Massive Graph Management & Analytics

Parallel Computing on Distributed Data

MapReduce with Spark

Nacéra Seghouani

Computer Science Department, CentraleSupélec
Laboratoire Interdisciplinaire des Sciences du Numérique, LISN
nacera.seghouani@centralesupelec.fr

2024-2025

MapReduce Computation Model

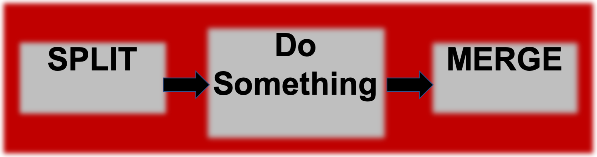
Motivation

- Design of parallel algorithms
- Big data processing pipelines
- Trade the communication cost against the degree of parallelism
- Processing pipeline based on MapReduce paradigm in a distributed environment.([1])
  - Google's internal implementation and Hadoop (Apache Foundation) to manage large-scale computations, to be tolerant of hardware faults.
  - HDFS, Hadoop Distributed File System, splits files into large blocks and distributes them across nodes in a cluster.
  - MapReduce programming model to manage many large-scale parallel computations

[1] https://static.googleusercontent.com/media/research.google.com/en//archive/mapreduce-osdi04.pdf

MapReduce Computation Model

☞ A style of computing

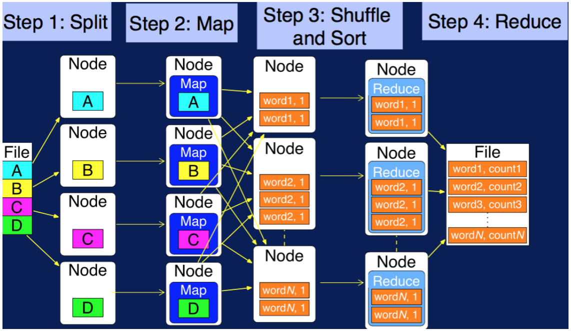


☞ All you need: define *Map* and *Reduce* functions, while the system

- ➜ manages the parallel execution on distributed data and coordination between them,
- ➜ deals with the possibility that one of these task execution fails.

Example: Word Counter

- ☞ A, B, C, D are file partitions distributed across different nodes (machines).
- ☞ *Map* task turns each partition into a sequence of pairs $(word, 1)$.
- ☞ Shuffle/sort task collects and groups the pairs by key/word $(word, [1, 1, \ldots])$ in order to guarantee that the same key will be processed by the same reduce task.
  Shuffling is a process of redistributing data from *Map* nodes to *Reduce* nodes.
- ☞ *Reduce* task takes as input $(word, [1, 1, \ldots])$ and produces pairs by key $(word, countWord)$.



Example: Word Counter

- *Map* task will typically process many words in one or more **chunks** (of about 64MB by default).
  - ➜ if a word w appears m times among all chunks assigned to that process, there will be m key-value pairs $(word, 1)$ among its output.
- To perform the grouping and distribution to the Reduce task, the master controller:
  - ➜ merges the pairs by key/word and produces a sequence of $(word, [1, 1, ..., 1])$.
  - ➜ knows how many reduce tasks there will be, say r, produces from 1 to r lists, puts a list in one of r local files destined for one of the Reduce tasks.
- Each key/word k is assigned as input to one and only one *Reduce* task.
- *Reduce* task executes one or more reducers (one by key). The outputs from all reducers are merged into a single file.

MapReduce : Map Function

More Formally

- What is map function?
  - $map_f([e_1, e_2, ..., e_n]) = [f(e_1), f(e_1), ..., f(e_n)]$
  - $map_{(*2)}([2, 3, 6]) = [4, 6, 12]$
- In MapReduce, *map* is a specific prototype of f function
  - $map_{f_m}([e_1, e_2, ..., e_n]) = [f_m(e_1), ...f_m(e_n)]$ where the function $f_m((e_i)) = (k_i, v_i)$ applied to each input element produces a pair key/value.
  - In word counter example:

$$map_{f_m}([\text{"}silent\text{"}, \text{"}sentinels\text{"}, \text{"}men\text{"}, \text{"}whale\text{"}, \text{"}mortal\text{"}, \text{"}men\text{"}, \text{"}ocean\text{"}, \text{"}whale\text{"}, \text{"}seas\text{"}, \text{"}ocean\text{"}, \text{"}whale\text{"}])$$

$$= [(\text{"}silent\text{"}, 1), (\text{"}sentinels\text{"}, 1), (\text{"}men\text{"}, 1), (\text{"}whale, 1), (\text{"}mortal\text{"}, 1), (\text{"}men\text{"}, 1), (\text{"}ocean\text{"}, 1), (\text{"}whale, 1), (\text{"}seas\text{"}, 1), (\text{"}ocean\text{"}, 1), (\text{"}whale\text{"}, 1)]$$

CentraleSupélec

MapReduce Pipeline: Shuffling/Grouping Function

More Formally

- $shuffle([\ldots,(k,v_1),\ldots,(k,v_n)]) = [\ldots,(k,[v_1,\ldots,,v_n]),\ldots]$
- Groups the values of the same key to produce a pair for each key
- In word counter example:

$$shuffle([(\text{"}silent\text{"},1),(\text{"}sentinels\text{"},1),(\text{"}men\text{"},1),(\text{"}whale,1),(\text{"}mortal\text{"},1),(\text{"}men\text{"},1),(\text{"}ocean\text{"},1),(\text{"}whale,1),(\text{"}seas\text{"},1),(\text{"}ocean\text{"},1),(\text{"}whale\text{"},1)]) =$$

$$[(\text{"}silent\text{"},[1]),(\text{"}sentinels\text{"},[1]),(\text{"}men\text{"},[1,1]),(\text{"}whale,[1,1,1]),(\text{"}mortal\text{"},[1]),(\text{"}ocean\text{"},[1,1]),(\text{"}seas\text{"},[1])]$$

CentraleSupélec

MapReduce Pipeline: Reduce Function

More Formally

- What is reduce function?
  - $reduce_f[v_1, v_2, ..., v_n] = f([v_1, v_2, ..., v_n])$
  - $reduce_{(sum)}([2, 3, 6]) = 11$
  - Example of f functions: sum, size, flatten, min, max, ...
- In MapReduce, reduce is a specific prototype of f function
  - $reduce_{f_r}((k, [v_1, ...])) = [.., (k\prime, f_r([v_1, ...])), ...]$
  - In word counter example:

$$reduce_{f_r}[(''silent'', [1]), (''sentinels'', [1]), (''men'', [1, 1]), (''whale, [1, 1, 1]), (''mortal'', [1]), (''ocean'', [1, 1]), (''seas'', [1])]$$

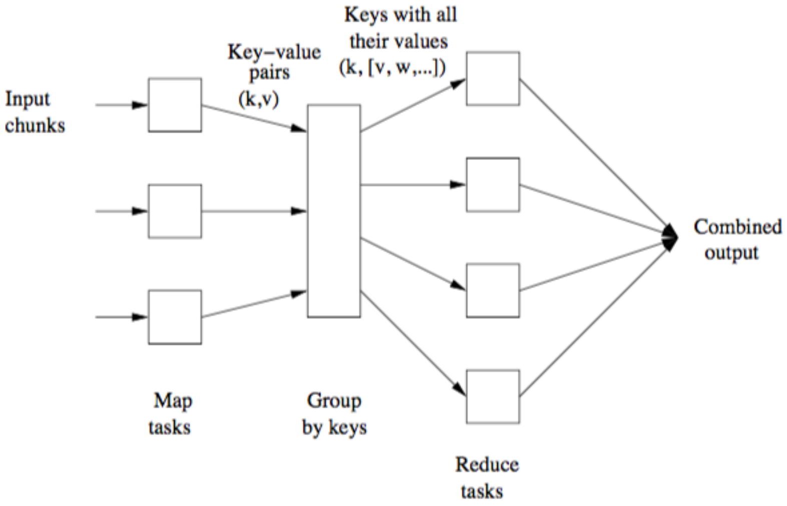
$$[(''silent'', 1), (''sentinels'', 1), (''men'', 1), (''whale, 3), (''mortal'', 1), (''ocean'', 2), (''seas'', 1)]$$

- MapReduce pipeline could be a composition $reduce_{f_r} \circ shuffle \circ map_{f_m}([...])$
  - $f_r(x) \circ f_r(y) = f_r(y) \circ f_r(x)$ Commutative function
  - $f_r(x) \circ (f_r(y) \circ f_r(z)) = (f_r(x) \circ f_r(y)) \circ f_r(z)$ Associative function

Overview of MapReduce Computation Paradigm

- ☞ *Map* task is given one or more chunks from HDFS, turns the chunk into a sequence of key-value pairs (k, v) determined by the Map function map_{f_m}.
- ☞ The key-value pairs (k, v) from each Map task are collected by a master controller and sorted by key.
- ☞ The key-value pairs (k, v) are then assigned to the Reduce tasks, all (k, v) with the same k are assigned to the same Reduce task.
- ☞ *Reduce* task works on one key k at a time, and combine all the values associated $(k, list(v))$ using the Reduce function $reduce_{f_r}$.
- ☞ Inputs to reduce tasks and outputs from map tasks of the key-value pair form allow the composition of MapReduce processes.

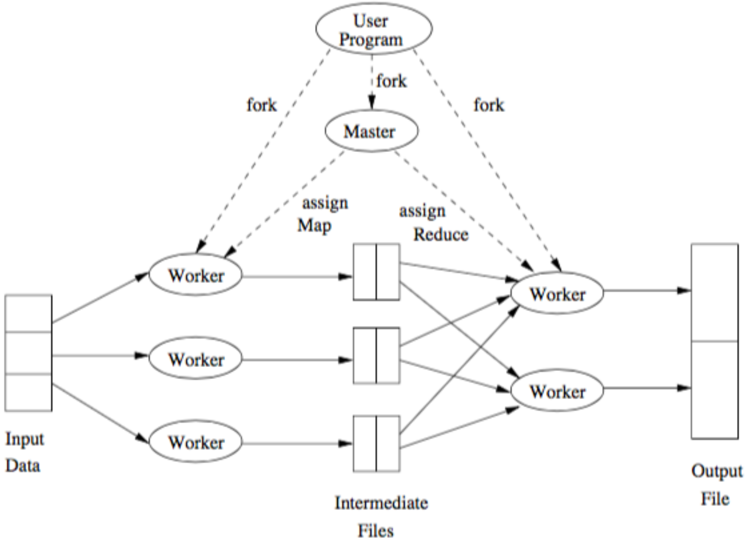
Overview of a MapReduce Computation Paradigm



MapReduce Execution Model

- ☞ The user program forks a master controller process and some number of worker processes at different compute nodes.
- ☞ The master creates some number of map tasks and some number of reduce tasks. It assigns the tasks to worker processes by taking into account the co-location.
- ☞ A worker handles either map tasks (a map worker) or reduce tasks (a reduce worker), but not both.
- ☞ A worker process reports to the master when it finishes a task, and a new task is scheduled by the master for that worker process.
- ☞ The master keeps track of the status of each map and reduce task (idle, executing at a particular worker, or completed).

MapReduce Execution Model



MapReduce Execution Model: Coping With Node Failures

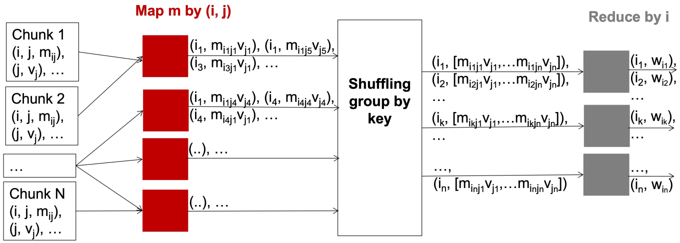
- ☞ If the master node fails the entire MapReduce job must be restarted.
- ☞ Work node failure is detected and managed by the master, because it periodically pings the worker processes.
- ☞ All the map tasks assigned to this worker have to be redone.

Algorithms by MapReduce

- MapReduce is not a solution to every problem
- It makes sense only when files are very large and are rarely updated.
- The original purpose of Google MapReduce implementation is to execute very large matrix-vector multiplications.

Matrix-Vector Multiplication by MapReduce

- Let $M[m_{ij}]$ be a squared matrix and V a vector of size n, the product $W = MV$ is defined : $w_i = \sum_j^n m_{ij} v_j$
- M and V are stored in a file of the DFS as triples (i, j, m_{ij}) and pairs (j, v_j)
- Map function, applied to each $((i, j), m_{ij})$ and (j, v_j), produces a key-value pair $(i, m_{ij} v_j)$
- Reduce function simply sums all the values associated with a given key i, produces a pair (i, w_i)
- All V's pairs should be available in each chunk.



Map m by (i, j)
Reduce by i
Chunk 1 (i, j, m_ij), (j, v_j), …
Chunk 2 (i, j, m_ij), (j, v_j), …
…
Chunk N (i, j, m_ij), (j, v_j), …
(i1, m_i1j1 v_j1), (i1, m_i1j5 v_j5), (i3, m_i3j1 v_j1), …
(i1, m_i1j4 v_j4), (i4, m_i4j4 v_j4), (i4, m_i4j1 v_j1), …
(..), …
(..), …
Shuffling group by key
(i1, [m_i1j1 v_j1, … m_i1jn v_jn]),
(i2, [m_i2j1 v_j1, … m_i2jn v_jn]),
…
(ik, [m_ikj1 v_j1, … m_ikjn v_jn]),
…
…,
(in, [m_inj1 v_j1, … m_injn v_jn])
(i1, w_i1),
(i2, w_i2),
…
(ik, w_ik),
…
…,
(in, w_in)

Matrix-Vector Multiplication by MapReduce

- ☞ Let $M[m_{ij}]$ be a squared matrix and V a vector of size n, the product $W = MV$ is defined : $w_i = \sum_j^n m_{ij} v_j$
- ☞ M and V are stored in a file of the DFS as triples (i, j, m_{ij}) and pairs (j, v_j)
- ☞ Map function, applied to each $((i, j), m_{ij})$ and (j, v_j), produces a key-value pair $(i, m_{ij} v_j)$
- ☞ Reduce function simply sums all the values associated with a given key i, produces a pair (i, w_i)
- ☞ All V's pairs should be available in each chunk.

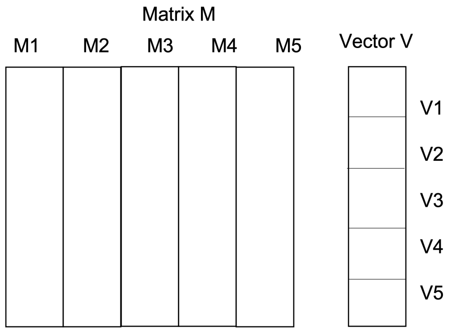
```
$map_{f_m}(key, val)$ /*$(key, val) = ((i, j), m_{ij})$*/
  $foreach(v_j \in V)$
    $if\ j = key(2)$
      $emit(key(1), val \times v_j)$ /*emit $(i, m_{ij} v_j)$*/
```

```
$reduce_{f_r}(key, val)$ /*$(key, val) = (i, [m_{i1} v_1, \ldots m_{in} v_n)]$*/
  $sum \leftarrow 0$
  $foreach(v \in val)$
    $sum \leftarrow sum + v$
  $emit(key, sum)$/*emit $(i, w_i)$*/
```

Matrix-Vector Multiplication by MapReduce

☞ The matrix M and the vector V each will be stored in a file of the DFS. If n is too large V cannot not fit in main memory of a work node, a large number of disk accesses are required.

- ➜ Divide the matrix into vertical stripes of equal width and divide the vector into an equal number of horizontal stripes, of the same height.
- ➜ Each Map task is assigned a chunk from one of the the matrix stripes and gets the entire corresponding stripe of the vector.
- ➜ In this example MV is equivalent to compute vectors $M_1 V_1$, $M_2 V_2$, $M_3 V_3$, $M_4 V_4$ and $M_5 V_5$



Matrix Multiplication by MapReduce

- Let M and N be matrixes of size $l \times r$ and $r \times t$ resp., the product $P = MN$ is a matrix of size $l \times r$, where $p_{ik} = \sum_{j=1}^{r} m_{ij} n_{jk}$
- Fom (M, i, j, m_{ij}) and (N, j, k, n_{jk}) the Map task produces $(j, (M, i, m_{ij}))$ and $(j, (N, k, n_{jk}))$
- The Reduce Function produces for each key j the key-value-pair $((i, k), m_{ij} n_{jk})$.
- Grouping and aggregation achieved by another MapReduce operation.
- The Map Function: just the identity.
- The Reduce Function: For each key (i, k), produce the sum of the list of values associated with this key $p_{ik} = \sum_j m_{ij} n_{jk}$
- M and N could be divided into n vertical and horizontal stripes of resp. (l, r_i) and (r_i, t) sizes, with $\sum_i^n r_i = r$

CentraleSupélec

Matrix Multiplication by MapReduce

☞ t is (M, i, j, m_{ij}) or (N, j, k, n_{jk})

```
$map_{f_{m_1}}(t)$
    $emit(key = t(3), value = (t(1), t(2), t(4)))$
    $t(3) = j$ and $(t(1), t(2), t(4)) = (M, i, m_{ij})$ or $(N, k, n_{jk})$

$map_{f_{m_2}}(key, val)$
    $emit(key, val)$
```

```
$reduce_{f_{r_1}}(key, val)$
  $foreach(v \in val(1))$
    $foreach(w \in val(1))$
      $if\ v(1) = M$ and $w(1) = N$
        $emit((v(2), w(2)), v(3)w(3))$
        $(v(2), w(2)) = (i, k)$ and $v(3)w(3) = m_{ij}n_{jk}$

$reduce_{f_{r_2}}(key, val)$
  $sum \leftarrow 0$
  $foreach(v \in val)$
    $sum \leftarrow sum + v$
    $emit(key, sum)$
    $key = (i, k)$ and $value = p_{jk}$
```

CentraleSupélec

Relational-Algebra Selection and Projection Operations by MapReduce

- Let $R(A_1, A_2, ...A_n)$ be a relation stored as a file in a DFS. The elements of this file are the tuples of R.
- Selection $\sigma_C(R)$
  - ➜ Map Function: For each tuple t in R, test if it satisfies C. If so, produce the key-value pair (t, t). That is, both the key and value are t.
  - ➜ Reduce Function: It simply passes each key-value pair to the output.
- Projection $\pi_A(R)$
  - ➜ Map Function: For each tuple t in R, construct a tuple $t\prime$ by eliminating from t attributes $\notin A$. Output the key-value pair $(t\prime, t\prime)$.
  - ➜ Reduce Function: For each key $t\prime$ produced by any of the Map tasks, there will be one or more key-value pairs $(t\prime, t\prime)$. The Reduce function turns $(t\prime, [t\prime, ...t\prime])$ into $(t\prime, t\prime)$, so it produces exactly one pair $(t\prime, t\prime)$ for this key $t\prime$.

Relational-Algebra Natural Join Operation by MapReduce

- ☞ $R(A) \bowtie_B S(C)$, with A, B, C sets of attributes, $B \subseteq A$ and $B \subseteq C$
  - ➜ The Map Function: For each tuple (a, b) of R, produce the key-value pair $(b, (R, a))$. For each tuple (b, c) of S, produce the key-value pair $(b, (S, c))$.

    a, b c are resp. the values of the attribues A, B and C
  - ➜ The Reduce Function: Each key value b will be associated with a list of pairs $(b, [(R, a), (S, c)])$.

Grouping and Aggregation Operations by MapReduce

- ☞ $\gamma_{A,\theta(B)}(R)$, where $A \cup B$ is the set of attributes of R, A is the set of grouping attributes with $A \cap B = \phi$.
  - ➜ The Map Function: For each tuple produce the key-value pair (a, b).
  - ➜ The Reduce Function: Each key a represents a group. Apply the aggregation operator θ to the list $(a, [b1, b2, \ldots, bn]$. The output is the pair (a,x), where x is the result of applying θ to the list.

Some issues

- ✎ **Locality**: Input data (managed by GFS) is stored on local disks of machines of the cluster. GFS divides each file into 64 MB blocks, stores several replicas of each block (typically 3 copies) on different machines.
  MapReduce master takes the location information of the input files into account, attempts to schedule a map task on a machine that contains the needed a replica. Failing that, it attempts to schedule a map task near a replica of that task's input data.
- ✎ **Granularity**: the map and reduce steps divided into M and R pieces. M and R should be much larger than the number of workers. Each worker performs different tasks: improves dynamic load balancing, speeds up recovery when a worker fails.
  Practical bounds on how large M and R can be: the master must make $M + R$ scheduling decisions and keeps $M \times R$ state in memory.
- ✎ **Refinements**: Partitioning input data using different functions according to the problem to be solved.
- ✎ **Ordering Guarantees**: The intermediate key/value pairs are generally processed in increasing key order to make it easy to generate a sorted output file per partition. But it not a guarantee.

SPARK Framework

Outline

☞ Motivation

☞ Spark's Parallel Computing Engine Overview ([2])

- ✘ Spark's Engine Properties & Components
- ✘ Spark's Execution Architecture

☞ *Resilient distributed datasets* - RDD - ([3])

- ✘ Spark Context
- ✘ RDD Transformations & Actions
- ✘ Examples in Python

2 `https://spark.apache.org/research.html`

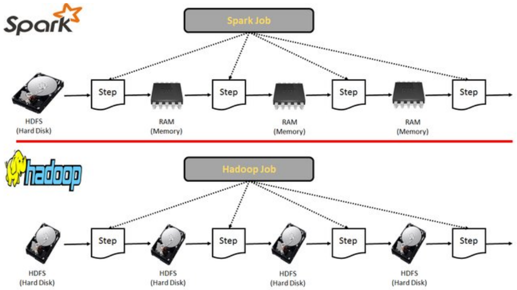
3 `http://people.csail.mit.edu/matei/papers/2012/nsdi_spark.pdf`

Motivation

- ☞ Data growing faster than processing speeds
- ☞ Only solution is to parallelize on large clusters
  - ✓ Wide use in both enterprises and web industry.
- ☞ Traditional Network Programming
  - ✓ Message-passing between nodes (e.g. MPI) $\rightarrow$ Difficult to scale.
  - ✓ How to split problem across nodes? $\rightarrow$ network & data locality must be considered.
  - ✓ How to deal with failures? with stragglers (node not failed, but slow)?
  - ✓ Ethernet networking not fast
  - ✓ Have to write programs for each machine
- ☞ Disk vs Memory. Network vs Local

Spark’s Engine Properties & Components

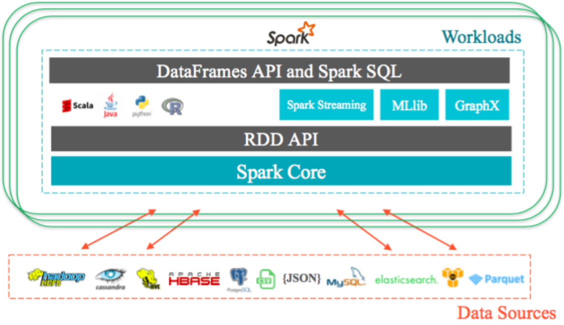
- Apache Spark is an open-source for large-scale data processing (Most active community in big data, with 50+ companies contributing).
- It provides an interface for programming clusters with implicit data parallelism and fault tolerance.
- It extends a programming language with a read-only data distributed over a cluster of machines *Resilient distributed datasets* (RDD), maintained in a fault-tolerant way.
  - ✓ RDDs were developed in 2012 in response to limitations in Hadoop’s MapReduce, which forces a particular linear dataflow as a sequence of HDFS read/write.
  - ✓ Apache Hadoop’s MapReduce and HDFS were inspired by Google research on MapReduce and Google File System (GFS).[4]
- Spark is fast (up to 100x faster than traditional Hadoop) due to in-memory data processing.



[4] https://research.google/pubs/pub62/

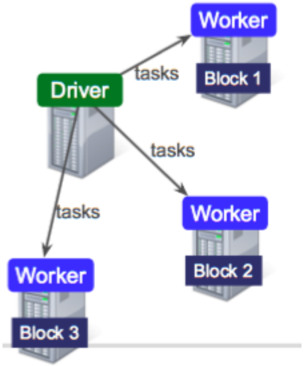
Spark's Engine Properties & Components

- ☞ Originally written in Scala, a high level language for JVM. Clean APIs in Java, Scala, Python, R
- ☞ Dataframe API was released as an abstraction on top of the RDD
- ☞ Learning and graph analytics through supplementary packages like MLlib and GraphX.
  - ✓ facilitates the implementation of both iterative algorithms, and interactive/exploratory data analysis, i.e., the repeated database-style querying of data.
- ☞ Spark requires a cluster manager and a distributed storage system.
  - ✓ cluster management: Spark supports standalone (native Spark cluster, where you can launch a cluster either manually or use scripts provided by the install package), Hadoop YARN, Apache Mesos or Kubernetes.
  - ✓ distributed storage, Spark can interface with a wide variety, including Alluxio, Hadoop Distributed File System (HDFS), MapR File System (MapR-FS), Cassandra, OpenStack Swift, Amazon S3, Kudu, Lustre file system.
- ☞ Spark data sources: different formats and databases



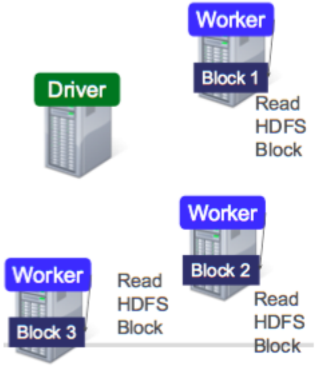
Spark's Execution Architecture

- ✓ Data is split into partitions (blocks)
- ✓ Driver assigns tasks to each worker



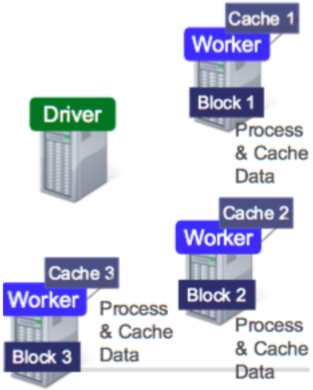
Spark's Execution Architecture

✓ Each worker read an HDFS Block



Spark's Execution Architecture

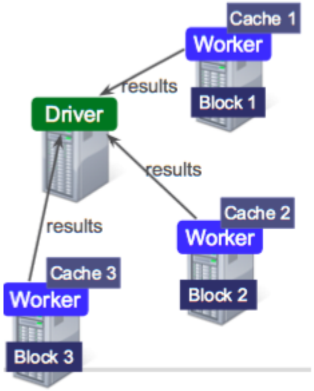
✓ Each worker has a cache, process & cache data (if necessary)



CentraleSupélec

Spark's Execution Architecture

✓ Results to Driver



Resilient Distributed Datasets (RDDs) Overview

- ☞ Immutable and distributed collection of objects spread across a cluster, stored in RAM or disk (persistent)
- ☞ Statically typed: RDD[T] has objects of type T. RDDs can contain any type of Python, Java, or Scala objects, including user-defined classes.
- ☞ Controlled partitioning & storage (memory, disk, ...).
- ☞ Built via parallel **transformations** and computed via parallel **actions** on distributed datasets
  - ➜ RDDs split into multiple partitions, which may be computed on different nodes of a cluster
  - ➜ RDDs built and manipulated through a parallel **lazy transformations** (map, filter, . . .) and **actions** (count, collect, . . .)
  - ➜ Creating new RDDs, transforming existing RDDs, or executing actions on RDDs to compute results
  - ➜ Inside Apache Spark the workflow is managed as a directed acyclic graph (DAG). Nodes represent RDDs while edges represent the operations on the RDDs.

CentraleSupélec

Programming with RDDs: Example in Python

Spark Context Creation

```
sc = pyspark.SparkContext(...)
```

Creating an RDD of strings with textFile()

```
lines = sc.textFile("README.txt")
```

Transformations: construct a new RDD from a previous one, one common transformation is filtering data that matches a predicate.

```
pythonLines = lines.filter(lambda line: "Python" in line)
```

Actions: compute a result based on an RDD, and either return it to the driver program or save it to an external storage system (e.g., HDFS).

```
pythonLines.first()
```

Programming with RDDs: Example in Python

- Transformations are operations on RDDs that return a new RDD.
- "Lazy Evaluation": transformed RDDs are computed lazily, only when you use them in an action.
- `filter()` operation does not mutate the existing input RDDs. Instead, it returns a pointer to an entirely new RDD.
- Many transformations work on one element at a time; but this is not true for all transformations (for ex. `union()`).

```
inputRDD = sc.textFile("log.txt")
errorsRDD = inputRDD.filter(lambda x: "error" in x)
warningsRDD = inputRDD.filter(lambda x: "warning" in x)
badLinesRDD = errorsRDD.union(warningsRDD)
badLinesRDD.first()
```

Programming with RDDs: Example in Python

- ✎ Actions are the operations that return a final value to the driver program or write data to an external storage system.
- ✎ Actions force the evaluation of the transformations required for the RDD they were called on in order to actually produce output.
- ✎ Each time a new action is called, the entire RDD must be computed “from scratch”. Otherwise, users can persist intermediate results (caching).
- ✎ In Spark, users are free to organize their program into smaller, more manageable operations.

```
print "Input had " + badLinesRDD.count() + " concerning lines"
print "Here are 10 examples:"
for line in badLinesRDD.take(10):
 print line
```

CentraleSupélec

Programming with RDDs

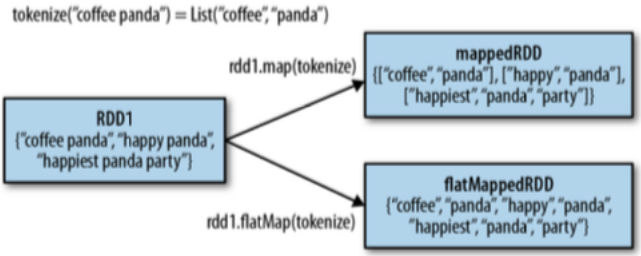
- ✎ Once a SparkContext `sc` set, it is used to build RDDs
- ✎ Driver program manages a number of workers. Different workers (on different machines/nodes) might count lines in different ranges of the file.
- ✎ Spark keeps track of the set of dependencies between different RDDs, called the *lineage graph*.

Programming with RDDs: map() and flatMap()

- ✎ `map()` is used for different purposes, from fetching a large collection to just squaring numbers. It transforms an RDD of length N into another RDD of length N.

```
nums = sc.parallelize([1, 2, 3, 4])
squared = nums.map(lambda x: x * x).collect()
for num in squared: print "%i " % (num)
```

- ✎ `flatMap()` transforms an RDD of length N into a collection of N collections, then flattens these into a single RDD of results.


tokenize("coffee panda") = List("coffee", "panda")
rdd1.map(tokenize)
mappedRDD
{["coffee", "panda"], ["happy", "panda"], ["happiest", "panda", "party"]}
RDD1
{"coffee panda", "happy panda", "happiest panda party"}
rdd1.flatMap(tokenize)
flatMappedRDD
{"coffee", "panda", "happy", "panda", "happiest", "panda", "party"}

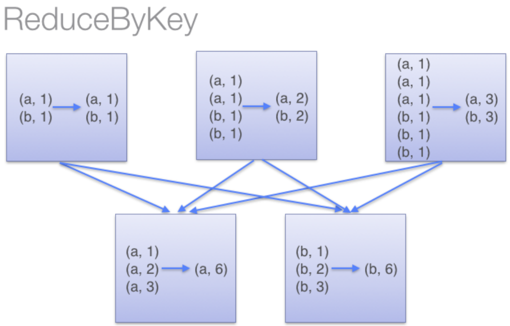
Programming with RDDs: reduce() and reducedByKey()

- ✎ `reduce()` is the most common action on basic RDDs which operates on two elements of the type in your RDD and returns a new element of the same type.

```
data = sc.parallelize([1, 2, 3])
data.reduce(lambda x, y: x + y)  => 6
```

- ✎ `reducedByKey()` operates on RDD of key-value pairs. It runs several parallel reduce operations, for each key, where each operation combines values that have the same key. It returns a new RDD consisting of each key and the reduced value for that key, therefore it is a transformation.

```
pets = sc.parallelize([("cat", 3), ("dog", 2), ("cat", 1)])
pets.reduceByKey(lambda x, y: x + y) => {("cat", 4), ("dog", 2)}
```


ReduceByKey
(a, 1) (b, 1) → (a, 1) (b, 1)
(a, 1) (a, 1) (b, 1) (b, 1) → (a, 2) (b, 2)
(a, 1) (a, 1) (a, 1) (b, 1) (b, 1) (b, 1) → (a, 3) (b, 3)
(a, 1) (a, 2) (a, 3) → (a, 6)
(b, 1) (b, 2) (b, 3) → (b, 6)

CentraleSupélec

Programming with RDDs: groupByKey() and sortByKey()

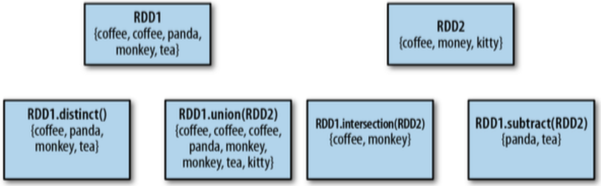
✎ Other useful transformation operations like `groupByKey()` and `sortByKey()` that operate on RDD key-value pairs

```
pets = sc.parallelize([("cat", 3), ("dog", 2), ("cat", 1)])
pets.groupByKey() => {("cat", [3, 1]), ("dog", [2])}
pets.sortByKey() => {("cat", 3),  ("cat", 1),  ("dog", 2)}
```

CentraleSupélec

Programming with RDDs: More transformations

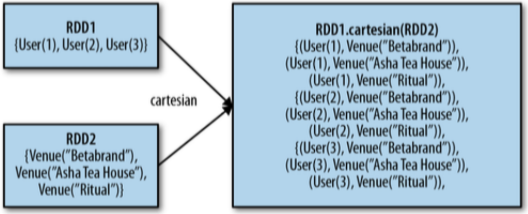
- `distinct()` transformation produces a new RDD with only distinct items.
- `union(other)` gives back an RDD consisting of the data from both sources. Duplicates are not removed.
- `intersection(other)` returns only elements in both RDDs without duplicates.
- `subtract(other)` takes in another RDD and returns an RDD that has only values present in the first RDD and not the second RDD.



CentraleSupélec

Programming with RDDs: More transformations

✎ `cartesian(other)` is a transformation which returns all possible pairs of (a, b) where a is in the source RDD and b is in the other RDD. Note that the Cartesian product is very expensive for large RDDs.


RDD1
{User(1), User(2), User(3)}
RDD2
{Venue("Betabrand"), Venue("Asha Tea House"), Venue("Ritual")}
cartesian
RDD1.cartesian(RDD2)
{(User(1), Venue("Betabrand")),
(User(1), Venue("Asha Tea House")),
(User(1), Venue("Ritual")),
{(User(2), Venue("Betabrand")),
(User(2), Venue("Asha Tea House")),
(User(2), Venue("Ritual")),
{(User(3), Venue("Betabrand")),
(User(3), Venue("Asha Tea House")),
(User(3), Venue("Ritual")),

Programming with RDDs: More transformations

- ✎ Joining data together is one of the most common operations on a pair RDD, including right and left outer joins, cross joins, and inner joins.
- ✎ `innerJoin` only keys that are present in both pair RDDs are output.
- ✎ `leftOuterJoin(other)` and `rightOuterJoin(other)` both join pair RDDs together by key, where one of the pair RDDs can be missing the key.

```
rdd = [(1, 2), (3, 4), (3, 6)] and
other = [(3, 9)]

join is an inner join
rdd.join(other) -> {(3, (4, 9)), (3, (6, 9))}

rdd is left
rdd.leftOuterJoin(other) - > {(1,(2,None)), (3, (4,9)), (3, (6,9))}
```

CentraleSupélec

Programming with RDDs: More actions

- `collect()` is used to retrieve the entire RDD. Useful if it filters RDDs down to a very small size to deal with it locally at the driver. The retrieved dataset must fit in memory on a single machine.
- `take()` is used to retrieve a small number of elements in the RDD at the driver program and then iterate over them locally.
- `top()` is used to extract the top elements.
- `takeSample(withReplacement, num, seed)` allows to take a sample of our data either with or without (no element occurs more than one time) replacement.
- It is useful to perform an action on all of the elements in the RDD, but without returning any result to the driver program. `foreach()` action performs computations on each element in the RDD without bringing it back locally.
- `count()` returns a count of the elements, and `countByValue()` returns a map of each unique value to its count.

Persistence - Caching

- ✎ Spark RDDs are lazily evaluated, and sometimes we use the same RDD multiple times. Naively, Spark will recompute the RDD and all of its dependencies each time we call an action on the RDD.
- ✎ To avoid computing an RDD multiple times, we can ask Spark to persist data using `persist()`.
- ✎ Notice that `persist()` call on its own doesn't force evaluation.
- ✎ If you cache too much data, Spark will automatically evict old partitions. For the memory-only storage levels, it will recompute these partitions the next time they are accessed, Caching unnecessary data can lead to eviction and more re-computation time.

Go to Practice

☞ See the NoteBook on Edunao