A Distributed Computing Environment

Introduction to Analytics using Hadoop: Week One Lecture

Welcome to Introduction to Analytics using Hadoop! This course is designed to provide an introduction to the Hadoop ecosystem and particularly to introduce HDFS, the Hadoop Distributed File System, and MapReduce, a functional programming paradigm that enables significant computation over massive amounts of data.

This course is purely introductory. The Hadoop ecosystem is large, with many deep facets from systems administration and cluster environments to rich analysis with algorithms in MapReduce to creating distributed data-driven applications in the cloud. I hope to whet your appetite for these topics in the next 60 hours over four weeks and introduce you to the core topics so that you'll be conversant in Hadoop. In particular, I'm excited to introduce you to a technology that might help make your analyses viable over larger data sets and open up a path to experimentation that you may not have considered.

The first week's lecture focuses on the motivation for Hadoop, as well as an introduction to the Hadoop environment. At the end of this lecture, you should understand the components of a Hadoop cluster, the various projects that are part of the ecosystem, and you should be ready to set up your own development environment.

Table of Contents

The Motivation for Hadoop

Problems with Traditional Computing

Routine data analyses have traditionally been performed over relatively small amounts of data with a significant amount of complex processing applied. When performing analyses of this type, the amount of computation you can leverage is completely dependent on the size and performance of a single processor. Therefore, for decades, the primary motivation of computing hardware was to achieve faster processors with more RAM, thus unlocking more significant computations over larger data sets. This led to the idea of "supercomputers" – monsters of copper and silicone with massively parallel processors that could compute at petaFLOPS speed.

Unfortunately, the velocity of data has far outstripped the economics of creating such super computing beasts. Traditional mainframe distributed systems evolved to allow developers to utilize multiple machines for single jobs, but even this was not a complete solution. Exchanging data between machines requires synchronization, and because of finite bandwidth and temporal dependencies in computation, it is difficult to deal with partial failures, and thus the focus of distributed computing was to deal with failure rather than the actual problem attempting to be solved.

Two of the most well known distributed computing applications are SETI@Home or Folding@Home, and I'm routinely asked how projects like these relate to Hadoop[1] - and the answer is that they do not. Both of these projects are in fact distributed applications that solve larger problems but they behave like traditional distributed computing systems that are simply looking for available processing power, namely the unused processor cycles on home PCs. To this end, like traditional distributed systems, they store data

[1] I am not the only one routinely asked this question. Tom White also provides an answer in *Hadoop, The Definitive Guide* on page 8.

in a central repository (often a SAN), and at compute time the data is copied across the network to the node performing the computation.

Because job failure is so common (e.g. if someone shuts off their computer in the middle of a computation), many algorithms have been developed to assign an optimal number of jobs to ensure success, and to prevent malicious users from submitting bad results by enforcing voting on multiple jobs. While this is fine for relatively limited amounts of data or a very narrow computational scope, it is clearly not ideal for larger scale analysis or dynamic calculations. Both SETI@Home and Folding@Home are certainly public relations victories, but they are primarily economic victories through democratic computing rather than computing ones, that is, they bring large scale computing to the masses but do not fundamentally shift the distributed computing technological landscape.

The problems with traditional distributed computing can be summarized as follows:

1. Input and output requires synchronization and bandwidth
2. Task failure and task output quality must be evaluated
3. Computations must be parallel and not temporally dependent
4. Data, both input and output, must be stored in a central location

The idea of breaking a larger task into smaller work units so that many machines can perform work simultaneously is a good one. However, this alone is not the solution to large-data analyses. Instead, it is ideal for relatively small data inputs that require *extremely* complex computations (like protein folding).

More Data, Bigger Data, Faster Data

It is not an overstatement to say that we live in the Age of Data. I could give you the common statistics about the petabytes of data that are being generated on a daily basis: there is a thriving industry of infographics dedicated to the task of inundating you with such knowledge[2]. Actually, it is kind of fun:

- Twitter generates 1.39 MB per second, 41.8321 TB per year (which is astounding for a data type of only 160 bytes).[3]
- Facebook is working on a query engine to analyze 250 PB of data.[4]
- Walmart controls customer transactions at a rate of 1.5 million transactions per hour, for a data volume of 1.43 GB per hour and a current store of 3 PB.[5]
- Pearsons' OpenClass program involved nearly 10 TB of data for a graph with 6.24 billion vertices and 121 billion edges for nearly 788 million courses with associated teachers, students, and universities.[6]

As you can see from the above examples, it's not just the magnitude of data. It's also the rate of data. In 2012, Gartner defined Big Data with the now oft cited "3 Vs": Volume, Velocity, and Variety (and also usually appended is Veracity).[7] No matter your industry--Government, Technology, Retail, Education, etc.--the benefits and requirements of Big Data seem obvious.

While Moore's law (the number of transistors on a chip doubles every 18 months) has so far held up, allowing us the ability to store the data, the bottleneck is in getting data off the disks into a place where we can compute with it. In fact, processor speeds (I'm writing on a Mac with 2.8 GHz

[2] 235 Terabytes of data has been collected in the library of congress in 2011! http://blog.getsatisfaction.com/2011/07/13/big-data/?view=socialstudies

[3] Computed from statistics from http://www.statisticbrain.com/twitter-statistics/

[4] Presto Engine Announced: http://gigaom.com/2013/06/06/facebook-unveils-presto-engine-for-querying-250-pb-data-warehouse/

[5] http://www.bigdatarepublic.com/author.asp?section_id=2747&doc_id=262692

[6] http://thinkaurelius.com/2013/05/13/educating-the-planet-with-pearson/

[7] http://www.gartner.com/DisplayDocument?id=2057415&ref=clientFriendlyUrl

processor) are not the problem when data can't get to the processor from the disk!

Consider the Western Digital RE WD4001FYYG 4TB 7200 RPM Enterprise Hard Drive. This high capacity/high speed drive retails on NewEgg.com (at the time of writing) for $369.99 and delivers an incredible (sustained) transfer rate of 182 MB/s for sequential data (because it doesn't have to move the spindle to different sectors of the disk). To transfer the entire contents of the disk to the processor it would take 6 hours, 6 minutes, and 18.02 seconds. This is assuming that you have perfectly engineered your data for sequential reads, and that you have 4TB of memory on your server to load the data into!

Although there are many strategies for speeding up disk access (using smaller disks, RAID, or new technology like solid state drives), they all involve tradeoffs. Clearly, a new approach to data loading to the processor is required.

Requirements for a Distributed Architecture

Based on the problems with traditional distributed computing and data storage and access, we can define a few requirements for a distributed processing system, all of which, to no surprise, will motivate the core concepts that are in the Hadoop architecture.

1. Support for failure: if a component fails, it should not result in the failure of the entire system. The system should gracefully degrade into a lower performing state.
2. If a failed component recovers, it should be able to rejoin the system.
3. Recoverability: in the event of failure, no data should be lost.
4. Consistency: the failure of one job/task should not affect the result.
5. Scalability: adding load leads to decline of performance, not failure; increasing resources should result in a proportional increase in capacity.

The first two requirements discuss how nodes in a distributed system should interact with the system. For instance, if a hard drive fails (which is common, considering you have an iron disk spinning super fast by a magnetic spindle) it should not lead to a system failure, and if the hard disk is replaced it should be able to contribute its storage to the system. Similarly if an entire computer goes down (because of a corrupted file or some other parameter) and is rebooted, it likewise should not affect the overall system.

These requirements for a fault-tolerant environment lead to an interesting economic development instead of purchasing high-class servers for computation, you can instead use commercial off the shelf (COTS) hardware to set up large-scale computation. This has significant implications for technology startups, academia, government, and any other resource-constrained computing environment. In fact, at my tech startup, we created a data computation cluster for less than $1500!

Figure 1: A seven-node COTS Hadoop cluster

This course doesn't get into the specifics of systems administration for a multi-node Hadoop cluster, though I would be happy to provide resources on request. We'll primarily focus on developing for these types of clusters so that you can get the most out of computing on them when you use your expense account to create the cluster!

A Brief History of Hadoop

The open source version of Hadoop has its origins in the Nutch project, an open source web crawler and search engine created by Doug Cutting as part of the Apache Lucene project, a text search library. Crawling and indexing the web is an expensive project, and at the time it was estimated that a billion page index would cost half a million dollars in hardware, with hosting and running costs of around thirty grand a month![8] However, Nutch's proprietary-source competitor Google released in 2003 and 2004 two seminal papers that would change the course of the search engine and launch Apache Hadoop.

In the 2003 paper *The Google File System*, Sanjay Ghemawat et al describe the architecture of a distributed file system called GFS in production at Google. This file system stores data locally to the processors that compute on it, in a distributed fashion with data overhead performed by a managing server, thus reducing the management cost of storage nodes. A second paper, *MapReduce: Simplified Data Processing on Large Clusters* added the computation piece of the puzzle the following year.

Apache Hadoop was born with the two famous modules as described by those papers: HDFS, the Hadoop Distributed File System, and a MapReduce API for operating on data in HDFS. Yahoo! became one of the early adopters, and announced in February 2008 that its production search engine was being run by a 10,000 node Hadoop cluster[9]. That same year, Apache elevated Hadoop to its own top-level project.

[8] Building Nutch: Open Source Search http://queue.acm.org/detail.cfm?id=988408

[9] http://developer.yahoo.com/blogs/hadoop/yahoo-launches-world-largest-hadoop-production-application-398.html

In one important feat, Hadoop broke the terabyte sort record and still holds the record at the time of this writing, able to sort 1.42 TB/min.[10] Google reports being able to sort a Petabyte of data in 6 hours![11]

Perhaps an even bigger indicator of success is the Linux-like advent of multiple distributions of Hadoop, including MapR, Cloudera, and Hortonworks to name a few. Amazon's EC2 also supports EMR- Elastic MapReduce, an on demand cloud based implementation of MapReduce that you can scale up as your computation needs require.

Basic Concepts in Hadoop

The core concepts of Hadoop implement the requirements that we outlined above, and also define how a cluster and MapReduce job operates.

1. Data is distributed as it's stored and nodes work on the data that's local to them so that data doesn't have to be transferred across the network.
2. Applications are written at a high level without concern for network programming, time, or low-level infrastructure.
3. The amount of network traffic between nodes is reduced. Each job should be independent and nodes should not have to communicate with each other.
4. Data is duplicated multiple times across the system to provide redundancy both at the data level and at the job level.
5. Data is stored in blocks of 64 or 128 megabytes.
6. Every Map task works on a single block of data.
7. Master programs allocate work to nodes such that Map tasks have locally available data so that many nodes can work in parallel, each on their own portion of the larger data set.

[10] http://sortbenchmark.org/

[11] http://googleresearch.blogspot.com/2011/09/sorting-petabytes-with-mapreduce-next.html

8. Jobs, nodes, and disks are failure tolerant through redundancy.

The long and short of it is actually very simple: HDFS and MapReduce work in concert to minimize the amount of network traffic and to ensure that data is local to the required computation. Duplication of both data and jobs ensures fault tolerance and consistency, and several processes manage the whole cluster so that they work in concert to provide a programming platform to abstract low level details away from developers.

A natural question that might occur to the reader is: "how are many local tasks federated into a complete computation?" The answer to this question is the MapReduce paradigm. Specifically, the Reduce operation of MapReduce merges the results of many single computations into an aggregated whole. We will discuss MapReduce in detail more later.

> **Hadoop is made up of two main components: HDFS and MapReduce. These components ensure a programming interface that abstracts developers from low-level cluster based computation details.**

The management processes are often called Masters (and can refer to a single machine running only a single master process). Master nodes detect failures and reassign work to different nodes. Tasks can be restarted independently, and if a slow or failing task restarts it will be added back to the system automatically by the Master node and assigned new tasks.

During the execution of a job, Master nodes also ensure that final computation is not affected by node failure by detecting slow tasks and redundantly assigning another instance of the task. Whoever finishes first wins, this is called "speculative execution". These basic concepts will be described in more detail in the following architectural sections.

The Hadoop Architecture

Hadoop is made up of two main components: HDFS and MapReduce. These components ensure a programming interface that abstracts developers from low-level cluster based computation details. A set of machines that is running HDFS and MapReduce is known as a cluster, and the individual machines are called nodes. A cluster can have a single node, or many thousands of nodes, but all clusters scale linearly, e.g. the more resources you add, you get a linear increase in capacity and performance.

The two components are implemented by five separate daemon processes:

1. **NameNode** – Contains metadata for HDFS operations
2. **Secondary NameNode** - Performs various housekeeping tasks for HDFS; *not* a backup NameNode
3. **DataNode** - Stores and manages HDFS blocks on a node
4. **JobTracker** – Manages MapReduce jobs and allocates tasks
5. **TaskTracker** – Runs and monitors individual Map and Reduce tasks

Each of these processes runs inside of its own Java Virtual Machine (JVM) so each daemon has is own system resource allocation and is managed independently by the operating system. Although it is technically possible to run all five daemons on the same node, this is not common. Instead nodes are usually one of two types:

1. **Master Nodes**: Run the NameNode, Secondary NameNode, and JobTracker processes. Only one of each of these daemons needs to run on the cluster, therefore there are typically only three Master Nodes.
2. **Slave Nodes**: These are the worker nodes of the cluster, and the majority of the node types. Each slave node has to run both the DataNode and the TaskTracker and is responsible for both storage and computation.

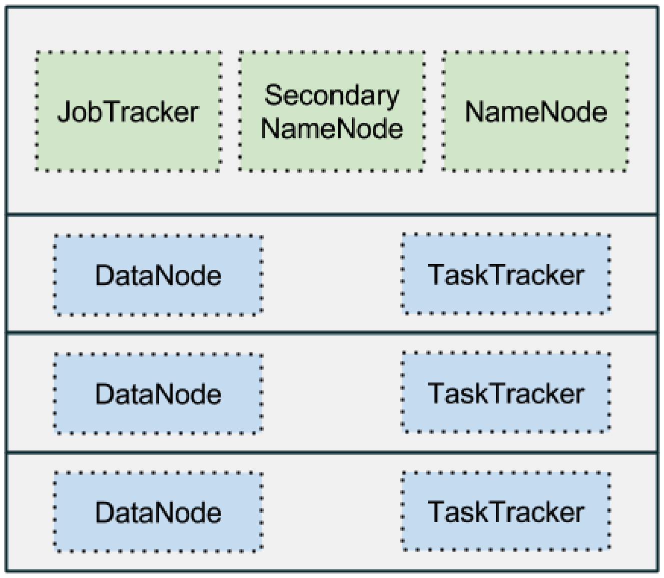


Figure 2: A small Hadoop cluster with one Master and three Slaves

In very small clusters of about 20-30 nodes, usually one machine is the Master Node, and the rest are Slave Nodes. However, more significant deployments require greater capacity for these daemons, and each process resides on its own machine.

In “Pseudo-Distributed Mode” a single machine runs all 5 daemons as though it were part of a cluster. In this mode, the benefits of a distributed architecture aren’t realized, but it is the perfect setup to develop on without having to worry about administering several machines. For the rest of this course we will develop in “Pseudo-Distributed Mode” inside of a Virtual Environment.

Installation and Setup

There are several mechanisms for you to set up your development environment in pseudo-distributed mode. Please refer to the instructions for installation in the week one resources section for more detail. For the purposes of this course, however, you have three options (from most detail to least detail):

1. Use the fully configured virtual machine provided by the instructor.

2. Download Cloudera's CDH4 quick start virtual machine and configure.
3. Follow the instructions to configure the environment from scratch.

If you have no systems administration or server experience, I recommend the first option. If you do have some Linux experience, it is a useful exercise to go through the third option. The second option is interesting: it gives you a Cloudera distribution, which at the time of writing is standard for Hadoop, but you must configure and manage it for the course work.

Hadoop Distributed File System (HDFS)

HDFS is a distributed file system that is based on the Google File System paper from 2003. HDFS provides redundant storage for Big Data by storing that data across a cluster of cheap, unreliable computers, thus extending the amount of available storage capacity that a single machine alone might have. However, because of the networked nature of a distributed file system, HDFS is more complex than traditional file systems.

In principle, HDFS is a software layer on top of a native file system such as ext4 or xfs, and in fact Hadoop generalizes the storage layer and can interact with local file systems and other storage types like Amazon S3. However, HDFS is the flagship distributed file system, and for most programming purposes you'll be interacting with it. HDFS is designed for storing very large files with streaming data access and as such, it comes with a couple of caveats:

1. HDFS performs best with a modest number of very large files, e.g. millions rather than billions of files that are 100 MB or more.
2. HDFS implements the WORM pattern – write once, read many. No random writes or appends to files are allowed.
3. HDFS is optimized for large, streaming reading of files not random reading or selection.

Therefore HDFS is best for storing raw input data for computations as well as files with intermediary computational data, and finally results files rather than modifying records for lookups in real time.

Blocks

HDFS files are split into blocks, usually of either 64 MB or 128 MB, though this is configurable at run time. The block size is the minimum amount of data that can be read or written to in HDFS, similar to the block size on a single disk file system. However, unlike blocks on a single disks, files that are smaller than the block size do not occupy the full blocks' worth of space.

Blocks allow big files to be split across and distributed to many machines at run time. Different blocks from the same file will be stored on different machines to provide for more efficient MapReduce processing. In fact there is a one-to-one connection between a task and a block of data.

Additionally, blocks will be replicated across the DataNodes. By default the replication is three-fold, but this is also configurable at run time. Therefore each block will exist on three different machines and three different disks, and if even two nodes fail, the data will not be lost.

Data Management

The master NameNode keeps track of what blocks make up a file and where those blocks are located. The NameNode communicates with the DataNodes, the processes that actually hold the blocks in the cluster. Metadata associated with each file is stored in the memory of the NameNode master for quick lookups, and if the NameNode stops or fails, the entire cluster will become inaccessible!

The Secondary NameNode is not a backup to the NameNode, but instead performs housekeeping tasks on behalf of the NameNode including (and especially) periodically merging a snapshot of the current data space with the edit log to ensure that the edit log doesn't get too large. If the NameNode fails, this merged record can be used to reconstruct the state of the DataNodes.

When a client application wants access to read a file it first requests the meta data from the NameNode to locate the blocks that make up the file, as well as the locations of the DataNodes that store the blocks. The application then communicates directly with the DataNodes to read the data. Therefore, the NameNode simply acts like a journal or a lookup table and is not a bottleneck to simultaneous reads.

The Small Files Problem

A small file in the context of Hadoop is one that is significantly smaller than the block size, which is by default 64 MB. Generally speaking users of Hadoop operate on extremely large data sets, which probably means that users with small files generally have lots of them. HDFS cannot handle lots of files. Every file, directory, and block in HDFS is stored in the NameNode's memory for quick access. This problem is usually called "the small files problem."[12]

Since each NameNode record occupies 150 bytes, for 50 million files each with a single block with two files per directory, the NameNode would require 18.75 GB worth of main memory. Although this might be feasible with larger servers, scaling quickly becomes a problem, and certainly a billion or billions of files would not be feasible at all. Furthermore, HDFS is designed for streaming access for large files; hopping through small files causes lots of seeks and is not optimal. Finally, MapReduce tasks are designed to operate on a single block- more blocks mean more tasks, which means more overhead. MapReduce jobs will run faster the fewer blocks there are.

[12] http://blog.cloudera.com/blog/2009/02/the-small-files-problem/

There are two main reasons that small files exist. First, the files are parts of a larger file. Because append is a recent addition to HDFS, unbounded files can be split without regard to block size. Second, the files can be small parts of a larger corpus. Each file is distinct, and there is no way to combine them into a larger file. In the first case, you can use `sync` to concatenate disjoint files, and in the second case you simply need to store the files in a single container file.

MapReduce

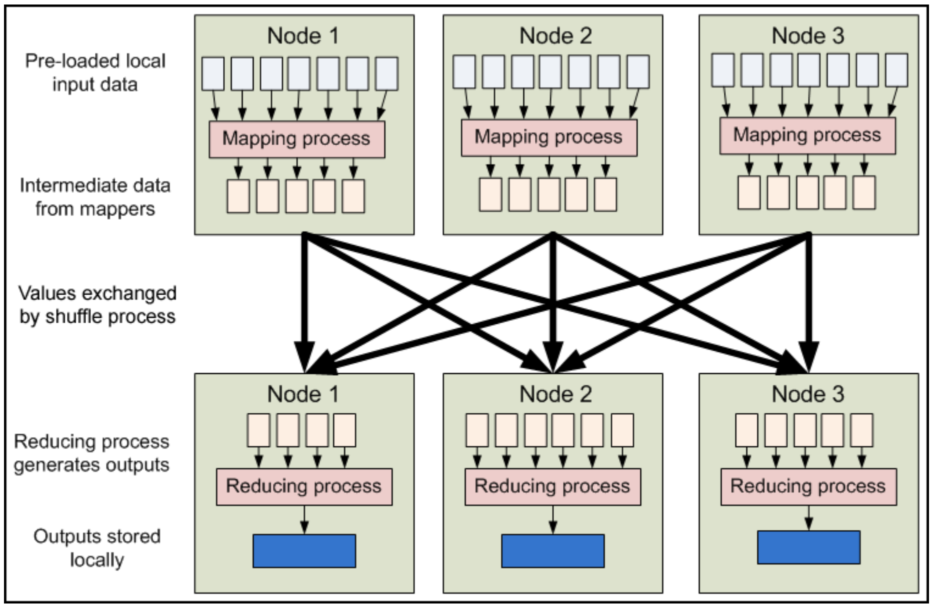
MapReduce is a functional programming paradigm eponymous with the programmatic API in Hadoop. In the Hadoop context, it is the methodology that allows the distribution of a task across multiple nodes in a cluster, and is a simple yet expressive model to represent analytical programs. Hadoop can run MapReduce programs in several languages including Java, Ruby, Python, and C++.

MapReduce is parallel by design, which means that it is extremely effective across large data sets in a distributed fashion--perfect for Hadoop! Hadoop goes further to ensure that each node operates on data stored on the executing node, thus minimizing the amount of network traffic that might be involved in traditional parallelism.

The implementation of MapReduce in Hadoop is written in Java. It provides automatic parallelization, job distribution, management, fault-tolerance, and monitoring. These features lead to a clean abstraction for programmers who typically program MapReduce jobs in Java. Programmers do not have to then worry about any of the housekeeping of MapReduce functionality, but instead focus on the two parts of a MapReduce task--Map and Reduce functions.

The JobTracker master node controls MapReduce jobs. Clients submit jobs to the JobTracker, which then assigns Map and Reduce tasks to the TaskTrackers on the cluster. The TaskTracker on each node is responsible for instantiating the Map or Reduce task and reporting progress in a heartbeat fashion back to the JobTracker. Some terminology is below:

- **Job:** A "full program": the complete execution of Map and Reduce functions across the entire input data. Managed by the JobTracker.
- **Task:** The execution of a single Map or Reduce function on a single block of data, presided over by TaskTrackers.
- **Task Attempt:** A particular instance of an attempt to execute a task. There are at least as many attempts as tasks, but if tasks become slow or unresponsive additional Task Attempts are instantiated using Speculative Execution to ensure consistency in computation.


Pre-loaded local input data
Intermediate data from mappers
Values exchanged by shuffle process
Reducing process generates outputs
Outputs stored locally
Node 1
Node 2
Node 3
Mapping process
Reducing process

- **Figure 3: MapReduce Pipeline**[13]

The phases of a MapReduce job are described in Figure 3, and can be summarized as follows.

1. Local data is preloaded to the mapping process, which accepts as input key/value pairs. Normally the input key is a document id and the input value is the lines of input text data.

[13] Image from http://developer.yahoo.com/hadoop/tutorial/module4.html used under a Creative Commons Unported License.

2. The Mappper must output zero or more key/value pairs, e.g. mapping computed values to a particular key.
3. The key/value pairs from the Map function are then sorted and shuffled based on the key and one or more reducers receive the key and the list of values associated with that key.
4. The Reducer then must output zero or more final key/value pairs, and the output is written to HDFS. In practice, the Reduce usually emits one key/value pair for each input key, hence the reduction of the value list into a single value.

The canonical example is the Word Count MapReduce program, and it seems to appear in every instructional text regarding MapReduce. Since it is the "Hello World" of MapReduce, who am I to change it? The following pseudo code illustrates MapReduce:

```
def map(String inputKey, String inputValue) {
     foreach word w in inputValue:
          emit(w, 1)
}

def reduce(String outputKey, Iterator<int> counts) {
     set count = 0
     foreach int v in counts:
          count += v
     emit(outputKey, count)
}
```

When executed with the following input, the flow of data is as follows:

Input to WordCount:

```
(31416, "the cat in the hat ran fast")
(27183, "the fast cat wears no hat")
```

Output from the Mapper:

```
("the", 1), ("cat", 1), ("in", 1), ("the", 1), ("hat", 1),
("ran", 1), ("fast", 1), ("the", 1), ("fast", 1), ("cat",
1), ("wears", 1), ("no", 1), ("hat", 1)
```

Intermediate Data from Shuffle and Sort:

```
("cat", [1, 1])
("fast", [1, 1])
("hat", [1, 1])
("in", [1])
("no", [1])
("ran", [1])
("the", [1, 1, 1])
("wears", [1])
```

And the final Reducer output:

```
("cat", 2)
("fast", 2)
("hat", 2)
("in", 1)
("no", 1)
("ran", 1)
("the", 3)
("wears", 1)
```

While this may seem excessive for simply computing a word count, as data gets larger, this paradigm becomes more and more effective. Also note the native sort and single key to reducer paradigm--this allows us to write extremely expressive algorithms and computations using MapReduce.

The final piece of the puzzle, often left out, is the combiner. The combiner acts as an intermediary reducer after a Map function, but before the shuffle and sort. Generally speaking, the Reduce function used in the job is also used for combining. This way, instead of an iterator of 1s passed to the reducer, some summation can be done before the shuffle and sort (summing the counts out of each Map phase). This speeds up the shuffle and sort as well as the final Reduce job.

The Hadoop Ecosystem

Hadoop specifically refers to the deployment of a combination of HDFS and MapReduce daemon processes to a cluster and the APIs used to interact with that cluster. The project containing the binaries and code bases is named as such. However, in general usage, Hadoop often refers to the ecosystem of projects that use Hadoop, usually both HDFS and MapReduce or simply just HDFS. The most noteworthy of these projects are also Apache or Apache Incubator, and many distributions come bundled with these projects. Since they are an important part of Hadoop workflows, we will briefly introduce a few here.

Hive

Hive is an abstraction of MapReduce that provides a framework for data warehousing on top of Hadoop, developed by the Facebook team. In particular it provides a HiveQL language that is very similar to SQL and allows analysts with strong SQL skills to run queries on high volumes of data residing in HDFS. The Hive Interpreter simply runs on a client machine and expands HiveQL queries into MapReduce jobs that are then submitted to the cluster.

While SQL isn't ideal for every data problem, it is a standard among data analysts, and a common bridge between other data applications for integration to software platforms not just people. It also allows programmers to express MapReduce jobs in an expressive data manipulation language.

Pig

Pig is another MapReduce abstraction that uses a dataflow scripting language called *Pig Latin*. Like the Hive interpreter, the Pig interpreter runs on a client machine and converts a Pig Latin script into MapReduce jobs and submits them to the cluster. Pig can also run in a local execution environment, not just distributed across a cluster.

Pig's main functionality is to ensure that complex MapReduce jobs abstract the long development time of writing jobs, executing them on a cluster and determining the results from there. Pig, therefore, is ideal for writing large and complex tasks and maintaining them.

Flume and Sqoop

Flume and Sqoop provide methods for importing data into HDFS.

Flume efficiently collects, aggregates, and moves large amounts of log data--harkening back to the original use of Hadoop to analyze Common Record log format generated from server logs. Flume focuses on importing the data in a fault-tolerant fashion as it is generated.

Sqoop provides a method to import data from tables in a relational database into HDFS and vice versa. It is particularly efficient as it is a Map-only job. Many existing data stores are already in relational tables, and Sqoop is by far the best way to get them out.

HBase

HBase is the "Hadoop database", a NoSQL data store that can store billions of rows and millions of columns. HBase is perhaps the original "Big Table" and can store massive amounts of tabular data through the terabyte range to perhaps even the petabyte range. HBase is scalable to provide hundreds of thousands of inserts per second, and does very well with sparse data (e.g. columns that are empty in records).

Conclusion

In this lecture you have been introduced to Hadoop, from its motivations and origins in Big Data and distributed computing to its history and core concepts. We have discussed the anatomy of a Hadoop cluster and went specifically into the two fundamental components of Hadoop: HDFS and

MapReduce. Finally, we learned about other projects in the Hadoop ecosystem.

In the next week we'll get our hands into Hadoop, organize some data on HDFS and execute some simple MapReduce jobs. For the rest of the week, take a look at the reading and assignments and dive into the discussion questions. You'll also have the opportunity to set up your development environment as we move into Hadoop-centric exercises next week.

Assignments

Please post your answers to the discussion question by Wednesday. The exercises for the first week are due on Friday.

Supplementary Reading:

1. *The Google File System* by S. Ghemawat, H. Gobioff, and S. Leung
2. *MapReduce: Simplified Data Processing on Large Clusters* by J. Dean and S. Ghemawat
3. A short introductory video by the instructor

Optional Reading

4. Chapters 1, 9, and 16 from *Hadoop, The Definitive Guide*

Discussion (graded)

Please post your answer to the discussion question below to the forum. The discussion questions are designed to spark debate, comments, and conversation- please check the forum regularly for updates and respond to your classmates!

1. Please post your answer to the discussion question below to the forum. The discussion questions are designed to spark debate, comments, and conversation- please check the forum regularly for updates and respond to your classmates!

   Consider a typical distributed system, either the SETI message passing interface, Hadoop, or some other parallelized architecture where many small tasks performed by independent machines make up a larger job. Let's discuss these types of systems in the context of large-scale analytics. Questions to answer could include the following. What are some strategies for dealing with data loss? What about for Job failure? What if you can't trust the results of a job? How would you characterize the difficulty of putting together such a system? What are some of the pros, the cons?

Questions for Thought (not graded)

1. Please describe some of the data sets that you analyze. How would they fit into a Hadoop/MapReduce context? Do they suffer from the Small Files problem? How would you fix that problem?
2. Three configurable options in a Hadoop cluster are the block size, replication factor, and number of Map tasks per node. How would increasing or decreasing these options affect the cluster? How are these configurable options related? Another configurable option is the number of Reduce tasks per job--how does changing this option affect Map Reduce jobs?
3. What types of algorithms could not be fit into the MapReduce functional paradigm?

Exercises (Graded)

There will be more details on each of these assignments, along with point breakdowns, on the assignments page in the course materials.

1. There is a small data set of Apache Log records in the Resources section of the course materials. In any programming language implement a program that computes the number of hits to the server per calendar month.
2. There is a data set containing the collected works of Shakespeare in the Resources section of the course materials. In any programming language implement a program that computes the average word length per letter of the alphabet.

Implementing these assignments in a MapReduce context will assist in following assignments in the next week!