Chapitre 4
Industrialiser Spark

1. Améliorer les performances de temps

1.1 Dimensionner adéquatement le cluster

Vous avez vu la manière dont la distribution est rendue possible dans le framework ainsi que les méthodes qui vous permettent de faire de l'enrichissement de données et de l'apprentissage automatique. À présent, nous allons voir les techniques pour partir sereinement en production avec Spark. Nous commencerons par rappeler des principes que nous avons vus dans les chapitres précédents.

Spark est un framework distribué capable de traiter de forts volumes de données. Cela ne signifie pas pour autant que vous ne rencontrerez jamais de problèmes de performances. Les spécifications de votre cluster ont en premier lieu un impact sur celles-ci. S'il est sous-dimensionné, en termes de mémoire par exemple, par rapport à la quantité d'informations que vous voulez traiter, vous risquez d'avoir des problèmes.

1.2 Choisir la bonne API

Une fois que vous avez paramétré votre cluster de manière adéquate, vous pouvez suivre quelques principes qui vous éviteront des désagréments.

Nous avons longuement parlé des API DataFrame, Dataset et RDD. Les deux premières sont à privilégier. Elles contiennent un moteur d'optimisation qui fait toute la différence. Les objets `RDD` ne doivent être utilisés qu'en tout dernier recours. Si vous avez l'habitude de développer avec Scala et que vous avez besoin de performances plus que de robustesse à la compilation, l'API DataFrame est alors indiquée. Vous perdrez vos types, mais gagnerez en performances. La différence est cependant moindre entre les API DataFrame et Dataset qu'entre les API haut niveau et bas niveau. C'est même très peu comparable en réalité.

1.3 Éviter les UDF

Les UDF qui vous permettent d'ajouter votre propre logique dans le moteur Spark. Évitez-les au maximum. Le framework contient de nombreuses fonctions que vous pouvez assembler pour parvenir à différentes fins. Si toutefois vous ne pouvez couper à la création d'une `UDF`, cantonnez-la à une forme simple, sans effet de bord, et testez-la comme n'importe quel autre morceau de code.

1.4 User précautionneusement des actions

Ce sont les actions qui déclenchent réellement les transformations. Si vous lancez deux actions à la suite à partir d'un même objet `DataFrame`, cela signifie que les traitements vont s'opérer deux fois. Par exemple, si vous faites un filtre à l'aide de la fonction `filter`, puis comptez les données avec `count` et les affichez avec `show`, le traitement est joué deux fois.

```
dataframe_filtree: DataFrame =
dataframe.filter(dataframe["taille"] > 21)
```

Nous avons ici l'acte de transformation. À cela, nous ajoutons deux actions.

```
dataframe_filtree.count() // Exécute le filtre
dataframe_filtree.show() // Exécute le filtre
```

Afin d'éviter de lancer le filtre plusieurs fois, commencez par vous demander si la deuxième action que vous voulez réaliser est nécessaire. Fréquemment, certaines actions sont laissées sans que nous nous rendions compte de leur impact. C'est dommage parce que préjudiciable. Soyez donc vigilant quand vous employez des actions. Ensuite, vous pouvez utiliser une fonction dont nous n'avons pas encore parlé. Il s'agit de `cache`. Elle permet de garder en mémoire les transformations accomplies lors du lancement de la première action. Il s'agit d'appeler la méthode `cache` depuis l'objet `DataFrame`.

```
dataframe_filtree: DataFrame =
dataframe.filter(dataframe["taille"] > 21).cache()
```

Le comportement des actions est alors différent puisque la première exécute les transformations et enregistre le résultat en mémoire.

```
dataframe_filtree.count() // Exécute le filtre et
enregistre en mémoire l'état du DataFrame
dataframe_filtree.show() // N'exécute pas le filtre,
mais va puiser dans la mémoire pour retrouver l'objet
```

Les actions peuvent dégrader les performances.

Lors d'une action de mise en cache, par défaut, Spark garde en mémoire tout ce qu'il peut et enregistre sur disque le surplus. Cela reste plus efficace que de lire depuis les sources.

Il y a plusieurs bénéfices à utiliser la fonction `cache`. Le programme consomme moins de mémoire en lisant en mémoire que depuis le disque. Une opération de cache permet un point de sauvegarde. Il sera plus facile de calculer le dernier objet `RDD` dont le calcul a échoué depuis le cache.

Il y a plusieurs formes de caches.

– `DISK_ONLY` pour enregistrer sur disque sous une forme sérialisée.
– `MEMORY_ONLY` pour conserver en mémoire.
– `MEMORY_AND_DISK` pour mémoriser tout ce que Spark peut mémoriser. Le surplus est rangé sur le disque.

– `OFF_HEAP` pour sauvegarder dans la mémoire dite *off-heap*.

En interne, la fonction `cache` appelle `persist`. Vous pouvez directement vous servir de `persist`. C'est avec `persist` que vous pouvez décider d'utiliser une autre forme de cache.

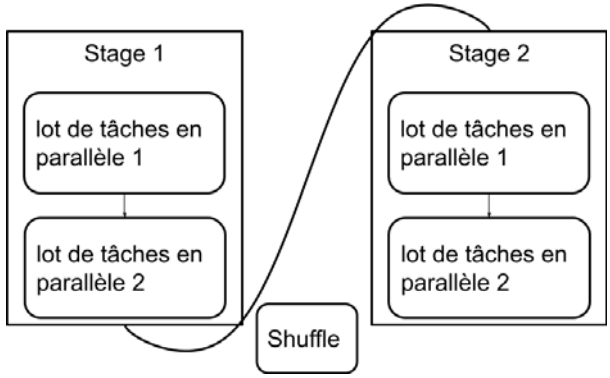
```
dataframe.persist(StorageLevel.MEMORY_ONLY)
```

`cache` est l'équivalent de `persist(StorageLevel.MEMORY_AND_DISK)`. Attention, la fonction `cache` peut être très gourmande. Elle ne solutionnera pas tous vos problèmes. Ne l'utilisez pas si vous n'avez pas deux actions qui se suivent. Cela dit, mieux vaut éviter d'avoir plusieurs actions à la suite. À condition évidemment que ce ne soit pas au détriment de vos applications. Il s'agit de trouver un juste équilibre.

1.5 Éviter le shuffle

1.5.1 Rappel du concept de shuffle

Dans Spark, des stages exécutent des lots successifs de tâches parallélisées. Entre chaque stage, s'exécute une étape de shuffle qui prend les données des partitions et les dispatche dans d'autres.



Stages et shuffles

Spark a recours à des étapes de shuffle quand il y est contraint. Il regroupe toutes les transformations étroites qui peuvent être faites sans l'aide de personne sur une seule et même partition. Au moment où une transformation large demande à être exécutée, il doit revoir ses plans et crée un nouveau stage. C'est par exemple le cas lors d'une jointure ou d'une agrégation par clé. Afin de réaliser l'action, Spark doit alors retrouver toutes les clés pour opérer la jointure ou l'agrégation correctement. C'est la raison pour laquelle il y a une étape de shuffle dans laquelle tous les éléments ayant les mêmes clés sont regroupés. Ils sont ensuite envoyés sur de nouvelles partitions. Les données sont redistribuées. Cette étape est coûteuse. Si nous pouvons l'éviter, autant le faire. C'est souvent difficile sans remettre en cause toute l'application. Cependant, il existe quelques solutions que nous allons voir dès maintenant.

1.5.2 Penser mégadonnées

Si vous voulez éviter d'avoir à faire de nombreuses jointures, vous devez cesser de penser le rangement de vos données comme s'il s'agissait d'une table de données classique. Si vous travaillez avec Spark, il y a de fortes chances pour que vos systèmes de stockage soient conçus pour de forts volumes de données. Vous ne devez donc pas multiplier les petites tables normalisées avec des clés étrangères. Ces concepts ne sont pas inutiles, mais plutôt adaptés aux systèmes avec peu de volumes de données qui ne cherchent à faire ni de l'analyse de données ni de l'apprentissage automatique. Dans notre cas, pour éviter les jointures, nous avons besoin d'enregistrer des tables/fichiers qui contiennent de nombreuses informations. Puisque nous les enregistrons la plupart du temps dans des formats adaptés comme Parquet, nous pouvons avoir un nombre large de colonnes. Cela n'évitera pas toutes les jointures. En réalité, dans une application Spark, nous en développons beaucoup parce que nous faisons de nombreuses manipulations. Cependant, penser forts volumes dénormalisés vous aidera à éviter quelques étapes de shuffle.

1.5.3 Différentes stratégies de jointure

Il existe plusieurs stratégies pour faire des jointures avec Spark. Nous n'allons pas toutes les énumérer, mais nous concentrer sur celle qui est appelée *broadcast* (« diffusion » en français). Quand vous joignez deux sources, si l'une d'entre elles est assez petite, Spark l'envoie sur tous les nœuds. Il n'y a pas de shuffle ici. Le travail est alors moins conséquent et votre application plus performante. Cependant, avec un trop fort volume de données, ce n'est pas une bonne idée. Vous aurez des problèmes de mémoire et donc de performances. C'est l'outil qui décide la stratégie à adopter. Vous pouvez cependant modifier cela avec la fonction `hint`.

```
dataframe.hint("broadcast").join(autre_dataframe, "clef")
```

C'est là une option à utiliser précautionneusement.

1.5.4 Les fonctions coalesce et repartition

Il y a aussi des actions qui entraînent davantage d'opérations de shuffle que d'autres. Par exemple, quand vous voulez enregistrer un seul fichier à la suite de transformations, deux possibilités s'offrent à vous. Par défaut, Spark étant un framework distribué, il enregistre en parallèle et ainsi crée plusieurs documents pour contenir les données. Pour modifier ce comportement, vous avez les fonctions `coalesce` et `repartition`. En termes de résultat, elles sont interchangeables. Vous demandez à rassembler ou à répartir le travail sur les données sur un nombre précis de partitions.

```
dataframe.repartition(1)
dataframe.coalesce(1)
```

Nous voyons la différence entre deux méthodes quand nous nous intéressons à leurs performances de calcul. La fonction `repartition` interprète la demande comme une augmentation du nombre de partitions, tandis que `coalesce` interprète la demande comme une diminution de leur nombre. La logique n'étant pas la même, `repartition` opérera davantage d'opérations de shuffle que `coalesce`. En fonction de ce que vous cherchez réellement à faire, il s'agit d'utiliser la bonne méthode.

2. Tester avec Spark

2.1 Tester sans Spark

Il est assez rare de ne pas avoir besoin de tester automatiquement un programme. Sauf si vous développez quelque chose d'expéditif voué à ne plus être utilisé dans les semaines à venir, un bouclier de tests automatisés est indispensable. Nous allons voir quelques astuces qui vous aideront à tester vos programmes avec Spark.

Nous parlerons ici de tests unitaires.

Nous basculons dans Scala pour un instant. Supposons que nous travaillons avec l'API Dataset. Nous créons une case class nommée `Diamant`.

```
case class Diamant(couleur: String, prix: Int)
```

Nous téléchargeons un fichier auquel nous appliquons la case class `Diamant` et obtenons ainsi un `Dataset`.

```
val diamants: Dataset[Diamant] =
spark.read.csv("diamants.csv").as[Diamant]
```

Nous voulons à présent récupérer uniquement la couleur du diamant.

```
val result: Dataset[String] = diamants.map(diamant => {
  diamant.couleur
})
```

Ici, il est possible de tester cette partie de code sans faire intervenir Spark. Ce que nous voulons vérifier, c'est notre capacité à extraire la couleur. Or, nous pouvons déplacer ce code dans une fonction.

```
def selectionneCouleur(diamant: Diamant): String = {
  diamant.couleur
}
```

Nous l'utilisons ensuite.

```
val resultat: Dataset[String] = diamants.map(selectionneCou-
leur(_))
```