# Exploitation de l'Interaction des Requêtes OLAP pour la Gestion de Cache et l'Ordonnancement de Traitements

Amira Kerkad*, Ladjel Bellatreche*, Dominique Geniet*

*LIAS/ENSMA- Université de Poitiers
Futuroscope, 86960 - France
(amira.kerkad, bellatreche, ageniet)@ensma.fr

**Résumé.** Le cache est l'une des composantes principales d'un système de gestion de bases de données (SGBD). Les SGBD manipulant des bases de données volumineuses comme les entrepôts de données stockent souvent les données sur le disque. En conséquence, l'interrogation nécessite un transfert des données du disque vers la mémoire centrale via le tampon. Un nombre important de travaux sur la gestion de tampon ont été proposés. Malheureusement, ils supposent que les requêtes soient ordonnées. Dans le contexte des entrepôts de données, les requêtes interagissent du fait qu'elles utilisent la table des faits. Cette interaction pourrait impacter la gestion de cache et offrir un bon ordonnancement de requêtes. Dans cet article, nous proposons d'étudier conjointement le problème de gestion de tampon (BMP) et le problème d'ordonnancement de requêtes (QSP). Trois algorithmes sont proposés pour résoudre le problème conjoint. Finalement, un simulateur et une validation sous Oracle11G sont proposés.

## 1 Introduction

La majorité des SGBD exploite la mémoire cache pour réduire le coût des requêtes. Le gestionnaire du cache est l'une des principales composantes d'un SGBD (Effelsberg et Härder (1984)), surtout avec un grand volume de données comme dans les Entrepôts de Données (ED) [1]. La technologie ED est le noyau des applications de BI (Business Intelligence), qui offre une capacité de stockage des données de sources hétérogènes, et des solutions d'accès efficaces par des requêtes OLAP (On-Line Analytical Processing). Actuellement, les applications de BI gèrent d'immenses volumes de données, ainsi les requêtes deviennent très gourmandes en temps de réponse. Les applications des ED Relationnels sont souvent modélisées par des schémas en étoile [2]. La Figure 1 montre un exemple de schéma en étoile pour le Star Schema Benchmark (O'Neil et al. (2009)) contenant une table de faits et des tables de dimension. Les requêtes qui interrogent ce genre de schéma sont dites *requêtes de jointure en étoile*. Dans les EDR, les jointures portent sur la table des faits, ainsi la présence de résultats communs entre les requêtes est fréquente. Ce phénomène est connu dans d'optimisation multi-requêtes (Sellis (1988)) et il a été largement considéré dans la sélection de structure d'optimisation comme

1. Par exemple, l'ED eBay contient 2 petaoctets de données.
2. Le schéma en flocon de neige est moins utilisé que le schéma en étoile

les vues matérialisées (Yang et al. (1997)). Le rôle principal du gestionnaire du cache dans un tel contexte, est de réduire les accès au disque. Les stratégies de remplacement comme LRU, FIFO, ...permettent de gérer le cache pour satisfaire d'autres requêtes. La majeure partie des travaux réalisés sur la gestion du cache en BD en général, et les EDs en particulier, considère que les requêtes sont ordonnées au préalable. Ainsi, pour exécuter la première requête, le SGBD (1) identifie des pages pertinentes sur le disque, (2) charge ces pages en cache depuis la mémoire centrale et (3) exécute la requête. La requête suivante doit tirer profit du contenu du cache si elle a des résultats intermédiaires communs avec la première. Sinon, le SGBD refait le même processus qu'avant et ainsi de suite. L'ordre des requêtes doit influencer positivement le coût d'exécution de la charge de requêtes. QSP est une instance du problème d'ordonnancement. Ce dernier est utilisé dans des environnements divers comme les flux de données accédant des ressources partagées (Tanenbaum (2007)). Les systèmes d'exploitation utilisent un ordonnanceur de processus (CPU) et de ressources (mémoire, réseau, etc.) pour gérer l'accès aux ressources au prochain quantum. Deux types d'ordonnancement sont distingués : (1) Hors-ligne avec la connaissance préalable des processus/flux qui surviennent durant la vie du système (Schild et Würtz (1998)), et (2) Enligne définissant l'ordre pendant l'exécution (Choquet-Geniet et Grolleau (2004)). En BD, l'ordonnancement a été utilisé pour optimiser des requêtes concurrentes (Sadeg et Amanton (2005)) en trouvant un ordre de requêtes réduisant leur coût d'exécution. Ce problème est prouvé NP-complet dans le context où les requêtes partagent des résultats intermédiaires en commun (Roy et al.; Thomas et al. (2006)). Malheureusement, les deux problèmes (BMP et QSP) sont souvent traités séparément. La mise en cache dépend fortement de l'ordre des requêtes. Cette interaction nous a motivés pour considérer le problème combiné incluant le BMP et le QSP et de proposer des algorithmes de résolution. Dans cet article, nous présentons un travail préliminaire traitant ce problème conjoint. Ce papier est structuré comme suit : la section 2 présente l'état de l'art, la section 3 donne les différents concepts liés à notre problème. La section 4 décrit nos trois algorithmes proposés : affinité, hill climbing et algorithme génétique. La section 5 contient les expériences validant nos propositions sur Oracle11g. La section 6 conclut le papier en résumant les principaux résultats et en présentant quelques pistes ouvertes.

## 2 Etat de l'art

Le problème de gestion du cache (Buffer Management Problem) a été étudié dans différents types de bases de données : (1) les BDR (Cornell et Yu (1989); Chou et DeWitt (1985); Effelsberg et Härder (1984)), (2) le Web Sémantique (Yang et Wu (2011)), (3) les EDs (Thomas et al. (2006)) et (4) les BD Flash (Ou et al. (2010)). Dans les BDR, les dépendances entre l'optimisation de requêtes et la gestion du cache (BM) ont été établies. A cause de la complexité de BMP, plusieurs travaux de recherches ont été concentrés sur deux principaux sous-problèmes : (1) la sélection des données candidates au cache, et (2) la stratégie de remplacement. Pour le premier, certaines études proposent l'utilisation d'algorithmes génétiques, du Data-mining, etc. pour sélectionner les données candidates au cache. Pour le second, des stratégies de remplacement (LRU, FIFO, LIFO, MRU etc.) ont été proposées.

La structure des EDR amène à réutiliser des résultats intermédiaires communs de requêtes (Yang et al. (1997)), et ainsi à l'élagage de l'espace de recherche car ses résultats deviennent candidats au cache. Roy et al. proposent des solutions de sélections de résultats intermédiaires

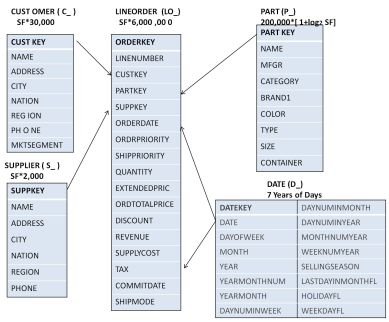


FIG. 1 – *Star Schema Benchmark*

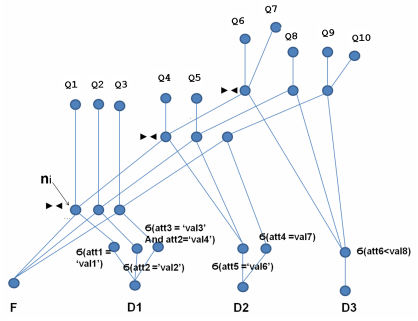


FIG. 2 – *MVPP avec 10 requêtes OLAP*

communs pour être cachés dans un espace cache limité. Récemment, la communauté du Web Sémantique s'est intéressée au BMP. Yang et Wu (2011) présentent une solution pour cacher les résultats intermédiaires des requêtes Sparql, où LRU est employée. L'ordonnancement de requêtes a suscité l'intérêt des communautés de l'informatique parallèle et distribuée : qualité des services (Wu et al. (2009)), ordonnancement temps-réel (Chipara et al. (2007)), etc. En BD, il a suscité moins d'intérêt comparé au BMP. Phan et Li (2008a) proposent une solution de sélection de vues matérialisées en considérant le QSP et utilisant LRU. Par conséquent, QSP est une partie entière du processus de matérialisation. Pour cela, deux algorithmes génétiques ont été utilisés : un pour extraire les meilleurs candidats par l'Advisor DB2 d'IBM, et l'autre pour ordonnancer les requêtes. Certains travaux (Gupta et al. (2001); Thomas et al. (2006)) ont considéré le problème de cache et d'ordonnancement dans les EDR avec des hypothèses simplificatrices. Des heuristiques sont proposées sans validation réelle.

# 3 Concepts et formalisation

Dans cette section, nous formalisons le problème, et définissons notre modèle de coût. L'identification des candidats au cache représente le cœur de notre proposition. On utilise la structure MVPP (Multiple View Processing Plan) (Sellis (1988)) engendré par la fusion des plans de requêtes, pour resortir les sous-expressions communes (Yang et al. (1997)) (Figure 2).

**Définition 1** *pour chaque nœud $no_i$ dans le plan d'une requête, on définit sa fréquence $f(no_i)$ comme le nombre de requêtes y accèdant* [3].

**Définition 2** *une classe de requêtes est un ensemble de requêtes partageant au moins un nœud dans le cache.*

**Formalisation du BMQSP :** Pour faciliter la compréhension et la formalisation du problème combiné, on cosidère la connaissance préalable de la charge (hors-ligne), un environnement centralisé et un cache initialement vide. On présente une formalisation de BMP et QSP séparément, pour identifier leur interaction. BMP est formalisé comme suit : (1) Entrées : un

3. Par exemple, la fréquence du nœud $n_i$ dans la Figure 2 est égale à 4

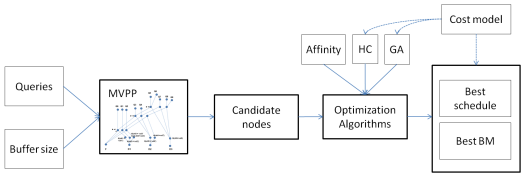


FIG. 3 – *Méthodologie d'optimisation*

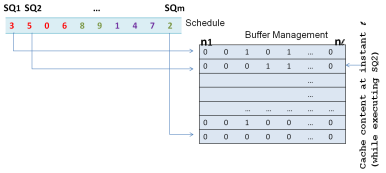


FIG. 4 – *Codage de solutions*

*EDR* et une charge de requêtes $Q$, (2) Contrainte : taille du cache $\mathcal{B}$, et (3) Sortie : stratégie d'allocation/libération du cache (Buffer Management BM). QSP est formalisé comme suit : (1) Entrées : un *EDR*, une charge $Q$, et une stratégie BM, et (2) Sortie : requêtes ordonnancées $SQ$. Notre problème combiné consiste à ordonnancer les requêtes et à gérer le buffer simultanément pour minimiser le coût de traitement total de $Q$. Une formalisation du problème joint de BMP et QSP (appelé BMQSP) s'exprime de la façon suivante : (1) Entrées : un *EDR*, et une charge $Q = \{Q_1, Q_2, \ldots, Q_m\}$, (2) Contrainte : taille du cache $\mathcal{B}$ et (3) Sortie : les requêtes ordonnancées $SQ = \{SQ_1, SQ_2, \ldots, SQ_m\}$, et la stratégie BM correspondante. La résolution du BMQSP nécessite une exploration d'un grand espace de recherche. Supposons que dans une charge de $m$ requêtes, il y ait $l$ nœuds. On aura donc $m!$ ordres possibles de requêtes. De plus, pour chaque ordre, il y a un sous-ensemble de nœuds qui seront mis en cache de façon dynamique [4]. Les différents cas possibles du cache sont alors : $\sum_{i=1}^{l} i!$ cas. En tout, pour trouver à la fois un ordre optimal et la mise en cache correspondante, l'espace de recherche est de $m! \sum_{i=1}^{l} i!$ cas différents. Pour cela, on doit définir une nouvelle approche.

# 4 Heuristiques proposées

Avant de présenter les heuristiques, on décrit l'architecture de notre proposition (Figure 3). A partir d'une charge de requêtes et d'un espace buffer, le MVPP est généré pour avoir l'ensemble de nœuds candidats à la bufferisation $\mathcal{N} = \{no_1, no_2 \ldots no_l\}$. Trois heuristiques sont proposées : Affinité, Hill Climbing et une Génétique : L'algorithme d'affinité exploite l'interaction des requêtes pour décider de l'ordre. Le HC et la Génétique explorent l'espace de solutions possibles en se basant sur certains opérateurs et un modèle de coût. La principale particularité dans nos heuristiques est l'utilisation du même ensemble de candidats $\mathcal{N}$ et du même codage. Le choix du HC et de l'AG est motivé par l'adoption de la communauté des ED de ces heuristiques pour la sélection de structures d'optimisation avancées comme les vues matérialisées (Bennett et al. (1991); Phan et Li (2008b)), le partitionnement (Bellatreche et al. (2006)) et les index (Chaudhuri (2004)).

**Codage des schémas :** Notre codage est représenté par deux parties connectées : (1) un vecteur représentant l'ensemble de $m$ requêtes, (2) chaque cellule $cel_i$ du vecteur représentant une requête $Q_i$ est connectée à un vecteur, noté $bitmap^{Q_i}$ de longueur $l$, représentant la présence ou non de nœuds en cache après l'exécution de $Q_i$. Une solution finale pour le scénario du cache (BM) est représentée par juxtaposition de ces vecteurs (Figure 4).

4. On suppose qu'un nœud est mis en cache au plus une fois durant l'exécution de la charge.

**Modèle de coût :** Pour évaluer nos solutions, nous définissons un modèle de coût pour estimer le nombre d'E/S requis pour exécuter la charge [5]. Le coût global est la somme des coûts des requêtes : $Cost\_total = \sum_{i=1}^{n} cost(Q_i)$, nous le décrivons sans et avec le cache :

**1) Sans cache :** la plupart des études développant des modèles de coûts pour sélectionner des structures d'optimisation ignore le cache. Ainsi, le coût d'une requête donnée $Q_i$ est estimé comme la somme des coûts des opérations $op_i$ dans son plan d'exécution [6]. Trois types d'opérations sont distingués : (1) première jointure (FJ), (2) résultat intermédiaire (IR) et (3) opération finale (AG) (agrégation et group by).

**2) Avec cache :** pour prendre le cache en considération dans notre modèle de coût, une extension du modèle précédent est nécessaire. Le contenu du cache est vérifié pour toute opération $op_i \in \{FJ, IR, AG\}$ dans le plan d'une requête $Q_i$. Soit $\theta(op_i)$ l'estimation d'accès au disque pour satisfaire la tâche $op_i$. On définit une fonction $b(op_i)$ qui vérifie si le résultat de $op_i$ est déjà en cache (=1) ou non (=0). Ainsi, de coût d'exécution de $Q_i$ impliquant $n_i$ opérations est : $Cost(Q_i) = \sum_{i=1}^{n_i} \left[\theta(op_i) \times \prod_{j=i}^{m}(b(op_j))\right]$

**Algorithme d'Affinité :** Pour présenter cet algorithme, deux définitions sont requises :

**Définition 3** *L'affinité entre $Q_i$ et $Q_j (i \neq j)$, notée $Aff(Q_i, Q_j)$ est le nombre de nœuds communs dans leur plan. Elle est calculée à partir du codage proposé comme le produit scalaire des vecteurs : $bitmap^{Q_i} \times bitmap^{Q_j}$ corresspondants* [7].

**Définition 4** *L'affinité réflexive de la requête $Q_i$ notée $Aff(Q_i, Q_i)$ est calculée comme : $Raff(Q_i) = \sum_{k=1,k\neq i}^{m} Aff(Q_i, Q_k)$*

Cette affinité réflexive donne la contribution individuelle d'une requête. Pour cela, nous considérons la première requête de la charge ayant un maximum d'affinité reflexive, nous l'exécutons, et nous allouons ses nœuds candidats en cache (si la taille buffer B le permet). L'ordre des requêtes est déterminé par le contenu du cache. Ainsi, pour chaque requête : (1) exécuter la requête en cours $SQ_i$, (2) lire et/ou mettre à jour le cache par les données nécessaires pour $SQ_i$, si des données deviennent inutiles, notre gestionnaire de buffer (BM) les enlèves du cache, (3) l'ordonnanceur dynamique réordonne les requêtes restantes en avançant celle qui utilise le maximum de données en cache. Enfin, (4) la matrice de buffer management BM est mise à jour par le scénario du cache de $SQ_i$. L'algorithme réitère jusqu'à la dernière requête.

**Hill Climbing :** Pour l'améliorer l'algorithme d'affinité, un hill-climber est développée pour trouver une solution proche de l'optimum. Ses principales étapes consistent à (Russell et Norvig (2003)) : (1) Considérer une solution initiale, obtenue par l'algorithme d'affinité et (2) Améliorer cette solution par des transformations jusqu'à atteindre la condition d'arrêt. On utilise une fonction de transformation aléatoire $Transform(sol_j) \rightarrow sol_k$ qui prend une position aléatoire du vecteur de requêtes (*schedule*) et change la valeur à cette position. Comme le vecteur représente les requêtes ordonnées, un changement de valeur dans une position $p$ implique le changement de toutes les positions qui suivent dans ce vecteur et les vecteurs correspondant

5. Les requêtes considérées sont de type : sélection-projection-jointure.
6. Le modèle de coût est détaillé dans : Kerkad et al. (2011)
7. Par exemple, l'affinité entre $Q_0$ et $Q_3$ est égale à 4.

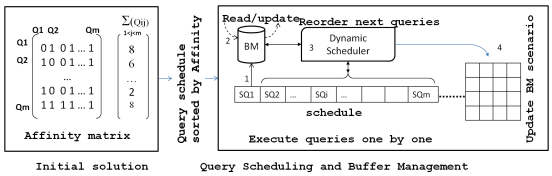


FIG. 5 – *Description d'algorithme d'Affinité*

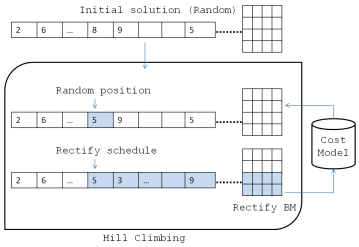


FIG. 6 – *Description du HC*

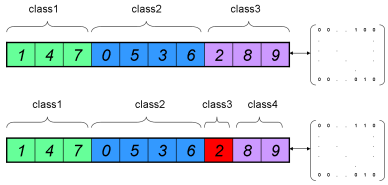


FIG. 7 – *Classes de requêtes*

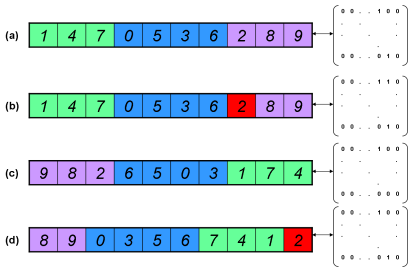


FIG. 8 – *Selection de chromosomes*

à l'état du cache dans la matrice BM. Ces deux tâches sont assurées par deux fonctions de rectification respectivement : $Rectify_{QS}$ et $Rectify_{BM}$ (Figure 6).

**Algorithme génétique :** Les algorithmes génétiques (AG) (Holland (1975)) deviennent largement utilisés dans les problèmes d'optimisation en BD. Contrairement au HC, Ils peuvent sortir des optima locaux grâce à la mutation. Un AG est défini par cinq éléments : (1) le codage des solutions, (2) la génération de population initiale, (3) la fonction fitness, (4) les opérateurs génétiques (sélection, croisement et mutation), et (5) paramétrage de l'algorithme (taille de population, taux de croisement, taux de mutation, et la condition d'arrêt). Initialement, les solutions sont aléatoirement engendrées pour avoir une population homogène. Le même codage est utilisé que le HC. La fonction fitness utilisée est basée sur notre modèle de coût. On a trois règles à considérer : (1) Si le nombre de nœuds cachés d'une requête est différent d'une solution à une autre, alors les classes de requêtes obtenues seront différentes (Figure 7). (2) Le changement de la position d'une requête d'une classe à une autre n'est pas bénéfique, car le cache contiendra des pages non pertinentes pour cette requête. (3) Changer n'importe quelle position de requête entraine un changement dans tout le scénario du cache à partir de cette requête là. Ainsi que dans toutes les requêtes à venir. Nos opérateurs génétiques sont adaptés selon ces règles : Les parents sont sélectionnés parmi la meilleur moitié de la population. Chaque parent A est croisé avec un parent adéquat B ayant la même classe de requêtes, contenant une requête aléatoire. Le croisement s'effectue en échangeant toute la classe entre A et B. Si aucune paire de parents n'est trouvée, les fils (*offsprings*) seront identiques aux parents. Pour éviter la multi-occurence de requêtes, la mutation est appliquée par échange de deux génomes du *Schedule*, avec une mise à jour de la partie *BM* correspondante.

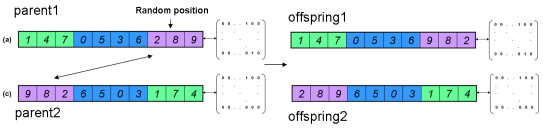


FIG. 9 – *Croisement adapté pour BMQSP*


FIG. 10 – *Mutation adaptée*

**Algorithme de remplacement du cache (*buffer replacement algorithm*) :** Dans cette section, on présente l'algorithme qu'on a utilisé dans notre approche pour gérer le contenu du buffer en considérant QSP. Quand une requête est choisie pour être exécutée, ses nœuds candidats [8] de cette requête sont cachés, si la taille du cache disponible le permet. La prochaine requête est celle qui utilise le maximum de données en cache. Si une requête utilise un nœud, la fréquence d'accès à ce nœud est décrémentée pour indiquer le nombre de requêtes à venir qui auront besoin de ce nœud. A chaque exécution, tous les nœuds ayant une fréquence nulle sont éliminés du cache. La même opération est appliquée jusqu'à la dernière requête.

# 5 Etude expérimentale

On conduit nos expériences théoriquement, suivies par une validation sous Oracle11g, sur un entrepôt SSB (Star Schema Benchmark ; 6000000 de faits). L'expérimentation théorique est faite par un simulateur multiplateforme développé en utilisant notre modèle de coût. La validation de nos résultats est faite sur une machine Core2 Duo de $4G$ de RAM et un SGBD Oracle11g. La charge considérée [9] contient 30 requêtes couvrant différents types de jointure.

**Outil de simulation :** Notre simulateur, développé en Java, contient 3 modules essentiels : (1) connexion et extraction de métadonnées d'un EDR, (2) visualisation et modification des requêtes et (3) optimisation et paramètrage des algorithmes (AG, HC et Affinité) et présente le résultat de façon compréhensible par l'administrateur. Si ce dernier n'est pas satisfait, l'optimiseur donne la main pour paramètrer et relancer les algorithmes. Sinon, l'administrateur peut déployer ses résultats sur son SGBD en utilisant un script approprié.

**Résultats obtenus :** Nous comparons les résultats de nos algorithmes proposés dans la Figure 13 : le HC et l'AG donnent une meilleure optimisation que l'algorithme d'affinité grâce au modèle de coût et à l'exploration de l'espace de solutions plus large. Le temps d'exécution requis pour chaque algorithme varie. La Figure 12 montre que le HC est beaucoup moins lent que l'AG. L'algorithme d'affinité est le plus rapide des trois. Pour comparer l'efficacité de nos algorithmes par rapport à LRU (utilisé dans la plupart des travaux) dans le contexte d'ED, une série d'expériences est conduite avec et sans ordonnancement. Les résultats obtenus (Figure 11) montrent que notre gestion du cache dynamique est plus adaptée que LRU dans les EDs. On conclut aussi que l'ordonnancement des requêtes permet d'améliorer le rendement de la gestion du cache.

8. Tous les nœuds ou un sous ensemble comme dans l'AG
9. Disponibles dans Kerkad et al. (2011)

**Validation des résultats sous oracle11g :** La validation des résultats théoriques est effectuée sous Oracle 11g avec le même jeu de données (SSB et 30 requêtes). Cette étape nécessite la préparation du cache et la réécriture des requêtes [10]. La Figure 14 montre les résultats finaux qui sont très similaires aux résultats théoriques. Ceci montre la qualité de notre modèle de coût.

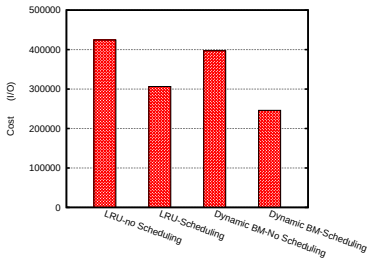


FIG. 11 – *Comparaison de la gestion dynamique de buffer avec LRU*

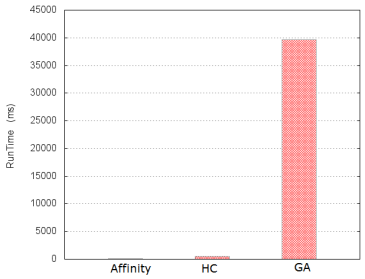


FIG. 12 – *Comparaison des temps d'exécution des algorithmes*

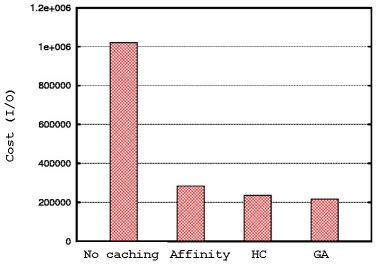


FIG. 13 – *Comparaison des performances des trois algorithmes*

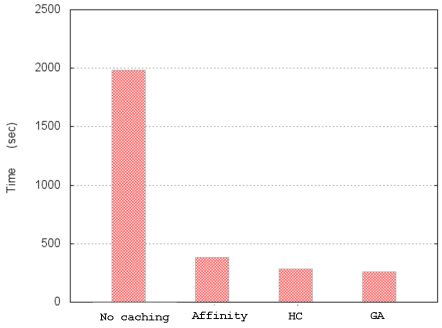


FIG. 14 – *Validation des résultats de simulation des algorithmes*

# 6 Conclusion

La gestion du cache et l'ordonnancement de requêtes sont deux problèmes importants dans le contexte de BDs volumineuses. Ils sont généralement traités de manière séparée, malgré leur corrélation. Les principales études conduites sur la gestion du cache suppose que les requêtes soient préalablement ordonnées. D'autre part, l'ordonnanceur nécessite la connaissance du contenu du cache pour établir l'ordre efficace. Cette interaction nous a conduit, dans ce papier, à considérer le problème conjoint de la gestion du cache et l'ordonnancement des requêtes dans les EDs. Ce problème a un impact sérieux sur le coût de traitement de requêtes et les structures d'optimisation de données. Pour répondre à ce problème, on propose d'abord une

10. Le tuning du SGBD pour préparer le cache, ainsi que le script de réécriture sont détaillés dans Kerkad et al. (2011)

formalisation de notre problème combiné, et une étude de complexité. Le haut niveau de complexité du problème a motivé le développement d'heuristiques, où on a proposé : un algorithme d'affinité, un Hill climbing et une génétique. Des expérimentations théoriques sont conduites basées sur un modèle de coût pour comparer les performances proposées. Finalement, une validation réelle sur Oracle 11g est faite. Elle est assurée par un module de réécriture adaptant la charge initiale à la mise en cache donnée par nos algorithmes. Les résultats obtenus sont encourageants. Actuellement, nous validons nos propositions sur un jeu de données plus important et sur une plateforme conséquente (serveur de 32 Go de RAM).

# Références

Bellatreche, L., K. Boukhalfa, et H. I. Abdalla (2006). Saga : A combination of genetic and simulated annealing algorithms for physical data warehouse design. In *23rd British National Conference on Databases*, Number 212-219.

Bennett, K. P., M. C. Ferris, et Y. E. Ioannidis (1991). A genetic algorithm for database query optimization. In *in Proceedings of the 4th International Conference on Genetic Algorithms*, pp. 400–407.

Chaudhuri, S. (2004). Index selection for databases : A hardness study and a principled heuristic solution. *IEEE TKDE 16*(11), 1313–1323.

Chipara, O., C. Lu, et G.-C. Roman (2007). Real-time query scheduling for wireless sensor networks. In *RTSS*, pp. 389–399.

Choquet-Geniet, A. et E. Grolleau (2004). Minimal schedulability interval for real-time systems of periodic tasks with offsets. *Theoret. Comp. Science 310*, 117–134.

Chou, H.-T. et D. J. DeWitt (1985). An evaluation of buffer management strategies for relational database systems. In *VLDB*, pp. 127–141.

Cornell, D. W. et P. S. Yu (1989). Integration of buffer management and query optimization in relational database environment. In *VLDB*, pp. 247–255.

Effelsberg, W. et T. Härder (1984). Principles of database buffer management. *ACM Trans. Database Syst. 9*(4), 560–595.

Gupta, A., S. Sudarshan, et S. Viswanathan (2001). Query scheduling in multi query optimization. In *IDEAS*, pp. 11–19.

Holland, J. H. (1975). *Adaptation in Natural and Artificial Systems*. The University of Michigan Press.

Kerkad, A., L. Bellatreche, et D. Geniet (2011). Heuristics for solving the integrated buffer management and query scheduling problem. Technical report, LIAS-ENSMA.

O'Neil, P., B. O'Neil, et X. Chen (2009). Star schema benchmark.

Ou, Y., T. Härder, et P. Jin (2010). Cfdc : A flash-aware buffer management algorithm for database systems. In *ADBIS*, pp. 435–449.

Phan, T. et W.-S. Li (2008a). Dynamic materialization of query views for data warehouse workloads. In *Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on*, pp. 436 –445.

Phan, T. et W.-S. Li (2008b). Dynamic materialization of query views for data warehouse workloads. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pp. 436–445.

Roy, P., K. Ramamritham, S. Seshadri, P. Shenoy, et S. Sudarshan. Don't trash your intermediate results, cache 'em. *CoRR*.

Russell, S. et P. Norvig (2003). *Artificial Intelligence : a modern approach* ($3^{th}$ ed.). Prentice Hall.

Sadeg, B. et L. Amanton (2005). Scheduling distributed real-time nested transactions. In *Proc. of International Symposium on Object-Oriented Real-Time Distributed Computing*, pp. 208–215. IEEE.

Schild, K. et J. Würtz (1998). Off-line scheduling of a real-time system. In K. Goerge et G. Lamong (Eds.), *Proc. of Symposium on Applied Computing*, Atlanta, GA, USA, pp. 29–38. ACM.

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

Tanenbaum, A. S. (2007). *Modern Operating Systems* ($3^{th}$ ed.). Prentice Hall.

Thomas, D., A. A. Diwan, et S. Sudarshan (2006). Scheduling and caching in multiquery optimization. In *COMAD*, pp. 150–153.

Wu, J., K.-L. Tan, et Y. Zhou (2009). Qos-oriented multi-query scheduling over data streams. In *DASFAA*, pp. 215–229.

Yang, J., K. Karlapalem, et Q. Li (1997). Algorithms for materialized view design in data warehousing environment. In *VLDB*, pp. 136–145.

Yang, M. et G. Wu (2011). Caching intermediate result of sparql queries. In *WWW (Companion Volume)*, pp. 159–160.

# Summary

Database management systems (DBMS) use external magnetic devices (disks) for the storage of mass data. In the context of very large databases, the query response time may be strongly influenced by two main factors: (i) the amount of data needed to be accessed from the disk to the main memory cache and (ii) the order of queries, especially if queries share some common intermediate results. These two factors are related to two well-known problems in databases: Buffer Management Problem (BMP) and Query Scheduling Problem (QSP). They represent also a crucial aspect for selecting various optimization structures (materialized views, partitioning, etc.). Usually, BMP and QSP are treated in an isolated way. As a result, their strong interaction is always ignored by algorithms selecting optimization structures. This situation motivates us to consider the integrated problem including BMP and QSP. In this paper, we first propose a formalization of the integrated problem and show complexity. Secondly, three heuristics dealing with our integrated problem are proposed: affinity, hill climbing and genetic algorithms. Finally, two types of intensive experiments are conducted: (1) theoretically using a cost model and (2) a real validation on Oracle11G.