

Spark optimisation

Introduction

Spark **est couramment utilisé** pour appliquer des transformations sur des données, structurées dans la plupart des cas.

Deux scénarios motivent particulièrement son recours.

- Le premier est **lorsque les données à traiter sont trop importantes vis-à-vis des ressources** de calcul et de mémoire à disposition. C'est ce que l'on appelle le phénomène *big data*.
- Enfin, c'est également une alternative lorsque l'on veut **accélérer un calcul**, en mettant à **contribution plusieurs machines** au sein d'un même réseau.

Dans ces deux cas, une préoccupation majeure est alors l'optimisation du temps de calcul d'un traitement nommé *job Spark*.

L'objectif est de proposer et détailler une **stratégie d'optimisation d'un *job Spark*** lorsque les ressources sont limitées. En effet, nous pouvons influencer sur de nombreuses **configurations de *Spark*** avant de recourir à l'élasticité du *cluster*. Cette stratégie peut ainsi être expérimentée en premier lieu.



Quels sont les principaux leviers sur lesquels nous pouvons jouer en priorité pour maximiser nos chances de réduire le temps de calcul ?

Partons d'un cas d'usage...

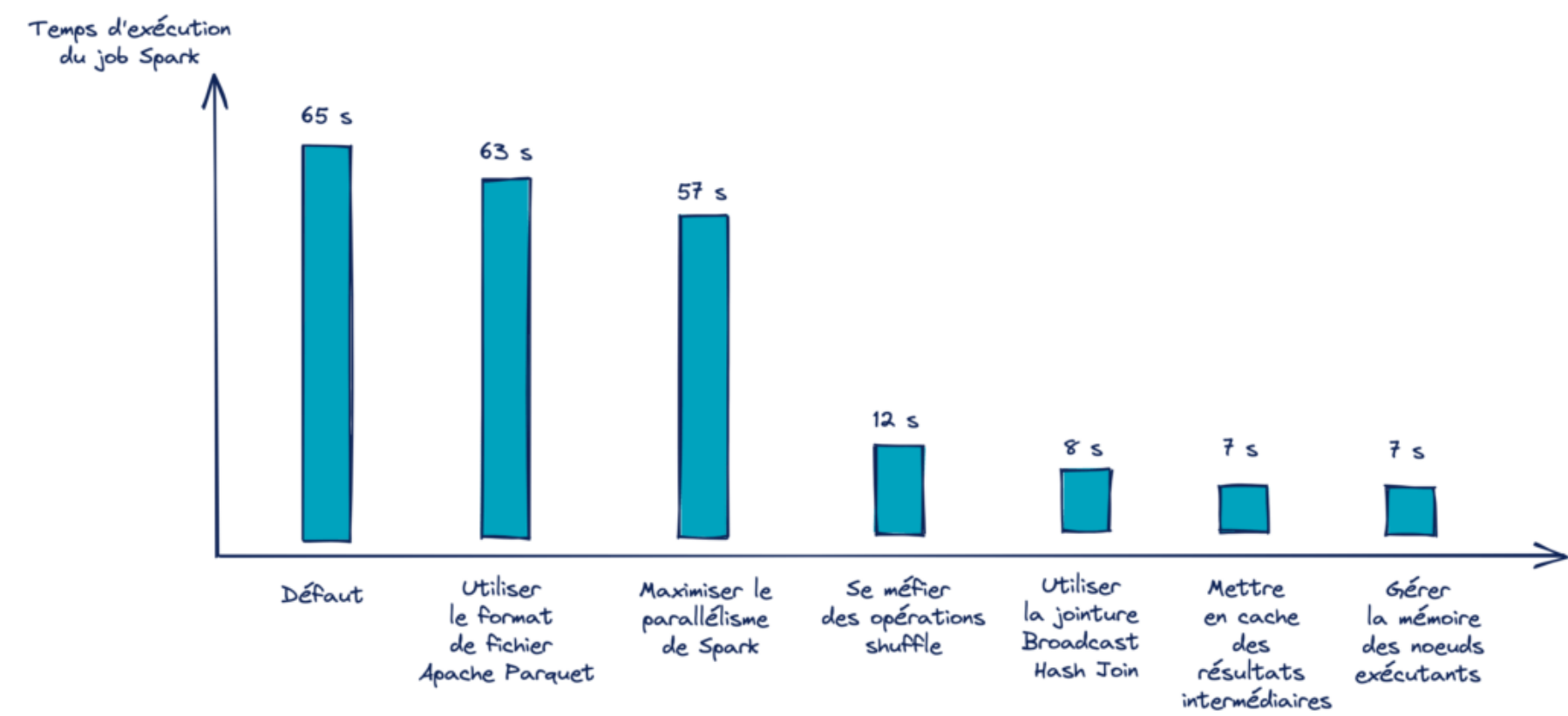
Cas d'usage

Nous nous sommes basés sur un cas d’usage pour construire un job Spark.

Le traitement permet de grouper des villes françaises selon des variables météorologiques et démographiques. Cette tâche est appelée classification non supervisée ou clustering en Machine Learning.

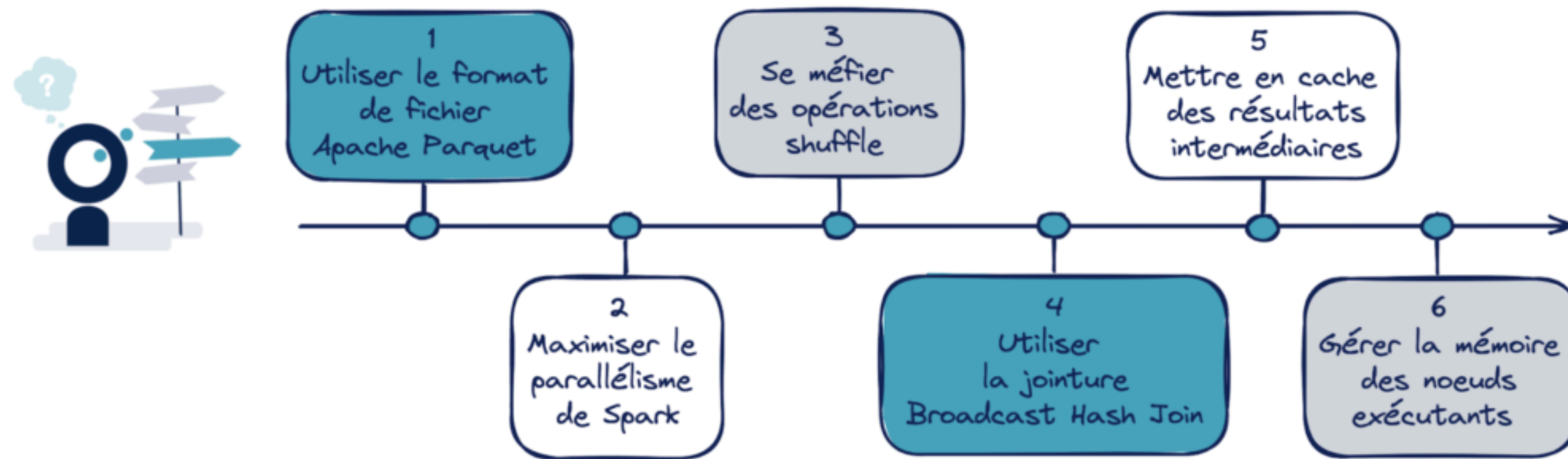
L’exemple illustre des fonctionnalités courantes d’un pipeline en Spark, c’est-à-dire une phase de prétraitement des données (chargement, nettoyage, fusion de différentes sources, feature engineering), l’estimation des paramètres d’un modèle de Machine Learning, et enfin l’écriture sur disque des résultats.

En agissant sur certains leviers nous pouvons observer l’influence des différentes optimisations effectuées sur le graphique suivant.



Cas d'usage

Concrètement, la stratégie exposée est dite *gloutonne*, c'est-à-dire que l'on fait le meilleur choix à chaque étape du procédé sans retour en arrière. La démarche est illustrée par un **fil conducteur en six recommandations**.



Comment modifier les configurations d'un job Spark ?

Il y a trois façons de modifier les configurations d'un job Spark:

- En utilisant les **fichiers de configuration** présents dans le dossier racine de *Spark*.

Nous pouvons par exemple personnaliser les fichiers standardisés suivants:

- *conf/spark-defaults.conf.template*
- *conf/ log4j.properties.template*
- *conf/spark-env.sh.template*
- Ces modifications affectent le *cluster Spark* et toutes ses applications.

- En **ligne de commande** à l'aide de l'argument **-conf**

Ex :

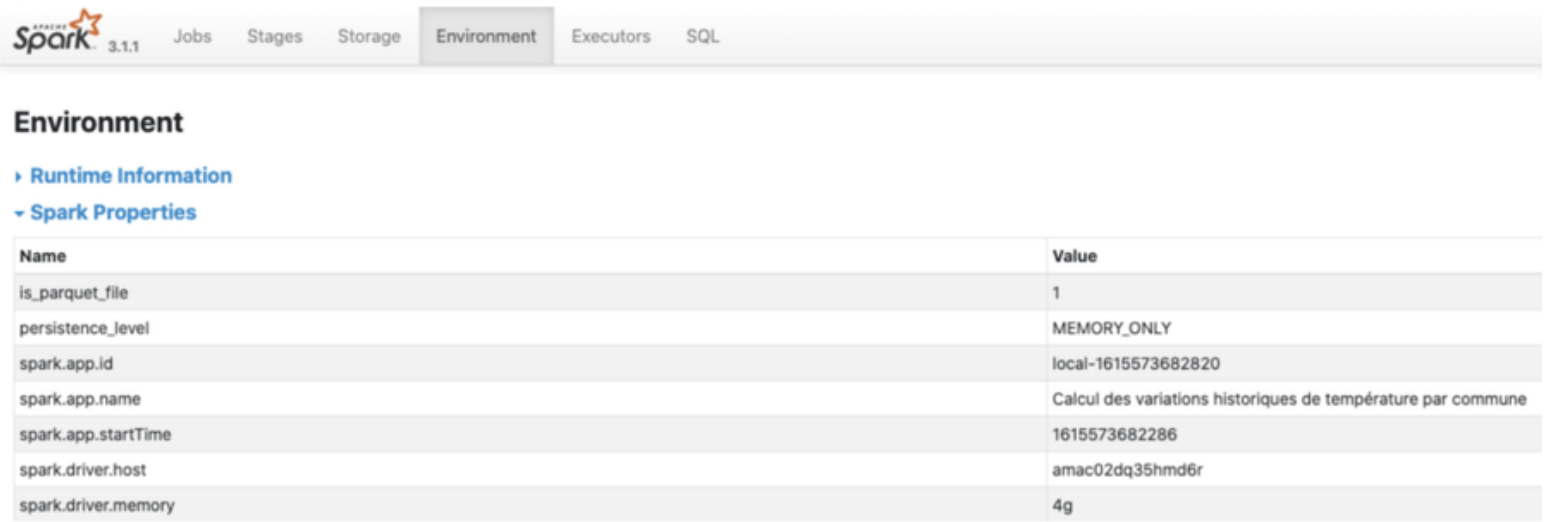
```
spark-submit --conf "spark.sql.shuffle.partitions=5" main.py
```

- Directement dans le **code de l'application Spark**

Ex:

```
spark = pyspark.sql.Session.builder.getOrCreate()
spark.conf.set("spark.sql.shuffle.partitions", 10)
```

Ces paramètres de configuration sont visibles en lecture seule dans l'onglet Environment de l'interface graphique de Spark.



The screenshot shows the Spark web interface with the 'Environment' tab selected. Below the tab, there are links for 'Runtime Information' and 'Spark Properties'. A table displays various Spark configuration properties and their values.

Name	Value
is_parquet_file	1
persistence_level	MEMORY_ONLY
spark.app.id	local-1615573682820
spark.app.name	Calcul des variations historiques de température par commune
spark.app.startTime	1615573682286
spark.driver.host	amac02dq35hmd6r
spark.driver.memory	4g

1ère recommandation : Utiliser le format de fichier Apache Parquet

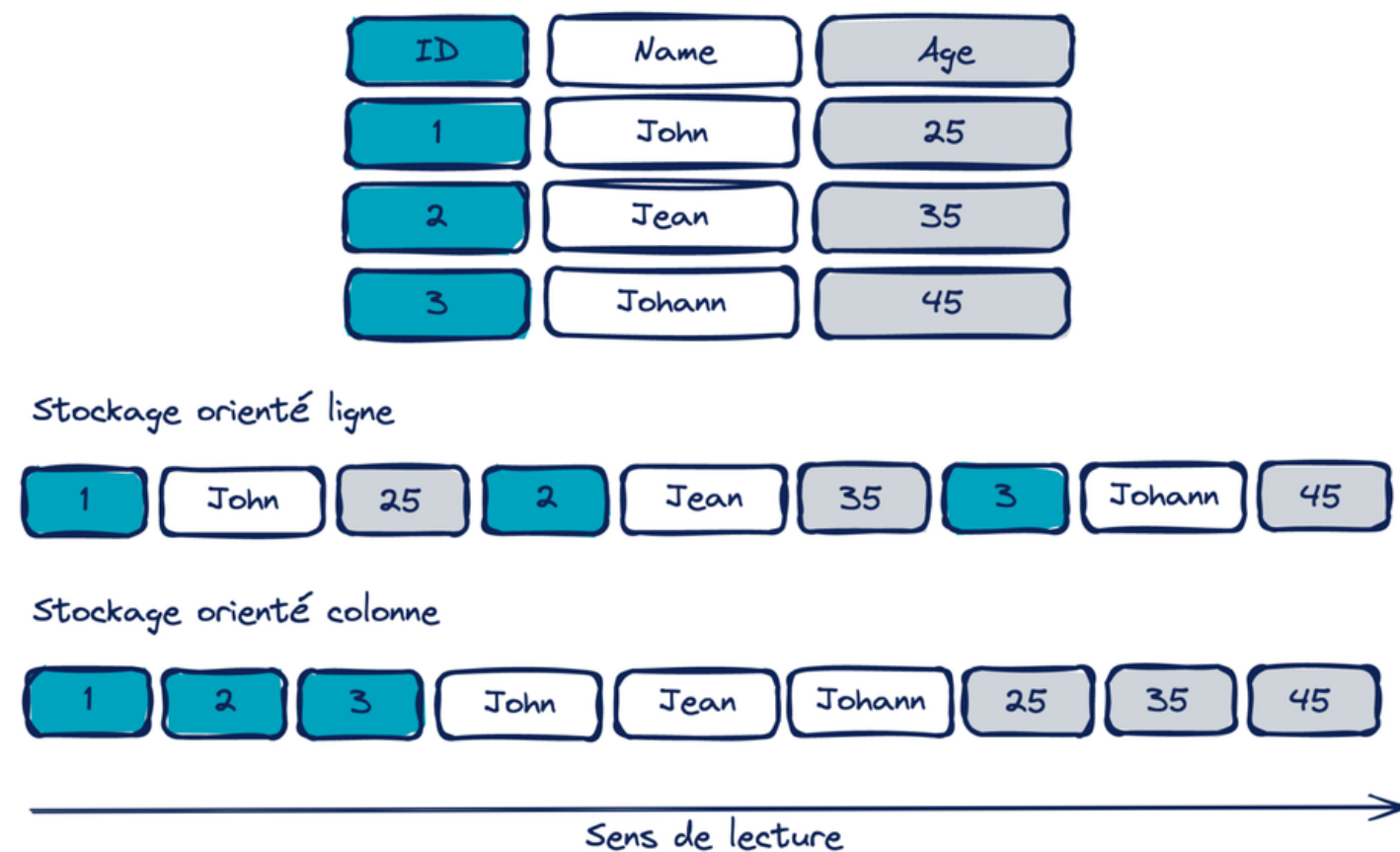
Le **format Parquet** est officiellement un **stockage orienté colonne**, en réalité il s'agit plutôt d'un **format hybride entre un stockage ligne et colonne**. Il est utilisé comme format pour des **données tabulaires**. Les données d'une même colonne sont stockées de façon contigüe.

Ce format est particulièrement souhaitable lorsque l'on effectue des requêtes (transformations) sur un sous-ensemble de colonnes et sur des données volumineuses.

En effet, cela permet de charger en mémoire seulement les données associées aux colonnes nécessaires.

Par ailleurs, le schéma de compression et l'encodage étant propres à chaque colonne selon le typage, cela améliore la lecture / écriture de ces fichiers binaires et leurs tailles sur disque.

Ces atouts en font donc une alternative très intéressante par rapport au format CSV. C'est d'ailleurs le format recommandé par Spark et celui par défaut en écriture.



2ème recommandation : Maximiser le parallélisme de Spark

Spark puise son efficacité dans sa capacité à **traiter plusieurs tâches en parallèle** et avec un passage à l'échelle.

Dès lors, plus on lui facilite le découpage des tâches, plus celles-ci seront effectuées rapidement.

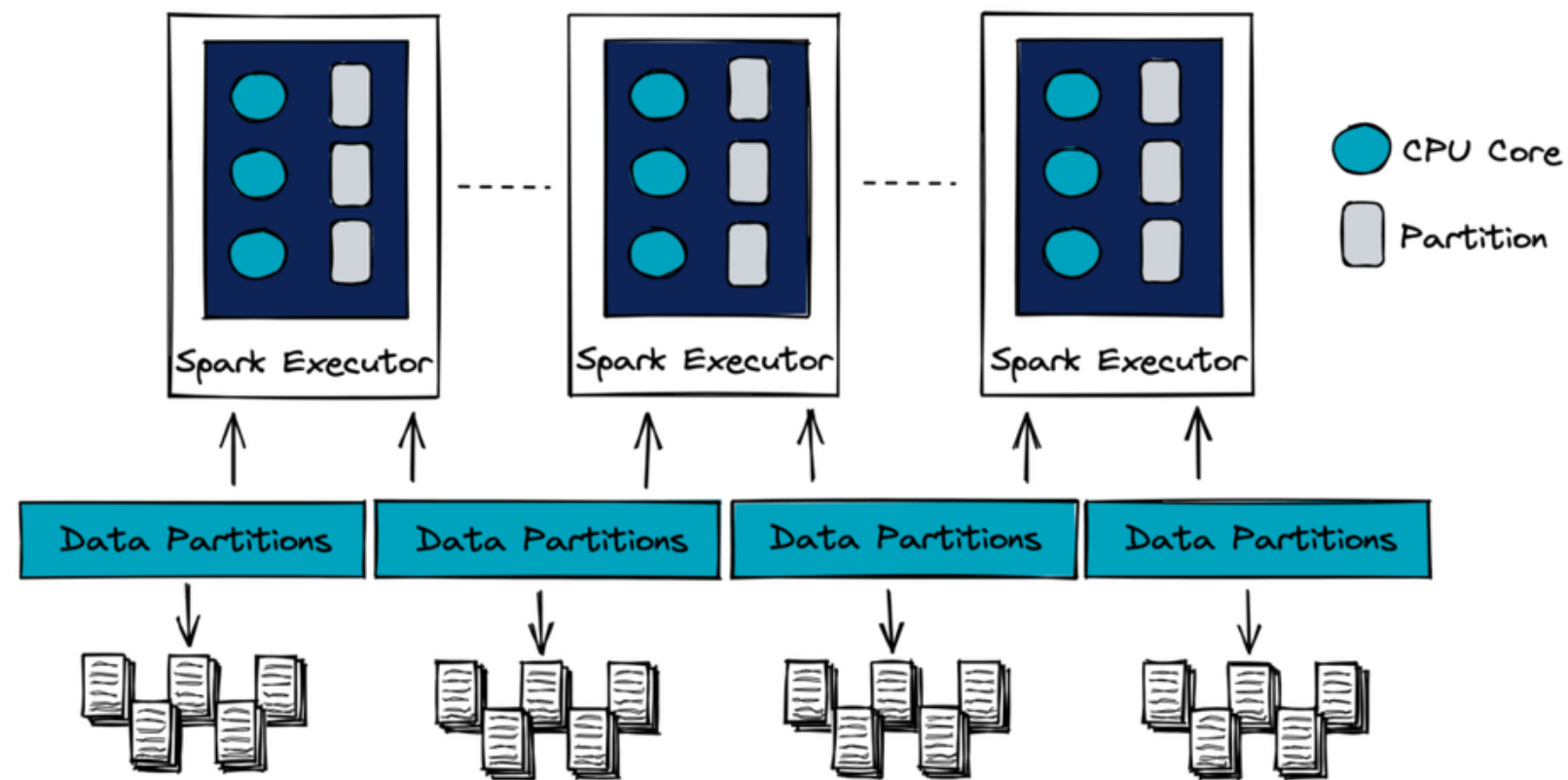
C'est pourquoi **l'optimisation d'un *job Spark* passe par la lecture et le traitement d'autant de données que possible en parallèle**. Et pour atteindre ce but, il est nécessaire de **fractionner un jeu de données en plusieurs partitions**.

Partitionner un *dataset* est une façon d'arranger les données en sous-ensembles configurables et accessibles en lecture par bloc de données contigües sur disque.

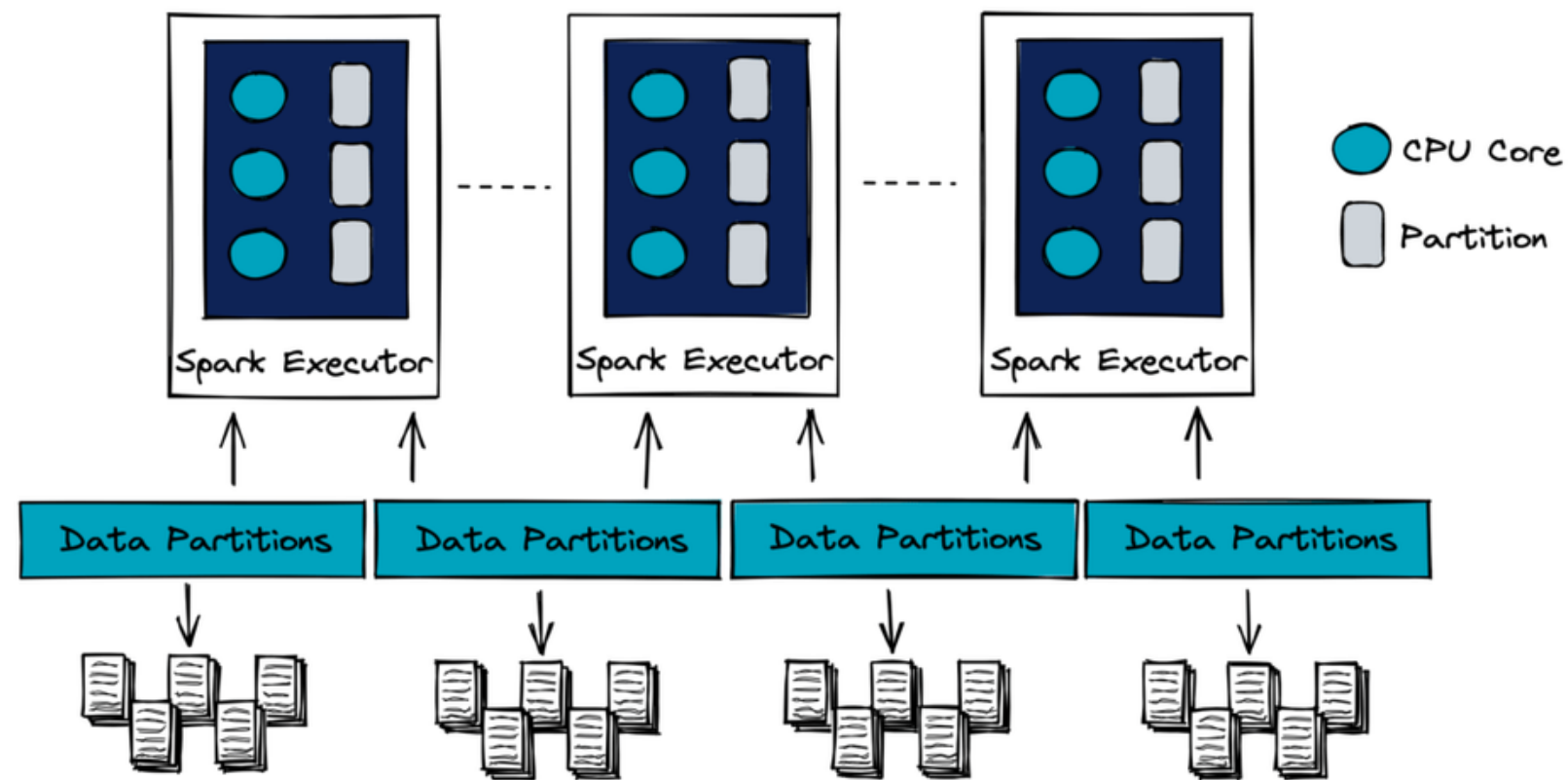
Ces partitions peuvent ainsi être lues et traitées de manière indépendante et en parallèle. **C'est cette indépendance qui permet un traitement massif de données**. Idéalement, *Spark* organise un *thread* par tâche et par cœur de *CPU*. Chaque tâche est relative à une partition distincte.

Ainsi, une première intuition est de configurer un nombre de partitions au moins aussi important que le nombre de coeurs *CPU* à disposition. Tous les coeurs doivent être le plus souvent occupés durant l'exécution du *job Spark*. Si l'un d'entre eux est disponible à un moment, il doit pouvoir traiter une tâche associée à une partition restante.

L'intérêt premier est d'éviter les goulots d'étranglement en découpant les *stages* du *job Spark* en un grand nombre de tâches. Cette fluidité est indispensable dans un *cluster* de calcul distribué. Le schéma suivant illustre ce découpage entre les machines du réseau.



2ème recommandation : Maximiser le parallélisme de Spark



Les partitions peuvent être créées:

- A la lecture des données en configurant le paramètre `spark.sql.files.maxPartitionBytes` (par défaut à 128 MB).
- Directement dans le code de l'application *Spark* à l'aide de l'API *Dataframe*.
- Un exemple:
 - `dataframe.repartition(100)`
 - `dataframe.coalesce(100)`

Cette dernière méthode `coalesce` permet de décroître le nombre de partitions en évitant un shuffle dans le réseau.

3ème recommandation : Se méfier des opérations shuffle



Catégories d'API Spark Data Frame

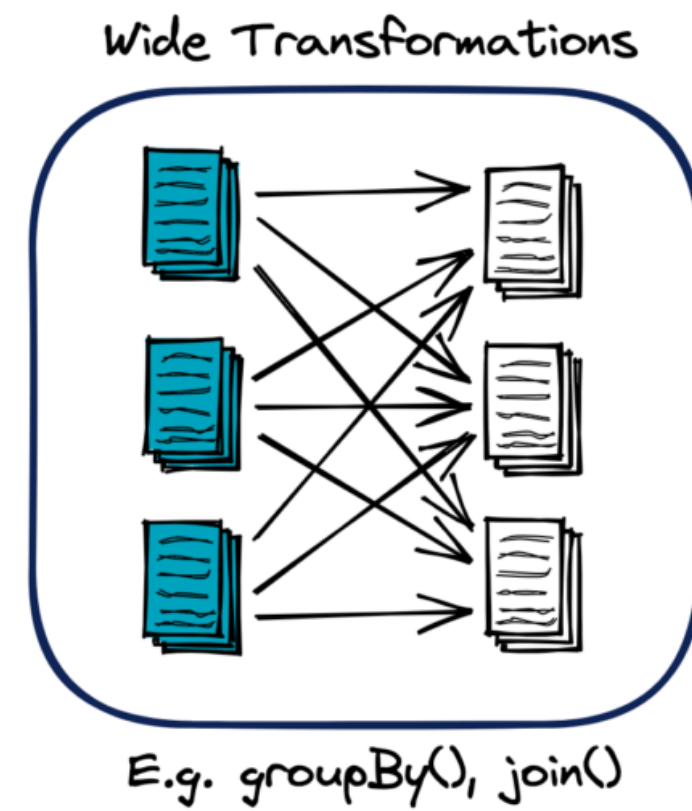
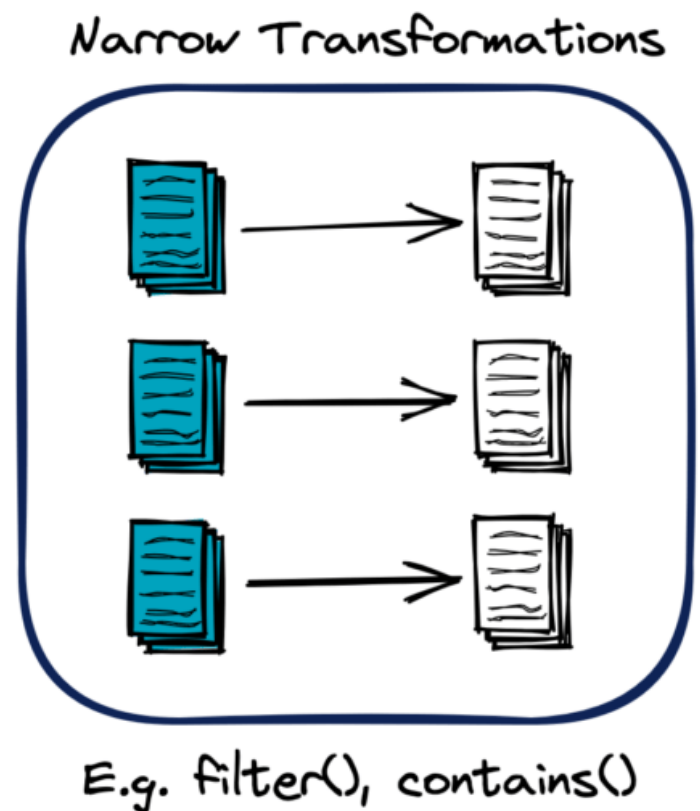
- **Transformations**

- Utilisé pour transformer les données
- Classification supplémentaire
 - Narrow dependency
 - Réalisé en parallèle sur des partitions de données
 - Exemple : `select()`, `filter()`, `With Column()`, `drop()`
 - Wide dependency
 - Effectué après le regroupement des données de plusieurs partitions
 - Exemple : `groupBy()`, `join()`, `cube()`, `rollup()` et `agg()`

- **Actions**

- Utilisé pour déclencher un travail (job)
- Exemple : `read()`, `write()`, `collect()`, `take()` et `count()`

3ème recommandation : Se méfier des opérations shuffle



Parmi les *transformations* opérées par *Spark*, on distingue deux catégories: les transformations:

- narrow
- et wide.

La différence entre ces deux types est le besoin d'une **redistribution des partitions de données dans le réseau entre les nœuds exécutants**. Ce phénomène important est appelé un *shuffle* dans la terminologie *Spark*.

Les transformations wide nécessitant un shuffle sont naturellement les plus coûteuses. Le traitement est plus long en fonction de la taille des données échangées et la latence réseau dans le cluster.

3ème recommandation : Se méfier des opérations shuffle

Il existe un type de partition spécifique dans *Spark* appelé *partition shuffle*. Ces partitions sont créées pendant les *stages* d'un job impliquant un *shuffle*, c'est-à-dire lorsqu'une *transformation* **de type wide** (e.g. *groupBy()*, *join()*) est effectuée notamment.

Le réglage de ces partitions **impacte à la fois le réseau ainsi que les ressources disques en lecture/écriture**.

On peut modifier la valeur de *spark.sql.shuffle.partitions* pour contrôler leur nombre. Par défaut, celui-ci est à 200 ce qui peut-être trop élevé pour certains traitements, et se traduit par un nombre trop important de partitions échangées dans le réseau entre les nœuds exécutants.

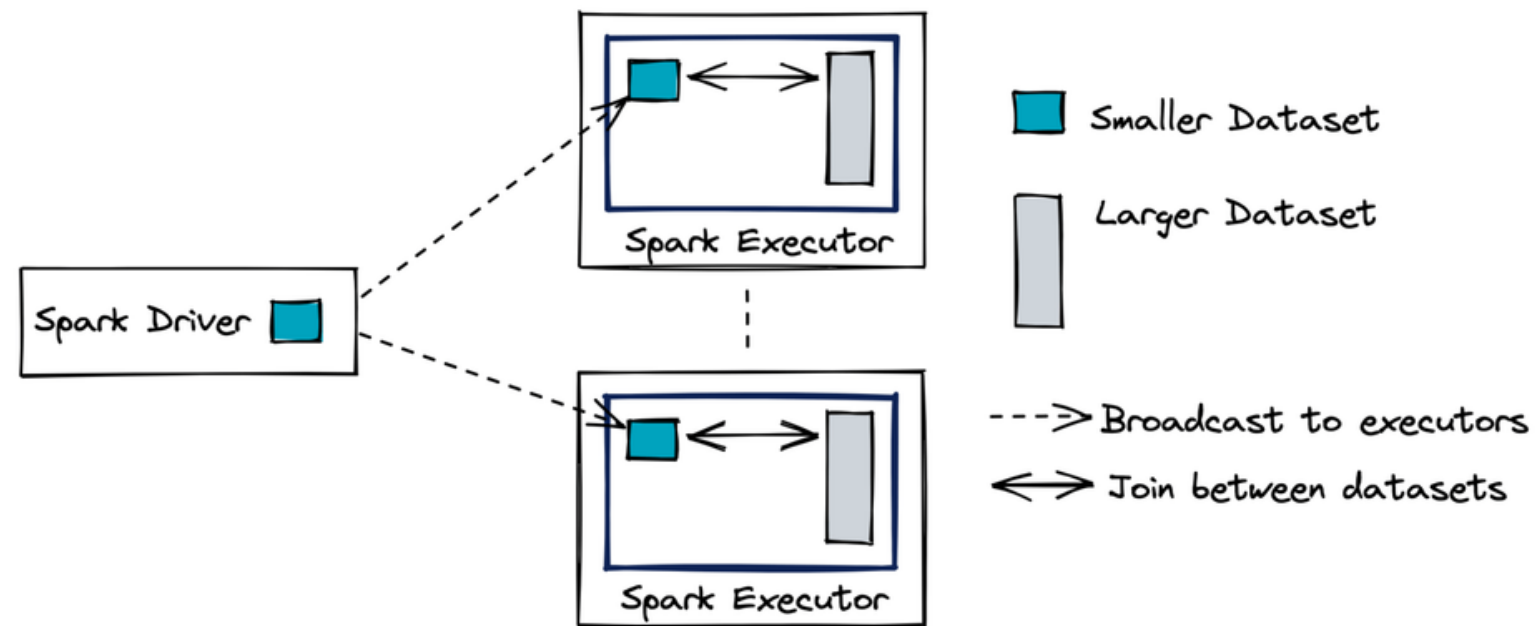
Il convient d'ajuster ce paramètre en fonction de la taille des données. Une intuition peut être de commencer **par une valeur égale au moins égale au nombre de coeurs CPU dans le cluster**.

Spark stocke les résultats intermédiaires d'une opération *shuffle* sur les disques locaux des machines esclaves, la qualité des disques notamment en lecture/écriture a donc son importance. Par exemple, l'usage de disques SSD améliorera sensiblement les performances sur ce type de transformations.

Le tableau ci-dessous décrit les principaux paramètres sur lesquels nous pouvons également influencer.

Configurations	Valeur par défaut, recommandation et description
<i>spark.driver.memory</i>	Par défaut <i>1g</i> (1GB). Cela représente la mémoire allouée au <i>driver Spark</i> pour recevoir les données de la part des noeuds exécutants lors d'opérations comme <i>collect()</i>
<i>spark.shuffle.file.buffer</i>	32 KB par défaut. Il est recommandé de le monter à 1MB. Cela permet à <i>Spark</i> d'effectuer plus de mise en tampon (<i>buffering</i>) avant l'écriture de résultats sur disque
<i>spark.file.transferTo</i>	<i>True</i> par défaut. On peut forcer <i>Spark</i> à utiliser davantage la mémoire tampon (<i>file buffer</i>) avant l'écriture sur disque en réglant ce paramètre à <i>False</i> . L'intérêt est de réduire l'activité I/O.

4ème recommandation : Utiliser la jointure Broadcast Hash Join



Une jointure entre plusieurs tables est une opération courante. En contexte distribué, un volume de données important est échangé dans le réseau entre les nœuds exécutants pour la réaliser.

En fonction de la taille des tables, cet échange entraîne de la latence réseau, ce qui ralentit le traitement.

Spark offre plusieurs stratégies de jointure pour optimiser cette opération.

L'une d'entre elles est particulièrement intéressante si elle peut être choisie, il s'agit du Broadcast Hash Join (BHJ).

Cette technique convient lorsqu'une des tables fusionnées est "suffisamment" petite pour être dupliquée en mémoire sur tous les nœuds exécutants (opération de broadcast). Le schéma ci-dessous illustre le fonctionnement de cette stratégie.

La seconde table est classiquement décomposée en partitions réparties entre les nœuds du cluster. En dupliquant la plus petite table, la jointure ne nécessite ainsi plus d'échange important de données dans le cluster hormis le broadcast de cette table au préalable.

Cette stratégie améliore donc grandement la vitesse d'exécution de la jointure. Le paramètre de configuration Spark à modifier est `spark.sql.autoBroadcastHashJoin`. Par défaut, sa valeur est de 10 MB, c'est-à-dire que cette méthode est choisie si une des deux tables a une taille inférieure à cette grandeur. Si l'on dispose de ressources mémoires suffisantes, il peut-être très intéressant d'augmenter cette valeur ou de la fixer à -1 pour forcer Spark à y recourir.

5ème recommandation :Mettre en cache des résultats intermédiaires

Les différentes options de mise en cache sont décrites dans le tableau ci-dessous:

StorageLevel	Description
<i>MEMORY_ONLY</i>	Les données sont stockées directement en tant qu'objet et seulement en mémoire RAM
<i>MEMORY_AND_DISK</i>	Stockage mémoire objet, mais si la mémoire n'est pas suffisante, le delta sera sérialisé et stocké sur disque.
<i>DISK_ONLY</i>	Les données sont sérialisées et stockées sur disque
<i>MEMORY_AND_DISK_SER</i>	Similaire à l'option <i>MEMORY_AND_DISK</i> , mais la sérialisation est aussi appliquée en RAM (les données sont toujours sérialisées lorsqu'elles sont stockées sur disque)

Pour optimiser ses calculs et gérer les ressources mémoires, *Spark* utilise notamment la *lazy evaluation* et un *DAG* pour décrire un *job*. Cela offre la possibilité de pouvoir **recalculer** rapidement **les étapes en amont d'une action** si besoin, et donc de n'exécuter qu'une partie du *DAG*.

Pour tirer pleinement avantage de cette fonctionnalité, il est alors très judicieux de **stocker des résultats intermédiaires coûteux** si plusieurs opérations les utilisent en aval du *DAG*. En effet, si une action est invoquée, son calcul pourra se baser sur ces résultats intermédiaires et donc rejouer une sous-partie du *DAG*.

Si cette mise en cache peut améliorer la rapidité d'exécution d'un job, elle a néanmoins un coût lors de l'écriture en mémoire et/ou disque de ces résultats.

Il convient donc de tester à différents endroits du pipeline de traitement si le gain de temps total l'emporte. C'est d'autant plus pertinent lorsqu'il y a plusieurs chemins sur le DAG.

On peut mettre en cache une table, par exemple après un chargement depuis le disque ou une transformation, à l'aide de la commande suivante:

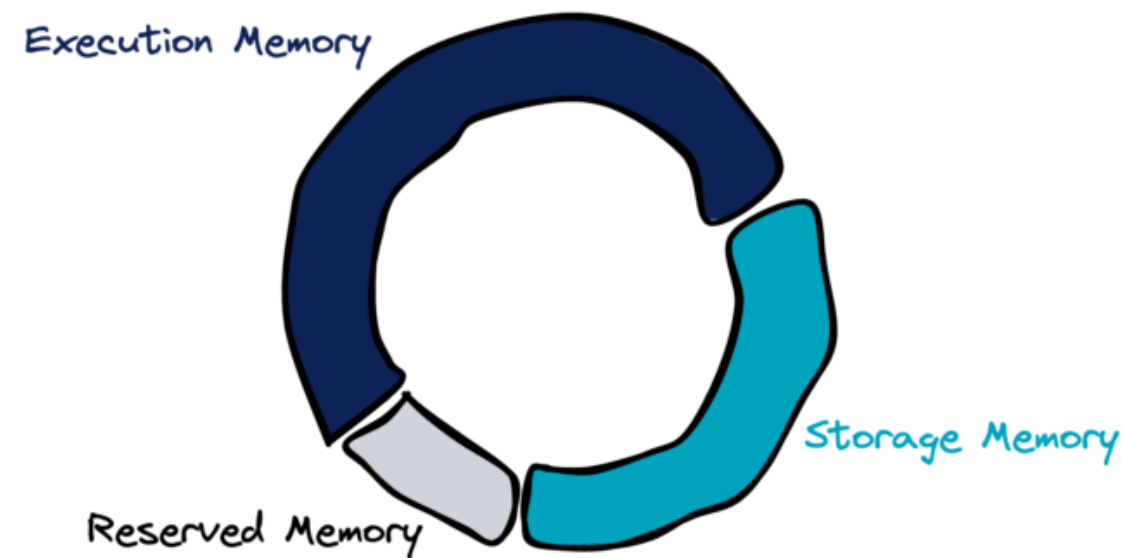
```
dataframe.persist(storageLevel="MEMORY_ONLY")
```

Les différentes options de mise en cache sont décrites dans le tableau ci-dessous:

.

6ème recommandation :Gérer la mémoire des noeuds

La mémoire d'un exécuteur Spark se décompose de la façon suivante:



$$\text{Execution Memory} = \text{spark.memory.fraction} * (\text{spark.executor.memory} - \text{Reserved Memory})$$

Par défaut, le paramètre *spark.memory.fraction* est fixé à 0,6. Cela veut dire que 60% de la mémoire est allouée à l'exécution et 40% pour le stockage, une fois retirée la mémoire réservée.

Celle-ci est de 300 MB par défaut et est utilisée pour prévenir les erreurs de type *out of memory (OOM)*.

Nous pouvons donc influencer sur les deux paramètres suivants:

- *spark.executor.memory*
- *spark.memory.fraction*

Conclusion

Nous avons détaillé une stratégie d'optimisation d'un *job Spark*.

Son principal objectif est de **fournir un cadre pour toute personne désireuse d'optimiser un traitement tout en disposant d'un temps restreint pour le faire.**

L'approche *gloutonne* en six concepts entend aller à l'essentiel pour maximiser les chances de réduire le temps de calcul.

Le schéma qui suit résume le fil conducteur en associant les recommandations proposées à chaque étape.

L'intérêt de chaque étape de ce procédé a été expliqué en lien avec le fonctionnement de Spark.

En effet, même si des recommandations sur les configurations sont proposées, il est essentiel de bien comprendre les rouages internes.

Chaque cas d'usage présente des spécificités propres, et aucune méthode ne peut être universelle.

.

