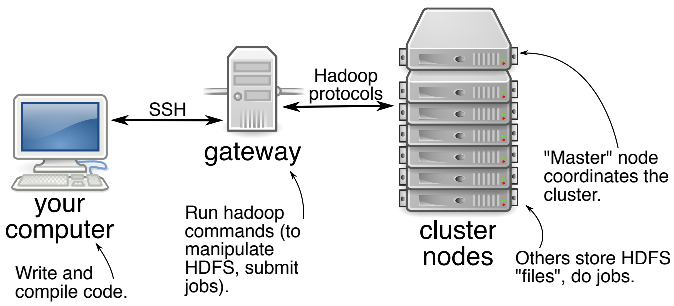
Hadoop Concepts

CMPT 732, Fall 2018

Our Cluster



* * *

Hadoop Pieces

The major software components of a Hadoop cluster, and understanding the way our work get distributed among them requires a little background…

HDFS

The problem: store (GB, TB, PB of) data in the cluster.

HDFS stores (blocks of) files on different nodes, replicated to make the data more available and to handle disk failure.

The *NameNode* coordinates everything and keeps track of who has what data.

The *DataNode*s actually store the data blocks (with each block on multiple nodes).

YARN

The problem: use the compute resources (processors, memory) in the cluster to do work.

The *ResourceManager* tracks resources and jobs in the cluster.

The *NodeManager*s do actual compute work (and are also HDFS DataNodes).

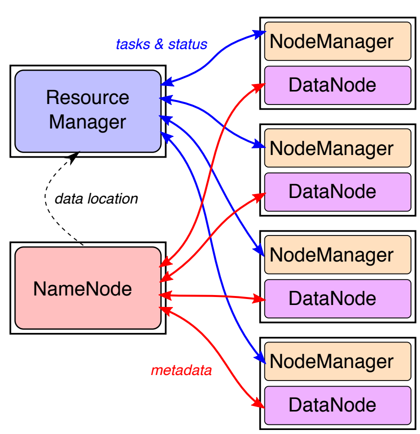
Each application running has a node as its *ApplicationMaster* to coordinate it (one of the NodeManagers).

The idea is that YARN will move a compute task to the data it's operating on: that's easier that moving the (possibly TB of data) to another node.

> “Moving Computation is Cheaper than Moving Data” [HDFS Architecture Guide]

That's why DataNodes and NodeManagers are on the same computers: to get the compute happening beside the data.

(Simplified) Cluster Overview



Work on Hadoop

A program you write to run on Hadoop is an *application* which could send multiple *jobs* to the cluster.

Different types of tasks can run on YARN and first we will see…

MapReduce

The *MapReduce* model is a way to describe distributed parallel computation in a way that we can use for lots of problems, without having to do the message passing ourselves.

The MapReduce tool will be responsible for getting the computation done, as long as we express it in a way that fits the model.

Apache Hadoop provides an implementation of the MapReduce model that it can run on a YARN cluster.

From now on “MapReduce” will mean Hadoop MapReduce…

MapReduce Stages

1. Map: apply a `map()` function to each piece of input. Output key/value pairs.
2. Shuffle: collect map output **with the same keys** together on a node so we can…
3. Reduce: call a `reduce()` function on each key and **each value** that a mapper produced for that key. Produce the final output for that key.

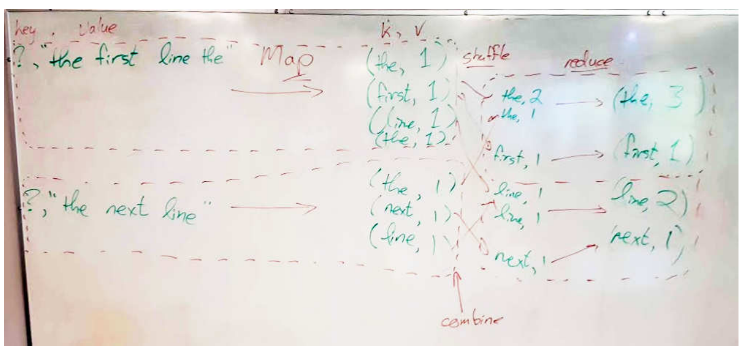
There can also be a step 1.5 Combiner: do reducer-like work on the data (on each node separately) to reduce the amount of data sent to the (expensive) shuffle.

Example: word count

Everybody's first example: count the number of times each word occurs in a collection of text files.

1. Map: for each “word” in the input, output (“word”, 1).
2. Shuffle: move all of the (“word”, n) pairs to the same node, and (“other”, n) to the same node, …
3. Reduce: Sum the values (the numbers) for each key (the words) to get a count of each word like: (“word”, 74), (“other”, 18), …

Whiteboard: Fall 2018



MapReduce Anatomy

```
public class WordCount extends Configured implements Tool {
  public static class TokenizerMapper
      extends Mapper<LongWritable, Text, Text, IntWritable>{
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    @Override
    public void map(LongWritable key, Text value, Context context
        ) throws IOException, InterruptedException {
      StringTokenizer itr = new StringTokenizer(value.toString());
      while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        context.write(word, one);
      }
    }
  }
⋮
}
```

```
public class WordCount extends Configured implements Tool {
  ⋮
  public static class IntSumReducer
      extends Reducer<Text, IntWritable, Text, IntWritable> {
    private IntWritable result = new IntWritable();
    @Override
```

```
    public void reduce(Text key, Iterable<IntWritable> values,
        Context context) throws IOException, InterruptedException {
      int sum = 0;
      for (IntWritable val : values) {
        sum += val.get();
      }
      result.set(sum);
      context.write(key, result);
    }
  }
  ⋮
}
```

```
public class WordCount extends Configured implements Tool {
  …@Override
  public int run(String[] args) throws Exception {
    Configuration conf = this.getConf();
    Job job = Job.getInstance(conf, "word count");
    job.setJarByClass(WordCount.class);
    job.setInputFormatClass(TextInputFormat.class);

    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);

    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    job.setOutputFormatClass(TextOutputFormat.class);
    TextInputFormat.addInputPath(job, new Path(args[0]));
    TextOutputFormat.setOutputPath(job, new Path(args[1]));

    return job.waitForCompletion(true) ? 0 : 1;
  } … }
```

```
public class WordCount extends Configured implements Tool {
  ⋮
  public static void main(String[] args) throws Exception {
    int res = ToolRunner.run(new Configuration(), new WordCount(),
        args);
    System.exit(res);
  }
}
```

Hadoop MapReduce Details

There are a bunch of moving pieces in a Hadoop MapReduce job to make this work

`InputFormat`
: Get the data we need, split it up and process it into key/value pairs for the Mapper.

`Mapper`
: Processes each key/value input pair. Output key/value pairs to…

Combiner
: An instance of `Reducer` that works as part of the mapper process.

`Partitioner`
: Which keys go to which reducer? Default usually okay.

`Reducer`
: Take a key and an iterable of values: combine all of the values for the key and output result key/value pairs. Can also be plugged in as a combiner.

`OutputFormat`
: Output the key/values from the reducer to… wherever it goes.

Summary Output

A MapReduce task produces some summary stats at the bottom of its output. e.g. WordCount running on the `wordcount-2` data set and with `-D mapreduce.job.reduces=3`:

```
    ⋮
    File System Counters
        FILE: Number of bytes read=1393403
        FILE: Number of bytes written=6484915
        FILE: Number of read operations=0
        FILE: Number of large read operations=0
        FILE: Number of write operations=0
        HDFS: Number of bytes read=11805917
        HDFS: Number of bytes written=1388747
        HDFS: Number of read operations=72
        HDFS: Number of large read operations=0
        HDFS: Number of write operations=8
```

```
    Job Counters
        Launched map tasks=20
        Launched reduce tasks=3
        Data-local map tasks=20
        Total time spent by all maps in occupied slots (ms)=65719
        Total time spent by all reduces in occupied slots (ms)=10722
        Total time spent by all map tasks (ms)=65719
        Total time spent by all reduce tasks (ms)=10722
        Total vcore-milliseconds taken by all map tasks=65719
        Total vcore-milliseconds taken by all reduce tasks=10722
        Total megabyte-milliseconds taken by all map tasks=67296256
        Total megabyte-milliseconds taken by all reduce tasks=10979328
```

```
    Map-Reduce Framework
        Map input records=257133
        Map output records=2136769
        Map output bytes=20179022
        Map output materialized bytes=1994018
        Input split bytes=2891
        Combine input records=2136769
        Combine output records=242437
        Reduce input groups=124619
        Reduce shuffle bytes=1994018
        Reduce input records=242437
        Reduce output records=124619
        Spilled Records=484874
        Shuffled Maps =80
        Failed Shuffles=0
        Merged Map outputs=80
        GC time elapsed (ms)=2359
        CPU time spent (ms)=54910
        Physical memory (bytes) snapshot=13991411712
        Virtual memory (bytes) snapshot=63790481408
        Total committed heap usage (bytes)=14790688768
```

```
    Shuffle Errors
        BAD_ID=0
        CONNECTION=0
        IO_ERROR=0
        WRONG_LENGTH=0
        WRONG_MAP=0
        WRONG_REDUCE=0
    File Input Format Counters
        Bytes Read=11803026
    File Output Format Counters
        Bytes Written=1388747
```

MapReduce Parallelism

The goal of all of this structure is to *do work in parallel* across all of the cores available.

The `InputFormat` decides how to split the input, and thus how many map processes there will be.

YARN is responsible for running the map tasks: it does this in parallel as much as possible, and tries to run them locally to the inputs.

With the [`TextInputFormat`] we have been using:

- Every file goes to at least one separate mapper.
- If the file is uncompressed (or uses a splittable compression method) and large enough, it will be split to multiple map tasks.

Each mapper can do its job in parallel, and send its output to the shuffle. If the amount of parallelism isn't right for the problem/cluster, then things are going to be slow.

Too few parallel tasks: few cores used.

Too many tasks: they get queued and the overhead of starting/stopping them dominates.

How can you change the number of mappers to make it “right”?

Option 1: override the `InputFormat` to change the way the input is split.

Option 2: fiddle with the input files to make the size/number you want.

I would almost certainly choose 2. [Fix your input], then start working with it. (And similar advice with Spark.)

We had to explicitly set the number of reducers: there's nothing the framework can do to guess the right number.

The [`Partitioner`] decides which key goes to which reducer.

Then each reducer can work in parallel and produces a separate output file. Again, YARN runs them.

Writables

All of the keys and values we have been using are wrapped in a [`Writable`] implementation.

Basically, the `Writable` implementations know how to (efficiently) serialize/deserialize their wrapped type for passing around the cluster.

The `WritableComparable` adds the ability to compare (for the shuffle).

Hadooop includes implementations for basic Java types:

- `int` → `IntWritable`
- `long` → `LongWritable`
- `float` → `FloatWritable`
- `double` → `DoubleWritable`
- `boolean` → `BooleanWritable`
- `String` → `Text` (should have been called `StringWritable` but it wasn't)

Implementing your own `Writable` (or `WritableComparable`) is easy if you have a more complex key or value.

e.g. the `LongPairWritable` from Assignment 1.

Example: word count

1. InputFormat: the default `TextInputFormat` splits files into lines (with byte-offset as key): `(241, "one text line")`.
2. Mapper: Break the line up into words, and count one occurrence of each: `("one", 1), ("text", 1), ("line", 1)`.
3. Combiner: Sum the values for each word on this node: `("line", 3), ("one", 32), ("text", 2)`.
4. Shuffle: Move equal keys to same reducer, as decided by `HashPartitioner`.
5. Reducer: Sum the values for each word: `("line", 12), ("one", 76), ("text", 6)`.
6. OutputFormat: the default `TextOutputFormat` outputs tab-separated keys and values:

```
line	12
one	76
text	6
```

About MapReduce

Many problems can be expressed as a MapReduce task (or maybe multiple chained MapReduce tasks).

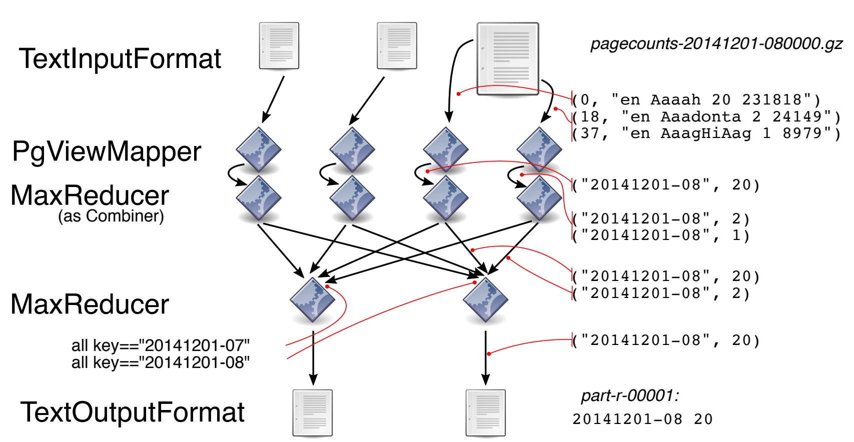
… but it's a little limiting. We'll see more flexibility in Spark.

MapReduce: One more way

[* meaning zero-or-more]

1. InputFormat: input → (k1, v1)*
2. Mapper: (k1, v1) → (k2, v2)*
3. Shuffle: move (k2, v2) to get equal k2 together (RawComparator decides what “equal” means)
4. Reducer: (k2, list(v2)) → (k3, v3)*
5. OutputFormat: (k3, v3) → output

MapReduce Data Flow



* *