Nouvelle stratégie pour le traitement distribué des processus décisionnels massifs dans un Big Data Warehouse

Rado Ratsimbazafy*, Fadila Bentayeb*, Omar Boussaid*

*Université de Lyon, Université Lumière Lyon 2, Laboratoire ERIC
5 Avenue Pierre Mendes France, 69676 Bron Cedex
prenom.nom@univ-lyon2.fr, http://eric.ish-lyon.cnrs.fr

Résumé. Cet article traite du problème de l'optimisation de l'exécution des charges de requêtes massives dans le cadre des entrepôts de données (ED) distribués où le nombre de processus simultanés à traiter se compte par milliers. En nous inspirant des techniques d'optimisation utilisées dans le contexte centralisé, nous proposons dans cet article une nouvelle stratégie de sélection et de stockage de vues matérialisées (MV) basée sur système de fichiers distribués ; puis nous abordons le traitement des charges de requêtes décisionnelles massives en utilisant les MV. Notre approche joue un rôle de médiateur entre les utilisateurs et l'entrepôt de données pour proposer de meilleurs plans d'exécution à leurs requêtes. Les premiers résultats que nous avons obtenus, à partir de nos expérimentations montrent que dans un environnement distribué notre approche améliore de plus de 50% le coût d'exécution d'une charge de requêtes par rapport au système fourni par défaut.

1 Introduction

L'entreposage et l'analyse en ligne des données massives (*big data*) sont devenus en quelques années l'activité principale de beaucoup d'entreprises et de chercheurs (Ahuja et al. (2009); Thusoo et al. (2010b)). L'intérêt porté à l'avènement des données massives a fait évoluer le système d'information décisionnel (SID), et par conséquent les entrepôts de données et l'OLAP (Online Analytical Processing) (Chaudhuri et al. (2011)). De nouveaux modèles d'entrepôts de données sont alors apparus (Figure 1) : les systèmes basés sur des systèmes de gestion de bases de données relationnelles (SGBDR) tels que *Teradata* [1], *Greenplum* [2], et les systèmes basés sur le paradigme *MapReduce* (Dean et Ghemawat (2004)), comme *Hive* (Thusoo et al. (2010a)), où les données sont stockées sur un système de fichiers distribués tels que GFS de *Google* ou HDFS de *Hadoop*.

Par ailleurs, l'intérêt grandissant autour du "*big data analytics*" a fait naître plusieurs techniques, stratégies et approches, mais aussi de nouveaux profils métiers (*data scientist, big data engineer,* ...). Cependant, l'analyse de telles quantités de données, pour récupérer les informations pertinentes, doit être réalisée dans une durée acceptable (Cohen et al. (2009)). Les

1. http ://www.teradata.com
2. http ://greenplum.org/

attentes des décideurs ont toujours été grandes, leur immense besoin d'analyse nécessite de nombreux processus d'interrogation. Ces derniers sont réalisés à l'aide de multiples requêtes complexes comportant plusieurs opérations sur des données volumineuses. Dans ce travail, nous continuons les efforts entrepris par d'autres chercheurs (Zaharia et al. (2009); Herodotou et Babu (2013); Wang et Chan (2013)) qui ont travaillé sur l'optimisation de traitement de processus d'analyse multiples dans les entrepôts de données, ou bases de données, utilisant le paradigme *MapReduce*.

Dans ce papier, nous décrivons en détail l'approche que nous proposons pour améliorer le temps de traitement des processus d'analyse en ligne massifs, dans les entrepôts de données fonctionnant dans un environnement *Hadoop* utilisant *MapReduce* comme outil d'exécution de requêtes. Dans notre approche, nous nous sommes inspirés des techniques d'optimisation dans les entrepôts de données basés sur un SGBDR, tels que le *Multiple Query Optimization (MQO)*, la matérialisation de vues (Sellis (1988); Bello et al. (1998); Gupta et Mumick (1999); Goldstein et Larson (2001)). Nous traitons plus précisément le problème de la sélection de vues (*Vue Selection Problem (VSP)*), qui a été largement étudié (Baralis et al. (1997); Zhang et al. (2001); Boukorca et al. (2014)) et défini comme étant NP-Complet (Gupta et Mumick (1999)). Nous avons exécuté plusieurs requêtes issues du banc d'essai TPC-DS [3] pour une première évaluation de notre approche. Nos premières expérimentations ont fait l'objet d'une comparaison avec le système standard fourni par *Hive*, et ont montré des résultats intéressants.

Le reste de cet article est organisé comme suit. Dans la section 2, nous présentons les approches existantes dans la littérature qui nous ont inspirés, ou qui sont proches de notre proposition. Dans la section 3, nous décrivons les processus et stratégies de notre approche pour résoudre le problème de traitement des requêtes décisionnelles massives. Dans la section 4, nous présentons l'évaluation des performances de notre approche. Enfin, la section 5 conclut l'article et présente quelques perspectives pour la suite de nos travaux.

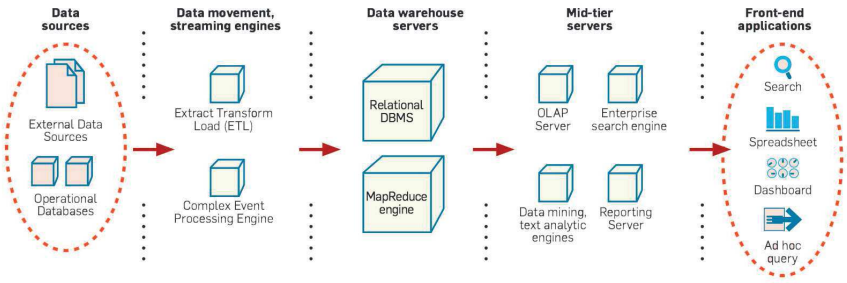

FIG. 1 – *Évolution du SID (Chaudhuri et al. (2011)).*

3. http ://www.tpc.org/tpcds/

2 État de l'art

Plusieurs recherches ont été menées sur les traitements de charges de requêtes, que ce soit dans les bases de données ou dans les entrepôts de données. Plusieurs approches ont été proposées. Nous focalisons notre attention sur les approches autour du partage de données, du partage de tâches ou de fragmentation de données. Sellis (1988) a proposé le *Multiple Query Optimization* (MQO), qui consiste à détecter les sous-expressions communes des requêtes exécutées en même temps, évaluer les sous expressions, regrouper les requêtes partageant les mêmes sous-expressions et partager les données communes entre les requêtes. Le MQO est une approche très intéressante, mais le coût de la recherche des sous expressions communes est élevé, car sur une charge donnée peu de requêtes partagent les mêmes sous-expressions et les requêtes n'arrivent pas toujours en même temps.

Dans une optique de partage de données, nous nous sommes intéressés aux vues matérialisées. Comment sont-elles choisies ? Combien de vues faut-il matérialiser ? Et quand faut-il les matérialiser ? Ces questions ont été largement analysées et prouvées comme étant des problèmes NP-Complet (Gupta et Mumick (1999)). Le problème de la sélection de vues a attiré beaucoup d'attention. Jian et al. (1997) propose une implémentation du MQO pour sélectionner les vues sur la base de spécification du *Multiple View Processing Plan*, qui matérialise les résultats intermédiaires partagés par chaque sous-ensemble de requête. Kotidis et al. (1999) avec DynaMat propose une approche qui matérialise dynamiquement les vues correspondantes à la charge de requêtes en entrée, il examine continuellement les requêtes reçues et matérialise le meilleur ensemble de vue en se basant sur un modèle de coût. L'approche semble être très intéressante, mais elle ne se concentre que sur un seul cube de données et nous n'avons pas pu déterminer quel type de requête a été exécuté. Sur le nombre de requêtes à traiter, Gacem et Boukhalfa (2013) ont proposé d'utiliser l'algorithme K-means pour regrouper les requêtes similaires afin de fragmenter horizontalement l'entrepôt de données. L'approche est innovante et a été testée pour supporter de très grandes charges de requêtes. L'efficacité de l'approche dépend du nombre de classes de requêtes, avec des requêtes hétérogènes, l'approche n'apporte aucune amélioration.

Tous les travaux cités précédemment ont été traités et prouvés dans les ED sur SGBDR. Dans les ED distribués utilisant MapReduce, où l'utilisation du *cloud computing* semble être privilégiée, nous avons détecté trois tendances :

— Les travaux se basant sur la planification des tâches (*scheduling*) (Isard et al. (2009); Zaharia et al. (2009)) qui traitent, réorganisent et réordonnent le traitement de chaque tâche MapReduce ;
— Les recherches qui utilisent l'élasticité du *cloud* et la répartition des requêtes sur les nœuds (Herodotou (2012) ; Marcus et Papaemmanouil (2016)) qui consistent à trouver le bon nombre de nœuds pour le traitement d'une charge donnée, tout en répartissant le traitement des requêtes de la charge sur les nœuds ;
— Les approches sur le partage de tâches ou de données (Nykiel et al. (2010); Wang et Chan (2013)) qui se basent sur la mutualisation des données lors des traitements pour réduire le nombre d'accès au système de fichiers distribués.

À partir de ces travaux, nous avons trouvé peu de solutions utilisant la matérialisation de vues pour résoudre les problèmes de requêtes massives. Nous avons pris en compte, pour notre solution, ce qui a été fait dans les ED sur les SGBR et considéré les stratégies dans

les environnements distribués. Dans ce papier, nous privilégions les méthodes sur le partage de données en utilisant les matérialisations de vues issues des SGBDR tout en considérant la nature même du *cloud computing* où l'élasticité est un point fort (stockage ou calcul). Les travaux sur la matérialisation de vues dans les ED sur système de fichiers distribués n'attirent pas beaucoup d'attention étant donné le coût que cela peu engendrer (Perriot et al. (2013)). Nous restons quand même convaincus que cette technique d'optimisation est très efficace pour l'analyse en ligne.

3 Stratégie de traitement des requêtes massives

3.1 Sélection de vues à matérialiser

Notre problème de la sélection de vues peut être formalisé comme suit : soit l'entrepôt de données DW tel que F représente la ou les tables des faits, $D = \{D_1, D_2, \ldots, D_*\}$ représente les dimensions et une charge de requêtes $Q = \{Q_1, Q_2, \ldots, Q_*\}$ exécutée sur DW ; comment sélectionner $MV = \{V_1, V_2, \ldots, V_*\}$ un ensemble de vues matérialisées capables de répondre à Q avec un coût et temps d'exécution réduit ? Pour nos travaux, nous basons notre sélection de vues sur les charges passées que l'entrepôt de données a reçues, nous récupérons notre première charge de requêtes depuis le journal des traitements. Cela nous permet d'avoir une charge de requêtes connues et qui est susceptible d'être réexécutée lors des futurs processus d'analyse. Cette phase de sélection de vues matérialisées est divisée en deux étapes : le regroupement des requêtes de la charge et le choix de la requête ou des requêtes représentative-s qui servira-ont à créer le ou les vues matérialisées.

3.1.1 Classification des requêtes

Description du problème

Nous avons en entrée une charge massive de requêtes et nous avons comme objectif de regrouper les requêtes similaires pour former des classes.

Préparation des données

Les traces d'exécutions des requêtes sont accessibles depuis le fichier journal du système de gestion de base de données. Une charge de requêtes donnée est alors considérée comme représentative si elle est collectée sur une période de temps. Sur cet ensemble de requêtes collectées, nous séparons chaque type de requête (interrogation, insertion, mise à jour, calcul…) et nous nous intéressons particulièrement aux requêtes de sélection (décisionnelles, analyses, rapports…). Nous pouvons décrire une requête décisionnelle Q comme suit $Q = \Pi_{A,S}(\sigma_R(F \bowtie D_1 \bowtie D_2 \bowtie \ldots \bowtie D_*))$ où A est l'ensemble d'attributs des dimensions D_i qui sont présents dans chaque requête Q_j ; S est l'ensemble des mesures agrégées depuis la table des faits F et R les prédicats sur les attributs des dimensions. Nous transformons cette charge de requêtes en une matrice où chaque ligne représente une requête Q_j (TAB. 1) et chaque colonne les attributs A_k. La valeur M_{jk} est égal à 1, si A_k est présent dans Q_j et prend la valeur 0 sinon (TAB. 2) .

Q_1	SELECT A2, A3, A6 FROM F,D1,D2 WHERE F.A1 = D1.A1 AND F.A1= D2.A1 GROUP BY ROOLUP
Q_2	SELECT A4, A7 FROM F,D1,D2 WHERE F.A1 = D1.A1 AND F.A1= D2.A1 GROUP BY F.A1
Q_3	SELECT A2, A5, A6 FROM F,D1,D2 WHERE F.A1 = D1.A1 AND F.A1= D2.A1 GROUP BY ROOLUP
Q_4	SELECT A3, A4, A7 FROM F,D1,D2 WHERE F.A1 = D1.A1 AND F.A1= D2.A1 GROUP BY F.A1

TAB. 1 – *Exemple de requêtes SQL*

	A2	A3	A4	A5	A6	A7
Q_1	1	1	0	0	1	0
Q_2	0	0	1	0	0	1
Q_3	1	0	0	1	1	0
Q_4	0	0	1	1	0	1

TAB. 2 – *Matrice binaire d'usage des attributs dans les requêtes*

Classification

La classification des requêtes similaires est une part importante de ce travail pour obtenir un sous-ensemble $C = \{C_1, C_2, \cdots, C_*\}$ où chaque C_i est une classe de requêtes. Il existe une pléthore d'algorithmes de classifications, nous avons choisi une implémentation de *proximus* Koyutürk et al. (2005) [4], qui nous fournit un temps de traitement linéaire et proportionnel aux nombres de requêtes (figure 2) par rapport à la nature de nos données et le nombre de requêtes qu'on traite.

L'algorithme *proximus* prend en entrée la matrice binaire M d'usage des attributs pour les requêtes, que nous avons préparée précédemment à partir des requêtes (TAB 1), et le radius Rd qui est une valeur entière utilisée pour comparer la similarité entre les vecteurs de M. Le résultat obtenu est une classification des requêtes selon leur degré de similarité où le nombre de classes varie selon Rd.

Nous varions la valeur de Rd, selon la qualité du regroupement, afin d'avoir une plus grande flexibilité. Nous avons comme contrainte $Rd > 1$ car si $Rd <= 1$, le nombre de classes C obtenu est égal au nombre de requêtes Q_j de la charge. Pour notre approche nous initialisons la valeur de $Rd = 2$. Pour choisir le radius nous calculons une mesure de similarité entre les requêtes de chaque cluster C avec le coefficient de similarité de *Jaccard* (Jaccard (1901)) et ainsi définir graduellement le bon Rd si la moyenne des coefficients de similarité intra clusters ($\wp$) est < 0.7 (algorithme 1).
Nous avons choisi le coefficient de similarité de *Jaccard* car nous souhaitons calculer le rapport entre l'intersection de chaque requête et l'ensemble des attributs de la classe de requêtes.

Soit $C_i = \{Q_1, Q_2, \ldots, Q_*\}$ un cluster $\in C$ de Q_j requêtes similaires

$$Jaccard(C_i) = \frac{|Q_1 \cap Q_2 \cap \ldots \cap Q_*|}{|Q_1 \cup Q_2 \cup \ldots \cup Q_*|} \quad (1)$$

3.1.2 Requêtes représentatives et sélection de vues matérialisées

La requête représentative (δ) est construite à partir d'un ensemble d'attributs présents dans Q. δ doit répondre à au moins une des trois possibilités suivantes :

4. http ://compbio.case.edu/koyuturk/software/proximus/

Algorithme 1 : Radius adaptatif pour clustering avec proximus

Input : Q Charge de requêtes
Output : C Classes de requêtes similaires
$M_{jk} \leftarrow$ Matrice de $A_k \in Q_j$;
$Rd \leftarrow 2$;
$\wp \leftarrow 0.0$;
do
  $C \leftarrow Proximus(M_{jk}, R)$;
  $N \leftarrow$ Nombre de $C_i \in C$;
  $Rd \leftarrow Rd + 1$;
  $\wp \leftarrow \frac{\sum_{i=0}^{N} Jaccard(C_i)}{N}$;
while $\wp > 0.7$;

Clusters	Requêtes	
C_1	Q_1	Q_3
C_2	Q_2	Q_4

TAB. 3 – *Classification des exemples de requêtes (TAB. 1) avec* $Rd = 2$

— $\delta \equiv Q_j$ si les réponses de δ sont les réponses de Q_j
— $\delta \subseteq Q_j$ si la réponse de δ correspond une partie des projections de Q_j
— $Q_j \subseteq \delta$ si la réponse de Q_j est une projection de δ

Dans le domaine de la fouille de données, cela peut se comparer à la recherche du centroïd d'un cluster, mais dans notre cas nous souhaitons un δ construit à partir des attributs des requêtes de C_i. Lors de cette phase, nous étions amenés à résoudre le problème du nombre idéal d'attributs pour notre requête représentative. Naïvement, prendre l'union ou l'intersection des attributs qui composent C_i serait plus simple, mais imposerait dans le cas de l'union un δ très complexe et pour l'intersection un δ qui ne couvre pas toutes les requêtes de C_i. Les deux cas ne nous conviennent pas, car on cherche un δ qui couvre toutes les requêtes sans prendre l'union.

Pour définir δ nous avons :

1. Construit la matrice d'affinités (TAB. 4) des attributs pour chaque classe. Notre programme calcule combien de fois chaque couple d'attributs apparaît dans une requête, en prenant C_i en entrée. Nous avons appelé ce résultat la *contribution* ;
2. Récupéré le nombre moyen d'attributs ($\bar{A}$) composant chacune des requêtes de C_i ;
3. Calculé la somme des *contributions* pour chaque combinaison de $\bar{A}$ parmi tous les attributs qui compose C_i.

Pour notre exemple de charge (TAB. 1) et de classes (TAB. 3), les attributs qui se retrouveront dans δ seront $\{A2, A3, A6\}$, avec $\bar{A} = 3$, car la *contribution*($A2$, $A3$, $A6$) dans C_1 sera la plus élevée.

Comme cela est décrit dans la section 2, la sélection de vues à matérialiser est un problème complexe. Pour notre part, nous pouvons réduire le nombre de vues candidates sur lesquelles le choix sera fait, en prenant les requêtes représentatives de chaque classe comme vue susceptible

	A2	A3	A5	A6
A2	1	1	1	2
A3	1	1	0	1
A5	1	0	1	1
A6	2	1	1	1

TAB. 4 – *Matrice d'affinité des attributs pour le cluster* $C1$

d'être matérialisée. Pour notre exemple de charge (TAB 1) la vue matérialisée pour le premier regroupement (TAB 3) contiendra les attributs de δ de C_1.

4 Évaluation de notre approche

Dans cette section, nous décrivons les expérimentations que nous avons menées afin de valider notre approche pour le traitement d'une charge de requêtes massives dans les très grands entrepôts de données, basés sur *MapReduce*. Nous avons développé chaque phase de notre approche en *Python* [5] pour la portabilité et la disponibilité de certaines librairies qui nous sont nécessaires. Nous tenons à préciser que, dans cette première expérimentation nous n'avons mesuré que les coûts d'exécution.

4.1 Environnement d'expérimentations

Les expérimentations, que nous avons menées ont été exécutées sur des machines Dell Precision T1700 avec intel Xeon e3 (3.1Ghz) CPU et 16 Go RAM. Nous avons utilisé la version 2.3 de Hadoop avec Hive 0.13 pour l'entreposage de données et *HiveQL* et *Spark SQL* [6] pour l'interrogation de l'entrepôt de données.

4.2 Jeu de données

Entrepôt de données : L'approche que nous présentons dans cet article a été testée sur le banc d'essai TPC-DS (Poess et al. (2007)). Cependant, vu la limite de stockage de nos machines, nous n'avons généré que 1 To de données (TPC-DS scale factor 1000).

Charge de requêtes : Celle que nous avons utilisée est extraite de TPC-DS. Nous avons intentionnellement choisi nos *templates* de requêtes parmi les 99 possibles pour ne pas biaiser nos résultats. Nous avons inclus des requêtes contenant des jointures, ainsi que des requêtes contenant un ou plusieurs niveaux d'agrégation. Nous avons catégorisé ces requêtes en deux lots :

— requêtes d'interrogations interactives : Q-19, Q-42, Q-55, Q-52
— requêtes de rapport : Q-3, Q-7, Q-27, Q-43

Dans le générateur de requête de TPC-DS, nous pouvons générer α requêtes différentes pour chaque *template* de requête, par exemple pour $\alpha = 5$, Q-19 sera générée 5 fois avec des prédicats différentes. Pour cette évaluation nous avons généré :

5. https ://www.python.org/
6. http ://spark.apache.org/sql/

pour $\alpha = (5, 10, 20, 40, 80, 160, 320)$ sur les huit *templates* choisis, cela nous donne (40, 80, 160, 320, 640, 1280, 2560) requêtes.

4.3 Protocole d'expérimentation

Les expérimentations que nous avons menées ont été réalisées selon les étapes suivantes :

1. Nous avons généré plusieurs requêtes pour tester la robustesse et l'efficacité de notre algorithme de classification. Nous avons ensuite récupéré les classes de requêtes qui nous serviront à construire les requêtes représentatives δ.
2. Nous avons matérialisé chaque δ sous forme de *RDD (Resilient Distributed Dataset)* (Zaharia et al. (2012)) qui nous permet, selon le cas, de matérialiser les vues en mémoire ou sur *HDFS* si la taille de la vue est importante.
3. Nous avons évalué notre implémentation en récupérant les coûts d'exécution de chaque requête de notre charge (section 4.2), sans l'utilisation des *RDD* et avec. Ce coût d'exécution renvoi une valeur de calcul logique du coût de la lecture des données depuis le *HDFS*, plus le coût de chaque jointure, sachant que *Hive* nous propose de calculer ce coût sur la base des tâches *MapReduce*.

Définition RDD :

Un RDD est une collection de données partitionnée, distribuée et tolérante aux fautes en lecture seule, qui ne peut être créée que par des opérations déterministes : soit à partir de données présentes dans un stockage stable, soit à partir d'autres RDD. Un RDD peut être stocké en mémoire ou sur disque[7]. Dans un environnement *Hadoop2* le RDD est manipulé dans des *conteneurs* qui encapsulent mémoire vive et processeurs. Ceci permet un traitement distribué.

Coût d'exécution :

Soit une tâche *MapReduce* $t \leftarrow \langle p, d, r, c \rangle$ qui traite un programme p écrit dans un language de programmation ou en *HiveQL* sur des données d, d'un entrepôt de données sous *Hive* en utilisant les ressources r, ensemble des nœuds de calcul avec les configurations c de l'environnement *Hadoop*.

Le coût d'exécution ($perf$) est ici une estimation F, donnée par le système, qui prend comme paramètres le temps d'exécution de p et les successions de tâches *Map* et *Reduce* nécessaires, la taille de d utile pour le traitement, le nombre de r disponibles et la configuration c où l'on peut trouver la taille de mémoire allouée pour chaque tâche.

$$perf = F(p, d, r, c) \tag{2}$$

4.4 Résultats et discussions

La première étape de notre stratégie consiste à la construction des classes de requêtes similaires issues de TPC-DS. La figure 2 montre le temps de regroupement des requêtes similaires

7. https ://spark.apache.org/docs/latest/programming-guide.html#resilient-distributed-datasets-rdds

Requêtes TPC-DS	Coût d'exécution
Q3	8 023 323
Q7	8 098 600
Q19	9 640 061
Q27	9 670 658
Q42	8 110 890
Q43	8 210 786
Q52	8 260 130
Q55	8 289 797

TAB. 5 – *Coût d'exécution des requêtes*

RDD	Coût de création
RDD_1	8 110 875
RDD_2	8 120 793
RDD_3	8 206 098
RDD_4	9 313 364
RDD_5	14 491 932

TAB. 6 – *Coût de création des* RDDs

par rapport au nombre de requêtes contenues dans la charge. Ce graphique nous montre que le temps de classification est linéaire par rapport au nombre de requêtes. Ce qui est important à noter est que cette phase, nécessaire pour la suite, est réalisable en temps raisonnable. Nous avons obtenu cinq classes de requêtes avec lesquelles nous avons choisi de matérialiser chaque requête représentative.

Le tableau 5 nous détaille le coût moyen $perf$ (section 4.3) de chaque requête issue des *templates* sources que nous avons extraits de TPC-DS. Le tableau 6 illustre le coût de création de chaque *RDD*, en termes de $perf$, issu des δ de chaque classe de requêtes. Les coûts d'exécution élevés peuvent s'expliquer par, les ressources r et les configurations c que nous avions à disposition pour nos expérimentations.

Pour le traitement d'une nouvelle charge de requêtes Q', nous analysons chaque $Q'_j \in Q'$ pour déterminer à quelle classe C_i, construite avec la première charge, elle appartient. Ainsi nous déterminons quel RDD peut éventuellement répondre à une ou toutes les parties de Q'_j. La réécriture Q'_j est faite de telle sorte que le RDD correspondant y est inclut et les attributs présents dans le RDD soient utilisés à la place des attributs de la table. Seules les données manquantes sont récupérées des tables concernées. Le tableau 7 indique les réécritures possibles pour chaque requête issue des *templates* choisis (section 4.2). Nous avons intégré à notre prototype un programme de réécriture, car *Hive* ne dispose pas encore d'optimiseur de requêtes capable de modifier une requête pour utiliser des données intermédiaires. Le tableau 8 montre les résultats des $perf$ de la nouvelle charge sur *Hive* et les résultats obtenus avec notre approche. Nous pouvons constater une très nette amélioration des $perf$, car en regroupant les données partagées nous avons réduit la taille de d intervenant dans le calcul et avec le traitement en mémoire, de certaines données, nous avons aussi réduit le temps d'exécution de p.

À travers ces résultats, nous observons que l'utilisation de vues matérialisées apporte des améliorations sur le traitement massif des processus décisionnels. Ceci est dû au fait que certains *RDD* moins volumineux sont chargés directement en mémoire et pour les plus volumineux les données sont co-localisées, ce qui réduit considérablement le coût d'exécution $perf$. La réécriture de requêtes joue aussi un rôle important pour indiquer au système un meilleur chemin pour accéder aux données. Notre approche est implémentée sous forme de prototype, plusieurs verrous restent à résoudre comme l'utilisation de RDD multiple dans une réécriture ou le rafraichissement des RDD stockés sur disque, cependant les résultats que nous avons obtenus nous donnent une piste sérieuse à creuser.

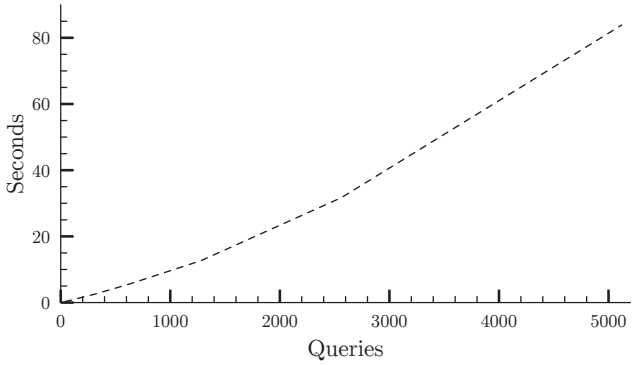

FIG. 2 – *Préparation des données et classification des requêtes générées à partir de TPC-DS*

	Réécriture (RW)				
Requêtes	RDD_1	RDD_2	RDD_3	RDD_4	RDD_5
Q3	-	RW	-	-	-
Q7	-	-	-	-	RW
Q19	-	-	-	RW	-
Q27	-	-	-	-	RW
Q42	RW	-	-	-	-
Q43	-	-	RW	-	-
Q52	-	RW	-	-	-
Q55	-	-	-	RW	-

TAB. 7 – *Réécriture des requêtes en utilisant les RDDs*

	Coût d'exécution	
Requêtes TPC-DS	Exécution sur Hive	Exécution avec réécriture
Q3	8 023 323	9 112
Q7	8 098 600	2 692 395
Q19	9 640 061	311 724
Q27	9 670 658	2 169 256
Q42	8 110 890	42 383
Q43	8 210 786	1 090 764
Q52	8 260 130	9 321
Q55	8 289 797	511 272

TAB. 8 – *Comparaison des coûts d'exécution*

5 Conclusion et perspectives

Dans cet article, nous avons proposé une stratégie pour le traitement massif des processus décisionnels dans les entrepôts de données utilisant *MapReduce*. Notre approche propose une nouvelle méthode pour résoudre le problème de sélection de vues matérialisées. À l'aide de l'algorithme *proximus*, nous avons classé la charge de requêtes selon la similarité des requêtes qui la compose, et à partir de chaque classe, nous avons réduit le nombre de vues candidates à la matérialisation en construisant, nous même, la requête représentative, qui deviendra la requête de construction de la vue matérialisée. Dans notre prototype, nous avons matérialisé ces vues sous forme de *RDD* qui peuvent être distribués en mémoire ou sur disque selon leurs tailles ce qui nous permet d'obtenir un coût d'exécution très réduit. Cela a été démontré par les résultats de nos expérimentations qui améliorent de plus de 50% le coût d'exécution.

Nous prévoyons pour la suite de nos travaux, de développer un processus de réécriture de requêtes, qui pour l'instant est basé sur la détermination de classe pour une nouvelle charge. Nous envisageons également de mener des expérimentations à l'échelle, car actuellement nous profitons des avancées de *Hadoop 2* qui nous a permis la parallélisation des traitements.

Références

Ahuja, M., C. C. Chen, R. Gottapu, J. Hallmann, W. Hasan, R. Johnson, M. Kozyrczak, R. Pabbati, N. Pandit, S. Pokuri, et al. (2009). Peta-scale data warehousing at yahoo ! In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pp. 855–862. ACM.

Baralis, E., S. Paraboschi, et E. Teniente (1997). Materialized views selection in a multidimensional database. In *VLDB*, Volume 97, pp. 156–165.

Bello, R. G., K. Dias, A. Downing, J. Feenan, J. Finnerty, W. D. Norcott, H. Sun, A. Witkowski, et M. Ziauddin (1998). Materialized views in oracle. In *VLDB*, Volume 98, pp. 24–27.

Boukorca, A., L. Bellatreche, et A. Cuzzocrea (2014). Slemas : an approach for selecting materialized views under query scheduling constraints. In *Proceedings of the 20th International Conference on Management of Data*, pp. 66–73. Computer Society of India.

Chaudhuri, S., U. Dayal, et V. Narasayya (2011). An overview of business intelligence technology. *Communications of the ACM 54*(8), 88–98.

Cohen, J., B. Dolan, M. Dunlap, J. M. Hellerstein, et C. Welton (2009). Mad skills : new analysis practices for big data. *Proceedings of the VLDB Endowment 2*(2), 1481–1492.

Dean, J. et S. Ghemawat (2004). Mapreduce : Simplified data processing on large clusters. In *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, pp. 137–150.

Gacem, A. et K. Boukhalfa (2013). Very large workloads based approach to efficiently partition data warehouses. In *Modeling Approaches and Algorithms for Advanced Computer Applications*, pp. 285–294.

Goldstein, J. et P.-Å. Larson (2001). Optimizing queries using materialized views : a practical, scalable solution. In *ACM SIGMOD Record*, Volume 30, pp. 331–342. ACM.

Gupta, H. et I. S. Mumick (1999). Selection of Views to Materialize Under a Maintenance Cost Constraint. In *Database Theory - ICDT'99*, pp. 453–470. Springer.

Herodotou, H. (2012). *Automatic tuning of data-intensive analytical workloads*. Ph. D. thesis, Duke University.

Herodotou, H. et S. Babu (2013). A what-if engine for cost-based mapreduce optimization. *IEEE Data Eng. Bull. 36*(1), 5–14.

Isard, M., V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, et A. Goldberg (2009). Quincy : fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pp. 261–276. ACM.

Jaccard, P. (1901). *Distribution de la Flore Alpine : dans le Bassin des dranses et dans quelques régions voisines*. Rouge.

Jian, Y., K. Kamalakar, et Q. Li (1997). Algorithms for Materialized view design in data warehousing environnement. *VLDB 97*, 25—-29.

Kotidis, Y., Y. Kotidis, N. Roussopoulos, et N. Roussopoulos (1999). DynaMat : a dynamic view management system for data warehouses. *Proc. 1999 {ACM} {SIGMOD} international conference on Management of data*, 371–382.

Koyutürk, M., A. Grama, et N. Ramakrishnan (2005). Compression , Clustering , and Pattern Discovery in Very High-Dimensional Discrete-Attribute Data Sets. *Knowledge and Data Engineering, IEEE Transactions on 17*(4), 447–461.

Marcus, R. et O. Papaemmanouil (2016). Wisedb : A learning-based workload management advisor for cloud databases. *CoRR abs/1601.08221*.

Nykiel, T., M. Potamias, C. Mishra, G. Kollios, et N. Koudas (2010). Mrshare : sharing across multiple queries in mapreduce. *Proceedings of the VLDB Endowment 3*(1-2), 494–505.

Perriot, R., J. Pfeifer, L. d'Orazio, B. Bachelet, S. Bimonte, et J. Darmont (2013). Modèles de coût pour la sélection de vues matérialisées dans le nuage, application aux services amazon ec2 et s3. In *9èmes journées francophones sur les Entrepôts de Données et l'Analyse en ligne (EDA 13), Blois*, Volume B-9 of *Revue des Nouvelles Technologies de l'Information*, Paris, pp. 53–68. Hermann.

Poess, M., R. Nambiar, et D. Walrath (2007). Why you should run TPC-DS : a workload analysis. *. . . conference on Very large data bases*, 1138–1149.

Sellis, T. K. (1988). Multiple-query optimization. *ACM Transactions on Database Systems 13*(1), 23–52.

Thusoo, A., J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, et R. Murthy (2010a). Hive-a petabyte scale data warehouse using hadoop. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pp. 996–1005. IEEE.

Thusoo, A., Z. Shao, S. Anthony, D. Borthakur, N. Jain, J. Sen Sarma, R. Murthy, et H. Liu (2010b). Data warehousing and analytics infrastructure at facebook. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pp. 1013–1020. ACM.

Wang, G. et C.-Y. Chan (2013). Multi-query optimization in mapreduce framework. *Proceedings of the VLDB Endowment 7*(3), 145–156.

Zaharia, M., D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, et I. Stoica (2009). Job scheduling for multi-user mapreduce clusters. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-55*.

Zaharia, M., M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, et I. Stoica (2012). Resilient distributed datasets : A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pp. 2–2. USENIX Association.

Zhang, C., X. Yao, et J. Yang (2001). An evolutionary approach to materialized views selection in a data warehouse environment. *Systems, Man, and Cybernetics, Part C : Applications and Reviews, IEEE Transactions on 31*(3), 282–294.

Summary

This article deals with the optimization problem of the execution of massive analytical processing on distributed data warehouses (ED) where the number of simultaneous queries is counted by thousands. While taking as a starting point the techniques of optimization used in the undistributed context, we propose a new strategy of selection and storage of materialized views (MV) on distributed file system; then we handle the processing of the decisional queries workload by using the MV. Our approach plays a role of mediator between the users and the data warehouse to propose a better execution plan to their queries. The first results make us believe that in a distributed environment, our approach improves more than 50% the execution cost of a request compared to the system provided by default.