Curso de extensão em Data Science

GERÊNCIA DE INFRAESTRUTURA PARA BIG DATA

Prof. Tiago Ferreto – tiago.ferreto@pucrs.br

ESCOLA POLITÉCNICA

JAVA MAPREDUCE API

Java MapReduce API – Basic concepts

- Serialization is widely present in Java MapReduce API
  - Conversion of data structures to byte streams and vice versa (deserialization)
- Hadoop Datatypes
  - Keys and values are serializable objects
    - Utilization of built-in data object types (Box Classes) instead of primitives types (int, long, char)
    - Requires using specific methods to access/modify data
  - All serializable objects use the Writable interface
    - Define accessor and mutator methods
    - Example: methods readFields and write
  - Sorting is provided by a WritableComparable interface
    - Provides methods compareTo, equals and hashCode

Hadoop Box Class	Java Primitive
BooleanWritable	boolean
ByteWritable	byte
IntWritable	int
FloatWritable	float
LongWritable	long
DoubleWritable	double
NullWritable	null
Text	String

Java MapReduce API – Basic concepts

- InputFormats
  - Specifies how data (keys and values) are extracted from a file
  - Provide a factory for RecordReader objects → used to extract data from an Input Split (typically a HDFS block)
  - Common InputFormats
    - **TextInputFormat**: InputFormat for plain files. Files are broken into lines. Keys are the position in the file, and values are the line of text.
    - **KeyValueTextInputFormat**: Similar to TextInputFormat, but with each line divided into a key and value by a separator
    - **SequenceFileInputFormat**: InputFile for SequenceFiles (special Hadoop format)
    - **NLineInputFormat**: Similar to TextInputFormat, and specifies how many lines should go to each map task
    - **DBInputFormat**: InputFormat to read data from a JDBC data source
    - **FixedLengthInputFormat**: InputFormat to read input files which contain fixed length records

Java MapReduce API – Basic concepts

- SequenceFiles
  - Binary encoded, serialized data files designed for use in Hadoop
  - Contain metadata defining datatypes for key and value objects within the file
  - Can be uncompressed or compressed
  - Efficient in multi-stage workflow (output of one MapReduce used as input for another MR job)
  - Not accessible from other languages (only Java)
    - Alternative: Avro format → provides a cross-language serialization format

Java MapReduce API – Basic concepts

- OutputFormats
  - Determine how data is written out to files
  - Common OutputFormats
    - FileOutputFormat: Writes output data to a file
    - DBOutputFormat: Writes output data to a JDBC data source

Components of a MapReduce Program

- A typical MapReduce Program contains the following components:
  - Driver – code executed on the client which sets up and starts the MapReduce application
  - Mapper – Java class that contains the map() method
  - Reducer – Java class that contains the reducer() method

Driver

- It submits the application and its configuration to the ResourceManager
- Jobs can be submitted
  - Synchronously: waits for the application to complete before performing another action
  - Asynchronously: does not wait the application to complete
- Can configure and submit more than one application
  - For example: workflow of MapReduce applications
- Typical parameters
  - Path to input data
  - Path to output data
  - HDFS/YARN cluster to be use for the application
- Implemented as a Java class containing the entry point (main() method) for the program

Job Object

- Stores Job configuration
  - Classes to be used as Mapper and Reducer
  - Input and output directories
  - Job name to be displayed in the YARN Resource Manager UI
  - Among other options
- Can also be used to control application's submission and execution and to query the state of the application

Mapper

- Java class containing the map() method
- Each Mapper instance iterates through its assigned InputSplit
  - Executes its map() method against each record read from the InputSplit, using a defined InputFormat and its associated RecordReader
- Number of InputSplits (usually the number of HDFS blocks) in the input data determines the number of Map tasks in a MR application
- Implementation concerns
  - **No saving or sharing of state** – data should not be shared between Map tasks
  - **No side effects** – map tasks may be executed in any sequence or executed more than once without creating side effects
  - **No attempt to communicate with other map tasks** – map tasks are not intended to communicate with one another
  - **Careful on performing external IO operations (writing to NFS or accessing a service)** – can be perceived as a DDoS attack or event storm

Reducer

- Runs against a partition and respective sorted keys
  - Its associated values are passed to the reduce() method
- The same implementation concerns for Mappers should be applied when implementing a reducer

Hello World in MapReduce → Word Count

- Counts the occurrence of words in a text file
- Useful in real-life problems
  - Counting specific event occurrences on log files
  - Text mining functions
  - Word clouds
- Input → text file
- Map task → splits a line of text into a collection of words (tokenization) and outputs each word as key with a value of 1
- Shuffle-and-sort → groups the values for each word and sort keys in alphabetical order
- Reducer → sums the values for each word (key)

WordCountDriver.java

```
public class WordCountDriver extends Configured implements Tool {
  public int run(String[] args) throws Exception {
    if (args.length != 2) {
     System.out.printf("Usage: %s [generic options] <inputdir> <outputdir>\n",
getClass().getSimpleName()); return -1; }
    Job job = new Job(getConf());
    job.setJarByClass(WordCountDriver.class);
    job.setJobName("Word Count");
    FileInputFormat.setInputPaths(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    job.setMapperClass(WordCountMapper.class);
    job.setReducerClass(WordCountReducer.class);
    job.setMapOutputKeyClass(Text.class);
    job.setMapOutputValueClass(IntWritable.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    boolean success = job.waitForCompletion(true);
    return success ? 0 : 1;
  }
 public static void main(String[] args) throws Exception {
    int exitCode = ToolRunner.run(new Configuration(), new WordCountDriver(), args);
    System.exit(exitCode);
  }
}
```


WordCountMapper.java

```
public class WordCountMapper extends Mapper<LongWritable, Text, Text, IntWritable>
{
  private final static IntWritable one = new IntWritable(1);
  private Text wordObject = new Text();

  @Override
  public void map(LongWritable key, Text value, Context context)
    throws IOException, InterruptedException {

   String line = value.toString();
   for (String word : line.split("\\W+")) {
    if (word.length() > 0) {
     wordObject.set(word);
     context.write(wordObject, one);
    }
   }
  }
}
```


WordCountReducer.java

```
public class WordCountReducer extends Reducer<Text, IntWritable, Text, IntWritable>
{
  private IntWritable wordCountWritable = new IntWritable();
  @Override
   public void reduce(Text key, Iterable<IntWritable> values, Context context)
            throws IOException, InterruptedException {
        int wordCount = 0;
        for (IntWritable value : values) {
            wordCount += value.get();
        }
        wordCountWritable.set(wordCount);
        context.write(key, wordCountWritable);
    }
}
```


Compiling, Packaging and Submitting

- Compiling
  - Put all files in the same folder
  - `$ javac -classpath `$HADOOP_HOME/bin/hadoop classpath` *.java`
- Driver, Mapper and Reducer must be packaged in a jar file
  - Can use Maven or javac/jar executables
  - `$ jar cvf wc.jar *.class`
- Submission using the hadoop jar command
  - `$ $HADOOP_HOME/bin/hadoop jar wc.jar WordCountDriver inputdir outputdir`

Advanced MapReduce API Concepts

- Combiners
  - Decrease the amount of intermediate data sent between Mappers and Reducers
  - combiner() function is identical to the reduce() function
  - Output key and value object types from map() function match the input to the Combiner
  - Output key and value object types from the Combiner match the input key and value object types used in the Reducer's reduce() method
  - Operation performed must be commutative and associative
  - Declaration in Driver class
    - `job.setCombinerClass(WordCountReducer.class);`

Advanced MapReduce API Concepts

- Partitioners
  - Divides the output keyspace for a MapReduce application controlling the data each Reducer gets
  - Useful for process distribution, load balancing or segregating output (separating a file for each month of the year)
  - HashPartitioner is used by default
    - Uses a hash function to separate the keyspace in roughly equal parts among Reducers
  - Declaration of a Custom Partitioner in Driver Class
    - `job.setPartitionerClass(MyCustomPartitioner.class);`
  - Extends the base Partitioner class and has a getPartition() method

Advanced MapReduce API Concepts

- Distributed Cache
  - Used to disseminate additional data or class libraries at runtime to Mappers or Reducers in a fully distributed Hadoop cluster environment
  - DistributedCache pushes the data or libraries to all slave nodes as a prerequisite background task before any task for the application is executed
  - The distributed data is available in read-only format on any node running a task for the application
  - After the application terminates, the files are removed automatically
  - Items can be added using the ToolRunner class in the Driver (arguments -files, -archives, -libjars)
  - Example: adding stopwords to the WordCount application
    - `$ hadoop jar wc.jar WordCountDriver -files stopwords.txt inputdir outputdir`
  - Example: accessing files in the DistributedCache
    - `File f = new File("stopwords.txt");`

MapReduce Streaming API

- Enables implementing Map and Reduce functions in languages other than Java (e.g., Perl, Python, Ruby)
- Requires that key-value pairs are presented to the Map or Reduce script using standard input (STDIN), and key-value pairs are emitted from the script using standard output (STDOUT)
- Some drawbacks:
  - Presents additional overhead → performance is poorer than implementing in Java
  - Requires Java for implementing additional API constructs (e.g., InputFormats, Writables, Partitioners)
  - Only suitable for data represented as text

Word Count in Python

```python
#!/usr/bin/env python
# wordmapper.py
import sys
for line in sys.stdin:
  line = line.strip()
  words = line.split()
  for word in words:
   print '%s\t%s' % (word, 1)
```

```python
#!/usr/bin/env python
# wordreducer.py
import sys

thisword = None
wordcount = 0
word = None
for line in sys.stdin:
  line = line.strip()
  word, count = line.split('\t', 1)
  count = int(count)
  if thisword == word:
    wordcount += count
  else:
  if thisword:
    print '%s\t%s' % (thisword, wordcount)
  wordcount = count
  thisword = word
if thisword == word:
  print '%s\t%s' % (thisword, wordcount)
```


Submitting a Streaming MapReduce Job

```
$ hadoop jar \
  $HADOOP_HOME/share/hadoop/tools/lib/hadoop-streaming-
*.jar \
  -input inputdir \
  -output outputdir \
  -mapper wordmapper.py \
  -reducer wordreducer.py \
  -file wordmapper.py \
  -file wordreducer.py
```

MAPREDUCE – HANDS-ON