Tuning Map-Reduce Job

How many mappers?

- Hadoop will use 1 map task per split, each split is at most 1 block
- If you have n cores per node, you want n splits so that the block is processed locally by n cores
- If there are more cores than the number of splits
  - You can create multiple splits per block by setting **mapreduce.input.fileinputformat.split.maxsize**
  - E.g., for 256MB block, you can set it to 128MB to get 2 splits/block
- If there are fewer cores than map tasks, each core gets multiple tasks
- If you have a large number of small files (each less than 1 block), Map-Reduce will be slow (too much bookkeeping overhead)
  - Use `CombineFileInputFormat`, instead of `FileInputFormat`
  - `CombineFileInputFormat` packs many files into each split so that each mapper has more to process

How Many Reducers?

- Default is 1 reducer
- Not good: the job will be very slow since all the intermediate data will flow through a single reduce task
- To use n reducers, use `job.setNumReduceTasks(n)`
- Increasing the number of reducers ⇒ more parallelism
  - ... but don't create too many small files
- Rule of thumb: each reducer should produce at least one block

Combiner

- Check whether your job can take advantage of a combiner to reduce the amount of data passing through the shuffle
- It does partial reduction after map but before shuffling
- It reduces the amount of data passing through the shuffle

Types:

- map(k, v) → list(< k', v'>)
- combine(k', list(v')) → list(< k', v'>)
- reduce(k', list(v')) → list(< k", v">)

Example 1:
Word Count Revisited:
   Problem: Count frequency of each word

MapReduce Pseudocode
```
  map(k, v)
    split v into words
    for each w in words
      emit(w, 1)

  reduce(k, v)
    emit(k, len(v))
```

MapReduce Pseudocode with Combiner
```
  map(k, v)
    split v into words
    for each w in words
      emit(w, 1)

  combine(k, values)
    count = 0
    for each v in values
       count +=v
    emit(k, count)


  reduce(k, values)
    count = 0
    for each v in values
       count +=v
    emit(k, count)
```

Note: You need to set the combiner class inside the run method
- **job.setCombinerClass(MyCombiner.class);**

Associativity
- Reduce tasks apply an associative and commutative operation
    - **avg** is not associative
        - Consider **avg(S) = sum(S)/count(S)**
        - **avg(S1, S2)** cannot be expressed in terms of **avg** of **avg(S1), avg(S2)**
        - (1+2+3+4+5)/5=3 not equals to **avg** of first 3 elements and next 2 elements
            - (1+2+3)/3=2, (4+5)/2=4.5; (2+4.5)/2=3.25
    - Solution: use an associative accumulation for the combiner and do the final aggregation at the reducer
- For **avg(S)**, we use the pair **(sum(S), count(S))**
    - Accumulating (sum(S), count(S)) is done at the combiner

        - (sum(S1 ∪ S2), count(S1 ∪ S2)) = (sum(S1) + sum(S2), count(S1) + count(S2))
    - Final accumulation and avg is done at the reducer

Example 2
The Avg-by-key Example Revisited

- Given a CSV file *S* with int-double pairs, we want to group data by the first column and, for each group, we want to calculate the average value of the second column

MapReduce Pseudocode

```
map(key, line):
  parse the line into a long key and a double value
  emit(key, value)

reduce(key, values):
  sum = 0.0
  count = 0
  for each v in values:
    count++
    sum += v
  emit(key, sum/count)
```

MapReduce Pseudocode with Combiner

```
map(key, line):
  parse the line into a long key and a double value
  emit(key, (value, 1))

combine(key, values):
  sum = 0.0
  count = 0
  for each v in values:
    count++
    sum += v.value
  emit(key, (sum, count))

reduce(key, values):
  sum = 0.0
  count = 0
  for each v in values:
    sum += v.sum
    count += v.count
    emit(key, (sum/count))
```

Pros:

- Reduce the amount of intermediate data that is shuffled across the network

Issues

- Doesn't actually reduce the number of key-value pairs that are emitted by the mappers in the first place
  - Intermediate key-value pairs are still generated on a per-split, only to be "compacted" by the combiners
  - This process involves unnecessary object creation and destruction
    - Garbage collection takes time
  - Object serialization and deserialization
    - When intermediate key-value pairs fill the in-memory buffer holding map outputs and need to be temporarily spilled to disk
- The combiner is provided as a semantics-preserving optimization to the execution framework, which has the option of using it, perhaps multiple times, or not at all
  - Hadoop makes no guarantees on how many times the combiner is applied, or that it is even applied at all

In-Mapper Combining

- Instead of using a combiner, one may combine values in the mapper using a static Hashtable H

Mapreduce Pseudocode

```
map(key, v):
    setup()
     H = new HashMap<>

    parse the line into a long key and a double value

    if(!H.containsKey(k))
      H.put(k, (v,1))
    else
      pair = H.get(k)
      pair.sum = pair.sum + v
      pair.count = pair.count + 1
      H.put(k, pair)

    cleanup()
      for each k in H
        emit(k, H.get(k);

 reduce(key, values):
    sum = 0.0
    count = 0
    for each v in values:
        count +=v.count
        sum += v.sum
    emit(key, sum/count)
```

Note:

- map() is called once for each key/value pair in the input split
- setup() is called once at the start of the task
- cleanup() is called at the end of the task

Pros

- Programmer has the full control over local aggregation
- Emit is deferred until cleanup()
    - Only generate those key-value pairs that need to be shuffled across the network

Issues

- Might encounter scalability bottleneck
    - Depends on having sufficient memory to store intermediate results until the mapper has completely processed all key-value pairs in an input split
    - Solution 1
        - Flush intermediate key-value pairs once memory usage has crossed a 'threshold'
    - Solution 2
        - Emit partial results after processing every n key-value pairs using a counter variable
    - Challenge
        - In Hadoop, physical memory is split between multiple tasks that may be running on a node concurrently
            - Tasks are not aware of each other and are competing for finite resources

Summary

- The extent local aggregation efficiency depends on
    - The size of the intermediate key space
    - The distribution of keys themselves, and
    - The number of key-value pairs that are emitted by each individual map task

Optimize Join
Example 3
Join Example Revisited

- Join two datasets R(A, C) and S(B,D) on R.A and S.B:

```
select R.C, S.D
from R, S
where R.A = S.B
```

Reduce-Side Join

- Map-Reduce now has two mappers:
    - a mapper for R that uses the join key R.A
    - a mapper for S that uses the join key S.B

- The mappers will send the R and S values with the same keys R.A = S.B to the same reducer
- Must tag R tuples with 0 and S tuples with 1 so that the reducer can tell them apart

```
Map1 (key, r)
    emit(r.A, (0,r));

Map2 (key, s)
    emit(s.B, (1,s));

Reduce (key, values)
    for each (0, r) ∈ values
      for each (1, s) ∈ values
        emit(key, (r.C, s.D));
```

Map-side Join

- Partition and sort both datasets R and S in the same way
- For example, suppose R and S were both:
  - Divided into same number of partitions
    - Partitioned in the same manner by the join key
  - In each partition, the tuples were sorted by the join key
  - All the records for a particular key reside in the same partition
- In this case, we simply need to merge join:
  - The first file of R with the first file of S
  - The second file with R with the second file of S, etc
- This can be accomplished in parallel, in the map phase of a MapReduce job
  - Hence, a map-side join
- In practice, we map over one of the datasets (the larger one) and inside the mapper read the corresponding part of the other dataset to perform the merge join

Note: No reducer is required, unless the programmer wishes to repartition the output or perform further processing

- A map-side join is far more efficient than a reduce-side join
  - There is no need to shuffle the datasets over the network
- But is it realistic to expect that the stringent conditions required for map-side joins are satisfied?
  - In many cases, yes
- The reason is that relational joins happen within the broader context of a workflow, which may include multiple steps
- Therefore, the datasets that are to be joined may be the output of previous processes (either MapReduce jobs or other code)
- If the workflow is known in advance and relatively static (both reasonable assumptions in a mature workflow), we can engineer the

previous processes to generate output sorted and partitioned in a way that makes efficient map-side joins possible
  - In MapReduce, by using a custom partitioner and controlling the sort order of key-value pairs

Memory-backed Join

- Assume that one dataset, say R, can fit in the memory of every node
- The memory-backed join algorithm:
  - Broadcast the R dataset to all worker nodes
  - Before a Map-Reduce job starts, create a hash table from R
  - A mapper joins the input splits of S with the entire R hash table by probing the hash table
  - The Map-Reduce job needs a map stage only (no reduce stage)
- This is known as a simple hash join by the database community

Intermediate compression

- Job execution time can almost always benefit from enabling map output compression
- Two benefits:
  - Space needed to store files
  - It speeds up data transfer across the network or to or from disk
- Compression algorithms exhibit a space/time trade-off:
  - Faster compression and decompression speeds usually come at the expense of smaller space savings
- A codec is the implementation of a compression-decompression algorithm
  - In Hadoop, a codec is represented by an implementation of the CompressionCodec interface
  - So, for example, GzipCodec encapsulates the compression and decompression algorithm for gzip
- Needs to support splitting
  - Otherwise creating a split for each map won't work
  - A single map might have to process all the HDFS blocks, most of which may not be local to the map
- Solution:
  - Use a compression format that supports splitting, such as bzip2 (although bzip2 is fairly slow), or one that can be indexed to support splitting, such as LZO, or
  - Split the file into chunks in the application, and compress each chunk separately using any supported compression format (it doesn't matter whether it is splittable)
  - In this case, you should choose the chunk size so that the compressed chunks are approximately the size of an HDFS block

Custom serialization

- If you are using your own customWritableobjects or custom comparators, make sure you have implementedRawComparator
- Every time Map-Reduce compares keys in sorting/grouping, it deserializes the keys
- Expensive if the key is a complex object

```
class Pair implements WritableComparable {
      public int X;
      public int Y;
      public int compareTo (Pair o) {
      return (X == o.X)? Y-o.Y : X-o.X;
      }
      public void write (DataOutput out) throws IOException {…}
      public void readFields (DataInput in) throws IOException {…}
}
```

- We want to compare keys without deserializing them
- One optimization that Hadoop provides is the RawComparator extension of Java's Comparator:

```
package org.apache.hadoop.io;
import java.util.Comparator;

public interface RawComparator<T> extends Comparator<T> {
public int compare(byte[] b1, int s1, int l1, byte[] b2, int s2, int
l2); }
```

- This interface permits implementors to compare records read from a stream without deserializing them into objects, thereby avoiding any overhead of object creation
- For example, the comparator for IntWritables implements the raw compare() method by reading an integer from each of the byte arrays b1 and b2 and comparing them directly from the given start positions (s1 and s2) and lengths (l1 and l2)
- A RawComparator compares two serialized keys (i.e, as bytes sequences)
- WritableComparator is a general-purpose implementation of RawComparator

```
class IntPairComparator extends WritableComparator {
    public IntPairComparator() {
      super(IntPairWritable.class, true);

    }


    //byte[] b1: The byte array representing the serialized form of the
first IntPairWritable object
    //int s1: The starting index in b1 from which the comparison should
begin (offset).
    //int l1: The length of the serialized data in b1 that should be
considered for comparison.
    @Override
    public int compare(byte[] b1, int s1, int l1, byte[] b2, int s2,
int l2) {
        int first1 = readInt(b1, s1);
        int first2 = readInt(b2, s2);
        return Integer.compare(first1, first2);
    }
}
```

Use a Custom Partitioner to Reduce Data Skew

- The default partitioner (can be overwritten):

```
class MyPartitioner extends Partitioner<KEY, VALUE> {
   public int getPartition (KEY key, VALUE value, int numPartitions)
      return Math.abs(key.hashCode()) % numPartitions;
   }
 }

   job.setPartitionerClass (MyPartitioner.class);
```

- Another way: you overwrite the hashCode() method
- Total sorting: you want to use many reducers but need to use a partitioner that respects the total order of the output ⇒ range partitioning

```
class MyPartitioner extends Partitioner<LongWritable, VALUE> {
   public int getPartition (LongWritable key, VALUE value, int
numPartitions
   int n = maxKeyValue % numPartitions + 1;
   return key/n;
   }
}
```

The MapReduce Web UI

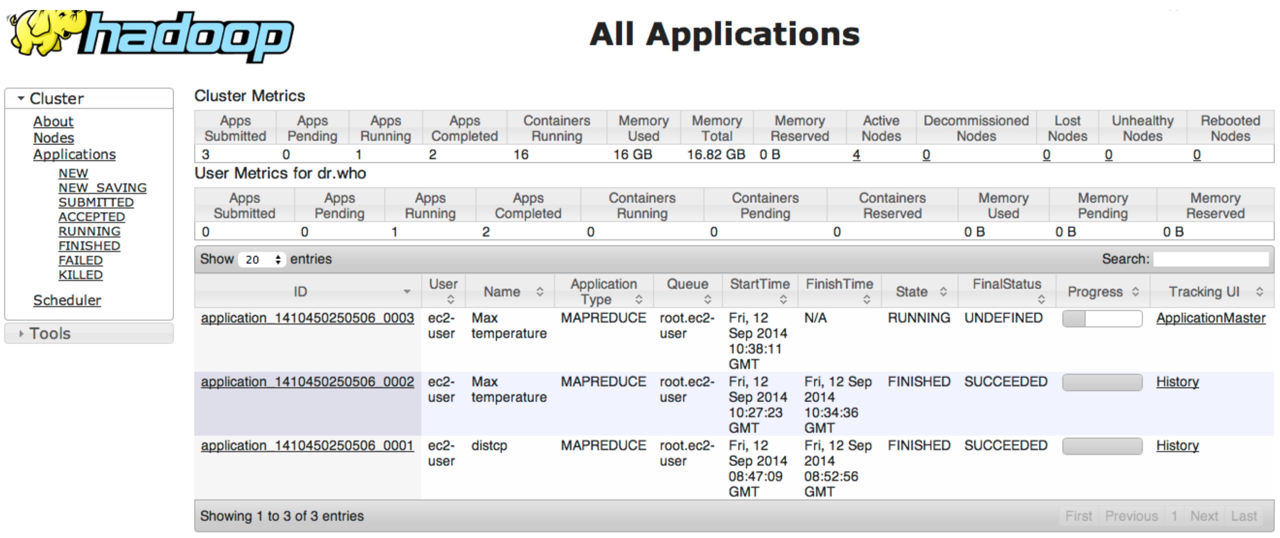
- Hadoop comes with a web UI for viewing information about your jobs
- It is useful for following a job's progress while it is running, as well as finding job statistics and logs after the job has completed
- You can find the UI at `http://resource-manager-host:8088/` or `http://localhost:8088/`

The resource manager page

- The "Cluster Metrics" section gives a summary of the cluster
- This includes the number of applications currently running on the cluster (and in various other states), the number of resources

available on the cluster ("Memory Total"), and information about node managers

- The main table shows all the applications that have run or are currently running on the cluster
- There is a search box that is useful for filtering the applications to find the ones you are interested in
- The main view can show up to 100 entries per page, and the resource manager will keep up to 10,000 completed applications in memory at a time before they are only available from the job history page
- Note also that the job history is persistent, so you can find jobs there from previous runs of the resource manager, too



All Applications

Cluster Metrics

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Memory Used	Memory Total	Memory Reserved	Active Nodes	Decommissioned Nodes	Lost Nodes	Unhealthy Nodes	Rebooted Nodes
3	0	1	2	16	16 GB	16.82 GB	0 B	4	0	0	0	0

User Metrics for dr.who

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Containers Pending	Containers Reserved	Memory Used	Memory Pending	Memory Reserved
0	0	1	2	0	0	0	0 B	0 B	0 B

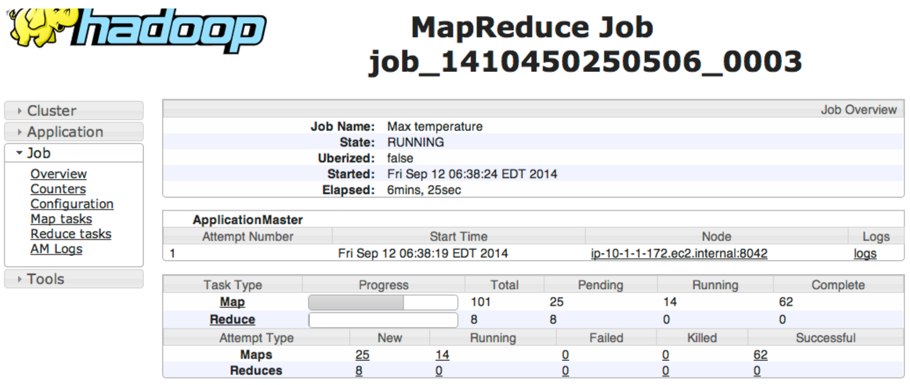
Show 20 entries Search:

ID	User	Name	Application Type	Queue	StartTime	FinishTime	State	FinalStatus	Progress	Tracking UI
application_1410450250506_0003	ec2-user	Max temperature	MAPREDUCE	root.ec2-user	Fri, 12 Sep 2014 10:38:11 GMT	N/A	RUNNING	UNDEFINED		ApplicationMaster
application_1410450250506_0002	ec2-user	Max temperature	MAPREDUCE	root.ec2-user	Fri, 12 Sep 2014 10:27:23 GMT	Fri, 12 Sep 2014 10:34:36 GMT	FINISHED	SUCCEEDED		History
application_1410450250506_0001	ec2-user	distcp	MAPREDUCE	root.ec2-user	Fri, 12 Sep 2014 08:47:09 GMT	Fri, 12 Sep 2014 08:52:56 GMT	FINISHED	SUCCEEDED		History

Showing 1 to 3 of 3 entries

The MapReduce job page

- Clicking on the link for the "Tracking UI" takes us to the application master's web UI (or to the history page if the application has completed)
- In the case of MapReduce, this takes us to the job page



MapReduce Job job_1410450250506_0003

Job Overview

Job Name:	Max temperature
State:	RUNNING
Uberized:	false
Started:	Fri Sep 12 06:38:24 EDT 2014
Elapsed:	6mins, 25sec

ApplicationMaster

Attempt Number	Start Time	Node	Logs
1	Fri Sep 12 06:38:19 EDT 2014	ip-10-1-1-172.ec2.internal:8042	logs

Task Type	Progress	Total	Pending	Running	Complete
Map		101	25	14	62
Reduce		8	8	0	0

Attempt Type	New	Running	Failed	Killed	Successful
Maps	25	14	0	0	62
Reduces	8	0	0	0	0

- While the job is running, you can monitor its progress on this page
- The table at the bottom shows the map progress and the reduce progress
- "Total" shows the total number of map and reduce tasks for this job (a row for each)
- The other columns then show the state of these tasks: "Pending" (waiting to run), "Running," or "Complete" (successfully run)
- The lower part of the table shows the total number of failed and killed task attempts for the map or reduce tasks
- Task attempts may be marked as killed if they are speculative execution duplicates, if the node they are running on dies, or if they are killed by a user
- There also are a number of useful links in the navigation
    - For example, the "Configuration" link is to the consolidated configuration file for the job, containing all the properties and their values that were in effect during the job run
    - If you are unsure of what a particular property was set to, you can click through to inspect the file

Logs
- Hadoop produces logs in various places, and for various audiences

System daemon logs
- For administrators
- Each Hadoop daemon produces a logfile (using log4j) and another file that combines standard out and error
- Written in the directory defined by the HADOOP_LOG_DIR environment variable

HDFS audit logs
- For administrators
- A log of all HDFS requests, turned off by default
- Written to the namenode's log, although this is configurable

MapReduce job history logs
- For users
- A log of the events (such as task completion) that occur in the course of running a job
- Saved centrally in HDFS

MapReduce task logs
- For users
- Each task child process produces a logfile using log4j (called syslog), a file for data sent to standard out (stdout), and a file for standard error (stderr)

- Written in the userlogs subdirectory of the directory defined by the YARN_LOG_DIR environment variable

Optional:
Memory Backed Join when neither datasets fit into memory
Approach 1:

- What if neither dataset fits in memory?
- The simplest solution is to divide the smaller dataset, let's say S, into n partitions, such that S = S1 ∪ S2 ∪ . . . ∪ Sn
- We can choose n so that each partition is small enough to fit in memory, and then run n memory-backed hash joins
- This, of course, requires streaming through the other dataset n times

Approach 2:

- A distributed key-value store can be used to hold one dataset in memory across multiple machines while mapping over the other
- The mappers would then query this distributed key-value store in parallel and perform joins if the join keys match
  - In order to achieve good performance in accessing distributed key-value stores, it is often necessary to batch queries before making synchronous requests (to amortize latency over many requests) or to rely on asynchronous requests
- The open-source caching system memcached can be used for exactly this purpose, and therefore this approach is called memcached join
- Reference: Jimmy Lin, Anand Bahety, Shravya Konda, and Samantha Mahindrakar. Low- latency, high-throughput access to static global resources within the Hadoop framework. Technical Report HCIL-2009-01, University of Maryland, College Park, Maryland, January 2009.

Custom InputFormat

- Normally, not needed
- Here is an example where you may need it
- You want to process many random nodes (without any input data)
  - E.g., generate random data for performance evaluation
- You want a generator that generates small input splits, one per worker node
- Generate num tiny files, where each one has 1 pair (start, length):

```
void generator (String directory, long num, long length) {
   for (int i = 0; i < num; i++) {
      Path path = new Path(directory);
```

```
            SequenceFile.Writer writer = SequenceFile.createWriter
         (path.getFileSystem(conf), conf, path, LongWritable.class,
         LongWritable.class, SequenceFile.CompressionType.NONE);
            writer.append(new LongWritable(i*length),new LongWritable(length));
            writer.close();
            }
         }

         class GeneratorInputFormat extends FileInputFormat<LongWritable,
   LongWritable> {
           public static class GeneratorRecordReader extends RecordReader
   <LongWritable, LongWritable> {
               final long offset;
               final long size;
               long index;

               public GeneratorRecordReader (FileSplit split, TaskAttemptContext
               context ) throws IOException {
                     Configuration conf = context.getConfiguration ();
                     Path path = split.getPath();
                     FileSystem fs = path.getFileSystem(conf );
                     SequenceFile.Reader reader = new
                     SequenceFile.Reader(path.getFileSystem(conf), path, conf);
                     LongWritable key = new LongWritable();
                     LongWritable value = new LongWritable();
                     reader.next(key, value );
                     offset = key.get();
                     size = value.get();
                     index = 0;
                     reader.close ();
               }

         public boolean nextKeyValue () throws IOException {
               return index++ < size;
         }

         public LongWritable getCurrentKey () throws IOException {
               return new LongWritable(index);
         }

         public LongWritable getCurrentValue () throws IOException {
               return new LongWritable(offset+index);
         }

         public float getProgress () throws IOException {
               return index/(float) size;
               }
         }
```

```
public RecordReader createRecordReader (InputSplit split,
TaskAttemptContext context) throws IOException
      return new GeneratorRecordReader((FileSplit) split, context);
      }
```