Équilibrage de Distribution de Données d'une Base en Mémoire Parallèle Partitionnées par Intervalle

Djahida Belayadi*, Khaled-Walid Hidouci*
Ladjel Bellatreche** Carlos Ordonez***

*Laboratoire LCSI, Ecole Nationale Supérieure d'Informatique, Alger, Algérie
d_belayadi@esi.dz, w_hidouci@esi.dz
**LIAS/ISAE-ENSMA, 86960 Futuroscope, France
bellatreche@ensma.fr
https://www.lias-lab.fr/members/bellatreche
***Carlos Ordonez
Université de Houston, USA
carlos@central.uh.edu

Résumé. Grâce à la disponibilité de plus grandes capacités de mémoire principale, nous assistons à une présence des systèmes parallèles de bases de données gérées en mémoire offrant une performance accrue contrairement aux bases de données traditionnelles. Le partitionnement de données est une pré-condition de ces bases de données, car il permet d'améliorer de manière significative les performances de certaines requêtes (par ex. le cas de requêtes d'intervalle). Le partitionnement engendre un problème d'équilibrage de distribution des données. En conséquence, il peut contribuer à la dégradation de la performance de requêtes. Le groupe de Ganesan et al. a proposé un algorithme offrant un faible rapport entre les charges maximale et minimale des nœuds tout en exploitant des informations de la charge globale. Ces informations sont stockées dans une structure de données, appelée skip graphs nécessitant l'échange de (log p) messages entre les p nœuds de la machine parallèle lors du processus d'équilibrage. L'objectif de notre travail est de réduire le nombre de ces messages. Pour ce faire, nous proposons un vecteur de statistiques approximatives des partitions (VSP), où les nœuds et les clients ont une vue approximative sur la distribution de données. Le coût de maintenance de ce vecteur est quasiment nul. Notre approche est validée sur un cluster de 8 nœuds et 2 clients.

1 Introduction

L'explosion des technologies de l'Internet des Objets et des réseaux sociaux a récemment donné naissance à une nouvelle génération d'applications Big Data. Ces applications exploitent la richesse des données historiques provenant de sources de données géographiquement distribuées pour fournir des informations exploitables et en temps réel. Le traitement parallèle est crucial dans le traitement moderne des données volumineuses en raison du volume de données

et de la nécessité de traiter ces données plus rapidement. Les SGBD parallèles ont été et restent toujours des technologies importantes pour relever les défis mentionnés ci-dessus. L'augmentation rapide du grand flux de données a déclenché de nouveaux problèmes. La distribution biaisée de gros volumes de données dans une base de données résidentes en mémoire vive est l'un des défis auxquels nous sommes confrontés afin de garantir des performances optimales d'interrogation des données.

Le partitionnement par intervalle distribue les enregistrement d'une relation sur l'ensemble des partitions selon une ou plusieurs clés de partitionnement. Une exigence clé dans un tel système est que les données doivent être partitionnées uniformément sur tous les nœuds. Cette exigence est difficile à appliquer lorsque les données d'entrée sont biaisées. Le déséquilibre de données appelé "Data Skew"est un problème connu dans le partitionnement par intervalle où certaines partitions (nœuds) peuvent être plus chargées que d'autres. Dans ce cas, les approches de migration de données constituent une solution attrayante. Le surplus de données doit être déplacé de la zone surchargée à celle sous-chargée afin de satisfaire à l'exigence d'équilibrage de stockage.

Le mouvement des données doit être accompagné d'un changement dans les statistiques de la partition (bornes de la partition, charges des nœuds voisins, position du nœud le plus chargé et le moins chargé, etc.). Tous les nœuds/clients du système doivent être au courant de ces changements afin de pouvoir adresser le bon nœud. L'une des mesures dominantes que nous souhaitons optimiser dans une telle situation est le coût de communication. L'accent est mis sur les solutions qui réduisent le coût de maintenance des informations de distribution de données.

Plusieurs méthodes ont été proposées pour traiter le problème de déséquilibre de charge de données tout en préservant leur ordre. Le déséquilibre de données est un problème important dans plusieurs domaines, telles que les bases de données parallèles Bellatreche et al. (2012); Rishel et al. (2014), Cloud computing Sun et al. (2017) et les systèmes Peer-To-Peer Risson et Moors (2006). La plupart des approches proposées sont basées sur la migration des tuples entre les nœuds Ganesan et al. (2004); Konstantinou et al. (2011); Hsiao et al. (2011); Chawachat et Fakcharoenphol (2015). Ces travaux utilisent généralement deux stratégies universelles : la migration des nœuds et l'échange des enregistrements entre les nœuds voisins. Cependant, la plupart de ces approches nécessitent des statistiques globales de charge pour maintenir l'équilibrage. Le coût d'obtention de ces statistiques exactes de partitionnement dans un système entièrement distribué est au moins O (log p) messages. D'autres approches ont été proposées pour minimiser l'effet de l'asymétrie de données, en particulier avec le partitionnement par intervalle, comme les histogrammes optimaux dans Koudas et al. (2000). Le défi commun entre tous ces travaux est de savoir comment maintenir les statistiques de partitionnement avec un faible coût de communication. Le travail dans Belayadi et Hidouci (2016) est une technique efficace pour réduire le coût de maintenance des statistiques de partitionnement. Notre travail est complémentaire à celui-ci.

Ganesan et al. (2004) ont proposé un algorithme d'équilibrage de distribution de données sur un ensemble de nœuds ordonnés linéairement. Leur algorithme appelé ADJUSTLOAD garantit un faible rapport de déséquilibre entre la charge maximale et minimale. Bien que l'algorithme ADJUSTLOAD soit facile, chaque opération d'équilibrage peut nécessiter des informations de charge globales, ce qui peut être coûteux en termes de coûts de maintien de ces informations. Leur algorithme utilise une structure de données appelée 'skip graph' Aspnes et

Shah (2003) pour maintenir les informations de charge et répondre aux requêtes à intervalles. Chaque opération d'équilibrage nécessite des informations globales avec un coût de O (log p) messages. De plus, une modification des bornes de partitions durant l'équilibrage de charge nécessitera un changement dans les deux Skip graphs utilisés.

Dans ce papier, nous améliorons le travail de l'équipe Ganesan et al. (2004) en réduisant le coût de maintien des statistiques de partitionnement. Nous proposons une technique d'équilibrage de données partitionnées avec des informations approximatives. Notre algorithme utilise les mêmes primitives, NBRADJUST et REORDER que dans Ganesan et al. La primitive NBRADJUST transfère le surplus de données du nœud courant à l'un de ses voisins. La primitive REORDER modifie l'ordre des nœuds pour atteindre les exigences d'équilibrage du stockage. Par conséquent, les bornes des partitions changent, ainsi que les tailles des données de ces partitions. Cependant, notre algorithme n'est pas basé sur des Skip graphs pour maintenir les statistiques de partitions. Le point clé de notre contribution est le vecteur de statistiques des partitions ($\mathcal{VSP}$), où les nœuds et les clients ont des informations approximatives sur les statistiques de distribution de données. Chaque entrée $VSP[i]$ dans ce vecteur est une estimation des bornes de la partition et de la taille des données liées au nœud N_i. Après une opération d'équilibrage, les nœuds participants peuvent changer leurs bornes. Ces nœuds n'ont pas besoin d'informer les autres par ces changements. Les clients utilisent leur $\mathcal{VSP}$ pour diriger les requêtes. Par conséquent, les clients peuvent adresser un mauvais nœud lorsque leurs $\mathcal{VSP}$ sont obsolètes. Néanmoins, chaque fois qu'une interaction se produit entre deux pairs (nœud ou client), ils en profitent pour échanger leurs $\mathcal{VSP}$ afin de se corriger mutuellement. Notre solution surpasse les méthodes existantes en termes de coût de communication. Il n'y a pas de coût supplémentaire pour maintenir la statistique de charge comme dans Ganesan et al.

Avec notre solution, nous ciblons les bases de données parallèles ou les données sont résidentes en mémoire. Ces bases de données doivent traiter les demandes fréquentes des utilisateurs tout en étant continuellement alimentées en temps réel depuis plusieurs sources (par exemple, réseaux de capteurs sans fil où les données sont insérées en continu dans une base de données parallèle). Nous proposons d'améliorer le travail de Ganesan et al. et d'autres travaux qui utilisent des Skip graphs ou des sites coordinateurs pour éviter le coût supplémentaire lié au maintien des statistiques de partitionnement. Nous évitons l'utilisation d'un ou de plusieurs sites centraux pour gérer les statistiques des partitions. L'approche proposée peut être applicable à un large éventail d'applications et est transparente pour les utilisateurs. Un exemple de telles applications comprend les services de surveillance du trafic routier Wang (2010) et les opérations de base dans une base de données parallèle notamment le tri parallèle et la jointure parallèle dans un environnement fortement déséquilibré.

Cet article est organisé comme suit : dans la section 2, nous présentons les notions fondamentales nécessaires à la compréhension de notre proposition. Nous décrivons l'architecture de notre système dans la section 3. Dans la section 4, nous présentons notre approche d'équilibrage de distribution de données. Nous l'évaluons expérimentalement dans la section 5. Les travaux connexes sont décrits dans la Section 6. Enfin, la section 7 conclut l'article.

2 Préliminaires et motivation

2.1 Préliminaires

requête par intervalle : Une opération de base de données bien connue. Elle renvoie tous les tuples entre deux valeurs spécifiées (bornes supérieure et inférieure). Par exemple, renvoyer la liste de tous les employés ayant de 5 à 10 ans d'expérience. Les requêtes à intervalle parallèle Coman et al. (2007); Doulkeridis et al. (2009) ont été étudiées de manière approfondie ces dernières années, car le parallélisme réduit considérablement le temps de réponse. Toutefois, l'exécution de requêtes à intervalle dans un environnement souffrant d'un déséquilibre de données affecte le temps d'exécution de cette requête.

Opération d'équilibrage de données : Elle modifie les bornes des partitions et déplace les tuples entre les nœuds en fonction des nouvelles bornes, ce qui a pour effet de réduire l'asymétrie du système.

Ratio de déséquilibre : Une valeur mesurée indiquant le degré de déséquilibre, comme le rapport entre la plus grande et la plus petite charge. Plus le rapport est élevé, plus le degré de déséquilibre est élevé. Dans le cas d'un simple déséquilibre de données (plusieurs nœuds surchargés et un seul nœud sous-chargé par example), le ratio de déséquilibre est calculé en divisant la moyenne de charge sur la plus petite charge. Dans ce travail, nous visons les systèmes ou il y a un fort déséquilibre de données.

Échange des tuples entre les nœuds voisins : L'équilibrage de la distribution des données se fait en transférant les tuples des nœuds surchargés vers les nœuds les moins chargés. La nécessité de préserver l'ordre des données (pour un traitement efficace des requêtes à intervalle) nécessite que tout échange d'éléments doive être effectué uniquement entre des nœuds logiquement voisins, c'est-à-dire des nœuds gérant des intervalles contigus.

Ré-ordre des nœuds : Il existe des situations où les nœuds sous-chargés ne sont pas contigus aux nœuds surchargés. Dans ce cas, l'un des nœuds distants sous-chargés peut s'éloigner de sa zone logique, rejoindre le nœud surchargé et équilibrer la charge avec lui. Les deux concepts (échange des tuples entre nœuds voisins et réorganisation de nœud) sont présentés dans la figure 1.

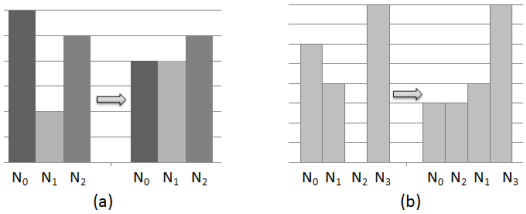

FIG. 1: (a). Échange de tuples entre les nœuds voisins : le nœud N_0 envoie la moitié de ses données vers N_1. (b) Ré-ordre des nœuds : N_2 change sa position et prend le moitié de données du nœud N_0.

2.2 La solution d'équilibrage de charge de Ganesan et al. (2004)

Le travail de Ganesan et al. est censé être représentatif de l'état de l'art. L'équipe de ce travail a suggéré un algorithme judicieux pour réduire l'asymétrie des données. Il garantit un faible rapport de déséquilibre σ entre la plus grande charge et la plus petite. Ce rapport est toujours limité par une petite constante de 4,24. L'algorithme utilise deux opérations :

— NBRADJUT : le nœud N_i transfère son surplus de données à l'un de ses voisins (N_{i+1} ou N_{i-1}, cela dépend du voisin le moins chargé). La migration de données peut modifier les bornes de N_i et le voisin recevant des données.

— REORDER : le nœud le moins chargé (N_r) parmi tous les nœuds du système, transfère tout son contenu à l'un de ses voisins et change sa position logique pour partager des données avec le nœud exécutant l'algorithme d'équilibrage de charge.

Dans les deux opérations, le nœud surchargé nécessite des informations non locales (charges des voisins, position du nœud le plus chargé et le moins chargé). Un nœud donné tente de réduire sa charge chaque fois qu'elle augmente d'un facteur δ. Pour une constante c, Ganesan et al. définissent une séquence de seuils $T_i = c\delta^i$, pour tout $i \geq 1$. Le nœud N_i tente de déclencher la procédure ADJUSTLOAD chaque fois que sa charge $L(N_i)$ est supérieure à son seuil T_i. Quand δ= 2, ils appellent leur algorithme l'algorithme de doublage. La procédure ADJUSTLOAD fonctionne aussi quand $\delta > \phi = (\sqrt{5}+1)/2 = 1,62$, le ratio d'or. Ils appellent leur algorithme qui fonctionne avec ce ratio, l'algorithme Fibbing. Ils prouvent que la procédure ADJUSTLOAD fonctionnant sur ce ratio garantit un rapport de déséquilibre σ de δ^3 = 4,237.

Ganesan et al. utilisent deux Skip graphs. Les Skip graphs sont des listes chaînées Pugh (1990), dans lesquelles chaque nœud a $log(p)$ pointeurs. Le routage entre deux nœuds nécessite O (log p) messages. Le premier Skip graph est utilisé pour avoir les charges des voisins (un message) et pour router les requêtes à intervalle vers le nœud approprié. Le deuxième Skip graph est utilisé pour obtenir les positions du nœud le plus chargé et le moins chargé dans le système (O (log n) messages pour la localité plus les coûts de mise à jour des deux Skip graphs).

3 Architecture du système proposé

Dans cette section, nous définissons une abstraction simple d'une base de données en mémoire vive et nous établissons quelques considérations :

— $N = \{N_1, N_2, ..., N_p\}$, un ensemble de p nœuds connectés par un réseau local comme dans une architecture sans partage de ressources (Shared-Nothing). Nous considérons une relation (ou un ensemble de données) divisée en p partitions sur la base d'un attribut de partitionnement, avec les bornes $R_0 \leq R_1 \leq ... \leq R_p$. Le nœud N_i gère l'intervalle $[R_{i-1}, R_i[$. Nous considérons que les nœuds sont ordonnés par leurs intervalles, cet ordre définit les relations gauche et droite entre eux. Les données résidentes en mémoire sont organisées en lignes.

— $C = \{C_1, C_2, ..., C_m\}$, un ensemble de m clients effectuant des requêtes d'insertion, de suppression ou requêtes à intervalle. Les requêtes de recherche d'un seul élément peuvent être considérées comme un cas particulier de requêtes à intervalle, où les limites supérieure et inférieure sont égales. Les clients peuvent rejoindre ou quitter le système à tout moment.

— Les nœuds et les clients n'ont pas nécessairement les informations réelles sur la distribution des données entre les nœuds (statistiques de partition). Au lieu de cela, ils ont leurs propres vecteurs de partitionnement approximatifs (VSP_n pour les nœuds et VSP_c pour les clients).
— Chaque nœud N_j a son propre vecteur de partitionnement approximatif VSP_{n_j}. Une entrée $VSP_{n_j}[i]$ dans ce vecteur (pour i différent de j), est une estimation des bornes de la partition et de la taille des données liées au nœud N_i. L'entrée $VSP_{n_j}[j]$ contient des informations exactes sur les bornes de la partition et la taille des données du nœud N_j.
— Chaque client C_j a son propre vecteur de partitionnement approximatif VSP_{c_j}. Une entrée $VSP_{c_j}[i]$ dans ce vecteur, est une estimation des limites de la partition et de la taille des données liées au nœud N_i. Les clients utilisent leurs vecteurs de statistiques des partitions approximatifs VSP_c pour trouver les nœuds concernés par les opérations d'insertion, de suppression ou de recherche.
— Chaque nœud N_i a un seuil de charge local. Lorsque la charge dépasse cette limite, le nœud effectue une opération d'équilibrage de charge avec ses voisins. Cette opération met à jour bornes d'intervalle des partitions des nœuds participants.
— Notre algorithme d'équilibrage de charge est appelé sur un nœud auquel l'insertion ou la suppression est en cours ou alors il est appelé pas un nœud qui reçoit des données de ses voisins. Nous évitons l'utilisation d'un site central pour diriger les requêtes. Nous ignorons également les problèmes de contrôle de concurrence et considérons uniquement l'ordonnancement en série des opérations d'insertions et de suppressions, entrelacé par les exécutions de l'algorithme d'équilibrage de charge. Une architecture de notre solution est présentée dans la figure 2.

4 Fonctionnement du système proposé

La principale caractéristique de notre approche est le concept $\mathcal{VSP}$, où chaque nœud ou client a une connaissance approximative des statistiques de partitions. Sur la base de cette connaissance, le nœud effectue un équilibrage de charge chaque fois que sa charge dépasse un seuil local. Ce nœud appelle la procédure NBRADJUST qui transfère le surplus de données et son vecteur vers ses voisins si cela est possible, sinon, il exécute la procédure REORDER. Le voisin, après avoir reçu les données, exécute l'algorithme d'équilibrage de charge et met éventuellement à jour son vecteur. Le processus est répété au niveau de chaque nœud recevant les données jusqu'à ce que tout le système soit dans un état d'équilibre. Ces opérations de maintien de l'équilibrage fonctionnent en arrière plan et en parallèle avec les opérations clientes (recherches, insertion et suppression).

De plus, à chaque fois qu'un message est échangé entre deux nœuds ou entre un nœud et un client, le vecteur de statistiques des partitions $\mathcal{VSP}$ est également inclus (greffé dans le message envoyé), de sorte que le nœud ou le client recevant le message peut comparer avec ses propres statistiques pour conserver la valeur la plus récente pour chaque entrée dans son propre vecteur. De cette manière, les valeurs les plus récentes, concernant les bornes des partitions et leurs tailles, sont ainsi propagées dans le système de manière asynchrone. Nous décrivons dans les sections suivantes le concept $\mathcal{VSP}$.

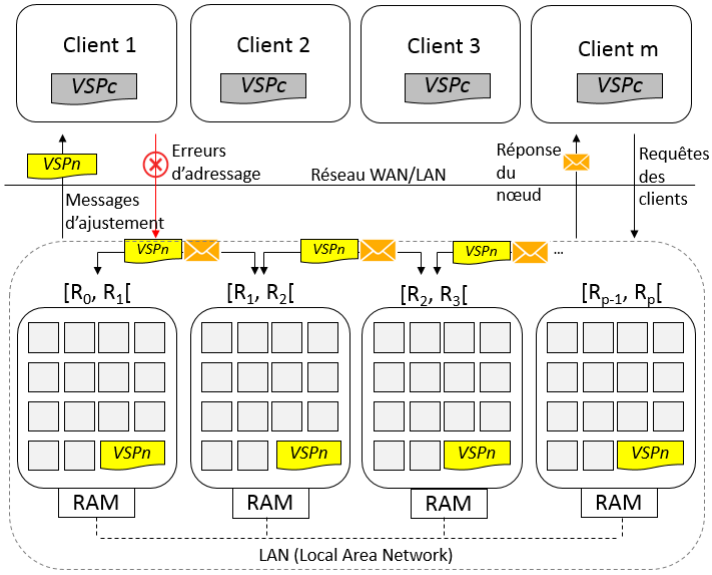

FIG. 2: Architecture du notre système. Dans cette architecture, p nœuds sont connectés entre eux et dédiés pour le stockage de données. Un ensemble de m clients envoient leurs requêtes vers les nœuds. Dans le cas de déséquilibre de données, les tuples sont transférés entre les nœuds.

4.1 Vecteur des statistiques des partitions d'un nœud ($VSPn$)

Considérons que les statistiques de partition d'un nœud N_j sont stockées dans un vecteur $VSP_{n_j}[1,p]$. Chaque $VSP_{n_j}[i]$ stocke une estimation des informations du nœud N_i. Les informations de nœud sont principalement la borne supérieure ($VSP_{n_j}[i].Borne\_sup$), la taille des données locales ($VSP_{n_j}[i].Charge$) et la date de la dernière mise à jour ($VSP_{n_j}[i].Maj$). Le dernier champ ($VSP_{n_j}[i].Maj$) est utilisé pour indiquer l'heure à laquelle l'entrée $VSP_{n_j}[i]$ a été mise à jour pour la dernière fois. Chaque nœud a un index i, $i \in [1,p]$, de sorte que , N_{i+1} est le successeur logique et N_{i-1} est le prédécesseur logique. Le vecteur de nœud est mis à jour dans les cas suivants :

— Requêtes d'insertion ou de suppression. Un client C_i envoie les données à insérer ou à supprimer avec son VSP_{c_i}. Dans ce cas, la taille des données locales et la date du dernière mise à jour sont ajustées. Les entrées qui concernent les autres nœuds sont mises à jour en utilisant le vecteur client VSP_{c_i} reçu.
— Requêtes à intervalle envoyées par les clients. Dans ce cas, seules les anciennes entrées du vecteur de nœud sont éventuellement mises à jour à l'aide du VSP_{c_i} reçu.
— Migration de données du nœud actuel vers l'un de ses voisins ou vice versa. Dans ce cas, la taille des données locales, les bornes et la date du dernière mise à jour sont mises à jour. D'autres entrées du vecteur sont éventuellement mises à jour en utilisant le VSP_n reçu.

4.2 Vecteur des statistiques des partitions d'un client ($VSPc$)

Les données stockées dans les nœuds sont manipulées via les requêtes d'insertion, de suppression et les requêtes à intervalle envoyées par les clients. Considérons que les statistiques de partition d'un client C_j sont stockés dans un vecteur $VSP_{c_j}[1, m]$, où chaque $VSP_{c_j}[i]$ stocke les informations sur le nœud N_i. Les informations d'un nœud sont essentiellement sa borne supérieure ($VSP_{c_j}[i].Borne\_sup$), la taille des données ($VSP_{c_j}[i].Charge$) et la date du dernière mise à jour ($VSP_{c_j}[i].Maj$). L'insertion, la suppression ou la recherche d'un tuple avec une clé k sont effectuées comme suit :

- Le client recherche un nœud N_i de sorte que : $VSP_{c_j}[i-1].Borne\_sup \leq k \leq APV_{c_j}[i].Borne\_sup$. Ensuite, il envoie un message à N_i contenant le tuple à insérer, supprimer ou rechercher. Le vecteur local VSP_{c_j} est également inclus dans le même message.
- Un nœud N_i recevant la requête, vérifie si la clé k est incluse dans son intervalle, si c'est le cas, il exécute la requête spécifiée, met éventuellement à jour ses statistiques de partition ($VSP_{c_j}[i].Charge$ et $VSP_{c_j}[i].Maj$ dans le cas d'une requête d'insertion ou de suppression) et envoie un accusé de réception positif constitué de son VSP_{n_i} au client.
- Si k est en dehors de l'intervalle du nœud, un message d'ajustement du vecteur (VAM) incluant un accusé de réception négatif et le VSP_{n_i} courant est envoyé au client.
- Si un client reçoit un accusé de réception positif du nœud, il met simplement à jour son vecteur s'il n'est pas à jour. Sinon, s'il reçoit un message d'ajustement, il met à jour son vecteur et répète l'opération (vers de nouveaux nœuds, selon les nouvelles valeurs de son VSP_{c_j}) jusqu'à ce qu'il reçoive un accusé de réception positif.

4.3 Algorithme d'équilibrage de distribution de données

Notre algorithme d'équilibrage de charge utilise les deux primitives universelles, REORDER et NBRADJUST comme dans le travail de Ganesan et al. Cependant, nous utilisons le concept $\mathcal{VSP}$ pour conserver les informations de charge globale au lieu d'utiliser les Skip graphs. Notre algorithme que nous appelons AJUSTER_CHARGE, est présenté ci-dessous (Algorithme 1).

Un nœud N_i exécute l'algorithme d'équilibrage de charge chaque fois que sa charge augmente au-delà d'un seuil T_i. L'algorithme utilise le vecteur de statistiques es partitions local pour vérifier si les données peuvent être partagées avec les voisins faiblement chargés (ligne 2). Si la charge de l'un de ces voisins est inférieure à la moitié de la charge de N_i, alors N_i exécute la primitive NBRADJUST pour équilibrer la charge avec ce voisin (lignes 3-9). Sinon, N_i tente d'exécuter REORDER avec le nœud le moins chargé du système N_r. L'algorithme obtient la position du nœud le moins chargé (N_r) à partir du vecteur VSP_n (ligne 11), si la charge de N_r est inférieure à un quart de la charge de N_i, N_r envoie toutes ses données à l'un de ses voisins faiblement chargé (ligne 15) et change de position pour prendre la moitié de la charge de N_i (ligne 16 -17). Si N_i est incapable d'effectuer ni échange d'élément ni réorganisation de nœud, nous concluons que la charge du système est équilibrée. Nous insistons sur le fait qu'il n'y a pas un surcout additionnel pour le mouvement de données.

Algorithme 1 : AJUSTER_CHARGE (N_i, VSP_{n_i})

```
1  Soit N_j le nœud le moins chargé entre N_{i+1} et N_{i-1};
2  if (VSP_{n_i}[i].Charge/2 ≥ VSP_{n_i}[j].Charge) then
3      NB = (VSP_{n_i}[i].Charge − VSP_{n_i}[j].Charge)/2;
4      Envoyer NB tuples à N_j;
5      VSP_{n_i}[i].Charge = VSP_{n_i}[i].Charge − NB;
6      VSP_{n_i}[j].Charge = VSP_{n_i}[j].Charge + NB;
7      mettre à jour VSP_{n_i}[i].Borne_sup;
8      AJUSTER_CHARGE (N_i, VSP_{n_i}[i]);
9      AJUSTER_CHARGE (N_j, VSP_{n_i}[j]);
10 else
11     Trouver r de tel sorte que : ∀k ∈ [1, p], VSP_{n_i}[k].Charge ≥ VSP_{n_i}[r].Charge;
12     //N_r est le nœud le moins chargé;
13     if (VSP_{n_i}[i].Charge/4 ≥ VSP_{n_i}[r].Charge) then
14         Soit N_j le nœud le moins chargé entre N_{r+1} et N_{r-1};
15         Envoyer VSP_{n_i}[r].Charge tuples à N_j;
16         N_r change sa position pour qu'il soit le voisin de N_i;
17         Envoyer VSP_{n_i}[i].Charge/2 à N_r
           VSP_{n_i}[i].Charge = VSP_{n_i}[i].Charge − (VSP_{n_i}[i].Charge/2);
18         VSP_{n_i}[r].Charge = VSP_{n_i}[i].Charge/2);
19         Mettre à jour les bornes supérieures de N_i, N_j, et N_r;
20         AJUSTER_CHARGE (N_i, VSP_{n_i});
21         Renommer les nœuds après le REORDER;
22     else
23         Absence du déséquilibre de distribution de données;
24     end
25 end
```

5 Experimental Evaluation

Dans cette section, nous présentons les résultats de la simulation de notre approche sur un cluster de 8 nœuds et 2 clients. Les algorithmes des nœuds de traitement et des clients ont été exécutés sur des machines équipées d'Intel (R) Core (TM) i7-5500U CPU@2.40GHz et de 8GiB de RAM. Les nœuds et les clients étaient connectés via un réseau Ethernet Gigabit. Nous avons utilisé des machines homogènes, mais nous estimons que les résultats ne changent pas dans le cas des machines hétérogènes. Les algorithmes sont implémentés en langage C en utilisant la bibliothèque de passage de messages (Open MPI). Dans les expérimentations, nous présentons que le scénario d'insertion. Ce cas est plus simple à analyser et fournit des idées générales sur la manière de traiter le cas général. Il est également d'un intérêt pratique car dans de nombreuses applications, comme dans un partage de fichiers, les suppressions se produisent rarement. Afin d'évaluer la nouvelle approche dans un environnement fortement déséquilibré, le système est étudié sous un modèle de simulation que nous appelons $HOTSPOT$. Toutes les opérations d'insertion sont dirigées vers un seul nœud actif. Nous utilisons une séquence

de $5 * 10^4$ opérations d'insertion fréquentes. Les facteurs de performance de notre mécanisme d'équilibrage de charge sont :

- Le rapport de déséquilibre entre le nœud le plus chargé et le moins chargé.
- Le nombre d'erreurs d'adressage des clients.
- Déplacement de données, tous les algorithmes d'équilibrage de charge doivent déplacer des données d'un nœud à un autre.
- Le nombre d'invocation de l'algorithme AJUSTER_CHARGE.

Il est à noter que dans ce travail, le calcul du temps de latence n'est pas très important par ce qu'il dépend fortement de matériel utilisé. C'est pourquoi, nous allons pas visualiser les temps requis pour les opérations des clients. Nos opérations d'équilibrage de données s'opèrent en arrière plan et n'ont aucune influence sur les opérations des clients

5.1 Rapport de déséquilibre

Tout d'abord, nous évaluons le rapport de déséquilibre σ. Nous mesurons le rapport de déséquilibre comme le rapport entre la plus grande et la plus petite charge après chaque opération d'insertion. Nous considérons qu'au début, toutes les charges sont au moins 1. Comme le seuil du système est une séquence géométrique croissante et infinie, comme dans le travail de Ganesan et al., Nous mesurons le rapport de déséquilibre avec trois valeurs du facteur $\delta, (\delta = \phi, \delta = 2, \delta = 4)$. ϕ est le ratio d'or, $\phi = (\sqrt{5} + 1)/2 = 1,62$. La figure 3 montre les rapports de déséquilibre (axe Y) par rapport au nombre d'opérations d'insertion (axe X).

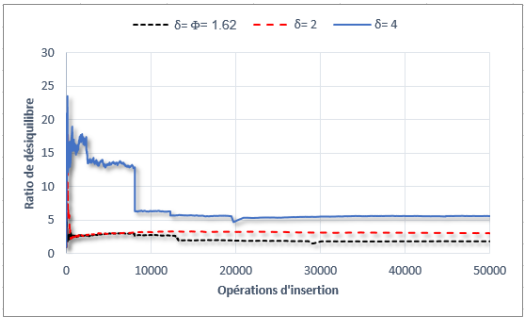

FIG. 3: Le rapport de désiquilibre avec $\delta = \phi = 1.62$, $\delta = 2$, et $\delta = 4$.

Quand $\delta = 1,62$, la courbe montre que le ratio de déséquilibre est toujours borné par une constante 6 et converge vers 1,8 après $2 * 10^4$ opérations contrairement au ratio de déséquilibre de Ganesan et al., qui est borné par 4,24 et converge vers 3,3. Les pics dans la courbe signifient qu'une invocation de l'algorithme AJUSTER_CHARGE a été lancée. La courbe présente plusieurs variations au début car les nœuds sont en phase de croissance où les données sont chargées. Au début, toutes les opérations d'insertion sont envoyées vers le nœud N_0. Les invocations successives de l'algorithme d'équilibrage de charge conduisent à une variation des bornes d'intervalle des partitions. Cependant, les résultats de la troisième expérience $\delta = 4$ sont différents des précédents, les valeurs du ratio sont plus grandes et convergent vers 5.

Des simulations ont été effectuées comparant les performances de notre algorithme d'équilibrage de charge et la procédure ADJUSTLOAD de Ganesan et al. Ganesan et al. (2004). Les

TAB. 1: Comparaison entre l'algorithme de Ganesan et al.ADJUSTLOAD et notre algorithme AJUSTER_CHARGE algorithm.

Procédure	Charge maximale (Tuples)	Charge moyenne (Tuples)	Rapport de déséquilibre	Performance de requêtes
ADJUSTLOAD	1885	781	2.41	21%
AJUSTER_CHARGE	1492	781	1.91	0
ADJUSTLOAD	2102	1048	2.02	27%
AJUSTER_CHARGE	1568	1048	1.49	0

données de la table 1 représentent les résultats comparatifs de l'équilibrage de charge des deux procédures. Le paramètre de comparaison est le rapport de déséquilibre qui est mesuré ici comme le rapport entre la plus grande charge et la charge moyenne du système.

Lorsque la performance du système est mesurée par le temps de réponse de la requête, elle est proportionnelle à la plus grande charge. La performance relative la plus défavorable de l'algorithme AJUSTER_CHARGE par rapport à la procédure ADJUSTLOAD est la différence entre 1 et le rapport entre les deux rapports de déséquilibre, ou 1,0- (1,91 / 2,41) = 0,21. On peut s'attendre à réduire le temps de réponse des requêtes jusqu'à 28% par rapport à un système utilisant le ADJUSTLOAD.

5.2 Messages d'ajustements des clients

Après une opération d'équilibrage, deux nœuds au moins modifient leurs limites de partition en raison de la migration des données, ce qui conduit à modifier les vecteurs de partitionnement de ces deux nœuds. Le client avec un vecteur obsolète peut adresser un mauvais nœud qui a changé ses bornes d'intervalle. Dans notre série d'expériences, nous étions intéressés à déterminer avec quelle efficacité un client obtient une vue réelle sur les nœuds. Nous mesurons le nombre de fois que le client envoie une requête à un mauvais nœud et donc effectue une erreur d'adressage et reçoit un message d'ajustement de vecteur des statistiques (VAM). Les résultats montrés dans la figure 4 présentent une augmentation rapide du nombre d'erreurs d'adressage dans la phase de croissance (de 1 à 1000 opérations d'insertion). Cela revient au fait qu'il y a une invocation fréquente de l'algorithme d'équilibrage, et donc un changement fréquent des bornes d'intervalle des partitions.

5.3 Cout de mouvement de données

Pour équilibrer les charges entre les nœuds, nous nous intéressons également à la minimisation du coût du mouvement autant que possible. Après avoir mesuré le rapport de déséquilibre et l'ajustement du vecteur client, nous mesurons ensuite le coût du mouvement de données. La figure 5(a) trace le nombre cumulé de tuples migrés par notre algorithme (axe Y) par rapport au nombre d'opérations d'insertion (axe X) lors d'une exécution avec $\delta = 1,62$, $\delta = 2$. Nous observons que dans la phase de croissance, le nombre de tuples migrés pour des valeurs δ différentes augmente, maintenir le système bien équilibré entraîne un plus grand nombre

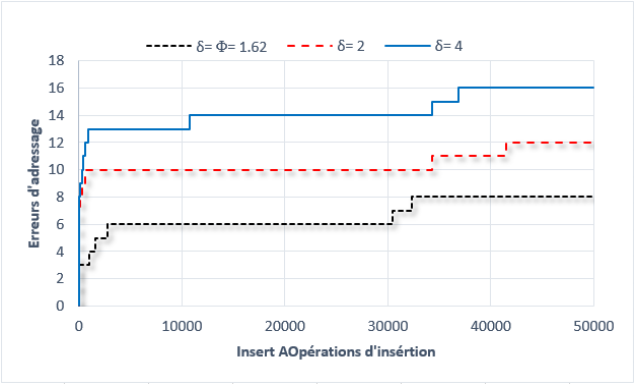

FIG. 4: Le nombre des erreurs d'adressage ($\delta = \phi = 1.62$, $\delta = 2$, et $\delta = 4$)

d'opérations de rééquilibrage. Figure 5(b) trace le coût du mouvement de données lorsque $\delta = 4$.

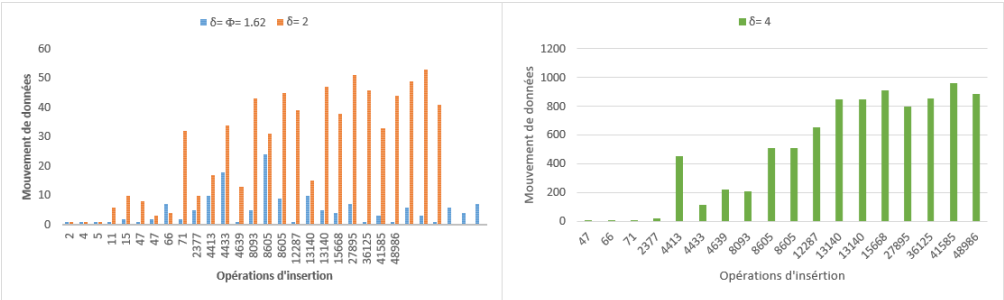

FIG. 5: (a) : Cout de mouvement de données avec $\delta = \phi = 1.62$, $\delta = 2$. (b) : Cout de mouvement de données avec $\delta = 4$

La figure 6 illustre le nombre d'invocations de notre algorithme d'équilibrage de charge (axe Y) par rapport au nombre d'opérations d'insertion (axe X) au cours d'une exécution . L'observation que nous faisons est le nombre d'invocations de l'algorithme augmente pour les trois valeurs de δ pendant la phase de croissance. Le nombre d'invocations d'algorithme a été mesuré avec différentes valeurs de δ ($\delta = 1,62$, $\delta = 2$ et $\delta = 4$). Nous observons que le nombre d'invocations est relativement faible lorsque $\delta = 1,62$. Cela revient au fait que nous supportons certaines situations de déséquilibre.

6 Travaux connexes

Dans cette section, nous décrivons les recherches en cours qui se rapportent à notre méthode d'équilibrage de charge tout en supportant les requêtes par intervalle.

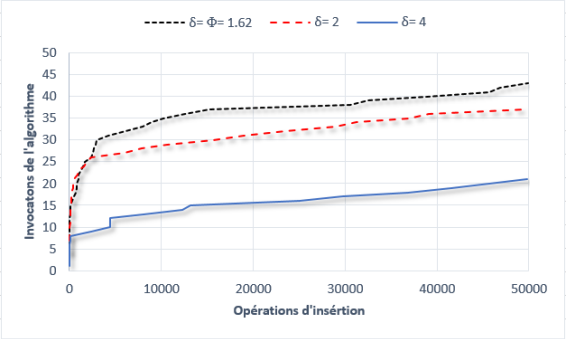

FIG. 6: Le nombre d'invocation de l'algorithme AJUSTER_CHARGE avec $\delta = 1.62$, $\delta = 2$ et $\delta = 4$.

Réseau Peer-To-Peer : dans les réseaux P2P, un certain nombre d'approches récentes d'équilibrage de charge ont été proposées Mirrezaei et Shahparian (2016); Chawachat et Fakcharoenphol (2015); Antoine et al..

SkipNet Harvey et al. (2003) et skip graph Aspnes et Shah (2003) sont des structures de données pour la recherche d'intervalle de données qui ne sont pas basées sur des DHT. Les deux sont adaptés de Skip Lists Pugh (1990). Le coût de calcul de la recherche d'objet dans un skip graphe et dans les réseaux SkipNet est O (log p), où p est le nombre de nœuds dans le réseau. De nombreuses autres structures de données répondent aux requêtes à intervalle et assurent un bon équilibrage de charge dans leurs expériences, par exemple, Mercury Bharambe et al. (2004) et Baton Jagadish et al. (2005). Cependant, Mercury a besoin d'un coût supplémentaire pour estimer la densité des nœuds sur l'anneau d'identification. Baton est basé sur une structure arborescente binaire équilibrée. Il garantit que les requêtes exactes et les requêtes à intervalle peuvent être résolues en O (log p) et que les opérations de mise à jour ont un coût de O (log p). Notre méthode nécessite un coût très faible pour gérer les statistiques de charge afin de répondre aux requêtes à intervalle et assurer l'équilibrage de la charge de données.

Bases de données parallèles/distribuées : une opération d'équilibrage de distribution de données dans une base de données parallèle est effectuée en tant que transaction. Ses conséquences n'affectent pas les autres transactions concurrentes jusqu'à ce que tous les tuples soient déplacés, les bornes des partitions sont mises à jour et la transaction est validée. Le travail dans Rishel et al. (2014) propose une opération de ré-ordonnancement multiple. Cette technique utilise des statistiques de partition qui incluent une estimation du nombre de tuples stockés sur chaque nœud pour chaque relation dans la base de données. Sur la base de cette information, le degré déséquilibre du système est calculé. Le problème que l'on pourrait noter est le coût de la maintenance des statistiques de partition.

7 Conclusion

Le présent article concerne l'équilibrage de charge dans un système MMDB parallèle. Nous avons proposé un algorithme efficace d'équilibrage de la charge de données en ligne qui traite du problème des données asymétriques. Les résultats expérimentaux montrent que notre

approche n'a pas besoin de coût supplémentaire pour maintenir les statistiques de partitionnement, contrairement au coût des solutions efficaces de l'état de l'art. Notre procédure nécessite pour les clients, un sur-coût très faible en moyenne (voire nul) afin de localiser n'importe quelle donnée et ce, même en présence d'un degré d'asymétrie extrêmement élevé. Ce qui rend notre proposition applicable de manière efficace dans des environnements hautement dynamiques où les insertions/suppressions se font en quasi-continu dans la base de données partitionnée en même temps que les requêtes de consultation clientes.

Bien que notre proposition ait été présentée dans le contexte de l'équilibrage de la charge de stockage, pour les requêtes de type « intervalle », elle peut facilement se généraliser pour prendre en considération l'équilibrage d'autres type de requêtes, principalement les tris et les jointures parallèles.

Références

Antoine, M., L. Pellegrino, F. Huet, et F. Baude. A generic API for load balancing in structured P2P systems. In *26th IEEE International Symposium on Computer Architecture and High Performance Computing Workshop, SBAC-PAD Workshop 2014, Paris, France, October 22-24, 2014*.

Aspnes, J. et G. Shah (2003). Skip graphs. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 12-14, 2003, Baltimore, Maryland, USA.*, pp. 384–393.

Belayadi, D. et W. Hidouci (2016). Dynamic range partitioning with asynchronous data balancing. In *2016 Intl IEEE Conferences on Ubiquitous Intelligence & Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress (UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld), Toulouse, France, July 18-21, 2016*.

Bellatreche, L., A. Cuzzocrea, et S. Benkrid (2012). Effectively and efficiently designing and querying parallel relational data warehouses on heterogeneous database clusters : The f&a approach. *J. Database Manag. 23*(4), 17–51.

Bharambe, A. R., M. Agrawal, et S. Seshan (2004). Mercury : supporting scalable multi-attribute range queries. In *Proceedings of the ACM SIGCOMM 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, August 30 - September 3, 2004, Portland, Oregon, USA*, pp. 353–366.

Chawachat, J. et J. Fakcharoenphol (2015). A simpler load-balancing algorithm for range-partitioned data in peer-to-peer systems. *Networks*.

Coman, A., J. Sander, et M. A. Nascimento (2007). Adaptive processing of historical spatial range queries in peer-to-peer sensor networks. *Distributed and Parallel Databases*.

Doulkeridis, C., A. Vlachou, Y. Kotidis, et M. Vazirgiannis (2009). Efficient range query processing in metric spaces over highly distributed data. *Distributed and Parallel Databases*.

Ganesan, P., M. Bawa, et H. Garcia-Molina (2004). Online balancing of range-partitioned data with applications to peer-to-peer systems. In *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, Toronto, Canada, August 31 - September 3 2004*, pp. 444–455.

Harvey, N. J. A., M. B. Jones, S. Saroiu, M. Theimer, et A. Wolman (2003). Skipnet : A scalable overlay network with practical locality properties. In *4th USENIX Symposium on Internet Technologies and Systems, USITS'03, Seattle, Washington, USA, March 26-28, 2003*.

Hsiao, H., H. Liao, S. Chen, et K. Huang (2011). Load balance with imperfect information in structured peer-to-peer systems. *IEEE Trans. Parallel Distrib. Syst.*.

Jagadish, H. V., B. C. Ooi, et Q. H. Vu (2005). BATON : A balanced tree structure for peer-to-peer networks. In *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*, pp. 661–672.

Konstantinou, I., D. Tsoumakos, et N. Koziris (2011). Fast and cost-effective online load-balancing in distributed range-queriable systems. *IEEE Trans. Parallel Distrib. Syst.*.

Koudas, N., S. Muthukrishnan, et D. Srivastava (2000). Optimal histograms for hierarchical range queries. In *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, May 15-17, 2000, Dallas, Texas, USA*, pp. 196–204.

Mirrezaei, S. I. et J. Shahparian (2016). Data load balancing in heterogeneous dynamic networks. *CoRR*.

Pugh, W. (1990). Skip lists : A probabilistic alternative to balanced trees. *Commun. ACM*.

Rishel, W. S., R. B. Rishel, et D. A. Taylor (2014). Load balancing in parallel database systems using multi-reordering. US Patent 8,849,749.

Risson, J. et T. Moors (2006). Survey of research towards robust peer-to-peer networks : Search methods. *Computer Networks*.

Sun, J., O. Afnan, et Y. Lin (2017). Data skew finding and analysis. US Patent App. 15/199,507.

Wang, F. (2010). Parallel control and management for intelligent transportation systems : Concepts, architectures, and applications. *IEEE Trans. Intelligent Transportation Systems*.

Summary

Due to the availability of larger main memory capacities, we are witnessing the presence of parallel memory-based database systems offering increased performance unlike traditional databases. Data partitioning is a precondition for these databases, because it significantly improves the performance of certain queries such as range queries. Partitioning usually causes the Data Skew problem that may contribute to the degradation of query performance. Ganesan et al. (2004) group has proposed an algorithm that offers a low ratio between the maximum and minimum loads among the nodes that offers a low ratio between the maximum and minimum loads of nodes, while exploiting information of the global load. This information is stored in a data structure, called skip graphs, which requires the exchange of (log p) messages between the p nodes of the parallel machine during the balancing process. The goal of our work is to reduce the number of these messages. To do so, we propose a vector of approximate partition statistics (VSP), in which the nodes and clients have a rough view of the data distribution. The maintenance cost of this vector is almost zero. Our approach is validated on a cluster of 8 nodes and 2 clients.