CSEP 524 – Parallel Computation University of Washington

Lecture 7: Big Data Analytics

Michael Ringenburg
Spring 2015

Projects !

- Project presentations start next week!
- Please submit slides via the course dropbox the night before your presentation, so I can load them all on to the same machine to speed transitions
- For those presenting next week (5/21) – there are only 3 of you, so I can be a bit more flexible. Please submit your slides by 4pm on 5/21.

Logistics

- Reminder: Next week's class moved to Thursday, 5/21
- Today:
  - First hour (6:30-7:30): Analytics Intro; Hadoop
  - Second "hour" (7:35-8:20): Discuss analytics performance
  - Third hour (8:20-9:20): Brad presents rest of chapel presentation
- Next Thursday: Analytics in Spark, 3 presentations
- Final two classes (5/26, 6/2): Presentations

Big Data

- As of 2012, 2.5 exabytes of data created *every day*
- Storage capacity doubling every 40 months
- Massive amount of data now exist/are being generated
- How can we understand/process/utilize this amount of data?
- Does it open new possibilities? New paradigms? New challenges?

The ~~3~~ ~~4~~ 5 V's of Big Data

- Challenges of big data typically lie in one (or more) of the following *V*'s
  - **Volume**: very large amount of data
  - **Velocity**: data coming in very rapidly
  - **Variety**: many different types of data
- Two additional V's are sometimes added:
  - **Veracity**: Large variance in the quality of the data/ difficulty in determining quality
  - **Variability**: Inconsistency in the data

Big Data Examples

- **Large Hadron Collider**: 600 million particle collisions per second
- **Twitter**: 500 million tweets per day
- **Cybersecurity**: Analyzing network/file/other log data – potentially high velocity, high volume
- **Banking**: Credit card fraud detection
- **Real estate**: Windermere using 100 million GPS trackers to estimate commute times for new home buyers

Big Data Examples

- **Social Media**: 50 billion Facebook photos per day
- **Bioinformatics/medecine**: Massive amounts of genomic data to analyze; patient treatment outcomes, etc
- **Financial Trading**: Analyzing stock/bond/option transactions; determining compliance with regulations; new trading algorithms
- **Medicare fraud detection**: analyzing medical billing records

Data Analytics

- Gleaning information from (typically "big") data sets
  - Summarizing large data sets
  - Finding patterns
  - Looking for anomalies
  - Developing new models/theories
- Data scientists: experts in data analytics. Typically combine backgrounds in:
  - Math/Statistics
  - Computer Science
  - Often domain-specific knowledge

Analytics: A New Application of Parallel Computing?

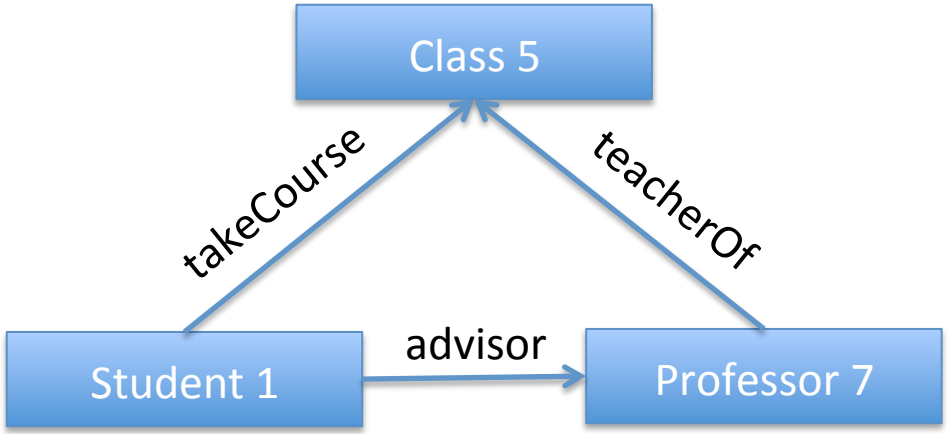
Big Data Analytics Challenge	Parallel Computing Solution
High volume of data	Many nodes = large amount of memory to store data (DRAM and disks)
High velocity of data – need to keep up with rapid streams of new data	Many processors/nodes/threads to handle incoming data. Can scale up as velocity increases. Can dedicate some nodes solely to handling incoming data streams, leaving others for more complex processing.
Extracting knowledge from large quantities of data	Large datasets tend to scale up well to large numbers of processes. Parallel versions of many common machine learning/statistical algorithms are well studied
Others?	**Your thoughts?**

Graph Analytics

- Many datasets can be treated as graphs (source vertex [attribute] – edge [attribute] – destination vertex [attribute])
  - Social networks: edges for followers, friends, likes, retweets, etc
  - Graph databases: subject-predicate-object
  - Network connections
- Especially valuable for finding unknown relationships
  - Apply graph algorithms to data, find hidden communities, connections, etc.

Graph Analytics

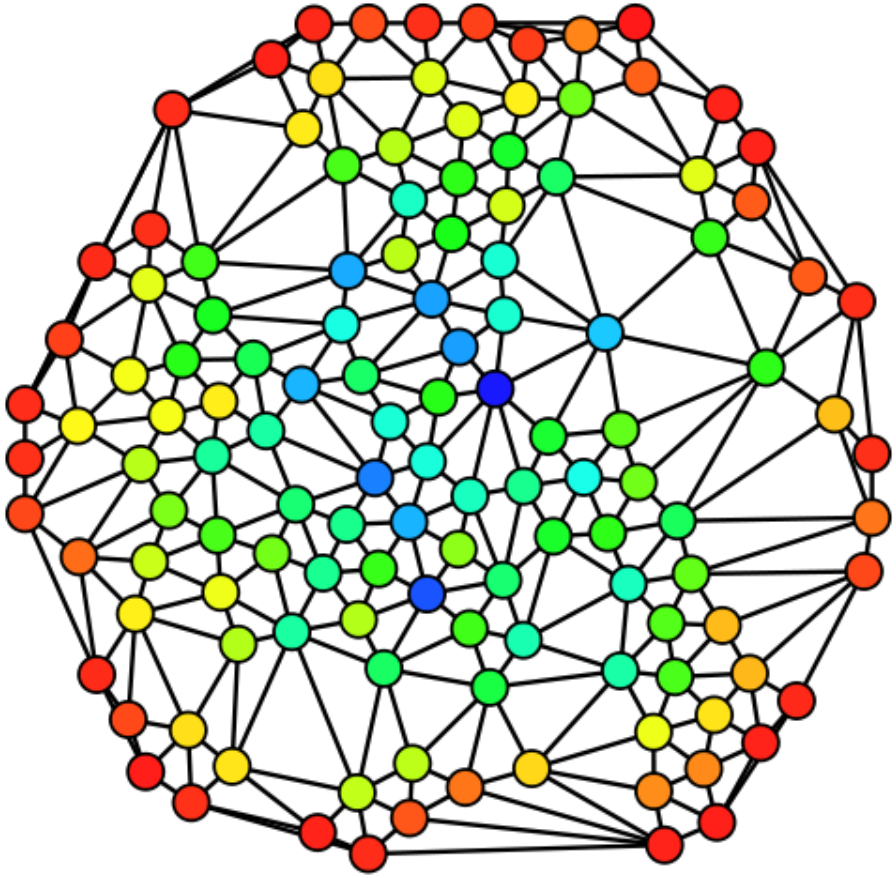
- One approach: query a graph database
- This query searches for a triangular relationship – students who takes classes taught by their advisor.
- Graph queries generally search for "graph patterns" like this. In relationship databases, queries like this require expensive joins of multiple tables.




```
SELECT ?X, ?Y, ?Z WHERE
{ ?X rdf:type ub:Student .
  ?Y rdf:type ub:Faculty .
  ?Z rdf:type ub:Course .
  ?X ub:advisor ?Y .
  ?Y ub:teacherOf ?Z .
  ?X ub:takesCourse ?Z}
```

Graph Analytics

- Other approach: apply graph algorithms
  - Betweenness centrality
  - Community Detection
  - BadRank
  - Triangle Counting
  - Connected Components
  - Shortest path
  - Clustering

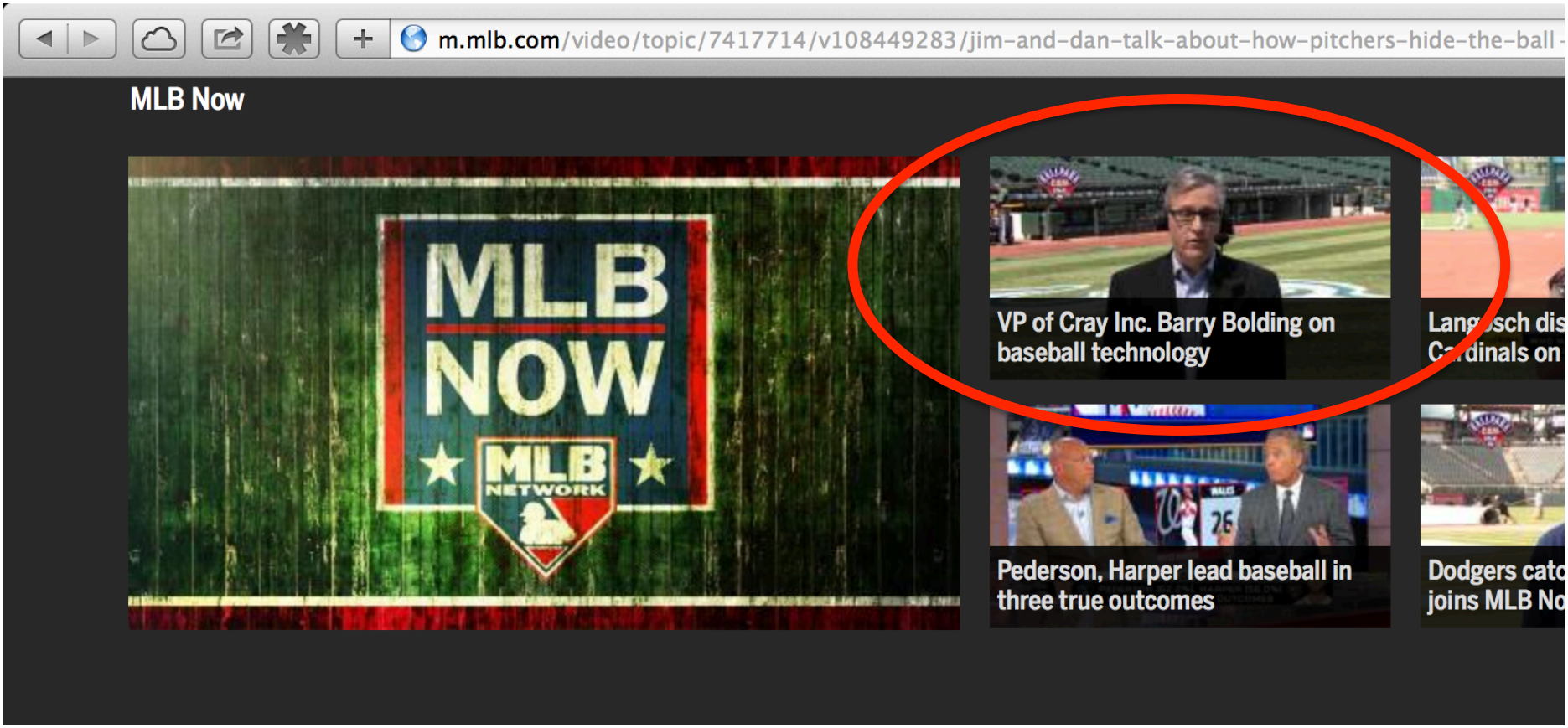


Graph Example: Twitter Analysis

- Discover twitter communities of users with similar interests
  - Find influencers (most retweeted), rebroadcasters (most retweets), connectors between communities
- Construct graph of users who interact (e.g., mentions)
- Run community detection algorithm
- Label communities based on most common topics (#hashtags)
- Roles based on retweet and retweeted count, membership in multiple communities

And Professional Sports!

- A Major League Baseball team is using a Cray graph analytics appliance



Analytics: Hardware Environments

- Typical HPC framework assumes:
  - Loosely connected cluster (e.g., 1 Gigabit Ethernet)
  - Cheap/unreliable hardware
  - Often, many "spindles" per node (local harddrives) – i.e., high bandwidth to local storage, but also high latency. Seeing more SSD-based solutions, however, in higher-end market.
- Frameworks optimized, configured, designed for this environment.

Analytics vs. Traditional Parallel Programming

Traditional Parallel Programming/HPC	New Analytics Frameworks
Lower level: C, C++, Fortran	Higher level: Java, Python, Scala, etc
APIs or pragmas to manage parallelism	Parallelism automatically managed by the framework
Difficult for less-experienced programmers to achieve performance/scalability	Less experienced programmers can achieve reasonable performance/scalability
Very experienced programmers can achieve very high performance scalability	Experience helps, but peak performance still tends to be significantly worse than hand-coded HPC algorithms
Architectural optimization up to user	Optimized for typical analytics HW
Your thoughts…	Your thoughts…

Are new high-productivity HPC languages (e.g., Chapel, X10) a middle ground?

Analytics vs. Traditional Parallel Programming

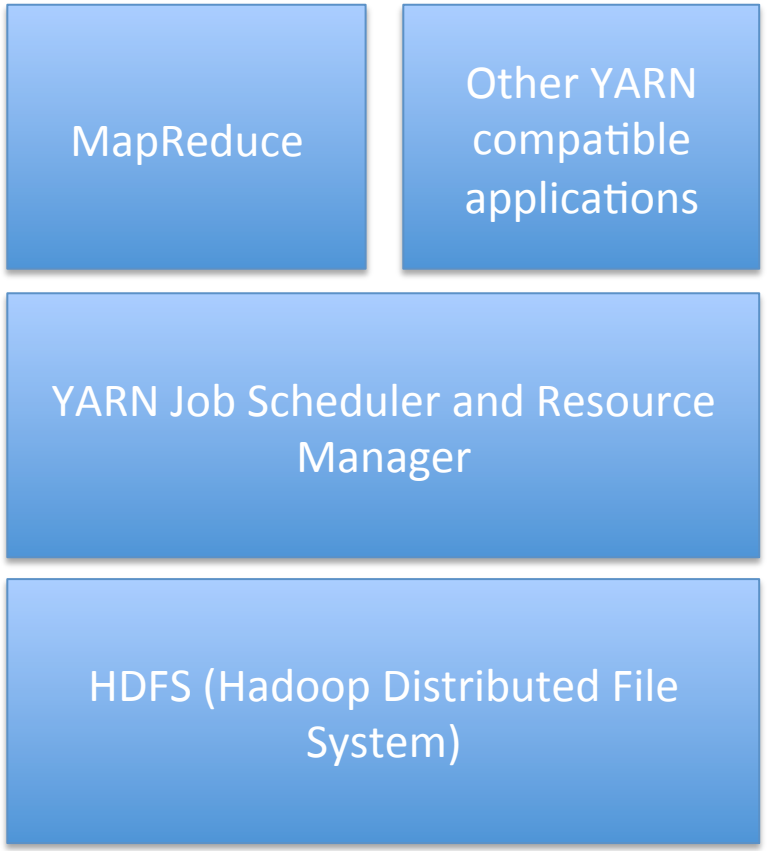
- High level: Analytics frameworks (Hadoop, Spark, Flink) tend to be much easier to write correct and scalable code, but currently best performance is limited by:
  - Framework overheads
  - Language overheads (e.g., garbage collection: overhead + stragglers…)
  - "One-size-fits-all" (making framework work across many types of platform and/or for many kinds of job)
  - Very coarse-grained synchronization
- But these frameworks are improving quickly (e.g., Spark Tungston project)
  - Low hanging fruit?
  - Great deal of interest in research and industry

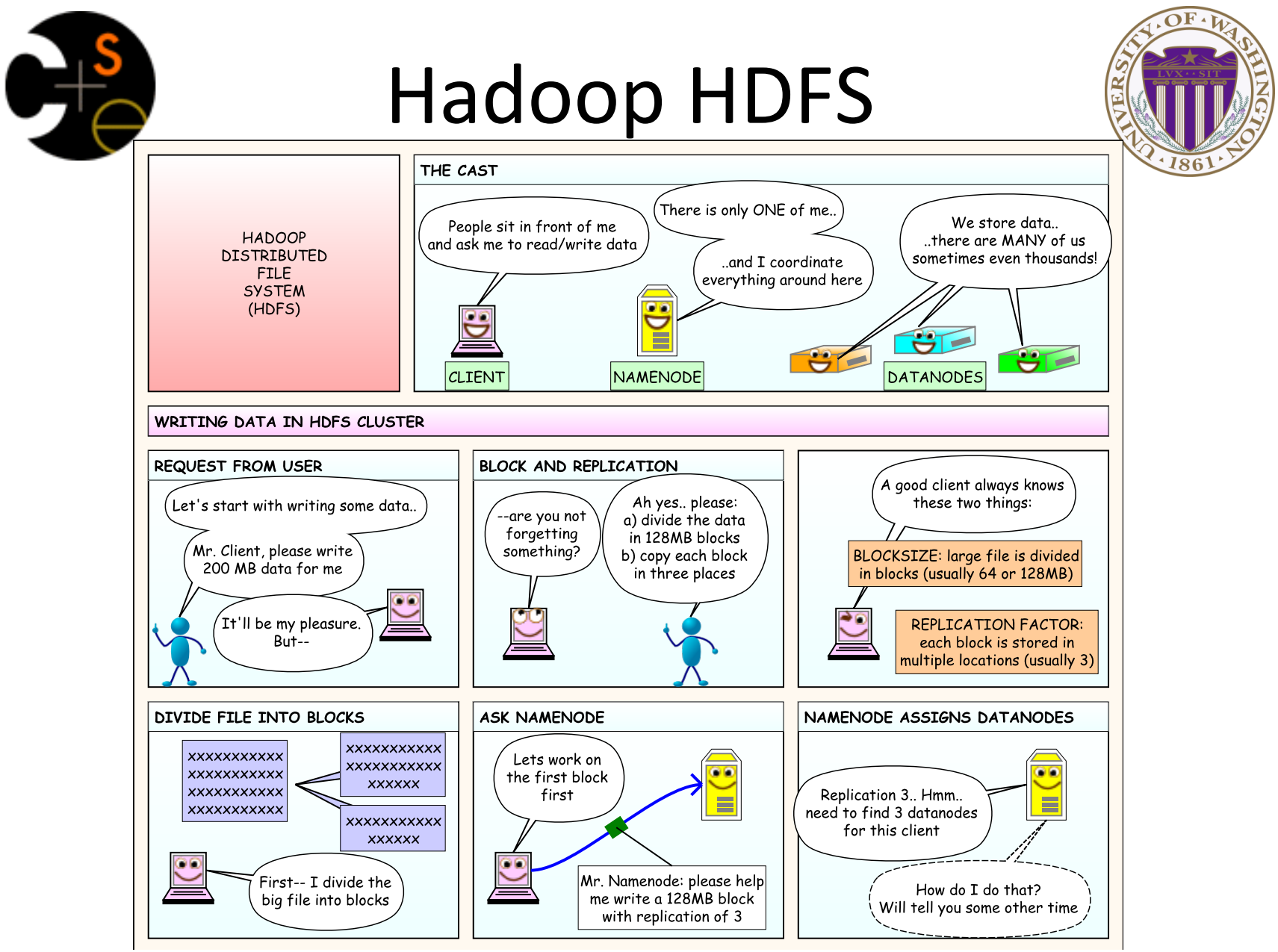
Analytics: Time To Solution

- Many analytics applications care more about time-to-initial-results vs. fastest possible performance
- For this case, frameworks that handle a lot of the low-level details are ideal
- Other cases demand the capabilities of HPC-oriented solutions
- Examples:
  - Weather forecasting: want to maximize performance/ enable running highest resolution model given forecast time constraints
  - Social network analysis to inform product marketing: Want something that we can run overnight *as soon as possible*

Hadoop Framework

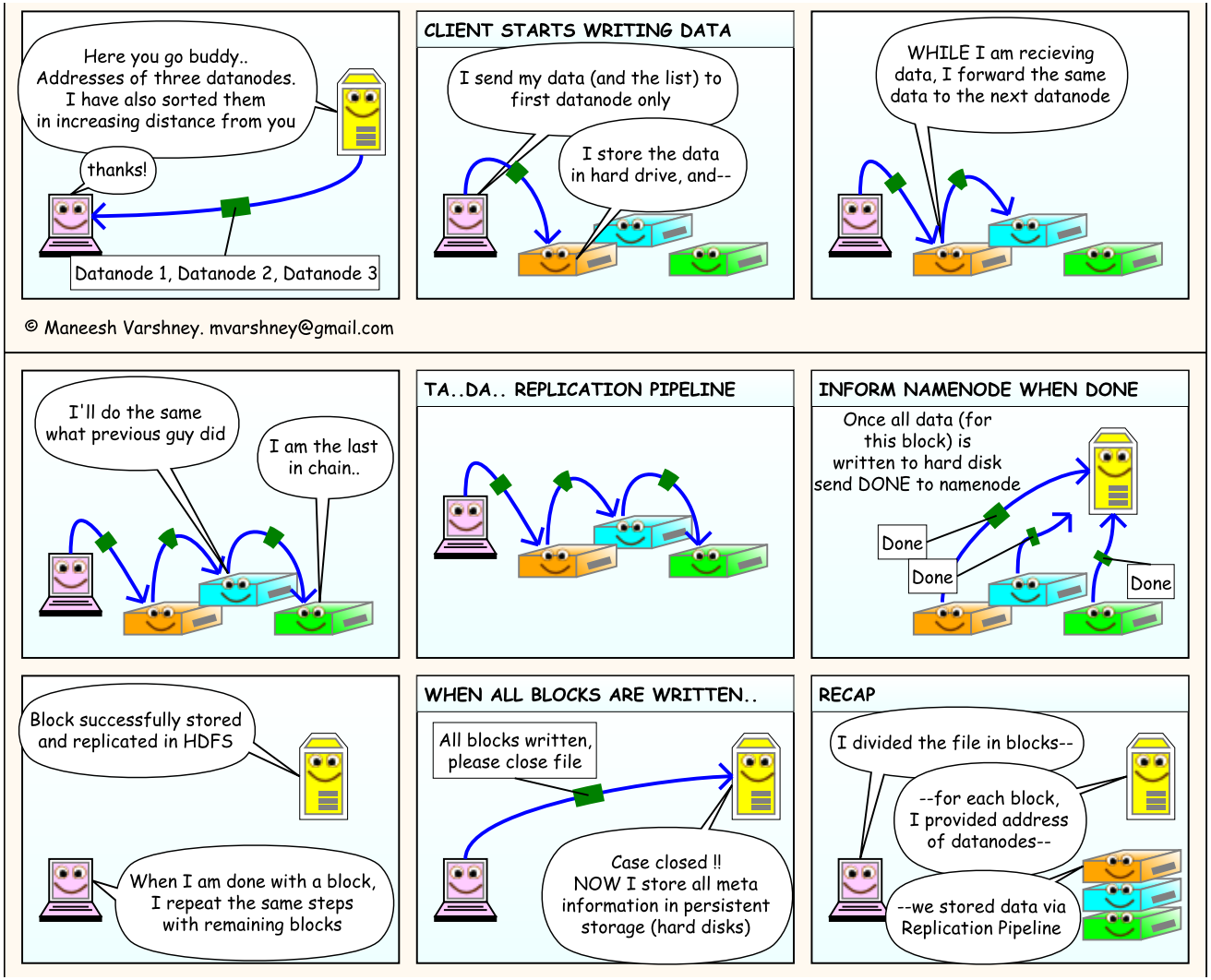
- HDFS: distributed file system, uses replication for fault tolerance
- YARN: assigns cluster resources (e.g., memory, cores) to jobs, schedules execution
- MapReduce: Application framework – maps and reduces, based on Jeff Dean/Google paper we read.
- Can run other frameworks on top of Yarn (e.g., Spark over Yarn, Giraph graph processing)



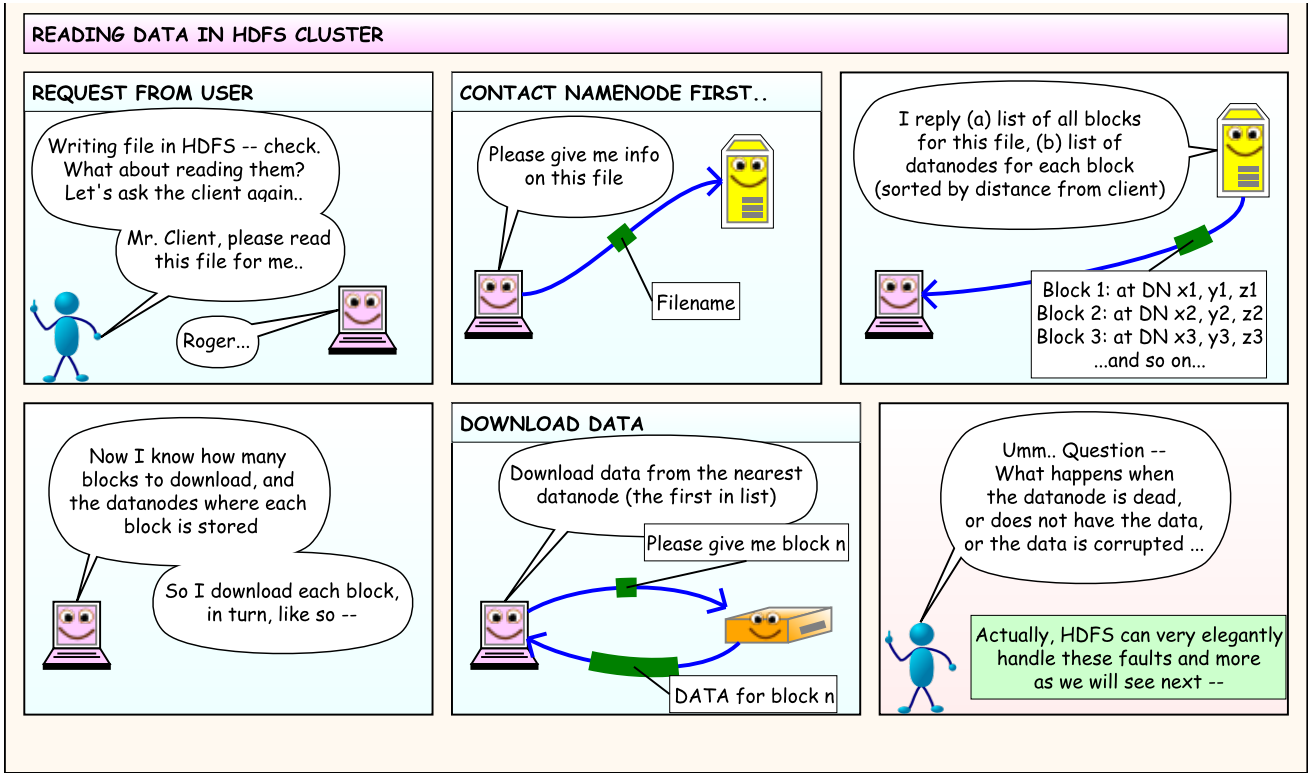

HADOOP DISTRIBUTED FILE SYSTEM (HDFS)
THE CAST
People sit in front of me and ask me to read/write data
There is only ONE of me..
..and I coordinate everything around here
We store data.. ..there are MANY of us sometimes even thousands!
CLIENT
NAMENODE
DATANODES
WRITING DATA IN HDFS CLUSTER
REQUEST FROM USER
Let's start with writing some data..
Mr. Client, please write 200 MB data for me
It'll be my pleasure. But--
BLOCK AND REPLICATION
--are you not forgetting something?
Ah yes.. please: a) divide the data in 128MB blocks b) copy each block in three places
A good client always knows these two things:
BLOCKSIZE: large file is divided in blocks (usually 64 or 128MB)
REPLICATION FACTOR: each block is stored in multiple locations (usually 3)
DIVIDE FILE INTO BLOCKS
First-- I divide the big file into blocks
ASK NAMENODE
Lets work on the first block first
Mr. Namenode: please help me write a 128MB block with replication of 3
NAMENODE ASSIGNS DATANODES
Replication 3.. Hmm.. need to find 3 datanodes for this client
How do I do that? Will tell you some other time

Hadoop HDFS

Hadoop HDFS


Here you go buddy.. Addresses of three datanodes. I have also sorted them in increasing distance from you
thanks!
Datanode 1, Datanode 2, Datanode 3
CLIENT STARTS WRITING DATA
I send my data (and the list) to first datanode only
I store the data in hard drive, and--
WHILE I am recieving data, I forward the same data to the next datanode
© Maneesh Varshney. mvarshney@gmail.com
I'll do the same what previous guy did
I am the last in chain..
TA..DA.. REPLICATION PIPELINE
INFORM NAMENODE WHEN DONE
Once all data (for this block) is written to hard disk send DONE to namenode
Done
Done
Done
Block successfully stored and replicated in HDFS
When I am done with a block, I repeat the same steps with remaining blocks
WHEN ALL BLOCKS ARE WRITTEN..
All blocks written, please close file
Case closed !! NOW I store all meta information in persistent storage (hard disks)
RECAP
I divided the file in blocks--
--for each block, I provided address of datanodes--
--we stored data via Replication Pipeline

Hadoop HDFS



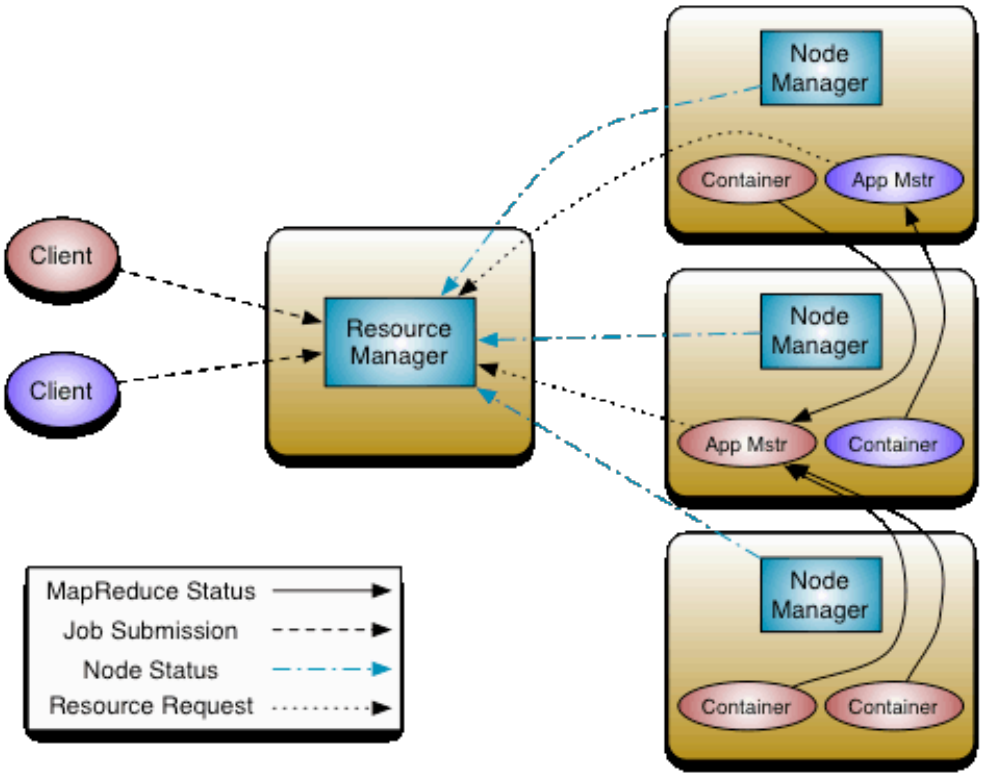
- Cartoon ©Maneesh Varshney
- More (detecting/responding to failures, etc) at: https://docs.google.com/open?id=0B-zw6KHOtbT4MmRkZWJjYzEtYjI3Ni00NTFjLWE0OGItYTU5OGMxYjc0N2M1

HDFS – More Details

- Replication
  - Default is 3 ways: 1st on local rack (or random rack if client not part of cluster), 2nd on different rack than 1st, 3rd on same rack as 2nd. Balance between reducing inter-rack traffic, providing tolerance to rack failures
- Detecting failures
  - **Node failures**: DataNode heartbeats
  - **Network failures**: ACKs
  - **Data corruption**: Checksums on transmissions
- Write failures: skip node in replication pipeline (underreplication detected, corrected later)
- Read failures: read other replica

Hadoop Yarn

- **Resource Manager**: Global engine responsible for arbitrating job resource requests, determining when jobs run
- **Containers**: Job processes run inside *containers* – essentially the resources allocated to the job on each node
- **Node Manager**: Per-machine, launches application containers, ensures they don't exceed resource allocations
- **Application master**: Runs in first container, requests resources for rest of job, manages job



From hadoop.apache.org

Hadoop MapReduce

- We saw this concept in the Jeff Dean paper earlier (and in our discussion of algorithms that can be formulated as reductions)
- Mappers (implement `Mapper` interface) process (ideally local) key-value pairs, convert ("map") them to new key-value pairs.
- Data shuffled and sorted, so that all pairs with same key land on same node
- One reducer process per key (implements `Reducer` interface) processes/summarizes all data with the same key

MapReduce Word Count


```
public static class Map extends MapReduceBase implements
  Mapper<LongWritable, Text, Text, IntWritable> {

  private final static IntWritable one = new IntWritable(1);
  private Text word = new Text();

  public void map(LongWritable key, Text value,
                  OutputCollector<Text, IntWritable> output,
                  Reporter reporter) throws IOException {
    String line = value.toString();
    StringTokenizer tokenizer = new StringTokenizer(line);
    while (tokenizer.hasMoreTokens()) {
      word.set(tokenizer.nextToken());
      output.collect(word, one);
    }
  }
}
```

MapReduce Word Count


```
public static class Reduce extends MapReduceBase implements
  Reducer<Text, IntWritable, Text, IntWritable> {

  public void reduce(Text key, Iterator<IntWritable> values,
                     OutputCollector<Text, IntWritable> output,
                     Reporter reporter) throws IOException {
    int sum = 0;
    while (values.hasNext()) {
      sum += values.next().get();
    }
    output.collect(key, new IntWritable(sum));
  }
}
```

Combiners

- Optional class – performs local reductions
  - Mappers collect key-value pairs in lists: one per key
  - Combiner method applied to each list prior to sending to reducer
  - When combiner buffer full, flushed by sending to reducer
  - Reduces communication
  - Word Count example – combiner adds counts:
    - (the, 1), (the, 1), (and,1), (the,1) -> (the,3), (and,1)

MapReduce: More Complex Algorithms?

- More complex algorithms often require multiple passes of MapReduce
  - Each pass generates key-value pairs
  - Next pass uses key-value pairs from previous pass as input
- Some of you are discussing examples in your projects (e.g., KMeans)
- New frameworks like Spark are more flexible/ don't require such contortions

Hadoop Performance Issues

- Conventional wisdom: gated on IO bandwidth, network interconnect bandwidth
  - Shuffle: Writing mapper output to disk, sending over network, reading reducer input form disk
  - Complex algorithms often require multiple phases of map-reduce – HDFS I/O between each
  - Rule of thumb: "spindle-per-core" (or per 2 cores for compute intensive jobs)
- Intermediate data lost – often have to regenerate during next map-reduce

Other Frameworks: Spark

- Tries to address two of key performance issues of Hadoop:
  - Allows "persisting" intermediate data
  - Can pipeline as many operations in a single job, keeps in memory except when shuffle necessary (or spill if runs out of memory)
- Nice performance gains
- Also, programming flexibility – not constrained to MapReduce paradigm
- Focus of next lecture

Other Frameworks: Graph based

- Giraph – iterative graph processing framework
  - *Bulk Synchronous Parallel* Model: A series of supersteps consisting of local compute-communicate-barrier synchronization
  - Based on Google Pregel
- GraphLab – machine learning via updates to data graph. User defines
  - Local vertex updates
  - Data Consistency Model determining amount of parallel computation overlap (May neighbors update simulataneously? Neighbors of neighbors?)
  - Sync mechanism for aggregating data across graph (e.g., average values)
  - Scheduling primitives defining order of computation
- SPARQL query engines: Urika-GD, AllegroGraph, Fuseki, …
- GraphX: Graph Analytics in Spark

Discussion

- Common (mis??)conceptions about data analytics performance:
  - Optimize Network
  - Optimize IO
  - Stragglers are tricky/important
- Ousterhout et al NSDI15 paper claims these aren't accurate
  - Computation is now the real bottleneck
- Do you agree? Why or why not?
- Were the workloads representative enough? (Does this matter?)
- What should we focus on to improve performance/scaling?
- Hard stop @ 8:20 to give Brad an hour.