Data Management in Large-Scale Distributed Systems

Apache Spark

Thomas Ropars

thomas.ropars@univ-grenoble-alpes.fr

http://tropars.github.io/

2022

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
- *Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing*, M. Zaharia et al. NSDI, 2012.
- One of the most popular Big Data project today.

Motivations

Limitations of Hadoop MapReduce

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 of Spark

- 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

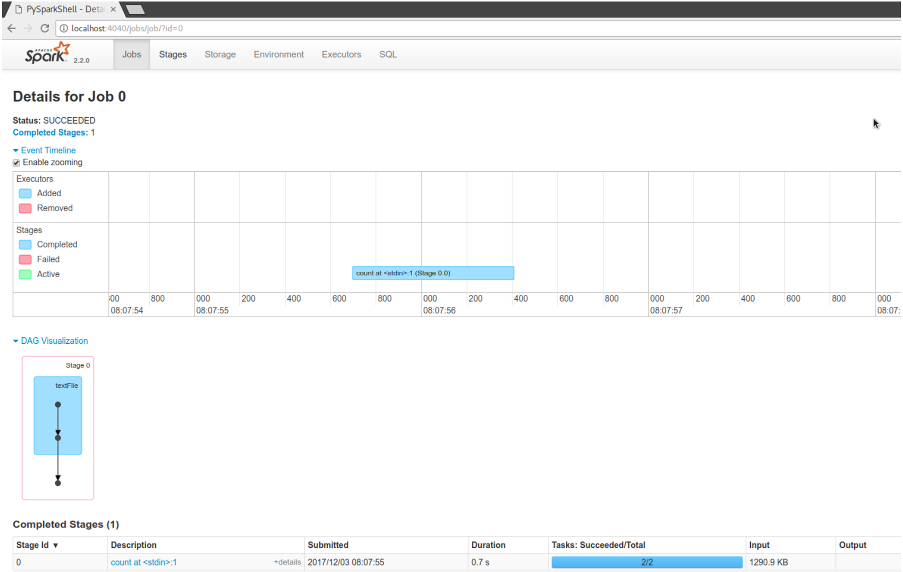
A very first example with pyspark

Counting lines

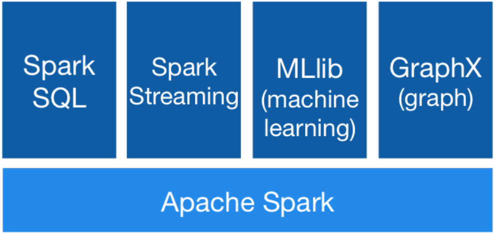
File Edit View Search Terminal Help

```
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 2.2.0
      /_/

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



The Spark built-in libraries



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

Agenda

In-memory computing: Insights

See Latency Numbers Every Programmer Should Know

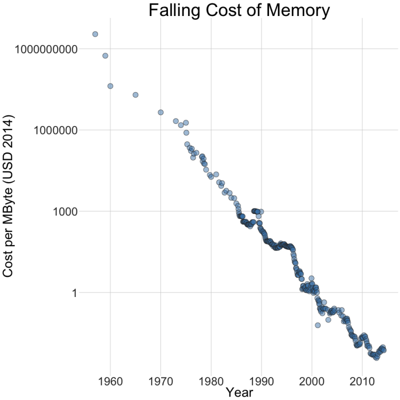
Memory is way faster than disks

Read latency

- HDD: a few milliseconds
- SDD: 10s of microseconds (100X faster than HDD)
- DRAM: 100 nanoseconds (100X faster than SDD)

In-memory computing: Insights

Graph by P. Johnson

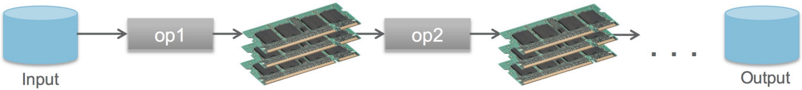


Cost of memory decreases = More memory per server

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

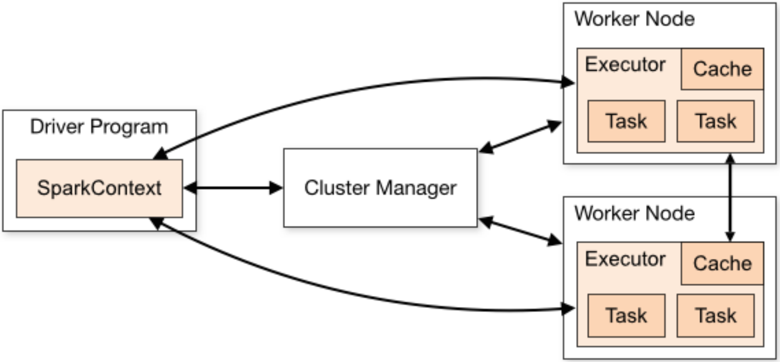
- The data source on disk
- The sequence of applied transformations
  - Same transformation applied to all elements
  - Low footprint for storing a lineage

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

see https://spark.apache.org/docs/latest/cluster-overview.html



Spark runtime

- Cluster Manager: The system in charge of allocating resources to applications
- Worker nodes: Nodes of the cluster on which the Spark applications are run
- Driver: Main program of a spark application
  - Created when an application is submitted
  - Translates the user's program into a graph of tasks
  - Assigns tasks to executors
- Executor: A dedicated process (a new JVM) created on a worker to execute an application
  - Created when an application is submitted
    - By default a Spark apps tries to use all resources of the cluster
    - One executor per worker – An executor uses all cores of the worker
  - Can include multiple executor threads
  - Execute tasks on partitions

Partitioning

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

Partitions are the unit of parallelism in Spark

- RDDs are divided into partitions
- To execute an operation on a RDD, a task per partition is created
- Tasks can be executed in parallel

Partitions and executors

- All items of one partition are on the same executor
- An executor can process multiple partitions

More on partitioning

See https://luminousmen.com/post/spark-partitions

Number of partitions

- RDDs are automatically partitioned based on the configuration of the target platform (nodes, CPUs)
  - As many partitions as the number of available cores
- If the input data are already partitioned:
  - Same number of partitions as in the input data
  - Example: RDD from HDFS file – 1 partition per HDFS block
- The number of partitions in a RDD can be changed by the programmer
  - repartition(): change the number of partitions
  - coalesce(): merge partitions

Distribution of data in partitions

Two default partitioners

- Range partitioner
  - Default partitioner for raw data
  - Consecutive items are put in the same partition
- Hash partitioner
  - Applied after "ByKey" operations
  - $partition = key.hashCode() \mod numPartitions$
- The user can define its own partitioning function

RDD dependencies

Transformations create dependencies between RDDs.

2 kinds of dependencies

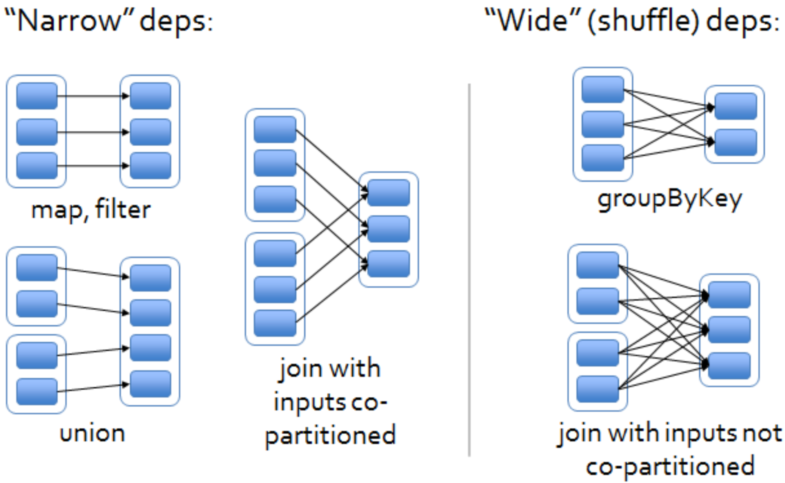
- Narrow dependencies
  - Each partition in the parent is used by at most one partition in the child
- Wide (shuffle) dependencies
  - Each partition in the parent is used by multiple partitions in the child

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



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

About shuffle operations

Costly operations

- Triggered by:
  - *ByKey* operations
  - repartition operations
  - etc.

- May involves significant communication over the network

- Involves disk I/O operations
  - In each source partition, data split by destination partitions are saved to disk.
  - Purpose: limit the number of operations to re-execute in case of crash

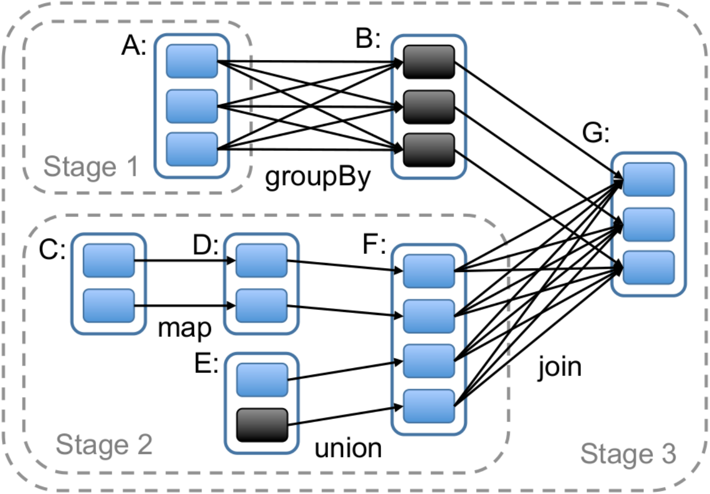
Job scheduling

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 launched 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

Persist a RDD

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

Main idea

- 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()` means that data have to be persisted in memory

Persist a RDD

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

Different options

- `MEMORY_ONLY`: RDDs stored in memory as deserialized objects (default)
- `MEMORY_AND_DISK`: Move data to disk if not enough space in memory
- `MEMORY_ONLY_SER`: serialize data
- etc.

Agenda

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, 2)
```

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* returns *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, 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.
- foreach(f): Run function *f* on each element (*f* usually has side effects such as writing to external storage)

An example

```python
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.cache()
# 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

```
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

```
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])]
```

About distributed execution

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

```
counter = 0
rdd = sc.parallelize(data)

def increment_counter(x):
    global counter
    counter += x

rdd.foreach(increment_counter)

print("Counter value: ", counter) # displays 0
```

What is the problem?

About distributed execution

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

```
1      counter = 0
2      rdd = sc.parallelize(data)
3
4      def increment_counter(x):
5          global counter
6          counter += x
7
8      rdd.foreach(increment_counter)
9
10     print("Counter␣value:␣", counter) # displays 0
```

What is the problem?

- We have multiple JVMs, and so, multiple *counter* variables
  - *counter* in lines 1 and 10 is in the JVM of the driver
  - In lines 5 and 8, we create one counter per executor JVM

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

```
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)
```

Additional references

Mandatory reading

- Chapter 5 on *Large-Scale Dataflow Engines* of the redbook (*Readings in Database Systems, 5th Edition*), P. Bailis, J. Hellerstein, M. Stonebraker – `http://www.redbook.io/ch5-dataflow.html`

Agenda

Introduction to Apache Spark

Spark internals

Programming with PySpark

Additional content

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

```
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()
```