Cloud Computing

Cloud Storage Systems 云存储系统

Minchen Yu

SDS@CUHK-SZ

Fall 2024

香港中文大學（深圳）
The Chinese University of Hong Kong, Shenzhen

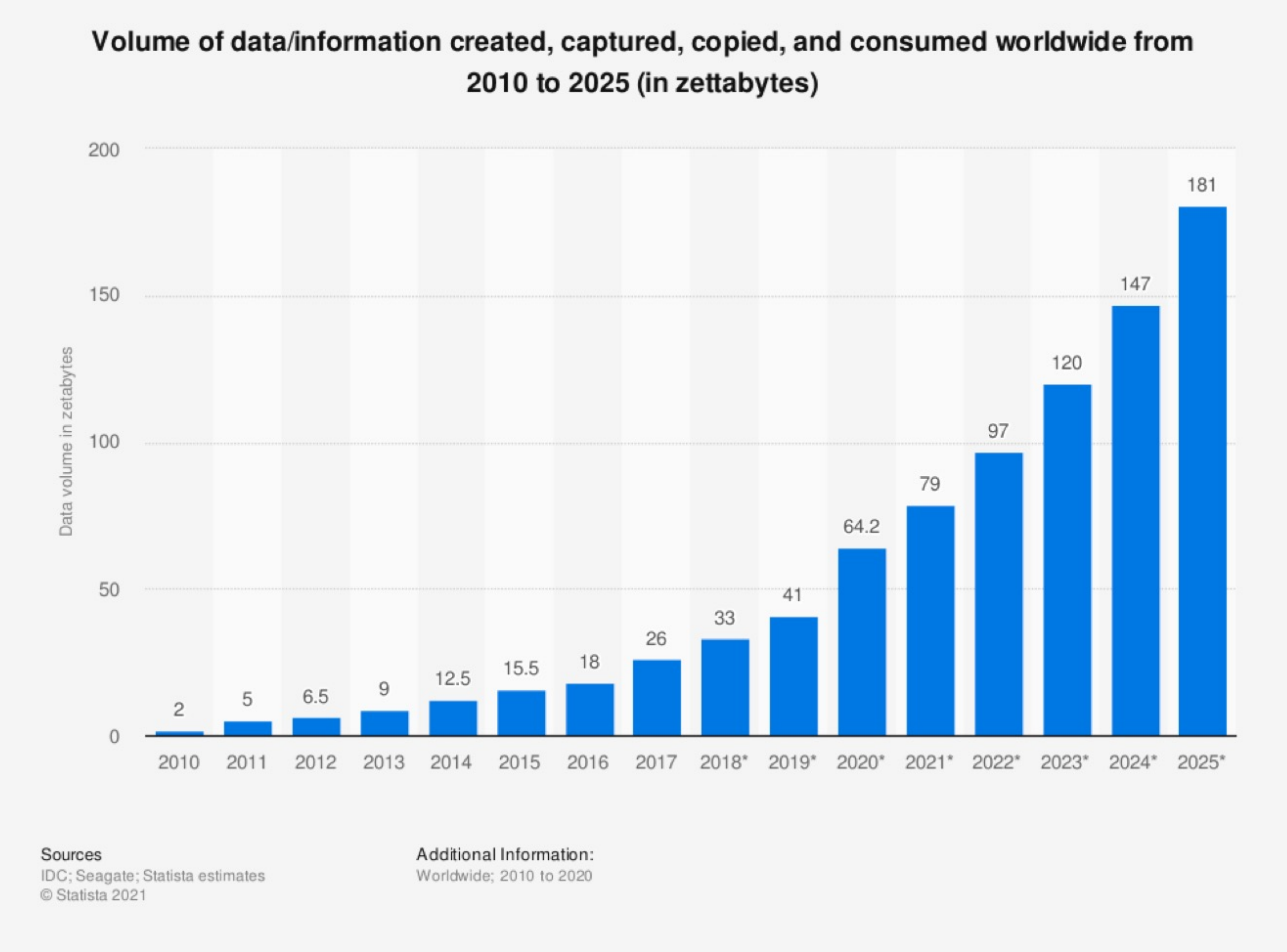
SCHOOL OF DATA SCIENCE
數據科學學院

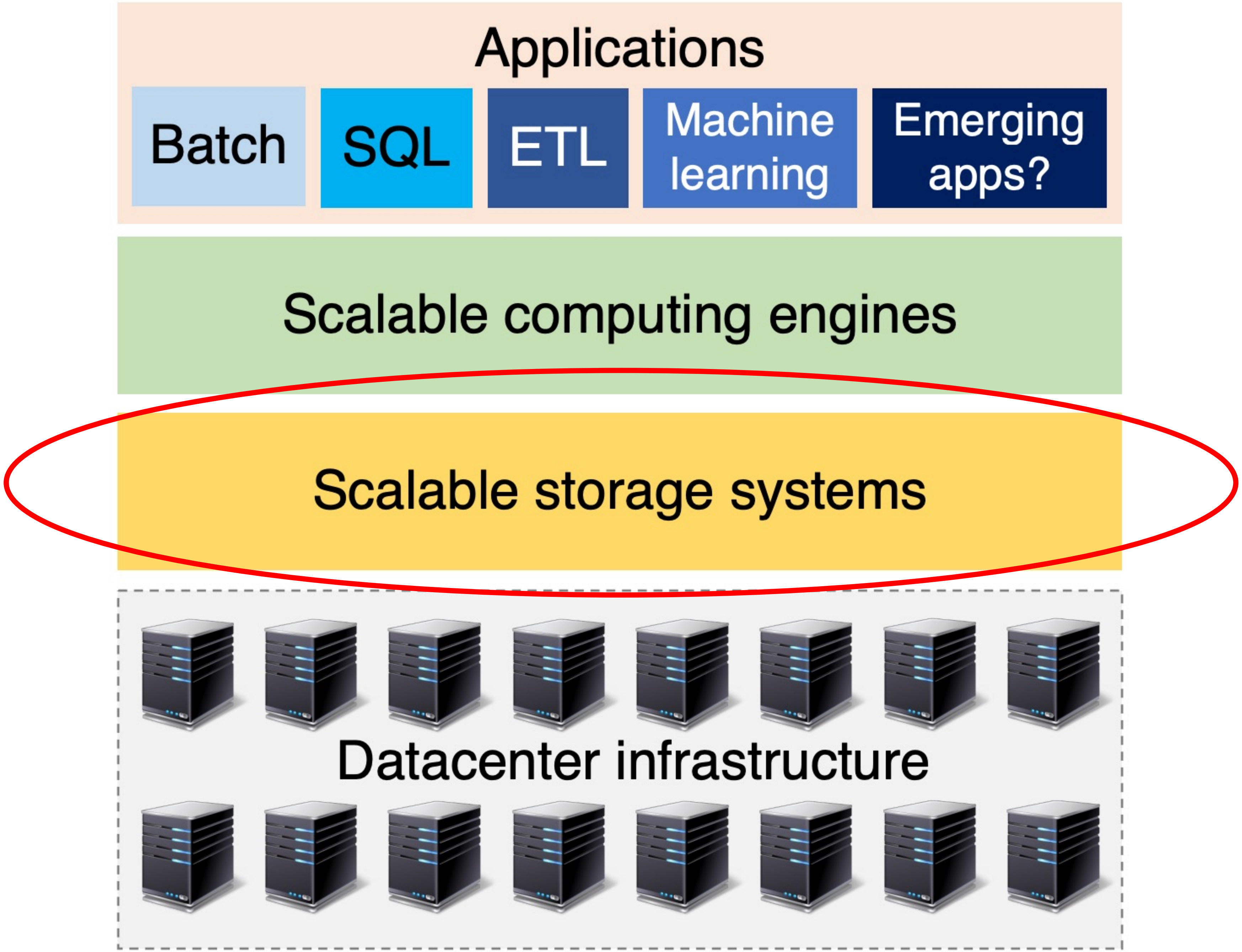
Massive data volume

Processes 20 PB a day (2008)
Crawls 20B web pages a day (2012)
Search index is 100+ PB (5/2014)
Bigtable serves 2+ EB, 600M QPS (5/2014)

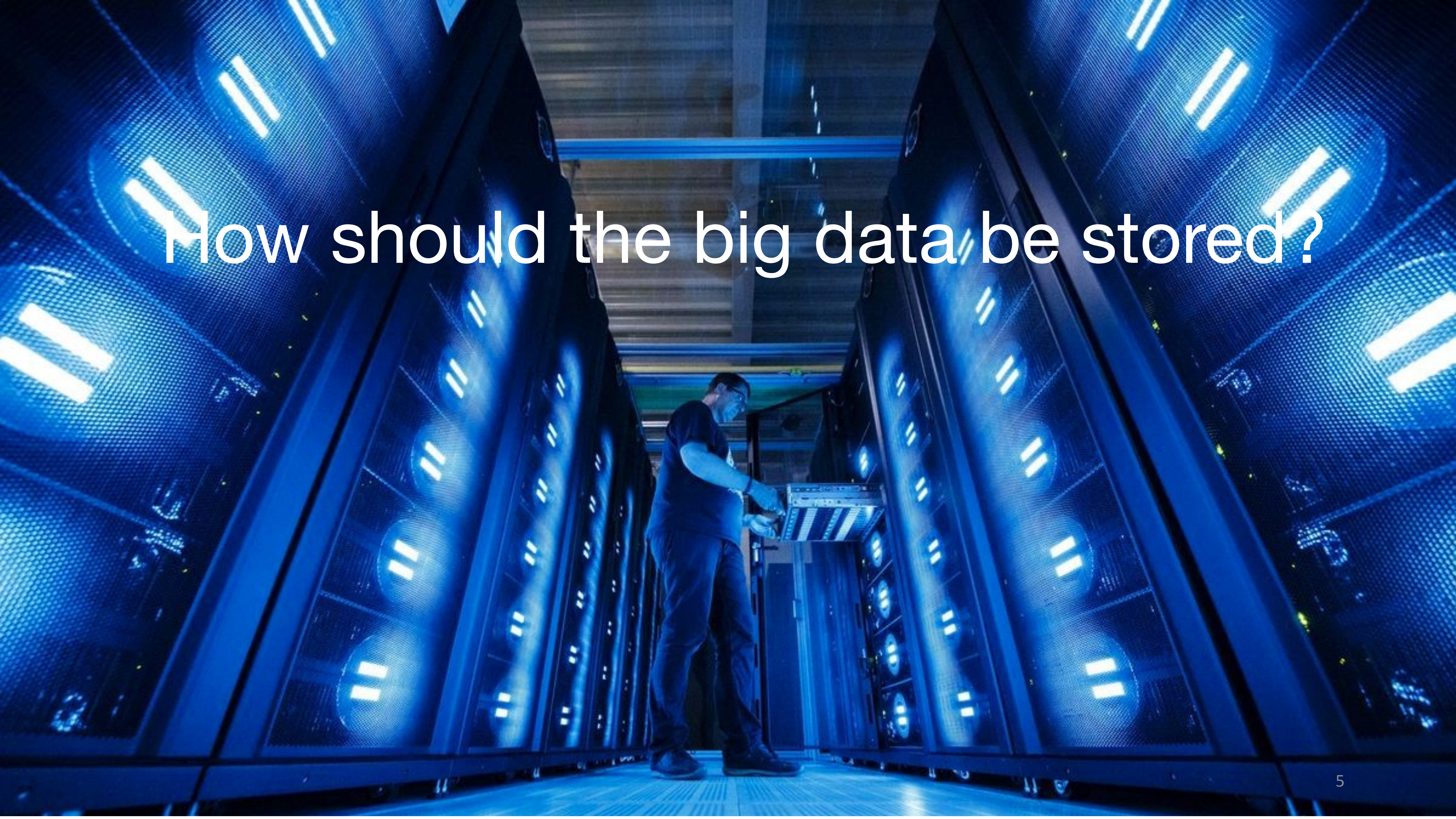
300 PB data in Hive +
600 TB/day (4/2014)

S3: 2T objects, 1.1M
request/second (4/2013)


Volume of data/information created, captured, copied, and consumed worldwide from 2010 to 2025 (in zettabytes)
Data volume in zetabytes
200
150
100
50
0
2
5
6.5
9
12.5
15.5
18
26
33
41
64.2
79
97
120
147
181
2010
2011
2012
2013
2014
2015
2016
2017
2018*
2019*
2020*
2021*
2022*
2023*
2024*
2025*
Sources
IDC; Seagate; Statista estimates
© Statista 2021
Additional Information:
Worldwide; 2010 to 2020


Applications
Batch
SQL
ETL
Machine learning
Emerging apps?
Scalable computing engines
Scalable storage systems
Datacenter infrastructure

How should the big data be stored?



Can we just have a BIG disk?

Motivating app: Web search 激励应用：网络搜索

Crawl the whole web

Store it all on “one BIG disk”

Process users’ searches on “one BIG CPU”

抓取整个网络，将所有内容存储在“一个大磁盘”上
在“一个大 CPU”上处理用户的搜索

Does it scale?

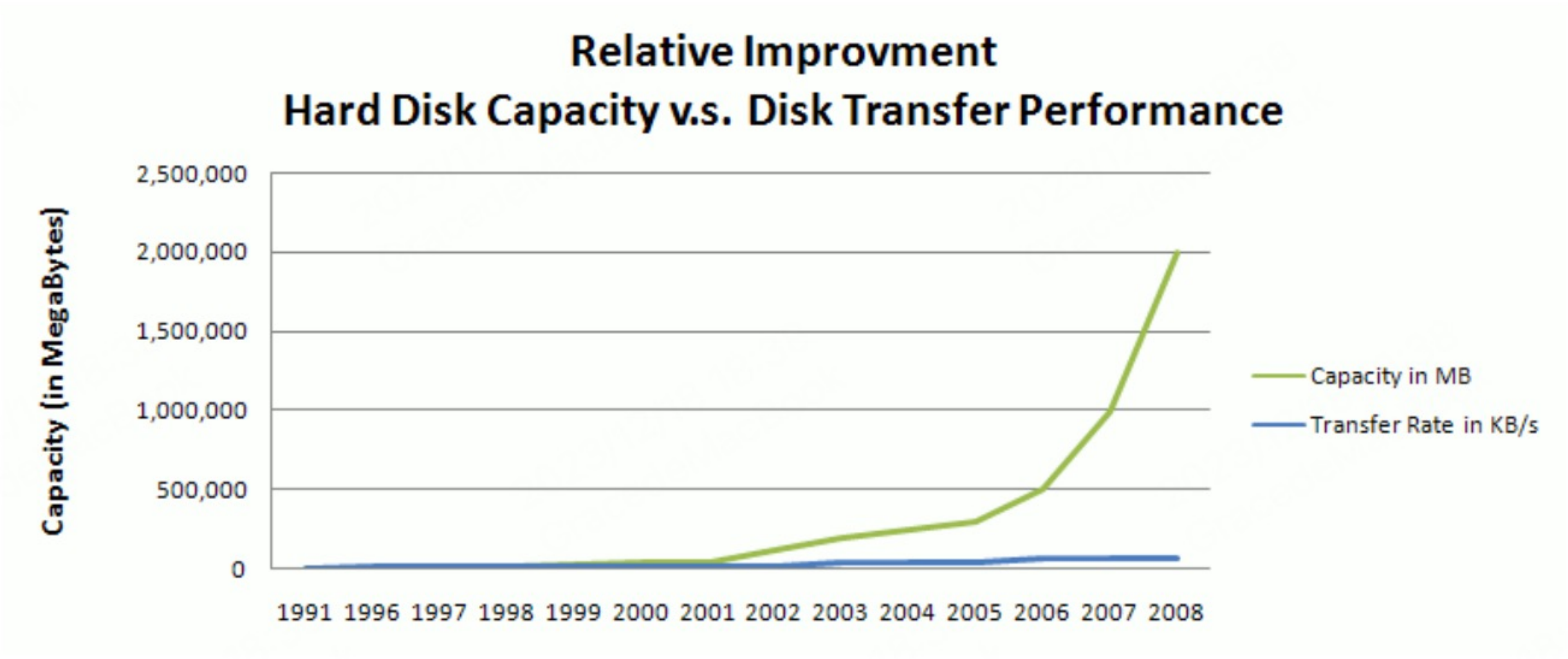
I/O is a bottleneck

Suppose we have crawled 100 Tb worth of data

State-of-the-art 7,200 rpm SATA drive has 3 Gbps I/O

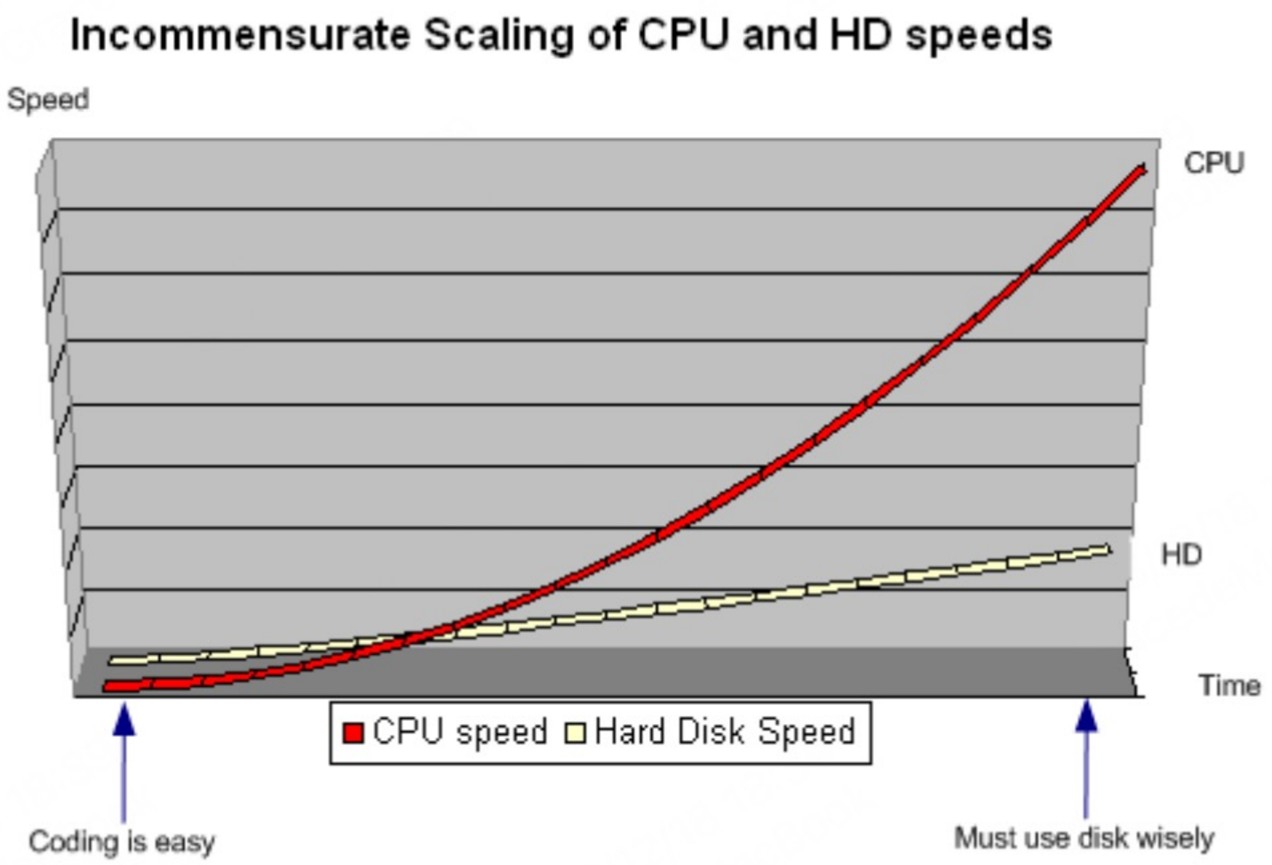
It takes 100 Tb / 3 Gbps = **9.3 hours** to scan through the entire data!

I/O lags far behind



Source: R1Soft, http://wiki.r1soft.com/pages/viewpage.action?pageId=3016608

I/O lags far behind



Source: R1Soft, http://wiki.r1soft.com/pages/viewpage.action?pageId=3016608

Other issues

Building a high-end supercomputer is much, much costly

建造一台高端超级计算机的成本非常非常高

Storing all data in one place adds the risk of hardware failures

将所有数据存储在一个地方增加了硬件故障的风险

- putting all eggs in one basket!

Is there a way out?

Google’s answer: scale “out”, not “up”!

Scale “out”, not “up”!

Lots of cheap, commodity PCs, each with disk and CPU

大量廉价的商品PC，每台都带有磁盘和CPU

- 100s to 1000s of PCs in cluster in early days

早期集群中有100到1000台PC

High aggregate storage capacity

高集料储存能力

- No costly “big disk”

没有昂贵的 “ 大磁盘

将搜索处理扩展到多台机器上

Spread search processing across many machines

- High I/O bandwidth, proportional to the # of machines

高I/O带宽，与机器的数量成正比

- Parallelize data processing

并行化数据处理

Scale “out”, not “up”!

Suppose we have crawled 100 Tb worth of data and stored in a 1000-node cluster 假设我们抓取了 100 TB 的数据，并将其存储在 1000 个节点的集群中

- State-of-the-art 7,200 rpm SATA drive has 3 Gbps I/O
- 1000 nodes provide 3 Tbps aggregated I/O, 1000x faster than
- the BIG disk

It takes 100 / 3 = **33 seconds** to scan through the entire data!

A cool idea! But wait…

The joys of real HW in the 1st year

~0.5 overheating (power down most machines in <5 mins, ~1-2 days to recover)

~1 PDU failure (~500-1000 machines powered down, ~6 hours)

~1 rack-move (~500-1000 machines powered down, ~6 hours)

~1 network rewiring (rolling ~5% of machines down over 2-day span)

~20 rack failures (40-80 machines instantly disappear, 1-6 hours to get back)

~5 racks go wonky (40-80 machines see 50% packet loss)

~8 network maintenance (4 might cause ~30-min connectivity loss)

~1000 individual machine failures, ~thousands of hard drive failures, slow disks, bad memory, misconfigured machines, flaky machines, etc.

过热；PDU 故障；机架移动；网络重新布线；机架故障；机架出现问题；网络维护；单机故障；硬盘故障；磁盘速度慢；内存故障；机器配置错误；机器不稳定等等。

Implications

Stuff breaks 会导致损坏

- If you have one server, it may stay up 3 years (1,000 days)
- If you have 10k servers, expect to lose 10 a day 0.1%的损耗率

"Ultra-reliable" hardware doesn't really help 超可靠 " 硬件并没有真正的帮助

- At large scales, super-fancy reliable hardwares still fails, albeit less frequently 在大规模情况下，超高可靠性的硬件仍然会失效，虽然降低了失效概率
  - software still needs to be fault-tolerant 软件仍然需要容错
  - commodity machines w/o fancy h/w give better perf/$ 没有花哨硬件的商用机器提供更好的性能/价格

Reliability has to come from the software!

可靠性必须来自于软件!

GFS: The Google File System

A highly reliable storage system built atop highly unreliable hardwares

在高度不可靠的硬件上构建的高度可靠的存储系统

GFS

Goal: a global (distributed) file system that stores data across many machines 目标：一个跨多台机器存储数据的全局(分布式)文件系统

- Need to handle 100's TBs 需要处理100 TB

Google published details in 2003

Open source implementation: Hadoop Distributed File System (HDFS)

Hadoop分布式文件系统 (HDFS)

Outline

Target environment
Design decisions
General architecture
File read and write
Fault tolerance
Measurements
Conclusions

目标环境
设计决策
总体架构
文件读写
容错
测量
结论

Implications

Thousands of computers

Distributed 数千台计算机：分布式

- Computers have their own disks, and the file system spans those disks

  计算机有自己的磁盘，文件系统跨越这些磁盘

Failures are the norm

失败是常态

- Disks, networks, processors, power supplies, application software, OS software, human errors 磁盘、网络、处理器、电源、应用软件、操作系统软件、人为错误

Target environment 目标环境

Files are huge, but not many 文件很大，但数量不多

- >100M, usually multi-gigabyte
- a few million files

>100M，通常为几千兆字节；几百万个文件

Write-once, read-many 一次写入，多次读取

- Files are mutated by appending 通过附加内容改变文件
- Once written, files are typically only read 一旦写入，文件通常只能被读取
- Large streaming reads and small random reads are typical

大型流式读取和小型随机读取很常见

Target environment

I/O bandwidth is more important than latency I/O 带宽比延迟更重要

- Suitable for batch processing and log analytics

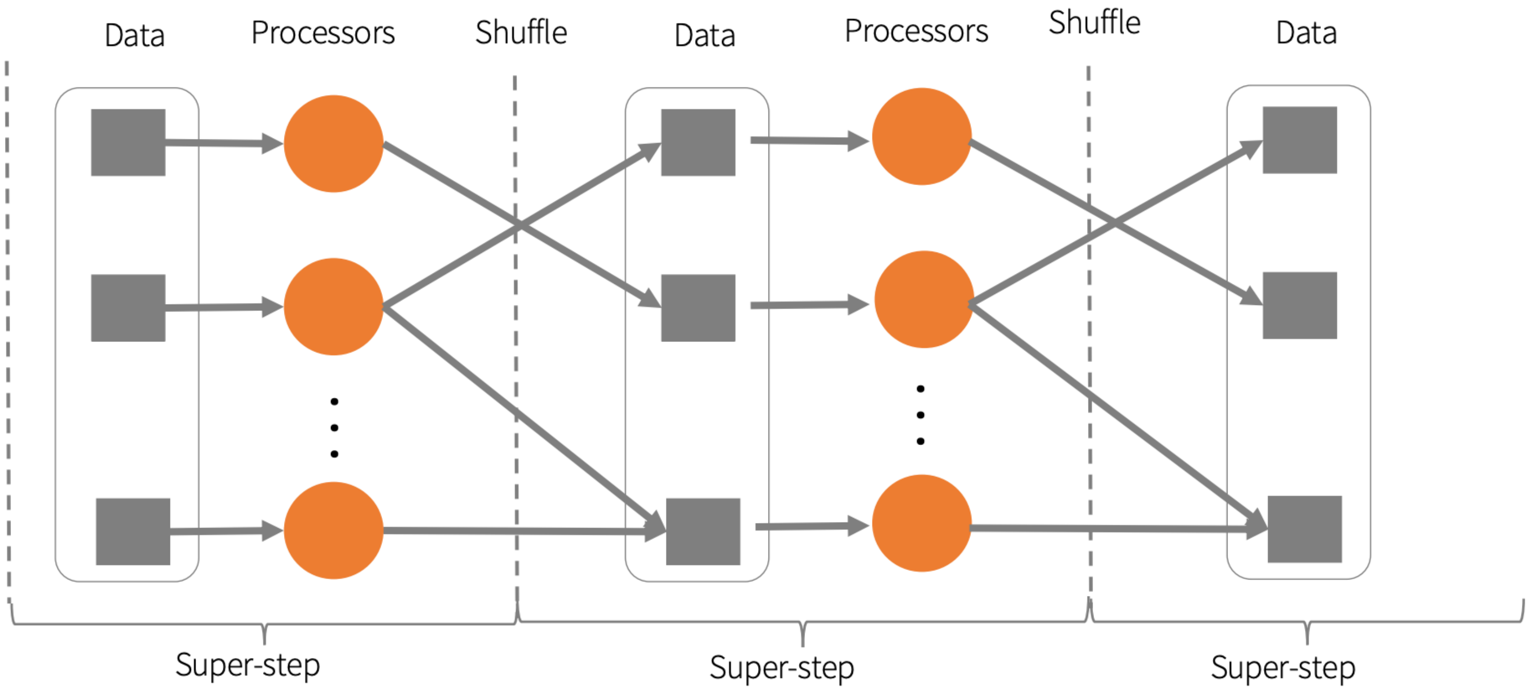
适用于批处理和日志分析

It’s helpful if the file system provides synchronization for concurrent appends

如果文件系统能够为并发追加提供同步功能，将会很有帮助

Example workloads

Bulk Synchronous Processing (BSP)   批量同步处理(BSP)

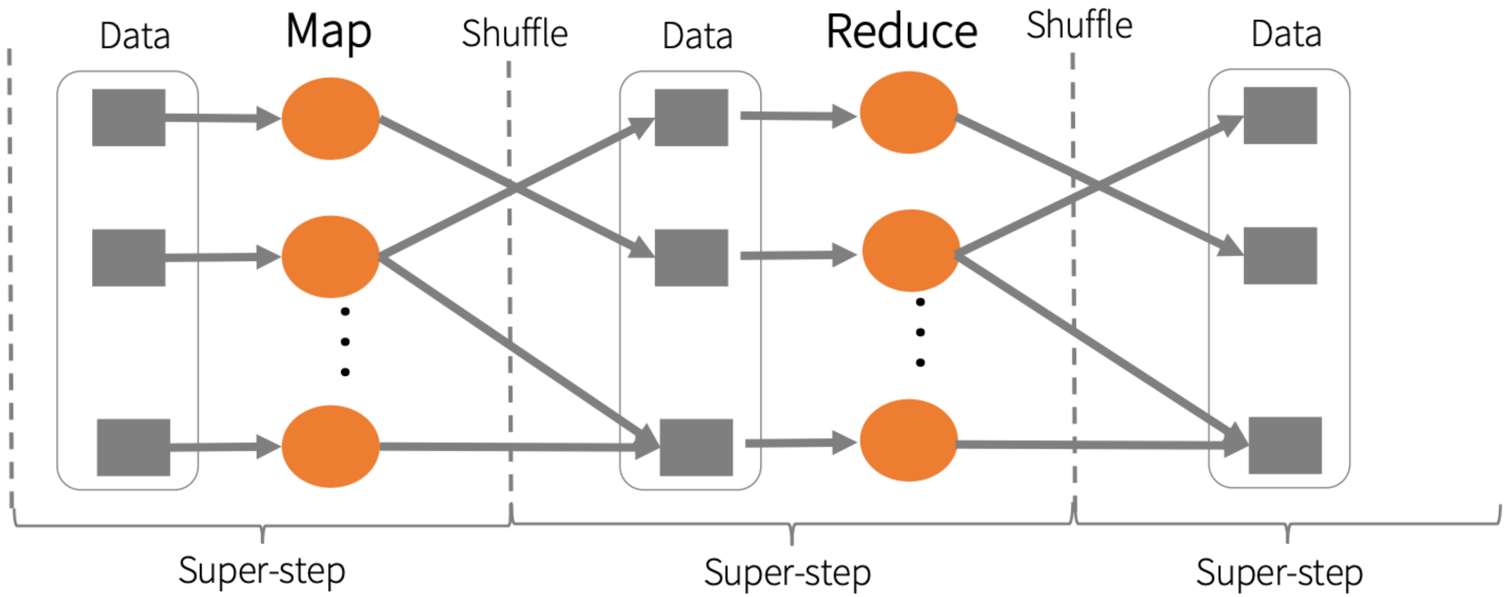


Leslie G. Valiant, A bridging model for parallel computation, Communications of the ACM, Volume 33 Issue 8, Aug. 1990

Example workloads

MapReduce as a BSP system

MapReduce 作为 BSP系统



Read entire dataset, do computation over it -> batch processing

读取整个数据集，对其进行计算->批处理

GFS Design Decisions

GFS 设计决策

File stores as/divided into chunks

文件存储作为/分为块

Large chunk size: 64 MB 大块大小：64 MB

- a file smaller than a single chunk does not occupy a full chunk's worth of storage 小于单个块的文件不占用完整块的值存储量

Why large chunks? 为什么是大块?

- minimizes the cost of disk seeks 最大限度地减少磁盘寻道成本
- pushes up file transfer rate to the disk transfer rate

将文件传输速率提升至磁盘传输速率

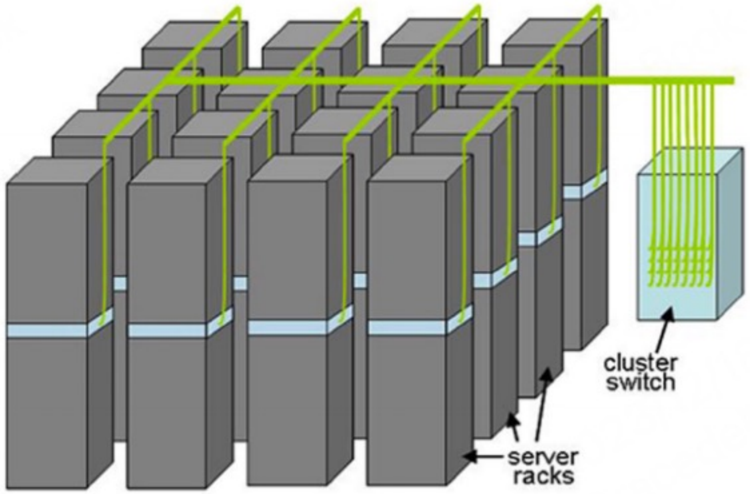
$$T_{File\ transfer} = T_{disk\ transfer} + \underset{0}{\underset{\|}{T_{disk\ seek}}}$$

时间：T文件传输=Tdisk 传输+Tdisk 寻道

Reliability through replication 通过复制实现可靠性

3-way replication 三向复制

- Each chunk replicated across 3+ chunkservers 每个块在3个以上的块服务器之间进行复制
  - 1 replica in the same rack 同一机架中有1个副本
  - 2+ in other, different racks 2+放在其他不同的架子上



Other design decisions 其他设计决策

Single master to coordinate access 单一主机协调访问

- keeps *metadata* 保留元数据
  - filename, permissions, chunk index, folder hierarchy, etc.
    文件名、权限、块索引文件夹层次结构等。
- file data stored in chunkservers 文件数据存储在chunkserver 中

Add record append operations 添加记录追加操作

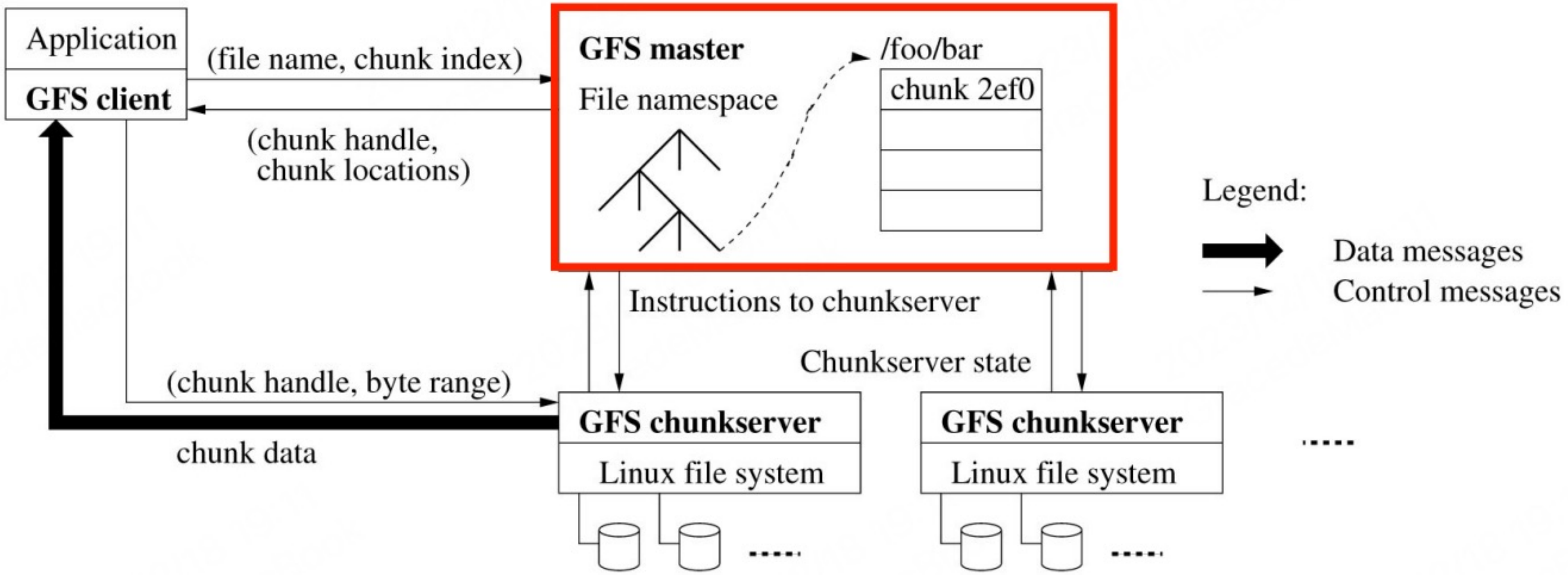
- Support concurrent appends 支持并发附加

General Architecture

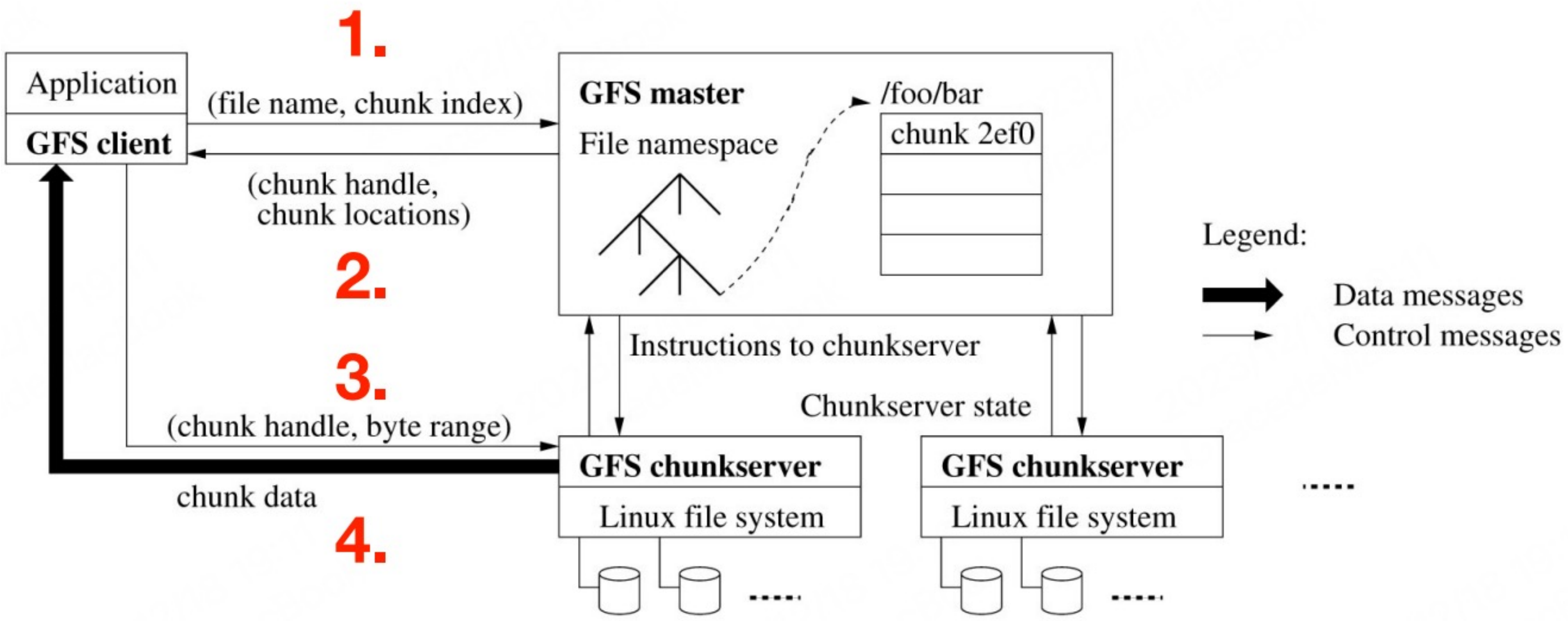
总体架构

Single master
Multiple chunkservers

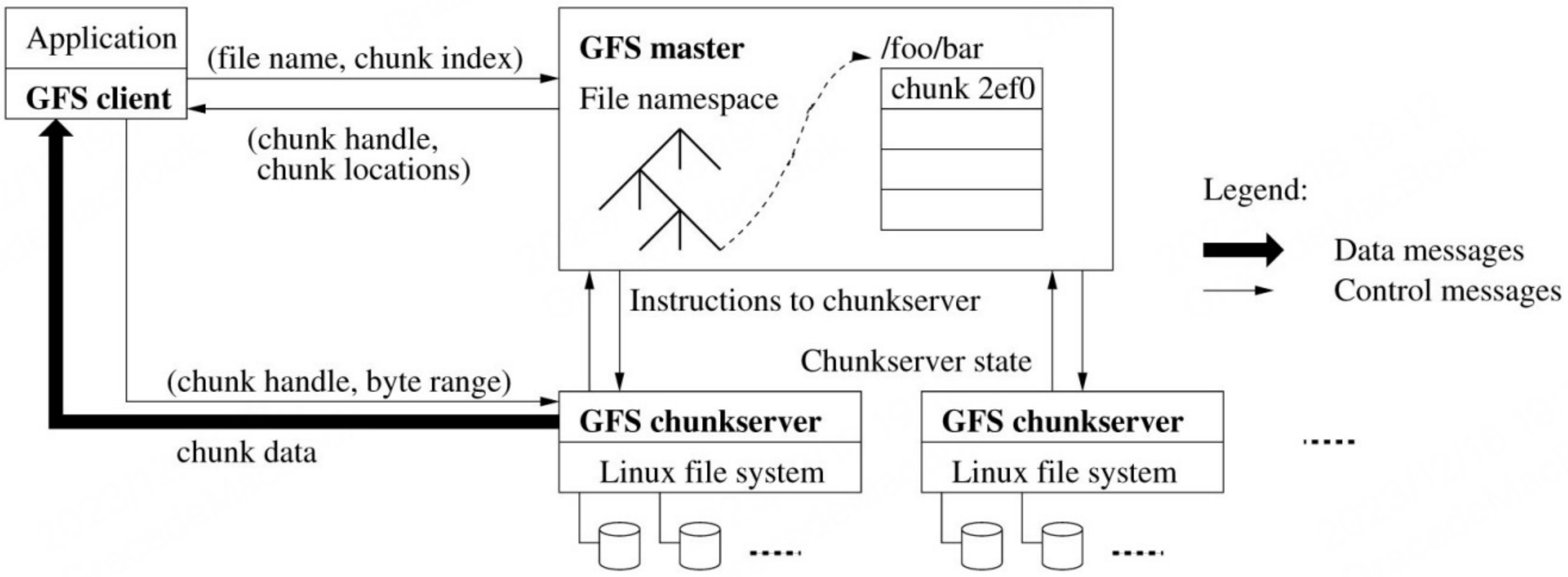
单个主机、多个块服务器



Single master
Multiple chunkservers



Anyone can see the potential problem of this design?



Single master

Problems and concerns

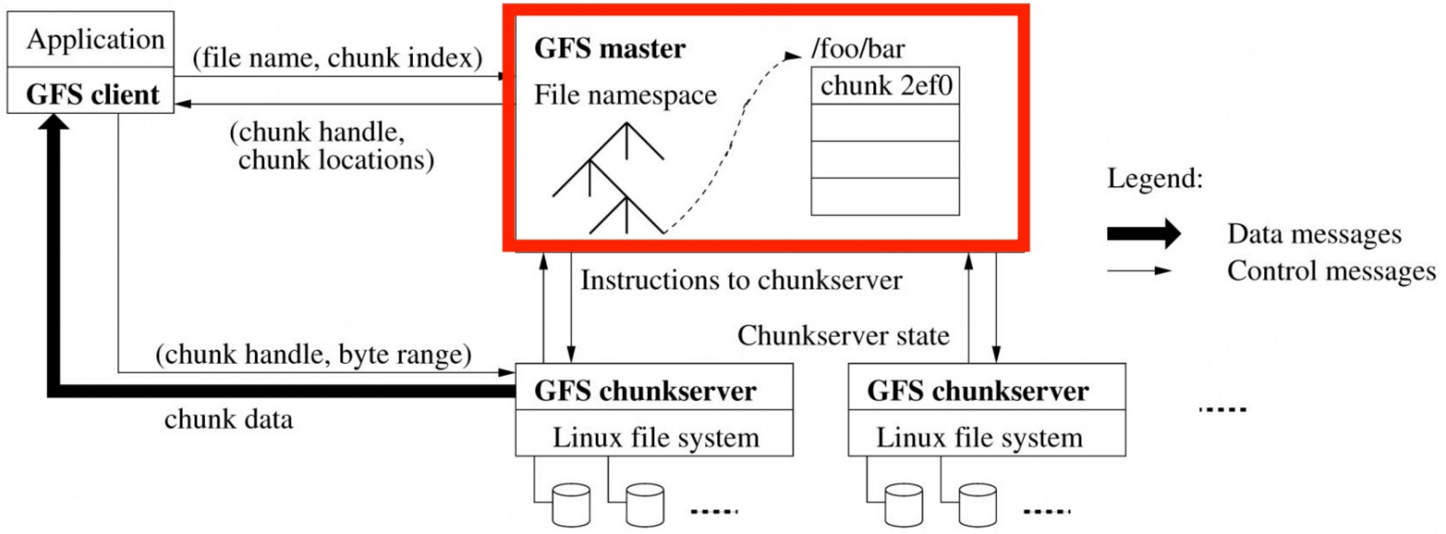
单点故障：如果主服务器离线怎么办?

- **Single point of failure:** what if master goes offline?

可扩展性瓶颈：如果主服务器超载怎么办?

- **Scalability bottleneck:** what if master is overloaded?

Any solutions?



GFS's answers

Master is the single point of failure Master 是单点故障

- add a **shadow master** 那么添加影子主机

Master can be overloaded Master 可能会超载

- Minimize master involvement to address the scalability issue 尽量减少主节点的参与，以解决可扩展性问题
  - Never move data through it, use only for metadata 切勿通过它移动数据，仅用于元数据
  - large chunk size: minimizes seeking/indexing time 大块尺寸：最小化查找/索引时间
  - **chunk leases**: master delegates authority to primary replicas in data mutations 块租约：主节点将数据变更的权限委托给主副本

Metadata

Three types of metadata, all kept *in memory* 三种类型的元数据均保存在内存中

- file and chunk namespaces 文件和块命名空间
- mappings from files to chunks (each chunk has a unique ID) 从文件到块的映射(每个块都有唯一的ID)
- locations of each chunk’s replicas 每个块副本的位置

Metadata

Three types of metadata, all kept *in memory*

- file and chunk namespaces
- mappings from files to chunks (each chunk has a unique ID)
- locations of each chunk's replicas

First two types are made persistent via an **operation log**

前两种类型通过操作日志持久化

Metadata

Three types of metadata, all kept *in memory*

- file and chunk namespaces
- mappings from files to chunks (each chunk has a unique ID)
- locations of each chunk’s replicas

Chunk replica locations learned by polling chunkservers at startup

Chunkserver is final arbiter of what chunks it holds

在启动时通过轮询块服务器了解块副本的位置，Chunkserver 是其所持有的块的最终管理者

Master’s responsibilities 主机职责：

Metadata storage 元数据存储

Namespace management/locking 命名空间管理/锁定

Periodic communication with chunkservers 与块服务器定期通信

- give instructions, collect state, track cluster health 给出指令、收集状态、跟踪集群健康状况

Chunk creation, re-replication, rebalancing 块创建、重新复制、重新平衡

- spread replicas across racks to reduce correlated failures 将副本分散到各个机架，以减少相关故障
- re-replicate data if redundancy falls below a threshold 如果冗余度低于阈值，则重新复制数据

Master’s responsibilities

Garbage collection 垃圾收集

- simpler, more reliable than traditional file delete 比传统文件删除更简单、更可靠
- master logs the deletion, renames the file to a hidden name master 记录删除，将文件重命名为隐藏名称
- lazily garbage collects hidden files 惰性垃圾收集隐藏文件

Stale replica deletion 删除过时的副本

- detect “stale” replicas using chunk version numbers 使用块版本号检测 “ 过时 ” 的副本

Chunkserver

块服务器

Stores 64 MB file chunks on local disk, each with **version number** and **checksum**

在本地磁盘上存储64 MB 文件块，每个文件块都有版本号和校验和

Read/write requests specify **chunk handle** and **byte range**

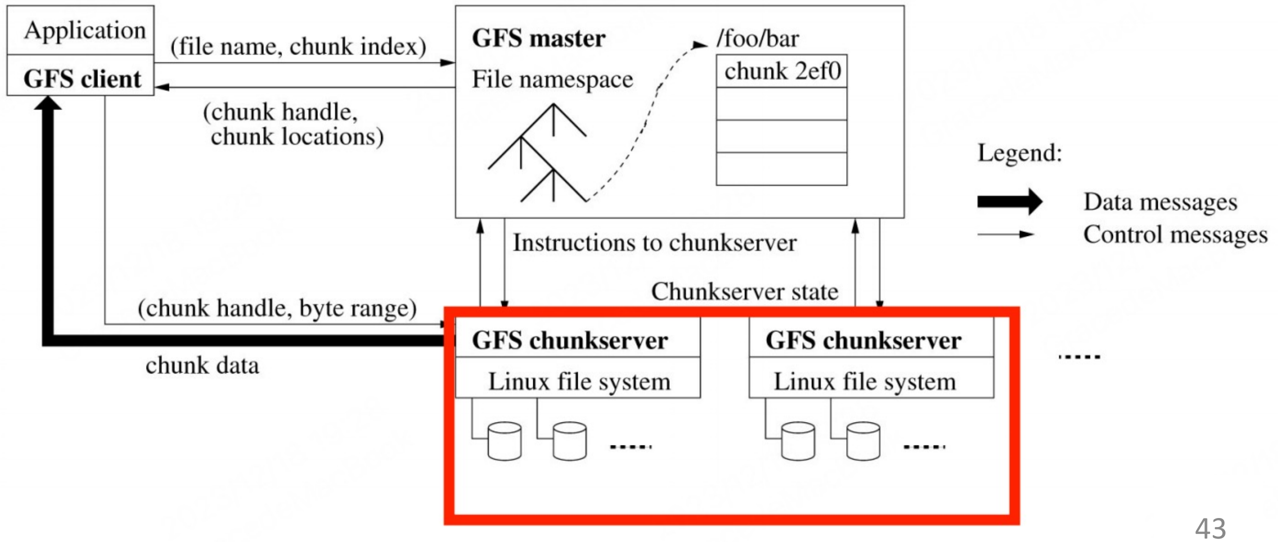
读/写请求指定块句柄和字节范围

Chunks replicated on configurable number of chunkservers (default: 3-way replication)

可配置数量的块服务器上复制的块(默认值：三向复制)

No file data caching

无文件数据缓存



Client 客户机

Issues control (metadata) requests to master

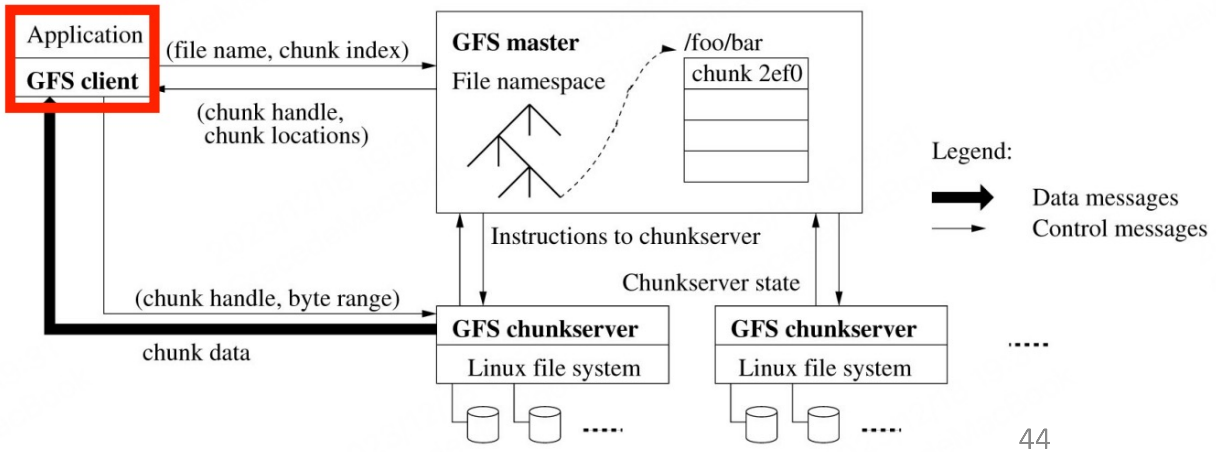
- e.g., ls 向主服务器发出控制(元数据)请求 例如，ls

Issues data requests directly to chunkservers, e.g., cat

- minimum master involvement 直接向块服务器发出数据请求，例如 cat ；最低限度的master参与

Caches no file data but metadata

不缓存文件数据，但缓存元数据

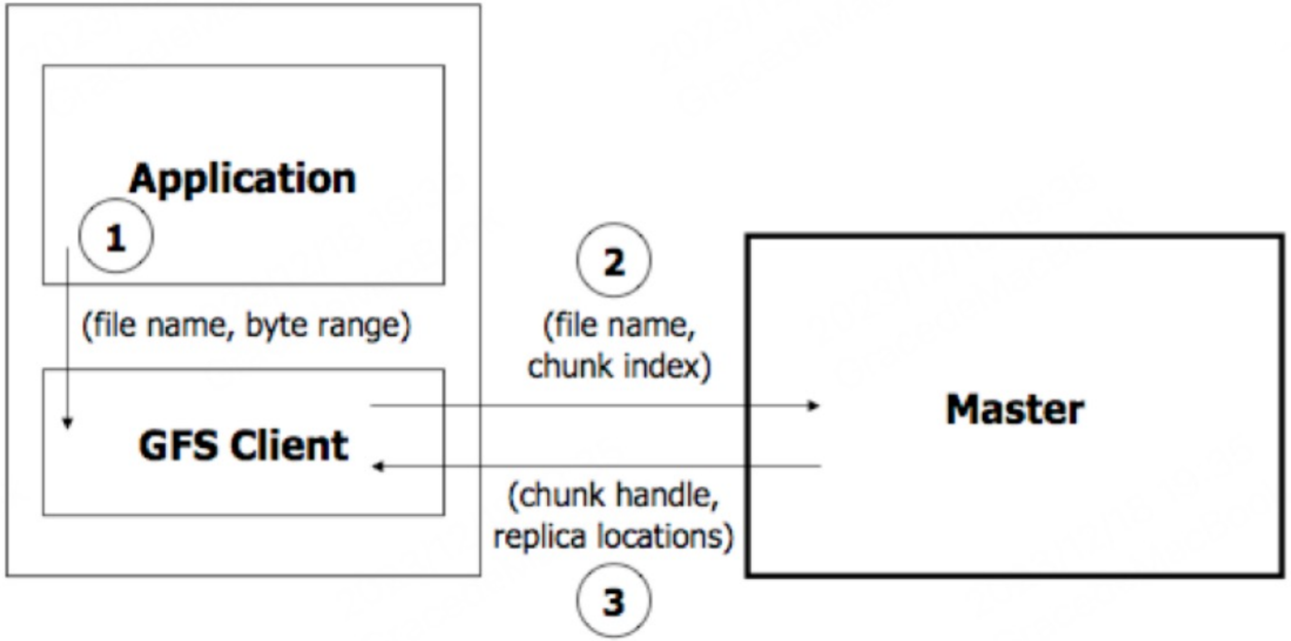


File read and write

文件读写

File read 读取文件

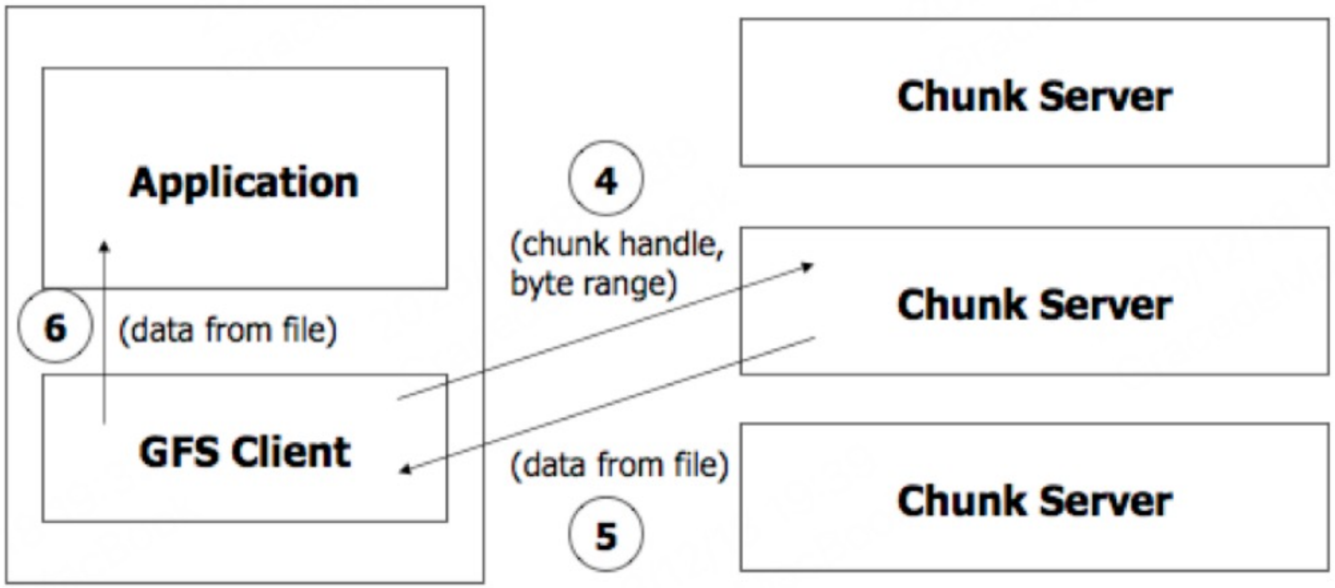
1. Application originates the read request 应用程序发起读取请求
2. GFS client translates request and sends it to master GFS 客户端转换请求并将其发送给主服务器
3. Master responds with chunk handle and replica locations 主服务器响应块句柄和副 本位置



File read

客户端选择 “ 最近 ” 的位置并发送请求

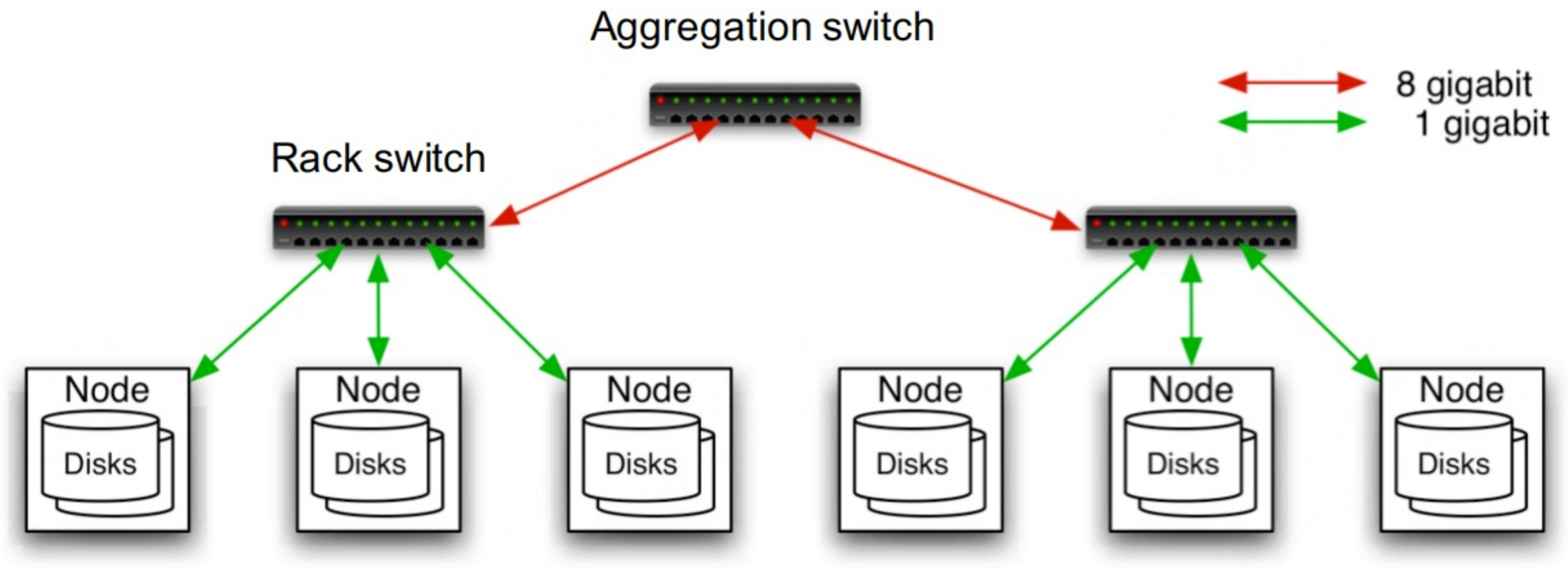
4. Client picks the “closest” location and sends the request
5. Chunkserver sends requested data to the client Chunkserver将请求的数据发送给客户端
6. Client forwards the data to the applicaiton 客户端将数据转发给应用程序



How to choose the “closest” block?

如何选择 “ 最近 ” 的区块?

Choosing the "closest" block



Choosing the "closest" block

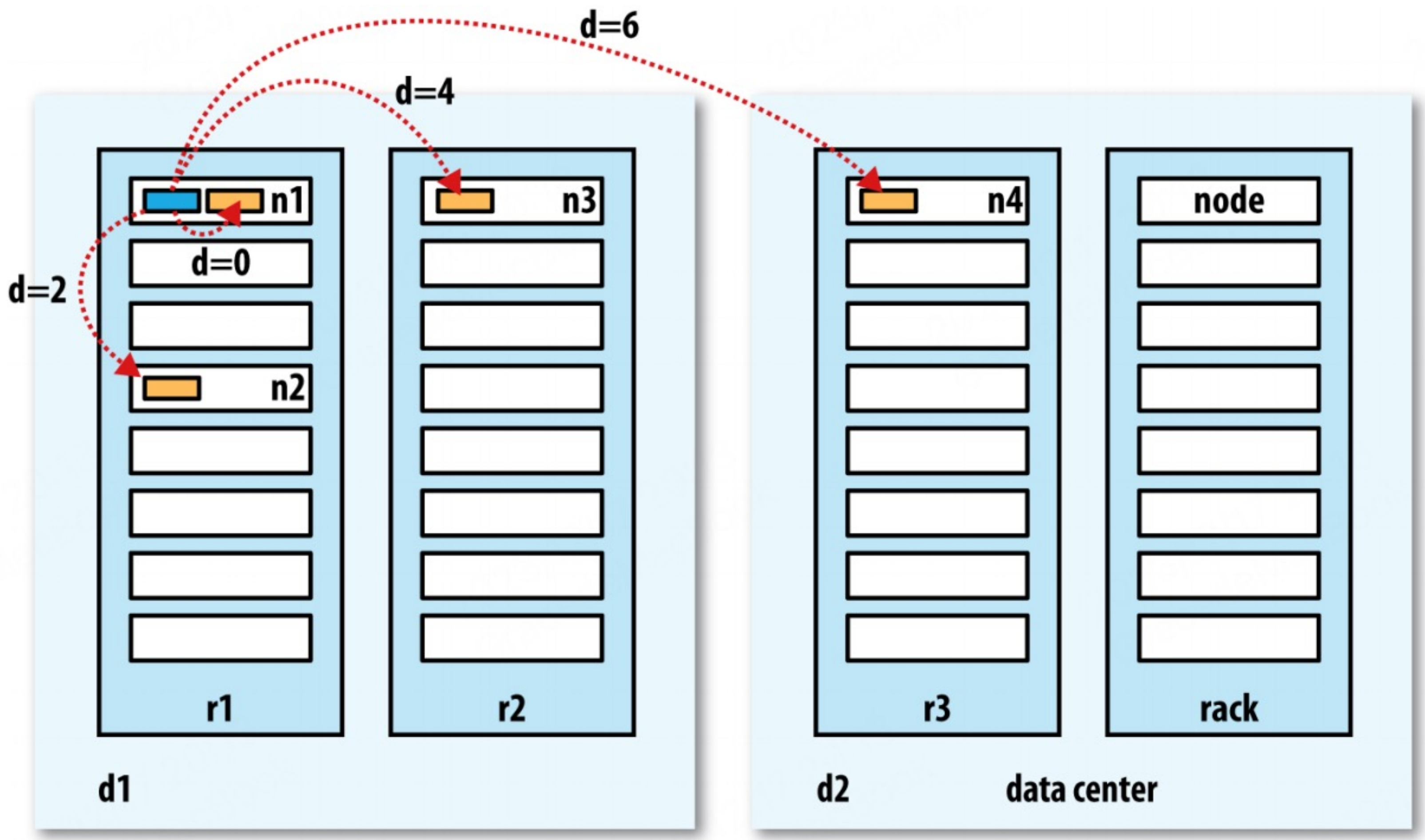
Computing the ***distance*** between two nodes 计算两个节点之间的距离

将 DC d1中机架r1上的节点n1表示为/d1/r1/n1

Denote a node *n1* on rack *r1* in DC *d1* by */d1/r1/n1*

- dist(*/d1/r1/n1*, */d1/r1/n1*) = 0 (process on the same node) 同一节点上的进程
- dist(*/d1/r1/n1*, */d1/r1/n2*) = 2 (different nodes on the same rack) 同一机架上的不同节点
- dist(*/d1/r1/n1*, */d1/r2/n3*) = 4 (nodes on different racks in the same datacenter) 同一数据中心中不同机架上的节点
- dist(*/d1/r1/n1*, */d2/r3/n4*) = 6 (nodes in different datacenters)

Distance between two nodes 两个节点之间的距离

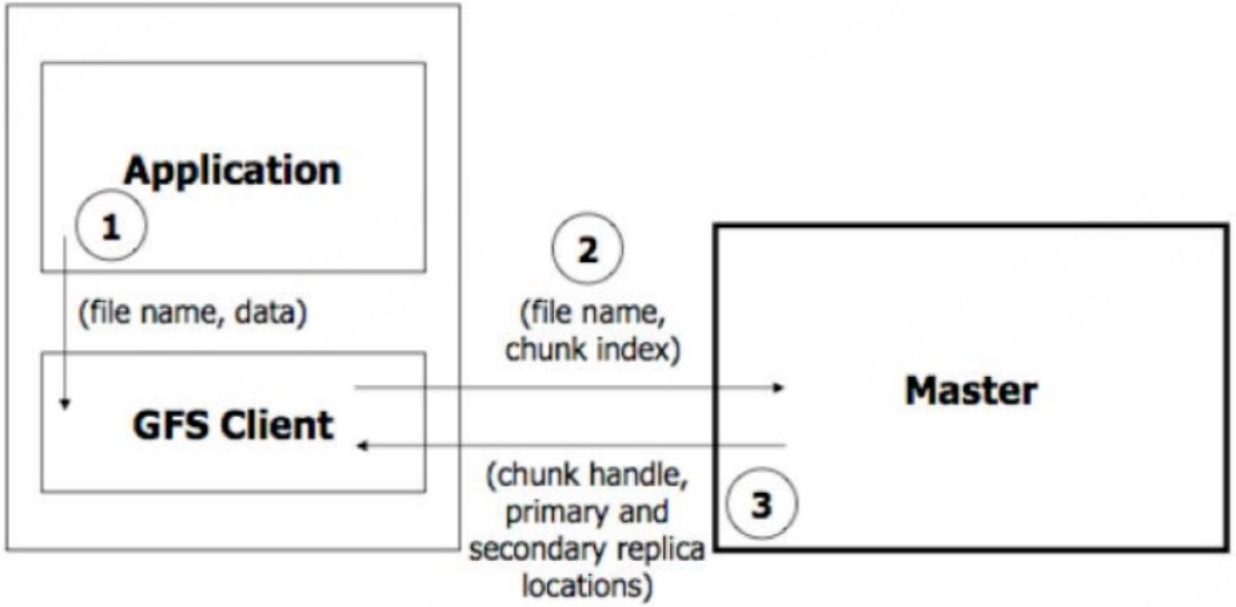


Source: T. White, “Hadoop: The Definitive Guide,” O’REILLY, 4th Eds., 2015.

File write 文件写入

1. Application originates the request 应用程序发起请求
2. GFS client translates request and sends it to master GFS 客户端转换请求并将其发送给主服务器
3. Master responds with chunk handle and replica locations

主服务器使用块句柄和副本位置进行响应



Block placement 块放置

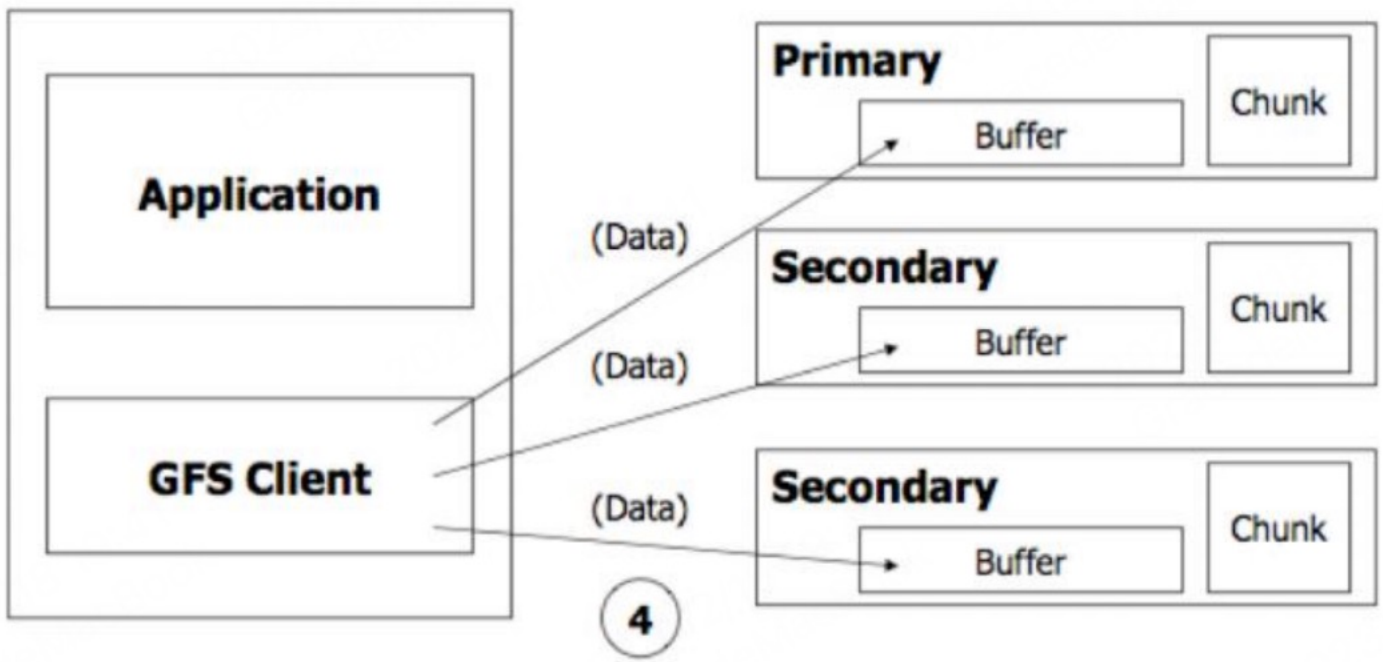
Current strategy in HDFS (replaceable with customized policy)

HDFS 中的当前策略(可用自定义策略替换)

- One replica on local node 本地节点上有一个副本
- 2nd and 3rd replica on two nodes of same remote rack 同一远程机架的两个节点上的第2和第3个副本
- Additional replicas are randomly placed 额外的副本随机放置

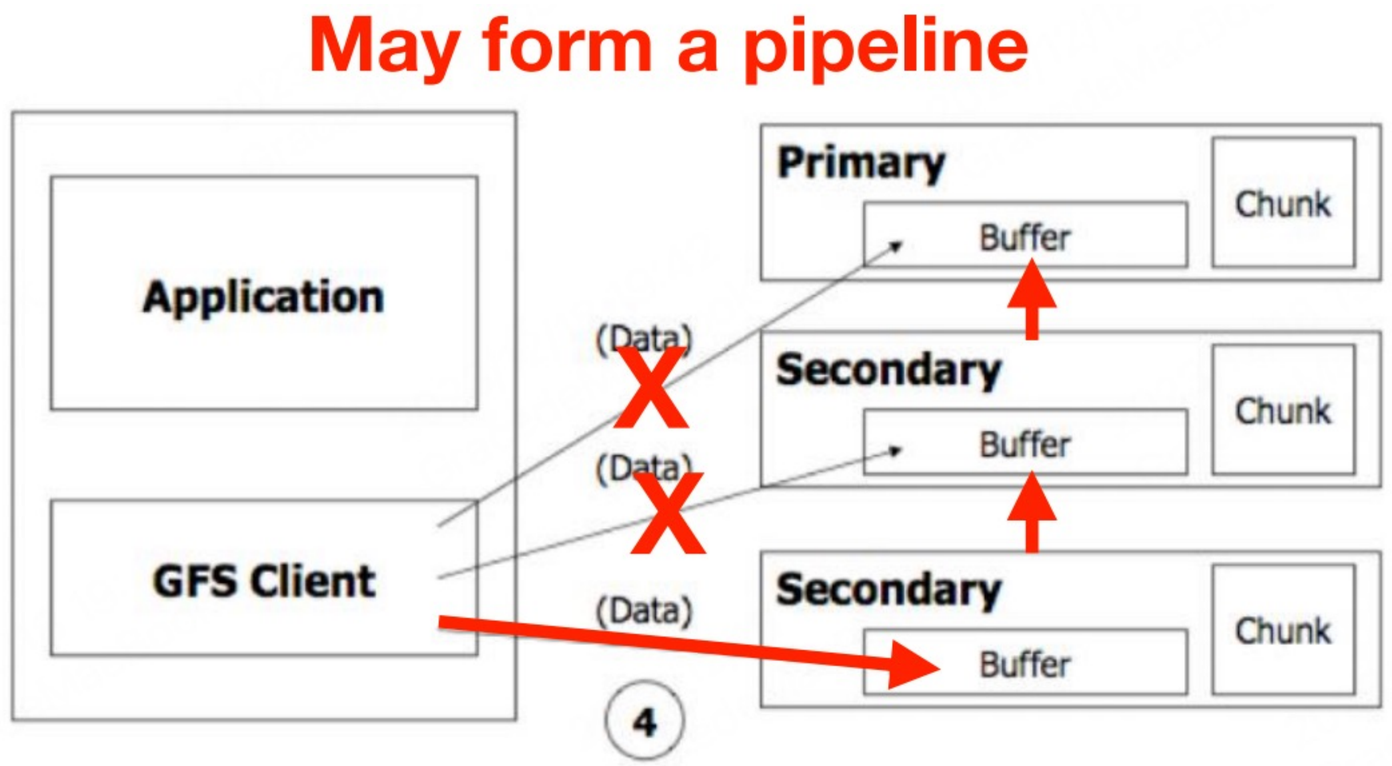
File write 文件写入

4. Client pushes write data to all locations. Data is stored in chunk server's internal buffers客户端将写入数据推送到所有位置。数据存储在块服务器的内部缓冲区中



File write

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

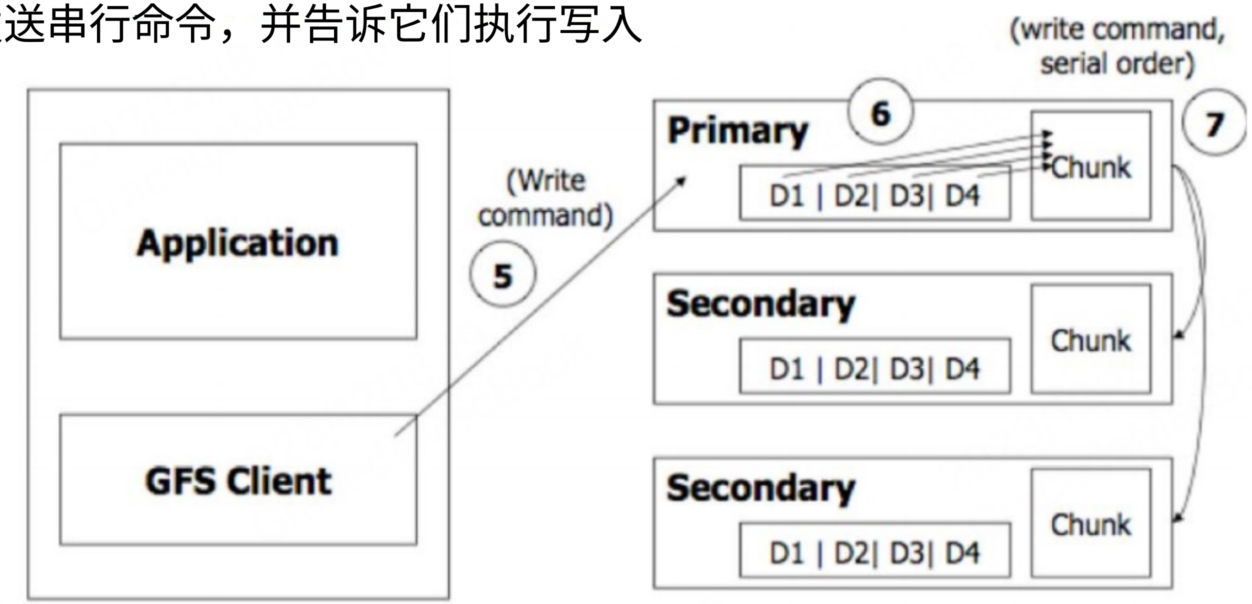


File write

5. Client sends write command to primary 客户端向主服务器发送写命令

6. **Primary determines serial order for data mutations** in its buffers and writes to the chunk in that order 主节点确定其缓冲区中数据变化的序列顺序，并按该顺序写入块

7. Primary sends the serial order to the secondaries and tells them to perform the write 主节点向从节点发送串行命令，并告诉它们执行写入



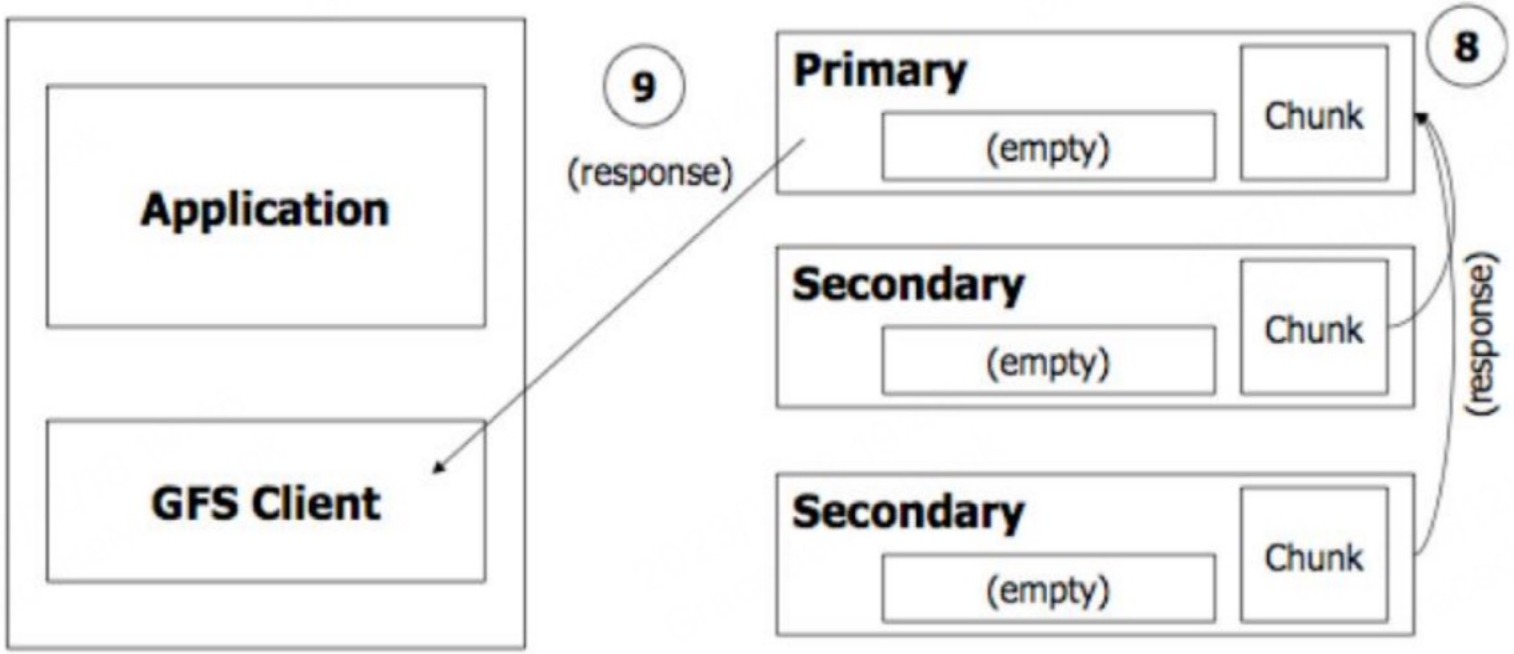
File write

次要节点对主要节点做出回应

8. Secondaries respond back to primary

9. Primary responds back to the client

Primary 向客户端做出回应



Fault Tolerance

容错

What if write fails?

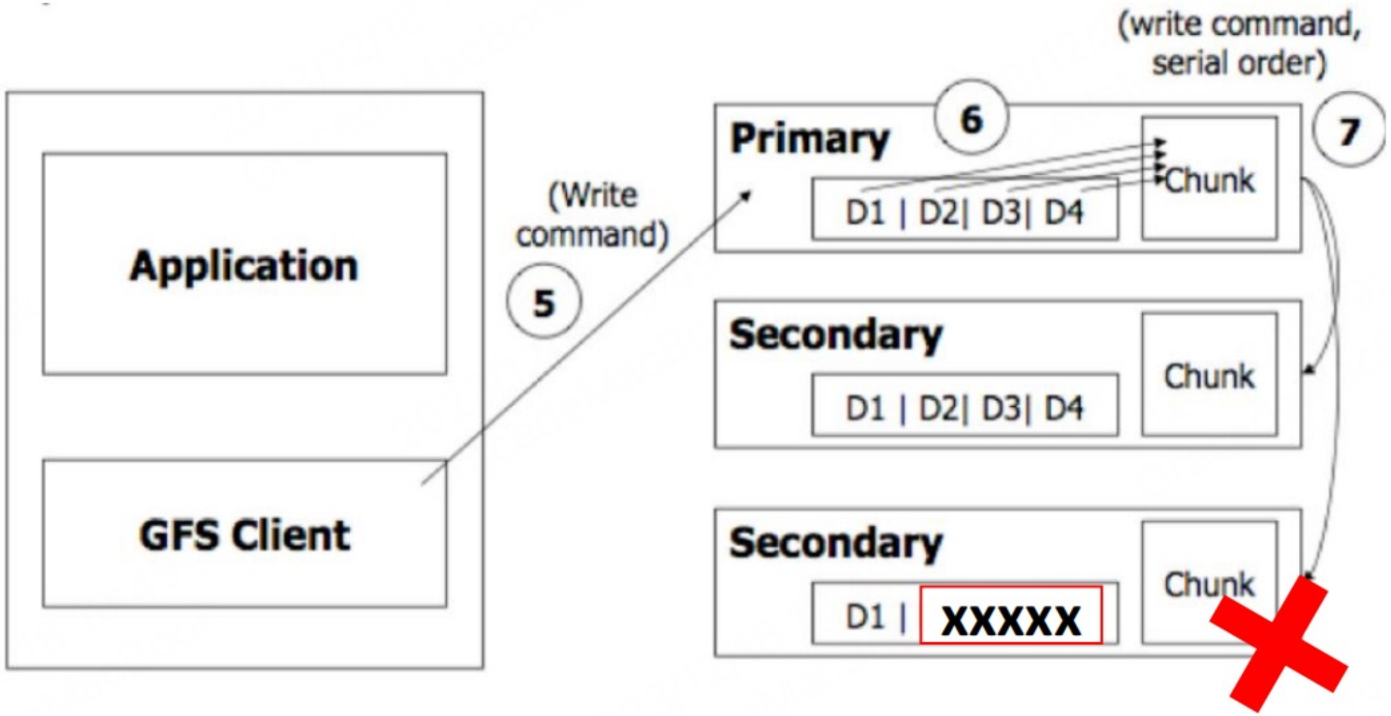
如果写入失败怎么办?

Rewrite 改写

直接向块服务器发出数据请求，例如cat

Issues data requests directly to chunkservers, e.g., cat

- Use checksum to skip inconsistent file regions 使用校验和跳过不一致的文件区域



What if chunkserver fails?

如果chunkserver 失败了怎么办?

Master 检测到一个Chunkserver 的 “ 心跳 ” 失败

Master detects a failed “heartbeat” of a chunkserver

Master decrements count of replicas for all chunks on dead chunkserver

Master 减少死机的Chunkserver 上所有Chunk 的副本数量

Master re-replicates chunks missing replicas elsewhere

主服务器在其他地方重新复制丢失的副本块

Summary of fault tolerance

容错性总结

High availability 高可用性

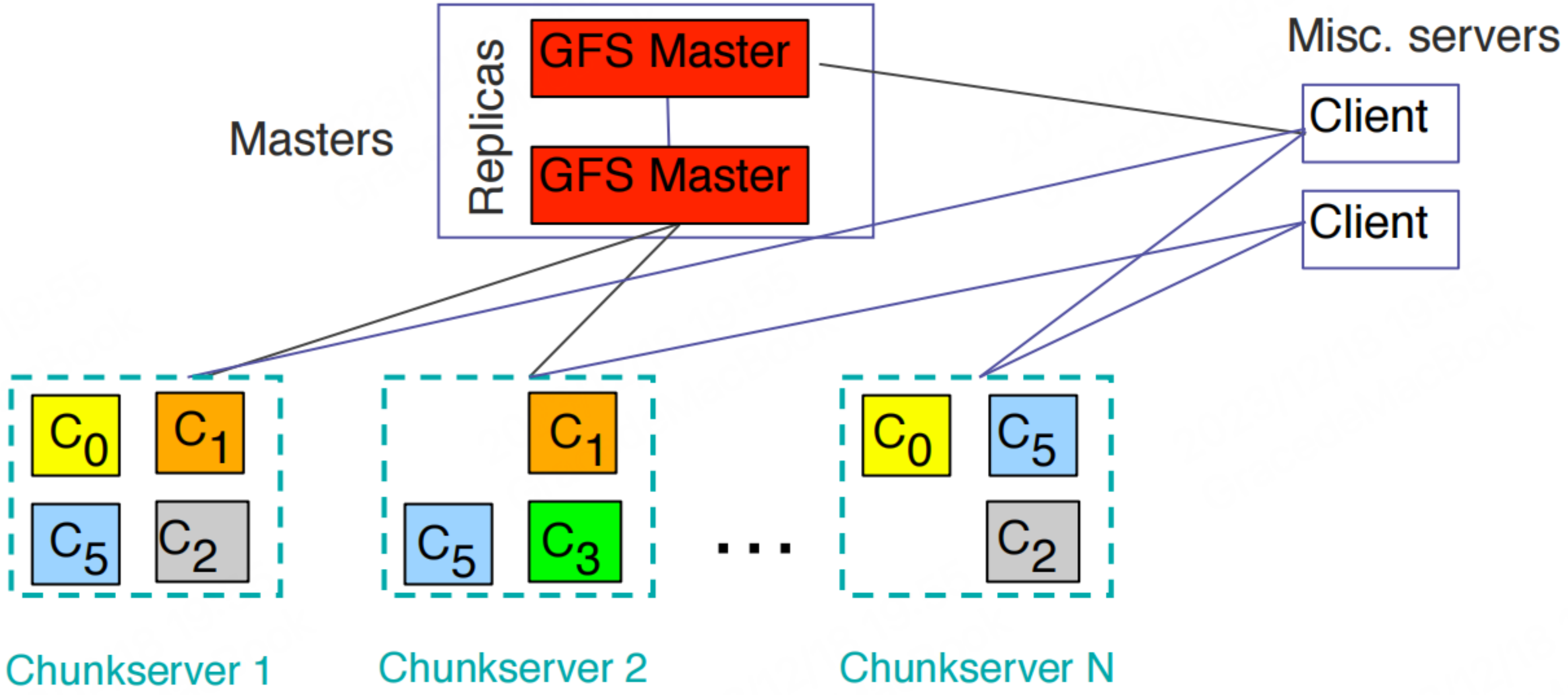
- Fast recovery 快速恢复 主服务器和块服务器可在几秒钟内重新启动
  - master and chunkservers restartable in a few second
- Chunk replication: 3 replicas by default 区块复制：默认3个副本
- Shadow master 影子主服务器

Data integrity 数据完整性

- Checksum every 64KB block in each chunk

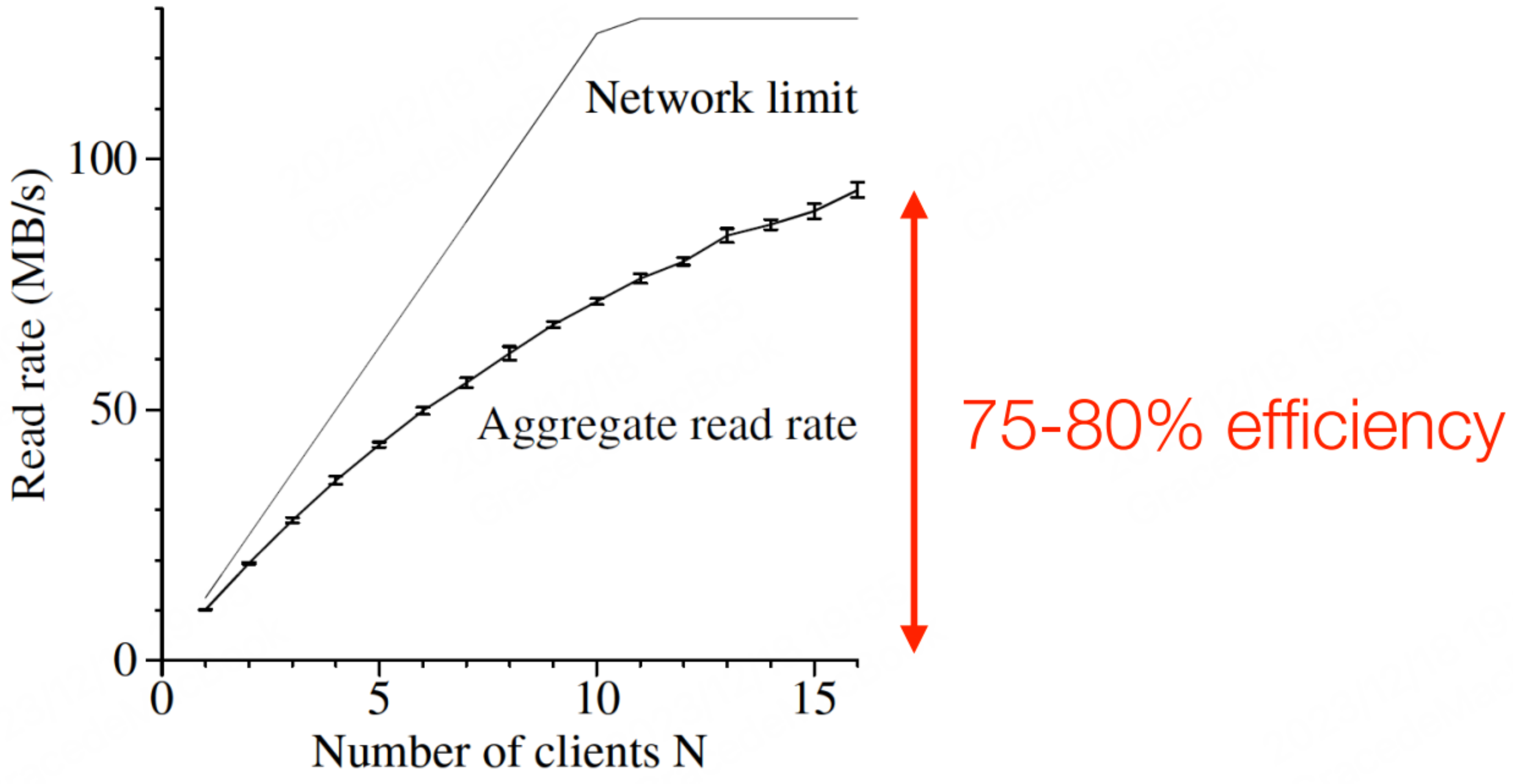
对每个块中的每个64KB 块进行校验

Summary of fault tolerance

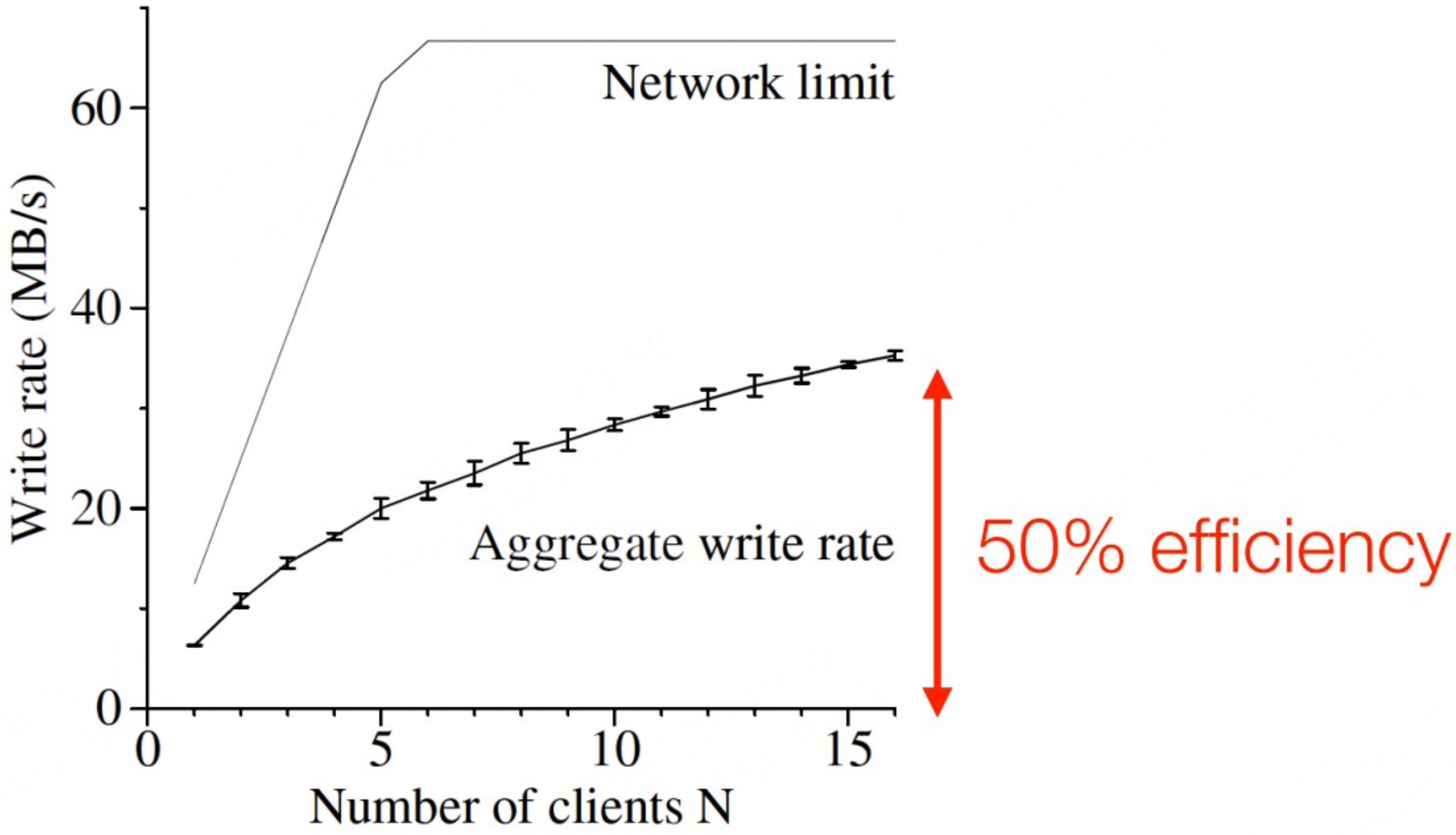


Measurements (2003)

Micro-benchmark: Reads



Micro-benchmark: Writes



Real cluster: Recovery time

Kill 1 chunkserver

23.2分钟内恢复了15,000个块(=600 GB数据)

- 15,000 chunks (= 600 GB of data) restored in 23.2 mins
- effective replication rate = 440 MB/s 有效复制率=440 MB/s

Kill 2 chunkservers

- 16,000 chunks on each (= 660 GB of data) 每个有16,000个块(=660 GB数据)
- reduced 266 chunks to having a single replica 将266个块减少为具有单个副本
- 266 chunks restored to at least 2x replications within 2 mins 60

266个块在2分钟内恢复到至少2x副本

Conclusions

GFS 演示如何在商用硬件上支持大规模处理工作负载

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

容忍频繁的组件故障(故障是常态，而不是例外)

- tolerates frequent component failures (failures as the norm rather than the exception)

针对大部分是附加然后依次读取的大文件进行优化

- optimizes for huge files that are mostly appended and then read sequentially

- delivers high aggregate throughput to many concurrent readers and writers

为许多并发读写提供高聚合吞吐量

Any limitations?

Limitations

限制

Assumes write-once, read-many workloads

承担一次写入、多次读取的工作负载

Assumes a modest number of large files

假设有一定数量的大文件

- **Large chunk size:** small files can create hot spots on chunkservers if many clients accessing the same file

大块大小：如果许多客户端访问同一个文件，小文件可能会在块服务器上产生热点

Credits

- Some slides adapted from course slides of COMP 4651 in HKUST
- Some slides adapted from course slides of DS5110 in UVA