Big Data Frameworks: Internals

Mohammad A. Hoque
mohammad.a.hoque@helsinki.fi
object WordCount {
  def main (args: Array[String]){  
    val driver = "spark://192.168.0.3:7077"
    val sc = new SparkContext(driver, "SparkWordCount")
    val numPartitions = 10
    val lines = sc.textFile("here"+"sometext.txt", numPartitions)
    val words = lines.flatMap(_.split(" "))
    val groupWords = words.groupBy { x => x }
    val wordCount = groupWords.map( x => (x._1,x._2.size))
    val result = wordCount.saveAsTextFile("there"+"wordcount")
  }
}
Spark Application Framework

• SparkContext initializes the application driver and gives the application execution to the driver.

• RDD is generated from the external data sources; such as HDFS.

• RDD goes through a number of Transformations; such a Map, flatMap, sortByKey, etc.

• Finally, the count/collect/save/take Action is performed, which converts the final RDD into an output for storing to an external source.
Spark Application Framework

- Machine A
- Machine B
- Machine C
- Machine D
- Machine E

Part. 1
Part. 2
Part. 3
Part. 4
Part. 5
RDD is generated from the input file stored in HDFS
• This is where the data for a MapReduce task is initially stored and the files typically reside in HDFS.

• The format of these files; text and binary
  • Text – Single Line (JSON)
  • Multi-Line (XML)
  • Binary file, with fixed size objects
- **FileInputFormat** is a class that
  - Selects the files or other objects that should be used for input.
  - Defines the *InputSplit* that break a file into tasks.
  - Provides a factory for *RecordReader* objects that read the file.
Hadoop comes with a number of FileInputFormats:

- **TextInputFormat**: the byte offset of a line is a `key` and the line is the `value`.
- **KeyValueInputFormat**: the text until the first tab is the key and the remaining is the value.
- **SequenceFileInputFormat**: Object files. Key and values are defined by the user.
• A MapReduce program applied to a data set, collectively referred to as a *Job*, is made up of several (possibly several hundred) tasks.

• An InputSplit describes a unit of work that comprises a single *map task* in a MapReduce program.

• A Map tasks may involve reading a whole file; they often involve reading only part of a file.
• Different File Input Formats break a file up into 64 MB chunks.

• Splitting a file allow multiple map tasks over a single file in parallel.

• If the file is very large, the performance can be improved significantly through such parallelism.
for (FileStatus file: files) {
    Path path = file.getPath();
    FileSystem fs = path.getFileSystem(job.getConfiguration());
    long length = file.getLength();
    BlockLocation[] blkLocations = fs.getFileBlockLocations(file, 0, length);
    if ((length != 0) && isSplitable(job, path)) {
        long blockSize = file.getBlockSize();
        long splitSize = computeSplitSize(blockSize, minSize, maxSize);
        long bytesRemaining = length;
        while (((double) bytesRemaining)/splitSize > SPLIT_SLOP) {
            int blkIndex = getBlockIndex(blkLocations, length-bytesRemaining);
            splits.add(new FileSplit(path, length-bytesRemaining, splitSize,
                              blkLocations[blkIndex].getHosts()));
            bytesRemaining -= splitSize;
        }
        if (bytesRemaining != 0) {
            splits.add(new FileSplit(path, length-bytesRemaining, bytesRemaining,
                              blkLocations[blkLocations.length-1].getHosts()));
        }
    } else if (length != 0) {
        splits.add(new FileSplit(path, 0, length, blkLocations[0].getHosts()));
    } else {
        splits.add(new FileSplit(path, 0, length, new String[0]));
    }
}
• The InputSplit defines a task, but does not describe how to access it.
• The RecordReader class actually loads the data from its source and converts it into (key, value) pairs suitable for reading by the Mapper.
• **LineRecordReader** treats each line of the input file as a new value. The key associated with each line is its byte offset in the file.

• The RecordReader is invoke repeatedly until the entire InputSplit has been consumed.

• Each invocation of the RecordReader leads to another call to the map() method of the Mapper.
public XmlRecordReader(FileSplit split, JobConf jobConf) throws IOException {
    startTag = jobConf.get("<article>").getBytes("utf-8");
    endTag = jobConf.get("<article>").getBytes("utf-8");
    start = split.getStart();
    end = start + split.getLength();
    Path file = split.getPath();
    FileSystem fs = file.getFileSystem(jobConf);
    fsin = fs.open(split.getPath());
    fsin.seek(start);
}

public boolean next(LongWritable key, Text value) throws IOException {
    if (fsin.getPos() < end) {
        if (readUntilMatch(startTag, false)) {
            try {
                buffer.write(startTag);
                if (readUntilMatch(endTag, true)) {
                    key.set(fsin.getPos());
                    value.set(buffer.getData(), 0, buffer.getLength());
                    return true;
                }
            } finally {buffer.reset();}
        }
    }
    return false;
}
RDD goes through a number of Transformations
object WordCount {
  def main (args: Array[String]){
    val driver = "spark://192.168.0.3:7077"
    val sc = new SparkContext(driver, "SparkWordCount")
    val numPartitions = 10
    val lines = sc.textFile("here"+"sometext.txt", numPartitions)
    val words = lines.flatMap(_.split(" "))
    val groupWords = words.groupBy { x => x }
    val wordCount = groupWords.map( x => (x._1,x._2.size))
    val result = wordCount.saveAsTextFile("there"+"wordcount")
  }
}
Spark Application Execution

– How does the spark submit the job to the worker?
  • (1) Map, (2) flatMap, (2) groupBy, (3) Map, Or
  • (1) Map -> flatMap, (2) groupBy, (3) Map

– How Many tasks per submission?
– Before submitting tasks of a job, how does the driver know about the resource information of the workers?
How does the spark submit the job to the workers?
Transformation

**Narrow**: All partitions of an RDD will be consumed by a single child RDD, no shuffling.
Wide: Shuffling takes place according to their key value.
DAG Scheduler Splits the graph according to the dependency and submits to the lower layer Scheduler.
DAG serves Fault tolerance

- If a partition is lost while executing a stage consisting of transformations with Narrow dependencies
  - It traces back and re-computes only the lost partition of the parent RDD, and
  - The responsible machine will take care.

- In the case of Wide dependencies, the lost partition can affect a lot of others
  - Spark mitigates this my persisting the last computed partitions before the shuffling takes place.

- There is also checkpoint API which enables to persist RDD on desired transformation.
Part. 1 → RDD → Part. 2

Map → Part. 1 → RDD → Part. 2 → Map

Filter → Part. 1 → RDD → Part. 2 → Filter

Persist

Narrow

Wide

Result
INFO [task-result-getter-3] (Logging.scala:59) - Finished task 27.0 in stage 0.0 (TID 27) in 2455 ms on localhost (28/43)
INFO [Executor task launch worker-1] (Logging.scala:59) - Finished task 28.0 in stage 0.0 (TID 28). 4565 bytes result sent to driver
INFO [sparkDriver- akka.actor.default-dispatcher-2] (Logging.scala:59) - Starting task 29.0 in stage 0.0 (TID 29, localhost, PROCESS_LOCAL, 1650 bytes)
INFO [Executor task launch worker-1] (Logging.scala:59) - Running task 29.0 in stage 0.0 (TID 29)

INFO [task-result-getter-0] (Logging.scala:59) - Finished task 28.0 in stage 0.0 (TID 28) in 2084 ms on localhost (29/43)
INFO [Executor task launch worker-1] (Logging.scala:59) - Finished task 29.0 in stage 0.0 (TID 29). 4728 bytes result sent to driver
INFO [sparkDriver- akka.actor.default-dispatcher-2] (Logging.scala:59) - Starting task 30.0 in stage 0.0 (TID 30, localhost, PROCESS_LOCAL, 1650 bytes)

INFO [Executor task launch worker-1] (Logging.scala:59) - Running task 9.0 in stage 1.0 (TID 52)
INFO [Executor task launch worker-1] (Logging.scala:59) - Partition rdd_9_9 not found, computing it
INFO [Executor task launch worker-1] (Logging.scala:59) - ensureFreeSpace(16880) called with curMem=471734, maxMem=4123294433
INFO [Executor task launch worker-1] (Logging.scala:59) - Block rdd_9_9 stored as values in memory (estimated size 16.5 KB, free 3.8 GB)
INFO [sparkDriver- akka.actor.default-dispatcher-14] (Logging.scala:59) - Added rdd_9_9 in memory on localhost:43449 (size: 16.5 KB, free: 3.8 GB)
INFO [Executor task launch worker-1] (Logging.scala:59) - Updated info of block rdd_9_9
How Many tasks per Stage?
Number of Tasks

- Number of InputSplit defines the number of tasks.
  - Hadoop FileInputFormat defines
  - Immediate Narrow transformations will follow the parent

- If GroupBy is used
  - The number of keys define the number of tasks
object WordCount {
  def main (args: Array[String]){ 
    val driver = "spark://192.168.0.3:7077"
    val sc = new SparkContext(driver, "SparkWordCount")
    val numPartitions = 10
    val lines = sc.textFile("here"+"sometext.txt", numPartitions)
    val words = lines.flatMap(_.split(" "))
    val groupWords = words.groupBy { x => x }
    val wordCount = groupWords.map( x => (x._1,x._2.size))
    val result = wordCount.saveAsTextFile("there"+"wordcount")
  }
}
object WordCount {
  def main (args: Array[String]){
    val driver = "spark://192.168.0.3:7077"
    val sc = new SparkContext(driver, "SparkWordCount")
    val numPartitions = 10
    val lines = sc.textFile("here"+"sometext.txt", numPartitions).coalesce(numPartitions)
    val words = lines.flatMap (_.split(" "))
    val groupWords = words.groupBy { x => x }
    val wordCount = groupWords.map( x => (x._1,x._2.size))
    val result = wordCount.saveAsTextFile("there"+"wordcount")
  }
}
How does the driver know about the resource information of the workers?
MESOS Architecture

- Hadoop scheduler
- MPI scheduler
- Mesos master
- Standby master
- Standby master
- ZooKeeper quorum

Mesos slave
- Hadoop executor
- MPI executor
- task
Resource Allocation Example

• Slave 1 reports to the master that it has 4 CPUs and 4 GB of memory free.
• The master then invokes the allocation policy module, which tells it that framework 1 should be offered all available resources.
• The master sends a resource offer describing what is available on slave 1 to framework 1.
• The framework’s scheduler replies to the master with information about two tasks to run on the slave, using <2 CPUs, 1 GB RAM> for the first task, and <1 CPUs, 2 GB RAM> for the second task.
Resource Allocation Example
Resource Allocation Example

• Finally, the master sends the tasks to the slave, which allocates appropriate resources to the framework’s executor, which in turn launches the two tasks (depicted with dotted-line borders in the figure). Because 1 CPU and 1 GB of RAM are still unallocated, the allocation module may now offer them to framework 2.

• Again the steps are repeated when some tasks are finished or resources become available.
Resource Allocation Example

• For example, how can a framework achieve data locality without MESOS knowing which nodes store the data required by the framework?

  – MESOS answers these questions by simply giving frameworks the ability to **reject** offers. A framework will reject the offers that do not satisfy its constraints and accept the ones that do.