MapReduce in Spark

Krzysztof Dembczyński

Intelligent Decision Support Systems Laboratory (IDSS)
Poznań University of Technology, Poland

Bachelor studies, eighth semester
Academic year 2018/19 (summer semester)

Review of the previous lectures

- Processing of massive datasets
- Evolution of database systems
- OLTP and OLAP systems
- ETL
- Dimensional modeling
- Data processing
  - Physical storage and data access
  - Materialization
  - Data partition
  - MapReduce

Outline

Outline

Spark

- Spark is a fast and general-purpose cluster computing system.

Spark

- Spark is a fast and general-purpose cluster computing system.
- Also known as a **unified analytics engine for large-scale data processing**.

Spark

- Spark is a fast and general-purpose cluster computing system.
- Also known as a **unified analytics engine for large-scale data processing**.
- It provides high-level APIs in Java, Scala, Python and R, and an optimized engine that supports general execution graphs.

Spark

- Spark is a fast and general-purpose cluster computing system.
- Also known as a **unified analytics engine for large-scale data processing**.
- It provides high-level APIs in Java, Scala, Python and R, and an optimized engine that supports general execution graphs.
- It also supports a rich set of higher-level tools including:

Spark

- Spark is a fast and general-purpose cluster computing system.
- Also known as a **unified analytics engine for large-scale data processing**.
- It provides high-level APIs in Java, Scala, Python and R, and an optimized engine that supports general execution graphs.
- It also supports a rich set of higher-level tools including:
  - Spark SQL for SQL and structured data processing,

Spark

- Spark is a fast and general-purpose cluster computing system.
- Also known as a **unified analytics engine for large-scale data processing**.
- It provides high-level APIs in Java, Scala, Python and R, and an optimized engine that supports general execution graphs.
- It also supports a rich set of higher-level tools including:
  - Spark SQL for SQL and structured data processing,
  - MLlib for machine learning,

Spark

- Spark is a fast and general-purpose cluster computing system.
- Also known as a **unified analytics engine for large-scale data processing**.
- It provides high-level APIs in Java, Scala, Python and R, and an optimized engine that supports general execution graphs.
- It also supports a rich set of higher-level tools including:
  - Spark SQL for SQL and structured data processing,
  - MLlib for machine learning,
  - GraphX for graph processing,

Spark

- Spark is a fast and general-purpose cluster computing system.
- Also known as a **unified analytics engine for large-scale data processing**.
- It provides high-level APIs in Java, Scala, Python and R, and an optimized engine that supports general execution graphs.
- It also supports a rich set of higher-level tools including:
  - Spark SQL for SQL and structured data processing,
  - MLlib for machine learning,
  - GraphX for graph processing,
  - and Spark Streaming.

Spark

- Spark is a fast and general-purpose cluster computing system.
- Also known as a **unified analytics engine for large-scale data processing**.
- It provides high-level APIs in Java, Scala, Python and R, and an optimized engine that supports general execution graphs.
- It also supports a rich set of higher-level tools including:
  - Spark SQL for SQL and structured data processing,
  - MLlib for machine learning,
  - GraphX for graph processing,
  - and Spark Streaming.
- For more check `https://spark.apache.org/`

Spark

- Spark collaborates with Hadoop which is a popular open-source implementation of MapReduce.

Spark

- Spark collaborates with Hadoop which is a popular open-source implementation of MapReduce.
- Hadoop works in a master/slave architecture for both distributed storage and distributed computation.

Spark

- Spark collaborates with Hadoop which is a popular open-source implementation of MapReduce.
- Hadoop works in a master/slave architecture for both distributed storage and distributed computation.
- Hadoop Distributed File System (HDFS) is responsible for distributed storage.

Installation of Spark

- Download Spark from

Installation of Spark

- Download Spark from

  `http://spark.apache.org/downloads.html`

Installation of Spark

- Download Spark from

  `http://spark.apache.org/downloads.html`
- Untar the spark archive:

Installation of Spark

- Download Spark from

  `http://spark.apache.org/downloads.html`
- Untar the spark archive:

  `tar xvfz spark-2.2.0-bin-hadoop2.7.tar`

Installation of Spark

- Download Spark from

  `http://spark.apache.org/downloads.html`
- Untar the spark archive:

  `tar xvfz spark-2.2.0-bin-hadoop2.7.tar`
- To play with Spark there is no need to install HDFS ...

Installation of Spark

- Download Spark from

  `http://spark.apache.org/downloads.html`
- Untar the spark archive:

  `tar xvfz spark-2.2.0-bin-hadoop2.7.tar`
- To play with Spark there is no need to install HDFS ...
- But, you can try to play around with HDFS.

HDFS

- Create new directories:

HDFS

- Create new directories:

```
hdfs dfs -mkdir /user
```

HDFS

- Create new directories:

```
hdfs dfs -mkdir /user
hdfs dfs -mkdir /user/myname
```

HDFS

- Create new directories:

```
hdfs dfs -mkdir /user
hdfs dfs -mkdir /user/myname
```

- Copy the input files into the distributed filesystem:

HDFS

- Create new directories:

```
hdfs dfs -mkdir /user
hdfs dfs -mkdir /user/myname
```

- Copy the input files into the distributed filesystem:

```
hdfs dfs -put data.txt /user/myname/data.txt
```

HDFS

- Create new directories:

```
hdfs dfs -mkdir /user
hdfs dfs -mkdir /user/myname
```

- Copy the input files into the distributed filesystem:

```
hdfs dfs -put data.txt /user/myname/data.txt
```

- View the files in the distributed filesystem:

HDFS

- Create new directories:

```
hdfs dfs -mkdir /user
hdfs dfs -mkdir /user/myname
```

- Copy the input files into the distributed filesystem:

```
hdfs dfs -put data.txt /user/myname/data.txt
```

- View the files in the distributed filesystem:

```
hdfs dfs -ls /user/myname/
```

HDFS

- Create new directories:

```
hdfs dfs -mkdir /user
hdfs dfs -mkdir /user/myname
```

- Copy the input files into the distributed filesystem:

```
hdfs dfs -put data.txt /user/myname/data.txt
```

- View the files in the distributed filesystem:

```
hdfs dfs -ls /user/myname/
hdfs dfs -cat /user/myname/data.txt
```

WordCount in Hadoop

```
import java.io.IOException;
import java.util.StringTokenizer;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCount {

 public static class TokenizerMapper
       extends Mapper<Object, Text, Text, IntWritable>{

    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(Object 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);
      }
    }
  }
  (...)
```

WordCount in Hadoop

```
  (...)
  public static class IntSumReducer
       extends Reducer<Text,IntWritable,Text,IntWritable> {
    private IntWritable result = new IntWritable();

    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 static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "word count");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

WordCount in Spark

- The same code is much simpler in Spark
- To run the Spark shell type: `./bin/spark-shell`
- The code

```
val textFile = sc.textFile("~/data/all-bible.txt")
val counts = textFile.flatMap(line => line.split(" ")).
                 map(word => (word, 1)).
                 reduceByKey(_ + _)
counts.saveAsTextFile("~/data/all-bible-counts.txt")
```

Alternatively:

```
val textFile = spark.read.textFile("~/data/all-bible.txt")
val wordCounts = textFile.flatMap(line => line.split(" ")).
                 groupByKey(identity).
                 count()
```

Matrix-vector multiplication in Spark

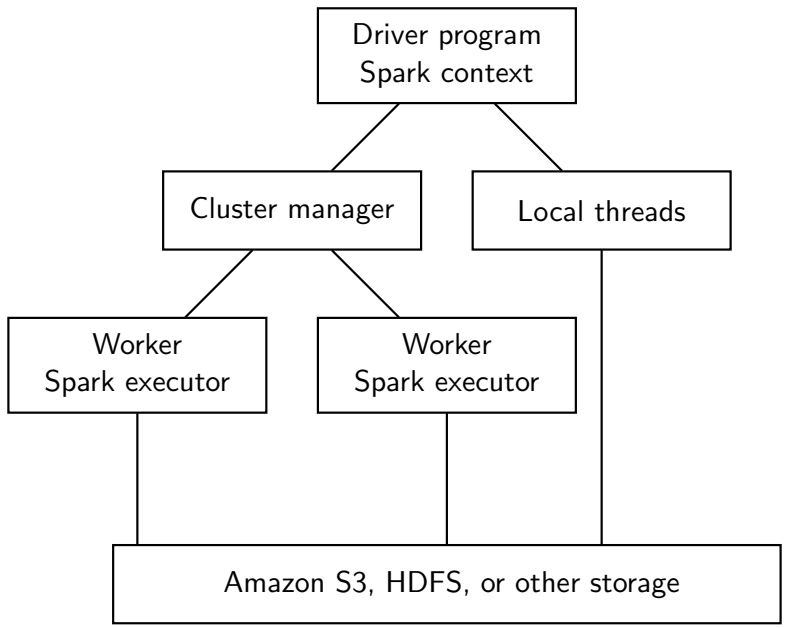
- The Spark code is quite simple:

```
val x = sc.textFile("~/data/x.txt").map(line => {val t = line.split(","); (t(0).
    trim.toInt, t(1).trim.toDouble)})
val vectorX = x.map{case (i,v) => v}.collect
val broadcastedX = sc.broadcast(vectorX)
val matrix = sc.textFile("~/data/M.txt").map(line => {val t = line.split(","); (t
    (0).trim.toInt, t(1).trim.toInt, t(2).trim.toDouble)})
val v = matrix.map { case (i,j,a) => (i, a * broadcastedX.value(j-1))}.reduceByKey(
    _ + _)
v.toDF.orderBy("_1").show
```

Outline

Programming in Spark

- Spark uses in-memory storage for storing immediate results, while Hadoop stores data on disk.
- A Spark program consists of two parts:
  - A driver program: runs on *your* machine.
  - Worker programs: run on cluster nodes or in local threads.
- A Spark program first creates a `SparkContext` object that tells how to access a cluster



Programming in Spark

- Three types of APIs:
  - RDD: an immutable collection of elements partitioned across the nodes of the cluster.
  - Dataset: a strongly-typed, distributed and immutable collection of data that can benefit of the optimized execution engine.
  - Dataframe: an immutable distributed collection of data organized into named columns (implemented as Dataset of type `Row`).

Resilient Distributed Datasets

- RDDs are immutable, distributed, lazy, and compile-time type-safe based on Scala collections API.
- They track lineage information to efficiently recompute lost data.
- Enable operations on collection of elements in parallel.
- Construction of RDD:
  - Parallelization of an existing collection in the driver program,
  - By transforming an existing RDDs,
  - From files in HDFS or any other storage system.
- The number of partitions is to be set by a programmer.

Programming in Spark

- Two types of operations:
  - Transformations: create a new dataset from an existing one in the lazy manner (do not run computations on data immediately).
  - Actions: return a value to the driver program after running a computation on the dataset or storing the results to the file system.

Transformations

- Transformations are recipes for creating a result.
- Lazy evaluation: results not computed right away – instead Spark remembers set of transformations applied to the base dataset.
- Spark optimizes the required calculations.
- Spark recovers from failures and slow workers.
- Examples:
  - `map`, `flatMap`
  - `filter`
  - `distinct`
  - `union`, `intersection`
  - `join`, `cartesian`
  - `reduceByKey`, `groupByKey`, `sortByKey` ($\Leftarrow$ MapReduce-style operations working on pair RDDs)

Actions

- Actions cause Spark to execute recipe to transform input data.
- Examples:
  - `reduce`,
  - `collect`,
  - `count`,
  - `first`, `take(1)`, `take(n)`,
  - `saveAsTextFile`, `saveAsSequenceFile`,
  - `countByKey`,
  - `foreach(func)`.

Caching of results

- One of the most important capabilities in Spark is persisting (or caching) a dataset in memory across operations.
- When you persist an RDD, each node stores any partitions of it that it computes in memory and reuses them in other actions on that dataset (or datasets derived from it).
- To persist RDD use the `persist()` or `cache()` methods on it.

```
val textFile = sc.textFile("~/data/all-shakespeare.txt")
textFile.count()
textFile.count()
textFile.cache()
textFile.count()
textFile.count()
```

Lifecycle of Spark Program

- Create RDDs from external data or parallelize a collection in your driver program,
- Lazily transform them into new RDDs,
- Cache some RDDs for reuse.
- Perform actions to execute parallel computation and produce results.

Closure

- Spark automatically creates closures for:
  - Functions that run on RDDs at workers.
  - Any global variables used by those workers.
- One closure per worker:
  - Sent for every task.
  - No communication between workers.
  - Changes to global variables at workers are not sent to driver.

Shared Variables

- Broadcast Variables:
  - Efficiently send large, read-only value to all workers.
  - Saved at workers for use in one or more Spark operations.
  - Like sending a large, read-only lookup table to all the nodes.
- Example:

```
scala> val broadcastVar = sc.broadcast(Array(1, 2, 3))
broadcastVar: org.apache.spark.broadcast.Broadcast[Array[Int]]
    = Broadcast(0)

scala> broadcastVar.value
res0: Array[Int] = Array(1, 2, 3)
```

Shared Variables

- Accumulators:
  - Aggregate values from workers back to driver.
  - Only driver can access value of accumulator.
  - For tasks, accumulators are write-only.
  - Use to count errors seen in RDD across workers.
- Example:

```
scala> val accum = sc.longAccumulator("My Accumulator")
accum: org.apache.spark.util.LongAccumulator = LongAccumulator
    (id: 0, name: Some(My Accumulator), value: 0)

scala> sc.parallelize(Array(1, 2, 3, 4)).foreach(x => accum.
    add(x))
...

scala> accum.value
res2: Long = 10
```

Let us check some code

- Word count I:

```
val textFile = sc.textFile("~/data/all-bible.txt")
val counts = (textFile.flatMap(line => line.split(" "))
                      .map(word => (word, 1))
                      .reduceByKey(_ + _))
counts.saveAsTextFile("~/data/all-bible-counts.txt")
```

Let us check some code

- Matrix-vector multiplication:

```
val x = sc.textFile("~/data/x.txt").map(line => {val t = line.
    split(","); (t(0).trim.toInt, t(1).trim.toDouble)})
val vectorX = x.map{case (i,v) => v}.collect
val broadcastedX = sc.broadcast(vectorX)
val matrix = sc.textFile("~/data/M.txt").map(line => {val t =
    line.split(","); (t(0).trim.toInt, t(1).trim.toInt, t(2).
    trim.toDouble)})
val v = matrix.map { case (i,j,a) => (i, a * broadcastedX.
    value(j-1))}.reduceByKey(_ + _)
v.toDF.orderBy("_1").show
```

Dataframes and Datasets

- Alternative for RDDs
- Rather *What* than *How* programming style $\Rightarrow$ More optimizations possible
- Datasets are strongly typed, but Dataframes not.
- They use the `SqlContext`.
- SQL-like queries: either from Scala or SQL.

Dataframes and Datasets

- A sample code:

```
val df = spark.read.json("examples/src/main/resources/people.json")

// Displays the content of the DataFrame to stdout
df.show()
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+

df.printSchema()
// root
// |-- age: long (nullable = true)
// |-- name: string (nullable = true)

// Select only the "name" column
df.select("name").show()
// +-------+
// |   name|
// +-------+
// |Michael|
// |   Andy|
// | Justin|
// +-------+
```

Dataframes and Datasets

- One can also use SQL directly:

```
df.createOrReplaceTempView("people")

val sqlDF = spark.sql("SELECT * FROM people")
sqlDF.show()
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+
```

Dataframes and Datasets

- Creating dataframes and datasets:

```
case class Person(name: String, age: Long)

val path = "examples/src/main/resources/people.json"
val peopleDS = spark.read.json(path).as[Person]

val primitiveDS = Seq(1, 2, 3).toDS()
primitiveDS.map(_ + 1).collect() // Returns: Array(2, 3, 4)

// Create an RDD of Person objects from a text file, convert it to a Dataframe
val peopleDF = spark.sparkContext
  .textFile("examples/src/main/resources/people.txt")
  .map(_.split(","))
  .map(attributes => Person(attributes(0), attributes(1).trim.toInt))
  .toDF()
```

Dataframes and Datasets

- Data sources:
  - The default data source is `parquet`, which is highly efficient columnar format.
  - Other data sources are also supported like `json`, databases via `jdbc`, `hive` databases, and many others.

```
val usersDF = spark.read.load("examples/src/main/resources/users.parquet")
usersDF.select("name", "favorite_color").write.save("namesAndFavColors.parquet")

val peopleDF = spark.read.format("json").load("examples/src/main/resources/people.
    json")
peopleDF.select("name", "age").write.format("parquet").save("namesAndAges.parquet")
```

- For file-based data source, it is also possible to bucket and sort or partition the output.

```
peopleDF
  .write
  .partitionBy("favorite_color")
  .bucketBy(42, "name")
  .saveAsTable("people_partitioned_bucketed")
```

Let us check some code

- Dataframes and Datasets:

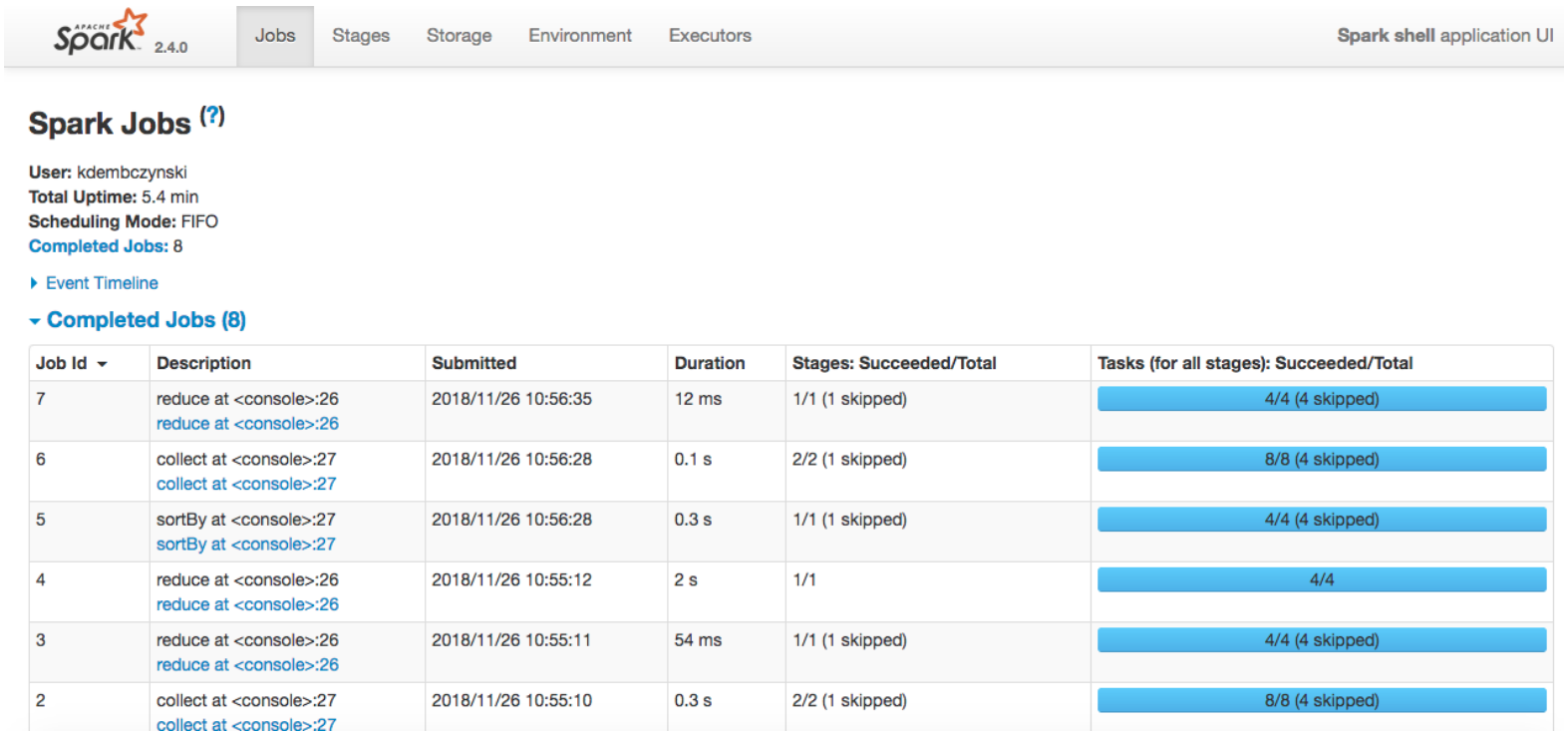
```
val songs = spark.read.
                  option("delimiter", ",").
                  csv("songs").
                  toDF("song_id", "track_long_id", "song_long_id", "artist", "song"
    )

val facts = spark.read.
                option("delimiter", ",").
                csv("facts").
                toDF("id", "user_id", "song_id", "date_id")

facts.groupBy("song_id").
      count.
      join(songs, facts("song_id")===songs("song_id")).
      select("song", "count").
      orderBy(desc("count")).
      show(10)
```

Monitoring Spark

- Every SparkContext launches a web UI, by default on port 4040.
- It displays useful information about the application:
  - A list of scheduler stages and tasks,
  - A summary of RDD sizes and memory usage,
  - Environmental information,
  - Information about the running executors.
- To access the interface, you can open in your web browser the following page:

  `http://<driver-node>:4040` (e.g. `http://localhost:4040`)
- If multiple SparkContexts are running on the same host, they will bind to successive ports beginning with 4040 (4041, 4042, etc).

Monitoring Spark



Outline

Summary

- MapReduce in Spark
- Resilient Distributed Datasets (RDD)
- Dataframes and Datasets

Bibliography

- J. Leskovec, A. Rajaraman, and J. D. Ullman. *Mining of Massive Datasets*. Cambridge University Press, 2014
  `http://infolab.stanford.edu/~ullman/mmds.html`
- J.Lin and Ch. Dyer. *Data-Intensive Text Processing with MapReduce*. Morgan and Claypool Publishers, 2010
  `http://lintool.github.com/MapReduceAlgorithms/`
- Ch. Lam. *Hadoop in Action*. Manning Publications Co., 2011
- `https://spark.apache.org/docs/`
- Anthony D. Joseph. Introduction to Big Data with Apache Spark, 2016