Impact de la réplication sur la latence dans les bases de données distribuées et application à Cassandra

Noël Gillet, Nicolas Hanusse, Frédéric Lalanne

Impact de la réplication sur la latence dans les bases de données distribuées et application à Cassandra

Noël Gillet[1] Nicolas Hanusse[2] et Frédéric Lalanne[1]

[1]*LaBRI, Université de Bordeaux*
[2]*LaBRI, CNRS*

Minimiser la latence est un enjeu dans les bases de données distribuées. La réplication permet, selon certaines conditions, d'obtenir des gains de performance. Nous étudions des bornes théoriques et proposons un algorithme distribué d'équilibrage de charge qui s'avère très compétitif pour la base de données distribuée NoSQL Cassandra.

Mots-clefs : Algorithme distribué, Bases de données distribuées, réplication, débit, latence

1 Introduction

Contexte et motivations Minimiser la latence de requêtes dans des applications distribuées a un impact pour les utilisateurs et un coût financier: dans des moteurs de recherches tels que Google ou Bing, l'ajout artificiel de délai entre 400ms et 2 secondes entraîne une diminution jusqu'à 4% de requêtes sur 1 mois [SB09]. Toutefois, minimiser la latence est un objectif difficile à atteindre. En effet, les requêtes à traiter sont nombreuses et émises en continu par de multiples utilisateurs. De plus, chaque requête entraîne des accès à des données de plus en plus massives, impliquant un coût de traitement important. Enfin, certains nœuds de stockage peuvent se retrouver *surchargés* car ils stockent des données très populaires. Les temps d'accès et donc de latence sont de plus en plus élevés dans ce cas de figure.

Notre contexte de données massives rend les opérations de migrations ou de copies vers d'autres nœuds extrêmement coûteuses. On remarque toutefois que dans la plupart des bases de données distribuées dédiées telles que Cassandra [LM10], les données sont répliquées f fois par défaut. Le premier usage est avant tout de garantir la disponibilité des données en cas de panne, mais il existe également des mécanismes de répartition de charges pour équilibrer les flux de requêtes. L'algorithme `Glouton` par exemple consiste à choisir parmi f nœuds celui qui est le moins chargé afin d'y affecter une requête. Le choix de ce facteur de réplication f a un impact important sur les performances. En effet, si la majorité des requêtes implique des opérations de mises à jour sur les données, augmenter f peut se révéler préjudiciable sur la latence (sans compter le surcoût en mémoire). Nous nous plaçons donc dans le contexte où les requêtes sont uniquement des *requêtes de lecture*. Dans Cassandra, tout nœud peut recevoir une requête d'un client, qu'il possède ou non les données requises. Il joue le rôle de coordinateur pour les requêtes reçues qui sont ensuite transférées à un ou plusieurs nœuds selon le niveau de cohérence souhaité. Dans le mode ONE par exemple, un seul nœud reçoit et traite la requête. Le choix de f dépend de nombreux paramètres: taille des données, que nous appellerons objets par la suite, nombre des objets m, surcoût de gestion, stratégie de réplication et niveau de cohérence. De nombreux benchmarks mettent en relation ces paramètres pour Cassandra, HBase ou MongoDB [DSK+13, KKR14] et montrent qu'augmenter f n'est pas synonyme d'augmentation de débit. Toutefois si on met de côté les aspects de cohérence et de coût de gestion de copies, il existe dans la littérature plusieurs modèles justifiant de l'intérêt de la réplication pour des requêtes de lecture. Cependant, la distribution est souvent considérée comme aléatoire et uniforme, essentiellement dans le cas où $f = 2$ [ABKU99]. Considérons un ensemble fini de R requêtes à répartir. L'algorithme `Glouton`, pour n nœuds et $R = n$ requêtes aléatoires, peut mener à un temps de complétion $\Theta(\log\log n/\log f)$.

Des modèles. On peut considérer deux types de modèles de traitement de requêtes: *hors ligne* et *en ligne*. En hors ligne, les requêtes sont toutes connues avant leur affectation alors qu'en ligne, la décision d'affectation doit être prise à l'arrivée de chaque requête. Dans la littérature, ces modèles correspondent à des traitements en *batch* ou *flux*. Ces deux modèles peuvent être rapprochés à l'aide d'un modèle intermédiaire, le *micro-batch*: à une fréquence donnée, le système reçoit un ensemble de requêtes à allouer en parallèle. Dans notre contexte, les nœuds du système se connaissent et peuvent tous émettre des requêtes. De plus ils possèdent tous une file de requêtes en attente de traitement. Sauf mention contraire, la charge d'un nœud correspond au nombre de requêtes dans sa file. De manière générale, les performances considérées portent sur le *temps de complétion T* (modèle batch) d'un ensemble de requêtes donné, sur la *latence* (temps de traitement moyen d'une requête) ou sur le *débit* (nombres de requêtes traitées par seconde) pour les flux de requêtes selon des distributions aléatoires. Dans notre modèle, nous considérons que les requêtes ont un coût unitaire. Le temps de complétion idéal est donc R/n.

Résumé de nos contributions. Nos contributions sont de deux types:

- Une analyse théorique dans les modèles batchs: (1) Quel que soit le placement, il existe des distributions de requêtes pour lesquels $T = \Omega(\frac{R}{n}(m/n)^{1/f})$. (2) une analyse d'un nouvel algorithme de décision différée appelé `Avr`. Combiné à un placement d'objets aléatoire, il garantit des temps de complétion et de débit qui sont des f-approximations de l'optimal *quelle que soit la distribution de requêtes* en ajoutant un total de n copies supplémentaires. Dans le cas spécifique où le nombre d'objets ou de partitions de données est $m = n^{\alpha}$ et où on choisit $f = (\alpha - 1)\log n$, le temps de complétion optimal est de l'ordre du temps de complétion idéal.
- Une comparaison expérimentale de Cassandra avec deux algorithmes basés sur la décision différée (`Avr` et `Naive`): selon le taux d'injection de requêtes, on divise la latence moyenne mesurée dans Cassandra par 300. Le débit est également beaucoup plus stable que celui de Cassandra.

2 Bornes théoriques sur l'impact de la réplication en centralisé

Nous modélisons le placement d'objets par *l'hypergraphe du stockage G* dont les hyperarêtes d'arité f représentent les objets. Les sommets correspondent aux nœuds du cluster. De même, pour un ensemble de R requêtes, on construit *l'hypergraphe des requêtes H*. S'il n'y a pas de répétition de requêtes, H est un sous-graphe de G. Sur toutes les affectations de requêtes possibles, la valeur minimale de T correspond à la densité maximum sur tous les sous-graphes de H. Notre placement d'objets, réalisé par des fonctions de hachage aléatoires, est donc modélisé par un hypergraphe aléatoire. Un objet est *populaire* si sa fréquence est supérieure à R/n. Nous leur ajoutons un nombre de copies proportionnel à leur popularité.

Dans le modèle batch, nous montrons, s'il n'y pas d'objet populaire, que la densité maximale du graphe de requêtes est de l'ordre de $O(\frac{R}{n}(m/n)^{1/f})$=$O(R/n)$ avec grande probabilité si $f = (\alpha - 1)\log n$. Informellement, pour tout ensemble de requêtes, augmenter f permet de répartir équitablement les requêtes sur n nœuds. *Quel que soit le placement*, il existe une distribution de requêtes pour laquelle cette borne est également une borne inférieure sur T. S'il y a des objets populaires, en les plaçant sur des nœuds différents, chaque nœud reçoit un surcoût de R/n ce qui ne change que peu le temps de complétion. Le calcul d'affectation des requêtes peut se faire de manière optimale avec des algorithmes de flots mais *il nécessite une connaissance totale de toutes les requêtes*.

3 Les algorithmes distribués

Principe de trois algorithmes Dans un environnement réel, les coûts de communication comptent. Nos algorithmes doivent être décrits avec une description asynchrone. Sans information sur l'état du cluster, les nœuds traitent les requêtes de leur file d'attente. Cassandra utilise le principe de *décision immédiate*: la copie ayant la latence la plus faible dans le passé est choisie. Nous utilisons deux principes: la *gestion distribuée de copies* et la *décision différée*, consistant à envoyer les requêtes à toutes les copies. La première copie prête à traiter la requête demande aux autres copies de supprimer la requête de leur file d'attente (Algorithme `Naive`). Dans l'algorithme `Avr`, chaque nœud possède une estimation de la charge moyenne

et si sa charge de travail est inférieure à $(1+\varepsilon)$ fois la moyenne, il s'affecte toutes ses requêtes et ordonne aux autres copies de supprimer les requêtes correspondantes. La charge moyenne correspond au degré moyen du graphe de requêtes.

Performances Dans les modèles batchs, l'algorithme `Avr` revient à "éplucher" le graphe des requêtes. On n'affecte jamais plus que $(1+\varepsilon)$ fois le degré moyen, soit $(1+\varepsilon)f$[†] la densité du graphe courant à chaque nœud. Nous avons proposé une version distribuée asynchrone de l'algorithme de [FCT14], ce qui garantit une affectation qui est une f-approximation de l'affectation optimale dans les modèles batchs. L'écriture asynchrone permet d'adapter l'algorithme décrit dans le modèle micro-batch en traitement de flux distribué.

La difficulté pour la gestion de copies est de proposer une solution distribuée qui calcule la popularité des n objets les plus populaires en utilisant peu de mémoire ($O(n)$ et non pas $O(m)$). Nous donnerons quelques détails par la suite. Les résultats sur les modèles batch se transfèrent au modèle flux selon certaines conditions: il faut que les algorithmes de gestion de copies et d'affectation de requêtes prennent moins de temps que le traitement d'un nombre constant de requêtes.

4 Expériences

Modifications de Cassandra Apache Cassandra[‡] est une base de données distribué orientée colonne et les données sont représentées sous forme de tables à plusieurs colonnes. Une ligne d'une table est identifiée par une clé primaire définie sur des colonnes. Chaque nœud du cluster Cassandra est chargé de stocker une partie des données en fonction des valeurs de hash de la clé de partition, un sous-ensemble de la clé primaire. Dans nos expériences, un objet est une ligne de la table et une requête consiste à trouver et lire un objet. Chaque nœud recevant une requête d'un client joue le rôle de coordinateur pour cette requête. Un ou plusieurs nœuds, appelés *germes* ou *racines*, possèdent une *connaissance globale* sur le système (latence moyenne et état de chaque nœud du système, ...) et sont périodiquement interrogés par tous les nœuds. On précise qu'à tout moment un nœud peut avoir des requêtes allouées ou en attente d'allocation. Nous utilisons une nouvelle notion de charge tirée de [CSW07] qui donne plus de poids aux requêtes déjà allouées. Pour la *gestion de copies*, nous avons modifié Cassandra: chaque nœud peut retrouver les copies de chaque objet et connaît en plus les copies additionnelles des n objets les plus populaires pour le cluster et pour les requêtes qu'il a traité en tant que coordinateur. L'estimation de la popularité est effectuée par la racine qui maintient grâce à l'algorithme `SpaceSaving` [MAE05] un tableau de taille $2n$ correspondant aux objets populaires. A intervalle régulier, le germe vérifie la popularité des objets et commande un ajout ou une suppression de copies d'objets en cas de changement significatif. Nous avons implémenté la gestion distribuée de files de requêtes nécessaire pour `Avr` et `Naive`. Pour `Avr`, le germe stocke également la charge totale des nœuds du cluster.

Comparaison avec Cassandra Nos expériences ont pour objectif de montrer, en pratique, l'intérêt de la gestion des copies des objets populaires ainsi que les meilleurs performances des algorithmes à décision différée (Avg et Naif) par rapport au principe de décision immédiate (Cassandra). Nous utilisons $f=2$, $n=32$ et $m=256$ et des objets de 10 Mo. Le temps de traitement unitaire d'une requête correspondant à une lecture prend 30 ms. La latence est nécessairement supérieure car il faut ajouter les communications entre coordinateurs et nœuds stockant l'objet ainsi que les surcoûts induis par Cassandra et les modifications apportées. Le scenario DYNAMIQUE montre les très bonnes performances de `Avr` et `Naive` vis à vis de Cassandra: 400 000 requêtes sont insérées en alternant une distribution uniforme et de Zipf de paramètre 2 toutes les 80 000 requêtes (toutes les 320

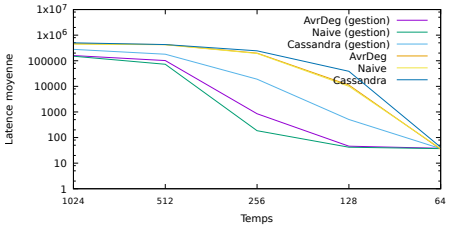

Fig. 1: Évolution de la latence (ms)

[†] la somme des degrés est fm pour un hypergraphe d'arité f.

[‡] `http://cassandra.apache.org/`

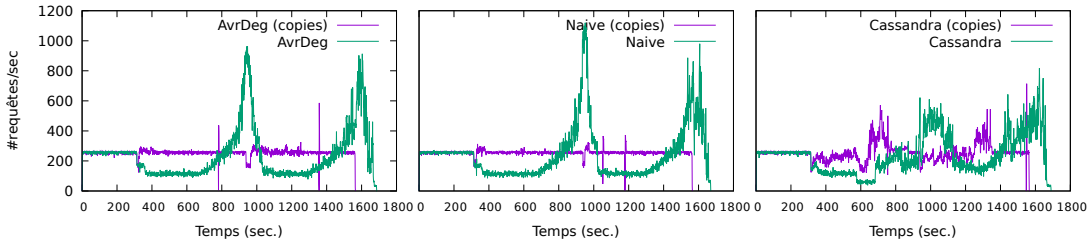

Fig. 2: Débit pour taux d'injection de 256 req. par seconde pour cassandra, average et naif

secondes environ). Dans l'idéal le système pourrait faire $32 * 1/0.03 = 1067$ requêtes par seconde si l'équilibrage était parfait. En faisant varier le taux d'insertion de requêtes (Fig. 1), nous observons que le cluster sature dès 128 requêtes par seconde pour Cassandra alors que `Naive` et `Avr` se comportent très bien à 256 requêtes/seconde: on constate un débit extrêmement stable (Fig. 2) et la latence moyenne est nettement améliorée (Fig. 3). Ces performances s'expliquent en partie par le processus de décision immédiate de Cassandra qui tient compte d'une estimation de charge de chaque nœud dans le passé. Cette information est extrêmement instable lorsque le système sature. D'autres expériences, non détaillées ici, portent (1) sur le temps de complétion lorsque le système est faiblement ou assez saturé et (2) *sur l'impact du choix de* f. En faisant varier $f = 2, 4, 8$ et 16, sur des distributions adversariales, le temps de complétion est effectivement divisé avec un facteur de l'ordre de f. Par contre, sur des jeux de données réels (avec quelques objets populaires), le temps de complétion augmente à partir de $f = 8$. L'équilibrage est déjà bon pour $f = 4$ et au delà, on paye un surcoût important dû au fort taux de réplication: certaines requêtes sont traitées simultanément par plusieurs nœuds avant que les files d'attente n'aient pu faire des opérations de suppression.

Le bilan est que la gestion des copies apporte un véritable gain en pratique: les changements de distribution sont vite repérés et les copies supplémentaires sont créées en quelques secondes. Sachant qu'augmenter le facteur de réplication pour tous les objets peut être préjudiciable, nous proposons d'avoir $fm+n$ copies d'objets avec f petit. Le principe de la décision différée est également plus efficace que la décision immédiate dans toutes nos expériences. Seul `Avr` apporte une garantie théorique sur le débit mais `Naive` ne nécessite pas de connaissance sur la charge moyenne du cluster.

Algorithmes	**latence moyenne**	
Gestion	oui	non
`Cassandra`	19	244
`Avr`	0.8	198
`Naive`	0.2	192

Fig. 3: Latence (sec.)

References

[ABKU99] Yossi Azar, Andrei Z. Broder, Anna R. Karlin, and Eli Upfal. Balanced allocations. *SIAM J. Comput.*, 29(1):180–200, 1999.

[CSW07] Julie Anne Cain, Peter Sanders, and Nicholas C. Wormald. The random graph threshold for *k*-orientiability and a fast algorithm for optimal multiple-choice allocation. In *SODA*, pages 469–476. SIAM, 2007.

[DSK+13] Elif Dede, Bedri Sendir, Pinar Kuzlu, Jessica Hartog, and Madhusudhan Govindaraju. An evaluation of cassandra for hadoop. In *2013 IEEE Sixth International Conference on Cloud Computing, Santa Clara, CA, USA, June 28 - July 3, 2013*, pages 494–501. IEEE Computer Society, 2013.

[FCT14] Martin Farach-Colton and Meng-Tsung Tsai. Computing the degeneracy of large graphs. In Alberto Pardo and Alfredo Viola, editors, *LATIN 2014: Theoretical Informatics - 11th Latin American Symposium, Montevideo, Uruguay, March 31 - April 4, 2014. Proceedings*, volume 8392 of *Lecture Notes in Computer Science*, pages 250–260. Springer, 2014.

[KKR14] Jörn Kuhlenkamp, Markus Klems, and Oliver Röss. Benchmarking scalability and elasticity of distributed database systems. *PVLDB*, 7(12):1219–1230, 2014.

[LM10] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *Operating Systems Review*, 44(2):35–40, 2010.

[MAE05] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *ICDT*, volume 3363 of *Lecture Notes in Computer Science*, pages 398–412. Springer, 2005.

[SB09] Eric Schurman and Jake Brutlag. Performance related changes and their user impact. In *velocity web performance and operations conference*, 2009.