**CS 677**: Big Data

# Research Papers

Lecture 4

# Reading Research Papers

- Research papers tend to not be the most riveting reading material
- They can be difficult to understand at times
- You might even feel that some papers are impossible to interpret correctly
  - …and you'd be right!

# Why so hard to understand?

- So, why aren't researchers better writers?
- Easiest answer: it's hard to write about these topics. They're complicated!
- Sometimes complexity is a "shield" against lazy reviewers
  - Reviewers are busy and would love to have a reason to reject your paper ASAP
- Funding, promotions, etc. are often tied to publications

# Some Advice

- It's okay to not understand a paper 100%
  - In some cases, it's nearly impossible unless you also get a copy of the writers' brains
- Many times, you have to use your best guess to determine how things actually worked
- Don't forget to search online. Maybe they published some slides or additional material you tap into

# As You Read

- Take note of things that are confusing
- Look for areas where details are left out
- Focus on uncovering insightful tidbits of information
- Think about the trade-offs being made and how you could tackle the same problem differently

# The Motivation for Doing This

- Why even bother with reading these?!
- If you are on the cutting edge of industry, you will still have to read papers (and maybe write them)
- Written communication and presentation is crucial for your careers
  - You will be amazed at how much time you spend writing docs and presenting your work
- I promise not to worry too much about the minor details (grammar, spelling). Just get the idea across!

# Reading Strategies

- Check out Keshav's "How to Read a Paper" on the schedule page
  - Proposes a 3-pass approach
- This is a good way to break the paper down
- Big Idea: don't read from start to finish

# Reading Steps

1. Figure out what the authors are trying to do
   - Read abstract, conclusions, section headings, and figure captions
     - Note any unknown jargon
2. Determine what components their system or approach has
   - Then figure out how the components interact. Sometimes it helps to draw a picture
3. Dive into the details
   - Ok, the paper uses algorithm X to provide its main contribution. How does the algorithm work?

# Let’s try this.

Let’s have a “reading break” so we can skim over the HDFS paper (if you haven’t read it already).

Then I’ll do a demo research presentation.

# Discussion

- What did you think?
- What new concepts/terminology was introduced?
- Can we fully grasp how the system works?
- What trade-offs are being made here?
- How would you change the design if you could take your own approach?

# Before we Start

- This is a “demo” of a research paper presentation.
- This is one approach. You don’t have to do it like this (but you definitely can!)

# Talk Outline

1. HDFS Background
2. System Design & Components
3. Benchmarks

# Talk Outline

1. **HDFS Background**
2. System Design & Components
3. Benchmarks

# History

- HDFS was created by Yahoo! in ~2006 and released under the Apache open source license
  - 25,000 nodes, 25 PB of data in ~2010
- Heavily inspired by Google's GFS
- Is the storage backbone for many legacy and modern big data processing frameworks
  - Higher-level abstractions can be built on top of HDFS. For example, HBase provides tabular storage and query support

# Hadoop

- Traditionally, HDFS was paired with ***Hadoop***, Yahoo!'s open source MapReduce implementaiton
  - Tight coupling between storage and computation
- HDFS can be used separately from Hadoop
  - And ***technically***, later versions of Hadoop evolved a bit from the old MapReduce model

# The Ecosystem

- Avro – serialization format
- HBase – Column-oriented storage
- Hive – Data warehouse
- Hadoop MapReduce – distributed computation framework
- Pig – dataflow language
- Zookeeper – Cluster management and coordination
- Spark – Iterative, in-memory processing
- Storm – streaming data processing

# Goals and Non-Goals

- Provide a distributed file system interface that is similar to standard POSIX file interface
  - (what's POSIX?)
  - Performance is more important than exact compatibility, though.
- Up front, HDFS does not:
  - Use RAID / striping mechanisms. Replicas provide fault tolerance
  - Distribute metadata; all metadata for files is stored on a single node.

# Relevance

- Hadoop + HDFS were used heavily up to about 2015 or so, but the computation side of things (Hadoop) has seen extensive evolution
  - Tools such as Spark have largely superseded Hadoop
- HDFS remains relevant today: used as a backbone to store large blobs of data for higher-level abstractions
- Alternatives:
  - Cassandra, HBase (slightly different data model)
  - Amazon S3 (and other cloud competitors)

# Talk Outline

1. HDFS Background
2. **System Design & Components**
3. Benchmarks

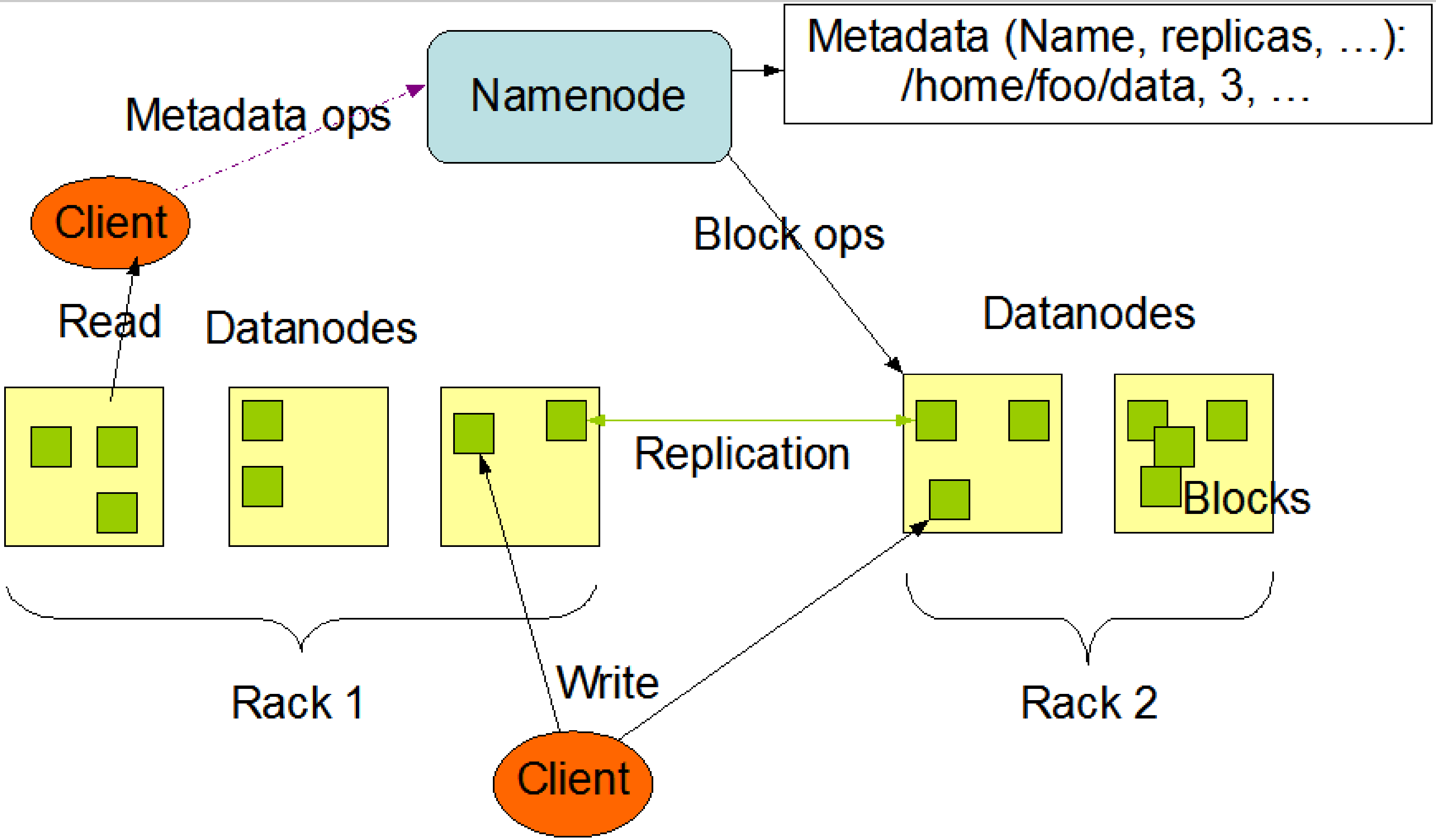
# Main Components

- NameNode
  - (and Secondary NameNode)
- File **blocks**
- DataNode
- Others:
  - CheckpointNode
  - BackupNode
  - Balancer

# Main Components: Our Focus

- We won't cover the Secondary NameNode, CheckpointNode, BackupNode, or Balancer.
- Fault tolerance for the NameNode has changed significantly from the publication of this paper
  - Has had some twists and turns over the years and not all the approaches worked well

# Architecture Diagram



# NameNode

- Manages a Namespace
  - Metadata: files, directories, permissions, quotas, etc.
  - Stored entirely in RAM
  - Maintains an on-disk ***journal*** of changes that can be replayed when the cluster restarts
- Main purpose: providing the file system hierarchy and a `file:node` mapping
  - Uses DataNode IDs, **not** host names / ports / etc
- Manages cluster health: nodes failing, replication, etc.

# Blocks

- Each file stored in HDFS is composed of one or more **blocks**
  - Block sizes are configurable (both as a default setting or on a per-file basis)
- Blocks are distributed and replicated across DataNodes
- Only appends are allowed: no in-place edits
  - Mirrors GFS' approach

# Accessing Blocks

- Blocks are **not** immediately available after storage
  - Heartbeat updates inform the NameNode of the new blocks
- During file retrievals or MapReduce jobs, replicas can stand in for the original file
  - Better data locality, more parallelism
- If an **append** operation is underway, the blocks can be locked to allow read-only access

# Managing Metadata

- Each block entry at the NameNode takes space; since the Namespace is an in-memory structure the NameNode must have lots of RAM
- If many small files are stored in the system (such as from the output of MapReduce jobs) index space is consumed rapidly
  - Solution: **HAR** file (Hadoop Archive) that bundles the small files into one large, indexed file
    - Kind of like a .zip

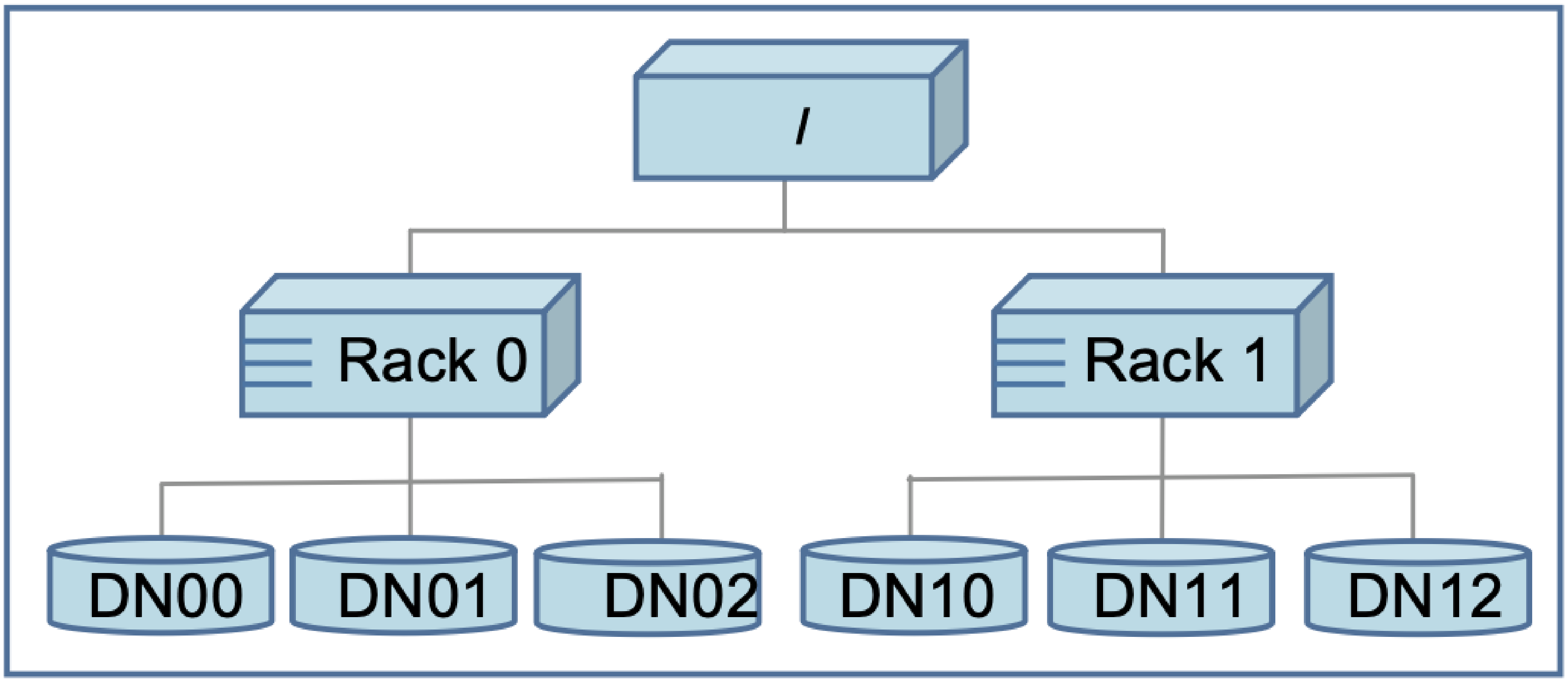
# Fault Tolerance

- Yahoo found that with three replicas, the probability of losing a block during one year is less than $0.005$.
- According to their tests, about $0.8$ percent of the nodes fail per month.
- With short heartbeat times, recovery is fast (and scales very well as the cluster expands)

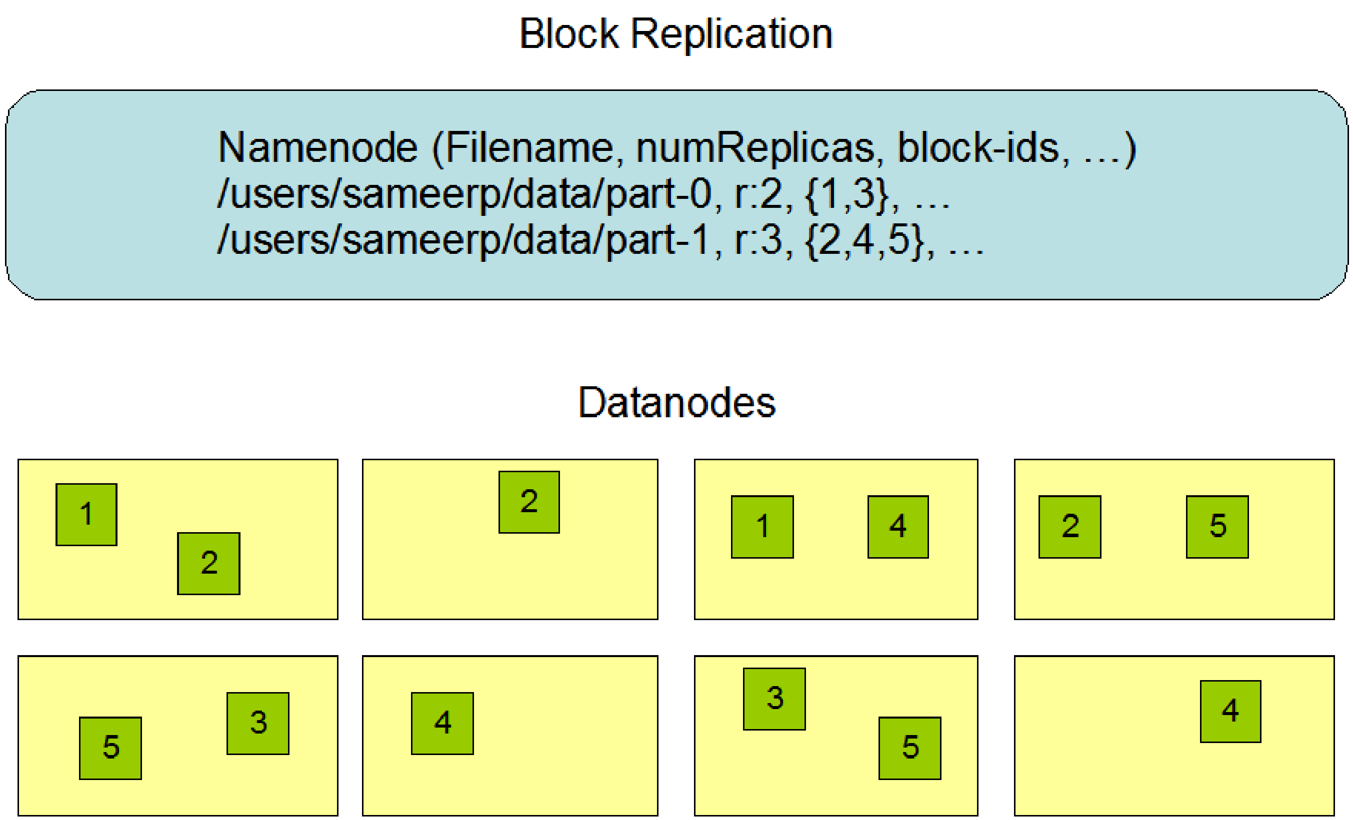
# Block Placement [1/2]

- HDFS is aware of “racks” and “datacenters”, allowing replicas to be geographically distributed
- First two replicas go to different racks
- Additional replicas are placed randomly
  - (but no two file replicas can be placed on the same physical machine!)

# Block Placement [2/2]



# Replication



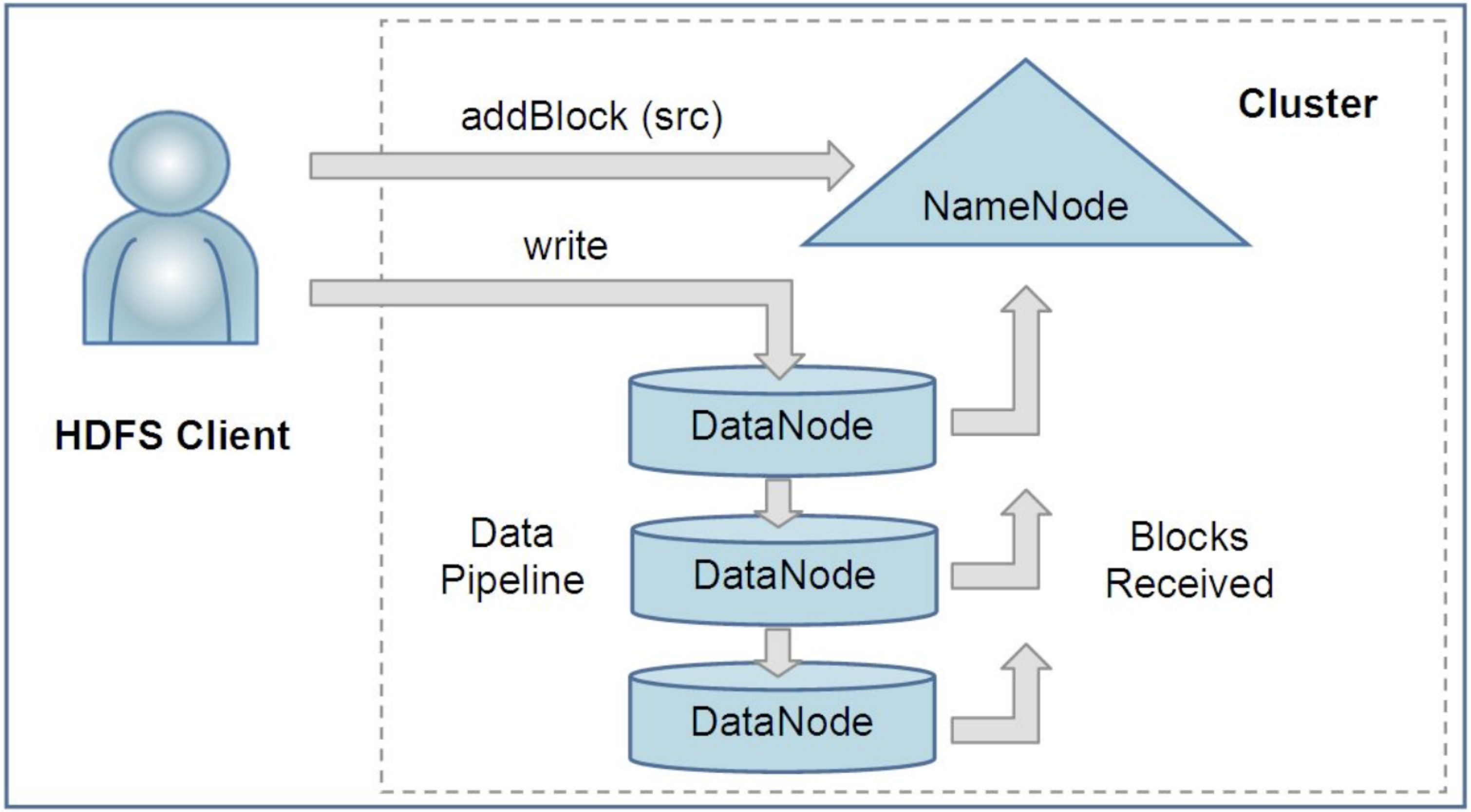
# DataNodes

- DataNodes are locked to a specific NameSpace ID
  - Restart the cluster with a new ID? DataNodes will not start up
    - Helps ensure data safety
- On startup, the node is assigned a NodeID
- Each block that a DataNode stores is represented as two files on the local host's (native) file system:
  - The data itself
    - No extra padding if the full block is not used
  - Metadata, including the block checksum

# Heartbeats

- Each DataNode sends a heartbeat every $3s$ (by default) to the NameNode to inform it of any file changes
  - Must be frequent or the system will take a long time to converge on a steady state
- If necessary, the NameNode will respond with instructions to replicate/remove blocks, shut down, or send a block report
  - In other words: DataNodes don't actively listen on a port for NameNode instructions

# Storage Flow [1/2]



# Storage Flow [2/2]

- Note that the NameNode receives **no** file data!
  - It does choose where the blocks go, though.
- The client only sends the blocks once. DataNodes handle pipelining to the others
- During the heartbeats, DataNodes will report the new blocks
- User can do an `hflush` operation to wait for all pending operations to be committed

# Snapshots

- HDFS supports creating a single ***snapshot*** of the current namespace state
- Produces duplicate metadata on the NameNode
- Produces duplicate files on the DataNodes
- Allows the cluster to ***roll back*** to a previous state but is **expensive**!

# Talk Outline

1. HDFS Background
2. System Design & Components
3. **Benchmarks**

# Test Setup

- 3500 node cluster
- 2 quad core Xeon processors @ 2.5ghz
- 16 GB RAM
- 4 directly attached SATA drives (one terabyte each)
- 1 gbps Ethernet

# Benchmark 1: I/O

- Used the **DFSIO** benchmark to measure IO speed per node
- “Empty” cluster:
  - DFSIO Read: 66 MB /s per node
  - DFSIO Write: 40 MB /s per node
- “Busy” cluster:
  - Busy Cluster Read: 1.02 MB/s per node
  - Busy Cluster Write: 1.09 MB/s per node

# Benchmark 2: Sorting

| Bytes (TB) | Nodes | Maps | Reduces | Time | HDFS I/0 Bytes/s | |
|---|---|---|---|---|---|---|
| | | | | | Aggregate (GB) | Per Node (MB) |
| 1 | 1460 | 8000 | 2700 | 62 s | 32 | 22.1 |
| 1000 | 3658 | 80 000 | 20 000 | 58 500 s | 34.2 | 9.35 |

# Benchmark 3: NameNode Performance

| Operation | Throughput (ops/s) |
|---|---|
| Open file for read | 126 100 |
| Create file | 5600 |
| Rename file | 8300 |
| Delete file | 20 700 |
| DataNode Heartbeat | 300 000 |
| Blocks report (blocks/s) | 639 700 |

# Conclusions [1/2]

- HDFS has enjoyed widespread use, and at this point is very solid/reliable
  - Also "boring"… but maybe in a good way?
- Has several well-documented weaknesses (the paper authors don't try to hide them)
  - Resource usage at the NameNode
  - NameNode failures
  - Handling small files
  - etc…

# Conclusions [2/2]

- There hasn't been a ***ton*** of development in this area, since it's a largely "solved" problem
  - HDFS (or its competitors) is good enough for most small or medium size organizations
- Most large organizations (Big Tech) have an in-house solution that usually supports:
  - Distributed namespaces (and failures)
  - Small files
  - Random access patterns, writes
  - Additional security measures

# One Last Thing

- You might be wondering… **why** break files into blocks anyway?
  - Sure, it spreads things out… but at a large enough organization, you'd have enough large files that things would gradually even out over time
- The **REAL** reason: it helps push the 'parallelizable portion' of our algorithm toward 100%
  - Your algorithm has to handle files that are split up… and that means it's *embarrassingly parallel* to process!