Data Management in Large-Scale Distributed Systems
Apache Spark

Thomas Ropars

thomas.ropars@univ-grenoble-alpes.fr

http://tropars.github.io/

2018
References

- The lecture notes of V. Leroy
- The lecture notes of Y. Vernaz
In this course

- The basics of Apache Spark
- Spark API
- Start programming with PySpark
Agenda

Introduction to Apache Spark

Spark internals

Programming with PySpark

Additional content
Apache Spark

- Originally developed at Univ. of California
- One of the most popular Big Data project today.
Motivations

Limitations of Hadoop MapReduce

• Limited performance for iterative algorithms
  ▶ Data are flushed to disk after each iteration
  ▶ More generally, low performance for complex algorithms

Main novelties

• Computing in memory
• A new computing abstraction: Resilient Distributed Datasets (RDD)
Motivations

Limitations of Hadoop MapReduce

• Limited performance for iterative algorithms
  ▶ Data are flushed to disk after each iteration
  ▶ More generally, low performance for complex algorithms

Main novelties

• Computing in memory
• A new computing abstraction: Resilient Distributed Datasets (RDD)
Spark vs Hadoop

Spark added value

• Performance
  ▶ Especially for iterative algorithms
• Interactive queries
• Supports more operations on data
• A full ecosystem (High level libraries)
• Running on your machine or at scale
Programming with Spark

Spark Core API
- Scala
- Python
- Java

Integration with storage systems
Works with any storage source supported by Hadoop
- Local file systems
- HDFS
- Cassandra
- Amazon S3
Many resources to get started

- https://spark.apache.org/
- https://sparkhub.databricks.com/
- Many courses, tutorials, and examples available online
Starting with Spark

Running in local mode

• Spark runs in a JVM
  ▶ Spark is coded in Scala
• Read data from your local file system

Use interactive shell

• Scala (spark-shell)
• Python (pyspark)
• Run locally or distributed at scale
A very first example with pyspark

Counting lines

```
Using Python version 3.6.3 (default, Nov 20 2017 20:41:42)
SparkSession available as 'spark'.
>>> lines = sc.textFile("./The_Iliad_by_Homer.txt")
>>> lines.count()
26175
```
The Spark Web UI

Details for Job 0

Status: SUCCEEDED
Completed Stages: 1

Event Timeline
Enable zooming

Executors
- Added
- Removed

Stages
- Completed
- Failed
- Active

DAG Visualization

Completed Stages (1)

<table>
<thead>
<tr>
<th>Stage Id</th>
<th>Description</th>
<th>Submitted</th>
<th>Duration</th>
<th>Tasks: Succeeded/Total</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>count at &lt;stdin&gt; 1</td>
<td>2017/12/03 08:07:55</td>
<td>0.7 s</td>
<td>2/2</td>
<td>1280.9 KB</td>
<td></td>
</tr>
</tbody>
</table>
The Spark built-in libraries

- **Spark SQL**: For structured data (Dataframes)
- **Spark Streaming**: Stream processing (micro-batching)
- **MLlib**: Machine learning
- **GraphX**: Graph processing
Agenda

Introduction to Apache Spark

Spark internals

Programming with PySpark

Additional content
In-memory computing: Insights
See Latency Numbers Every Programmer Should Know

Memory is way faster than disks

Read latency

- HDD: a few milliseconds
- SDD: 10s of microseconds (100X faster than HDD)
- DRAM: 100 nanoseconds (100X faster than SDD)
Cost of memory decreases = More memory per server
Efficient iterative computation

**Hadoop:** At each step, data go through the disks

**Spark:** Data remain in memory (if possible)
Main challenge

Fault Tolerance

Failure is the norm rather than the exception

On a node failure, all data in memory is lost
Resilient Distributed Datasets

Restricted form of distributed shared memory

• Read-only partitioned collection of records

• Creation of a RDD through deterministic operations (transformations) on
  ▶ Data stored on disk
  ▶ an existing RDD
Transformations and actions

Programming with RDDs

- An RDD is represented as an object

- Programmer defines RDDs using **Transformations**
  - Applied to data on disk or to existing RDDs
  - Examples of transformations: map, filter, join

- Programmer uses RDDs in **Actions**
  - Operations that return a value or export data to the file system
  - Examples of actions: count, reduce
Fault tolerance with Lineage

Lineage = a description of an RDD

- The data source on disk
- The sequence of applied transformations
  - Same transformation applied to all elements
  - Low footprint for storing a lineage

Fault tolerance

- RDD partition lost
  - Replay all transformations on the subset of input data or the most recent RDD available
- Deal with stragglers
  - Generate a new copy of a partition on another node
Spark runtime
Figure by M. Zaharia et al

- **Driver (= Master)**
  - Executes the user program
  - Defines RDDs and invokes actions
  - Tracks RDD’s lineage

- **Workers**
  - Store RDD partitions
  - Perform transformations and actions
    - Run tasks
Persistence and partitioning

See https://spark.apache.org/docs/latest/rdd-programming-guide.html#rdd-persistence

Different options of persistence for RDDs

• Options:
  ▶ Storage: memory/disk/both
  ▶ Replication: yes/no
  ▶ Serialization: yes/no

Partitions

• RDDs are automatically partitioned based on:
  ▶ The configuration of the target platform (nodes, CPUs)
  ▶ The size of the RDD
  ▶ User can also specify its own partitioning

• Tasks are created for each partition
RDD dependencies

Transformations create dependencies between RDDs.

2 kinds of dependencies

- **Narrow dependencies**
  - Each partition in the parent is used by at most one partition in the child

- **Wide (shuffle) dependencies**
  - Each partition in the parent is used by multiple partitions in the child

Impact of dependencies

- Scheduling: Which tasks can be run independently
- Fault tolerance: Which partitions are needed to recreate a lost partition
RDD dependencies

Figure by M. Zaharia et al.

“Narrow” deps:
- map, filter
- union

“Wide” (shuffle) deps:
- join with inputs co-partitioned
- groupByKey
- join with inputs not co-partitioned
Executing transformations and actions

Lazy evaluation

• Transformations are executed only when an action is called on the corresponding RDD

• Examples of optimizations allowed by lazy evaluation
  ▶ Read file from disk + action first(): no need to read the whole file
  ▶ Read file from disk + transformation filter(): No need to create an intermediate object that contains all lines
Persist a RDD

- By default, a RDD is recomputed for each action run on it.
- A RDD can be cached in memory calling `persist()` or `cache()`
  - Useful is multiple actions to be run on the same RDD (iterative algorithms)
  - Can lead to 10X speedup
  - Note that a call to persist does not trigger transformations evaluation
  - `cache()` mean that data have to be persisted in memory
Job scheduling

Main ideas

• Tasks are run when the user calls an action

• A Directed Acyclic Graph (DAG) of transformations is built based on the RDD’s lineage

• The DAG is divided into stages. Boundaries of a stage defined by:
  ▶ Wide dependencies
  ▶ Already computed RDDs

• Tasks are launch to compute missing partitions from each stage until target RDD is computed
  ▶ Data locality is taken into account when assigning tasks to workers
Stages in a RDD’s DAG

Figure by M. Zaharia et al

Cached partitions in black
Agenda

Introduction to Apache Spark

Spark internals

Programming with PySpark

Additional content
The SparkContext

What is it?

• Object representing a connection to an execution cluster
• We need a SparkContext to build RDDs

Creation

• Automatically created when running in shell (variable `sc`)
• To be initialized when writing a standalone application

Initialization

• Run in local mode with nb threads = nb cores: `local[*]`
• Run in local mode with 2 threads: `local[2]`
• Run on a spark cluster: `spark://HOST:PORT`
The SparkContext

Python shell

$ pyspark --master local[*]

Python program

import pyspark

sc = pyspark.SparkContext("local[*]"
The first RDDs

Create RDD from existing iterator

• Use of SparkContext.parallelize()
• Optional second argument to define the number of partitions

```
data = [1, 2, 3, 4, 5]
distData = sc.parallelize(data)
```

Create RDD from a file

• Use of SparkContext.textFile()

```
data = sc.textFile("myfile.txt")
hdfsData = sc.textFile("hdfs://myhdfsfile.txt")
```
Some transformations
see https://spark.apache.org/docs/latest/rdd-programming-guide.html#transformations

• map(f): Applies f to all elements of the RDD. f generates a single item

• flatMap(f): Same as map but f can generate 0 or several items

• filter(f): New RDD with the elements for which f return true

• union(other)/intersection(other): New RDD being the union/intersection of the initial RDD and other.

• cartesian(other): When called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements)

• distinct(): New RDD with the distinct elements

• repartition(n): Reshuffle the data in the RDD randomly to create either more or fewer partitions and balance it across them
Some transformations with $<K,V>$ pairs

- **groupByKey()**: When called on a dataset of $(K, V)$ pairs, returns a dataset of $(K, \text{Iterable}<V>)$ pairs.

- **reduceByKey(f)**: When called on a dataset of $(K, V)$ pairs, Merge the values for each key using an associative and commutative reduce function.

- **aggregateByKey()**: see documentation

- **join(other)**: 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.
Some actions

see
https://spark.apache.org/docs/latest/rdd-programming-guide.html#actions

• **reduce(f)**: Aggregate the elements of the dataset using \( f \) (takes two arguments and returns one).

• **collect()**: Return all the elements of the dataset as an array.

• **count()**: Return the number of elements in the dataset.

• **take(n)**: Return an array with the first \( n \) elements of the dataset.

• **takeSample()**: Return an array with a random sample of \( num \) elements of the dataset.

• **countByKey()**: Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key.
Additional references

Mandatory reading


Suggested reading

•
Agenda

Introduction to Apache Spark

Spark internals

Programming with PySpark

Additional content
from pyspark.context import SparkContext
sc = SparkContext("local")

# define a first RDD
lines = sc.textFile("data.txt")
# define a second RDD
lineLengths = lines.map(lambda s: len(s))
# Make the RDD persist in memory
lineLengths.persist()
# At this point no transformation has been run
# Launch the evaluation of all transformations
totalLength = lineLengths.reduce(lambda a, b: a + b)
An example with key-value pairs

```python
lines = sc.textFile("data.txt")
pairs = lines.map(lambda s: (s, 1))
counts = pairs.reduceByKey(lambda a, b: a + b)

# Warning: sortByKey implies shuffle
result = counts.sortByKey().collect()
```
Another example with key-value pairs

```python
rdd = sc.parallelize([("a", 1), ("b", 1), ("a", 1)])
# `mapValues` applies `f` to each value
# without changing the key
sorted(rdd.groupByKey().mapValues(len).collect())
# [(‘a’, 2), (‘b’, 1)]
sorted(rdd.groupByKey().mapValues(list).collect())
# [(‘a’, [1, 1]), (‘b’, [1])]"
Shared Variables

see https://spark.apache.org/docs/latest/rdd-programming-guide.html#shared-variables

Broadcast variables

- Use-case: A read-only large variable should be made available to all tasks (e.g., used in a map function)
- Costly to be shipped with each task
- Declare a broadcast variable
  - Spark will make the variable available to all tasks in an efficient way
Example with a Broadcast variable

```python
b = sc.broadcast([1, 2, 3, 4, 5])
print(b.value)
# [1, 2, 3, 4, 5]
print(sc.parallelize([0, 0]).flatMap(lambda x: b.value).collect())
# [1, 2, 3, 4, 5, 1, 2, 3, 4, 5]
b.unpersist()
```
Shared Variables

Accumulator

• Use-case: Accumulate values over all tasks
• Declare an Accumulator on the driver
  ▶ Updates by the tasks are automatically propagated to the driver.
• Default accumulator: operator '\+=' on int and float.
  ▶ User can define custom accumulator functions
Example with an Accumulator

```python
file = sc.textFile(inputFile)
# Create Accumulator[Int] initialized to 0
blankLines = sc.accumulator(0)

def splitLine(line):
    # Make the global variable accessible
    global blankLines
    if not line:
        blankLines += 1
    return line.split("␣")

words = file.flatMap(splitLine)
print(blankLines.value)
```