Chapitre 1-4
Installer son environnement de travail

1. Introduction

Il ne s'agit ici que de CPython, l'implémentation de référence de Python, et non de PyPy ou Jython.
Quel que soit votre système d'exploitation, vous pouvez installer Python en lisant ce chapitre puis, dans un second temps, installer des bibliothèques tierces au gré de vos besoins (cf. section Installer une bibliothèque tierce) et vous pourrez créer des environnements virtuels (cf. section Créer un environnement virtuel).
Si vous souhaitez installer d'un seul coup Python ainsi que Jupyter (anciennement IPython) et la plupart des bibliothèques scientifiques ou d'analyse de données, vous pouvez aller directement à la section Installer Anaconda, pour installer celui-ci en lieu et place de Python. Vous disposerez alors d'autres méthodes pour gérer les environnements virtuels et pour installer des bibliothèques tierces.

2. Installer Python

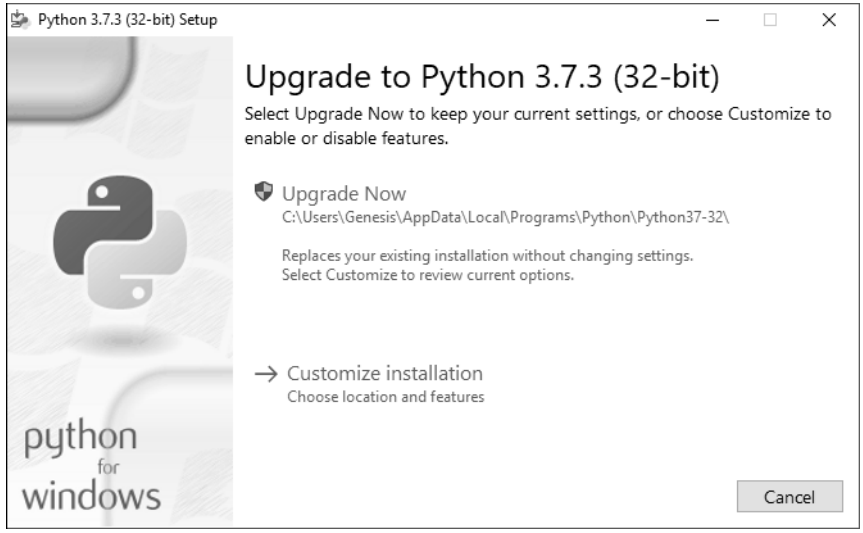
2.1 Pour Windows

Le système d'exploitation Windows requiert usuellement l'utilisation d'un installateur pour pouvoir installer un logiciel quel qu'il soit. Si vous disposez de Windows, vous devriez en avoir l'habitude. Python ne déroge pas à la règle.
Pour installer Python, vous devez donc aller sur le site officiel (https://www.python.org/downloads/) pour télécharger l'installateur adéquat. Comme vous pourrez le constater, on vous met en avant un accès rapide à la dernière version (au moment où ces lignes sont écrites, la 3.11.0), puis un accès aux dernières versions encore actives (actuellement la version 3.10 qui reçoit encore des corrections d'anomalies, puis les versions 3.9 à 3.7 qui reçoivent des corrections de sécurité uniquement).

Il est également possible de télécharger la toute dernière version de la branche 2.7 qui est en fin de vie (elle n'est plus mise à jour), car il existe encore de nombreux projets n'ayant pas encore migré.
Le support correctif dure 2 ans après la première sortie de la version et le support de sécurité dure 5 ans.
Pour notre part, nous vous conseillons la dernière 3.x, mais vous êtes libre d'installer celle que vous souhaitez ou même d'en installer plusieurs suivant vos contraintes, il n'y a pas d'objection à cela.
Une fois le téléchargement effectué, vous devez lancer l'installateur (et éventuellement passer quelques protections de votre système qui vous demande d'accorder votre confiance à cet installateur), pour observer l'écran suivant :

Comme vous pouvez le constater, il est possible de personnaliser l'installation en choisissant le chemin d'installation du logiciel ou en choisissant de ne pas sélectionner quelques fonctionnalités, mais nous ne le conseillons pas.
Nous vous recommandons en revanche de cocher la case **Add python.exe to PATH** afin de configurer la variable PATH du terminal pour rendre Python accessible plus facilement.

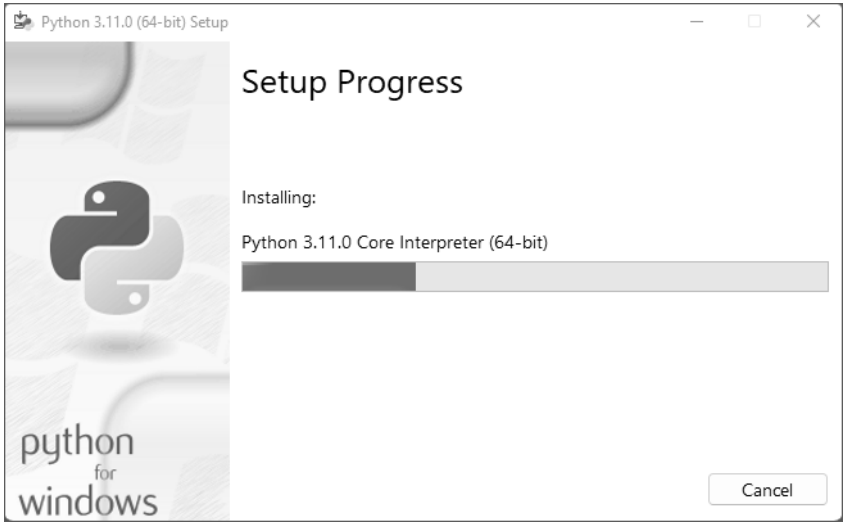
Si vous avez déjà une ancienne version de Python installée de la même branche (dans cet exemple, Python 3.7.2 est déjà installé), vous pourrez la mettre à jour à l'aide du même installeur :



Par contre, si vous avez déjà la version 3.7.1 et que vous installez la version 3.11, cette dernière ne viendra pas remplacer la précédente, mais s'installera à côté. Si vous souhaitez remplacer, il vous faudra donc désinstaller proprement la toute dernière version installée, ce qu'il est possible de faire en relançant l'installeur d'origine.

Nous vous encourageons à garder les installeurs sur votre PC, car ils pourraient devenir indisponibles au téléchargement si trop vieux.

Quel que soit le scénario, vous arriverez devant un écran vous montrant la progression de l'installation et vous n'aurez qu'à fermer la fenêtre une fois celle-ci terminée :



Vous êtes maintenant prêt à utiliser Python.

2.2 Pour Mac

Il faut savoir qu'une version de Python est déjà préinstallée sur Mac, car Mac OS X l'utilise pour ses propres besoins et Python est intégré à son propre cycle de développement. Cependant, si vous souhaitez une version différente de celle qui est déjà présente, vous pouvez l'installer, sachant qu' il n'y a pas de contre-indication à posséder plusieurs versions de Python sur la même machine.

Pour installer Python sur Mac OS X, la procédure à suivre est similaire à celle pour Windows. Il faut donc se rendre sur le site officiel (https://www.python.org/downloads/mac-osx/), télécharger un installateur correspondant à sa configuration et suivre les étapes.

Pour les utilisateurs de Mac, sachez que Python dispose d'une bonne intégration de ses spécificités, en particulier vis-à-vis de Objective-C, le langage de programmation avec lequel est développé Mac OS X, et Cocoa, interface de programmation de Mac OS X.

2.3 Pour GNU/Linux et BSD

Les différentes distributions libres utilisent nativement Python, notamment pour des parties sensibles. Python y est donc tout naturellement déjà installé, généralement sous la dernière version de la branche 2.x. Cependant, ici comme ailleurs, il n'y a pas d'objections à utiliser plusieurs versions de Python.
Le plus simple reste d'utiliser votre gestionnaire de paquets, ce qui peut se faire via un outil graphique, comme Synaptic pour Debian :

Il suffit alors de faire une recherche sur le mot-clé **`python`** pour voir les différentes versions (sur une ancienne Debian, ce sont Python 2.6, 2.7 et 3.2).
Par contre, tous les paquets python3-xxxxx que vous pouvez voir ici sont des bibliothèques tierces et non Python lui-même. Nous en parlerons plus tard dans ce chapitre.
Une fois les paquets souhaités sélectionnés, il ne manque plus qu'à les installer en cliquant sur le bouton **Appliquer**.
Notez que tout ceci peut se faire par la simple ligne de commande, toujours en utilisant votre gestionnaire de paquets qui peut être apt-get, aptitude, yum, emerge, pkg_add ou autre.
Voici par exemple pour une distribution Debian ou Ubuntu :

```
$ sudo aptitude install python3
```

Ceci ne permet cependant pas de choisir la version que l'on souhaite, à moins d'aller trouver des sources alternatives. Si l'on veut avoir la toute dernière version de Python, il faudra la plupart du temps passer par la compilation.

2.4 Par la compilation

Compiler Python n'est pas en soi une tâche très complexe. C'est par contre souvent une tâche imposée lorsque l'on ne travaille pas avec des conteneurs. En effet, en entreprise, on développe souvent des applications qui sont destinées à être hébergées. Il est alors impératif de travailler sur votre propre poste avec une version de Python qui soit la même que celle existante sur la machine de production.

Sous GNU/Linux, mais aussi sous d'autres systèmes, il est possible de compiler la version de Python que l'on souhaite. Après tout, Python n'est rien d'autre qu'un programme écrit en C. Pour ce faire, il faut aller télécharger le code source (https://www.python.org/downloads/source/), qui prend la forme d'une archive, puis décompresser celle-ci, se placer dans le répertoire ainsi obtenu et taper ces quelques commandes :

```
$ ./configure –prefix=/path/to/my/python/directory
$ make
$ sudo make altinstall
```

Notez que dans cette dernière ligne, nous n'utilisons pas la commande **`make install`**, qui aurait pour effet de remplacer votre Python système par le Python que vous compilez, ce qui pourrait avoir des conséquences indésirables voire désastreuses.

Notez également que vous choisissez lors de la configuration le chemin dans lequel vous placerez vos bibliothèques Python. En général, l'usage veut que l'on utilise **`/opt`**, mais il n'y a pas de règle, tout dépend des pratiques de votre entreprise ou votre expérience en la matière.

Si vous venez d'installer Python 3.5 par cette méthode, vous aurez alors maintenant accès à ce programme en l'appelant ainsi, depuis votre terminal :

```
$ python3.5
```

Par cette même méthode, vous pouvez installer les dernières versions (https://www.python.org/download/pre-releases/) de Python qui ne sont pas encore sorties (alphas ou betas), ce qui vous permet de les tester en avant-première !

Notons que, par cette méthode, toutes les bibliothèques de Python ne fonctionneront pas. En effet, lorsqu'elles ont besoin d'autres bibliothèques C, il faut effectuer des compilations croisées et utiliser les différents en-têtes de ces bibliothèques. C'est le cas par exemple pour faire fonctionner Curses, ReportLab (génération de fichiers PDF) ou encore PyUSB (accès aux ports matériels USB).

Dans ce cas-là, la commande **`./configure`** devra recevoir des arguments supplémentaires et vous devrez trouver un tutoriel en ligne pour vous indiquer la démarche, laquelle peut être plus ou moins complexe.

2.5 Pour un smartphone

Installer une machine virtuelle Python sur un smartphone est possible. Pour Android, la procédure est assez simple puisqu'il existe un produit dédié (http://qpython.com/), tout comme sur Windows Phone (https://apps.microsoft.com/store/detail/python-39/9P7QFQMJRFP7). Pour iOS, c'est une autre paire de manches (https://github.com/linusyang/python-for-ios) étant donné que l'utilisateur est enfermé dans un système sur lequel il n'a aucun contrôle.

3. Installer une bibliothèque tierce

Remarque

Si vous abhorrez le terminal, sachez que vous pouvez installer une bibliothèque tierce depuis votre IDE, ce qui sera probablement plus aisé pour vous.

3.1 À partir de Python 3.4

Pour installer une bibliothèque tierce, vous devez simplement connaître son nom. Celui-ci est généralement assez intuitif. Par exemple, la bibliothèque permettant de communiquer avec un serveur Redis s'appelle redis.
Il peut y avoir des variations. Par exemple, la bibliothèque de référence pour traiter du XML est lxml et, plus complexe, celle pour BeautifulSoup est bs4. En recherchant comment répondre à un besoin sur le Net ou sur PyPi (https://pypi.python.org/pypi), vous trouverez rapidement une bibliothèque de référence.
Sur des sujets plus confidentiels, il vous arrivera de trouver plusieurs petites bibliothèques. Vous pourrez alors les tester et choisir celle que vous utiliserez pour votre projet.
Sachez que vous pouvez aussi conduire une recherche directement depuis votre terminal :

```
$ pip search xml
$ pip search soup
```

Cela vous donnera une liste de bibliothèques accompagnée d'une courte description, à la manière de ce que font les gestionnaires de paquets sous Linux (lesquels sont écrits en Python, au passage).
Sachez que **`pip`** existe quel que soit votre système d'exploitation (vous devez être familier avec le terminal de votre système, cependant) et que depuis la version 3.4 de Python, il est installé automatiquement avec celui-ci. Si ce n'est pas votre cas, consultez la section suivante : Pour une version inférieure à Python 3.4.
`pip` est un outil formidable. Si vous utilisez une version de Python qui est celle du système, vous utiliserez alors la commande **`pip`** pour gérer les bibliothèques. Si vous utilisez une autre version, telle que Python 3.5, alors vous utiliserez la commande **`pip-3.5`**. Pour Python 3.3, ce sera **`pip-3.3`**. Dans les exemples suivants, il vous faudra prendre en compte cette particularité.
Cet outil vous permettra d'installer une bibliothèque à sa dernière version ainsi que toutes les bibliothèques dépendantes. En effet, il n'est pas rare qu'une bibliothèque de Python ait besoin d'une autre bibliothèque (ou de plusieurs) pour fonctionner. Par exemple, l'installation de redis se fait par cette commande :

```
$ pip install redis
```

On peut aussi choisir la version à installer :

```
$ pip install -Iv redis==2.10.5
```

Ou mettre à jour la bibliothèque à une version précise :

```
$ pip install -U redis==2.10.5
```

Ou à la dernière version :

```
$ pip install -U redis
```

Et on peut la désinstaller :

```
$ pip uninstall redis
```

Une fonctionnalité très importante permet d'obtenir la liste des bibliothèques déjà installées (quelle que soit la manière dont elles ont été installées) :

```
$ pip freeze
```

Ce que l'on peut mettre dans un fichier :

```
$ pip freeze > requirements.txt
```

Pour installer tous les paquets ainsi listés, il faut procéder ainsi :

```
$ pip install -r requirements/base.txt
```

Cette méthode est particulièrement utile dans le cadre d'un environnement virtuel ; nous y reviendrons.
Il est possible de retrouver des informations sur un paquet déjà installé :

```
$ pip show django-redis
---
Name: django-redis
Version: 4.3.0
Location: /path/to/my/env/lib/python3.4/site-packages
Requires: redis
```

On voit ici que le paquet **`django-redis`** a une dépendance vers **`redis`** : en l'installant, on installe automatiquement **`redis`**.
Mettre à jour ce paquet met à jour automatiquement les dépendances :

```
$ pip install -U django-redis
```

Si on ne veut pas mettre à jour les dépendances, on peut procéder ainsi :

```
$ pip install -U --no-deps django-redis
```

On peut aussi installer plusieurs bibliothèques en même temps :

```
$ pip install django-redis==4.3.0 bs4 lxml
```

Cette commande installera donc automatiquement redis s'il n'est pas installé, car il est déclaré comme dépendance.
Cette commande a cependant des limites. En effet, si vous installez une bibliothèque tierce qui utilise une bibliothèque C, vous devrez disposer des en-têtes C correspondants (paquets **`dev`** pour Debian ou **`devel`** pour Fedora). Il faut donc avoir un peu de pratique dans ce genre de situation pour savoir déjouer ces pièges.

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.

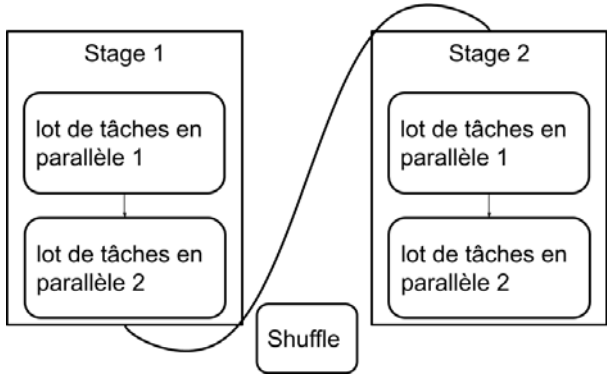
```
dataframe.persist(StorageLevel.MEMORY_ONLY)
```

`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(_))
```