Module 9: Cloud Computing and MapReduce

The common use of the term *cloud computing* refers to a business model for leasing computer resources. *Utility Cloud* systems enable people and organizations to rent any computer resource configuration under the terms that are mutually agreed upon by both consumers and the providers. More formally, cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. An interesting class of cloud computing technologies revolves around massively parallel, distributed computing techniques on large data sets. It is the latter that will be discussed for most of the remainder of this course. First, however, we will devote a few words to discussing the utility cloud.

The Utility Cloud

According to the National Institute of Standards and Technologies (NIST), there are five essential characteristics of a utility cloud. These characteristics allow users to provision resources autonomously, quickly, and in the desired quantity. They also enable cloud providers to amortize the cost of their resources among multiple consumers and to charge consumers for the resources they request. The characteristics of a cloud system are listed below:

- On-demand self-service – Consumers of the cloud can unilaterally provision computing capabilities, such as server time and network storage, as needed automatically without requiring human interaction with a cloud service provider.

- Broad network access – Capabilities are available over the network and are accessed through standard mechanisms (e.g., mobile phones, tablets, laptops, and workstations).

- Resource pooling – The cloud provider's computing resources are pooled to serve multiple consumers. Resources are assigned and reassigned according to consumer demand.

- Rapid elasticity – Capabilities can be elastically provisioned and released to scale rapidly with demand. To the consumer, the capabilities available for provisioning often appear to be unlimited and can be appropriated in any quantity at any time.

- Measured service – Cloud systems automatically control and optimize resource use by leveraging a metering capability appropriate to the type of service (e.g., storage, processing, bandwidth, and active user accounts). Resource usage can be monitored, controlled, and reported, providing transparency for both the provider and consumer of the utilized service.

Clouds provide services that support all aspects of consumer computer system needs. The classes of services that are provided by clouds are binned into three broad categories.

- Software as a Service (SaaS) – Cloud providers provide to consumers *application* software that runs on the cloud infrastructure. The software applications are accessible from various client device interfaces (e.g., standard web browsers, mobile phones, REST, JDBC, etc.). The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, storage, or even individual application capabilities, except for session specific application configuration settings.

- Platform as a Service (PaaS) – Cloud providers provide consumers with the capability to deploy consumer-created or acquired *middleware* on the cloud infrastructure. Consumers do not manage or control the underlying cloud infrastructure including network, servers, operating systems, or storage, but have control over the deployed applications and possibly configuration settings for the application-hosting environment.

- Infrastructure as a Service (IaaS) – Cloud providers provided consumers with the ability to provision processing, storage, networks, and other fundamental *operating system level* resources. The consumer does not manage or control the underlying cloud infrastructure but has control over operating systems, storage, and deployed applications; and possibly limited control of select networking components (e.g., host firewalls).

In essence, customers can provision resources at the application (SaaS), middleware (PaaS), operating system (IaaS) layers, or any combination thereof. Consumers negotiate with cloud providers and/or cloud brokers to lease resources with the agreed upon usage plan. Aspects of the plan can cover any of lease length, bandwidth, quality of service, disk space usage, etc. Agreements are usually protected by a contract that both parties enter into.

There are four types cloud deployment models for cloud providers.

- Private cloud – A cloud infrastructure is provisioned for exclusive use by a single organization comprising multiple consumers. It may be owned, managed, and operated by the organization, a third party, or some combination of them, and it may exist on or off premises.

- Community cloud – A cloud infrastructure is provisioned for exclusive use by a specific community of consumers from organizations that have shared concerns (e.g., mission, security requirements, policy, and compliance considerations). It may be owned, managed, and operated by one or more of the organizations in the community, a third party, or some combination of them, and it may exist on or off premises.

- Public cloud – A cloud infrastructure is provisioned for open use by the general public. It may be owned, managed, and operated by a business, academic, or government organization, or some combination of them. It exists on the premises of the cloud provider.

- Hybrid cloud – A cloud infrastructure is a composition of two or more distinct cloud infrastructures (private, community, or public) that remain unique entities, but are bound together by standardized or proprietary technology that enables data and application portability.

Cloud Privacy and Security

Since multiple consumers lease resources on the cloud simultaneously, there are concerns that *privacy* can be compromised. One consumer's software could potentially examine, modify, or delete another consumer's data stored on the cloud. Moreover, service providers themselves may compromise consumers' privacy. To address privacy issues, cloud providers usually place mechanisms into their cloud systems to protect consumer privacy.

Security techniques are often employed by cloud providers to protect and ensure confidentiality, integrity, and availability of consumer information. Some security techniques include *cryptography* and *virtualization*. Cryptographic techniques protect passwords, stored data, and communications. Virtualization is a family of techniques that establish virtual instances of resources that are torn down after their use. For example, consumers do not place software directly onto provider operating systems. Rather, the cloud providers create a virtual operating system layer on top of the real operating system and consumers run their software on the virtualized operating system. This way, if viruses and other malware establish themselves in the operating system kernel, they will only establish themselves on the virtual operating system. When the consumer software finishes running, the virtual operating system is torn down along with the malware that planted itself into the virtual operating system. Thus, the real operating system is protected from the malware. Virtualization techniques are an active area of research and development among cloud vendors and university research laboratories. They are beyond the scope of this module.

The Utility Cloud Benefits

A significant benefit realized by cloud computing is the reduced cost of computing resources to consumers. Consumers pay incrementally for just the services they need without having to invest in large amounts of hardware and software. The cloud computing paradigm makes it easier and cost efficient for startup projects, programs, and even companies to get right to work without having to devote too much money and time into buying and establishing an IT infrastructure.

Another benefit realized by cloud computing is increased storage. Organizations can store more data than on private computer systems because providers typically purchase large amounts of storage.

In addition, consumers experience more flexibility because they can rapidly expand and contract their resource usage. A company might normally use a terabyte of storage, but every once in a while requires 10 terabytes. Instead of the company purchasing 10 terabytes of disk space that mostly sits empty, the cloud model enables the company to lease one terabyte most of the time, rapidly expand to 10 terabytes for short-term needs, and reduce back to one terabyte of storage. Rapid expansion and contraction of cloud computing resources provides more flexibility than past computing models.

Another benefit is increased mobility. Consumers can access information wherever they are, rather than having to remain stationary. This is beneficial for consumers that need access to data from multiple geographic locations.

Finally, cloud computing reduces the need for consumer IT staff. Consumers no longer worry about constant server updates and other infrastructure maintenance issues. Consumers are free to concentrate on their tasks while leaving IT issues and actions up to the cloud service providers.

In summary, cloud computing is essentially a business model for provisioning computer resources at the scales needed by consumers. It enables consumers to pool their resources to achieve an overall savings to their organizations. This was a short discussion of the utility cloud and we just touched upon some of its high-level issues. For more information, a few references are provided at the end of this module.

MapReduce

MapReduce is a programming model and associated implementation for processing large data sets. The programming model's roots are in the Lisp *mapcar* function:

A well-known equality is:

$$(1^3 + 2^3 + 3^3 + \ldots + n^3) = (1 + 2 + 3 + \ldots + n)^2 = (\sum_{x=1}^{n} x)^2$$

Using *mapcar*, we can calculate this function in two ways. Calculating the left side of the equality, we can write the following lisp code that uses *mapcar* to apply a *cubed* function (assuming one is defined) to each element of a list of numbers, and then apply the *sum* function (expressed as +) to the list:

```
(+ (mapcar 'cubed (1 2 3 … n)))
```

The first step of the evaluation maps the *cubed* function to each element of the list. This is called the *map* step.

```
(+ ((cubed 1) (cubed 2) (cubed 3) … (cubed n)))
```

(+ (1 8 27 … n^3))

The second step of the operation adds all the elements in the list. This is the called the *reduce* phase. The result is the sum of all the elements of the list squared.

$$(\sum_{x=1}^{n} x)^2$$

The MapReduce paradigm borrows from this notion. It applies the *Map* function to process large datasets in parallel and generates an intermediate set of data. The *Reduce* function merges and processes the intermediate data to produce results. It turns out that many applications can be expressed in this form. The key similarity among these applications is that there is an initial step where a large amount of data is processed independently in parallel and the intermediate results are sorted and reduced to produce a result.

The underlying MapReduce architecture automatically parallelizes the Map and Reduce functions. It takes care of partitioning the data, scheduling the map and

reduce functions, handling machine failures, and managing the inter-machine communications. Programmers just need to write the map and reduce code.

History

During the years 2002-2004, Doug Cutting worked on Nutch, which is a crawler-based search engine built on Lucene. Scaling was clearly the hardest problem to solve because it had to process over 100M web pages.

During the years 2004-2006, Google published the Google File System (GFS) and MapReduce papers. Doug Cutting saw its potential to help scale his crawler-based search engine by processing lots of web pages in parallel. He and another developer added this architecture to Nutch. In doing so, he reached the scalability that he desired.

In 2006-2008, Yahoo! hired Cutting and wrote the public domain versions of MapReduce and GFS called Hadoop and Hadoop Distributed File System (HDFS), respectively. They achieved "web scale" early in 2008. Hadoop and HDFS are publicly available at the Apache site (http://hadoop.apache.org/).

Motivating Problems

The following unit describes two separate problems that MapReduce handles very well. The first problem is creating a word count array from a large corpus of documents. The second problem is to determine how many web pages on the Internet point to a single web page, for a single web page on the Internet:

(Watch Module 9A – Motivating Problems for MapReduce)

There have been hundreds of MapReduce functions that have been written. The following is a small list:

- **Distributed Grep**
  - Map emits a line if it matches a given pattern.
  - Reduce is an identity function that just copies the intermediate results to the output.
- **Count of URL Access Frequency**
  - Map processes logs of web page requests and outputs <URL, 1>.
  - Reduce adds all values for the same URL and emits a <URL, total count> pair.
- **Graph Traversal**
  - Map processes the *from* and *to* nodes.
  - Reduce sorts to nodes and gets ready for the next Graph Traversal step.
- **Machine Learning**
  - Example: Find the K nearest neighbors to a value
  - Map processes the seed element.
  - Reduce identifies the K elements most like the seed element.
- **Term-Vector per Host**
  - A term vector summarizes the most important words that occur in a set of documents as <word, frequency> pairs.

  - Map emits a <hostname, term vector> pair for each input document.
  - Reduce adds these term vectors together, throwing away infrequent terms, and then emits a final <hostname, term vector> pair.
- **Inverted Index**
  - Map parses each document, and emits a sequence of <word, documentID> pairs.
  - Reduce accepts all pairs for a given word, sorts the corresponding document IDs and emits a <word, list(document ID)> pair.
- **Joins**
  - Map finds keys for rows matching a search criteria.
  - Reduce finds other elements that have the corresponding foreign key.

MapReduce Conceptual Architecture

Figure 1 depicts the MapReduce conceptual architecture. The MapReduce library in the user program *shards* the input files into M pieces of typically 16 - 64 MB per shard. The assumption is that the file size is M times the size of a shard, which can be 64*M megabytes. The size of a shard is much larger than a normal Network File System (NFS) page, which is usually no larger than 8KB.

The Master process starts multiple copies of a Map program on a cluster of machines. The Master assigns copies of the Map program to multiple workers that operate on different shards of the input file in parallel. The Master selects idle workers and assigns each one a Map task or a Reduce task. The master assigns M Map tasks and R Reduce tasks.

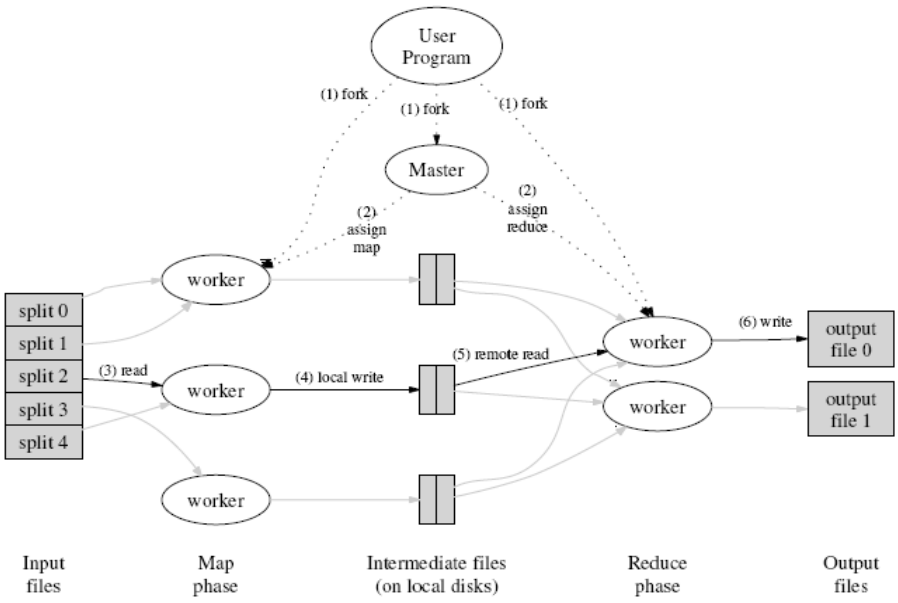


Figure 1. MapReduce Conceptual Architecture

A Map worker reads the contents of its corresponding input shard. It parses key/value pairs from the input data by passing each pair to the user-defined Map function. The Map workers write intermediate key/value pair outputs to memory

buffers, which are then written to local disk. They are partitioned into R regions by the partitioning function and the locations of these buffered pairs on the local disk are passed back to the Master.

The system reads the buffered data from the local disks and sorts the intermediate keys so that all occurrences of the same key are grouped together. Reduce workers are notified by the Master about these locations. For each unique intermediate key encountered, the Master passes the key and the corresponding set of intermediate values to a corresponding Reduce worker. The Reduce worker iterates over the sorted intermediate data and processes the data. The output of each Reduce process is appended to a final output file.

Upon completion of all Map and Reduce tasks, the Master wakes up the user program and control is returned back to the user program. The output of the MapReduce execution is available in the output file.

A more detailed view of the MapReduce architecture is depicted in Figure 2. Each node contains a disk and a processor and there are many nodes in the MapReduce architecture. Inputs are read from the HDFS file system and are split into shards. Record readers (RR) iterate through the shards and pass data to the Map workers. The Map workers produce intermediate <key, value> pairs that are partitioned and sorted by keys. The sorted values may be reshuffled to other nodes that will process them. Reduce workers read the sorted intermediate <key, value> pairs, operate on them, and produce final <key, value> pairs. The final <key, value> results are collected and written back into the HDFS system.

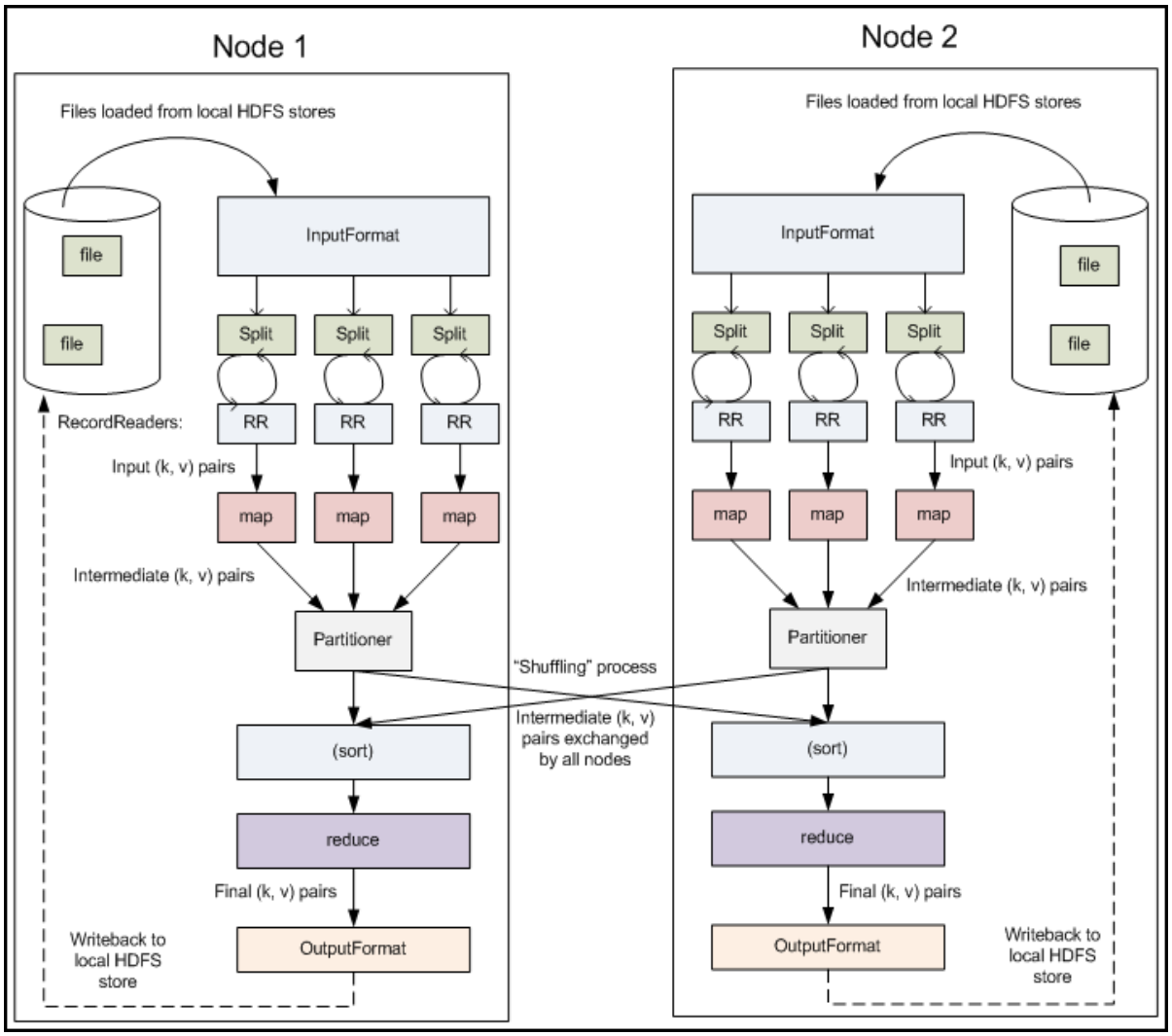


Figure 2. Another View of the MapReduce Architecture.

There are occasions where Map processes produce many more intermediate values than required. For example, the Word Count program that counts all the words in every document in a corpus is often written such that the Map process emits a single <*word*, 1> key-value pair for every instance of every word it encounters. If a document contains 12 instances of the word "cat", the pair <"cat", 1> is emitted twelve times. Consequently, each pair would become another output value that needs to be sorted and needs to be sent to the appropriate Reducer. However, more optimal performance would be achieved if a single instance of the pair <"cat", 12> would be emitted. It would reduce file space, sort time, and message passing bandwidth in the architecture.

To achieve this, a Combiner step is added to run after the Map but before the reduce steps. It serves as a "mini-reduce" process on data generated from each Map process before the intermediate data is sorted and shuffled among the nodes. This is depicted in Figure 3. Optional instances of the Combiner are run on every node that has run Map tasks. The input to the Combiner is data emitted by the Mapper instances on each node. The output from the Combiner is then sent to the Reducers

instead of the output from the Mappers. Often, the same Reduce code can usually be used as Combiner code!

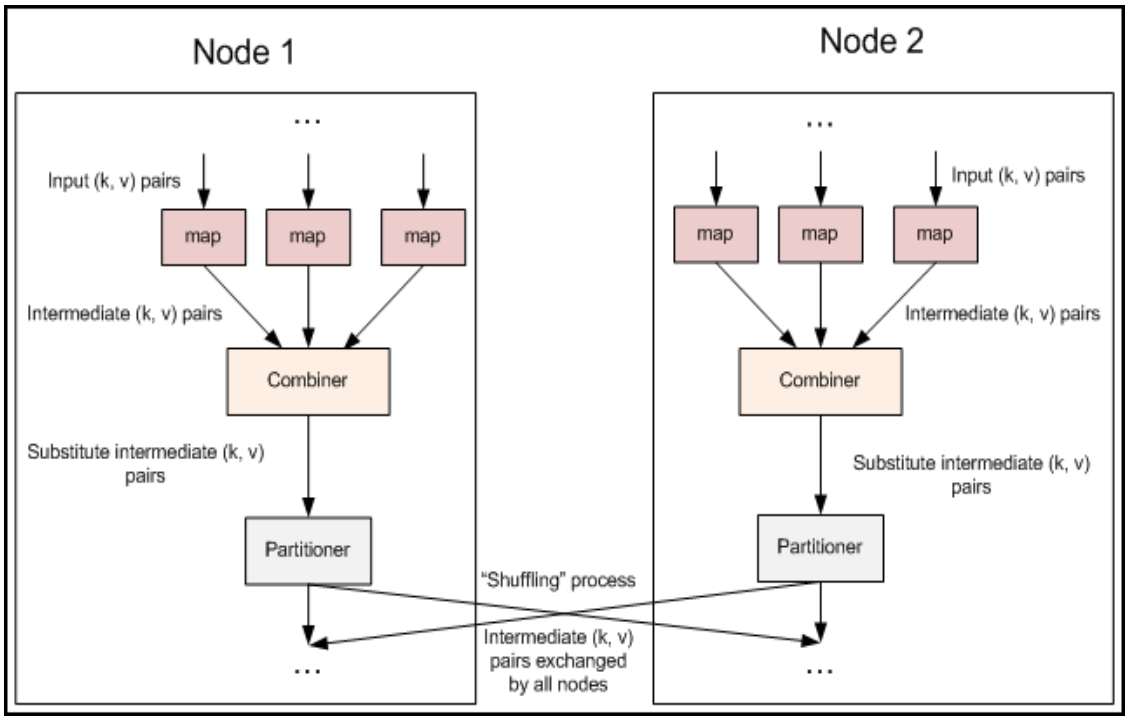


Figure 3. The MapReduce Process with a Combiner

It should be noted that a Combiner does not take the place of a Reducer. One site may emit <"cat", 12> and another site may emit <"cat", 19>. The sort and Reduce stages are needed to achieve an output of <"cat", 31>.

Hadoop

Hadoop is Apache's public domain equivalent of Google's MapReduce. It is a large-scale distributed batch processing infrastructure that can scale to hundreds or thousands of computers. It efficiently distributes large amounts of work across many machines. Hadoop has been shown to scale to terabytes and even to petabytes of data! In general, it operates on data sets that will not fit onto a single computer's hard drive or memory. To address this, it also provides a specialized distributed file system called the Hadoop Distributed File System (HDFS), as described above. HDFS is the public domain equivalent of Google's GFS.

General Distributed Processing

In general, distributed data processing approaches must overcome many issues. In large distributed processing systems, processor failures are expected and common. The systems need to manage them accordingly. Other issues that they face are that they may need to deal with multiple software versions, which is common, and that the processors may become desynchronized, in which case the system must employ strategies that resynchronize them. In addition, distributed data processing systems must deal with network failures and manage network congestion to ensure that the processing is completed in reasonable time.

Distributed data processing systems in general must deal with memory issues. They must recover gracefully from disk crashes, redistribute data when the disk space is low on one disk, and must address corrupt data. In addition, they must handle cases when locks are not released properly and when resource contention forces deadlocks. For transaction-oriented applications, distributed systems must deal with concurrency control and recovery issues. They also have to deal with security issues.

From the perspective of computational complexity, general distributed data processing systems must deal with cost-based operations and optimizations, much like the query processing issues discussed earlier in the course. Distributed systems must plan for executing distributed processes by taking into account processor speeds, memory access speeds, network communications costs including bandwidth and latency, as well as memory size issues. Distributed systems must manage intermediate data sizes, memory size constraints, network routing and re-routing, bandwidth, and re-routing tasks under resource failures and degradations. In all, generally distributed computing architectures have to address many issues to run efficiently, if at all.

Hadoop Approach

In contrast, the Hadoop architecture is designed to address a meaningful subset of distributed algorithms that does not require it to address many of the issues described in the previous paragraphs. It executes only MapReduce algorithms. These algorithms operate on massive data sets in two phases. The first phase applies an algorithm that can run in parallel on different shards of an input file. The results are sorted according to intermediate key values. The second phase applies a parallel reduction algorithm to the intermediate sorted values. While this class of algorithm is only a subset of distributed algorithms, it has distinct advantages over general distributed algorithms. It is a simple programming model that automatically and efficiently distributes and manages large data sets. It also does not need to address many of the issues of the last section.

Hadoop achieves its efficiency through an architecture that isolates the processes. Map jobs and reduce jobs run independently and in-parallel without explicit inter-process communications. Hadoop limits the communication among processors, which actually makes the framework more reliable. Conventional distributed systems' algorithms explicitly send byte streams from process to process; Hadoop communication is performed *implicitly* within the MapReduce paradigm. User defined processes do not send data from one process to another at all.

Hadoop achieves reliability through distribution and processor independence. Node failures are addressed by restarting tasks on other machines. This is possible because processes do not communicate with one another. Therefore, their processing does not rely on messages exchanged by user programs. Furthermore, since data is not modified, transaction processing issues do not need to be addressed. This means that Hadoop does not have to implement rollback or checkpoint procedures when restarting Map or Reduce processes. Furthermore, other nodes can continue to operate as though nothing went wrong because they do not rely on data from any of the parallel processes. It is only the Hadoop system itself that must deal with the bookkeeping of partially restarting processes.

Hadoop's design supports linear scaling. Almost no work is required to scale up to adding many machines. The Hadoop system manages the data and hardware resources in the same way it manages all other nodes. Furthermore, because of the processor-independent architecture and because no messages are exchanged among Map (and among Reduce) processes, Hadoop provides performance growth proportionate to the number of machines available.

The Hadoop system should only be applied to applications on large data sets. The overhead of Hadoop may be too much for small amounts of data and a small number of nodes. Other distributed programming paradigms such as MPI (Message Passing Interface) may perform much better with smaller data. However, adding new hardware to MPI does not scale linearly because of the complexity in communication introduced by more processors. To make programs scale on larger MPI systems, programs may need to be rewritten when scaling from ten to one hundred or even one thousand machines.

Hadoop MapReduce Example

Hadoop MapReduce is a platform specifically geared to embarrassingly parallel problems. One such problem occurred in the early days of search engines. Many search engines at the time evaluated the content of web pages to determine the best set of keywords that described the page. When a person entered search terms into a search text box, the search engine would match the search terms against the pages whose keywords would best match the search terms. Results were mixed. Often, people had to scroll many pages to find the results they were looking for. To rank pages better, most search engines tried hard to evaluate web pages better to best match search keywords.

Google had a wonderful insight into this problem. They incorporated a "popularity" metric into their search. Part of the returned page order (page rank) would depend on a web pages' popularity with respect to the search terms. For example, if someone were looking for a page about butterflies, Google would account for the butterfly web pages that were the most popular on the web. A good way to measure the popularity is to count the number of other web pages that are pointing to a specific web page. For instance, a butterfly web page with a 1000 other pages pointing to it will generally rank higher than a butterfly web page that has only 10 other pages pointing to it.

The challenge, of course, is that web pages only contain URLs that point to other pages. They have no knowledge of other web pages that point to it. Calculating the popularity of a page, therefore, is a two-step process. First, for each page on the

entire Internet, determine which pages to which they point. This is easy because each page contains a list of URLs to which they point. The result is a giant list of URL pairs in the form [fromURL, toURL]. Second, we reverse the list into a giant list in the form of [toURL, fromURL]. For each toURL, we count the number of corresponding from URLs. This number provides a popularity measure for each of the toURL on the Internet.

When the Internet was small, Google could read all the pages on the Internet onto a single computer and run the page rank algorithm. However, as the Internet grew in size, it was no longer possible to perform this algorithm in a reasonable amount of time. To address this issue, MapReduce was born. They created the architecture described in the previous sections of this document. Google read the pages of the Internet into massively parallel processors and disks. They wrote MapReduce algorithms to calculate page rank. With the MapReduce architecture, they were able to calculate page rank much faster.

Straightforward (non-MapReduce) Algorithm

The page rank pseudocode described in the previous paragraphs is given here:

```
Array targetToSourceArray[target, numSources];

for each (sourceURL) {
        for each (targetURL in sourceURL) {
                find index of targetURL in targetToSourceArray;
                if (index == -1) {
                        index =
                        targetToSourceArray.addTarget(targetURL);
                }
                targetToSourceArray.incrementSource(targetURL, 1);
        }
}
```

There are two loops. The outer loop iterates over all source URLs and finds the target URLs on that source URL page. Then, for each target URL in the source, it creates an entry in the array targetToSourceArray. If the targetURL does not exist, it creates an entry for the new targetURL and a count of 1. If the targetURL already exists in the array, the code increments the count for that targetURL by 1. Notice that the inner *for* loop can be run in parallel! This gives us a hint about writing both the mapper and reducer in a MapReduce architecture. The elements not in gray are part of the Map code.

```
Array targetToSourceArray[target, numSources];

for each (sourceURL) {
        for each (targetURL in sourceURL) {
                find index of targetURL in targetToSourceArray;
                if (index == -1) {
                        index =
                        targetToSourceArray.addTarget(targetURL);
                }
                targetToSourceArray.incrementSource(targetURL, 1);
```

```
        }
}
```

The elements in gray are part of the Reduce code where results are aggregated.

MapReduce – Map and Reduce Algorithms

The map algorithm becomes simply:

```
map(key: null, value: source):
        for each (target in source) {
                write(target, 1);
}
```

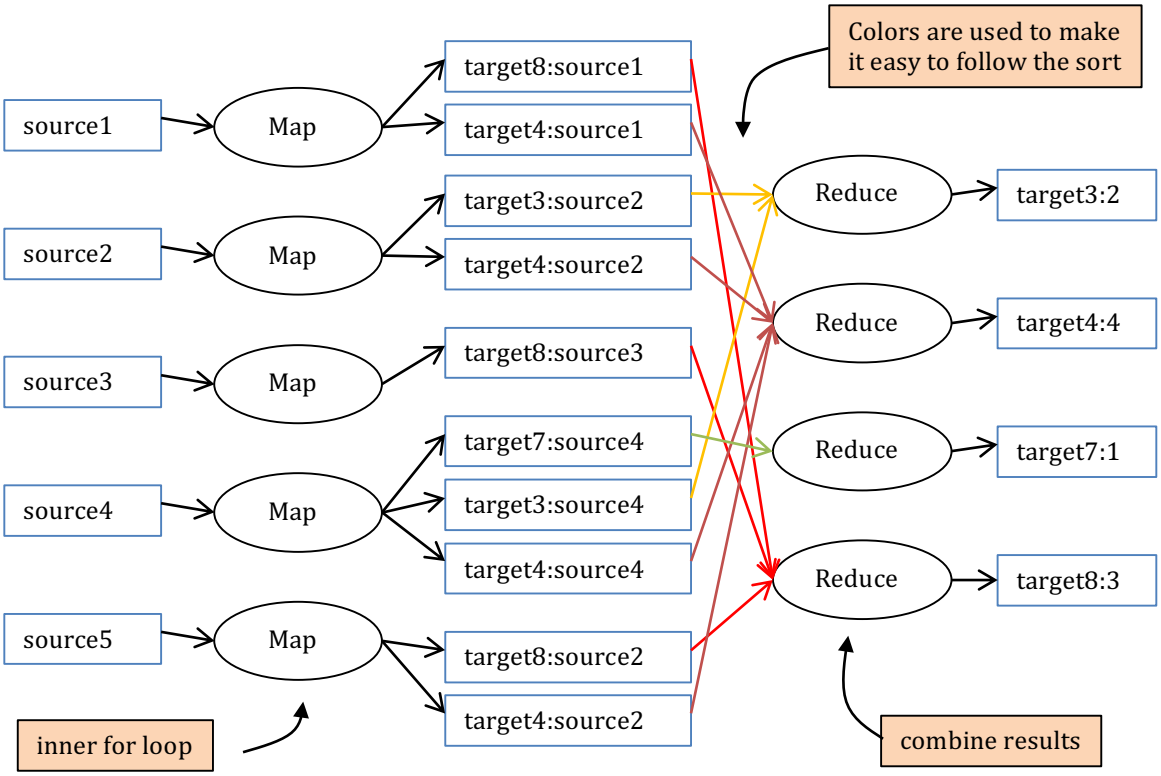
For each targetURL in the sourceURL, write an intermediate array that gives a count of 1 to each targetURL. The intermediate array is sorted by targetURL, placing all targetURLs of the same value adjacent to each other.

The reduce algorithm becomes simply:

```
reduce(key: target, value: number):
        sum = 0;
        for each (target) {
                sum += number;
        }
write(target, sum);
```

Here, each unique set of targetURLs and their corresponding numbers (in our case, the numbers are just 1) are passed to the reduce code. The reduce code just adds up the number of times a targetURL was found in a sourceURL. This number is used to determine its popularity with respect to other targetURLs. Note that the reduce code can be run in parallel and multiple reduce algorithms can be run in parallel

Below is a diagram that illustrates the MapReduce process.



Map Code Written in Java

The following code is the actual Java map code:

```
import java.util.StringTokenizer;
import java.net.URL;

// KEYIN, VALUEIN, KEYOUT, VALUEOUT
public class URLMapper
   extends Mapper<LongWritable, Text, Text, IntWritable> {

      // Context contains CONTEXT of Job, Mapper, and Task
      @Override
      public void map(LongWritable key, Text value,
                      Context context)
            throws IOException, InterruptedException {

            String textString;
            StringTokenizer st = new stringTokenizer(value, " ");
            while (st.hasMoreElements()) {
                  textString = st.nextElement();
                  if (isURL(textString)) {
                        // Context contains CONTEXT of Job,
                        // Mapper, and Task
                        context.write (new Text(textString),
```

```
                                                    new IntWritable(1));
                        }
                }
        }

        // Check for a valid URL format
        isURL (String text) {
                try {
                        URL url = new URL(text);
                        return true:
                }
                catch  (MalformedURLException mue) {
                        return false;
                }
        }
}
```

Reduce Code Written in Java

The following code is the actual Java reduce code:

```
// Key/Value pair
public class URLReducer
        extends Reducer<Text, IntWritable, Text, IntWritable> {

        // Key/Value and Context
        @Override
        public void reduce(Text key, Iterable<IntWritable> values,
                                Context context)
                throws IOException, InterruptedException {

                int numSources = 0;
                for (IntWritable value : values) {
                        Integer intObject = new Integer(value.toString());
                        numSources += intObject.intValue();
                }

                // Write Key/Value pair
                context.write(key, new IntWritable(numSources));
        }
}
```

Putting it Together

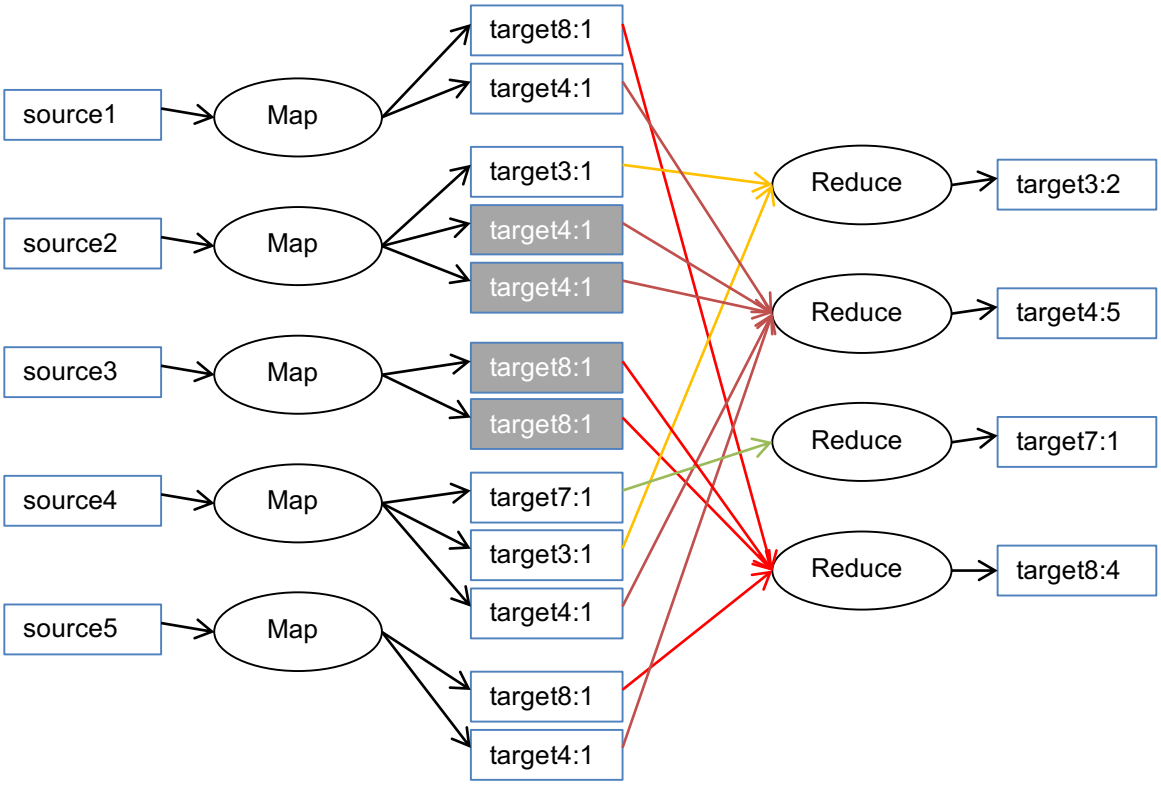
Now, we need to create a driver that puts the Map and Reduce code together. We must set up the job; define the input and output files set the mapper, reduce, and optionally the combiner (to be discussed soon), define the output key values, and run the job.

```
public class URLDriver extends Configured implements Tool {
```

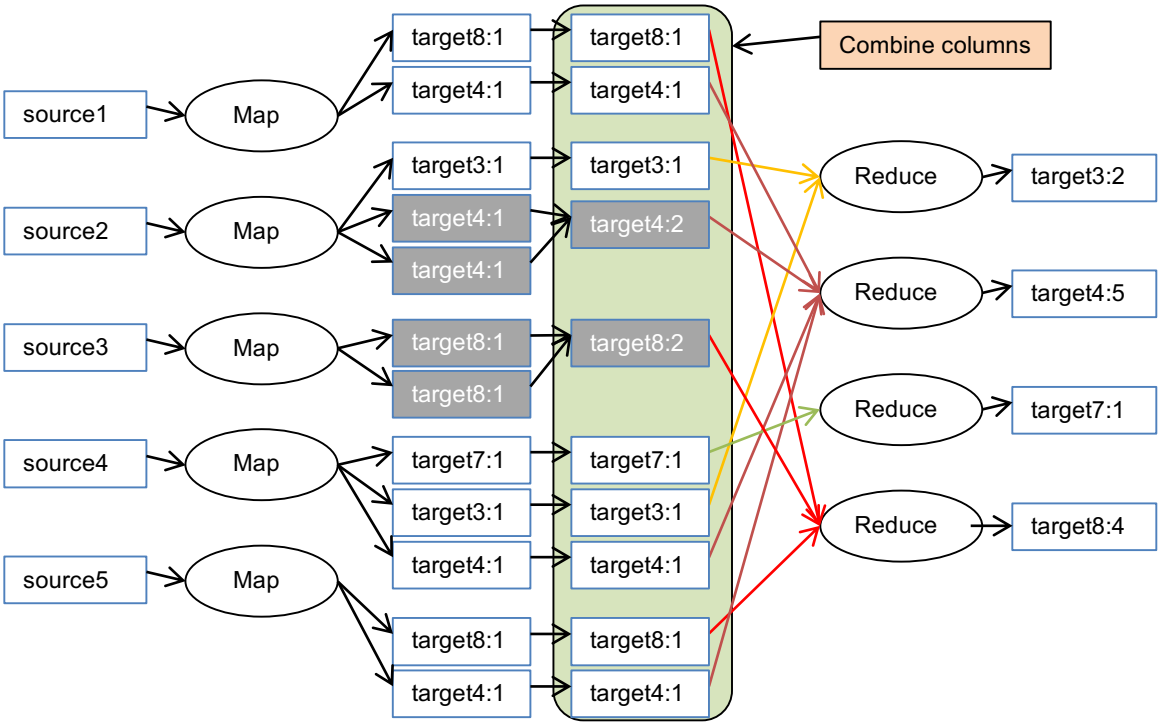
```
        @Override
        public int run(String[] args) throws Exception {
                if (args.length != 2) {
                    System.err.printf("Usage: %s [generic options] <input>"
                                      + " <output>\n",
                                      getClass().getSimpleName());
                    ToolRunner.printGenericCommandUsage(System.err);
                    return -1;
                }

                // getConf() gets the Hadoop configuration
                Job job = new Job(getConf(), "URL Ranker");

                // Set the Jar by finding where a given class came from
                job.setJarByClass(getClass());

                // Set input and output paths
                FileInputFormat.addInputPath(job, new Path(args[0]));
                FileOutputFormat.setOutputPath(job, new Path(args[1]));

                // Set mapper, combiner, and reducer classes
                job.setMapperClass(URLMapper.class);
                job.setCombinerClass(URLReducer.class);
                job.setReducerClass(URLReducer.class);

                // Set output KEY/VALUE pair types
                job.setOutputKeyClass(Text.class);
                job.setOutputValueClass(IntWritable.class);

                // Run the job and return
                return job.waitForCompletion(true) ? 0 : 1;
        }

        // Finally, the main() code
        public static void main(String[] args) throws Exception {
                int exitCode = ToolRunner.run(new URLDriver(), args);
                System.exit(exitCode);
        }
}
```

Combiners

Combiners are used to optimize the MapReduce process. In the diagram below, notice that the combination [target4:1] repeats twice as a result of the second map and twice as a result of the third map. The repetition in general puts more burden on the intermediate result memory size and on the reducers that process the intermediate pairs.



Adding combiners removes this load. After the Map process, but before storing the results in the intermediate memory, a combiner algorithm can be applied to reduce this size. Often, the combiner is the same code as the reducer. Look at the effect of adding



Notice that the targets that can be combined before stored in the intermediate memory are combined. This reduces the size of the intermediate memory as well as the size for input to the reducers.

Now What?

To run the code above, we need to:

1. Configure Hadoop
2. Test the code
3. Run the code
4. Monitor the results
5. Debug the code

The following sections will describe each step of this process.

Configure Hadoop

Hadoop can be configured to run in three modes. The first is standalone (or local) mode. There are no daemons that run – everything runs in a single JVM. This is the best mode for development because it is easy to test and debug. The second is pseudo-distributed mode. Hadoop daemons run on the local machine simulating a small-scale cluster. The third is fully distributed mode where Hadoop daemons run on a cluster of machines.

Each of these modes is represented in XML files with the correct property values set. These are represented in the following matrices.

Mode	Configuration File
Standalone	hadoop-local.xml
Pseudo-distributed	hadoop-localhost.xml
Fully distributed	hadoop-cluster.xml

Component	Property	Standalone	Pseudo-distributed	Fully distributed
Common	fs.default.name	file:///	hdfs://localhost/	hdfs://namenode/
HDFS	dfs.replication	N/A	1	3
MapReduce	mapred.job.tracker	local	localhost:8021	jobtracker:8021

For local mode, the **hadoop-local.xml** file should be configured as follows:

```
<?xml version="1.0"?>
	<configuration>
		<property>
			<name>fs.default.name</name>
			<value>file:///</value>
		</property>
		<property>
			<name>mapred.job.tracker</name>
			<value>local</value>
		</property>
</configuration>
```

For pseudo-distributed mode, the **hadoop-localhost.xml** file should be configured as follows:

```
<?xml version="1.0"?>
	<configuration>
		<property>
			<name>fs.default.name</name>
			<value>hdfs://localhost/<value>
		</property>
		<property>
			<name>mapred.job.tracker</name>
			<value>localhost:8021</value>
		</property>
</configuration>
```

For distributed mode, the **hadoop-cluster.xml** file should be configured as follows:

```
<?xml version="1.0"?>
	<configuration>
		<property>
			<name>fs.default.name</name>
			<value>hdfs://namenode/<value>
		</property>
		<property>
			<name>mapred.job.tracker</name>
			<value>jobtracker:8021</value>
		</property>
</configuration>
```

There are several calls for accessing configuration files. To instantiate a new configuration object, call the following in the main MapReduce code:

```
Configuration conf = new Configuration();
```

To read a configuration file, call addResource().

```
conf.addResource("hadoop-local.xml");
```

To get the value of a configuration, call get().

```
String tracker = conf.get("mapred.job.tracker");
```

To set configuration within the code, call set().

```
conf.set("fs.default.name", "file:///");
conf.set("mapred.job.tracker", "local");
```

In code, it either looks like:

```
Job job = new Job(getConf(), "URL Ranker");
```

or:

```
Configuration conf = new Configuration();

conf.set("fs.default.name", "file:///");
conf.set("mapred.job.tracker", "local");

Job job = new Job(conf, "URL Ranker");
```

Running Hadoop on the Command Line

The standard call to Hadoop on the command line is:

```
% hadoop URLDriver –conf conf/hadoop-local.xml \
    input/webpages output
```

To set the file system to be the local file system and to set the Job Tracker (we will see this shortly) to local, the command line call is:

```
% hadoop URLDriver –fs file:/// jt local \
    input/webpages output
```

To run on the cluster, the following command line call is invoked:

```
% hadoop jar URL.jar URLDriver –conf conf/hadoop-cluster.xml \
    input/webpages output
```

Tracking Progress of a MapReduce Job

The method waitForCompletion() polls the cluster periodically for progress. It writes a line summarizing the progress of the mappers and reducers whenever either changes. An example:

```
09/04/11 08:15:52 INFO mapred.FileInputFormat: Total input paths to process : 101
09/04/11 08:15:53 INFO mapred.JobClient: Running job: job_200904110811_0002
```

```
09/04/11 08:15:54 INFO mapred.JobClient: map 0% reduce 0%
09/04/11 08:16:06 INFO mapred.JobClient: map 28% reduce 0%
09/04/11 08:16:07 INFO mapred.JobClient: map 30% reduce 0%
...
09/04/11 08:21:36 INFO mapred.JobClient: map 100% reduce 100%
09/04/11 08:21:38 INFO mapred.JobClient: Job complete: job_200904110811_0002
09/04/11 08:21:38 INFO mapred.JobClient: Counters: 19
09/04/11 08:21:38 INFO mapred.JobClient: Job Counters
09/04/11 08:21:38 INFO mapred.JobClient: Launched reduce tasks=32
09/04/11 08:21:38 INFO mapred.JobClient: Rack-local map tasks=82
09/04/11 08:21:38 INFO mapred.JobClient: Launched map tasks=127
09/04/11 08:21:38 INFO mapred.JobClient: Data-local map tasks=45
09/04/11 08:21:38 INFO mapred.JobClient: FileSystemCounters
09/04/11 08:21:38 INFO mapred.JobClient: FILE_BYTES_READ=12667214
09/04/11 08:21:38 INFO mapred.JobClient: HDFS_BYTES_READ=33485841275
09/04/11 08:21:38 INFO mapred.JobClient: FILE_BYTES_WRITTEN=989397
09/04/11 08:21:38 INFO mapred.JobClient: HDFS_BYTES_WRITTEN=904
09/04/11 08:21:38 INFO mapred.JobClient: Map-Reduce Framework
09/04/11 08:21:38 INFO mapred.JobClient: Reduce input groups=100
09/04/11 08:21:38 INFO mapred.JobClient: Combine output records=4489
09/04/11 08:21:38 INFO mapred.JobClient: Map input records=1209901509
09/04/11 08:21:38 INFO mapred.JobClient: Reduce shuffle bytes=19140
09/04/11 08:21:38 INFO mapred.JobClient: Reduce output records=100
09/04/11 08:21:38 INFO mapred.JobClient: Spilled Records=9481
09/04/11 08:21:38 INFO mapred.JobClient: Map output bytes=10282306995
09/04/11 08:21:38 INFO mapred.JobClient: Map input bytes=274600205558
09/04/11 08:21:38 INFO mapred.JobClient: Combine input records=1142482941
09/04/11 08:21:38 INFO mapred.JobClient: Map output records=1142478555
09/04/11 08:21:38 INFO mapred.JobClient: Reduce input records=103
```

Job Tracker

Hadoop comes with a web user interface for viewing job information including the job progress during the run and the job statistics and logs after the run. The URL is:

http://jobtracker-host:50030/

The first section contains the details of the Hadoop installation including the compiler version number and the current state of the job tracker. Next, it presents a summary of the cluster including capacity and utilization, number of maps and reduces currently running, the total number of job submissions, the number of tasktracker nodes currently available, the cluster capacity, the number of available slots per node on average, the number of tasktrackers that have blacklisted (due to failure). Next, it displays the job scheduler that is running. One can click through to see the job queues. Next, it displays details of the running, completed and failed jobs. Finally, the bottom of the page displays a link to the job tracker's logs and history. By default, 100 jobs are displayed. Here is an example:

ip-10-250-110-47 Hadoop Map/Reduce Administration

Quick Links

State: RUNNING
Started: Sat Apr 11 08:11:53 EDT 2009
Version: 0.20.0, r763504
Compiled: Thu Apr 9 05:18:40 UTC 2009 by ndaley
Identifier: 200904110811

Cluster Summary (Heap Size is 53.75 MB/888.94 MB)

Maps	Reduces	Total Submissions	Nodes	Map Task Capacity	Reduce Task Capacity	Avg. Tasks/Node	Blacklisted Nodes
53	30	2	11	88	88	16.00	0

Scheduling Information

Queue Name	Scheduling Information
default	N/A

Filter (Jobid, Priority, User, Name)
Example: 'user:smith 3200' will filter by 'smith' only in the user field and '3200' in all fields

Running Jobs

Jobid	Priority	User	Name	Map % Complete	Map Total	Maps Completed	Reduce % Complete	Reduce Total	Reduces Completed	Job Scheduling Information
job_200904110811_0002	NORMAL	root	Max temperature	47.52%	101	48	15.25%	30	0	NA

Completed Jobs

Jobid	Priority	User	Name	Map % Complete	Map Total	Maps Completed	Reduce % Complete	Reduce Total	Reduces Completed	Job Scheduling Information
job_200904110811_0001	NORMAL	gonzo	word count	100.00%	14	14	100.00%	30	30	NA

Failed Jobs

none

Local Logs

Log directory, Job Tracker History

Hadoop, 2009.

The job page is displayed by clicking on a JobID. The top section displays the job owner, job name, and run time. The next section displays the job progress including the total number of map and reduce tasks; the pending, running, complete, and failed tasks; the graphs representing progress; and the reduce completion graph. The job counters are displayed in the middle of the page. For example:

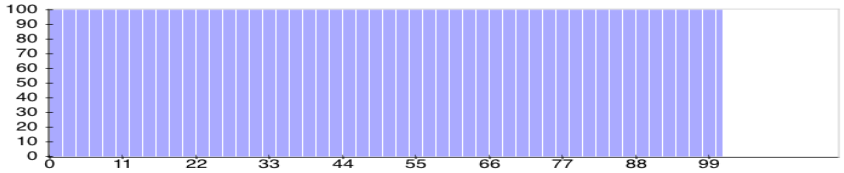
Hadoop job_200904110811_0002 on ip-10-250-110-47

User: root
Job Name: Max temperature
Job File: hdfs://ip-10-250-110-47.ec2.internal/mnt/hadoop/mapred/system/job_200904110811_0002/job.xml
Job Setup: Successful
Status: Running
Started at: Sat Apr 11 08:15:53 EDT 2009
Running for: 5mins, 38sec
Job Cleanup: Pending

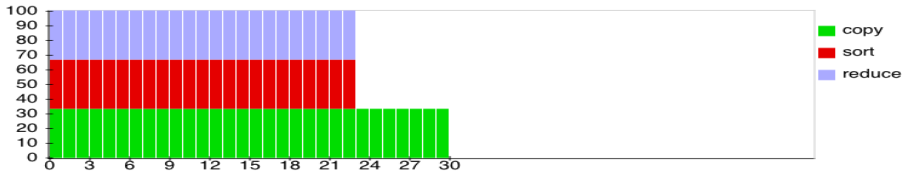
Kind	% Complete	Num Tasks	Pending	Running	Complete	Killed	Failed/Killed Task Attempts
map	100.00%	101	0	0	101	0	0 / 26
reduce	70.74%	30	0	13	17	0	0 / 0

	Counter	Map	Reduce	Total
Job Counters	Launched reduce tasks	0	0	32
	Rack-local map tasks	0	0	82
	Launched map tasks	0	0	127
	Data-local map tasks	0	0	45
FileSystemCounters	FILE_BYTES_READ	12,665,901	564	12,666,465
	HDFS_BYTES_READ	33,485,841,275	0	33,485,841,275
	FILE_BYTES_WRITTEN	988,084	564	988,648
	HDFS_BYTES_WRITTEN	0	360	360
Map-Reduce Framework	Reduce input groups	0	40	40
	Combine output records	4,489	0	4,489
	Map input records	1,209,901,509	0	1,209,901,509
	Reduce shuffle bytes	0	18,397	18,397
	Reduce output records	0	40	40
	Spilled Records	9,378	42	9,420
	Map output bytes	10,282,306,995	0	10,282,306,995
	Map input bytes	274,600,205,558	0	274,600,205,558
	Map output records	1,142,478,555	0	1,142,478,555
	Combine input records	1,142,482,941	0	1,142,482,941
	Reduce input records	0	42	42

Map Completion Graph - close



Reduce Completion Graph - close



Go back to JobTracker

Hadoop, 2009.

Retrieving Results

The files are placed in the Hadoop Data File System (HDFS) **output-dir** directory. The files are named **part-r-00000** to **part-r-xxxxx**. There are two methods for retrieving results.

First:

```
% hadoop fs -getmerge output-dir output-local
% sort output-local | less
```

Second:

```
% hadoop fs -cat output-dir/*
```

Debugging

Suppose we do not think we are getting all the URL's that we are supposed to be getting. How do we determine what is wrong? One method of debugging is to set counters in the code. These counters are displayed on the Job Page in the Counters section.

Let's modify the Map job above and add counters. Additions to the code are shown in red.

```
import java.util.StringTokenizer;
import java.net.URL;

// KEYIN, VALUEIN, KEYOUT, VALUEOUT
public class URLMapper
   extends Mapper<LongWritable, Text, Text, IntWritable> {

      enum BadURL {
            BAD_PARSE
      }

      // Context contains CONTEXT of Job, Mapper, and Task
      @Override
      public void map(LongWritable key, Text value,
                        Context context)
            throws IOException, InterruptedException {

            String textString;
            StringTokenizer st = new stringTokenizer(value, " ");
            while (st.hasMoreElements()) {
                  textString = st.nextElement();
                  if (isURL(textString)) {
                        // Context contains CONTEXT of Job,
                        // Mapper, and Task
                        context.write (new Text(textString),
                                          new IntWritable(1));
                  }
                  else {
                     System.err.println("Ignoring possibly corrupt"
                        + "input: " + textString);
                     context.getCounter(BadURL.BAD_PARSE).
                        increment(1);
                  }
            }
      }
```

```
        // Check for a valid URL format
        isURL (String text) {
                try {
                        URL url = new URL(text);
                        return true:
                }
                catch  (MalformedURLException mue) {
                        return false;
                }
        }
}
```

Fault Tolerance

Fault tolerance is achieved through Hadoop restarting tasks. TaskTrackers (task nodes) are in constant communication with the head node of the system called the JobTracker. If a TaskTracker does not communicate with the JobTracker for a period of time (by default, 1 minute), the JobTracker assumes that the TaskTracker has crashed.

If the job is in the mapping phase, other TaskTrackers will be asked to re-execute the Map tasks run by the failed TaskTracker. If the job is in the reducing phase, other TaskTrackers will re-execute all Reduce tasks run by the failed TaskTracker.

Fault tolerance is easily achieved because the Map and Reduce processes are independent and side effect free. If Mappers and Reducers were to communicate with each other or with other processes, it would be necessary to understand different states of the distributed processes to restart them. This would be very complex and error prone. However, since the MapReduce paradigm is side-effect free because they do not send messages to each other, achieving fault tolerance is as simple as restarting failed processes on other nodes.

Conclusion

The utility cloud model is a business model for leasing computer resources. Utility Cloud systems enable people and organizations to rent any computer resource configuration under the terms that are mutually agreed upon by both consumers and the providers. They enable small organizations to get their computer systems up and running without investing in hardware and software systems as well as investing in IT staff. They enable large organizations to amortize costs among a large number of hardware and software systems as well as among IT staff.

The MapReduce paradigm is a specific approach to performing analysis of large data stored across many data systems. Both Google's MapReduce and Apache's Hadoop architectures enable people to write simple MapReduce algorithms to run parallel processes across large data sets to analyze the data. The programming paradigm is simple and the system handles the complexities of managing many distributed Map

and Reduce processes for multiple user MapReduce programs. It enables fast processing of algorithms that do not scale under conventional serial architectures.