SGIA : Stratégie Intelligente de Groupement pour Améliorer le Traitement des Requêtes OLAP en MapReduce

Yassine Ramdane*, Nadia Kabachi**
Omar Boussaid*, Fadila Bentayeb *

*Université de Lyon, Lyon 2, ERIC EA 3083, 5, avenue Pierre Mendês 69676 Bron-France,
{Yassine.Ramdane, Omar.Boussaid, Fadila.Bentayeb}@univ-lyon2.fr
**Université de Lyon, université Claude Bernard Lyon 1, ERIC EA 3083, 43 boulevard du 11 novembre 1918, 69100, Villeurbanne-France
Nadia.Kabachi@univ-lyon1.fr

Résumé. L'amélioration des performances d'une requête OLAP dans un système distribué tel que Hadoop ou Spark est une tâche ardue. Une requête OLAP est composée de plusieurs opérations, tels que le filtrage, la jointure et le *Group By*. Chaque opération peut être exécutée dans la phase *map* ou la phase *reduce* avec un ou plusieurs cycles *MapReduce*. Étant donné qu'il est possible de collecter au préalable quelques connaissances sur le système distribué, certaines opérations, comme la jointure en étoile et le filtrage, peuvent être optimisées en utilisant une technique statique de partitionnement. Cependant, l'optimisation du *Group By* nécessite généralement l'utilisation d'une technique dynamique de partitionnement et de distribution qui permet d'équilibrer à la volée les charges des *reducers*, car nous ne pouvons pas collecter les informations pertinentes qui aident le système à établir le bon schéma qu'au moment de l'exécution de la requête. Dans cet article, nous proposons une méthode intelligente, appelée SGIA, permettant d'équilibrer les données d'entrées des *reducers*. Nous avons utilisé un système multi-agents qui permet d'équilibrer à la volée les charges des *reducers*. Les expérimentations révèlent que notre approche est plus performante que celles existantes en termes de temps d'exécution des requêtes.

1 Introduction

Les requêtes OLAP sont généralement des requêtes coûteuses qui nécessitent beaucoup de temps pour être exécutées sur des Entrepôts de Données Distribuées (EDD) massives. Dans les EDD massives, l'amélioration du traitement d'une requête OLAP est une tâche ardue. Une requête OLAP est composée de plusieurs clauses et prend généralement la forme "*Select... Function ()... From...Where... Join-Predicates ...Filters... GROUP BY...*". Chaque clause peut être exécutée dans la phase *map* ou dans la phase *reduce*, et chaque opération peut être exécutée avec un ou plusieurs cycles MapReduce ou stages de Spark, avec une quantité considérable de données transférée entre les nœuds. La jointure en étoile n'est pas la seule opération coûteuse

dans une requête OLAP. L'opération Group By peut elle aussi augmenter le coût de communication durant la phase de *shuffle* et peut perturber le traitement parallèle, suite à la distribution non homogène des données sur les *reducers*, en particulier avec les opérateurs *Rollup* et *Cube*. Nous avons proposé dans (Ramdane et al., 2019) une nouvelle conception physique d'un EDD massives qui aide le système distribué à exécuter la jointure en étoile d'une requête OLAP avec un seule stage de Spark, sans la phase de *shuffle*. Cet article est un complément de l'approche précédente, où nous proposons une méthode intelligente pour équilibrer à la volé les charges des *reducers*, ce qui améliore le temps d'exécutions de l'opération *Group By*.

La méthode actuelle pour améliorer le traitement de *Group By* consiste à équilibrer à l'avance les Résultats Intermédiaires (RI) et éviter d'obtenir une distribution non homogène des charges des *reducers*. Notez que nous pouvons équilibrage les données d'entrées d'HDFS (*split inputs*) à l'aide d'une technique d'équilibrage statique (Ramdane et al., 2019), puisque des informations pertinentes qui peuvent aider le système à établir un équilibrage quasi-optimal, peuvent être collecter à l'avance. Cependant, pour créer un bon schéma de partitionnement des Données d'Entrées des *Reducers* (DERs), le système doit acquérir certaines connaissances à la volée. De nombreuses études ont abordé ce verrou scientifique, nous pouvons les diviser en deux catégories. Dans la première catégorie (Gufler et al., 2012; Xu et al., 2012; Ramakrishnan et al., 2012; Ibrahim et al., 2013; Chen et al., 2014; Gao et al., 2017; Tang et al., 2018), les méthodes proposées essayent d'équilibrer les DERs à l'avance en modifiant le partitionnement de hachage utilisé par certains systèmes tels que Hadoop (Dean et Ghemawat, 2008) et Spark (Zaharia et al., 2010). Dans la deuxième catégorie (Kwon et al., 2012; Baert et al., 2016), les méthodes proposées se reposent sur des heuristiques pour équilibrer les tâches des *reducers* au cours de l'exécution des fonctions d'agrégat, c.-à-d. après la phase de *shuffle*. Notez que les algorithmes utilisés doivent être exécutés rapidement et ne doivent pas avoir d'impact sur le temps d'exécution de l'application elle-même.

Certains travaux de la première catégorie (Xu et al., 2012; Ramakrishnan et al., 2012; Tang et al., 2018) utilisent la technique d'échantillonnage pour obtenir un équilibrage plus ou moins acceptable des DERs. Bien que ces méthodes puissent établir un bon schéma de partitionnement, cependant l'utilisation de l'échantillonnage dans un EDD massives peut donner des résultats erronés, puisque certaines clés dans la phase *map* ne sont pas prises en compte dans la distribution. En outre, d'autres travaux (Gufler et al., 2012; Ibrahim et al., 2013) ont modifié le mécanisme par défaut de MapReduce, de sorte que le système doit attendre que toutes ou la majorité des *mappers* soient terminées afin de recueillir les informations sur la taille des fragments, puis il lance la phase de *reduce*. Notez que le changement de la méthode par défaut de *MapReduce* peut surcharger le tampon de mémoire intermédiaire (c.-à-d. RI) et le système crash. De plus, si les tâches des *mappers* sont trop déséquilibrées, le traitement parallèle fonctionne mal et par conséquence le temps d'exécution de l'application augmente. Les approches de la deuxième catégorie risquent également de dégrader les performances de l'application, puisque le système doit suspendre les tâches des *reducers* pour déclencher l'algorithme d'équilibrage. Notre approche est similaire à celle de (Gufler et al., 2012), en utilisant un système multi-agents (SMA). Cependant, le principale avantage de notre contribution est que nous conservons le mécanisme par défaut de MapReduce.

Dans de nombreux cas, la charge de travail des requêtes OLAP est stable pendant une période donnée et peut être légèrement mise à jour si l'analyse des besoins change, c.-à-d. que l'ensemble des colonnes utilisées dans les clauses *Where* et *Group By* restent stables avec le

changement des filtres [1]. Cette hypothèse a été observée de manière empirique dans certains travaux (Agarwal et al., 2012, 2013). Cela implique que nous avons une forte probabilité d'obtenir les mêmes fragments (un fragment est le sous-ensemble de tous les tuples ayant la même clé (Heintz et al., 2016)) produits par les *mappers* lors de l'exécution d'une requête sur un *cluster* de nœuds. Par conséquent, les fréquences d'occurrence de ces fragments restent également stables pendant une période donnée et peuvent être changer en cas de nouvelle charge de travail ou lorsque nous mettons à jour l'EDD. Pendant ce temps, dans certains Gestionnaires des Ressources (GRs) des systèmes distribués, tels que YARN, Mesos et Spark, l'exécution d'un ensemble de requêtes OLAP sur un *cluster* peut être effectuée simultanément ou séquentiellement. A savoir, le GR d'un système distribué peut partager, de manière dynamique ou statique, les ressources (ex. *executors* ou *workers*) d'un *cluster* pour exécuter plusieurs requêtes dont le niveau de complexité est différent. Cela signifie que le GR est en mesure d'affecter les ressources nécessaires pour chaque requête (chaque requête a un minimum et un maximum de ressources pour être exécutée). En d'autres termes, les fragments produits par une requête avant l'opération *Group By* seraient identiques dans chaque exécution (sauf lorsque nous mettons à jour l'EDD), cependant, pour chaque exécution d'une requête, le nombre des *mappers* et des *reducers* (ex. nombre des cœurs et les partitions des *reducers*) varie de temps en temps.

En exploitant ces spécifications, nous pouvons proposer une méthode intelligente permet d'équilibrer à la volée les DERs. SGIA peut tirer profit des précédentes exécutions des requêtes, de sorte que le système puisse réagir en ligne pour établir un bon schéma de partitionnement des DERs. Notre idée est la suivante : nous ajoutons à l'optimiseur d'un système distribué un SMA capable de distribuer les fragments sur les *reducers*, tel que les agents peuvent acquérir des connaissances lors de l'interaction avec l'environnement, et exploitent l'historique d'exécution des requêtes pour équilibrer les charges des *reducers*.

Le reste de cet article est structuré comme suit. Dans la Section 2, nous expliquons notre problème à travers un exemple. Dans la Section 3, nous présentons les travaux associés qui traitent le problème d'équilibrage des charges des *reducers*. Nous détaillons notre approche SGIA dans la Section 4. Nous présentons nos expérimentations dans la Section 5 et nous concluons dans la Section 6.

2 Explication du Problème et Motivation

L'exemple suivant explique notre problème. Supposons une requête OLAP Q, extraite du banc d'essai TPC-DS, tel que :

```
SELECT d_year, i_brand_id, SUM(ss_sales_price)
FROM  date_dim, item, store_sales
WHERE date_dim.d_date_sk = store_sales.ss_sold_date_sk
AND store_sales.ss_item_sk = item.i_item_sk
AND i_manufact_id = 128
GROUP BY d_year, i_brand_id;
```

Le plan d'exécution de Q avec Spark SQL est le suivant : pour chaque table, le système analyse les blocs HDFS, récupère les attributs sollicités par la requête, exécute les filtres (ex. i_manufact_id=128), puis exécute la jointure en étoile (ex. en utilisant l'approche de Ramdane et al. (2019)), et enfin *Group By* et la fonction *SUM*(ss_sales_price). La figure 1 montre une partie des étapes d'exécution de cette requête après l'exécution de la jointure en étoile.

1. Évidemment, la variation dans les filtres de la requête influe sur les fragments produits par les *mappers*

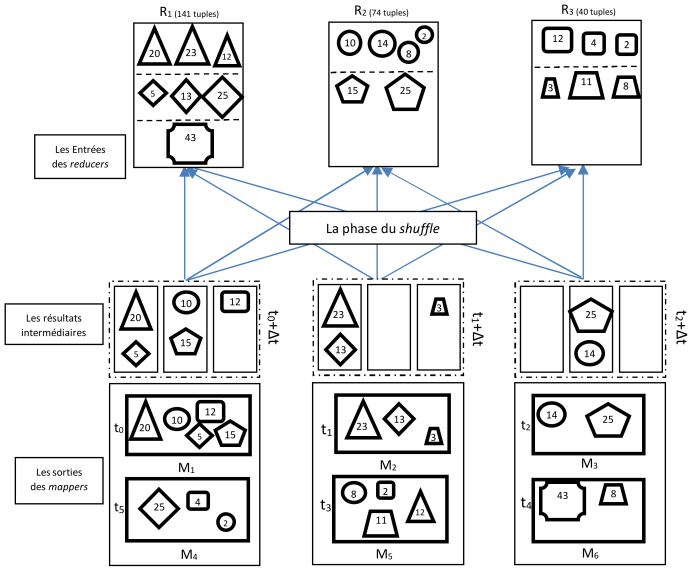


FIG. 1 – *Distribution des fragments (clé, liste(valeurs)) sur les reducers en utilisant le partitionnement par hachage)*

Dans Fig. 1, nous avons 6 tâches exécutées par 3 *mappers* et 3 tâches de réduction exécutées par 3 *reducers*. Les *mappers* génèrent 7 clés des fragments. Notez que chaque fragment (dans Fig. 1 un fragment est une forme géométrique) a une clé et le nombre à l'intérieur de la forme représente la liste des valeurs de ce fragment ou le nombre de tuples. Par exemple, la forme triangle de la tâche *map* $M1$ a la clé numéro 1 et la taille du fragment est 20 tuples, tel que chaque tuple contient une valeur de (d_year, i_brand_id, ss_sales_price). Notez que dans notre cas, le système ne prend pas en compte la phase *combine* de *MapReduce* [2]. Si nous utilisons la méthode aveugle (c.-à-d. le partitionnement par hachage), le *reducer* R_1 reçoit 7 fragments avec 141 tuples, le *reducer* R_2 reçoit 6 fragments avec 74 tuples, et R_3 reçoit 6 fragments avec 40 tuples. Nous remarquons que les DERs sont trop déséquilibrés. Évidement, le temps d'exécution du *Group By* est pris par le *reducer* lent R_1 qui contient 7 fragments.

Avec le mécanisme par défaut de *MapReduce*, lorsque le *mapper* exécute la première tâche M_1 et écrit le résultat dans les compartiments des RI, à l'instant $t_0 + \triangle t$ (notez que dans Fig. 1, $t_0 < t_1 <...< t_5$), la phase de *shuffle* peut commencer, en transférant les fragments sur les *reducers*. De plus, les *reducers* R_1, R_2 et R_3 ne peuvent pas commencer l'exécution de la fonction d'agrégat tant que tous les *mappers* n'ont pas terminé leur travail. Ce mécanisme implique que chaque *mapper* est aveugle aux actions prises par les autres *mappers*, ainsi que chaque *mapper* ne peut connaître ni le nombre des fragments produits par les autres *mappers*, ni leur taille. Par conséquent, nous devons développer une méthode capable de définir ces paramètres inconnus (c.-à-d. la clé du fragment produit, sa taille et leur placement approprié dans le compartiment des RI), afin d'équilibrer les charges des *reducers*.

2. Certaines fonctions, tel que l'écart-type et la variance, ne peuvent être agrégées que dans la phase *reduce*

Donc, si nous considérons que chaque requête de la charge de travail a une fréquence d'utilisation, c.-à-d. que de nombreux utilisateurs l'utilisent, un temps considérable est perdu si le système utilise le partitionnement aveugle. Supposons maintenant que des milliers de requêtes OLAP sont utilisées par des centaines d'utilisateurs. Le temps perdu sera alors exponentiel. Il est donc impératif de proposer une distribution intelligente afin de surmonter ce problème. Avant de détailler notre approche, nous présentons les travaux existants.

3 Travaux Existants

Dans cette section, nous présentons quelques travaux ayant abordé le problème d'asymétrie des clés et de l'équilibrage des charges des *reducers* dans le paradigme MapReduce. Nous commençons par l'algorithme de partitionnement par hachage, dénoté HashP, qui est le mécanisme par défaut dans quelques systèmes comme Apache Spark et Hadoop MapReduce. HashP ne peut obtenir de bonnes performances que lorsque la distribution des clés des fragments produits par les *mappers* est uniforme. Le deuxième algorithme naïf, appelé RangeP, est largement utilisé dans la technique de distribution des clés. Dans cette méthode, les n-uplets intermédiaires (clé, valeur) des fragments sont d'abord triés par clé, puis assignés pour chaque *reducer* de manière séquentielle selon cette plage de clés (c.-à-d. mécanisme du Round Robin). RangeP peut améliorer l'équilibrage des tâches de *reducers* si les clés des fragments sont uniformément distribuées. Tang et al. (2018) ont proposé l'algorithme SCID (*Splitting and Combination algorithm for skew Intermediate Data blocks*). SCID utilise une méthode d'échantillonnage pour prédire les clés fréquentes produites. Notez que bien que SCID utilise la technique d'échantillonnage comme (Xu et al., 2012) et (Chen et al., 2014) qui peut impacter sur la qualité d'équilibrage des charges des reducers, mais il est efficace même avec des EDD massives, puisqu'il distribue intelligemment les clés des *mappers* restantes qui ne sont pas être prises en compte dans d'échantillonnage. L'inconvénient de SCID est que la durée de l'algorithme échantillonnage peut ralentir le traitement lorsque la taille de L'EDD augmente.

Gufler et al. (2012) (dénoté DDCM) ont proposé une méthode dynamique pour distribuer à la volée les fragments produits par les *mappers* sur les partitions des *reducers*, en modifiant le mécanisme par défaut du MapReduce. Bien que l'approche DDCM fonctionne correctement avec n'importe quelle charge de travail utilisée, mais elle prend souvent beaucoup de temps, surtout si les tâches des *mappers* sont de forte asymétrie. De plus, il convient de noter que, bien que DDCM et SCID utilisent un algorithme de partitionnement qui peut achevé à un équilibrage quasi-optimal, toutefois, avec une fonction d'agrégat d'une requête OLAP, le fractionnement d'une partition aussi volumineuse ne sert à rien, puisque il crée d'autre cycles *MapReduce* pour évaluer cette fonction, et par conséquence le temps du traitement de la requête augmente. En outre, Ibrahim et al. (2013) ont développé l'approche LEEN pour minimiser le transfert de données pendant la phase de *shuffle*. Les auteurs de LEEN ont également changé le mécanisme de synchronisation de MapReduce afin de localiser les clés les plus fréquentes pour minimiser le taux de *shuffle* et pour établir un bon schéma des DERs. Cependant, LEEN peut ralentir le traitement de la requête lorsque les tâches des *mappers* sont trop déséquilibrées. D'autre travaux comme (Kwon et al., 2012) et (Baert et al., 2016) utilisent une technique intelligente pour équilibrer les charges des *reducers*. Bien que leur méthode soit efficace dans le cas d'une forte asymétrie des données dans les reducers, mais la technique d'arrêt utilisée pour déclencher le processus d'équilibrage peut perturber le traitement parallèle.

4 Notre Approche SGIA

Comme nous l'avons indiqué précédemment, nous pouvons résoudre notre problème à l'aide d'une méthode intelligente basée sur un système multi-agents, de telle sorte que chaque *mapper* devienne un *agent-mapper* (AM) et que chaque *reducer* devient un agent-reducer (AR). Nous pouvons considérer que le nombre des *mappers* et les *reducers* est égal au nombre de cœurs CPU assignés par le RM du *cluster*. Donc, le RM affecte le nombre approprié de ressources pour exécuter chaque requête (nous pouvons supposer que chaque requête est exécutée dans un petit *cluster* de nœuds). Dans ce petit *cluster*, nous avons un SMA composé d'un agent superviseur (AS), d'un ensemble de AMs, d'un ensemble de ARs et d'une base de connaissances (notée KB). Cette base est partagée entre tous les petits *clusters* existants. Avec la base KB, l'AS et les autres agents peuvent se coordonner pour créer un bon schéma des DERs. La base KB contient les méta-données de l'historique d'exécution des requêtes OLAP, qui peuvent aider les AMs à distribuer intelligemment les fragments sur les *reducers*. à savoir, un AM peut placer chaque fragment dans le compartiment approprié des RI, via KB. Notre objectif est d'obtenir une asymétrie faible des charges des *reducers* afin de minimiser le temps d'exécution de l'opération *Group By* et les fonctions d'agrégats. Avant de détailler l'architecture de SGIA, nous définissons certains concepts et notations utilisées dans cet article.

4.1 Concepts et Notations

Nous dénotons par n le nombre des AMs et les ARs assignées par le RM pour exécuter une requête OLAP. Notez que le nombre n représente le nombre de cœurs CPU exploités et pas le nombre de tâches *mappers* qui correspondent au nombre des fragments ou le nombre des blocs HDFS scannés. Cela signifier que chaque AM_i exécute plusieurs tâches *map*. Notez bien que le nombre de compartiments des RI de chaque AM (dénoté par p) est le même que le nombre des partitions des *reducers*, définis aussi par le RM au moment d'affecter les ressources [3]. Nous notons par $F.key$ la clé d'un fragment F et leur taille par $F.size$. Nous dénotons le profil de distribution Π_q^e, le profil où le système peut atteindre un équilibre quasi-optimal e lors d'exécution d'une requete q (c.-à-d. obtenir une asymétrie faible des charges des *reducers*). Nous notons aussi par $D = \{v_1, ..., v_p\}$ l'ensemble des charges des partitions [4], tel que $v_j = \sum_{k=1}^{f_j} Fk.size$ et f_j est le nombre de fragments Fk dans la partition j ($j \in 1..p$). Par exemple, dans Fig. 1, nous avons $D = \{141, 74, 40\}$. Dans la sous-section 4.2.2, nous expliquons comment calculer l'asymétrie d'un ensemble D.

4.2 Architecture de Notre Système

Pour construire à la volée un bon schéma de partitionnement des DERs, l'AS doit garder un historique suffisant de l'exécution des requêtes (voir Fig. 2). Aussi, l'AS doit bien exploiter cet historique afin de définir le meilleur profil Π_q^e. Cet historique est maintenu dans la base KB de notre système. Notez que notre système contient de nombreux nœuds, chaque requête est exécutée dans un petit *cluster* (c.-à-d. dans quelques ressources disponibles), comme indiqué dans Fig. 2. La base KB est partagée avec tout les nœuds du *cluster*. Dans SGIA, le compromis

3. Physiquement, les compartiments et les partitions sont des zones de la mémoire

4. Chaque partition est traitée dans une tâche *reducer*. Le nombre p est comme le paramètre : *spark.sql.shuffle.partitions* de Spark.)

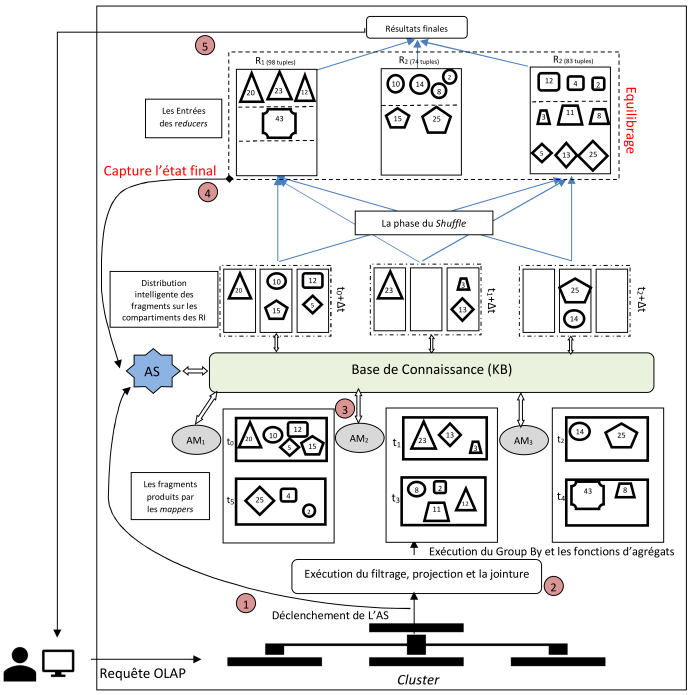


FIG. 2 – *Distribution intelligente des fragments sur les partitions des reducers dans un petit cluster de notre système*

entre le bénéfice à long terme et le coût à court terme devrait être abordé. A savoir : (1) la taille des méta-données stockées dans la base KB doit être aussi petite que possible (sinon, le système préfère de maintenir les résultats des requêtes au lieu ces méta-données) et (2) le temps d'exécution pour établir un bon schéma des DERs avec notre approche doit être rationnel et ne doit pas avoir d'impact sur les performances de la requête. Donc, avant de détailler le processus de notre architecture, nous définissons quelques contraintes et règles.

4.2.1 La taille de la base KB

Comme nous avons vu précédemment la taille de la base KB doit être aussi petit que possible, donc pour chaque exécution d'une requête q, l'AS est capable de capturer les méta-données des fragments (clé du fragment et sa taille) dans chaque partitions, avant que le système exécute la fonction d'agrégat, puis les sauvegarde dans des listes au niveaux du KB. Évidemment ces données stockées sont des valeurs numériques et prennent un espace mémoire négligeable. A partir de ces données, l'AS peut définir le Π_q^e dans les prochaines exécutions de q, et via le profil Π_q^e, les AMs peuvent établir un bon schéma de distributions des fragments sur les *reducers*. Dans KB, la structure des méta-données est un dictionnaire avec plusieurs clés, comme indiqué dans Fig. 3. Telle que la clé est une liste des noms des requêtes (ex. q_1 et q_5), et que la valeur du dictionnaire est une liste d'informations concernant les fragments (clé du fragment et sa taille). Pour chaque nouvelle exécution d'une requête q, le système crée

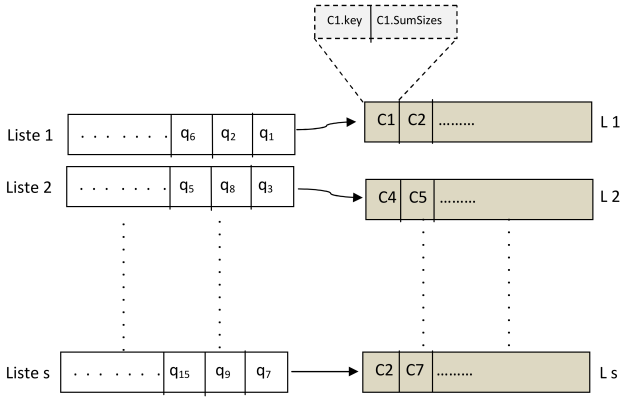


FIG. 3 – *Un exemple illustratif pour montrer comme les méta-données sont stocker dans KB*

une nouvelle liste (c.-à-d. une nouvelle clé) pour cette requête et la valeur correspond à cette clé (comme $L1$, $L2$,..,Ls dans Fig. 3). Pour ne pas confondre avec un fragment produit par un AM, nous appelons fragment-fusionné le fragment stocker dans KB. Un fragment-fusionné ayant la même clé du fragment produit par un AM, mais sa valeur est la somme de tous les tailles des autres fragments qui ont la même clé. Nous remarquons qu'il est possible de trouver des requêtes qui produisent les mêmes fragments comme le montre Fig. 3.

4.2.2 Mesure de l'équilibrage

Pour mesurer l'équilibrage de charges ou l'asymétrie des données, nous utilisons la formule de covariance suivante :

$$Cov = \frac{std}{mn} \times 100 \tag{1}$$

Où std est l'écart-type d'une distribution D (voir les notations dans la section 4.1), et mn est la valeur moyenne de D. Par exemple, dans Fig. 2, nous avons $D1 = \{98, 74, 83\}$ et $Cov1 = \frac{12.12}{85} \times 100 = 14,25$. Pour l'exemple de la section 2, nous avons $D2 = \{141, 74, 40\}$ et $Cov2 = \frac{51.39}{85} \times 100 = 60.45$. Nous remarquons que $Cov1 < Cov2$, le meilleur équilibre est donc celui qui a la valeur minimale de Cov. En SGIA on peut distinguer trois types d'asymétrie : (1) Asymétrie faible dans le cas où $Cov < 20.0$, et c'est l'équilibre optimal que nous cherchons ; (2) Asymétrie moyenne dans le cas où $20.0 \leqq Cov < 40.0$, et c'est un équilibre acceptable ; et (3) Asymétrie forte dans le cas où $Cov \geqq 40.0$, qui est un déséquilibre des charges que nous évitons d'obtenir.

4.2.3 Architecture détaillée

L'architecture de notre système est composée de cinq étapes, comme indiqué dans Fig. 2 (dans Fig. 2, le nombre à l'intérieur du cercle représente le numéro de l'étape). Nous résumons ces étapes dans Algorithme 1. Dans cet algorithme, les variables d'entrées sont : une requête q, lancée par un utilisateur u, et les méta-données des fragments-fusionnés stockés dans la base

KB (si les méta-données de q existent). Le résultat de l'Algorithme 1 est le meilleur profil Π_q^e qui peut aider les AMs à établir un bon schéma de partitionement des charges des reducers.

Algorithme 1 : Les étapes de SGIA

Input : $q \in W = \{q_1, q_2, ..., q_f\}$, KB
/* q est une requête OLAP lancée par un utilisateur u et KB est la base de connaissances. */
Output : résultas de la requête
1 **while** *le système fonctionne correctement* **do**
2 Lancement de la requête q;
3 Définition des variables n et p par le RM ; /* n est le nombre des AMs et ARs (CPU cores) et p est le nombre de partitions traitées par les ARs. Voir Fig. 2 */
4 Le système lance l'AS;
5 $exist$:=**CheckingExisting**(q) ; // $exist$ est une variable booléenne
6 Le système exécute le filtrage, projection et la jointure en étoile de q;
7 **if** $exist$==*true* **then**
8 **DefineBestProfile**() ; /* l'AS définis Π_q^e. */
9 **else**
10 les AMs distribuent les fragments selon le partitionnement par hachage ;
11 **if** *I_6 termine avant I_8* **then**
12 Suspension du système ; /* les AMs ne peuvent pas distribuer les fragments jusqu'à ce que l'AS construit Π_q^e. */
13 Les AMs continuent l'exécution en utilisant le mécanisme par défaut de MapReduce, soit en exploitant Π_q^e ou en utilisant le partitionnement par défaut pour distribuer les fragments;
14 **if** $exist$==*false* **then**
15 L'AS capture les fragments produits dans les partitions des *reducers* avant que les ARs commencent le calcule des fonctions d'agrégats.
16 Les ARs calculent les fonctions d'agrégats;
17 Résults final de q;

Algorithme 2 : Construction de Π_q^e

Input : $listF = \{F1, F2, ..., Fk\}$, P={P_1, P_2,..,P_p}
/* $listF$ est la liste des fragments-fusionnés, P est l'ensemble des partitions traitées par les ARs. */
Output : construction de Π_q^e
1 $stack$:=$listF$.sort ; // tri de $listF$
2 $range$:=$\lceil \frac{\sum_{j=1}^{k} Fj.size}{p} \rceil$; // taille moyenne des partitions
3 nP :=1, $round$:=1 ; // nP nombre actuel de la partition traitée
4 **for** j :=*1 to* p **do**
5 $P[j]$.size=0; // initialisation
6 **while** *(*$stack$*!=null) or (*$round$*<=*p*)* **do**
7 pos :=0; $decho$:=$stack$.size;
8 **while** *(*$P[nP-1].size \leq range$*) or (*pos*<*$stack.size$*)* **do**
9 $clust$=$stack[pos]$; // prend le premier element
10 $P[nP-1]$.size :=$P[nP-1]$.size+$clust$.size;
11 **if** *(*$P[nP-1].size \leq range$*) or (*$clust.size$*>=*$range$*)* **then**
12 $P[nP-1]$.add($clust$);
13 $stack$.deleteElement($clust$);
14 $round$:=1;
15 pos :=pos+$\lceil decho/2 \rceil$, $decho$:=$\lceil decho/2 \rceil$;
16 **else**
17 $P[nP-1]$.size :=$P[nP-1]$.size-$clust$.size;
18 pos :=pos+$\lceil decho/2 \rceil$, $decho$:=$\lceil decho/2 \rceil$;
19 **if** pos>=$stack.size$ **then**
20 $round$++;
21 nP :=nP mod p+1;
22 **DitributeRemindClusters**($stack$, P);
23 **CreateBestProfile**();

Comme mentionné dans l'instruction I_3 de l'Algorithme 1, le gestionnaire de ressources du *cluster* affecte les ressources suffisantes pour exécuter la requête q (c.-à-d. le RM définit les paramètres n et p). Après cela, le système (ex. "Application Master" ou "Driver" de Spark) déclenche l'AS dans I_4 (l'étape 1 de notre approche), l'AS vérifie l'existence de q dans KB

via la fonction **CheckingExisting(**q**)** (voir I_5). Dans cette fonction, nous pouvons utiliser une méthode d'analyse lexicale et syntaxique simple pour vérifier l'existence de q dans KB. Dans l'étape 2 (Instruction I_6 d'Algorithme 1), le système continue l'exécution des opérations de filtrage, de projection et de jointure en étoile de la requête q. Notez que les étapes 1 et 2 sont effectuées en parallèle. Lorsque l'AS assure l'existence de q dans KB, il lance la procédure **DefineBestProfile ()** dans I_8 ; Cette procédure est détaillée dans l'Algorithme 2.

L'étape 3 du système est détaillée de I_11 à I_13. La condition I_11 se réalise rarement, car le temps pris par la procédure **DefineBestProfile ()** est plus petit que celui pris par l'étape 2. Notez que si q n'existe pas dans KB, les AMs utilisent le partitionnement par hachage. De plus, dans notre cas, les AMs suivent le mécanisme par défaut du paradigme MapReduce (voir I_13) ; à savoir, après que le premier AM_i ait fini de distribuer ses fragments sur les compartiments des RI, la phase de *shuffle* peut commencer. Dans I_15 (étape 4 de Fig. 2), le système doit capturer les différents fragments des partitions avant que les ARs calculent les fonctions d'agrégats. Enfin (étape 5 de notre architecture), le système envoie le résultat à l'utilisateur. Les entrées de l'Algorithme 2 sont la variable $listF$ qui contient les fragments-fusionnés produits et la variable P qui représente la liste des partitions.

Le résultat de l'Algorithme 2 est la table Π_q^e, qui contient les différentes actions à prendre par les AMs ; c.-à-d. le meilleur emplacement de chaque fragment dans le compartiment des RI. Cette table est stockée dans KB et diffusée, de sorte que tous les AMs l'accèdent localement. Table 23 montre la structure de la table Π_q^e. Dans I_1, nous créons la liste $stack$ qui est la $listF$ triée de manière ascendante selon la taille des fragments. Dans I_2, nous calculons la taille moyenne de $stack$. Le paramètre $range$ utilisé pour équilibrer les charges des partitions des *reducers*. L'idée de notre approche est d'essayer de répartir uniformément les fragments-fusionnés sur les partitions de l'ensemble P, ce qui peut minimiser la covariance Cov (c.-à-d. obtenir le meilleur équilibre ou un équilibre acceptable, voir sous-section 4.2.2.). Nous pouvons distinguer deux problèmes majeurs qui peuvent empêche le système d'atteindre notre objectif : (1) la taille des fragments-fusionnés peut dépasser la moyenne $range$, et (2) l'asymétrie des tailles des fragment-fusionnées est forte. Les deux problèmes ci-dessus augmentent la valeur de Cov et peuvent dégrader les performance de *Group By*.

Pour remédier au problème (1), le système doit distribuer les fragments-fusionnés de grande taille à l'avance, voir condition ($clust$.size>= $range$) dans I_11). Puisque la division des grands fragments en petits morceaux (comme certaines approches (Gufler et al., 2012; Tang et al., 2018)) ne sert à rien dans notre cas (il implique d'additionner d'autre cycles MapReduce pour calculer les fonctions d'agrégats et évidemment le temps d'exécution de l'opération *Group By* augmente). Pour résoudre le problème (2), nous avons utilisé la technique de Round Robin pour répartir uniformément les fragments-fusionnés sur les partitions des *reducers*. De plus, nous avons utilisé la recherche par dichotomie pour analyser la liste $stack$, afin de remplir chaque partition avec des fragments de taille variante (voir les instructions I_15 et I_18). Cette

	Requête q			
Fragments	F1	F2	...	Fk
Actions	Pos_1	Pos_5	...	Pos_2

TAB. 1 – *La structure de* Π_q^e

technique permet de minimiser le degré d'asymétrie des charges des *reducers*. à la fin d'Algorithme 2, nous ajoutons la procédure **DitributeRemindClusters(**$Stack$**,** R**)** pour distribuer les fragments-fusionnés restants de $stack$ et la procédure **CreateBestProfile ()** (voir Algorithme 4) pour créer la table Π_q^e. L'idée de l'Algorithme 3 est simple, tel que le système place le fragment de taille supérieure dans la partition moins chargée.

Algorithme 3 : DitributeRemind-Clusters($Stack$, P)

```
1 while (stack !=null) do
2     clust :=stack.getElement() ;
        // recupèration du premier
        element
3     part :=min(P) ; // selection de la
        partition la moins chargée
4     part.add(clust) stack.delete(clust) ;
        // suppression du fragment
        de la liste stack
```

Algorithme 4 : CreateBestProfile()

```
1 indice=1;
2 for (j=1 to P.size) do
3     part=P[j];
4     for k=1 to part.listClusters.size do
5         Π_q^e[indice].key=part.listClusters[k].key
            ; // selectionné la clé du
            fragment
6         Π_q^e[indice].position=j ;   // placer
            le fragment dans la
            partition appropriée
7         indice++;
```

5 Expérimentations et Résultats

5.1 Environnement du développement

Nous avons évalué notre approche SGIA sur un *cluster* pratique. L'environnement expérimental est basé sur le système Ray (Moritz et al., 2018) qui permet de simuler le paradigme MapReduce. Ray [5] est un *framework* de traitement distribué flexible, de hautes performances, conçu pour les applications d'apprentissage par renforcement. Ray est une hybridation entre les systèmes parallèles tels que MapReduce et Apache Spark, et le modèle d'acteur pour les tâches asynchrones comme Akka [6]. Notez que dans ce papier, nous n'avons pas directement implémenté SGIA sur Spark ou Hadoop, pour deux raisons : La première raison est qu'il est non trivial de changer le mécanisme de synchronisation par défaut de ces systèmes, en particulier avec l'actuel API de Spark SQL (nous prévoyons de le faire dans les travaux à venir) ; La deuxième raison est que nous pouvons facilement simuler certaines approches de base sur le *cluster* Ray, telles que HashP, RangeP, etc. (voir section 3) et de les comparer avec SGIA.

5.2 Implémentation

Nous avons évalué SGIA sur un *cluster* composé de 6 nœuds, chaque nœud possède 8 cœurs CPU et 16 GB de taille mémoire. Nous avons installé dans tous les nœuds Redis [7] serveur, Ray version 0.7.0.dev3, système Anaconda [8] avec python version 3.7.3 et d'autres bibliothèques de dépendances. Le *cluster* Ray n'est pas comme Hadoop ou Spark, chaque nœud peut être le maître à tout moment, il suffit d'exécuter l'instruction *ray start - - head - - redis-port* = 6379, tel que 6379 est le port du serveur Redis. Un nœud peut joindre le *cluster*

5. https ://github.com/ray-project/ray
6. Akka. https ://akka.io/.
7. disponible à partir du http ://download.redis.io/redis-stable.tar.gz
8. https ://repo.continuum.io/archive/Anaconda3-5.0.1-Linux-x86_64.sh

après avoir exécuté l'instruction "*ray start - - address-redis= address-IP :* 6379", de sorte que *address-IP* soit l'adresse du nœud principal défini précédemment. Bien évidemment, le système Ray ne peut pas utiliser toutes les ressources des nœuds. Nous devons donc conserver certaines ressources pour le système d'exploitation et d'autres applications. À savoir, nous utilisons dans chaque nœud 6 cœurs CPU et 12 GB pour le système Ray et le serveur Redis. Pour implémenter les agents AMs, ARs et l'AS, nous avons utilisé le modèle d'acteur de Ray. L'acteur fonctionne de manière autonome dans le nœud. Lorsque le système instancie un acteur, un nouveau *worker* est créé et les méthodes de cet acteur sont planifiées sur ce *worker* spécifique et peuvent accéder à l'état de ce dernier et le transforme [9]. Avec le système Ray, nous pouvons simuler le paradigme MapReduce avec des méthodes asynchrones.

Pour évaluer les avantages et les limitations de SGIA, nous l'avons comparé avec cinq approches de base, à savoir HashP, RangeP, DDCM, LEEN et SCID (voir les notations dans la section 3). Nous avons testé ces approches avec des requêtes de différentes complexités, comme nous le montrerons dans la sous-section suivante.

Préparation des données et la charge des requêtes. Dans les versions courantes de Ray, il n'y a pas des APIs riches comme Spark SQL pour exécuter les requêtes SQL. Donc, puisque notre objectif est d'améliorer l'opération *Group By*, nous n'avons pas besoin d'exécuter l'opération de filtrage et de jointure. Pour bien montrer l'asymétrie des charges des *mappers* et des *reducers*, nous avons utilisé le banc d'essai TPC-DS qui est conçu pour étudier la forte l'asymétrie dans les SGBDs (Rabl et al., 2013). Nous avons généré des données avec l'outil *spark-sql-perf* (voir https ://github.com/databricks/spark-sql-perf), en utilisant le langage Scala et Spark, où nous stockons les données directement dans HDFS. Nous avons défini le facteur de réplication de HDFS 1 et nous conservons les paramètres de Spark par défaut.

De plus, puisque nous utilisons un EDD en étoile, nous avons utilisé une partie de ce banc d'essai, à savoir, une table de faits parmi sept et neuf dimensions parmi 17 (voir Table 5.2). Nous avons généré deux EDDs massives, le premier, denoté DW1, a 10 GB de taille, et le deuxième, dénoté DW2, a 100 GB (en format Parquet). De plus, nous avons sélectionné et adapté 18 requêtes parmi les 99 du banc d'essai ; à savoir, nous sélectionnons uniquement les requêtes qui sollicitent les tables de l'EDD, et nous éliminons certaines requêtes imbriquées et celles qui utilisent les opérateurs *RullUp* et *CUBE* [10]. Nous ne pouvons pas exécuter ces requêtes directement dans Ray, nous devons les transformer comme indiqué dans la Table 3. Nous supprimons d'abord *Group By* et les fonctions d'agrégats, puis nous gardons les autres opérations. Après cela, nous exécutons les requêtes transformées avec Spark et nous sauvegardons le résultat sur HDFS. Nous pouvons supposer que chaque résultat représente une vue matérialisée, stockée sous forme de fichiers dans HDFS. Enfin, nous pouvons charger ces fichiers et exécuter l'opération *GROUP By* et les fonction d'agrégats des requêtes originales avec Ray system.

5.3 Évaluation

Puisque notre objectif est d'améliorer les performances de *Group By* et les fonctions d'agrégats d'une requête OLAP, et d'optimiser le traitement parallèle du système distribué, nous

9. Le *worker* est un processus qui exécute des tâches (des fonctions distantes), invoquées par un *Driver* de l'application ou un autre worker (Moritz et al., 2018)

10. Notez que dans ce papier, nous ne considérons pas les opérateurs *CUBE* et *RullUp* qui nécessite de générer d'autres clés dans les RI

	Tables de l'EDD
1	*store_sales*
2	*customer*
3	*customer_address*
4	*customer_demographics*
5	*item*
6	*time_dim*
7	*date_dim*
8	*household_demographics*
9	*promotion*
10	*store*

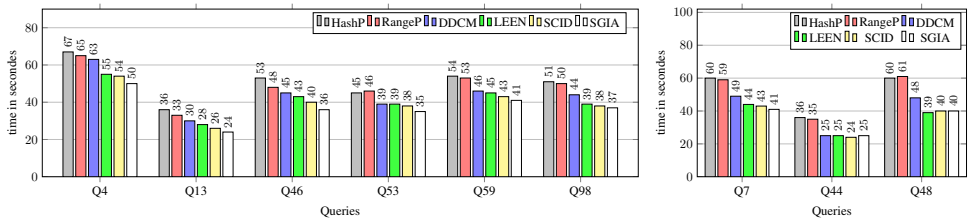
TAB. 2 – *Caractéristiques de EDD*

TAB. 3 – *Transformation d'une requête*

requête originale	requête transformée
SELECT d_year, i_brand_id, SUM(ss_sales_price) FROM date_dim, item, store_sales WHERE date_dim.d_date_sk=store_sales .ss_sold_date_sk AND store_sales.ss_item_sk=item.i_item_sk AND i_manufact_id=128 GROUP BY d_year, i_brand_id;	SELECT d_year, i_brand_id, ss_sales_price FROM date_dim, item, store_sales WHERE date_dim.d_date_sk=store_sales .ss_sold_date_sk AND store_sales.ss_item_sk=item .i_item_sk AND i_manufact_id = 128;

avons évalué et comparé SGIA avec cinq approches de base citées précédemment, sous l'aspect du temps d'exécution. Nous avons divisé les 18 requêtes sélectionnées en deux catégories, celles qui ont une asymétrie de données faible et moyenne de distribution des clés des *mappers* et d'autres ont une distribution fortement asymétrique. Figure 4 indique le temps d'exécution des requêtes avec DW1 et Fig. 5 avec DW2, selon différentes approches. Notez que par souci de clarté, nous avons montré la durée d'exécution de 9 requêtes parmi les 18.

Avec une asymétrie faible et moyenne (voir Fig. 4 (a) et Fig. 5 (a)), SGIA surpasse presque toutes les autres approches. Nous remarquons que les mauvais résultats sont obtenus avec les approches aveugles HashP, RangeP. Dans Fig. 4 (a), SCID a légèrement les mêmes performances que SGIA, car avec DW1, le nombre de clés n'est pas trop important, ce qui implique que le temps pour faire l'échantillonnage dans SCID est négligeable. Dans Fig. 5 (a), LEEN donne à peu près les mêmes résultats que SGIA avec les requêtes Q4, Q13 et Q46. La raison est vu que LEEN se focalise sur l'optimisation de la localité des données, il gagne plus de temps dans la phase *shuffle* par rapport aux autres approches. En outre, nous pouvons constater que, bien que DDCM puisse offrir un meilleur équilibrage des charges que toutes les approches,

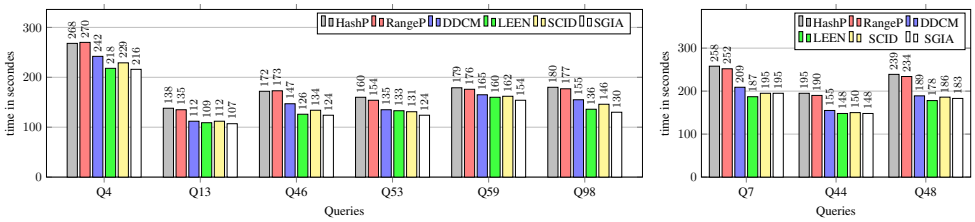


(a) Avec une faible et moyenne asymétrie

(b) Avec une forte asymétrie

FIG. 4 – *Temps d'exécution des requêtes avec DW1*

cependant, elle prend beaucoup de temps pour rassembler les informations pertinentes qui permettent de prendre une bonne décision d'équilibrage. Par conséquent, les résultats en DDCM ne sont pas promis. Par ailleurs, avec une asymétrie forte des données avec DW1, les résultats dans SGIA, LEEN et SCID sont à peu près identiques. La raison est que dans SGIA, nous cherchons pas d'atteindre un équilibrage optimal, ce qui implique d'avoir certains retardateurs dans les *reducers* avec une asymétrie forte. Dans Fig. 5 (b)), en raison de l'augmentation du volume de données transférées dans la phase *shuffle*, LEEN est légèrement meilleur que SGIA.



(a) Avec une faible et moyenne asymétrie (b) Avec une forte asymétrie

FIG. 5 – *Temps d'exécution des requêtes avec DW2*

De plus, nous remarquons que DDCM donne toujours des faibles performances. Finalement, nous observons que le temps d'exécution le plus long est pris dans Q4, puisque dans cette requête, nous sélectionnons de nombreux attributs sans utiliser des filtres.

6 Conclusion et travaux futurs

Dans cet article, nous avons présenté une nouvelle méthode permettant d'optimiser l'opération *Group By* d'une requête OLAP sur un système distribué, sans changer le mécanisme par défaut de MapReduce. Nos expérimentations révèlent que notre approche est plus appropriée dans le cas d'une asymétrie faible et moyenne de la distribution des clés des *mappers* et acceptable dans le cas d'une asymétrie forte. Nous avons également vu comment modéliser le paradigme MapReduce à l'aide des tâches asynchrones via le système Ray. En outre, bien que l'approche SGIA ne permet pas d'atteindre un équilibrage optimal, mais SGIA repose sur une technique simple et efficace pour n'importe quel type de requête OLAP.

Dans les travaux futurs : (1) nous prendrons en compte d'autre opérateurs comme *CUBE* et *RullUp*, (2) nous intégrerons l'approche de LEEN avec SGIA pour minimiser le taux de shuffle, et (3) nous intégrerons notre algorithme dans les systèmes Spark SQL et HiveQL.

Références

Agarwal, S., S. Kandula, N. Bruno, M.-C. Wu, I. Stoica, et J. Zhou (2012). Reoptimizing data parallel computing. In *Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*, pp. 281–294.

Agarwal, S., B. Mozafari, A. Panda, H. Milner, S. Madden, et I. Stoica (2013). Blinkdb : queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pp. 29–42. ACM.

Baert, Q., A. C. Caron, M. Morge, et J.-C. Routier (2016). Fair multi-agent task allocation for large data sets analysis. In *International Conf. on Practical Applications of Agents and Multi-Agent Systems*, pp. 24–35. Springer.

Chen, Y., Z. Liu, T. Wang, et L. Wang (2014). Load balancing in mapreduce based on data locality. In *International Conference on Algorithms and Architectures for Parallel Processing*, pp. 229–241. Springer.

Dean, J. et S. Ghemawat (2008). Mapreduce : simplified data processing on large clusters. *Communications of the ACM 51*(1), 107–113.

Gao, Y., Y. Zhou, B. Zhou, L. Shi, et J. Zhang (2017). Handling data skew in mapreduce cluster by using partition tuning. *Journal of Healthcare Engineering 2017*.

Gufler, B., N. Augsten, A. Reiser, et A. Kemper (2012). The partition cost model for load balancing in mapreduce. In *Cloud Computing and Services Science*, pp. 371–387. Springer.

Heintz, B., A. Chandra, R. K. Sitaraman, et J. Weissman (2016). End-to-end optimization for geo-distributed mapreduce. *IEEE Transactions on Cloud Computing 4*(3), 293–306.

Ibrahim, S., H. Jin, L. Lu, B. He, G. Antoniu, et S. Wu (2013). Handling partitioning skew in mapreduce using leen. *Peer-to-Peer Networking and Applications 6*(4), 409–424.

Kwon, Y., M. Balazinska, B. Howe, et J. Rolia (2012). Skewtune : mitigating skew in mapreduce applications. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pp. 25–36. ACM.

Moritz, P., R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, et al. (2018). Ray : A distributed framework for emerging {AI} applications. In *13th {USENIX} Sympo. on Operat. Syst. Design and Implem. ({OSDI} 18)*, pp. 561–577.

Rabl, T., M. Poess, H.-A. Jacobsen, P. O'Neil, et E. O'Neil (2013). Variations of the star schema benchmark to test the effects of data skew on query performance. In *Proceed. of the 4th ACM/SPEC Intern. Conf. on Performance Engineering*, pp. 361–372. ACM.

Ramakrishnan, S. R., G. Swart, et A. Urmanov (2012). Balancing reducer skew in mapreduce workloads using progressive sampling. In *Proceedings of the Third ACM Symposium on Cloud Computing*, pp. 16. ACM.

Ramdane, Y., B. Omar, K. Nadia, et B. Fadila (2019). Conception physique d'un entrepôt de données distribuées basée sur k-means équilibré. In *EGC*, pp. 177–188.

Tang, Z., X. Zhang, K. Li, et K. Li (2018). An intermediate data placement algorithm for load balancing in spark computing environment. *Future Generation Comp. Systems 78*, 287–301.

Xu, Y., P. Zou, W. Qu, Z. Li, K. Li, et X. Cui (2012). Sampling-based partitioning in mapreduce for skewed data. In *2012 seventh ChinaGrid annual conference*, pp. 1–8. IEEE.

Zaharia, M., M. Chowdhury, M. J. Franklin, S. Shenker, et I. Stoica (2010). Spark : Cluster computing with working sets. *HotCloud 10*(10-10), 95.

Summary

Enhancing OLAP query performance in a distributed system such as Hadoop and Spark is a challenging task. An OLAP query is composed of several operations, such as projection, filtering, join, and grouping operations. Each operation can be executed in the map or in the reduce phase with one or several Spark stages. While some operations, such as star join and filtering, can be enhanced by using a static partitioning technique and load balancing for the data since we have the prior knowledge of the load balancing decision. However, optimizing

SGIA : Stratégie Intelligente de Groupement

Group By and aggregate functions, requires in general, a dynamic technique of partitioning and distributing to make a good partition scheme of the reducer inputs since we can only pick up the relevant information at query runtime. In this paper, we propose a smart method, called SGIA, to balance on the fly the reducer inputs. We used a multi-agent system that can balance smartly the reducer loads for Group By task. Our experiments reveal that our proposal outperforms existing approaches in terms of query execution time.