Conception physique d'un entrepôt de données distribuées basée sur K-means équilibré

Yassine Ramdane*, Omar Boussaid*
Nadia Kabachi**, 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. Le partitionnement horizontal est l'une des techniques les plus performantes pour améliorer l'exploitation de données sur les plateformes de traitements parallèles comme Hadoop et Spark. Dans les entrepôts de données distribués (EDD), l'opération la plus coûteuse est la jointure en étoile qui nécessite plusieurs cycles MapReduce lors de son exécution. Dans ce papier, nous proposons une nouvelle stratégie de placement des données d'un entrepôt volumineux dans Hadoop, en se basant sur l'algorithme K-means équilibré (*K-means balanced*). Ce schéma de placement permet d'exécuter des opérations de certaines requêtes OLAP, dont la jointure en étoile, en une seule étape de Spark. Dans notre approche, nous prenons en compte les caractéristiques physiques du *cluster* et le volume des données. Pour évaluer notre proposition, nous avons effectué des expérimentations sur un *cluster* de 5 nœuds avec un entrepôt de données issu du banc d'essai TPC-DS. Les résultats obtenus montrent un gain de temps d'exécution, de certaines requêtes OLAP, allant jusqu'à 60% par rapport à d'autres approches existantes.

1 Introduction

Un entrepôt de données (ED) est une grande base de données conçue pour analyser les données. La taille d'un ED peut atteindre des dizaines de teraoctets (To). Il est modélisé à l'aide d'un schéma en étoile ou en flocons de neige, comprenant une ou plusieurs tables de faits et plusieurs dimensions.

Plusieurs techniques de partitionnement horizontal ont été utilisées pour améliorer les performances des entrepôts de données distribuées (EDD), comme l'équilibrage des charges de données ou les stratégies de placement et de distribution des bases de données (Zamanian et al., 2015; Lu et al., 2017). On peut distinguer deux types de partitionnement : statique et dynamique. Dans les techniques statiques, on effectue le placement et la distribution des données avant de traiter une requête en se basant soit sur le schéma de l'entrepôt (Eltabakh et al., 2011; Dittrich et al., 2010), soit sur une charge de requêtes stable (Arres et al., 2015). Dans

les techniques dynamiques, la distribution des données se fait au moment de l'élaboration du plan d'exécution de la requête (Zamanian et al., 2015; Tang et al., 2018). Certaines techniques, comme dans (Eltabakh et al., 2011; Dittrich et al., 2010), n'utilisent pas de charge de requêtes, mais plutôt des fichiers logs, cependant, elles ne sont pas adaptables dans le contexte des EDD et les requêtes de jointure en étoile.

Dans Hadoop, les bases de données sont constituées de tables, dont les données sont accessibles via un langage de requêtes tels que Hive-QL (Thusoo et al., 2009) ou Spark-SQL (Armbrust et al., 2015). Les tables, dans Hive ou dans Spark-SQL, sont similaires aux tables d'une base de données relationnelle. Dans un entrepôt, les données des tables sont sérialisées et chaque table possède un répertoire HDFS correspondant. Hadoop utilise des techniques de partitionnement et d'équilibrage de charges pour améliorer l'exécution des requêtes. Cependant, la distribution aléatoire des blocs, effectuée par HDFS, peut diminuer les performances des EDD, plus particulièrement avec les requêtes OLAP.

Les requêtes OLAP se composent de plusieurs opérations, comme le filtrage, la projection, la jointure et l'agrégation. Chaque opération peut s'exécuter lors de la phase *Map* ou celle de *Reduce*. Ainsi, chaque opération génère un coût d'E/S ou de CPU. L'opération de jointure est la plus coûteuse et peut générer un coût de communication considérable. Elle peut nécessiter $n-1$ ou $2*(n-1)$ cycles MapReduce, où n est le nombre des tables utilisées dans la requête. Pour minimiser le nombre de ces cycles et améliorer le traitement des requêtes, plusieurs travaux ont été proposés (Purdilă et Pentiuc, 2016; Brito et al., 2016). Cependant, autant que nous sachions, il n'existe aucun travail antérieur pouvant exécuter l'opération de jointure en étoile en une seule étape de Spark sur la plateforme Hadoop.

Dans ce papier, nous proposons un nouveau schéma de placement de données massives d'un EDD sur un cluster de nœuds homogènes, en utilisant l'équilibrage des charge de données sans se baser sur une charge de requêtes donnée. Nous prenons en compte les caractéristiques physiques du cluster et la distribution des clés primaires et étrangères des dimensions. Notre stratégie permet d'exécuter plusieurs opérations d'OLAP dont la jointure en étoile avec un seul cycle de Spark. Pour développer et évaluer notre approche, nous avons utilisé le langage Scala, la plateforme Hadoop-YARN avec Spark, le système Hive et le banc d'essai TPC-DS.

Le reste de cet article est structuré comme suit. La Section 2 résume les travaux liés aux différents types de jointure en MapReduce. Dans la Section 3, nous détaillons notre approche. Nous présentons nos expérimentations dans la Section 4 et nous concluons dans la Section 5.

2 Etat de l'art

La plupart des algorithmes de jointure en *MapReduce* reposent sur des techniques de partitionnement dynamiques, comme *repartition join* et *broadcast join* (Blanas et al., 2010), *multi-way join* (Afrati et Ullman, 2011) ainsi que le travail récent de (Kalinsky et al., 2016). Par contre, il y a peu de travaux qui utilisent des techniques statiques comme (Dittrich et al., 2010) et (Azez et al., 2015). Ce genre de technique nécessite la connaissance au préalable du type de traitement à faire. De plus, bien que le partitionnement dynamique effectué par les algorithmes de (Afrati et Ullman, 2011) et (Kalinsky et al., 2016) sont performants pour l'exécution de la jointure en étoile, cependant, le temps des transferts des données entre les nœuds durant la phase *Shuffle* peut être considérable, surtout dans le cas des dimensions de grande taille.

D'autre part, la technique de réplication totale de certaines dimensions, utilisée dans les travaux de (Blanas et al., 2010) et (Abouzeid et al., 2009), est une méthode inappropriée pour les tables de grande taille. Ainsi, la méthode de pré-jointure effectuée par l'approche JOUM (Azez et al., 2015), n'est pas une solution fléxible pour les entrepôts de données massives et peut occuper un espace disque trop important. D'autres travaux, comme (Purdilă et Pentiuc, 2016; Brito et al., 2016), ont proposé des méthodes pour réduire le nombre des cycles *MapReduce* dans l'opération de jointure. Telle que la stratégie de Purdilă et Pentiuc (2016) qui exécute la jointure en étoile en deux itérations *MapReduce* ; Brito et al. (2016) proposent deux algorithmes : *Spark Broadcast Join (SBJ)* et *Spark Bloom-Filtered Cascade Join (SBFCJ)* qui permettent de minimiser le coût de communication pour cette opération.

Spark-SQL utilise aussi par défaut le *Hash-BroadCast join (HBJ)* si les dimensions sont de faible taille. Dans ce cas, la jointure en étoile s'exécute dans la phase *Map*, sinon, Spark-SQL utilise le Shuffle join (c.à.d., *repartition join* (Blanas et al., 2010)) qui nécessite plusieurs cycles de Spark. Zamanian et al. (2015) utilisent un schéma de partitionnement dynamique implémenté dans un système de traitement des bases de données parallèles, appelé "*predicate-based reference*". Leur stratégie assure automatiquement la localité des données en co-partitionnant les tables qui partagent la même clé de jointure dans le même "*Bulk*" (partition).

Notre stratégie de placement est une technique de partitionnement statique comme (Dittrich et al., 2010). Nous traitons le problème des données dupliquées, comme dans (Zamanian et al., 2015), en utilisant l'algorithme de *clustering* "*K-means Balanced*". Avec notre technique et indépendamment de la charge de requêtes utilisée, nous pouvons exécuter le filtrage (c.à.d., prédicats dans la clause *Where*), la projection et la jointure en étoile en un seul cycle de Spark. Nous avons utilisé le moteur de traitement parallèle Spark et la plateforme Hadoop-YARN. L'ED utilisé est en schéma en étoile. Nous avons employé la technique de compartiment de données (*Bucketing*), qui existe en Hive et Spark-SQL, appelée *Sort-Merge-Bucket (SMB) join*.

3 L'approche proposée

Notre approche consiste à construire des fragments horizontaux des tables de faits et de dimensions, puis de les distribuer de façon équilibrée sur les différents nœuds d'un cluster, afin d'exécuter la jointure en étoile localement et en un seul cycle de Spark. Nous disposons au préalable du schéma de l'ED et des caractéristiques physiques du cluster. Notre démarche est composée de deux phases : (1) construction des *buckets* des tables de faits et de dimensions ; (2) placement des *buckets* qui partagent la même clé de jointure dans le même nœud. La Fig. 1 montre les étapes de notre approche. Dans ce qui suit, nous détaillons notre solution en commençant par la formalisation de notre problème.

3.1 Formulation du problème

Soit un schéma en étoile d'un ED, E={F, $D1$, $D2$,..,Dk}, tel que F est la table des faits et Dd, $d \in 1..k$, sont les dimensions. Notons par FK={fk_1, fk_2,..,fk_k}, l'ensemble des clés étrangères dans F en provenance de différentes dimensions, et par PK={pk_1, pk_2,..,pk_k}, l'ensemble des clés primaires de différentes dimensions. Nous symbolisons par "$index$" la clé de *bucketing* qui va servir à partitionner les tables de faits et de dimensions, tel qu'"$index \in$

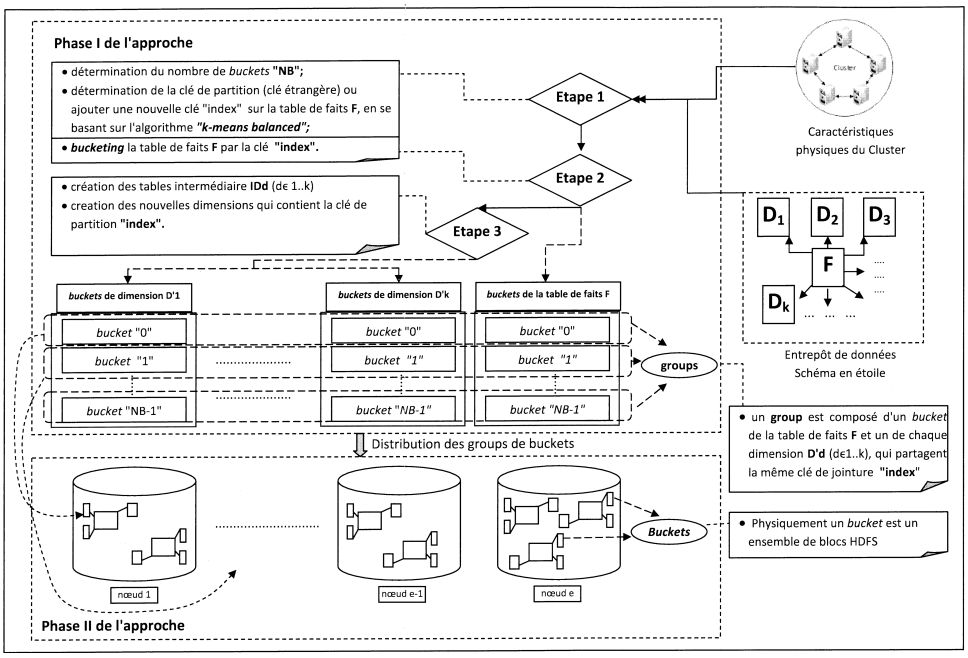

FIG. 1 – *les étapes de l'approche*

FK", ou bien "$index$" pourrait être une nouvelle colonne à ajouter à toutes les tables de l'ensemble E. Nous notons par NB le nombre des fragments (*buckets*) construits. Nous notons aussi par $CF = \{bucketF_0, bucketF_1, ..., bucketF_{NB-1}\}$, l'ensemble des *buckets* de la table des faits F, par $CDd = \{bucketDd_0, bucketDd_1, ..., bucketDd_{NB-1}\}$, ceux des dimensions, et par $W_{CF} = \{\|bucketsF_0\|, \|bucketF_1\|, ..., \|bucketF_{NB-1}\|\}$ l'ensemble des tailles des *buckets* de la table des faits F et $W_{CDd} = \{\|bucketDd_0\|, \|bucketDd_1\|, ..., \|bucketDd_{NB-1}\|\}$, $d \in 1...k$, celles des dimensions. Nous notons par "$group$", l'ensemble des *buckets* qui partagent la même clé de jointure "$index$", composé d'un *bucket* de F et un *bucket* de chaque dimension Dd, $d \in 1..k$. N=$\{n_1, n_2, .., n_e\}$ est l'ensemble des nœuds du *cluster*.

Notre objectif est de savoir comment choisir la clé $index$ et le nombre NB, pour obtenir des tailles plus ou moins égales des *buckets* de l'ensemble W_{CF} et W_{CDd}, $d \in 1..k$, et comment distribuer ces *buckets* sur les nœuds du cluster afin d'exécuter des opérations d'OLAP dont la jointure en étoile localement et en un seul cycle de Spark. Dans ce que suit, nous détaillons notre approche.

3.2 Construction des *Buckets*

Cette phase est composée de trois parties : (1) détermination du nombre de *buckets* "NB" et de la clé de partitionnement "$index$" ; (2) construction des *buckets* de la table des faits "CF" ; et (3) construction des *buckets* des dimensions "CDd, $d \in 1..k$".

3.2.1 Détermination du nombre *NB* et de la clé *index*

Pour déployer notre stratégie de partitionnement, la détermination du nombre idéal NB et de la clé de partition $index$, est une tâche primordiale.

a) Détermination du nombre NB : nous devons sélectionner celui-ci comme suit :

$$NB \in [min\_NB, .., max\_NB] \quad (1)$$

tel que min_NB est le nombre minimum des *buckets*, et max_NB le nombre maximum. Pour déterminer ces valeurs, nous employons les règles suivantes :

– **Règle 1**. Pour paralléliser notre traitement, il est préférable d'exploiter tous les CPU *cores*. Donc, au minimum, nous avons $min\_NB = N_{ct}$, tel que N_{ct} est le nombre total des CPU *cores* des *DataNodes* du *cluster*. C.à.d., N_{ct} est le nombre total des CPU *cores* affectés aux *executors* de Spark [1]. Notre démarche consiste à affecter pour chaque partition de Spark un CPU *cores* (dans notre cas une partition est un $group$, voir les notations dans la Section 3.1).

– **Règle 2.** Le choix d'un grand nombre NB peut pénaliser le traitement distribué, dû à l'augmentation du nombre d'E/S et à la taille des méta-données persistant en mémoire dans le *NameNode*. Pour palier à cela, le max_NB est déterminé comme suit :

$$max\_NB \leq \lfloor min\_NB \times \left(\frac{V_E}{V_M}\right) \rfloor \; et \; max\_NB \leq |T| \quad (2)$$

tel que V_E est la taille de l'entrepôt E, V_M est la somme des tailles mémoires (RAM) de tous les *DataNodes*, et T est la dimension ayant la plus faible taille dans E. Si $V_E < V_M$, on met $\frac{V_E}{V_M} = 1$. Notre raisonnement ici porte sur la première partie de la formule (2) ; ceci signifie que si nous avons un espace mémoire suffisant (c.à.d. $V_M \approx V_E$), $max\_NB \approx min\_NB$, nous pouvons exécuter des partitions de grande taille. Cependant, si la taille de la mémoire est faible ($V_M \ll V_E$), le max_NB augmente. Dans ce cas, le traitement de petites partitions est préférable. La seconde partie de l'équation, c.à.d. $max\_NB \leq |T|$, permet d'éviter l'obtention des *buckets* vides dans CF et CDd.

– **Règle 3.** Pour accélérer la sélection du meilleur nombre NB, nous commençons par exécuter des requêtes avec NB= N_{ct}, et à chaque fois, nous incrémentons la valeur NB, c.à.d. NB = NB + N_{ct}, jusqu'à ce que NB=max_NB, ou lorsque le temps d'exécution des requêtes augmente.

b) Détermination de la clé $index$: le premier objectif de déterminer la clé $index$ est de créer des "$groups$" de *buckets*. Nous distinguons deux techniques pour choisir la clé de jointure $index$: (1) nous choisissons la clé $index$ comme l'une des clés étrangères de la table des faits F, tel que le nombre de ses valeurs distinctes soit supérieur ou égal au nombre NB (pour éviter l'obtention des *buckets* vides) et pour qu'elle ait la meilleure distribution homogène de ses valeurs (c.à.d. la plus faible valeur du coefficient d'asymétrie [2]) ; (2) dans la deuxième technique, nous ajoutons une nouvelle clé de type entier à toutes les tables de l'entrepôt E. Cependant, afin d'obtenir une meilleure *classification* des enregistrements de F et déterminer

1. *un executor* est un processus intelligent permettant de lancer d'une façon autonome les tâches d'une application.
2. dans notre cas, le coefficient d'asymétrie Sk est calculé par la formule : Sk=$\frac{n}{(n-1)(n-2)} \sum(\frac{x_i-\mu}{\sigma})^3$
où n est le cardinal de l'ensemble $Dist$, x_i est l'élément i de $Dist$, σ est l'écart type de $Dist$ et μ est la moyenne de $Dist$, (c.f. https ://en.wikipedia.org/wiki/Skewness).

les valeurs d'$index$ pour minimiser l'écart-type des ensembles W_{CF} et W_{CDd}, $d \in 1..k$, nous appliquons l'algorithme de *clustering* "*K-means balanced*" (c.f. dans la Section 3.4).

3.2.2 Construction des *buckets* de la table des faits

La construction de l'ensemble CF est basée sur les deux paramètres : le nombre "NB" et la clé de partition "$index$". Quelque soit la technique utilisée pour choisir la clé $index$, nous appliquons la formule suivante pour construire les *buckets* de l'ensemble CF : $enregistrements\ du\ bucketF_i \equiv enregistrements\ de\ F\ qui\ ont\ la\ même\ valeur\ de\ la\ clé$ "$index$".

3.2.3 Construction des *buckets* des dimensions

Nous construisons à present les ensembles de *buckets*, CDd, $d \in 1..k$, pour les tables de dimensions. Nous distinguons deux techniques : (1) dans la première, nous la désignons par $FkKey$, nous partitionnons la dimension Dm, dont la clé étrangère fk_m vérifie les deux conditions expliquées dans la Section 3.2.1.b, et nous créons ses *buckets* en utilisant la même formule dans 3.2.2 (où nous remplaçons F par Dm). Cependant pour partitionner les Di, $i \in 1..k/m$, nous devons d'abord construire une table intermédiaire IDi, composée de deux colonnes, la première contenant la clé étrangère fk_i et la deuxième, contient une clé, notée fk_m ; ses valeurs sont calculées par la formule : $fk_m[j]\ modulo\ NB$, tel que les $fk_m[j]$ sont les valeurs de la clé étrangère fk_m dans F et $j \in 1..|F|$. Puis, nous supprimons tous les enregistrements dupliqués dans la table IDi. Enfin, nous faisons une jointure entre Di et IDi pour obtenir une nouvelle dimension $D'i$ qui contient la même clé de jointure de la dimension Dm. Et nous construisons l'ensemble $CD'i$, en utilisant la formule 3.2.2. La Fig. 2 donne un exemple comment créer une nouvelle dimension avec cette technique. (2) Dans la deuxième technique, notée par $NewKey$, après avoir ajouter la clé $index$ à la table des faits, en utilisant l'algorithme "*K-means balanced*" et la création de l'ensemble CF, nous devons ajouter la clé $index$ à toutes les dimensions pour créer les CDd, $d \in 1..k$. Pour ce faire, nous suivons ces deux étapes : (a) premièrement, nous construisons une table intermédiaire IDd pour chaque dimension Dd, $d \in 1..k$. IDd est composée de deux attributs, fk_d et $index$, tel que fk_d est la clé étrangère dans F qui correspond à la dimension Dd, et $index$ est la clé de partition ajoutée à F. La table IDd a la même taille, c.à.d. le même nombre d'enregistrements que F. Avant de faire la jointure entre IDd et Dd, pour obtenir une nouvelle dimension $D'd$ qui contient la clé de jointure $index$, nous supprimons tous les enregistrements dupliqués dans la table IDd; (b) nous construisons après les ensembles $CD'd$ en appliquant la même formule dans 3.2.2 (où nous remplaçons F par $D'd$). Notons que la taille d'une nouvelle dimension $D'd$ change selon la valeur du nombre NB et la façon de déterminer la clé $index$. La Fig. 3 montre un exemple de construction d'une nouvelle dimension et ses *buckets* avec la technique $NewKey$.

3.3 Placement des *Buckets*

Dans la phase II de notre approche, nous distribuons équitablement les $groups$ construits sur les nœuds du *cluster*. Formellement, si on note par $group_i = bucketF_i \uplus_{d=1}^{k} bucketD'd_i$, $i \in 0..NB-1$, nous commençons par placer le $group_0$ sur le nœud 1, le $group_1$ sur le nœud 2,..., et le $group_{p-1}$ sur le nœud e, tel que e=$p\ modulo\ NB$ et $p <= NB$. Nous recommençons l'opération de la même manière, nous plaçons le $group_p$ sur le nœud 1, le $group_{p+1}$ sur le

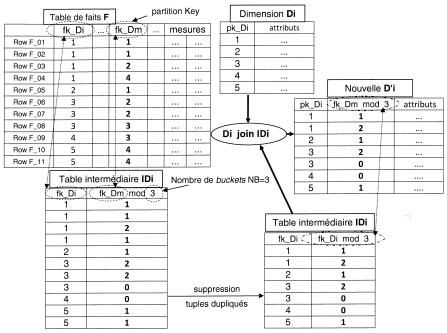

FIG. 2 – *Construction d'une nouvelle dimension selon la technique* $FkKey$

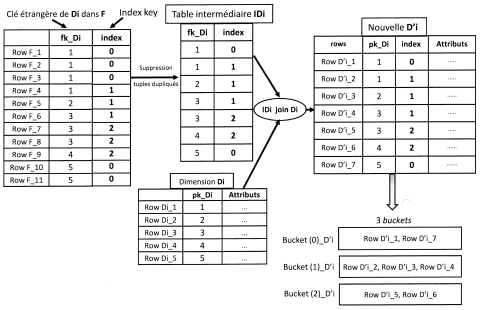

FIG. 3 – *Construction d'une nouvelle dimension selon la technique* $NewKey$ *et ses 3 buckets*

nœud 2,..., jusqu'au $group_{NB-1}$. Cette technique de placement permet d'exécuter toutes les opérations de jointure en étoile localement et avec un seul cycle de Spark.

3.4 Détermination de la Clé *Index*

Nous remarquons que la taille des *buckets* dépend des deux paramètres NB et $index$. Nous pouvons déterminer la valeur du NB en suivant les étapes de la Section 3.2.1. Cependant, le choix des valeurs de la clé $index$ est une tâche très difficile à gérer. Donc, puisque en général, les tailles des dimensions sont négligeables par rapport à la table des faits, nous cherchons comment minimiser l'écart-type de l'ensemble W_{CF}. Cependant, il y a un facteur important qui influe directement sur la taille des nouvelles dimensions construites. Celui-ci est la similarité des enregistrements des *buckets* de F. Pour palier à ce problème, nous proposons une solution approximative basée sur l'algorithme "*K-means Balanced*". Les étapes de notre méthode sont :

1. A partir de la table F, nous construisons la matrice *MV*, tel que :

$$MV = \begin{bmatrix} V_{FD11} & V_{FD21} & \cdot\cdot & V_{FDk1} \\ V_{FD12} & V_{FD22} & \cdot\cdot & V_{FDk2} \\ \cdot\cdot & \cdot\cdot & \cdot\cdot & \cdot\cdot \\ \cdot\cdot & \cdot\cdot & V_{FDdj} & \cdot\cdot \\ \cdot\cdot & \cdot\cdot & \cdot\cdot & \cdot\cdot \\ V_{FD1n} & W_{FD2n} & \cdot\cdot & V_{FDkn} \end{bmatrix}$$

ici, V_{FDdj} est la valeur de la colonne fk_d dans la ligne j et $n = |MV| = |F|$.

2. Une fois *MV* obtenue, nous construisons les NB *clusters* de *MV*. Donc, nous avons choisi l'algorithme *K-means balanced* (Malinen et Fränti, 2014) pour regrouper les vecteurs dans *MV*, pour deux raisons : la première est d'obtenir des tailles presque égales de *buckets* de CF ; la deuxième est de minimiser l'erreur quadratique moyenne (MSE) et d'augmenter la similarité interne des *clusters*. *MSE* est calculée comme suit :

$$MSE = \sum_{j=1}^{k} \sum_{X_i \in C_j} \frac{\|X_i - C_j\|^2}{n} \tag{3}$$

ici, les X_i sont des vecteurs de la matrice *MV*, les C_j sont les centres des groupes ou des clusters et $n = |MV|$.

Nous obtenons (n mod NB) *clusters* de taille $\lceil \frac{n}{NB} \rceil$, et NB-(n mod NB) *clusters* de taille $\lfloor \frac{n}{NB} \rfloor$.

3. Finalement, nous affectons les valeurs obtenues à l'issue du *clustering* aux valeurs de la clé $index$.

Dans notre technique de regroupement, nous n'avons pas inclus un autre facteur qui influe sur la taille des nouvelles dimensions. Celui-ci est le nombre d'attributs d'une dimension. Certaines dimensions peuvent avoir peu d'attributs alors que d'autres en ont plusieurs (voire des centaines). Cependant, avec les formats de stockage en colonne comme *Parquet* et *ORC*, seuls les attributs sollicités par les requêtes sont chargés en mémoire.

3.5 Transformation des requêtes

Dans notre approche, nous faisons quelques transformations aux niveaux des prédicats de jointure pour obtenir des résultats non erronés. Dans la technique $NewKey$, pour chaque $d \in 1..k$, le prédicat $F.fk_d = Dd.pk_d$ devient $F.index = D'd.index$, tel que $D'd$ est la nouvelle dimension construite. De plus, les deux conditions de jointures "$F.index = D'd.index$" et "$F.fk_d = D'd.pk_d$" ne doivent pas être dans la même requête. Pour la technique $FkKey$, le prédicat $F.fk_d = Dd.pk_d$ devient $F.fk_m \ modulo \ NB = D'd.fk_m$. Cette nouvelle condition de jointure augmente le coût du CPU suite à l'opération de $modulo$, comme nous le montrons dans les Sections 4.2 et 4.3.

4 Expérimentations

4.1 Configuration de l'Environnement

Dans cette Section, nous présentons les étapes d'implémentation de notre approche. Premièrement, nous générons l'ED en utilisant le banc d'essai TPC-DS. Nous stockons directement en HDFS *Parquet* format. Nous avons implémenté les deux phases de notre approche. Nous avons utilisé un *cluster* de 5 machines esclaves (*DataNodes*) et une machine maître (*NameNodes*), caractérisées par : des CPU Pentium I7 avec 8 cores, une mémoire vive de 16 GB et un disque dur de 1 TB. Nous avons installé sur tous les nœuds Hadoop-YARN V-2.9.1, Hive V-2.3.3, Spark V-2.3.2, le banc d'essai TPC-DS, le langage Scala et l'outil "SBT".

Nous avons configuré Spark comme suit : *spark.executor.instances*=10, *spark.executor.memory*=6 *GB*, *spark.executor.cores*=3 *CPU cores*, la taille des blocs HDFS est de 128 MB et le facteur de réplication est égal à 3. Pour chaque nœud, nous gardons 4 GB de mémoire et 2 CPU *cores* pour le "système d'exploitation", "*executors*", et pour "*Application Master*". Avec cette configuration, nous pouvons exécuter $10 \times 3 = 30$ tâches au même temps.

4.1.1 Génération des données

Nous adoptons l'application *spark-sql-perf*[3] en utilisant Scala et Spark, pour générer une partie de *DW*, composé d'une table des faits et de 9 dimensions. Suite aux limitations physiques de notre *cluster*, nous générons la table des faits $store\_sales$ par partition, où, nous avons choisi la clé étrangère ss_store_sk de la dimension $store$ comme clé de partition, puisque l'attribut ss_store_sk possède le moins de valeurs distinctes comparé aux clés des autres dimensions (voir Tab. 1).

3. disponible sur le site https ://github.com/databricks/spark-sql-perf.

Nom de la table	Entrepôt de données (*DW*)	
	nombre de tuples	taille en Parquet
$store\_sales$	2 879 995 413	142.6 GB
$customer$	12 000 000	607.8 MB
$customer\_address$	6 000 000	111.4 MB
$customer\_demographics$	1 920 800	7.4 MB
$item$	300 000	27.3 MB
$time\_dim$	86 400	1 126 KB
$date\_dim$	73 049	1 740 KB
$household\_demographics$	7 200	30.0 KB
$promotion$	1 500	76.0 KB
$store$	1 002	88.0 KB

TAB. 1 – *Caractéristiques de DW*

Nom	code de la requête
$Q1$	select dt.d_year, item.i_brand, item.i_brand_id, from date_dim dt, item, store_sales where dt.d_date_sk = store_sales.ss_sold_date_sk and store_sales.ss_item_sk = item.i_item_sk and item.i_manufact_id = 128 and dt.d_moy=11 limit 100;
$Q2$	select dt.d_year, dt.d_month_seq, item.i_brand, item.i_brand_id, item.i_class from date_dim dt, item, store_sales where dt.d_date_sk = store_sales.ss_sold_date_sk and store_sales.ss_item_sk = item.i_item_sk and item.i_manufact_id = 128 and dt.d_moy=11 limit 100;
$Q3$	select c_customer_id, c_first_name, c_last_name, c_preferred_cust_flag, c_birth_country, c_login , c_email_address, d_year, d_month_seq from customer, date_dim, store_sales where c_customer_sk = ss_customer_sk and ss_sold_date_sk = d_date_sk limit 100;
$Q4$	select a.ca_city, d.d_month_seq, i.i_brand from customer_address a, date_dim d, item i, store_sales s where a.ca_address_sk = s.ss_addr_sk and s.ss_sold_date_sk = d.d_date_sk and s.ss_item_sk = i.i_item_sk and i.i_manufact_id = 128 and d.d_moy=11 limit 100;
$Q5$	select a.ca_city, d.d_month_seq, i.i_brand from customer_address a, date_dim d, item i, store_sales s where a.ca_address_sk = s.ss_addr_sk and s.ss_sold_date_sk = d.d_date_sk and s.ss_item_sk = i.i_item_sk limit 100;
$Q6$	select c.c_customer_id, c.c_first_name, c.c_last_name, c.c_preferred_cust_flag, c.c_birth_country, c.c_login, a.ca_city, a.ca_state, a.ca_country, d.d_month_seq, d.d_date, i.i_brand, i.i_class, i.i_product_name from customer_address a, customer c, date_dim d, item i, store_sales s where a.ca_address_sk = s.ss_addr_sk and c.c_customer_sk = s.ss_customer_sk and s.ss_sold_date_sk = d.d_date_sk and s.ss_item_sk = i.i_item_sk limit 100;

TAB. 2 – *Les 6 requêtes sélectionnées*

4.1.2 Implémentation de l'Approche

Pour implémenter la première phase de notre approche, nous avons utilisé trois composants de Spark : $Dataframe$, $Dataset$ et $ArrayBuffer$. Pour placer les goupes des *buckets*, nous ne changeons pas la politique de placement par défaut d'HDFS comme (Eltabakh et al., 2011). Car la modification du nouveau framework API d'Hadoop V-2.x, est une tâche très difficile à effectuer. Notre stratégie est comme le *balancer* externe d'Hadoop.

4.2 Évaluation

Pour évaluer notre approche, nous générons un ED, noté *DW*, dont la taille est de 500 GB en format CSV (environ 143 GB en format compressé *Parquet*, voir les caractéristiques dans Tab. 1). Nous avons sélectionné puis adapté 6 requêtes de TPC-DS (voir Tab. 2). Nous supprimons "*Group by*" clause les agrégations, puisque cette opération s'exécute dans la phase *Reduce* et nécessite d'autre cycles (*stages*) de Spark. Les caractéristiques des requêtes sont : dans $Q1$, nous joignons 2 dimensions de faible taille avec la table des faits, ainsi nous sélectionnons quelques attributs avec l'utilisation de 2 filtres ; $Q2$ est comme $Q1$, mais avec la sélection de plusieurs attributs ; dans $Q3$, nous joignons 2 dimensions, dont une de large taille, avec la table de faits, sans utiliser des filtres ; la requête $Q4$ est comme $Q1$, avec l'ajout d'une dimension de grande taille $customer\_address$; $Q5$ est comme $Q4$, sans utiliser des filtres ; et $Q6$ est comme $Q5$, mais nous joignons 4 dimensions avec la table de faits.

Nous avons exécuté les 6 requêtes avec 4 approches : (1) SSH (partitionnement et distribution par défaut d'Hadoop et Spark), qui utilise *Shuffle Hash join* (*SH* join est comme *répartition join* de (Blanas et al., 2010)). Dans ce cas, nous désactivons le *Hash Broadcast Join* (*HBJ*) ; (2) l'approche SHB (partitionnement et distribution par défaut d'Hadoop et Spark), en activant *HBJ* ; (3) $SSMBO_{NB}$ est notre propre approche de partitionnement et de distribution avec la technique $NewKey$, qui utilise l'algorithme de *clustering K-means Balanced*, en activant le *Sort-Merge-Bucket join* "*SMB*" de Spark-SQL ; et enfin (4) $SSMBO'_{NB}$ qui est comme $SSMBO_{NB}$, mais avec la technique $FkKey$ pour le choix de clé de partitionnement [4]

Le nombre NB est sélectionné selon les recommandations de la Section 3.2.1. Dans SSH et SHB, nous mettons "$spark.sql.shuffle.partitions$" égale à 180. pour désactiver le *HBJ*

4. Notons que durant la génération des tables, nous avons partitionné la table des faits $store\_sales$ par la clé ss_store_sk qui a 1002 valeurs distinctes. Si nous avons partitionné par la clé "$ss\_customer\_sk$" qui a plus de 12 millions de valeurs, nous obtenons de mauvais résultats dans SSH et SHB.

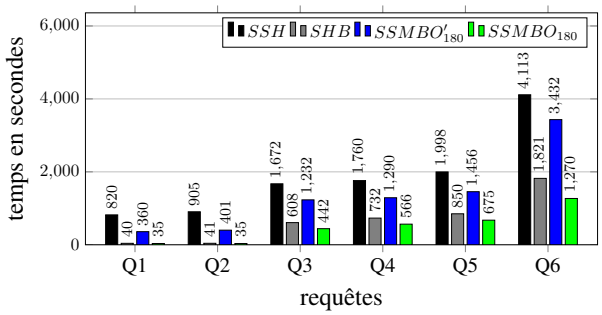

FIG. 4 – *Temps d'exécution des 6 requêtes*

en Spark, nous utilisons l'instruction : *Session.conf.set("spark.sql.auto Broad castJoinThreshold",-1)*. Avec notre configuration, nous pouvons exécuter 30 tâches en parallèle. Donc, min_NB = 30 et $max\_NB \leq \lfloor min\_NB \times \left(\frac{V_E}{V_M}\right) \rfloor = \lfloor 30 \times \left(\frac{500}{80}\right) \rfloor = 187$ (noter que $187 \leq (|T| = 1002)$). Où $V_E = 500GB$ en CSV format et V_M = $(16GB \times 5)$ = 80 GB. La Fig. 4 montre le temps d'exécution des 6 requêtes. Pour montrer l'impact de NB sur ces mauvais rèsultats, nous avons exécuté les 3 requêtes $Q1$, $Q3$ et $Q6$ avec 5 valeurs de NB, (30, 60, 90, 180, 360) (voir Fig. 5). Dans la Fig. 6, nous comparons la taille des dimensions dans les approches $SSMBO$ et $SSMBO'$, et le partitionnement par défaut de Spark et Hadoop (SSH ou SHB). Notons que cette taille concerne une seule réplication des blocs HDFS.

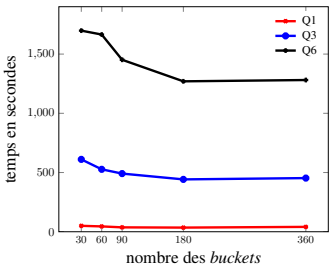

FIG. 5 – *L'impact de NB sur le temps d'exécution des requêtes*

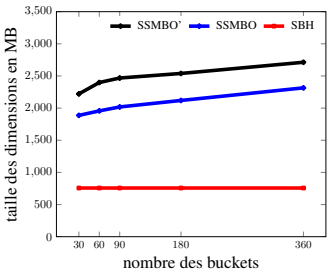

FIG. 6 – *L'impact de NB sur les tailles des dimensions*

4.3 Discussion

Concernant les résultats de la Fig. 4, notre approche $SSMBO$ a permis d'améliorer les temps d'exécution de 25% à 60% par rapport à SHB. Pour toutes les requêtes, des mauvais résultats sont obtenus par les 2 approches SSH et $SSMBO'$ (en appliquant la technique $FkKey$). Dans SSH, ceci est dû au taux élevé du *shuffle* durant la phase *Reduce*. Alors que dans $SSMBO'$, nous obtenons de mauvais résultats bien que nous ayons partitionné toutes les tables avec une seule clé tout en activant SMB join de SPARK-SQL. Dans ce cas, l'optimiseur *Catalyst* de Spark doit planifier d'autres cycles pour calculer la condition de jointure ($F.fk_m$ $modulo$ $NB = D'd.fk_m$), ce qui augmente le coût de CPU.

Dans $Q1$ et $Q2$, nous sélectionnons quelques attributs des dimensions faibles, ($item$ et $date\_dim$), avec 2 filtres, et puisque les données sont stockées en format *Parquet*, le taux d'amélioration de $SSMBO$ est juste entre 15% et 20% par rapport à SHB. Nous remarquons aussi, que les meilleurs résultats sont obtenus avec $Q3$ et $Q6$. Ceci est dû au fait d'avoir utilisé 2 dimensions de grande taille, ($customer$ et $customer\_address$), et la jointure en étoile devient alors lourde pour SHB. Les requêtes $Q3$ et $Q6$ sont exécutées respectivement en 4 et 7 *cycles* dans SHB, tandis que dans $SSMBO$ cela ne nécessite qu'un seul *cycle*. Donc, avec $SSMBO$, et comme nous avons partitionné la table de faits et toutes les dimensions avec la même clé "$index$", nous avons ainsi réussi à utiliser correctement le *SMB join* de Spark-SQL.

La Fig. 5 montre l'impact du nombre NB sur les performances des requêtes. Pour toutes les requêtes, les meilleurs résultats sont obtenus avec la valeur de $NB \in \{90, .., 180\}$, et cela montre la fiabilité de notre méthode pour déterminer le nombre NB. De plus, bien que la Fig. 6 montre qu'il y a un impact du nombre NB sur la taille des nouvelles dimensions, cependant, cela n'influe pas réellement sur l'espace disque, puisque, nous remarquons que le volume des données n'a augmenté qu'entre 2.5 et 2.8 fois par rapport aux tailles originales des dimensions. Notons par exemple, si nous répliquons la dimension $customer$ sur les différents nœuds du *cluster*, l'espace disque occupé par cette dimension se sera $5 \times 607.8 = 3\ 039MB$ en format parquet (voir Tab. 1). Nous remarquons aussi, avec notre algorithme "*K-means Balanced*", nous avons gagné entre 20 et 23% d'espace disque par rapport à $SSMBO'$.

5 Conclusion et travaux futurs

Dans cet article, nous avons présenté une nouvelle stratégie de placement de données volumineuses dans un entrepôt distribué sur un *cluster* de nœuds homogènes, en se basant sur l'algorithme de *clustering K-means balanced*. Notre approche permet d'exécuter plusieurs opérations de certaines requêtes OLAP dont la jointure en étoile avec un seul cycle de Spark. Nous pouvons améliorer davantage le temps d'exécution des requêtes, si on partitionne certaines tables par les attributs des prédicats les plus fréquemment utilisés.

Il y a plusieurs perspectives dont les plus importantes sont : (1) l'améliorer de notre conception à travers une charge de requêtes ; (2) l'ajout d'une partie dynamique pilotée par un Système Muli-Agents pour équilibrer les partitions générées lors les résultats intermédiaires.

Références

Abouzeid, A., K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, et A. Rasin (2009). Hadoopdb : an architectural hybrid of mapreduce and dbms technologies for analytical workloads. *Proceedings of the VLDB Endowment 2*(1), 922–933.

Afrati, F. N. et J. D. Ullman (2011). Optimizing multiway joins in a map-reduce environment. *IEEE Transactions on Knowledge and Data Engineering 23*(9), 1282–1298.

Armbrust, M., R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, et al. (2015). Spark sql : Relational data processing in spark. In *Proc. of the 2015 ACM SIGMOD Int Conf. on Management of Data*, pp. 1383–1394. ACM.

Arres, B., N. Kabachi, et O. Boussaid (2015). Optimizing olap cubes construction by improving data placement on multi-nodes clusters. In *Parallel, Distributed and Network-Based Processing (PDP), 2015 23rd Euromicro International Conference on*, pp. 520–524. IEEE.

Azez, H., M. H. Khafagy, et F. A. Omara (2015). Joum : An indexing methodology for improving join in hive star schema. *Int. J. Sci. Eng. Res 6*, 111–119.

Blanas, S., J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, et Y. Tian (2010). A comparison of join algorithms for log processing in mapreduce. In *Proceedings of the 2010 ACM SIGMOD Int. Conf. on Management of data*, pp. 975–986. ACM.

Brito, J. J., T. Mosqueiro, R. R. Ciferri, et C. D. de Aguiar Ciferri (2016). Faster cloud star joins with reduced disk spill and network communication. *Procedia Computer Sc. 80*, 74–85.

Dittrich, J., J.-A. Quiané-Ruiz, A. Jindal, Y. Kargin, V. Setty, et J. Schad (2010). Hadoop++ : making a yellow elephant run like a cheetah. *Proc. of the VLDB Endow. 3*(1-2), 515–529.

Eltabakh, M. Y., Y. Tian, F. Özcan, R. Gemulla, A. Krettek, et J. McPherson (2011). Cohadoop : flexible data placement and its exploitation in hadoop. *Proceedings of the VLDB Endowment 4*(9), 575–585.

Kalinsky, O., Y. Etsion, et B. Kimelfeld (2016). Flexible caching in trie joins. *arXiv preprint arXiv :1602.08721*.

Lu, Y., A. Shanbhag, A. Jindal, et S. Madden (2017). Adaptdb : adaptive partitioning for distributed joins. *Proceedings of the VLDB Endowment 10*(5), 589–600.

Malinen, M. I. et P. Fränti (2014). Balanced k-means for clustering. In *Joint IAPR International Workshops on Statistical Techniques in Pattern Recognition (SPR) and Structural and Syntactic Pattern Recognition (SSPR)*, pp. 32–41. Springer.

Purdilă, V. et Ş.-G. Pentiuc (2016). Single-scan : a fast star-join query processing algorithm. *Practice and Experience 46*(3), 319–339.

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. Sys. 78*, 287–301.

Thusoo, A., J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, et R. Murthy (2009). Hive : a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment 2*(2), 1626–1629.

Zamanian, E., C. Binnig, et A. Salama (2015). Locality-aware partitioning in parallel database systems. In *Proc. of the 2015 ACM SIGMOD Int. Conf. on Management of Data*, pp. 17–30.

Summary

Horizontal partitioning has been widely used to optimize query processing in distributed system such as Hadoop and Spark. In distributed data warehouses, the most expensive operation for OLAP queries is star join which requires many MapReduce cycles to perform it. In this paper, we propose new data placement in Hadoop based on K-means balanced algorithm. This schéma allows to perfom star join operation in only one Spark stage. In our technique, we take into account the physical characteristics of the cluster and the volume of data. To evaluate our approach, we conducted some experiments on a cluster of 5 nodes. Where, our approach has improved the execution time of some OLAP queries by 60% over some existing approaches.