Projet : Data Cleaning d'un dataset Clash Royal

Villot Yliane, Alhousseine bah

1. Analyse du dataset.

Dans un premier temps il a fallu analyser la structure des données contenu dans le dossier `/user/auber/data_ple/clash2024.nljson` afin de comprendre quels sont les potentiels points critiques de notre dataset.

Voici la structure de donnée récupérée :


```
{
  "date":"2024-09-23T16:04:46Z",
  "game":"pathOfLegend",
  "mode":"Ranked1v1_NewArena",
  "round":0,
  "type":"pathOfLegend",
  "winner":1,
  "players":[
    {
      "utag":"#U82CQ9C8Q",
      "ctag":"#QYPVC8RG",
      "trophies":5498,
      "exp":32,
      "league":1,
      "bestleague":2,
      "deck":"00010512213c5b5c",
      "evo":"",
      "tower":"6e",
      "strength":10.75,
      "crown":0,
      "elixir":12.41,
      "touch":1,
      "score":0
    },
    {
      "utag":"#8QRCGQJC",
      "trophies":7109,
      "exp":43,
      "league":1,
      "bestleague":5,
      "deck":"080c111416235b66",
      "evo":"08",
      "tower":"70",
      "strength":11.1875,
      "crown":1,
      "elixir":2.74,
      "touch":1,
      "score":0
    }
  ]
}
```


2. Implémentation du data cleaning.

La classe `Battle.java` et `Player.java` proviennent du code fait par le professeur. Ces classes contiennent des `@JSONProperty` permettant de récupérer les données d'un json et de les associer à des membres de ces classes.

Le data cleaning que l'on cherche à implémenter consiste à éliminer les doublons, c'est-à-dire les parties contenant les mêmes données.

Voici les points critiques à observer :

- Vérifier la justesse de certaines données importantes
- Parties miroirs suivant le point de vue (Joueur A dans la partie 1 = Joueur B dans la partie 2)
- Légère différence dans l'enregistrement d'une même partie au niveau de la date. On suppose une tolérance de 10 secondes sur la date pour une partie ayant les mêmes données.

Le data cleaning sera fait avec un map-reduce.

Une fonction verifyData() a été ajoutée dans `Battle.java` qui vérifie les données importantes (si il y a bien 2 joueurs, 8 cartes par decks, etc..)
Cette fonction est appelée dans le mapper de façon à vérifier toutes les parties du dataset. Le mapper enverra ensuite une association clé-valeur `<Gamekey,Text>` au reducer.

La classe `GameKey.java` impélemente `WritableComparable<GameKey>`. Cette classe permet de créer une clé personnalisée en définissant nos propres règles de regroupement lors du passage du mapper au reducer, grâce à la redéfinition de la méthode compareTo.

compareTo(Gamekey k) va comparer les gameKey entre elles et considérer qu'elles appartiennent au même groupe en fonction de certaines conditions.

La classe `GameKey.java` contient une fonction toString() qui génère une chaîne de caractères incluant des données permettant d'identifier une partie, telles que le jeu, le mode, le type, le round, ainsi que les joueurs player1 et player2, triés par ordre lexicographique pour éviter les problèmes de point de vue.

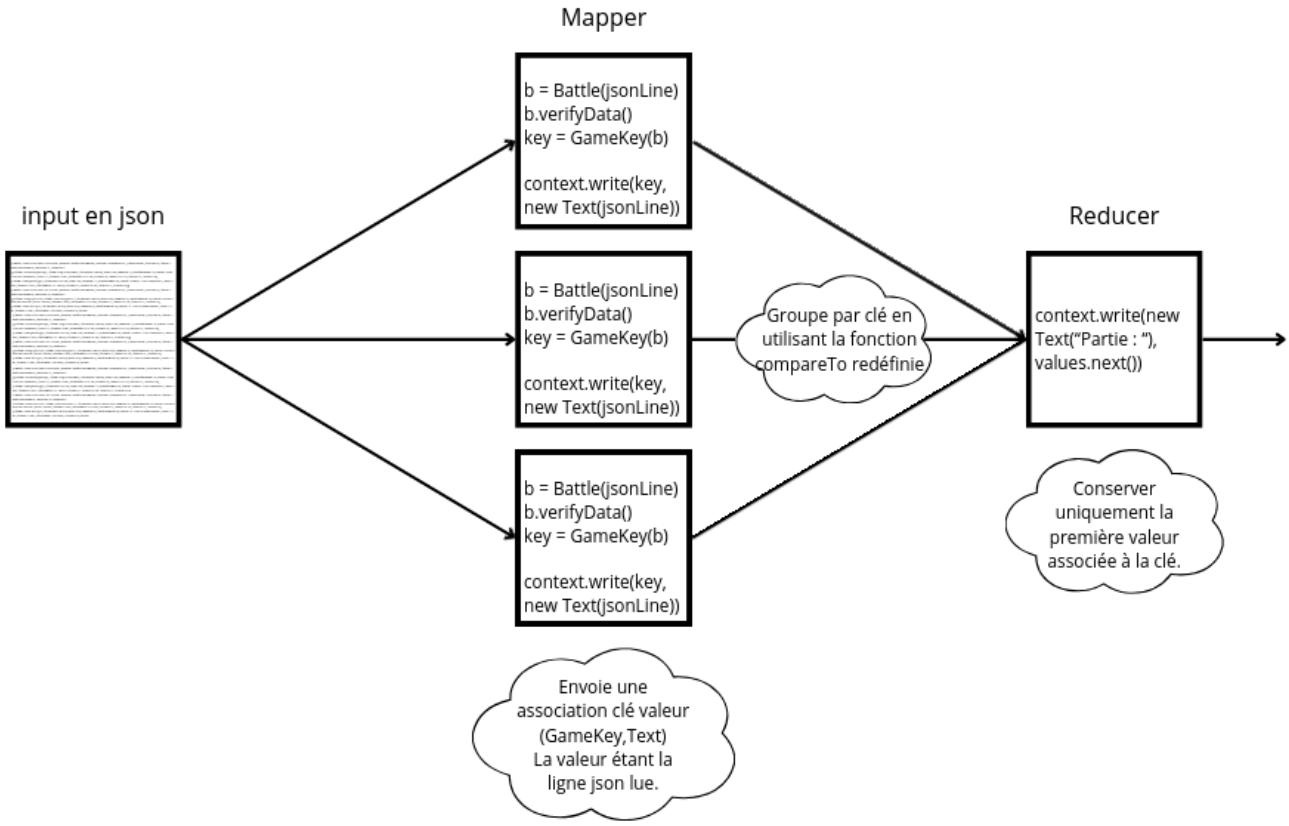
La méthode compareTo(GameKey k) compare d'abord les deux chaînes générées par toString() pour vérifier que les données sont identiques. Si elles le sont, elle compare ensuite les dates, converties en secondes. Si deux parties ont les mêmes données et que l'écart entre leurs dates est inférieur ou égal à 10 secondes, elles sont considérées comme étant la même partie.

Avec toute cette implémentation, les parties considérées comme des doublons seront directement regroupées sous la même key dans le reducer.
Ainsi nous n'avons plus qu'à écrire une seule de ces parties dans le fichier de sortie. Le reducer renvoie donc un `<Text,values.next()>`

Le mapper a donc pour entrée-sortie :
`<LongWrittable, Text, Gamekey, Text>`

Le Reducer a donc pour entrée-sortie :
`<GameKey, Text, Text, Text>`

Schéma du map reduce mis en place :



3. Analyse du résultat.

Entrées Map : 2,781,989 enregistrements ont été lus par les mappeurs.
Sorties Map : 2,765,874 enregistrements ont été émis par les mappeurs.
Perte de 16,115 enregistrements (2,781,989 - 2,765,874) due aux filtrages.
Entrées Reduce : 2,381,199 enregistrements ont été reçus par les réducteurs.
Groupement Reduce : 2,184,191 groupes ont été créés à partir des enregistrements d'entrée.
Sorties Reduce : 2,184,191 enregistrements ont été émis par les réducteurs après traitement.

On a bien Groupement Reduce = Sortie Reduce, ce qui est logique, car on voulait garder qu'une seule partie par regroupement de parties.

On remarque que 12 mappers sont lancés.
En ce qui concerne le choix du nombre du reducer nous devons prendre en compte le nombre d'octets qui sortent du mapper, à savoir :

Une ligne de json ~= 523 octets.
On a 2,184,191 parties en sortie

523 * 2184191 = 1142331893 octets
Soit 1142Mo

1142 / 128 ~= 9.

Il faudrait donc en théorie environ 9 reducer pour avoir de bonnes performances.