Data processing in Apache Spark

Pelle Jakovits

5 October, 2015, Tartu
Outline

• Introduction to Spark
• Resilient Distributed Datasets (RDD)
  – Data operations
  – RDD transformations
  – Examples
• Fault tolerance
• Frameworks powered by Spark
Spark

- Directed acyclic graph (DAG) task execution engine
- Supports cyclic data flow and in-memory computing
- Spark works with Scala, Java, Python and R
- Integrated with Hadoop Yarn and HDFS
- Extended with tools for SQL like queries, stream processing and graph processing.
- Uses Resilient Distributed Datasets to abstract data that is to be processed
Hadoop YARN
Performance vs Hadoop

- **K-Means Clustering**
  - Hadoop: 155 seconds
  - Spark: 4.1 seconds

- **Logistic Regression**
  - Hadoop: 110 seconds
  - Spark: 0.96 seconds

Introduction to Spark – Patrick Wendell, Databricks
Resilient Distributed Datasets

• Collections of objects spread across a cluster, stored in RAM or on Disk

• Built through parallel transformations

• Automatically rebuilt on failure
Working in Java

• Tuples

```java
Tuple2 pair = new Tuple2(a, b);
pair._1 // => a
pair._2 // => b
```

• Functions

- In Java 8 you can use lambda functions
  - ```java
  JavaPairRDD<String, Integer> counts = pairs.reduceByKey((a, b) -> a + b);
  ```
- But in older Java you have to use predefined function interfaces:
  - Function,
  - Function2, Function 3
  - FlatMapFunction
  - PairFunction
Java Spark Function types

class GetLength implements Function<String, Integer> {
    public Integer call(String s) {
        return s.length();
    }
}

class Sum implements Function2<Integer, Integer, Integer> {
    public Integer call(Integer a, Integer b) {
        return a + b;
    }
}
Java Example - MapReduce 😊

```java
JavaRDD<Integer> dataSet = jsc.parallelize(l, slices);

int count = dataSet.map(new Function<Integer, Integer>() {
    public Integer call(Integer integer) {
        double x = Math.random() * 2 - 1;
        double y = Math.random() * 2 - 1;
        return (x * x + y * y < 1) ? 1 : 0;
    }
}).reduce(new Function2<Integer, Integer, Integer>() {
    public Integer call(Integer integer, Integer integer2) {
        return integer + integer2;
    }
});
```
Python example

• Word count in Spark's Python API

```python
file = spark.textFile("hdfs://...")

file.flatMap(lambda line: line.split())
    .map(lambda word: (word, 1))
    .reduceByKey(lambda a, b: a+b)
```
Persisting data

• Spark is Lazy

• To force spark to keep any intermediate data in memory, we can use:
  
  – `lineLengths.persist(StorageLevel);`
  
  – which would cause `lineLengths` RDD to be saved in memory after the first time it is computed.

• Should be used in case we want to process the same RDD multiple times.
Persistance level

- **DISK_ONLY**
- **MEMORY_ONLY**
- **MEMORY_AND_DISK**
- **MEMORY_ONLY_SER**
  - More efficient
  - Use more CPU
- **MEMORY_ONLY_2**
  - Replicate data on 2 executors
RDD operations

• **Actions**
  – Creating RDD’s
  – Storing RDD’s
  – Extracting data from RDD on the fly

• **Transformations**
  – Restructure or transform RDD data
Spark Actions
Loading Data

Local data

```java
int[] data = {1, 2, 3, 4, 5};
JavaRDD<Integer> distData = sc.parallelize(data, slices);
```

External data

```java
JavaRDD<String> input = sc.textFile("file.txt")
sc.textFile("directory/*\.txt")
sc.textFile("hdfs://xxx:9000/path/file")
```
Broadcast

* Broadcast a copy of the data to every node in the Spark cluster:

```java
Broadcast<int[]> broadcastVar = sc.broadcast(new int[] {1, 2, 3});

Int[] values = broadcastVar.value();
```
Storing data

- `counts.saveAsTextFile("hdfs://...");`
- `counts.saveAsObjectFile("hdfs://...");`

- `DataCube.saveAsHadoopFile("testfile.seq", LongWritable.class, LongWritable.class, SequenceFileOutputFormat.class);`
Other actions

• **Reduce()** – we already saw in example
• **Collect()** – Retrieve RDD content.
• **Count()** – count number of elements in RDD
• **First()** – Take first element from RDD
• **Take(n)** - Take n first elements from RDD
• **countByKey()** – count values for each unique key
Spark  RDD Transformations
Map

JavaPairRDD<String, Integer> ones = words.map(new PairFunction<String, String, Integer>() {
    public Tuple2<String, Integer> call(String s) {
        return new Tuple2<>(s, 1);
    }
});
groupBy

JavaPairRDD<Integer, List<Tuple2<Integer, Float>>> grouped = values.groupBy(new Partitioner(splits), splits);

class Partitioner extends Function<Tuple2<Integer, Float>, Integer>{
    public Integer call(Tuple2<Integer, Float> t) {
        return r.nextInt(partitions);
    }
}

reduceByKey

JavaPairRDD\textless String, Integer\textgreater counts = ones.reduceByKey(
    new Function2\textless Integer, Integer, Integer\textgreater() { 
        public Integer call(Integer i1, Integer i2) { 
            return i1 + i2;
        }
    }
);
Filter

JavaRDD<String> logData = sc.textFile(logFile);

long ERRORS = logData.filter(
    new Function<String, Boolean>() {
        public Boolean call(String s) {
            return s.contains("ERROR");
        }
    }
).count();

long INFOS = logData.filter(
    new Function<String, Boolean>() {
        public Boolean call(String s) {
            return s.contains("INFO");
        }
    }
).count();
Other transformations

- **sample**(*withReplacement, fraction, seed*)
- **distinct**(*[numTasks]*)
- **union**(*otherDataset*)
- **flatMap**(*func*)
- **groupByKey**()
- **join**(*otherDataset, [numTasks]*) - When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key.
- **cogroup**(*otherDataset, [numTasks]*) - When called on datasets of type (K, V) and (K, W), returns a dataset of (K, Seq[V], Seq[W]) tuples.
Fault Tolerance
Lineage

• Lineage is the history of RDD
• RDD’s keep track of all the RDD partitions
  – What functions were applied to produce it
  – Which input data partition were involved
• Rebuild lost RDD partitions according to lineage, using the latest still available partitions.
• No performance cost if nothing fails (as opposite to checkpointing)
Frameworks powered by Spark

- Spark SQL
- Spark Streaming
- MLlib (machine learning)
- GraphX (graph)

Apache Spark
Frameworks powered by Spark

• Spark SQL- Seemlessly mix SQL queries with Spark programs.
  – Similar to Pig and Hive
• MLlib - machine learning library
• GraphX - Spark's API for graphs and graph-parallel computation.
Advantages of Spark

• Much faster than solutions built on top of Hadoop when data can fit into memory
  – Except Impala maybe
• Hard to keep in track of how (well) the data is distributed
• More flexible fault tolerance
• Spark has a lot of extensions and is constantly updated
Disadvantages of Spark

• What if data does not fit into the memory?
• Saving as text files can be very slow
• Java Spark is not as convinient to use as Pig for prototyping, but you can
  – Use python Spark instead
  – Use Spark Dataframes
  – Use Spark SQL
Conclusion

• RDDs offer a simple and efficient programming model for a broad range of applications

• Spark achieves fault tolerance by providing coarse-grained operations and tracking lineage

• Provides definite speedup when data fits into the collective memory
Thats All

• This week`s practice session
  – Processing data with Spark

• Next week`s lecture is about higher level Spark
  – Scripting and Prototyping in Spark
    • Spark SQL
    • DataFrames
  – Spark Streaming