The Hadoop Distributed Filesystem: Balancing Portability and Performance

Jeffrey Shafer, Scott Rixner, Alan L. Cox – Rice University

Hadoop

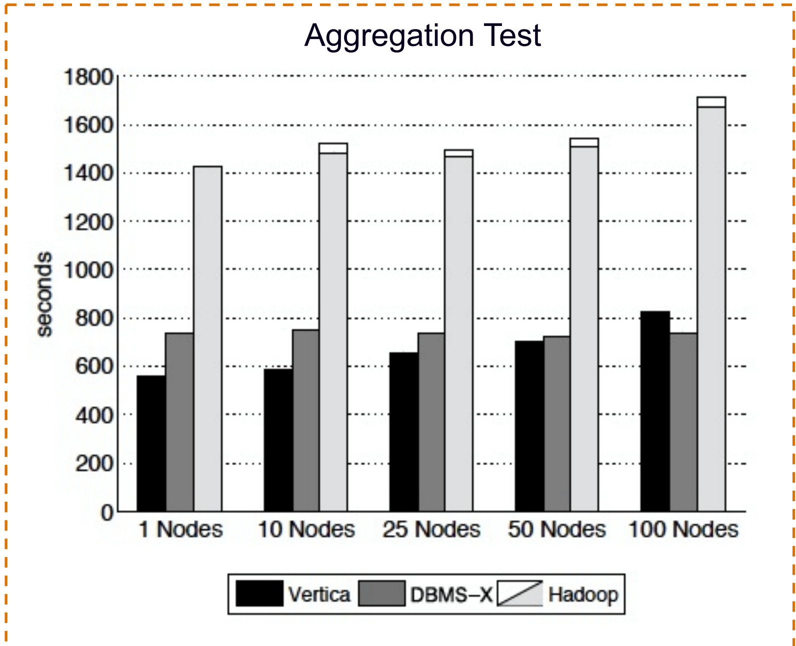
- Open source **MapReduce** framework
  - Scalable way to perform data-intensive computation on a commodity cluster computer
  - Inspired by Google's web indexing framework
- Designed for **portability**
  - Written in **Java**
  - Runs on Linux, FreeBSD, Solaris, OS/X, Windows, …
  - Uses native filesystems to store data: ext4, XFS, UFS2, NTFS, …
- In widespread use today
  - Amazon, Facebook, Microsoft Bing, Yahoo, …

Hadoop

- Large clusters are built from commodity hardware
  - x86 processors, SATA disks, Ethernet
  - Yahoo cluster
    - 4000 nodes (32000 total CPU cores)
    - 4 1TB disks per node (16PB total storage)
- Hadoop software ties the cluster together
  - Scheduling – Distribute jobs across cluster
  - Storage – User-level filesystem for applications
  - Reliability – Data replication, re-spawning failed jobs

Hadoop Performance – Slow?

- Widely publicized paper in 2009 compared Hadoop performance against parallel databases for similar workloads[1]
- Claim: Parallel databases are 2-3 times faster than MapReduce
  - *"The MapReduce model on multi-thousand node clusters is a brute force solution"*



(1) A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. Dewitt, S. Madden, and M. Stonebraker, "A Comparison of Approaches to Large-Scale Data Analysis," *SIGMOD 2009*

Ongoing Debate

- Debate in paper focuses on best high-level programming style
  - MapReduce or Parallel Database?
  - Assumption: High-level differences are causing the performance gap
- Different hypothesis
  - Performance gap caused by low-level Hadoop **implementation bottlenecks**
  - Data-intensive computing – Is Hadoop using the storage system efficiently?
- Today's talk:
  - Explore the low-level implementation of Hadoop
  - Analyze the interaction between Hadoop and storage
  - Fix performance bottlenecks

Outline

Hadoop Architecture

Hadoop Characterization

Hadoop Optimizations

Conclusions

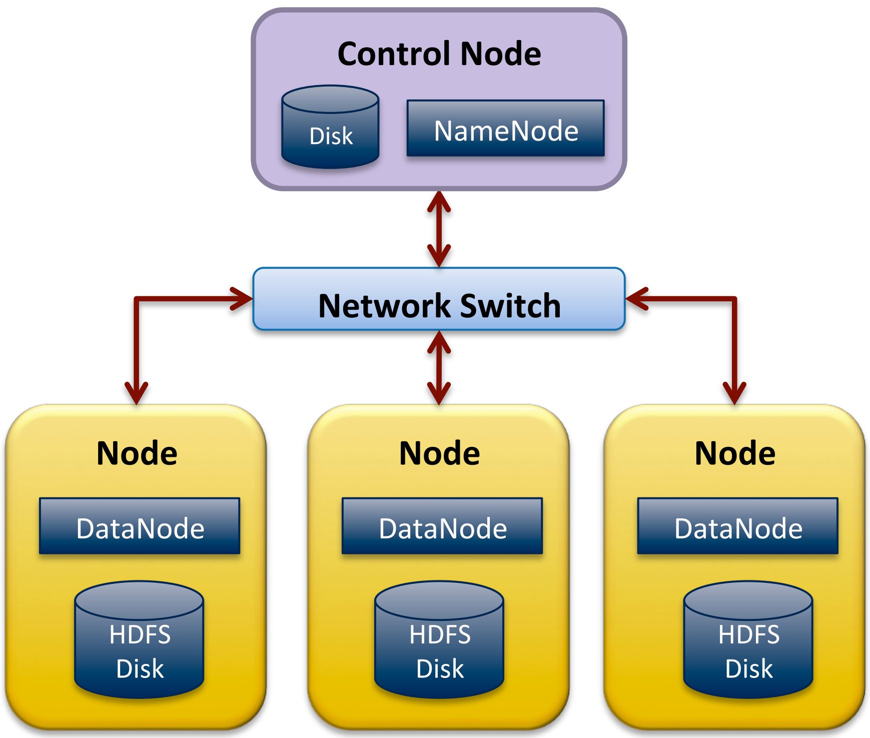
Hadoop Distributed Filesystem (HDFS)

- Global filesystem used by Hadoop applications
  - Clone of Google Filesystem (GFS) [3]
  - Any client can access any file anywhere in the cluster
  - Simple access semantics: Write-once, read-many
- Each (large) HDFS file composed of multiple 64MB blocks
  - Each block can be saved to any node in the cluster
  - Each block can be replicated to many nodes for redundancy
- Clients prefer to access data from local nodes (when given a choice)

(3) Ghemawat, S., Gobioff, H., and S. Leung, "The Google File System", SOSP 2003

Hadoop Distributed Filesystem (HDFS)

- NameNode
  - Stores filesystem namespace
  - Stores mapping from filename to HDFS block(s)
  - Coordinates allocation and replication
  - Single point of failure
- DataNode
  - Store HDFS blocks (64MB)
  - Each block is independent file in native filesystem



Hadoop Software Components

- Layering Hadoop on top of native OS produces a deep software stack
  - Hadoop applications – Access a 2TB file in HDFS…
  - Hadoop framework
  - HDFS global filesystem – Access many 64MB HDFS blocks…
  - Java virtual machine
  - Native operating system (e.g., Linux) – Access native file
  - Native filesystem (e.g., ext4) – Access many 16kB native blocks
  - Hardware (disks)
- How well does this work together?

Outline

Hadoop Architecture

Hadoop Characterization

Hadoop Optimizations

Conclusions

Search Benchmark

- Used many synthetic programs to characterize Hadoop
- Focus here on large **search** benchmark (i.e. distributed *grep*)
  - Simple to understand
  - Easy to show contributions
- Partition input data across all nodes in HDFS (10GB / node)
- Split search operation into **thousands** of map / reduce tasks
  - 1 task per HDFS block
  - Simplifies scheduling

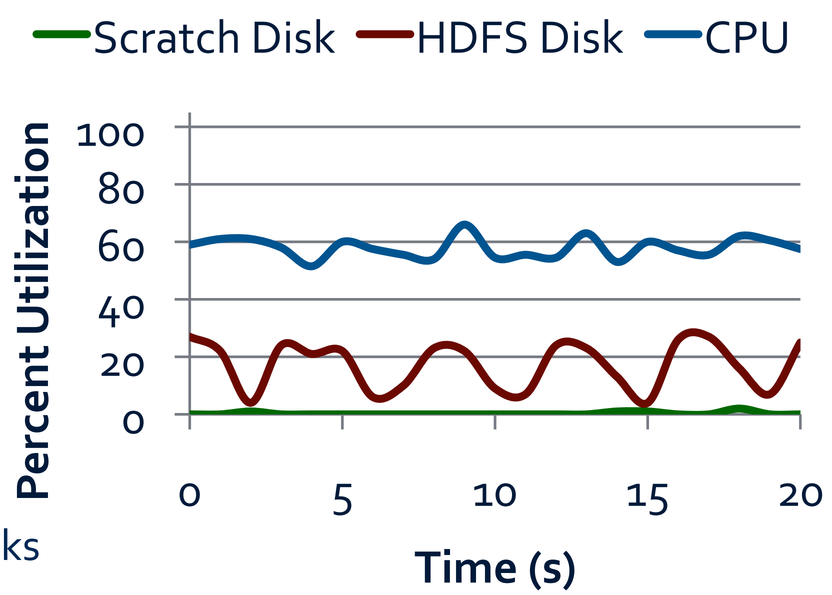
- Map phase (one task per node)
  - Read input data from HDFS (from local disk)
  - Inspect each value for match
    - If match, emit key/value pair for later
    - Excessive matches will spill from RAM to scratch disk
- Reduce phase
  - Pull data from map nodes for search matches
  - Write output data to HDFS (to local disk)

Search Benchmark

- **Desired** behavior
  - Disk bound, not CPU bound
  - Map task
    - Read data from HDFS disk continuously
    - Write matching values to scratch disk periodically
- What is the **actual** behavior of this test?
  - Average HDFS disk utilization: 30%
  - Average processor utilization: 60%
- Why so low?

Problem – Periodic Access

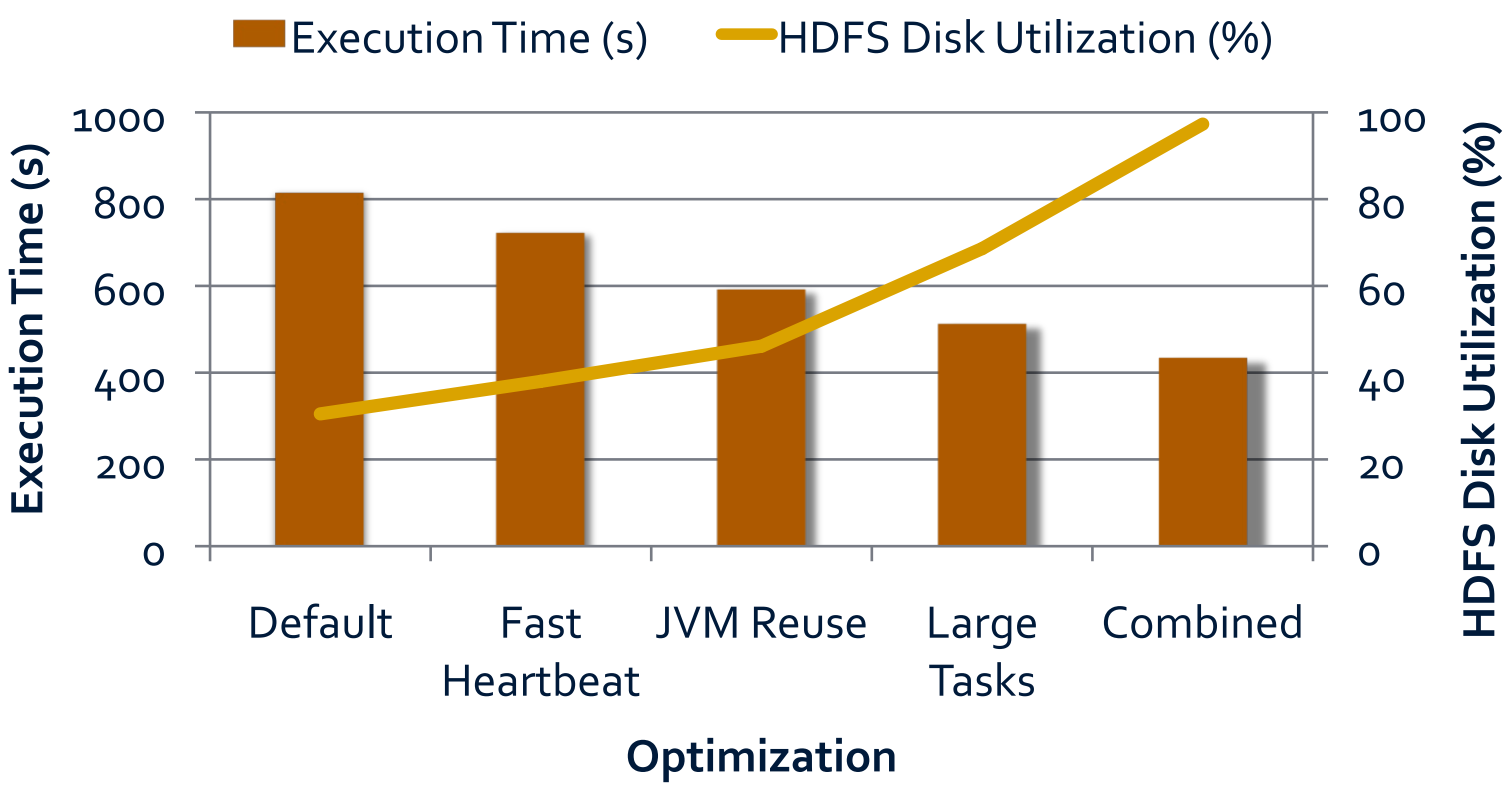
- Map phase of search benchmark
- Scratch disk rarely used
  - Search hits are rare
- Processor utilized continuously, but HDFS disk is not!
  - Periodic access pattern
- Cause of idle HDFS disk
  - Delay in issuing and starting new tasks
- Must start new tasks frequently
  - Each task only processes a single 64MB HDFS block (simplifies scheduling)



Fix – Accelerating Task Startup

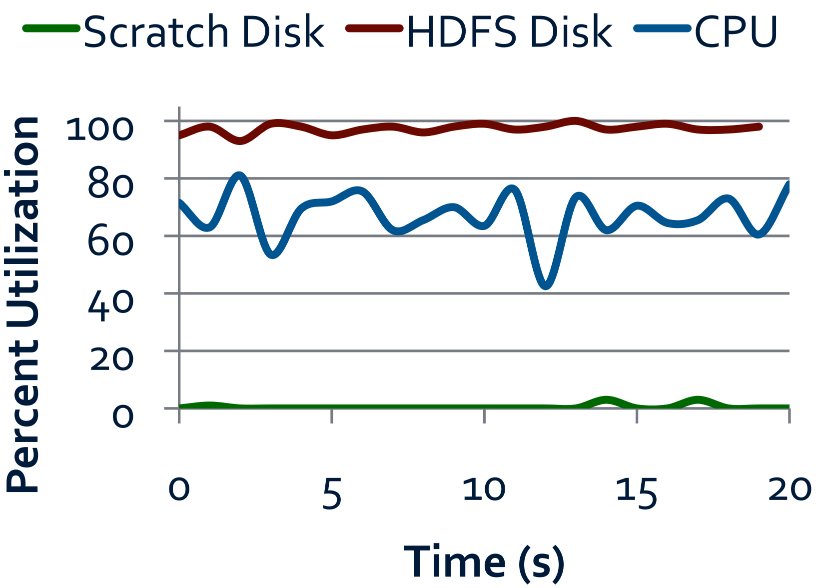
- Fast Heartbeat
  - Default: Clients send heartbeat every 3 seconds to report status + request new work
  - Change: Decrease interval to 0.3 seconds
- JVM Re-use
  - Default: Clients start new JVM for every task
  - Change: Re-use existing JVM
- Large Tasks
  - Default: Clients process 64MB of data per task
  - Change: Clients process 5GB of data per task

Fix – Accelerating Task Startup



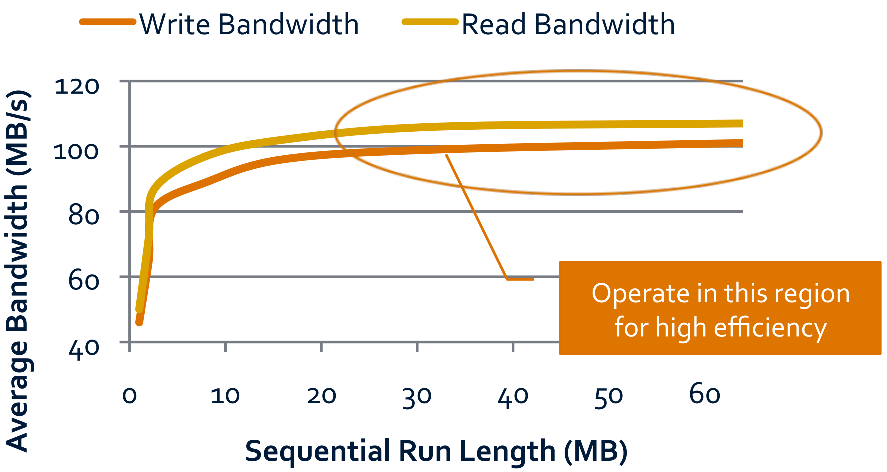
Search Benchmark

- Combine all optimizations together
- HDFS disk access is now streaming, not periodic
  - Higher CPU usage (for more bandwidth)
- Now we're using the disk continuously and heavily, but are we using it **efficiently?**



Spinning Disks

- Data-intensive computing clusters use hard drives
  - Flash memory (SSDs) are too expensive for bulk storage
- How do I use a spinning disk efficiently?
  - Minimize seeks
  - Large requests (streaming)



Hidden Dependencies

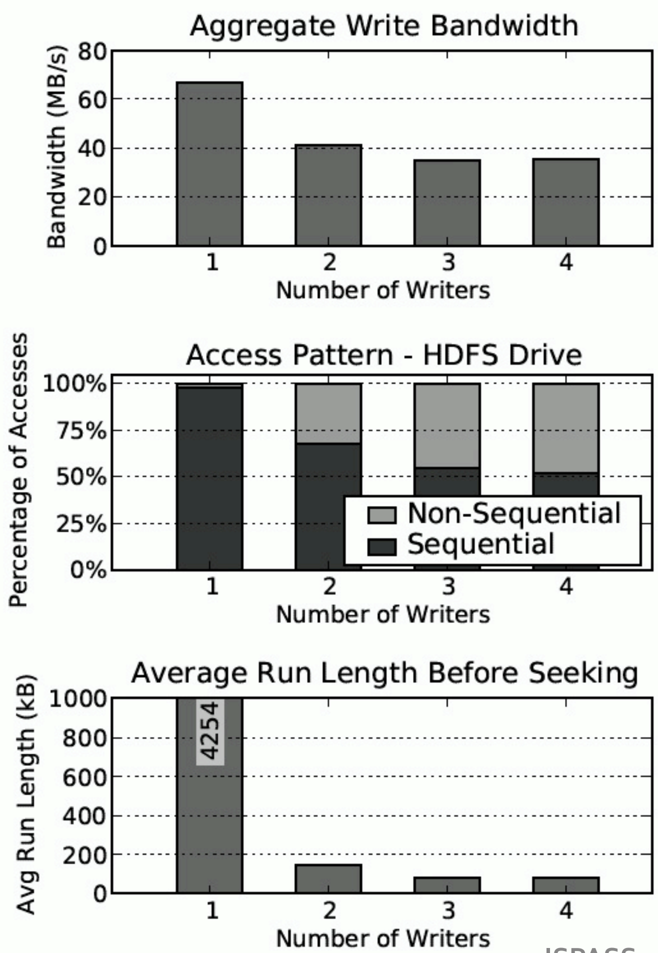
- Hadoop ***should*** be very “friendly” to spinning disks
  - HDFS uses large blocks (64MB) that can minimize seeks
  - HDFS uses streaming access patterns
- Hidden challenge
  - HDFS relies on the native OS disk scheduler and filesystem (Linux and ext4 or XFS, FreeBSD and UFS2, etc…)
- Native OS has control over
  - Disk allocation (affects fragmentation)
  - Disk scheduling (affects sharing between multiple clients)

Problem – Disk Scheduling

- Testing concurrent writers in Hadoop
  - 1-4 writers per node
  - Concurrent readers show similar behavior
  - Results from FreeBSD 7.2 / UFS2
  - Other OS / filesystems show similar behavior

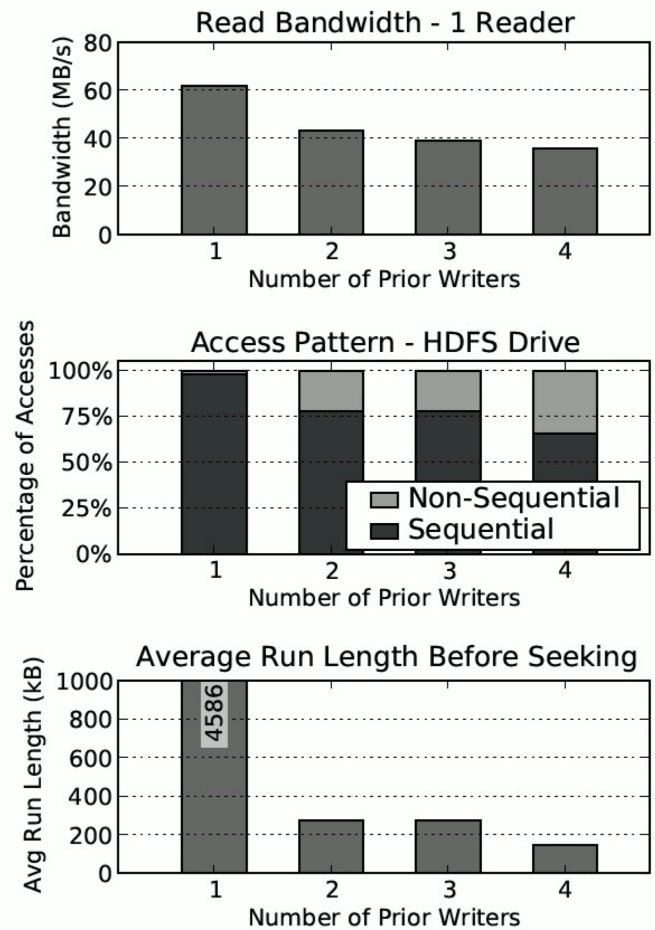
- As concurrent writers increase
  - Aggregate bandwidth drops
  - Random seeks become frequent
  - Run length plummets
  - Drive operates in inefficient region

- Big problem! Concurrent access is <u>common</u>
  - Replication
  - Multiple tasks over multiple CPU cores



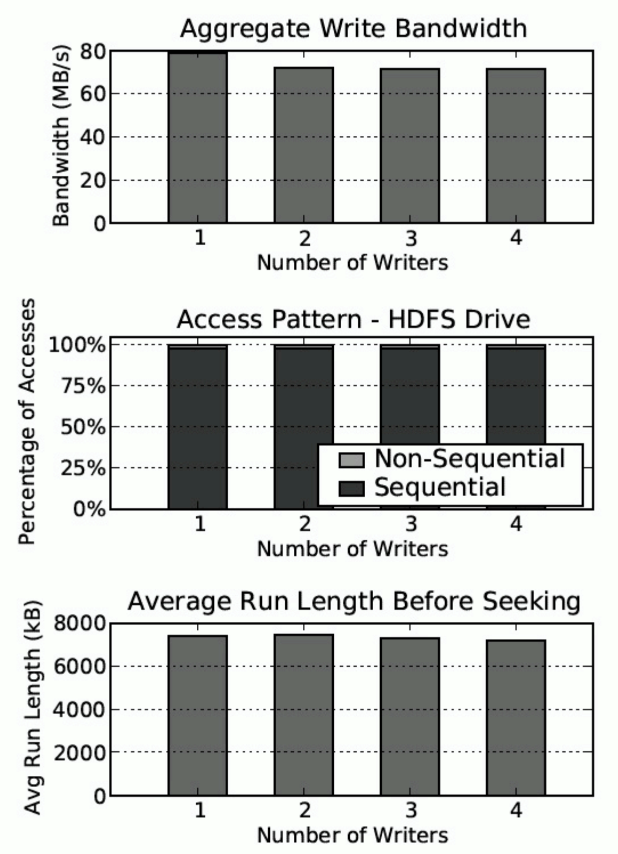
Problem – Fragmentation

- Minimal fragmentation when only 1 writer is using disk
- Fragmentation increases with multiple writers
- Poor placement decisions
  - Filesystem is only attempting to maintain small extents (128kB)
    - Fine for general purpose, but…
  - For Hadoop, we would like massive extents! (64MB)



Fix – HDFS-Level Scheduling

- Fix both problems by making HDFS smarter
  - Present requests to OS in the order we want them processed
- Buffer pending requests in memory and schedule them (per disk) at a large 64MB granularity
  - From perspective of OS, only one client is accessing each disk
- Benefits both disk scheduling (shown) and fragmentation (not shown)



Non-Portable Optimizations

- Chose HDFS-level scheduling to maintain portability
  - What if we didn't care about that goal?
- Reduce disk fragmentation
  - OS hints
    - *fallocate()* – Pre-allocate 64MB block in ext4 or XFS filesystem without immediately providing data. Linux-only
  - Only support certain filesystems
    - Custom configure filesystem to use large extents
- Reduce CPU overhead - Cache bypass
  - O_DIRECT to transfer data from disk to user-space buffer, bypassing cache
  - Not supported in Java (would need to use Java Native Interface)

Outline

Hadoop Architecture

Hadoop Characterization

Hadoop Optimizations

Conclusions

Hadoop Portability

- Classic notion of software portability
  - Does the application run on multiple platforms?
- Better (broader) notion of portability
  - Does the application perform well on multiple platforms?
- HDFS is (only) portable in the original sense
  - Its performance is highly dependent on the behavior of underlying software layers
  - Example: Concurrent access stresses OS disk scheduler / allocator, which was designed for general-purpose workloads

Conclusions

- Hadoop framework is complicated
  - Black-box design hides bottlenecks from user-level profiling
  - Example: Periodic hardware utilization
- Impact on current debate (Parallel Databases vs MapReduce)
  - Parallel databases are hard to tune – authors spent significant effort
  - If a similar effort had been expended on optimizing Hadoop, the performance “gap” would narrow significantly
- Hadoop architectural improvements
  - Task dispatching – increase resource utilization
  - HDFS-level scheduling – reduce disk seeks due to scheduling / fragmentation
    - Boost application performance
    - Improve node efficiency - More computation with the same hardware

Questions?

? ? ?