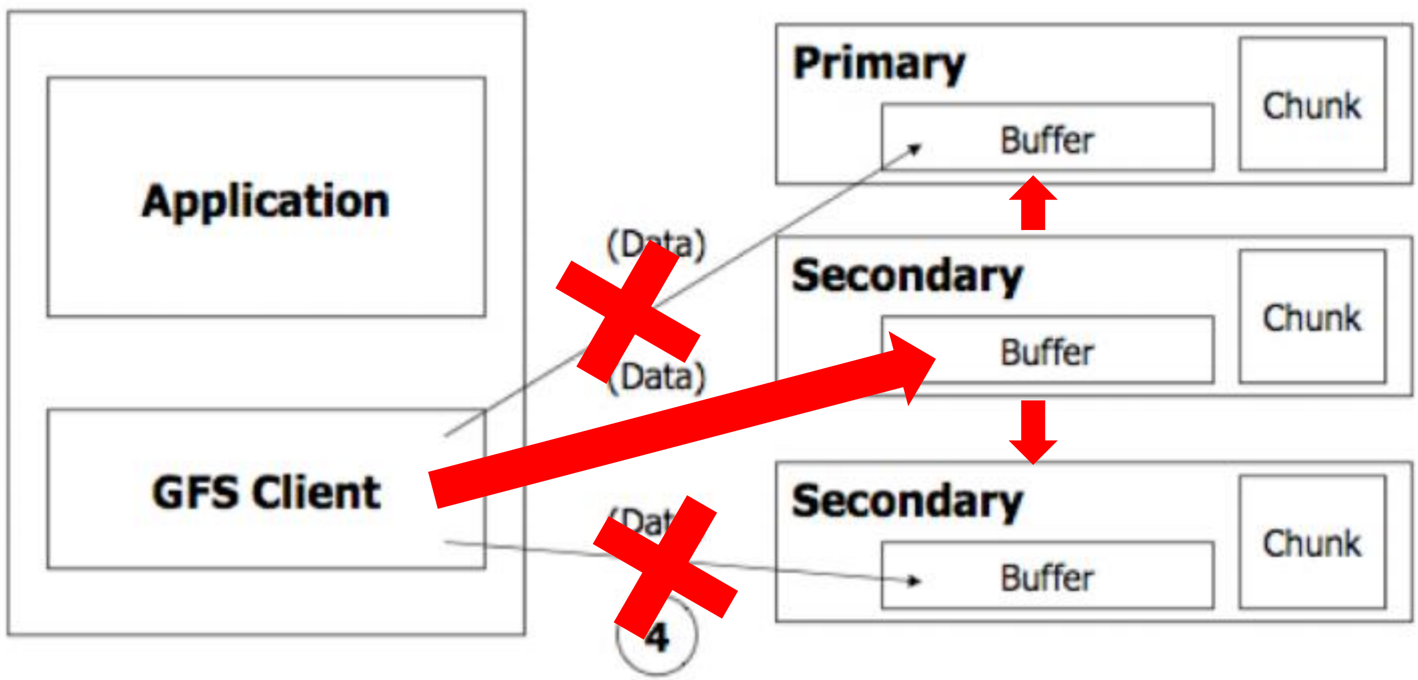
Write Algorithm

4. Client pushes write data to all locations. Data is stored in chunkserver’s internal buffers



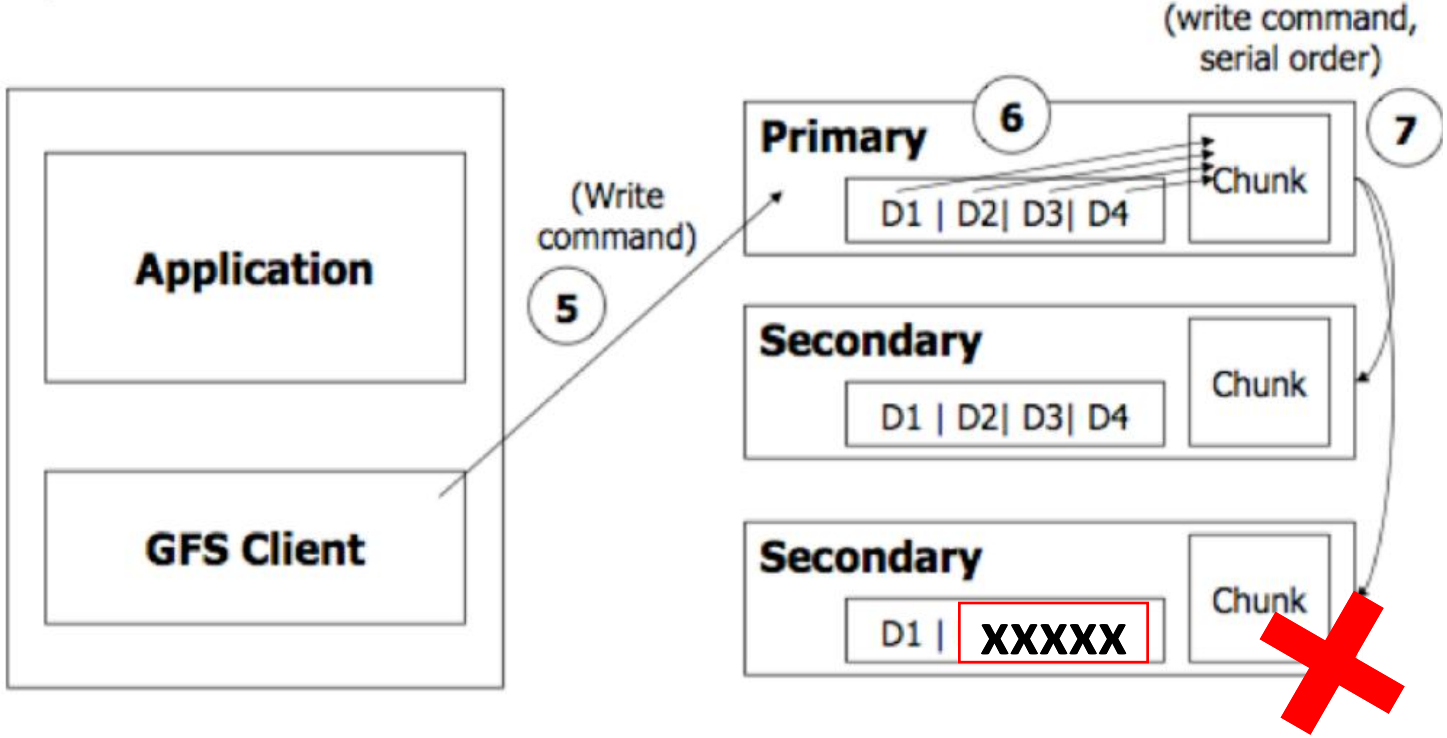
Write Algorithm

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



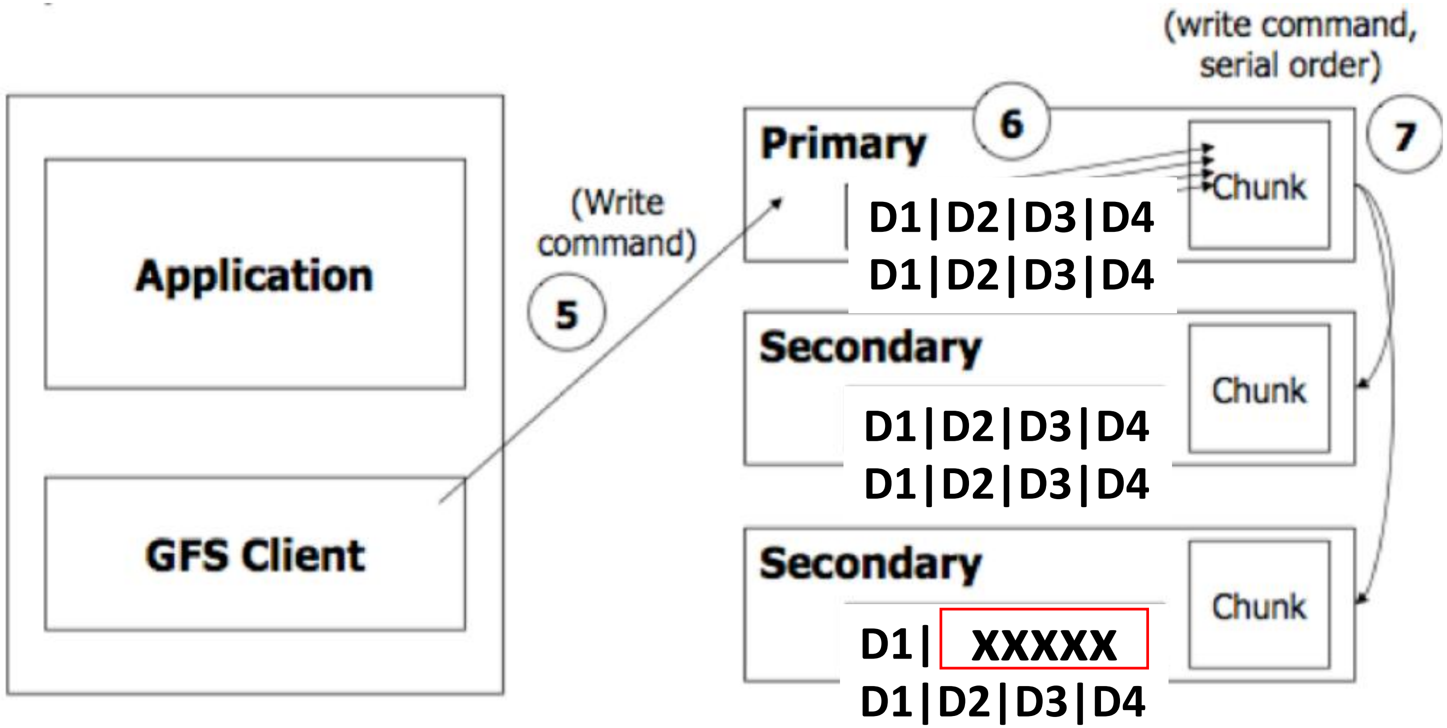
Write Failure

- Applications need to deal with failures
  - Rewrite
  - Chunks are not bitwise identical
    - Use checksum to skip inconsistent file regions



Write Failure

- Applications need to deal with failures
  - Rewrite
  - Chunks are not bitwise identical
    - Use checksum to skip inconsistent file regions



Questions?