Programmation large échelle

Partie II - Spark

Villot Yliane
Bah Alhousseine

A) Analyse des paramètres au sein de la commande pour exécuter Spark.

- **MASTER=yarn-cluster** définit le mode de déploiement de Spark et s'exécute sur un cluster YARN.
- **spark-submit** soumet une application Spark au cluster YARN.
- **- - num - executors** spécifie le nombre d'exécuteurs à allouer permettant la parallélisation de l'application.
- **- - driver - memory** spécifie la mémoire alloué au processus driver qui est responsable de la coordination globale des exécuteurs.
- **- - executor - memory** définit la quantité de mémoire allouée aux exécuteurs.

Il est important de modifier ces trois derniers paramètres en fonction de l'entrée sur laquelle on travaille afin d'avoir une répartition des capacités qui soit cohérente. Nous avons utilisé le nombre de tâches effectuées par l'application en fonction de l'entrée pour définir le nombre d'exéxuteurs (par ex : 28 tâches pour clash_big.nljson donc 28 exécuteurs pour une expérience optimale). En ce qui concerne l'utilisation de la mémoire, il a été beaucoup plus difficile de déterminer quelle valeur y attribuer. Nous avons effectué des tests en fonction du champ " Memory Spills " pour essayer de trouver une valeur optimale.

B) Analyse du code Spark et optimisations apportées.

Nous allons produire une analyse de chaque classe Java et détailler les fonctions de celles-ci, en précisant les possibles problèmes d'optimisation.
Dans cette police nous noterons les optimisations faîtes sur le code, et expliquerons en quoi ce sont des optimisations.

I. Classes Battle et Player

Les classes Battle.java et Player.java sont conçues pour contenir uniquement les données membres directement liées aux valeurs extraites du JSON (grâce à Gson dans CRTools.java).

Pour optimiser l'utilisation des ressources, les données inutilisées dans ces classes ont été supprimées, évitant ainsi le transfert de données inutiles sur le réseau.

Voici la liste des membres concernés :

Player = Player\{exp,score}
Battle = Battle\{game,mode,type,warclan}

Nous constatons une baisse de données transitant sur le réseau.

Avant sur clash_big.nljson

Input Size / Records	36.4 MB / 65827	128.1 MB / 231593	128.1 MB / 232811	128.1 MB / 236477	128.1 MB / 241708
Shuffle Write Size / Records	9.5 MB / 65827	33.4 MB / 231593	33.6 MB / 232811	33.8 MB / 236477	35.2 MB / 241708

Après sur clash_big.nljson

Input Size / Records	36.4 MB / 65827	128.1 MB / 231593	128.1 MB / 232811	128.1 MB / 236477	128.1 MB / 241708
Shuffle Write Size / Records	8.6 MB / 65827	30.2 MB / 231593	30.4 MB / 232811	30.5 MB / 236477	31.3 MB / 241708

La différence peut sembler infime mais nous avons estimé que c'est le genre d'optimisations qui sont importantes quand on passe à l'échelle.

II. Classe Deck

La classe Deck.java est utilisée pour implémenter la logique des decks et en extraire des statistiques sur leurs performances, notamment les victoires avec une limite du nombre de joueurs posée à 10.
Elle permet également d'effectuer une fusion (**merge(Deck d)**) entre un deck donné et le deck courant permettant d'additionner les victoires parties jouées et les forces, mise à jour des ligues et trophées (en gardant la valeur max) ainsi que la combinaison des statistiques de cartes (evo et tour).

Cependant nous avons remarqué une utilisation de **TreeSet** qui est une structure de données extrêmement coûteuse.

Ces TreeSet nous semblent injustifiés car aucune utilisation de l'ordre de trie, c'est pour cette raison que nous l'avons remplacé par des Hashset avec une mémoire allouée à 10, permettant ainsi une allocation

restrictive avec une structure de données moins coûteuse. Cependant nous n'avons pas réussi à mesurer l'impact de cette optimisation.

III. Classe CRTools

CRTools contient une fonction **getDistinctRawBattles()** permettant de faire un data cleaning sur les données. Nous allons analyser point par point les étapes de cette fonction.

- **1 :** Dans un premier temps, elle va créer un objet **JavaRDD** à partir d'une source nljson et effectuer un premier tri via la fonction **filter()** pour ignorer les lignes vides. **JavaRDD** est un ensemble de données partitionnées sur les différents nœuds du cluster et **filter()** est une fonction de transformation effectuée sur ces partitions et évaluer sur chaque ligne si elles respectent une certaine condition (ici **!x.isEmpty()**)

- **2 :** Un autre objet **JavaRDD** est créé par un **mapToPair()** utilisé sur les partition du rdd. La fonction **mapToPair()** est une fonction transformative qui va changer les éléments des partitions en clé-valeur. Ici nous convertissons d'abord les lignes json en objets **Battle** afin de produire une paire de clé-valeur **<String,Battle>**. Les données **{date, round, p1.utag, p2.utag}** sont utilisées pour faire les clés.

  Une chaîne de transformation est effectuée sur ce nouvel objet via les fonctions **distinct()** et **values()**.
  - La fonction **distinct()** permet de supprimer les doublons, c'est une fonction coûteuse car elle doit faire un shuffle et comparer toutes les paires entre elles.
  - La fonction **values()** permet de ne garder que la partie valeur des paires c-a-d les Battle. C'est pour cette raison que notre objet est est typé **JavaRDD** et non en **JavaPairRDD**

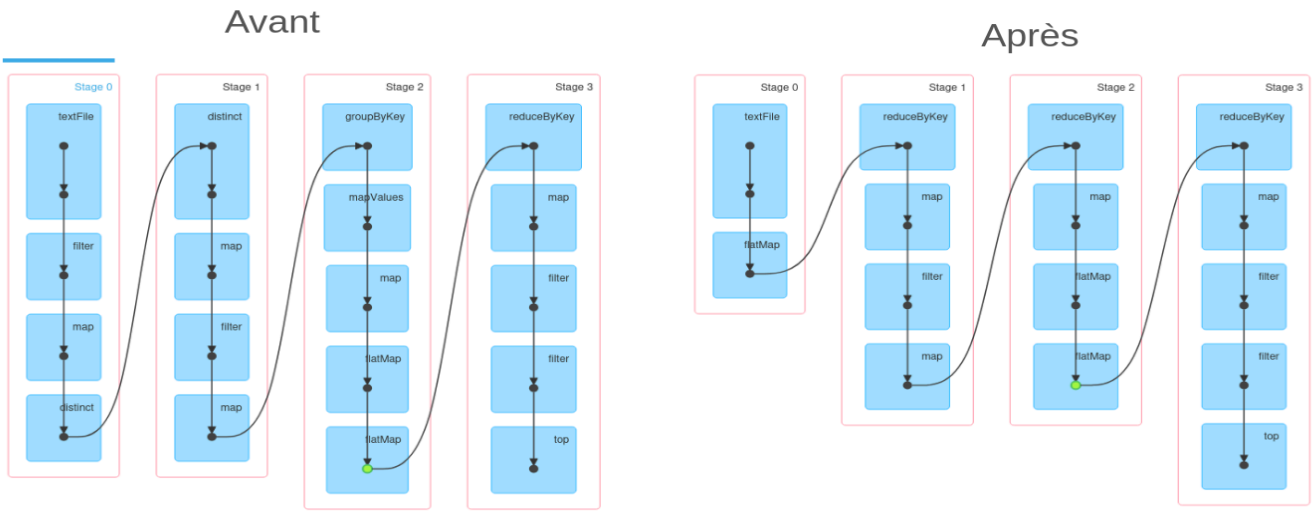
- **3 :** Ensuite cet objet est re-défini par lui-même auquel on applique la fonction **fitler()** afin de ne garder que les parties qui ont commencé il y a moins de 9 semaines.

Dans notre code optimisé nous avons fusionné ces trois premières étapes. Le fichier json est directement analysé dans la fonction mapToPair().

De plus nous avons remarqué dans DeckGenerator.java des opérations (comme la vérification de 8 cartes dans un deck, la force des player, etc) qui pourraient être faîtes directement dans getDistinctRawBattle(). Nous avons donc décidé de regrouper tous ces tests dans cette fonction, permettant de limiter le nombre de map et un trie en une seule fois.

Nous avons d'abord penser que l'utilisation de distinct() devait être remplacée par une utilisation de reduceByKey() car nous pensions qu'à l'instar de distinct() qu'elle combinait d'abords les paires localement avant de les regrouper globalement, permettant ainsi un allégement du transit sur le réseau. Cependant, après vérification, nous avons vu que le code source de distinct() est un appel à reducebyKey(). Il n'y a donc pas de différence de performance.

Nous avons également ajouté la fonction filter() la chaîne de transformation évitant ainsi la re-définition de notre objet JavaRDD. Voir Stage 0 et Stage 1 dans le schéma ci-dessous.



Nous avons constaté qu'une très petite amélioration au niveau du temps d'exécution mais tellement infime qu'elle nous semble négligeable. Nous ne savons pas si notre optimisation est si conséquente que cela.

- **4 :** Par la suite, un objet **JavaPairRDD** est créé à partir d'un **mapToPair()** appliqué sur le **JavaRDD** afin de créer une paire de clé-valeur **<String, Battle>**. Les données **{p1.utag, p2.utag, p1.elixir, p2.elixir}** sont utilisées pour fabriquer la clé. Les paires sont regroupées via la fonction **groupByKey()** qui est une fonction très coûteuse car elle transfère les paires similaires sur un même nœud, ce qui entraîne un transit élevé sur le réseau.

- **5 :** Pour finir les battles sont comparés et regroupés en fonction de la date avec une tolérance de 10 secondes afin de supprimer les doublons. Nous avons remarqué un appel excessif à Instant.parse() qui peut être une opération coûteuse.

Nous avons ajouté un membre parsedDate dans la classe Battle, qui sera directement utilisée pour les comparaisons de date afin d'éviter l'appel excessif de Instant.parse().

IV. Classe DeckGenerator

DeckGenerator contient de nombreuses fonctions que nous allons toutes décortiquer.

IV. a) factorial(int num)

C'est une fonction récursive qui permet de calculer la factorielle d'un entier. L'utilisation de la récursion n'est pas optimisée car elle utilise plus de mémoire dû aux multiples appels de la fonction s'ajoutant à la stack. Cette fonction est appelée dans la fonction **combination(n,k)**

IV. b) combination(int n, int k)

Cette fonction produit une combinaison entre n éléments et k éléments, implémentant la formule C(n,k) = n! / (k! * (n-k)!). Cependant, cette fonction fait appel de multiples fois à une méthode récursive sans même tester les cas limites. Cette fonction ne semble pas être utilisée dans le code.

La fonction n'est jamais appelée dans le code (c'est donc aussi le cas pour factorial(num)).
Nous avons donc enlevé ces fonctions du code.
Même si nous ne les avons pas modifiées, nous avons remarqué qu'elles auraient pu l'être en mettant en place un calcul itératif plutôt que récursif permettant de meilleures performances dû à une moindre utilisation de la mémoire via un empilement d'appels de fonctions sur une stack.

IV. c) generateCombinations(int n, int k)

Cette fonction permet de générer une liste de liste de toutes les combinaisons possibles de taille k à partir d'un ensemble d'entiers allant de 0 à n−1n. Elle fait appel à la fonction **generate()** pour générer les combinaisons.

IV. d) generate(ArrayList<Integer> current, int start, ArrayList<Integer> elements, int k, ArrayList<ArrayList<Integer>> result)

La fonction **generate()**est une fonction récursive qui va générer les combinaisons. Elle se comporte un peu comme le parcours d'un arbre en utilisant du backtracking. Il serait possible d'optimiser cette fonction en y ajoutant des tests aux limites afin d'éviter d'explorer des possibilités inutiles et en ayant un algorithme plus intelligent.

IV. e) treatCombination(Battle x, ArrayList<String> tmp1, ArrayList<String> tmp2,ArrayList<Tuple2<String, Deck>> res, ArrayList<Integer> cmb)

Cette fonction permet de générer les decks à partir des combinaisons et des cartes récoltées dans les battles. Dans un premier temps elle génère les decks sous forme d'une liste de chaîne de caractères tmp1 et tmp2. Ensuite, elle va créer deux objets Deck en vérifiant s' ils sont des decks avec évolution ou non et va ajouter ces decks dans une liste de **Tuple2<String, Deck>** représentant une paire de clé-valeur pour un deck, avec l'id du deck pour en guise de clé.

Nous avons constaté une duplication de code et de multiples accès à de même calculs dans cette fonction, nous avons donc corrigé cela. De plus, nous avons remplacé les StringBuffer par des StringBuilder pour stocker l'id du deck avec une taille pré-alloué de 16.

IV. f) main()

Main est le point d'entrée de notre application Spark. Elle contient plusieurs étapes que nous allons expliciter.

- **1 :** Dans un premier temps une boucle for est faîte sur une liste **CARDGRAMS** afin d'ajouter les combinaisons possibles dans une liste **combs.**

- **2 :** Ensuite Spark est initialisé et on appelle la fonction **getDistinctRawBattle()** vu précédemment, qui va cleaner le dataset et instancier un **JAVARDD** appelé **clean**.

- **3 :** Un **flatMapToPair()** est appliqué sur **clean** instanciant ainsi un **JavaPairRDD<String, Deck>** appelé **rddecks**. On crée une liste de chaîne de caractères tmp1 et tmp2 en prenant la valeur deck de player1 et player2 en partitionnant la chaîne de caractères par 2 : deux caractères représentant une carte. La combinaisons de tous les decks est créée via une boucle for-each sur **combs** appelant **treatCombination()** à chaque itération, utilisant les chaînes divisées.

- **4 :** Un **JavaRDD<Deck>** est instancié, faisant merge les deck regroupés via un **reducebykey()** et ne gardant que les decks ayant au moins 10 joueurs et 80 parties. Cela permet de créer les statistiques des decks avec comme condition de jouer au moins 80 parties (car on pourrait avoir 100% de winrate avec une seule partie gagnée)

- **5 :** Une liste de **JavaRDD<Deck>** appelée **statistics** qui va répertorier des decks qui ont une tailles correspondant à une un nombre de combinaison du n-gram (ce sont les subdeck, ex : les combinaisons de 2 cartes avant de calculer les combinaisons de 8 cartes)

- **6 :** Finalement, les résultats sont écrits sur un fichier de sortie, en priorisant les decks avec un meilleur winrate afin qu'ils soient affichés en premier. **Une boucle for avec une ligne commentée est présente dans le code, nous l'avons donc enlever car elle était inutile.**

Dans cette partie du code, la première optimisation a été de rajouter des cache() aux rdd qui sont réutilisés de façon à conserver leur mémoire en RAM et en avoir un accès beaucoup plus rapide.
Avec cette optimisation nous avons constaté une amélioration du temps d'exécution même si il est compliqué de mesurer le temps d'exécution car en fonction du contexte il semblerait que cela peut prendre plus ou moins de temps (par exemple quand un grand nombre de personnes travaillent sur le cluster)
Nous pensons que la génération des combinaisons pourrait être largement optimisée, notamment par l'intermédiaire d'un cache qui garderait en mémoire les valeurs déjà calculées, cependant nous n'avons pas réussi à faire une implémentation avec des résultats concluants.