Introduction to Apache Spark

Thomas Ropars

thomas.ropars@univ-grenoble-alpes.fr

http://tropars.github.io/

2018

References

The content of this lectures is inspired by:

- The lecture notes of Yann Vernaz.
- The lecture notes of Vincent Leroy.
- The lecture notes of Renaud Lachaize.
- The lecture notes of Henggang Cui.

Goals of the lecture

- Present the main challenges associated with distributed computing
- Review the MapReduce programming model for distributed computing
  - Discuss the limitations of Hadoop MapReduce
- Learn about Apache Spark and its internals
- Start programming with PySpark

Agenda

Agenda

- Computing at large scale
- Programming distributed systems
- MapReduce
- Introduction to Apache Spark
- Spark internals
- Programming with PySpark

Distributed computing: Definition

A distributed computing system is a system including several computational entities where:

- Each entity has its own local memory
- All entities communicate by message passing over a network

Each entity of the system is called a node.

Distributed computing: Motivation

There are several reasons why one may want to distribute data and processing:

- Scalability
  - The data do not fit in the memory/storage of one node
  - The processing power of more processor can reduce the time to solution
- Fault tolerance / availability
  - Continuing delivering a service despite node crashes.
- Latency
  - Put computing resources close to the users to decrease latency

Increasing the processing power

Goals

- Increasing the amount of data that can be processed (weak scaling)
- Decreasing the time needed to process a given amount of data (strong scaling)

Two solutions

- Scaling up
- Scaling out

Vertical scaling (scaling up)

Idea

Increase the processing power by adding resources to existing nodes:

- Upgrade the processor (more cores, higher frequency)
- Increase memory capacity
- Increase storage capacity

Pros and Cons

Vertical scaling (scaling up)

Idea

Increase the processing power by adding resources to existing nodes:

- Upgrade the processor (more cores, higher frequency)
- Increase memory capacity
- Increase storage capacity

Pros and Cons

☺ Performance improvement without modifying the application

☹ Limited scalability (capabilities of the hardware)

☹ Expensive (non linear costs)

Horizontal scaling (scaling out)

Idea

Increase the processing power by adding more nodes to the system

- Cluster of commodity servers

Pros and Cons

Horizontal scaling (scaling out)

Idea

Increase the processing power by adding more nodes to the system

- Cluster of commodity servers

Pros and Cons

☹ Often requires modifying applications

☺ Less expensive (nodes can be turned off when not needed)

☺ *Infinite* scalability

Horizontal scaling (scaling out)

Idea

Increase the processing power by adding more nodes to the system

- Cluster of commodity servers

Pros and Cons

☹ Often requires modifying applications

☺ Less expensive (nodes can be turned off when not needed)

☺ *Infinite* scalability

Main focus of this lecture

Large scale infrastructures

Figure: Google Data-center

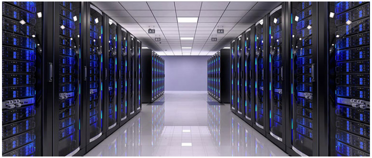


Figure: Amazon Data-center

Figure: Barcelona Supercomputing Center

Programming for large-scale infrastructures

Challenges

- Performance
  - How to take full advantage of the available resources?
  - Moving data is costly
    - How to maximize the ratio between computation and communication?
- Scalability
  - How to take advantage of a large number of distributed resources?
- Fault tolerance
  - The more resources, the higher the probability of failure
  - MTBF (Mean Time Between Failures)
    - MTBF of one server = 3 years
    - MTBF of 1000 servers $\simeq$ 19 hours (beware: over-simplified computation)

Programming in the Clouds

Cloud computing

- A service provider gives access to computing resources through an internet connection.

Pros and Cons

Programming in the Clouds

Cloud computing

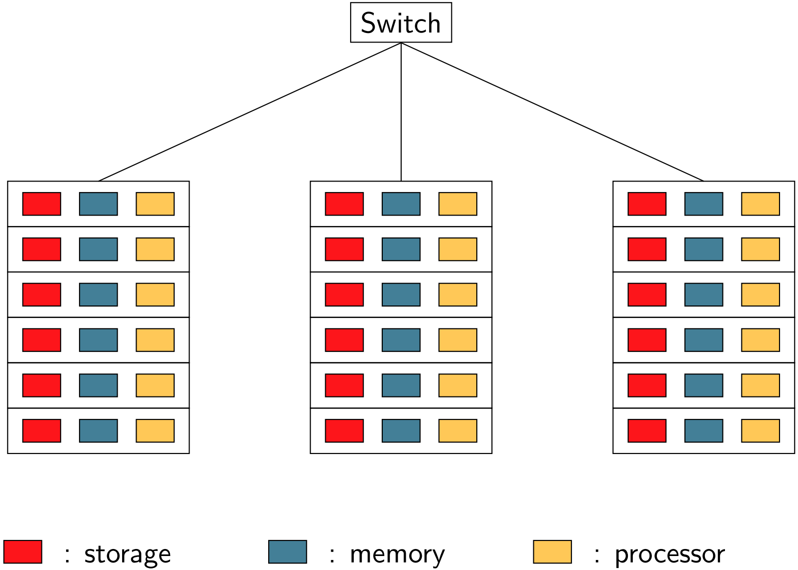
- A service provider gives access to computing resources through an internet connection.

Pros and Cons

- ☺ Pay only for the resources you use
- ☺ Get access to large amount of resources
  - Amazon Web Services features millions of servers
- ☹ Volatility
  - Low control on the resources
  - Example: Access to resources based on bidding
  - See "The Netflix Simian Army"
- ☹ Performance variability
  - Physical resources shared with other users

Architecture of a data center

Simplified



Architecture of a data center

A shared-nothing architecture

- Horizontal scaling
- No specific hardware

A hierarchical infrastructure

- Resources clustered in racks
- Communication inside a rack is more efficient than between racks
- Resources can even be geographically distributed over several datacenters

A warning about distributed computing

You can have a second computer once you've shown you know how to use the first one. (P. Braham)

Horizontal scaling is very popular.

- But not always the most efficient solution (both in time and cost)

Examples

- Processing a few 10s of GB of data is often more efficient on a single machine that on a cluster of machines
- Sometimes a single threaded program outperforms a cluster of machines (F. McSherry et al. "Scalability? But at what COST!". 2015.)

Agenda

Computing at large scale

Programming distributed systems

MapReduce

Introduction to Apache Spark

Spark internals

Programming with PySpark

Summary of the challenges

Context of execution

- Large number of resources
- Resources can crash (or disappear)
  - Failure is the norm rather than the exception.
- Resources can be slow

Objectives

- Run until completion
  - And obtain a correct result :-)
- Run fast

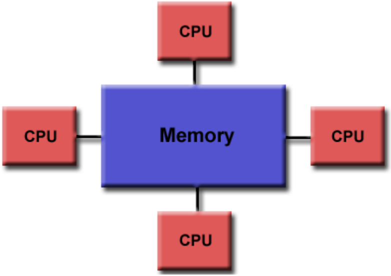
Shared memory and message passing

Two paradigms for communicating between computing entities:

- Shared memory
- Message passing

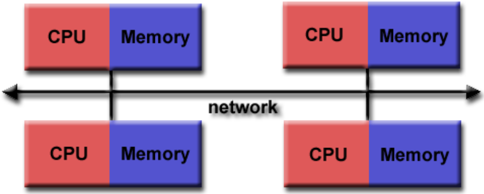
Shared memory

- Entities share a global memory
- Communication by reading and writing to the globally shared memory
- Examples: Pthreads, OpenMP, etc



Message passing

- Entities have their own private memory
- Communication by sending/receiving messages over a network
- Example: MPI



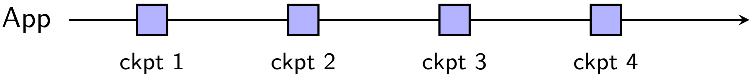
Dealing with failures: Checkpointing

Checkpointing

App

Dealing with failures: Checkpointing

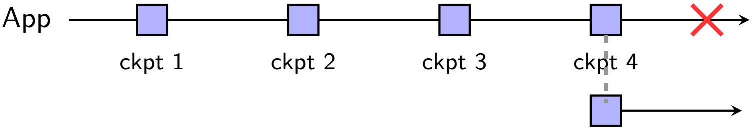
Checkpointing



- Saving the complete state of the application periodically

Dealing with failures: Checkpointing

Checkpointing



- Saving the complete state of the application periodically
- Restart from the most recent checkpoint in the event of a failure.

About checkpointing

Main solution when processes can apply fine-grained modifications to the data (Pthreads or MPI)

- A process can modify any single byte independently
- Impossible to log all modifications

Limits

- Performance cost
- Difficult to implement
- The alternatives (passive or active replication) are even more costly and difficult to implement in most cases

About slow resources (stragglers)

Performance variations

- Both for the nodes and the network
- Resources shared with other users

Impact on classical message-passing systems (MPI)

- Tightly-coupled processes
  - *Process A waits for a message from process B before continuing its computation*

```
Do some computation
new_data = Recv(from B) /*blocking*/
Resume computing with new_data
```

Figure: Code of process A. If B is slow, A becomes idle.

The Big Data approach

Provide a distributed computing execution framework

- Simplify parallelization
  - Define a programming model
  - Handle distribution of the data and the computation
- Fault tolerant
  - Detect failure
  - Automatically takes corrective actions
- **Code once (expert), benefit to all**

Limit the operations that a user can run on data

- Inspired from functional programming (eg, MapReduce)
- Examples of frameworks:
  - Hadoop MapReduce, Apache Spark, Apache Flink, etc

Agenda

MapReduce at Google

References

- *The Google file system*, S. Ghemawat et al. SOSP 2003.
- *MapReduce: simplified data processing on large clusters*, D. Jeffrey and S. Ghemawat. OSDI 2004.

Main ideas

- Data represented as key-value pairs
- Two main operations on data: Map and Reduce
- A distributed file system
  - Compute where the data are located

Use at Google

- Compute the index of the World Wide Web.
- Google has moved on to other technologies

Apache Hadoop

Apache Hadoop

In a few words

- Built on top of the ideas of Google
- A full data processing stack
- The core elements
  - A distributed file system: HDFS (Hadoop Distributed File System)
  - A programming model and execution framework: Hadoop MapReduce

MapReduce

- Allows simply expressing many parallel/distributed computational algorithms

MapReduce

The Map operation

- Transformation operation
- $\text{map}(f)[x_0, ..., x_n] = [f(x_0), ..., f(x_n)]$
- $\text{map}(*2)[2, 3, 6] = [4, 6, 12]$

The Reduce operation

- Aggregation operation (fold)
- $\text{reduce}(f)[x_0, ..., x_n] = [f((x_0), f((x_1), ..., f(x_{n-1}, x_n)))]$
- $\text{reduce}(+)[2, 3, 6] = (2 + (3 + 6)) = 11$

Hadoop MapReduce

Key/Value pairs

- MapReduce manipulate sets of Key/Value pairs
- Keys and values can be of any types

Functions to apply

- The user defines the functions to apply
- In Map, the function is applied independently to each pair
- In Reduce, the function is applied to all values with the same key

Hadoop MapReduce

About the Map operation

- A given input pair may map to zero or many output pairs
- Output pairs need not be of the same type as input pairs

About the Reduce operation

- Applies operation to all pairs with the same key
- 3 steps:
  - Shuffle and Sort: Groups and merges the output of mappers by key
  - Reduce: Apply the reduce operation to the new key/value pairs

A first MapReduce program

Word Count

Description

- Input: A set of lines including words
  - Pairs $<$ line number, line content $>$
  - The initial keys are ignored in this example
- Output: A set of pairs $<$ word, nb of occurrences $>$

Input

- $<$ 1, "aaa bb ccc" $>$
- $<$ 2, "aaa bb" $>$

Output

- $<$ "aaa", 2 $>$
- $<$ "bb", 2 $>$
- $<$ "ccc", 1 $>$

A first MapReduce program

Word Count

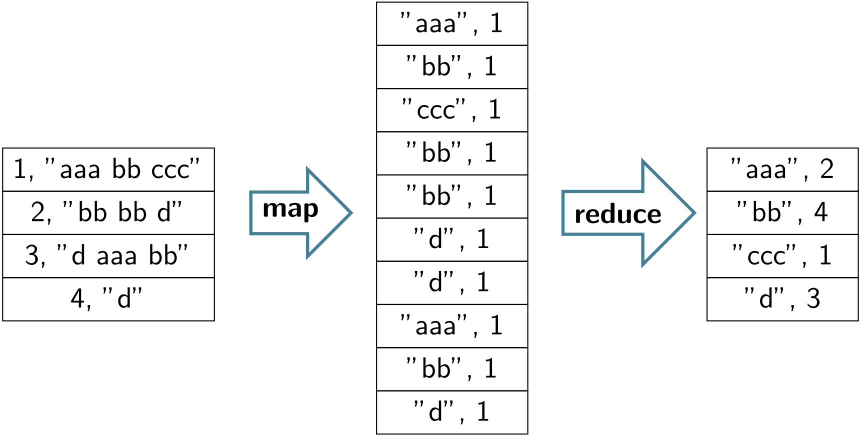
```
map(key, value):  /* pairs of {line num, content} */
  foreach word in value.split():
    emit(word, 1)



reduce(key, values):  /* {word, list nb occurences} */
  result = 0
  for value in values:
    result += value
  emit(key, result) /* -> {word, nb occurences} */
```

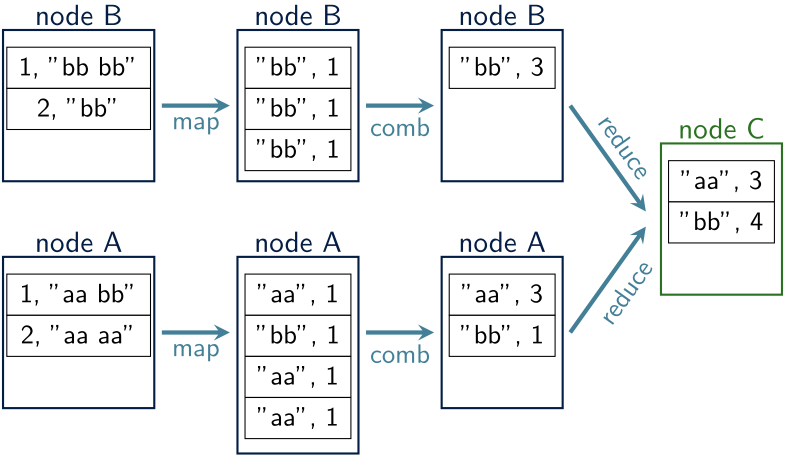
A first MapReduce program

Word Count



Logical representation (no notion of distribution)

Distributed execution of Word Count



Example: Web index

Description

Construct an index of the pages in which a word appears.

- Input: A set of web pages
  - Pairs $<$ URL, content of the page $>$
- Output: A set of pairs $<$ word, set of URLs $>$

Example: Web index

```
map(key, value):  /* pairs of {URL, page_content} */
  foreach word in value.parse():
    emit(word, key)
```

```
reduce(key, values): /* {word, URLs} */
  list=[]
  for value in values:
    list.add(value)
  emit(key, list)  /* {word, list of URLs} */
```

Running at scale

How to distribute data?

- Partitioning
- Replication

Partitioning

- Splitting the data into partitions
- Partitions are assigned to different nodes
- Main goal: Performance
  - Partitions can be processed in parallel

Replication

- Several nodes host a copy of the data
- Main goal: Fault tolerance
  - No data lost if one node crashes

Hadoop Distributed File System (HDFS)

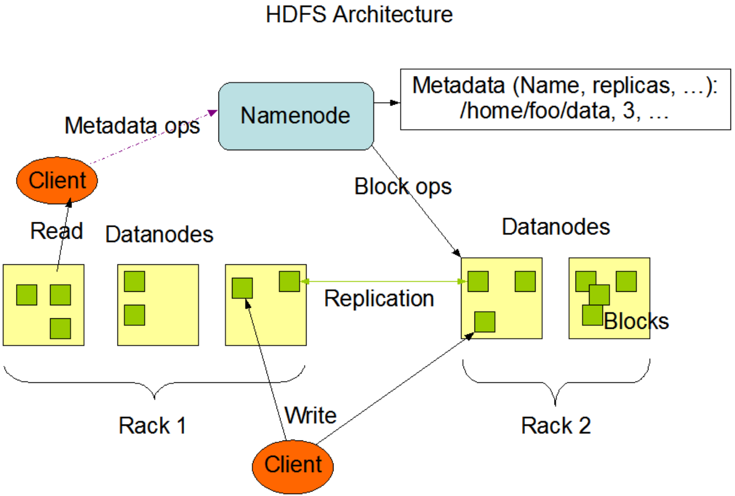
Main ideas

- Running on a cluster of commodity servers
  - Each node has a local disk
  - A node may fail at any time

- The content of files is stored on the disks of the nodes
  - Partitioning: Files are partitioned into blocks that can be stored in different *Datanodes*
  - Replication: Each block is replicated in multiple *Datanodes*
    - Default replication degree: 3
  - A *Namenode* regulates access to files by clients
    - Master-worker architecture

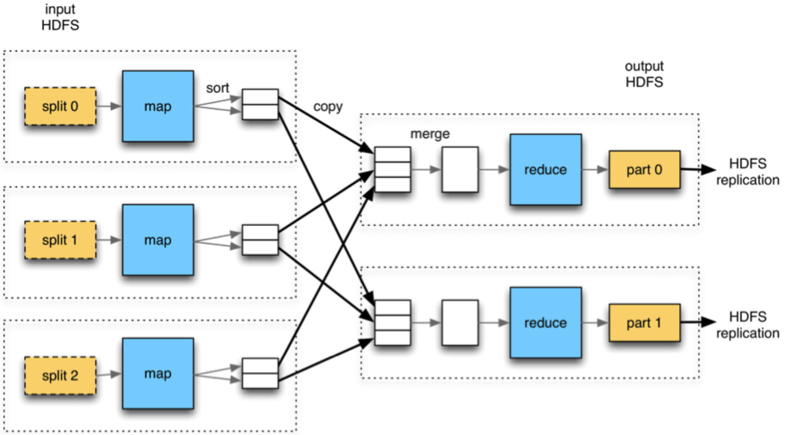
HDFS architecture

Figure from `https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html`



Hadoop data workflow

Figure from
https://www.supinfo.com/articles/single/2807-introduction-to-the-mapreduce-life-cycle



Hadoop workflow: a few comments

Data movements

- Map tasks are executing on nodes where the data blocks are hosted
  - Or on close nodes
  - Less expensive to move computation than to move data
- Load balancing between the reducers
  - Output of mappers are partitioned according to the number of reducers (modulo on a hash of the key)

Hadoop workflow: a few comments

I/O operations

- Map tasks read data from disks
- Output of the mappers are stored in memory if possible
    - Otherwise flushed to disk
- The result of reduce tasks in written into HDFS

Fault tolerance

- Execution of tasks is monitored by the master node
    - Tasks are launched again on other nodes if crashed or too slow

Agenda

Computing at large scale

Programming distributed systems

MapReduce

Introduction to Apache Spark

Spark internals

Programming with PySpark

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.

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

Main novelties

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

Programming with Spark

Spark Core API

- Scala
- Python
- Java

Integration with Hadoop

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

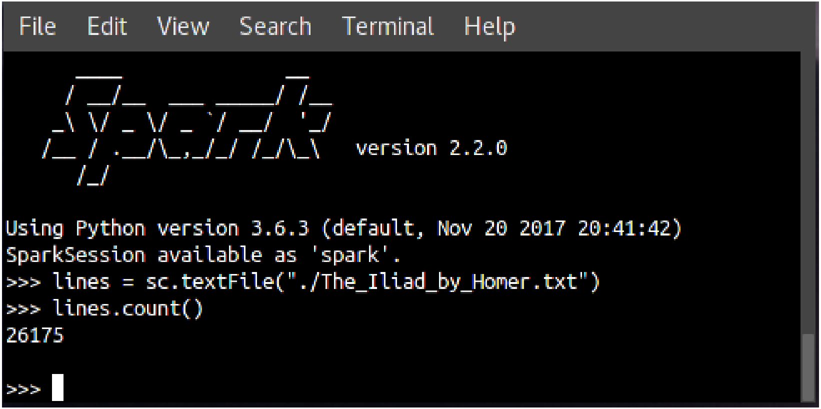
- Spark runs in a JVM
  - Spark is coded in Scala
- Read data from your local file system

Use interactive shell

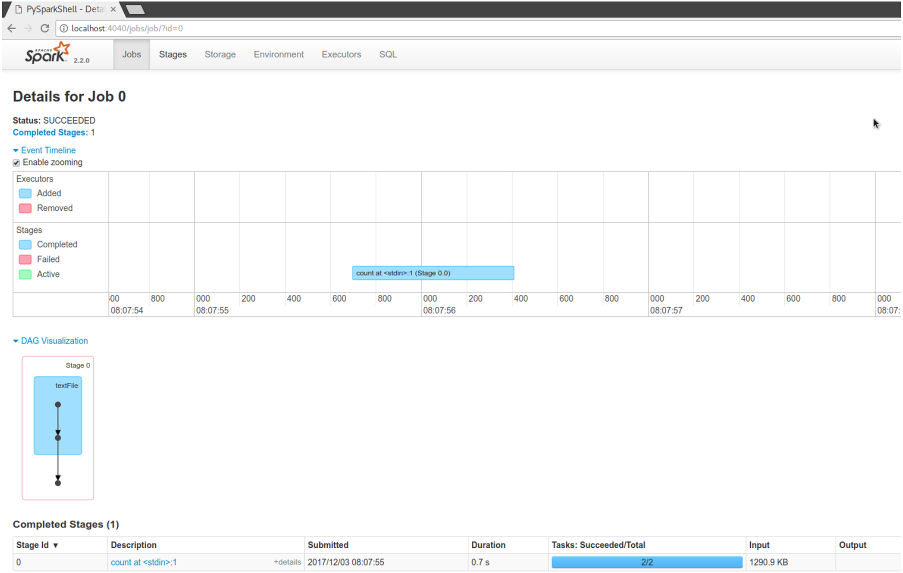
- Scala (*spark-shell*)
- Python (*pyspark*)
- Run locally or distributed at scale

A very first example with pyspark

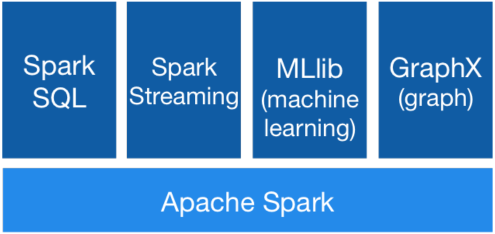
Counting lines



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

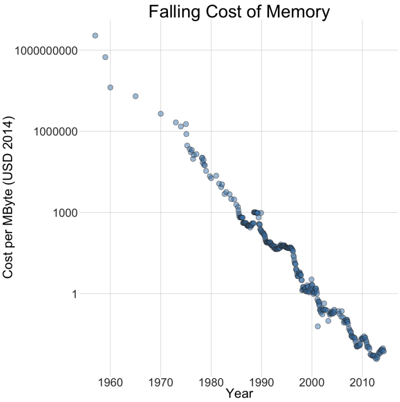
Memory is way faster than disks

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

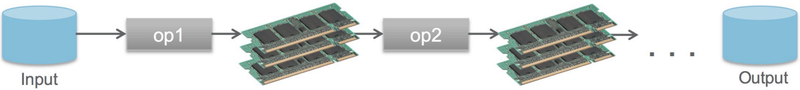


Cost of memory decreases = More memory per server

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 an RDD through deterministic operations (transformations) on either:
  - 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 a 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

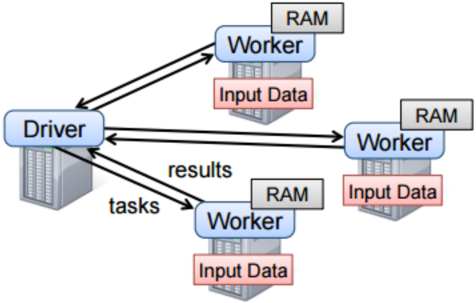
- RDD partition lost
  - Replay all transformations on the subset of input data or the most recent RDD available
- Deal with stragglers
  - Generate a new copy of a partition on another node

Spark runtime

Figure by M. Zaharia et al

- Driver
  - 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

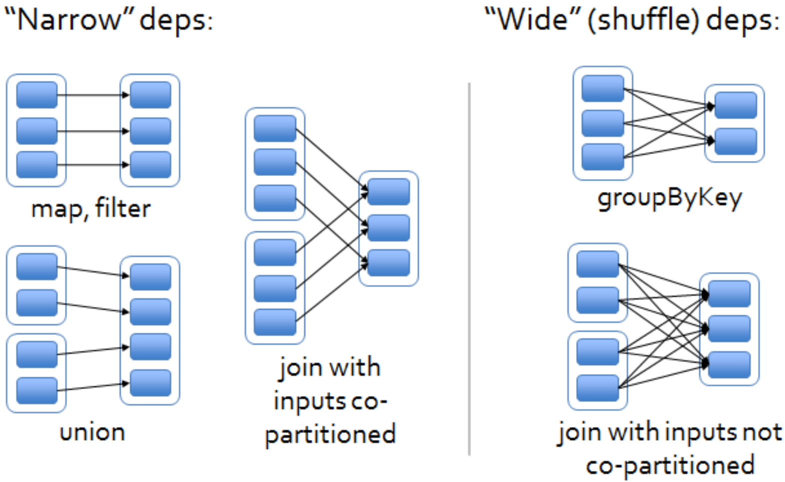
- Narrow dependencies
  - Each partition in the parent is used by at most one partition in the child
- Wide (shuffle) dependencies
  - Each partition in the parent is used by multiple partitions in the child

Impact of dependencies

- Scheduling: Which tasks can be run independently
- Fault tolerance: Which partitions are needed to recreate a lost partition
- Communication: Shuffling implies large amount of data exchanges

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

Persist an RDD

- By default, an 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

Agenda

Computing at large scale

Programming distributed systems

MapReduce

Introduction to Apache Spark

Spark internals

Programming with PySpark

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, 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.

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.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

```
lines = sc.textFile("data.txt")
words = lines.flatMap(lambda s: s.split('␣'))
pairs = words.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])]
```

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()
```

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 slides

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