CSx55: Distributed Systems [Spark]

Spark: What fuels it?

Memory residency, of course
With frugal I/O that it must reinforce

How? By …
Procrastinating (through lazy evaluations)
Avoiding repeated sweeps
And doing it only as a last resort

Shrideep Pallickara
Computer Science
Colorado State University

Frequently asked questions from the previous class survey

- Can datanodes be "promoted" to being a "namenode" … like what happens in super-peer networks?
- How do you know if the checksums itself have been corrupted?
- Why do bit flip occurrences show up more commonly in data centers?
- In distributed copy (distcp), how come a sole destination is "pulling" data? Shouldn't mappers have data locality?
- If a file size increases after compression, does it not defeat the very purpose of compression?

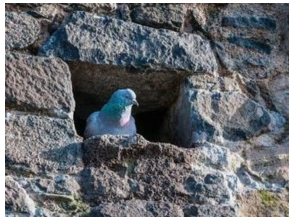
Topics covered in this lecture

- HDFS Wrap-up
- Spark
  - Software stack
  - Interactive shells in Spark
  - Core Spark concepts

HDPS Compression Wrap-up

Compression formats that can be used with Hadoop

Compression format	Tool	Algorithm	Filename extension	Splittable?
DEFLATE	N/A	DEFLATE	*.deflate*	No
Gzip	Gzip	DEFLATE	.gz	No
Bzip2	Bzip2	Bzip2	.bz2	Yes
LZO	Lzop	LZO	.lzo	No*
Snappy	N/A	Snappy	.snappy	No



Pigeonhole principle

Compression Algorithms

- Exhibit a **space-time** trade-off
  - Faster compression/decompression speeds usually result in smaller space savings
- Tools give some control over this trade-off at compression time
  - 9 different options
  - -1 means optimize for speed
  - -9 means optimize for space

Compression characteristics

- gzip is a *general purpose* compressor
  - Middle of the space/time trade-off
- bzip2 compresses more effectively than gzip
  - But it is slower
  - bzip2 decompression speed is faster than its compression speed
    - But slower than other formats still
- LZO and Snappy optimize for speed
  - Order of magnitude faster but less effective compression than gzip

A **codec** is the implementation of a compression-decompression algorithm in Hadoop

Compression format	Hadoop CompressionCodec
DEFLATE	`org.apache.hadoop.io.compress.DefaultCodec`
gzip	`org.apache.hadoop.io.compress.GzipCodec`
bzip2	`org.apache.hadoop.io.compress.BZip2Codec`
LZO	`com.hadoop.compression.lzo.LzopCodec`
Snappy	`org.apache.hadoop.io.compress.SnappyCodec`

CompressionCodec

- To compress data being written to an output stream
  - Use `codec.createOutputStream(OutputStream out)`

- To decompress data being read from an input stream
  - Use `codec.createInputStream(InputStream in)`

Using compression

```
public class StreamCompressor {

    public static void main(String[] args) throws Exception {
      String codecClassname = args[0];
      Class<?> codecClass = Class.forName(codecClassname);
      Configuration conf = new Configuration();
      CompressionCodec codec = (CompressionCodec)
        ReflectionUtils.newInstance(codecClass, conf);
      CompressionOutputStream out =
          codec.createOutputStream(System.out);
      IOUtils.copyBytes(System.in, out, 4096, false);
      out.finish();
    }
}
```

Compresses data read from standard input and writes it to standard output

Compression and input splits

- Let's look at an uncompressed file stored in HDFS
  - With an HFDS block size of 64 MB, a 1 GB file is stored as 16 blocks
  - MapReduce job will create 16 input splits
    - **Processed independently** as separate map tasks

If the gzip compressed file is 1 GB

- HDFS stores files as 16 blocks
- Creating a split for each block does not work
  - Impossible to start reading at an *arbitrary block* in the zip stream
  - Impossible for map task to read its split *independently of others*

Storing gzipped streams

- Gzip uses DEFLATE, which stores data as a **series** of *compressed blocks*
- The **start of each block is not distinguished** in a way that allows:
  - Reader positioned at arbitrary point in stream to advance to the beginning of the next block
    - There is **no self-synchronizing** with the stream
  - Gzip does not support splitting

HDFS does not split `gzip` files

- Single map will process 16 HDFS blocks
- Most of these blocks will not be local to the map
  - Loss of locality
  - Job is not granular … takes much longer to run

The same story plays out if you were dealing with LZO files, but …

- It is possible to *preprocess* LZO files using an indexer tool
- Build an **index** of split points

Bzip2

- This does provide a **synchronization marker** between blocks
  - 48-bit approximation of pi
- The marker is used to support splitting

Dealing with large, unbounded files [Log files]

1. Store the files uncompressed

2. Use compression format that supports
   - Splitting: Bzip2
   - Indexing to support splitting: LZO

3. Split the file into chunks in the application and compress each chunk separately
   - Choose chunk sizes such that the *compressed chunks* are approximately the <u>size of an HDFS block</u>

Using compression in MapReduce

- To compress the output of MapReduce job
  - In the job config set mapred.output.compress property to true
  - Use mapred.output.compression.codec to specify the codec
- Alternatively, we can do this using the `FileOutputFormat`

Using the FileOutputFormat

```
public class MaxTemperatureWithCompression {

  public static void main(String[] args) throws Exception {
       Job job = Job.getInstance();
       job.setJarByClass(MaxTemperature.class);
       FileInputFormat.addInputPath(job, new Path(args[0]));
       FileOutputFormat.setOutputPath(job, new Path(args[1]));
       job.setOutputKeyClass(Text.class);
       job.setOutputValueClass(IntWritable.class);

       FileOutputFormat.setCompressOutput(job, true);
       FileOutputFormat.setOutputCompressorClass(job, GzipCodec.class);

       job.setMapperClass(MaxTemperatureMapper.class);
       job.setCombinerClass(MaxTemperatureReducer.class);
       job.setReducerClass(MaxTemperatureReducer.class);
       System.exit(job.waitForCompletion(true) ? 0 : 1);
  }

}
```

Main reason why Hadoop does not use Java Serialization

- Deserialization creates new instance of each object being deserialized
- Writable objects can be (and are often) reused
- Large MapReduce jobs often serialize/deserialize billions of records
  - Savings from not having to allocate new objects is significant

Apache Spark

As distributed data analytics have grown common …

- Practitioners have sought **easier tools** for the task
- Apache Spark has emerged as one of the most popular
  - Extending and generalizing MapReduce

Spark: What is it?

- **Cluster computing platform**
  - Designed to be fast and general purpose
- Speed
  - Extends MapReduce to support more types of computations
    - Interactive queries, iterative tasks, and stream processing
- Why is speed important?
  - Difference between waiting for hours versus exploring data interactively

Spark: Influences and Innovations

- Spark has inherited parts of its API, design, and supported formats from other existing computational frameworks
  - Particularly DryadLINQ
- Spark's internals, especially how it handles failures, differ from many traditional systems
- Spark's ability to leverage **lazy evaluation** within memory computations makes it particularly unique

Where does Spark fit in the Analytics Ecosystem?

- Spark provides methods to process data in parallel that are **generalizable**
- On its own, Spark is ***not*** a data storage solution
  - Performs computations in Spark JVMs that last only for the duration of a Spark application
- Spark is used in tandem with:
  - A distributed storage system (e.g., HDFS, Cassandra, or S3)
    - To house the data processed with Spark
  - A cluster manager — to orchestrate the distribution of Spark applications across the cluster

Key enabling idea in Spark

- **Memory resident data**

- Spark loads data into the memory of worker nodes
  - Processing is performed on memory-resident data

A look at the memory hierarchy

Item	time	Scaled time in human terms (2 billion times slower)
Processor cycle	0.5 ns (2 GHz)	1 second
Cache access	1 ns (1 GHz)	2 seconds
Memory access	70 ns	140 seconds
Context switch	5,000 ns (5 μs)	167 minutes
Disk access	7,000,000 ns (7 ms)	162 days
Quantum	100,000,000 ns (100 ms)	6.3 years

Source: Kay Robbins & Steve Robbins. *Unix Systems Programming*, 2nd edition, Prentice Hall.

Spark covers a wide range of workloads

- Batch applications
- Iterative algorithms
- Queries
- Stream processing

- This has previously required multiple, independent tools

Running Spark

- You can use Spark from Python, Java, Scala, R, or SQL
- Spark itself is written in **Scala**, and runs on the Java Virtual Machine (JVM)
  - You can run Spark either on your laptop or a cluster, all you need is an installation of Java
- If you want to use the Python API, you will also need a Python interpreter (version 2.7 or later)
- If you want to use R, you will need a version of R on your machine

Spark integrates well with other tools

- Can run in Hadoop clusters
- Access Hadoop data sources, including Cassandra

At its core, Spark is a **computational engine**

- Spark is responsible for several aspects of applications that comprise
  - Many tasks across many machines (compute clusters)

- Responsibilities include:
  1. Scheduling
  2. Distributions
  3. Monitoring

The Spark Software Stack

The Spark stack

Spark SQL structured data	Spark Streaming real-time	Mlib & ML machine learning	GraphX Graph processing

Spark Core

Standalone Scheduler	YARN	Mesos

Benefits of tight integration [1/2]

- All libraries and higher-level components benefit from improvements at the lower layers
- E.g.: Spark's core engine adds optimization? SQL and ML libraries automatically speed-up as well

Benefits of tight integration [2/2]

- Biggest advantage is ability to build applications that **seamlessly combine different processing models**
- An application may use ML to classify data in real time as it is being ingested
  - Analysts can query this resulting data, also in real time, via SQL (e.g.: join data with unstructured log-files)

Spark Core

- **Basic functionality** of Spark
- Task scheduling, memory management, fault recovery, and interacting with storage systems
- Also, the API that defines Resilient Distributed Datasets (**RDD**s)
  - Spark's *main programming abstraction*
  - Represents collection of data items dispersed across many compute nodes
    - Can be manipulated concurrently (parallel)

Spark SQL

- Package for working with **structured data**
- Allows querying data using SQL and HQL (Hive Query Language)
  - Data sources: Hive tables, Parquet, and JSON
- Allows intermixing queries with programmatic data manipulations support by RDDs
  - Using Scala, Java, and Python

(Semi)structured data and Spark SQL

- Spark SQL defines an interface for a (semi)structured data type, called **DataFrames**
  - And a (semi)structured, typed version of RDDs called **Datasets**
- Spark SQL is a very important component for Spark performance
- Much of what can be accomplished with Spark Core can be done by leveraging Spark SQL

Spark Streaming

- Enables processing of **live streams** of data from sources such as:
  - Logfiles generated by production webservers
  - Messages containing web service status updates
- Uses the scheduling of the Spark Core for streaming analytics on **minibatches** of data
- Has a number of unique considerations, such as the *window sizes* used for batches

MLib

- Library that contains common machine learning functionality
- Algorithms include:
  - Classification, regression, clustering, and collaborative filtering
- Low-level primitives
  - Generic gradient descent optimization algorithm
- Alternatives?
  - Mahout, sci–kit learn, VW, WEKA, and R among others

What about Spark ML?

- Has existed since Spark 1.2
- Spark ML provides a higher-level API than MLlib
  - Goal is to allow users to more easily create practical machine learning **pipelines**
  - Spark MLlib is primarily built on top of RDDs and uses functions from Spark Core, while ML is **built on top of Spark SQL DataFrames**
- The plan originally was to move over to ML and deprecate MLlib

Graph X

- Library for manipulating graphs
- Graph-parallel computations
- Extends Spark RDD API
  - Create a **directed graph**, with arbitrary properties attached to each vertex and edge

Cluster Managers

- Spark runs over a variety of cluster managers
- These include:
  - Hadoop YARN
  - Apache Mesos
  - Standalone Scheduler
    - Included within Spark

Storage Layers for Spark

- Spark can create distributed datasets from any file stored in HDFS
- Plus, other storage systems supported by the Hadoop API
  - Amazon S3, Cassandra, Hive, HBase, etc.

INTERACTIVE SHELLS IN SPARK

Spark Shells

- Interactive [Python and Scala]
  - Similar to shells like Bash or Windows command prompt
- *Ad hoc* data analysis
- Traditional shells manipulate data using disk and memory on a single machine
  - Spark shells allow interaction with **data that is distributed** across many machines
  - Spark manages complexity of distributing processing

Several software were designed to run on the Java Virtual Machine

- Languages that compile to run on the JVM and can interact with Java software packages but are *not actually* Java

- There are a number of non-Java JVM languages
  - The two most popular ones used in real-time application development: **Scala** and **Clojure**

Scala

- Has spent most of its life as an academic language
  - Still largely developed at universities
  - Has a rich standard library that has made it appealing to developers of high-performance server applications
- Like Java, Scala is a strongly typed object-oriented language
  - Includes many features from functional programming languages that are not in standard Java
  - Interestingly, since version 8, Java now incorporates several of the more useful features of Scala and other functional languages.

What is functional programming?

- When a method is compiled by Java, it is converted to instructions called byte code and …
  - Then largely disappears from the Java environment
    - Except when it is called by other methods
- In a functional language, **functions are treated the same way as data**
  - Can be stored in objects similar to integers or strings, returned from functions, and passed to other functions

What about Clojure?

- Based on Lisp
- Javascript?
  - Name was a marketing gimmick
  - Closer to Clojure and Scala than it is to Java

The contents of this slide-set are based on the following references

- *Learning Spark: Lightning-Fast Big Data Analysis. 1st Edition. Holden Karau, Andy Konwinski, Patrick Wendell, and Matei Zaharia. O'Reilly. 2015. ISBN-13: 978-1449358624.* [Chapters 1-4]
- Karau, Holden; Warren, Rachel. High Performance Spark: Best Practices for Scaling and Optimizing Apache Spark. O'Reilly Media. 2017. ISBN-13: 978-1491943205. [Chapter 2]
- *Real-Time Analytics: Techniques to Analyze and Visualize Streaming Data.* Byron Ellis. Wiley. [Chapter 2]
- Chambers, Bill,Zaharia, Matei. Spark: The Definitive Guide: Big Data Processing Made Simple. O'Reilly Media. ISBN-13: 978-1491912218. 2018. [Chapters 1, 2, and 3].