Apache Spark

Adapté de Kishore Pusukuri

1

1

Hadoop

MapReduce

- Programmes permettant de traiter en parallèle de gros volumes de données sur des clusters de manière fiable et tolérante aux pannes
- S'occupe de la programmation des tâches, de leur suivi et de la ré-exécution des tâches qui ont échoué

HDFS & MapReduce

- Fonctionne sur le même ensemble de nœuds ; nœuds de calcul et nœuds de stockage identiques (en gardant les données proches du calcul) ; très haut débit

YARN & MapReduce

- Un seul gestionnaire de ressources maître, un gestionnaire de nœuds esclaves par nœud, et un AppMaster par application

2

2

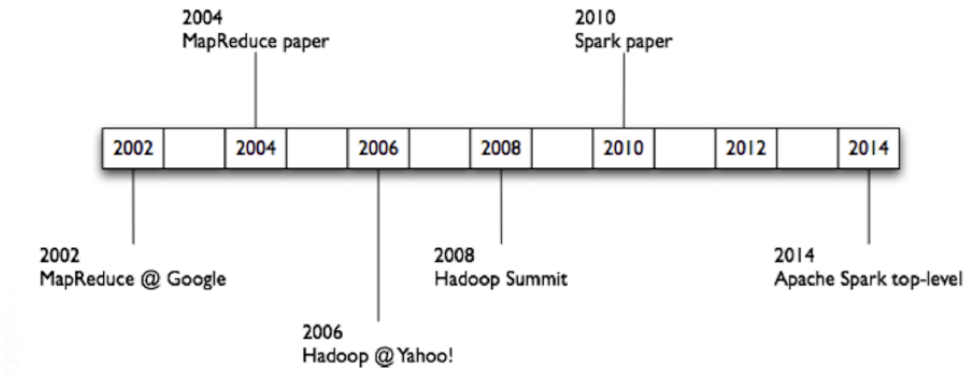
Plan

- Motivation
- Les bases de Spark
- Programmation Spark

3

3

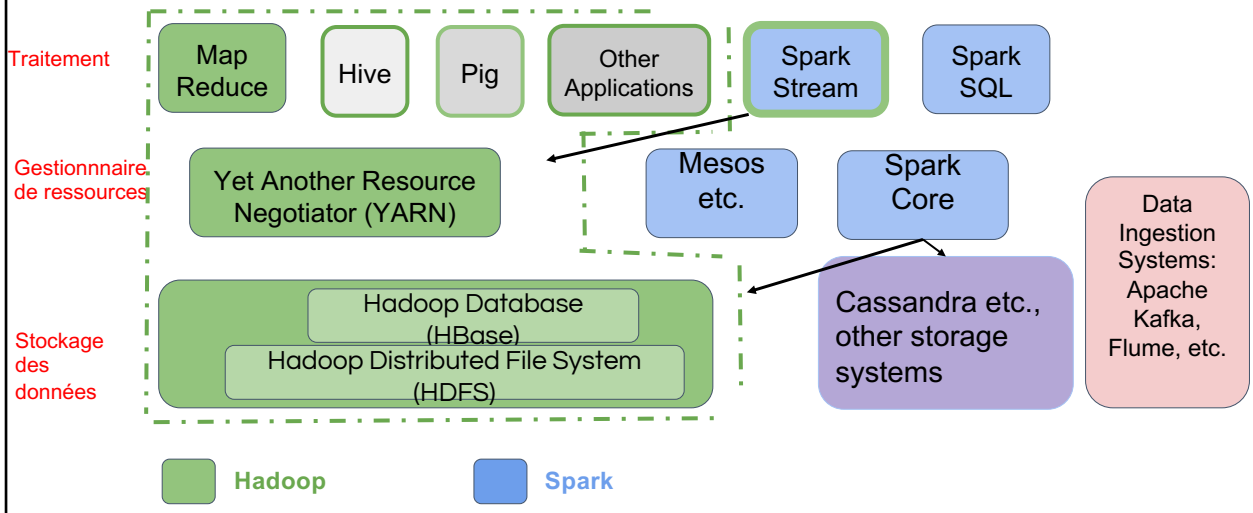
Histoire de Hadoop et de Spark



4

4

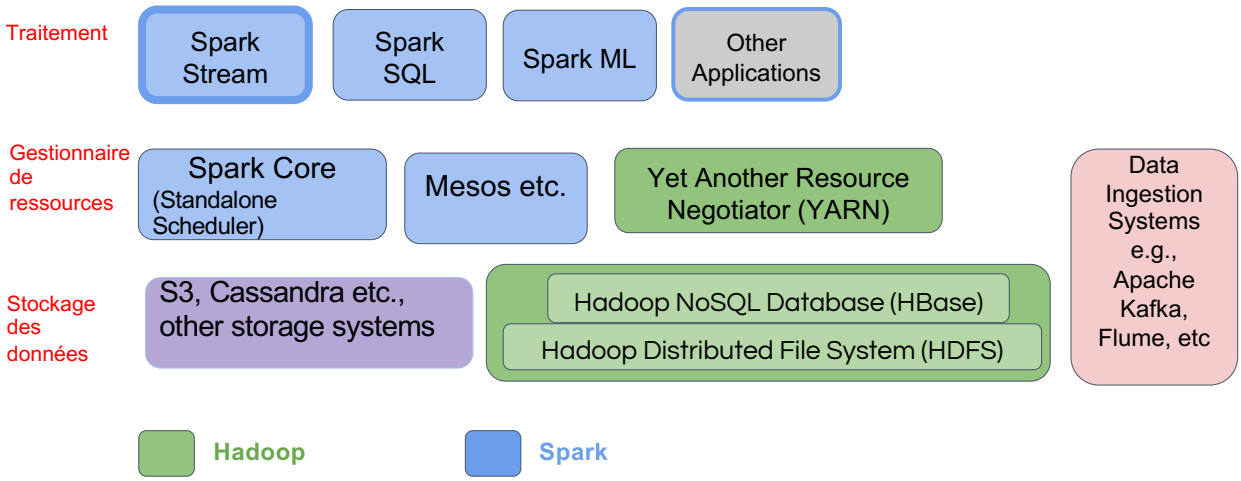
Apache Hadoop & Apache Spark



5

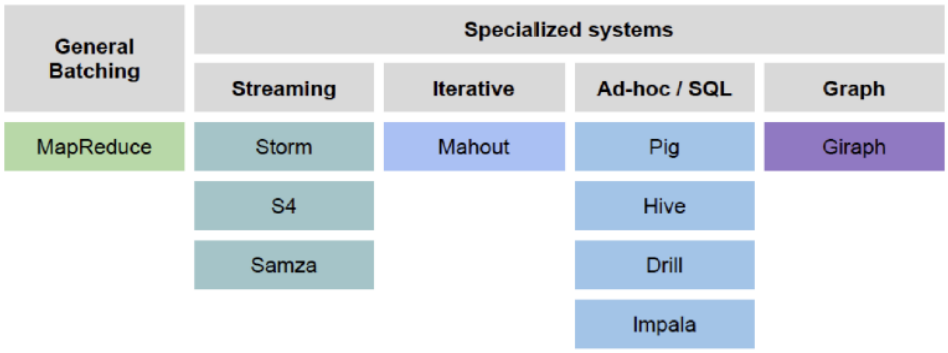
Apache Spark

** Spark peut se connecter à plusieurs types de *gestionnaires de clusters* (soit le propre gestionnaire de cluster autonome de Spark, Mesos ou YARN)



6

Hadoop : Pas de vision unifiée

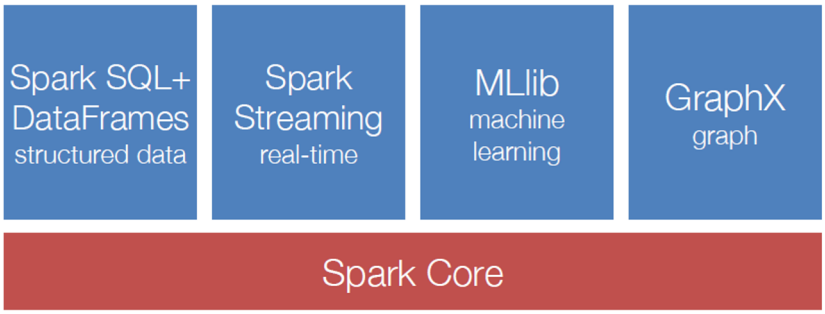


- Plusieurs modules “dispersés”
- Diversité des API
- Coûts opérationnels plus élevés

7

7

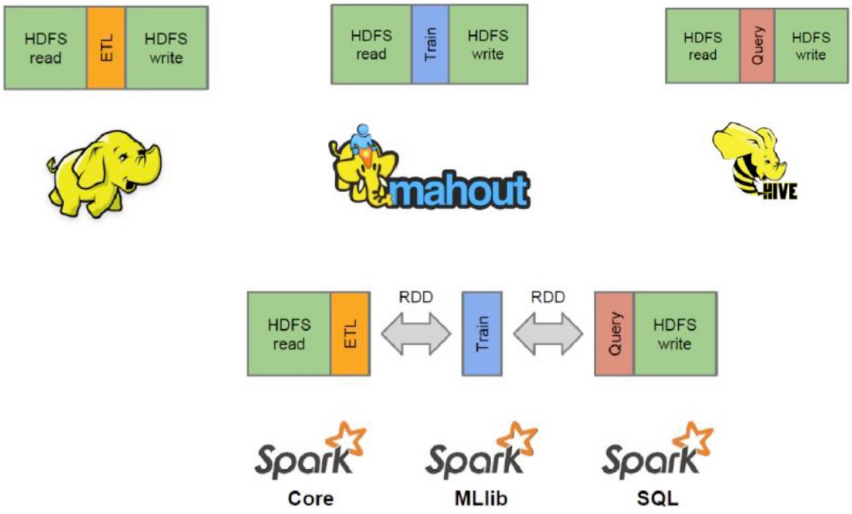
Ecosystème Spark : Un pipeline unifié



8

8

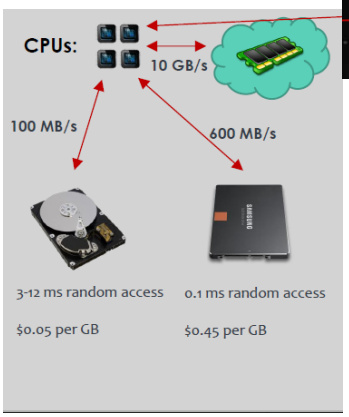
Spark vs MapReduce : Flux de données



9

9

Taux d'accès aux données

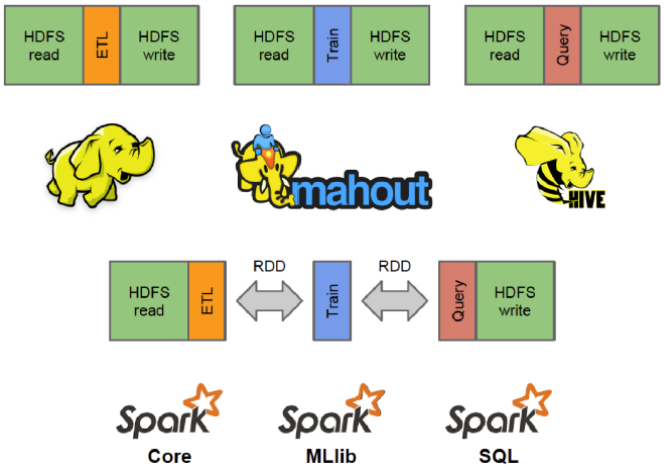


- Au niveau d'un nœud :
  - CPU à mémoire : 10 Go/sec
  - De l'unité centrale au disque dur : 0,1 Go/sec
  - CPU à SSD : 0,6 Go/sec
- Nœuds entre réseaux : 0,125 GB/sec à 1 GB/sec
- Nœuds dans le même rack : 0,125 Go/sec à 1 Go/sec
- Nœuds entre les racks : 0,1 Go/sec

10

10

Spark : Haute performance et flot de données simple



11

11

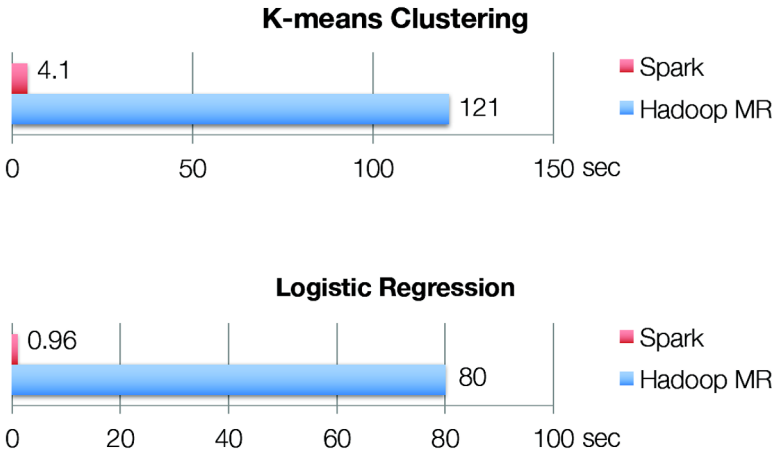
Performance : Spark vs MapReduce (1)

- Algorithmes itératifs
  - Spark est plus rapide : un flot de données simplifié
  - Évite de matérialiser les données sur HDFS après chaque itération
- Exemple : algorithme k-means, 1 itération
  - Lire HDFS
  - Map (Affecter l'échantillon de données au centroïde le plus proche)
  - GroupBy(Centroid_ID)
  - Shuffle du réseau (déplacement des données)
  - Reduce (Calculer les nouveaux centroïdes)
  - Wrtite HDFS

12

12

Performance : Spark vs MapReduce (2)



13

13

Code Hadoop vs Spark (comptage de mots)

```
public class WordCount {

public static class Map extends Mapper<LongWritable, Text, Text, IntWritable> {
   private final static IntWritable one = new IntWritable(1);
   private Text word = new Text();

   public void map(LongWritable key, Text value, Context context) throws IOException,
       String line = value.toString();
       StringTokenizer tokenizer = new StringTokenizer(line);
       while (tokenizer.hasMoreTokens()) {
           word.set(tokenizer.nextToken());
           context.write(word, one);
       }
   }
}

public static class Reduce extends Reducer<Text, IntWritable, Text, IntWritable> {

   public void reduce(Text key, Iterable<IntWritable> values, Context context)
     throws IOException, InterruptedException {
       int sum = 0;
       for (IntWritable val : values) {
           sum += val.get();
       }
       context.write(key, new IntWritable(sum));
   }
}

public static void main(String[] args) throws Exception {
   Configuration conf = new Configuration();

       Job job = new Job(conf, "wordcount");

   job.setOutputKeyClass(Text.class);
   job.setOutputValueClass(IntWritable.class);

   job.setMapperClass(Map.class);
   job.setReducerClass(Reduce.class);

   job.setInputFormatClass(TextInputFormat.class);
   job.setOutputFormatClass(TextOutputFormat.class);

   FileInputFormat.addInputPath(job, new Path(args[0]));
   FileOutputFormat.setOutputPath(job, new Path(args[1]));

   job.waitForCompletion(true);
}
```

```
val file = sc.textFile("hdfs://...")

val counts = file.flatMap(line => line.split(" "))
   .map(word => (word,1))
   .reduceByKey(_ + _)

counts.saveAsTextFile("hdfs://...")
```

- Code simple/léger
- Plusieurs étapes➔ Pipeline
- Opérations
  - Transformations : code d'utilisateur pour distribuer des données en parallèle
  - Actions : résultat final (output) à partir de données distribuées

14

14

Motivation (1)

MapReduce : Le moteur de traitement général et évolutif original de l'écosystème Hadoop

- Traitement des données basé sur le disque (fichiers HDFS)
- Résultats intermédiaires persistants (sur le disque)
- Les données sont rechargées à partir du disque à chaque itération → Input/Output coûteux
- Plus adapté au traitement batch de type ETL
- Input/Output coûteux → Ne convient pas aux charges de travail de traitement itératif ou en flux

15

15

Motivation (2)

Spark : Framework de calcul qui améliore les performances de MapReduce, mais conserve le modèle de base

- Framework de calcul sur des données basé sur la mémoire → évite les entrées/sorties coûteuses en gardant les résultats intermédiaires en mémoire
- Exploite la mémoire distribuée
- Se souvient des opérations appliquées aux données
- Calcul des données "en local" → Haute performance
- Meilleur pour les charges de travail itératives (ou de traitement de flux) et par lots

16

16

Résumé

- Le point de vue du génie logiciel
  - La base de codes Hadoop est énorme
  - Les contributions et les extensions de Hadoop sont lourdes
  - L'utilisation exclusive de Java entrave l'adoption de Hadoop à grande échelle, mais le support de Java est fondamental
- Point de vue du système
  - Un pipeline unifié
  - Un flux de données simplifié
  - Une vitesse de traitement plus rapide
- Point de vue de l'abstraction de données
  - Nouvelle abstraction fondamentale RDD
  - Facile à étendre avec de nouveaux opérateurs
  - Modèle de calcul plus descriptif

17

17

Plan

- Motivation
- Les bases de Spark
- Programmation de Spark

18

18

Les bases de Spark (1)

Spark : Cadre flexible de traitement des données en mémoire, écrit en Scala

Les objectifs :

- Simplicité (plus facile à utiliser) :
  - API pour Scala, Java et Python
- Généralités : API pour différents types de charges de travail
  - Batch, streaming, apprentissage automatique, graphes
- Faible latence (moins d'attente) : traitement en mémoire et mise en cache
- La tolérance aux fautes : Les fautes ne doivent pas être un cas particulier

19

19

Les bases de Spark(2)

Il y a deux façons de manipuler les données dans Spark

- Spark Shell :
  - Itératif - pour l'exploration des données
  - Python ou Scala
- Applications Spark
  - Pour le traitement des données à large échelle
  - Python, Scala ou Java

20

20

Le Shell Spark

Le Shell Spark permet l'exploration itérative des données (REPL)

Python Shell: `pyspark`

```
$ pyspark

Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 1.3.0
      /_/

Using Python version 2.7.8 (default, Aug 27
2015 05:23:36)
SparkContext available as sc, HiveContext
available as sqlCtx.
>>>
```

Scala Shell: `spark-shell`

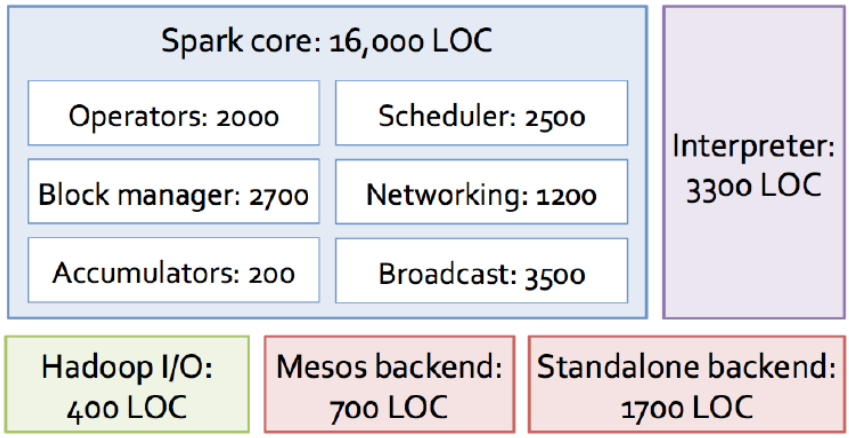
```
$ spark-shell

Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 1.3.0
      /_/

Using Scala version 2.10.4 (Java HotSpot(TM)
64-Bit Server VM, Java 1.7.0_67)
Created spark context..
Spark context available as sc.
SQL context available as sqlContext.

scala>
```

REPL Repeat/Evaluate/Print Loop

21

21

Spark Core : Code de base (2012)



22

22

Fondements de Spark

Exemple d'une application :

```
val sc = new SparkContext("spark://...", "MyJob", home,
   jars)

val file = sc.textFile("hdfs://...") // This is an RDD

val errors = file.filter(_.contains("ERROR")) // This is
   an RDD

errors.cache()

errors.count() // This is an action
```

- **Spark Context**
- **RDD : Données distribuées résilientes**
- **Transformations**
- **Actions**

23

23

Fondements de Spark

- Spark Context
- Ensembles de données distribuées résilientes (RDD)
- Transformations
- Actions

24

24

Spark Context (1)

- Chaque application Spark nécessite un contexte Spark qui est le point d'entrée principal de l'API Spark
- Shell Spark fournit un contexte Spark préconfiguré nommé "sc".

Python

```
Using Python version 2.7.8 (default, Aug 27 2015 05:23:36)
SparkContext available as sc, HiveContext available as sqlCtx.

>>> sc.appName
u'PySparkShell'
```

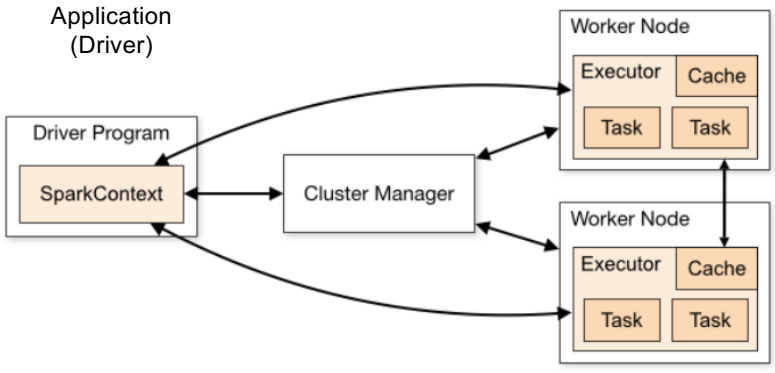
Scala

```
...
Spark context available as sc.
SQL context available as sqlContext.

scala> sc.appName
res0: String = Spark shell
```

25

Spark Context (2)

- Applications ➔ Driver ➔ Spark Context
- Le Spark Context représente le lien avec un cluster Spark

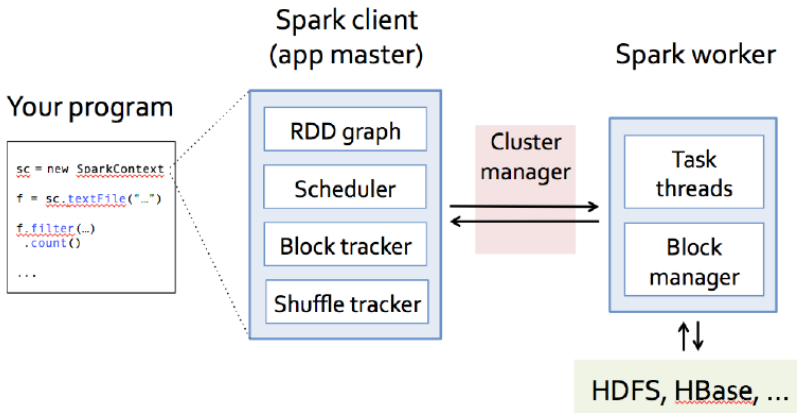


26

26

Spark Context (3)

- Le contexte Spark fonctionne comme un client et représente la connexion à un cluster Spark



27

27

Fondements Spark : RDD

Exemple d'une application :

```
val sc = new SparkContext("spark://...", "MyJob", home,
    jars)

val file = sc.textFile("hdfs://...") // This is an RDD

val errors = file.filter(_.contains("ERROR")) // This is
    an RDD

errors.cache()

errors.count() // This is an action
```

- Contexte de l'étincelle
- **Données distribuées résilientes (RDD)**
- Transformations
- Actions

28

28

RDD - données distribuées résilientes

Le **RDD** (Resilient Distributed Dataset) est l'unité fondamentale des données dans Spark **:** Une collection *immutable d'*objets (ou d'enregistrements) qui peuvent être exploités "en parallèle" (répartis sur un cluster)

Résilient - si des données en mémoire sont perdues, elles peuvent être recréées

- Récupération (recovery) si défaillance des nœuds
- Un RDD conserve la traçabilité; il peut être recréé à partir des RDD parents

Distribué - traité dans l'ensemble du cluster

- Chaque RDD est composé d'une ou plusieurs partitions
  - Plus de partitions - plus de parallélisme

Dataset -- les données initiales peuvent provenir d'un fichier ou être créées

29

29

RDDs

Idée clé : Ecrire des applications en termes de transformations sur des ensembles de données distribuées

- Objets répartis sur une couche de mémoire cache (cluster) qui stocke les données dans un cache distribué et tolérant aux pannes
- Peut utiliser le disque si l'ensemble de données ne tient pas en mémoire
- Construit par des transformations parallèles (map, filter, group-by, join, etc.)
- Reconstuit(s) automatiquement en cas d'échec
- Persistance contrôlable (mise en cache dans la RAM)

30

30

RDDs -- Immutabilité

- Immutabilité → traçabilité (Provenance)→ peut être recréé à tout moment → Tolérance aux pannes
- Évite les problèmes d'incohérence des données ; pas de mises à jour simultanées ; exactitude
- RDD en mémoire ou en cache sur disque ➔ Partage sûr entre les processus/tâches ➔Bonnes performances
- Compromis : (**Tolérance aux pannes et exactitude) vs** (**mémoire disque et CPU)**

31

31

Création d'un RDD

Trois façons de créer un RDD

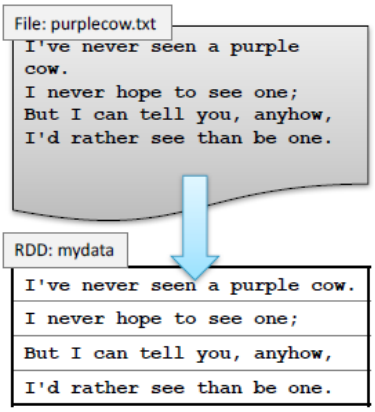
- A partir d'un fichier ou d'un ensemble de fichiers
- A partir des données en mémoire
- D'un autre RDD

32

32

Exemple : Un RDD basé sur un fichier

```
> val mydata = sc.textFile("purplecow.txt")
…
15/01/29 06:20:37 INFO storage.MemoryStore:
   Block broadcast_0 stored as values to
   memory (estimated size 151.4 KB, free 296.8
   MB)

> mydata.count()

…
15/01/29 06:27:37 INFO spark.SparkContext: Job
   finished: take at <stdin>:1, took
   0.160482078 s
4
```



33

33

Fondements de Spark

Exemple d'une application :

```
val sc = new SparkContext("spark://...", "MyJob", home,
   jars)

val file = sc.textFile("hdfs://...") // This is an RDD

val errors = file.filter(_.contains("ERROR")) // This is
   an RDD

errors.cache()

errors.count() // This is an action
```

- Contexte Spark
- **Données distribuées résilientes**
- **Transformations**
- **Actions**

34

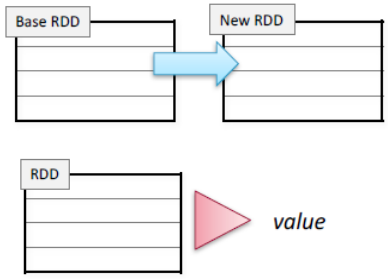
34

Opérations sur RDD

Deux types d'opérations

Transformations : Définir un nouveau RDD basé sur le(s) RDD actuel(s)

Actions : valeurs retournées (output)




```
val sc = new SparkContext("spark://...", "MyJob", home,
    jars)

val file = sc.textFile("hdfs://...") // This is an RDD

val errors = file.filter(_.contains("ERROR")) // This is
    an RDD

errors.cache()

errors.count() // This is an action
```

35

35

Transformations

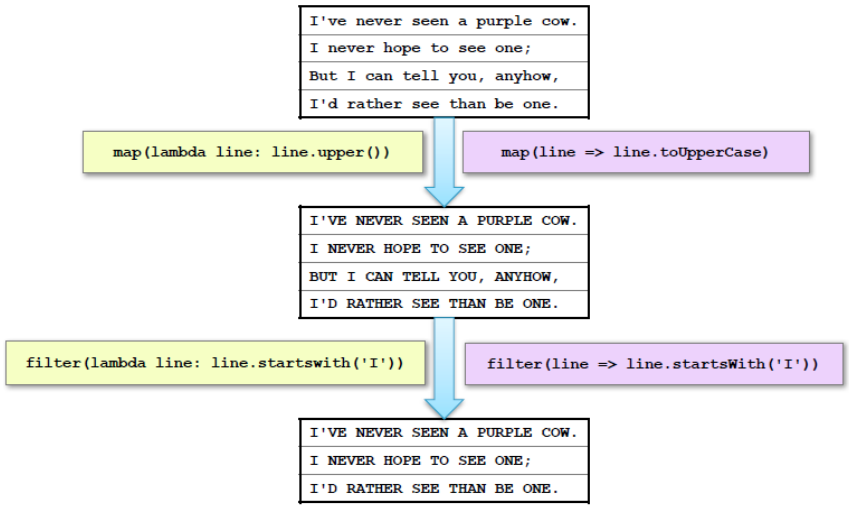
- Ensemble d'opérations sur un RDD qui définissent comment elles doivent être transformées
- Comme en algèbre relationnelle, l'application d'une transformation à un RDD donne un nouveau RDD (parce que les RDD sont immutables)
- Les transformations sont évaluées de façon paresseuse (lazy), ce qui permet des optimisations avant exécution

  map(), filter(), groupByKey(), sortByKey(), etc.

36

36

Exemple : map et filter



37

37

Actions

- Appliquer des chaînes de transformation sur les RDD, en effectuant éventuellement quelques opérations supplémentaires (par exemple, le comptage)
- Certaines actions ne stockent les données que dans une source de données externe (par exemple HDFS), d'autres vont chercher les données dans la RDD (et sa chaîne de transformation) sur laquelle l'action est appliquée, et les transmettent au Driver
- Actions connues
  - count() - renvoie le nombre d'éléments
  - take(*n*) - renvoie un tableau des *n* premiers éléments
  - collect()- renvoie un tableau de tous les éléments
  - saveAsTextFile(*file*) - enregistrer dans un fichier texte

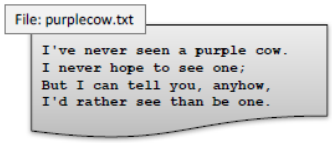
38

38

Exécution paresseuse des RDD (1)

Les données dans les RDD ne sont pas traitées tant qu'une action n'est pas effectuée

```
>
```



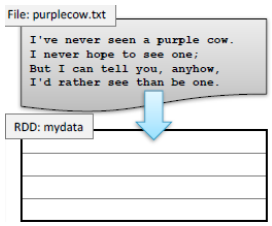
39

39

Exécution paresseuse des RDD (2)

Les données dans les RDD ne sont pas traitées tant qu'une action n'est pas effectuée

```
> val mydata = sc.textFile("purplecow.txt")
```



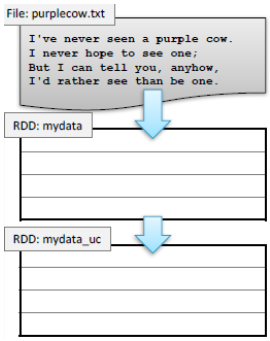
40

40

Exécution paresseuse des RDD (3)

Les données dans les RDD ne sont pas traitées tant qu'une action n'est pas effectuée

```
> val mydata = sc.textFile("purplecow.txt")
> val mydata_uc = mydata.map(line =>
  line.toUpperCase())
```



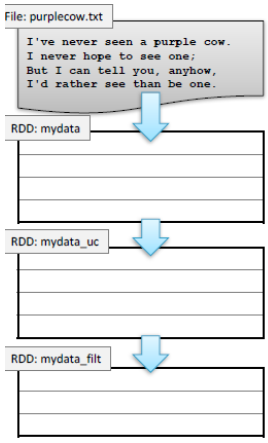
41

41

Exécution paresseuse des RDD (4)

Les données dans les RDD ne sont pas traitées tant qu'une action n'est pas effectuée

```
> val mydata = sc.textFile("purplecow.txt")
> val mydata_uc = mydata.map(line =>
  line.toUpperCase())
> val mydata_filt = mydata_uc.filter(line
  => line.startsWith("I"))
```



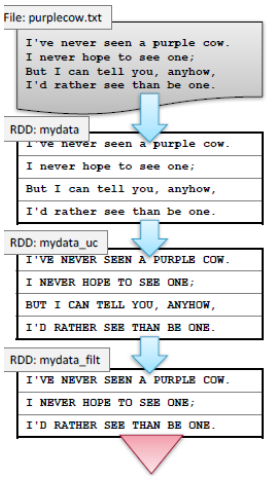
42

42

Exécution paresseuse des RDD (5)

Les données dans les RDD ne sont pas traitées tant qu'une action n'est pas effectuée

```
> val mydata = sc.textFile("purplecow.txt")
> val mydata_uc = mydata.map(line =>
  line.toUpperCase())
> val mydata_filt = mydata_uc.filter(line
  => line.startsWith("I"))
> mydata_filt.count()
3
```



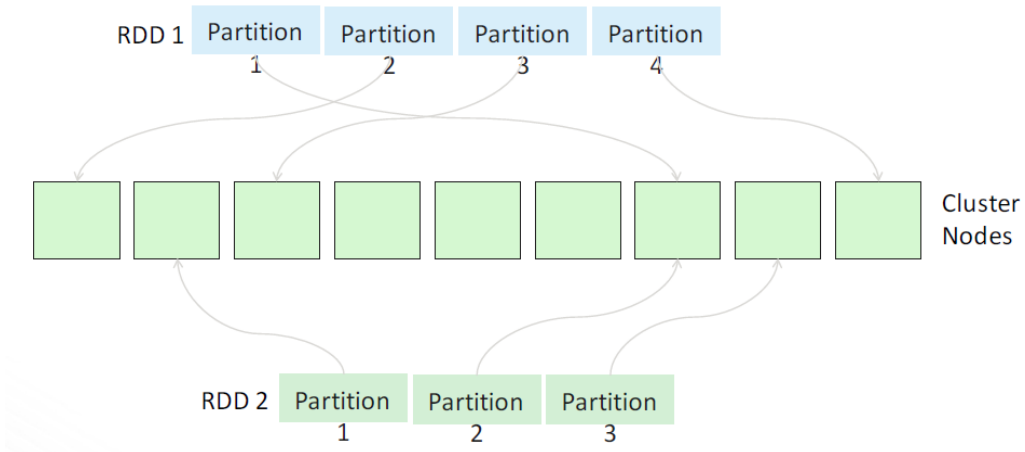
Exemple : Exploitation d'un log

Chargez les messages d'erreurs d'un log dans la mémoire, puis recherchez différents patterns (modèles) :

```
lines = spark.textFile("hdfs://...") HadoopRDD
erreurs = lines.filter(lambda s : s.startswith("ERREUR")) FilterRDD
messages = erreurs.map(lambda s : s.split("\t")[2])
messages.cache()
messages.filter(lambda s : "foo" dans s).count()
```

Résultat : recherche plein texte dans Wikipédia en 0,5 sec (contre 20 sec pour les données sur le disque)

RDD et Partitions

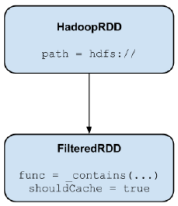


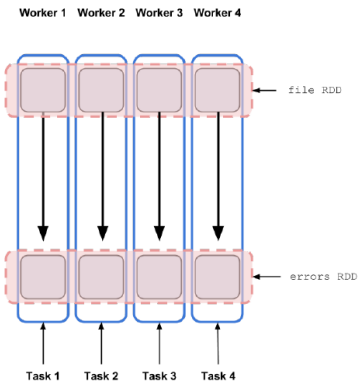
45

Graphes de RDD : Vues dataset vs. vues des partitions

Tout comme dans Hadoop MapReduce, chaque RDD est associé à des partitions (d'entrée)

```
val sc = new SparkContext("spark://...", "MyJob", home, jars)

val file = sc.textFile("hdfs://...") // This is an RDD

val errors = file.filter(_.contains("ERROR")) // This is an RDD

errors.cache()

errors.count() // This is an action
```





46

RDDs : Localité des données

- Principe de la localité des données
  - Même chose que pour Hadoop MapReduce
  - Évite les entrées/sorties de réseau, les travailleurs doivent gérer les données locales
- Localité et mise en cache des données
  - Première exécution : les données ne sont pas en cache, utilisez donc les préférences de localité de HadoopRDD (à partir de HDFS)
  - Deuxième tour : RDD filtré en cache, utilisez donc ses emplacements locaux
  - Si cache saturé, retour au HDFS

47

47

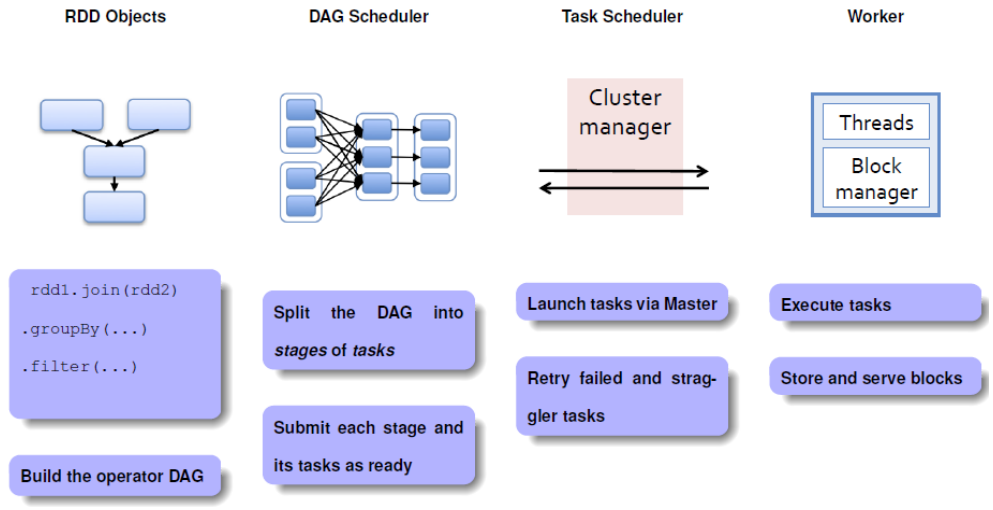
RDDs -- Résumé

- Les RDD sont des collections partitionnées, localisables et distribuées
  - Les RDD sont immutables (immuables)
- Les RDD sont des structures qui :
  - Soit pointent sur une source de données directe (HDFS)
  - Soit appliquent certaines transformations à (ses) RDD parent(s) pour générer de nouveaux éléments de données
- Calculs sur les RDD
  - Représenté par des DAG de provenance (lineage) évalués de façon paresseuse et composés de RDD chaînés

48

48

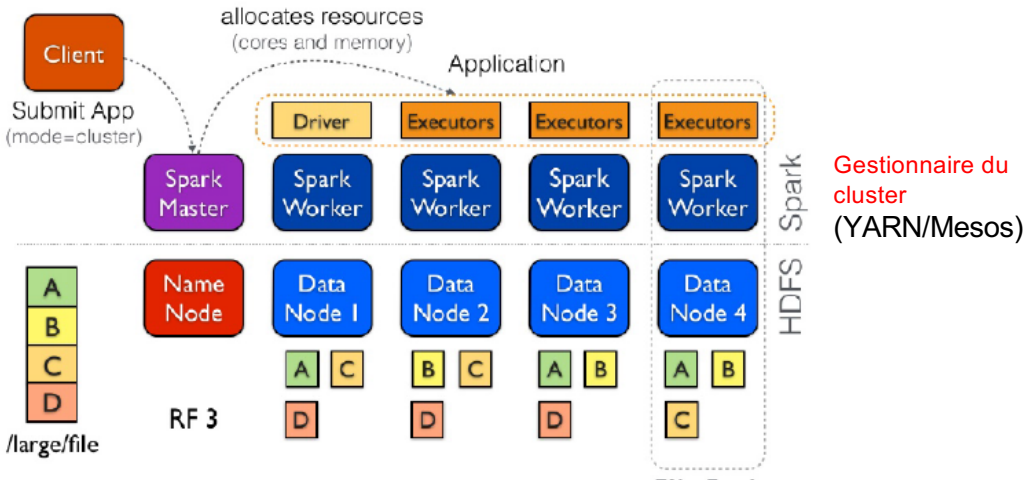
Cycle de vie d'un job Spark



49

49

Anatomie d'une application Spark



Gestionnaire du cluster (YARN/Mesos)

50

50

Mode d'utilisation typique d'un RDD

- Spark utilise la RDD pour transformer un objet d'entrée, pour générer une "nouvelle" version.
- Cette tâche pourrait transformer davantage le RDD avec des RDD supplémentaires, dans un style de programmation fonctionnelle.
- Finalement, certaines tâches consomment les résultats des RDD (ou peut-être plusieurs de ces RDD) dans le cadre d'un calcul de type MapReduce.

51

51

Techniques clés pour la performance

- Spark est un "moteur d'exécution pour le calcul des RDD", mais il décide également du *moment où il faut* effectuer le calcul proprement dit, de l'*endroit où* placer les tâches (sur le cluster Hadoop) et de la mise en cache éventuelle des résultats des RDD.
- Évite de recalculer un DDR en sauvegardant son résultat si celui-ci est à nouveau nécessaire, et de faire en sorte que les tâches soient exécutées à proximité de ces RDD mis en cache (ou bien dans un endroit où les tâches ultérieures utiliseront le même résultat de RDD)

52

52

Pourquoi est-ce une bonne stratégie ?

- Si les jobs MapReduce étaient des programmes quelconques, cela n'aurait pas d'importance.
- Mais en fait, le modèle MapReduce itère souvent les mêmes transformations sur les fichiers d'entrée.
- De plus, MapReduce est souvent exécuté de manière répétée jusqu'à ce qu'un modèle d'apprentissage machine converge, ou qu'un énorme lot d'entrée soit consommé
  - En mettant les RDD en cache, Spark peut éviter des efforts inutiles.

53

53

Plan

- Motivation
- Les bases de Spark
- Programmation Spark

54

54

Programmation Spark (1)

Créer des RDD

```
# Transformer une collection Python en RDD
sc.parallelize([1, 2, 3])

# Charger le fichier texte à partir du FS local, ou HDFS
sc.textFile("file.txt")
sc.textFile("directory/*.txt")
sc.textFile("hdfs://namenode:9000/path/file")

# Utiliser le format d'entrée Hadoop existant
(Java/Scala uniquement)
sc.hadoopFile(keyClass, valClass, inputFmt, conf)
```


Programmation Spark (2)

Transformations de base

```
nums = sc.parallelelize([1, 2, 3])

# Faire passer chaque élément par une fonction
carrés = nums.map(lambda x : x*x) // {1, 4, 9}

# Faire en sorte que les éléments passent un prédicat
even = carrés.filtre(lambda x : x % 2 == 0) // {4}
```

Programmation Spark (3)

Actions de base

```
nums = sc.parallelelize([1, 2, 3])

# Récupération du contenu des RDD en tant que collection locale
nums.collect() # => [1, 2, 3]

# Renvoyer les premiers éléments K
nums.take(2) # => [1, 2]

# Compter le nombre d'éléments
nums.count() # => 3

# Fusionner des éléments avec une fonction associative
nums.reduce(lambda x, y : x + y) # => 6
```

57

57

Programmation Spark (4)

Travailler avec des paires (clés, valeurs)

Les transformations de "réduction distribuée" de Spark opèrent sur des DDR de paires de valeurs clés

```
Python : paire = (a, b)
         paire [0] # => a
         paire [1] # => b

Scala :   val pair = (a, b)
         paire._1 // => a
         paire._2 // => b

Java : Paire de Tuple2 = nouveau Tuple2(a, b) ;
         paire._1 // => a
         paire._2 // => b
```

58

58

Programmation Spark (5)

Quelques opérations à valeur ajoutée

```
animaux de compagnie = sc.parallelize([("chat", 1), ("chien", 1),
("chat", 2)])

pets.reduceByKey(lambda x, y : x + y)      # => {(cat, 3), (dog,
1)}

pets.groupByKey()       # => {(chat, [1, 2]), (chien, [1])}

pets.sortByKey()        # => {(chat, 1), (chat, 2), (chien, 1)}
```

59

59

Exemple : Spark Streaming

Représente les flux comme une série de RDD au fil du temps (généralement des intervalles de moins d'une seconde, mais il est configurable)

```
val spammers = sc.sequenceFile("hdfs://spammers.seq")
sc.twitterStream(...)
   .filter(t => t.text.contains("Université de Santa Clara"))
   .transform(tweets => tweets.map(t => (t.user, t)).join(spammers))
   .print()
```

60

60

Combiner les bibliothèques (Unified Pipeline)

```
# Charger les données à l'aide de Spark SQL
points = spark.sql ("select latitude, longitude from tweets")

# Former un modèle d'apprentissage de la machine
modèle = KMeans.train(points, 10)

# L'appliquer à un flux
sc.twitterStream(...)
   .map(lambda t : (model.predict(t.location), 1))
   . reduceByWindow("5s", lambda a, b : a + b)
```

61

61

Définir le niveau de parallélisme

Toutes les opérations de RDD de la paire prennent un second paramètre optionnel pour le nombre de tâches

```
words.reduceByKey(lambda x, y : x + y, 5)
words.groupByKey(5)
visits.join(pageViews, 5)
```

62

62

MapReduce vs Spark (Résumé)

- Performance :
  - Alors que Spark est plus performant lorsque toutes les données tiennent en mémoire centrale (en particulier sur les clusters dédiés), MapReduce est conçu pour les données qui ne tiennent pas en mémoire
- Facilité d'utilisation :
  - Spark est plus facile à utiliser que Hadoop MapReduce car il est doté d'API conviviales pour Scala (son langage natif), Java, Python et Spark SQL.
- La tolérance aux fautes :
  - Traitement par lots : Réplication des HDFS
  - Traitement des flux : Réplication des RDD

63

63

Conclusion

1. Spark est un puissant "gestionnaire" pour le traitement des données.
2. Il est centré sur un planificateur de tâches (MapReduce) intelligent qui choisit l'endroit où exécuter chaque tâche ➔ co-localisation de la tâche avec les données.
3. Les structure de base sont des "RDD" : un mécanisme pour générer un fichier à partir d'une collection de données sous-jacente.
4. La mise en cache des RDD permet à Spark de fonctionner principalement à partir de données mappées en mémoire, pour des raisons de rapidité.

64

64