Votre Plan d'Exécution de Requêtes est un Circuit Intégré : Changer de Métier

Ahcène Boukorca*, Ladjel Bellatreche*, Sid-Ahmed Benali Senouci **, Zoé Faget *

* LIAS/ISAE-ENSMA, Téléport 2 - 1 avenue Clément Ader - BP 40109,
86961 Futuroscope Chasseneuil Cedex - France
(boukorca,bellatreche, zoe.faget)@ensma.fr,
** Mentors Graphics
38330 Montbonnot-Saint-Martin - France
sid-ahmed_senouci@mentor.com

Résumé. Dans la première génération des bases de données, les optimiseurs étaient conçus pour optimiser des requêtes individuelles. Après identification des interactions entre les requêtes, des optimiseurs étaient proposés pour offrir une optimisation multiple. La difficulté de cette optimisation est l'identification des expressions communes entre les requêtes. Ce problème est connu comme NP-difficileSellis et Ghosh (1990). Pour résoudre ce problème, des solutions basées sur la fusion des arbres algébriques des requêtes ont été proposées. Elles souffrent du problème de passage à l'échelle. Dans cet article, nous proposons l'utilisation de la théorie de graphes fortement utilisée dans le domaine de la conception des circuits intégrés pour la résolution de ce problème. Premièrement, nous définissons l'analogie entre notre problème et celui de la conception des circuits électroniques. Des algorithmes issus de la théorie des graphes sont ensuite proposés. Finalement, une évaluation de nos propositions est effectuée à l'aide de l'outil HMETIS.

1 Introduction

L'accès efficace à la donnée a toujours été un enjeu important dans le monde des bases de données. Cette dimension s'est accentuée avec l'arrivée du big data. Cet accès est assuré par l'intermédiaire de requêtes complexes souvent interdépendantes (Ahmad et al. (2011); Sellis (1988); Yang et al. (1997a)). Les utilisateurs des bases de données exigent la performance des accès aux données. Pour satisfaire cette contrainte, les optimiseurs de requêtes se sont développés. L'optimiseur a deux tâches principales à accomplir en un temps raisonnable : d'abord, il lui faut trouver l'ensemble des plans d'exécution pour une même requête. Ensuite, il lui faut choisir parmi ces alternatives la plus performante pour exécuter la requête en se basant sur certains critères. L'existence de ces multiples alternatives est due aux caractéristiques des opérations algébriques (commutativité, associativité du produit cartésien, inversion sélection-projection, inversion sélection-produit cartésien, etc.). Deux types d'optimisateur existent : (i) les optimiseurs de requêtes individuelles et (ii) les optimiseurs de requêtes multiples. Dans

le premier type, deux générations d'optimiseurs existent : (i) des optimiseurs basés sur des règles ($\mathcal{OBR}$) et (ii) des optimiseurs basés sur des modèles de coût. Dans la première génération, chaque requête est représentée par un arbre algébrique dont les feuilles représentent les tables de base, les noeuds intermédiaires décrivent soit une opération unaire (sélection, projection) ou une opération binaire (jointure, union, etc.), et la racine de cet arbre représente le résultat. Dans l'$\mathcal{OBR}$, un ensemble de règles est appliqué sur l'arbre algébrique. On peut citer : (a) l'exécution des sélections aussitôt que possible, (b) la combinaison de certaines sélections avec un produit cartésien précédent pour aboutir à une jointure, (c) la combinaison de séquences d'opérations unaires, (d) si le résultat d'une sous expression commune (une expression apparaissant plus d'une fois) n'est pas une grande relation, et si elle peut logée dans le buffer, alors il est avantageux de pré-calculer la sous-expression commune une fois et d'en mémoriser le résultat pour d'autres usages postérieurs. L'$\mathcal{OBR}$ a été récemment écartée par les optimiseurs car elle ne prend pas en compte les critères comme la taille des tables, les facteurs de sélectivité des prédicats de sélection et de jointure, l'ordre de jointure, la taille du buffer, etc. Pour pallier les défauts de l'$\mathcal{OBR}$, l'optimisation basée sur un modèle de coût ($\mathcal{OBMC}$) a été proposée. Pour chaque requête, l'optimiseur effectue les tâches suivantes : (1) la génération de tous les plans d'exécution, (2) l'énumération des tous les implémentations possibles pour une opération d'un plan. Par exemple, l'opération de sélection peut être implémentée en utilisant soit une lecture séquentielle ou un scan d'index. Une opération de jointure peuvent être mis en œuvre de différentes manières : boucle imbriquée, jointure par hachage, etc. (Figure 1) ; (3) la gestion d'autres opérateurs comme le tri ; (4) la gestion des résultats intermédiaires ; et finalement (5) l'optimiseur choisit plan d'exécution qui a le coût minimal. Cette optimisation est utilisée par la majorité des SGBD commerciaux.

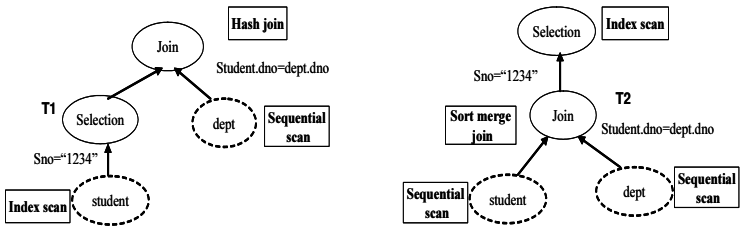

FIG. 1 – *Deux plans différents pour la même requête*

Comme son nom l'indique, l'optimisation multiple concerne un ensemble de requêtes ayant une forte interaction. Les applications d'entreposage de données ont favorisé cette interaction entre les requêtes, car les requêtes de jointure passent systématiquement par la table centrale qui est la table des faits. Au lieu de travailler sur un seul plan d'exécution, un plan global unifiant les plans individuels est défini (Sellis (1988)). La difficulté principale liée à la génération du plan global d'exécution des requêtes *(MVPP : Muli-Views Processing Plan)*, est double : (i) comment obtenir le plan global contenant les expressions communes et (ii) comment quantifier sa qualité. Ce problème est connu comme NP-difficile, surtout lorsqu'il y a une charge de requêtes très importantes à optimiser. Les travaux existants traitant ce problème proposent des heuristiques combinant des plans individuels et à l'aide de modèle de

coût sélectionnent le meilleurs plans. Le problème majeur lié à ces solutions est leur passage à l'échelle.

Dans le soucis de réduire cette complexité et d'avoir des algorithmes performant pour la génération de graphe unifié, nous nous sommes approchés de Dr. Senouci Sid Ahmed Benali, un spécialiste dans le domaine de partitionnement de circuits électroniques. Il est actuellement chef de projet Graphe et partitionnement de circuits microélectronique au sein de l'entreprise Mentor graphics [1], localisée dans la région de Grenoble, France. Lors de cette discussion, nous avons identifié la similarité entre un plan de requêtes unifié et un circuit électronique (Fig. 3). Un plan global de requêtes intégré peut être représenté par un hypergraphe. Les hypergraphes ont pour caractéristique que leurs hyperarêtes relient plusieurs sommets sans aucune exigence d'ordre entre ces sommets. En projetant notre problème, on trouve que les sommets (nœuds) d'un hypergraphe représentent les opérations des requêtes et les hyperarêtes représentent les requêtes. Chaque hyperarête relie les opérations de la requête. Cette analogie nous permet alors d'utiliser des algorithmes et des outils utilisés dans le domaine des circuits électroniques afin d'assurer le passage à l'échelle, sachant que les circuits électroniques manipulent un nombre important de portes (dans notre cas les portes représentent les nœuds intermédiaires).

Dans cet article, nous proposons des solutions empruntées au domaine de la microélectronique pour l'optimisation multi-requêtes. Nous avons adapté les outils de partitionnement utilisés dans la microélectronique pour répondre au problème de la sélection des vues matérialisées.

2 État de l'art

L'optimisation multi requêtes a été largement étudiée dans le contexte des bases de données traditionnelles (Sellis (1988)) et récemment dans les bases de données sémantiques (Le et al. (2012)). La difficulté de cette optimisation est l'identification des nœuds intermédiaires. Cette tâche est souvent réalisée à l'aide d'arbres. Rappelons que chaque requête peut être représentée sous forme d'un arbre algébrique (ElMasri et Navathe (1994)). La fusion des arbres donne lieu à un graphe global. Ce type d'optimisation a contribué à la résolution du problème de sélection des vues matérialisées (Gupta (1999); Yang et al. (1997b); Baralis et al. (1997)). Yang et al. (Yang et al. (1997a)) est le seul travail important présenté dans le contexte des entrepôts de données sur la génération de plan unifié de requêtes afin de matérialiser les nœuds intermédiaires. Etant donné que chaque requête peut avoir plusieurs arbres algébriques, les auteurs sélectionnent l'arbre optimal (en fonction d'un modèle de coût). Une fois les arbres optimaux identifiés, l'algorithme essaye de trouver des expressions communes entre ces arbres (ou nœud partagé). Finalement, les arbres sont fusionnés en un seul graphe, appelé *plan multiple d'exécution des vues*, en utilisant les nœuds partagés identifiés. Ce graphe a plusieurs niveaux. Les feuilles sont les tables de base de l'entrepôt et représentent le niveau 0. Dans le niveau 1, nous trouvons des nœuds représentant les résultats des opérations algébriques de sélection et de projection. Dans le niveau 2, les nœuds représentent les opérations ensemblistes comme la jointure, l'union, etc. Le dernier niveau représente les résultats de chaque requête. Chaque nœud intermédiaire de ce graphe est étiqueté par le coût de l'opération algébrique (sélection, jointure, union, etc.) et le coût de maintenance. Ce graphe est utilisé pour

1. Mentor Graphics compte parmi les leaders mondiaux du marché de l'automatisation de la conception électronique (**EDA**, Electronic Design Automation) http ://www.mentor.com/france/

rechercher l'ensemble des vues dont la matérialisation minimise la somme des coûts d'évaluation des requêtes et de maintenance des vues. La solution prend en considération l'existence de plusieurs expressions possibles pour une requête donnée. Chaque nœud intermédiaire est considéré comme une vue potentielle. Deux algorithmes ont été proposés. Le premier, appelé "A feasible Solution", génère tous les MVPP possibles à partir des plans individuels optimaux des requêtes, puis choisit celui qui a le coût minimum. Cet algorithme est très coûteux en terme de calcul et ne donne pas forcement le MVPP optimal. Pour simplifier l'algorithme précédent et augmenter l'espace de recherche, les auteurs ont proposé un autre algorithme basé sur la programmation binaire 0-1. Les différentes étapes cet algorithme sont : (1) générer tous les plans possibles p_i pour chaque requête ; (2) identifier tous les sous arbres de jointures possibles s_i pour chaque plan généré ; (3) construction d'une matrice d'usage binaire *A*, où le coefficient a_{ij} représente la possibilité ou non d'exécution de la requête q_i par le plan p_j ; (4) construction d'une matrice binaire *B*, où le coefficient b_{ij} représente l'appartenance ou non du sous-arbre de jointure s_j au plan p_i. Finalement, le problème de sélection de MVPP optimal est réduit à la sélection d'un ensemble de plans individuels qui minimise le coût d'exécution de la charge de requêtes $Cout_{total}$. Chaque requête q_i n'utilisant qu'un seul plan p_i, on a finalement $Cout_{total} = \sum_{i=1}^{m} \sum_{j=1}^{l} Ecost(s_j) b_{ij}$, où $Ecost(s_i)$ représente le coût de construction de sous-arbre s_i.

3 Analogie entre MVPP et circuit électronique

Avant de proposer nos contributions, nous discutons l'analogie entre le MVPP et un circuit électronique. Considérons le schéma de l'entrepôt de données SSBB [2]. L'entrepôt contient une table de faits *Lineorder*, et quatre tables de dimension : *Customer, Supplier, Part* et *Dates*. Sur ce schéma un ensemble de 30 requêtes est considéré. La Figure 2 décrit un MVPP. Lors de la discussion avec Dr. Senouci nous l'avons dessiné sous forme d'un circuit électronique où les noeuds intermédiaires deviennent des ports électroniques (AND, OR, XOR). La Figure 3 décrit cette analogie.

Notre travail entre dans le cadre d'optimisation multi-requêtes : étant donné un ensemble de requêtes, trouver l'ordre d'exécution et la meilleure combinaison possible des différentes opérations afin d'optimiser le coût global de la charge. A cette fin, nous présentons dans ce papier une nouvelle approche de génération du *MVPP* d'une charge de requêtes. Cette approche consiste à construire un MVPP par bloc de requêtes, sans passer par les plans optimaux individuels des requêtes, à contrario de la plupart des approches proposées dans la littérature comme celle de Yang (Yang et al. (1997a)). Dans l'approche de Yang, on peut parler de méthodes *ascendantes*, qui s'appuient sur les plans élémentaires pour construire le plan global.

A l'inverse, notre approche est une méthode *descendante*. Le MVPP va être généré en premier et les plans individuels des requêtes en seront déduits directement. L'idée fondamentale pour la génération du plan global d'exécution, basée sur le partage des nœuds entre les différentes requêtes, est de maximiser la réutilisation des résultats intermédiaires afin d'optimiser la charge de requêtes dans son ensemble (Gupta et al. (2001)).

Dans l'approche proposée nous manipulons en particulier les opérations de jointure qui sont les plus coûteuses pour les requêtes OLAP dans un entrepôt de données (Yang et al.

2. http ://www.cs.umb.edu/poneil/StarSchemaB.pdf

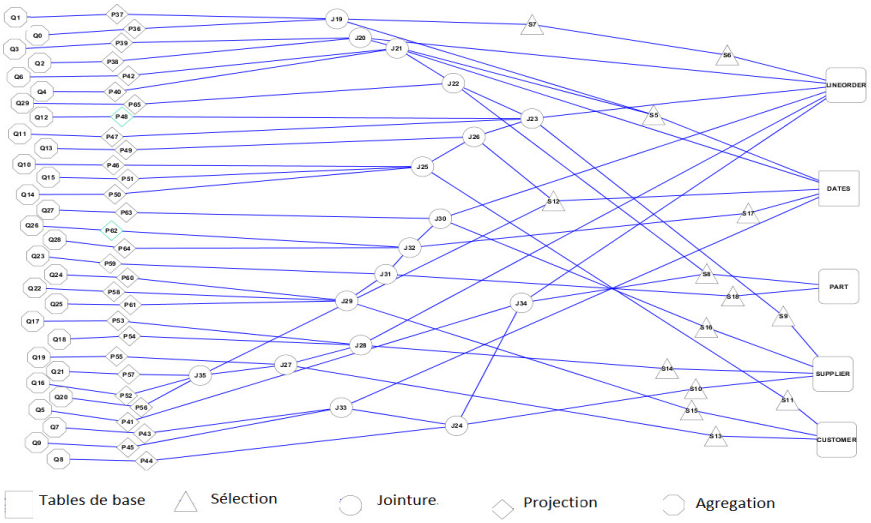

FIG. 2 – *Un MVPP global*

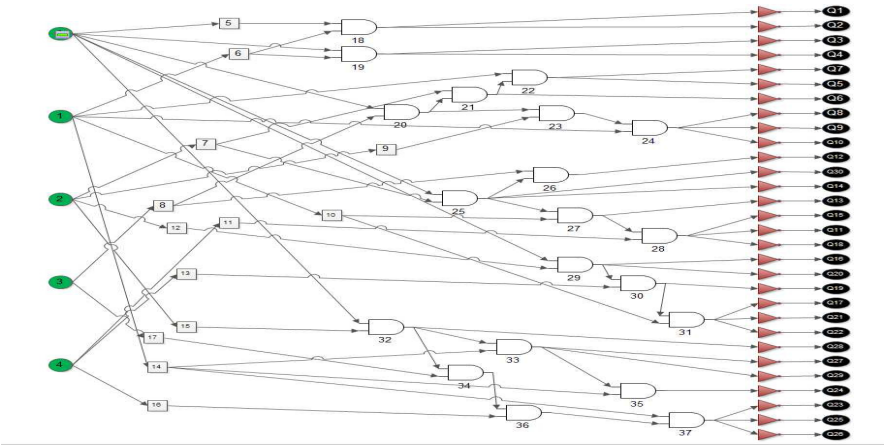

FIG. 3 – *Circuit correspondant à notre MVPP*

(1997a); Kerkad et al. (2012)). L'ensemble des nœuds de jointure sera divisé en plusieurs sous-ensembles disjoints appelés *composantes connexes*. Chaque *composante connexe* est un ensemble de nœuds, qui peuvent avoir des lien entre eux, pouvant être traitée indépendamment des autres composantes. A l'intérieur de chaque composante, les différents nœuds sont arrangés afin de construire le graphe global représentant le plan d'exécution optimal des requêtes de la composante.

Cette approche présente deux avantages motivants : (1) la division de la tâche de construction d'un plan global d'exécution en plusieurs sous-tâches indépendantes, pouvant de plus être exécutées en même temps (système parallèle par exemple), et simplifiées du fait du nombre limité de requêtes pour chaque sous-ensemble ; et (2) l'utilisation d'algorithmes de la théorie des graphes ayant prouvé leur efficacité en terme de vitesse de calcul et de résultats.

Notre approche de construction du MVPP d'une charge de requêtes consiste à réordonner les nœuds de jointure de sorte à maximiser le partage de ces nœuds entre les différentes requêtes, et donc à augmenter la réutilisation des nœuds intermédiaires lors de l'exécution des requêtes.

Il est bien connu que, pour une requête donnée, plusieurs plans d'exécution sont possibles suivant la combinaison choisie des opérations intermédiaires de sélection, jointure, projection etc. Si le résultat de la requête reste le même, le coût de deux plans d'exécution différents peut en revanche varier en terme d'Entrée/Sortie. Le plan optimal d'une requête est celui dont la combinaison des différentes opérations va donner un minimum de coût.

Afin de diminuer le coût d'exécution de la charge de requêtes, il est nécessaire de trouver les combinaisons des nœuds pour chaque requête afin de maximiser la réutilisation de certains résultats intermédiaire. L'utilisation de plans individuels non-optimaux peut donc donner de meilleur résultats (Gupta et al. (2001)).

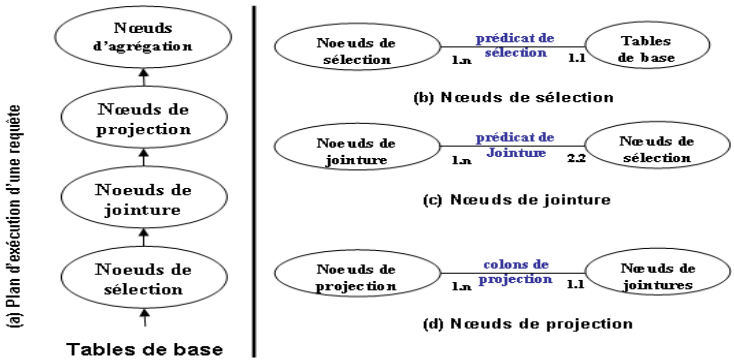

FIG. 4 – *Forme générale d'un plan d'exécution pour une requête*

Notre but est donc de trouver une combinaison des différents noeuds qui permette la construction d'un meilleur MVPP. Dans l'exécution des requêtes de type OLAP dans un entrepôt de données, les opérations de jointure sont les plus coûteuses (Kerkad et al. (2012); Antoshenkov et Ziauddin (1996)). Afin de minimiser le coût de ces opérations, on définit un ordre entre les différents types de nœuds, présenté par la figure 4a. Cet ordre est instauré par : (1) Pousser vers le bas les nœuds de sélection pour diminuer le coût des opérations de jointure

et éviter le produit cartésien des tables de base (Yang et al. (1997a)). En particulier, les requêtes de type OLAP ont toutes une jointure avec la table des faits, généralement très grande. La figure 4b illustre que tous les nœuds de sélection sont des résultats de l'opération de sélection sur des tables de base en appliquant un prédicat de sélection. (2) Les opérations de jointures sont appliquées sur les résultats de sélection. Les nœuds de jointure sont définis sans aucun ordre entre eux. Lorsqu'une opération de jointure utilise directement une table de base, on ajoute une opération de sélection sur cette table avec le prédicat vrai. La figure 4c montre que tous les nœuds de jointures sont des résultats d'opérations binaire sur des nœuds de sélection. (3) Les opérations de projection sont mises en haut. Nous avons évité de mettre les opérations de projection avant les opérations de jointure pour ne pas avoir une explosion de nombre de nœuds de jointure qui aurait pour conséquence de diminuer le nombre de réutilisation des résultats intermédiaires de ces opérations de jointure. (4) Les opérations d'agrégation sont à la fin si elles existent (elles sont ignorées dans notre cas). Les différents types de nœuds sont définis selon le paradigme suivant :

Nœuds de sélection : (1) Les nœuds de sélection sont déterminés par un prédicat de sélection. (2) déterminer *S*, l'ensemble des nœuds de sélection de la charge de requêtes. (3) Lorsque tous les nœuds de sélection de toutes les requêtes sont déterminés, on supprime de S les nœuds redondants (même table de base et même prédicat de sélection).

Nœuds de jointure : (1) Les nœuds de jointure sont déterminés par un prédicat de jointure et deux nœuds de sélection (2) déterminer *J*, l'ensemble des nœuds de jointure de la charge de requêtes. (3) Lorsque tous les nœuds de jointure de toutes les requêtes sont déterminés, on supprime de *J* les nœuds redondants (mêmes nœuds de sélection et même prédicat de jointure).

Nœuds de projection : Les nœuds de projection sont déterminés par un prédicat qui correspond aux colonnes projetées par les requêtes. Donc chaque requête aura un seul nœud de projection. Pour chaque requête, on détermine les nœuds de projection pour les placer dans *P*, l'ensemble des nœuds de projection de la charge de requêtes.

4 Les hypergraphes pour la génération de MVPP

Dans cette partie, nous décrivons d'abord la technique de représentation des opérations de jointure par un hypergraphe, appelé *hypergraphe de jointure*, puis l'algorithme de partitionnement choisi pour fractionner un hypergraphe en plusieurs hypergraphes disjoints. Enfin, nous présentons l'algorithme de transformation de l'hypergraphe en un graphe orienté optimal (ayant un coût minimum), qui représente le MVPP des requêtes.

4.1 Construction de l'hypergraphe de jointure

On commence par extraire tous les nœuds de sélection, de jointure, de projection et d'agrégation. L'ordre entre ces différents types de nœuds est déterminé par la forme générale de la requête exposée dans la figure 4a. Il reste à définir l'ordre des nœuds de jointure (opérations les plus coûteuses) qui permettra de minimiser le coût global. Notre but étant de maximiser la réutilisation des résultats des opérations de jointure par les différentes requêtes, il est indispensable de prendre en considération les relations et les interactions entre ces nœuds de jointure (Gupta et al. (2001)).

Plan global d'exécution des requêtes

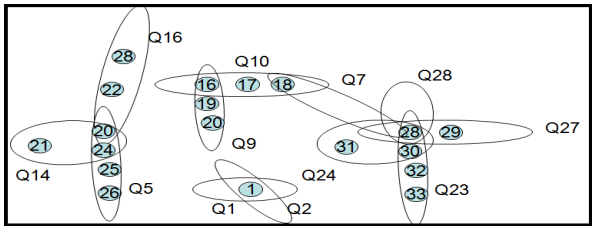

FIG. 5 – *Exemple d'un Hypergraphe de jointure*

L'idée de notre approche est de regrouper les nœuds de jointure en plusieurs composantes connexes, chacune contenant les nœuds qui peuvent avoir des interactions entre eux et donc la possibilité d'être réutilisés. Ce besoin nous a amené à penser à profiter des algorithmes de la théorie des graphes qui ont prouvé leur efficacité en termes de performance et de résultats (Karypis et al. (1999, 1997)).

Les nœuds de jointure extraits n'ont aucun ordre a priori, il est donc impossible de les représenter par des graphes orientés qui exigent qu'une arête reliant deux sommets ait un sens déterminé. En revanche, les hypergraphes permettent d'avoir des hyperarêtes reliant plusieurs sommets sans aucun ordre défini. Ainsi, nous avons choisi de représenter les groupes de requêtes ayant plusieurs nœuds de jointure par des hyperarêtes ayant plusieurs sommets.

Un hypergraphe est un ensemble de sommets V et un ensemble d'hyperarêtes E. Dans notre cas, V représente l'ensemble des noeuds de jointure tel que à chaque sommet $s_i \in V$ correspond un noeud de jointure n_j. De même, E représente la charge de requêtes Q, tel que à chaque hyperarête $h_i \in E$ correspond une requête q_j (une hyperarête h_i reliant les s_i de manière non-orientée correspond à une requête q_j, ensemble non-ordonné de noeuds de jointure n_j).

Un exemple d'hypergraphe de jointure est donné par la figure 5. L'hyperarête e_{23} correspond à la requête q_{23} qui relie les nœuds nj_{28},nj_{30},nj_{32} et nj_{33}, et l'hyperarête e_2 correspond à la requête q_2 qui n'a qu'un seul nœud nj_1.

La table 1 récapitule les correspondances entre la vision *graphe* et la vision *requête*.

Vision *graphe*	Vision *requête*
Hyperarête e_i	Requête q_j
Section (sous-hypergraphe)	Composante Connexe
Hypergraphe HA	Charge de requêtes Q
Graphe orienté	Plan d'exécution global : MVPP

TAB. 1 – *Correspondance graphe-requête*

4.2 Partitionnement de l'hypergraphe

Dans cette partie, nous adaptons un algorithme existant issu de la théorie des graphes afin de rassembler les noeuds de jointure en composantes connexes. Dans un premier temps, nous

avons utilisé des programmes de *HMETIS*[3] pour diviser l'hypergraphe en plusieurs hypergraphes (Karypis et Kumar (1999)). *HMETIS* est une bibliothèques de logiciels gratuits, développés par le laboratoire *Karypis*. Ces logiciels permettent de faire des partitionnements en série ou en parallèle d'hypergraphes. Les algorithmes de *HMETIS* sont basés sur un partitionnement multi-niveaux, le principe étant de diviser un hypergraphe en k partitions (nouveaux hypergraphes), de telle sorte que le nombre d'hyperarêtes coupées soit minimal (Karypis et al. (1999); Selvakkumaran et Karypis (2006); Karypis et Kumar (1999)). Plus précisément, les étapes principales de l'algorithme de partitionnement multi-niveaux sont : (1) Tout d'abord, on divise l'ensemble des sommets en deux sous-ensembles (appelés sections) de manière aléatoire. (2) Ensuite, on conduit une opération de raffinement afin de minimiser le nombre d'hyperarêtes coupées, c'est à dire d'hyperarête ayant des sommets dans deux sections différentes. Pour chaque hyperarête coupée, on déplace les sommets dans une partition (section) pour minimiser le nombre d'hyperarêtes coupées. L'opération de raffinement se répète jusqu'à ce qu'il n'y ait plus d'amélioration possible après un nombre déterminé d'itérations (dans notre cas nous avons utilisé 10 itérations). (3) Chaque bissection est ensuite re-divisée de la même façon jusqu'à obtenir le nombre de partitions demandé, k partitions.

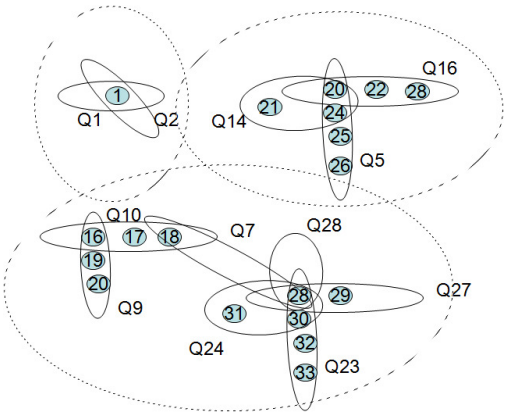

FIG. 6 – *Partitionnement d'un Hypergraphe de jointure*

Cependant, dans notre contexte, le nombre exact de partitions à faire est inconnu, en revanche nous voulons avoir toutes les partitions disjointes possibles (composantes connexes). Pour cela, nous adaptons l'algorithme initial à notre problème en modifiant le critère d'arrêt afin que l'algorithme ne s'arrête que lorsque il n'est pas possible de partitionner sans couper d'hyperarêtes. Précisément, l'algorithme fonctionne comme suit : (1) Tout d'abord, on divise l'ensemble des sommets en deux sous-ensembles (deux hypergraphes) si et seulement si le nombre d'hyperarêtes coupées est nul. (2) Ensuite, chaque hypergraphe résultat du partitionnement sera re-divisé de la même façon jusqu'à ce qu'aucun hypergraphe ne soit divisible. Le résultat du partitionnement de l'hypergraphe initial est un ensemble de petits hypergraphes disjoints. Chacun de ces hypergraphe sera traité séparément dans le processus de construction du MVPP des requêtes.

3. http ://glaros.dtc.umn.edu/gkhome/metis/hmetis/overview

La figure 6 montre le résultat du partitionnement de l'hypergraphe de jointure présenté dans la figure 5 : trois hypergraphes correspondant à trois composantes connexes.

4.3 Transformation d'un hypergraphe en graphe

Dans les sections précédentes, nous avons présenté la construction de l'hypergraphe global de jointure et le fractionnement de cet hypergraphe en plusieurs hypergraphes disjoints. L'étape restante consiste à trouver un ordre entre les nœuds de jointure dans chaque petit hypergraphe pour construire le MVPP des requêtes, c'est à dire transformer chaque hypergraphe en un graphe orienté.

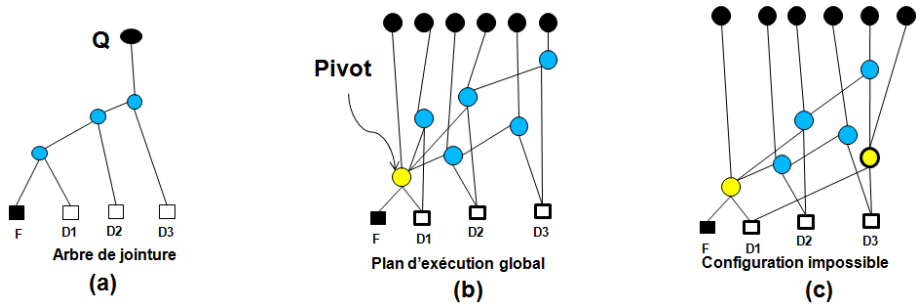

FIG. 7 – *Configurations possibles des nœuds de jointure dans un plan global d'exécution.*

Une caractéristique des requêtes OLAP est que toutes les opérations de jointure passent par la table des faits. La figure 7a, montre un arbre de jointure d'une requête dans un entrepôt de données. Les figures 7b et c montrent la configuration de positionnement des nœuds de jointure dans un MVPP. Cette configuration permet d'avoir toujours un nœud de départ qui correspond à la première opération de jointure. Ce nœud de départ est appelé un nœud *pivot*.

Algorithm 1 chercherPivot(HyperGraphe H)

1: $beneficemax \leftarrow 0$;
2: **for all** $v_i \in V$ **do**
3: $nbr \leftarrow nbrUse(v_i)$; {Avoir le nombre hyperarêtes qui utilisent le sommet v_i ; le nombre de réutilisation est nbr-1 }
4: $cost \leftarrow coutExecution(v_i)$; {le coût d'exécution de ce sommet v_i, dans le graphe G }
5: $benefice \leftarrow (nbr - 1) * cost - cost$ { Le bénéfice de réutilisation de sommet v_i }
6: **if** $benefice \succ beneficemax$ **then**
7: $beneficemax \leftarrow benefice$;
8: $pivot \leftarrow v_i$;
9: **end if**
10: **end for**
11: **return** $pivot$

Pour obtenir le MVPP, les hypergraphes résultats de l'opération de fractionnement doivent être transformés en un graphe orienté G qui représente le MVPP des requêtes. L'ajout d'une

orientation dans le graphe correspond à l'ordonnancement des nœuds de jointure de la requête. Pour cela, on cherche le nœud qui va maximiser le bénéfice de réutilisation de son résultat par les différentes requêtes par rapport au coût de sa première exécution (voir l'Algorithme 1). Le nœud qui a le maximum de bénéfice sera ajouté au graphe G, et supprimé de l'hypergraphe et ainsi de suite jusqu'à épuisement des nœuds.

Algorithm 2 transformerHyperGraphe(HyperGraphe H)

1: **while** V **not** $vide$ **do**
2: $pivot \leftarrow chercherPivot(H)$; { Algorithme 1}
3: **if** $pivot \in tous(\text{e} \in E)$ **then**
4: $ajouterSommetAuGraphe(pivot)$;{Ajouter le sommet $pivot$au graphe G}
5: $supprimerSommet(pivot)$;{Supprimer le sommet $pivot$au Vde H}
6: **for all** $\text{e}_i \in E$ **do**
7: **if** $|\text{e}_i|$=0 **then**
8: $supprimerHyperArete(\text{e}_i)$;{Supprimer les hyperarête qui ont pas de sommet}
9: **end if**
10: **end for**
11: **else**
12: $partitionnerPivot(pivot, H1, H2)$; {Algorithme 3}
13: $ajouterSommetAuGraphe(pivot)$;
14: $TransformerHyperGraphe(H1)$;
15: $TransformerHyperGraphe(H2)$;
16: **end if**
17: **end while**

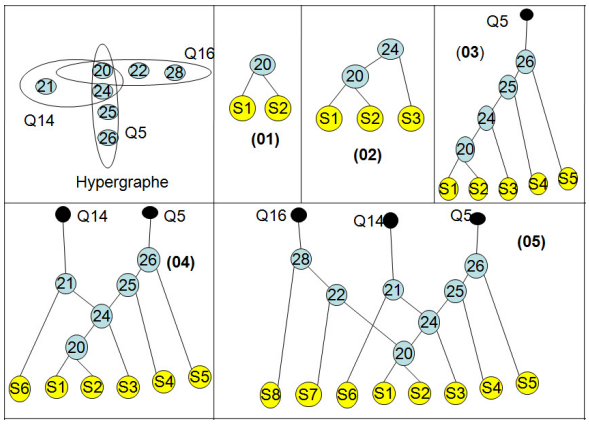

FIG. 8 – *Étapes de transformation d'un hypergraphe vers un graphe orienté*

L'Algorithme 2 décrit les étapes à suivre pour transformer un hypergraphe en un graphe orienté. (1) On commence par chercher le nœud $pivot$ de bénéfice maximum. (2) Si le nœud $pivot$ est utilisé par toutes les hyperarêtes (autrement dit toutes les requêtes), alors le nœud

$pivot$ sera ajouté au graphe G et supprimé de l'hypergraphe. Lors de la suppression du nœud $pivot$, les hyperarêtes qui ne relient plus aucun sommet sont supprimées automatiquement. (3) Si le nœud $pivot$ n'est pas utilisé par toutes les hyperarêtes, alors le nœud $pivot$ sera ajouté au graphe GRAPHE, et l'hypergraphe sera fractionné en deux hypergraphes disjoints *H1* et *H2* par l'algorithme 3. *H1* englobe les hyperarêtes qui utilisent le nœud $pivot$et *H2* comporte les autres hyperarêtes. Ces deux hypergraphes seront transformés de la même façon en des graphes orientés dans le graphe global G. La figure 8 montre les étapes d'exécution de l'algorithme de transformation d'un hypergraphe en un graphe orienté.

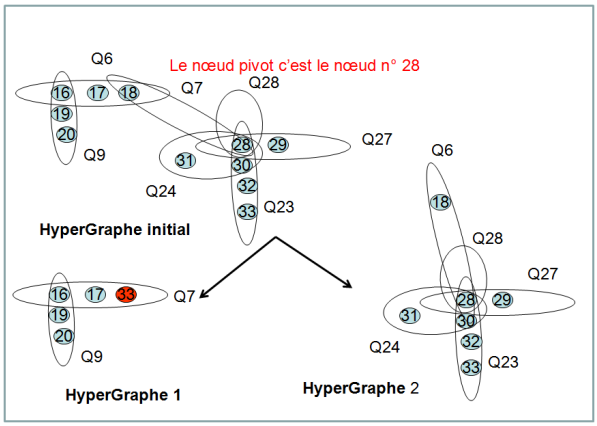

FIG. 9 – *Exemple de fractionnement d'un hypergraphe via un nœud pivot*

Algorithm 3 partitionnerPivot(Sommet $pivot$, HyperGraphe *H1*, HyperGraphe *H2*)

1: **for all** $e_i \in E$ **do**
2: **if** $pivot \in e_i$ **then**
3: $ajouterGraphe(e_i, H1)$;{Ajouter hyperarête e_i au premier hypergraphe H1 }
4: **else**
5: $ajouterGraphe(e_i, H2)$;{Ajouter hyperarête e_i au deuxième hypergraphe H2 }
6: **end if**
7: **end for**
8: **for all** $v_i \in H2.V$ **do**
9: **if** $v_i \in H1.Vi$ **then**
10: $changerIdentifiant(v_i, H2)$;{Changer le identifiant de sommet pour éviter le conflit au plan global }
11: **end if**
12: **end for**

L'Algorithme 1 permet de chercher le sommet (nœud) pivot dans un hypergraphe H qui donne le meilleur bénéfice possible en terme de nombre de réutilisations de son résultat par rapport à son coût d'exécution. Le bénéfice est égal au nombre de réutilisation multiplié par le coût d'exécution de ce nœud dans le graphe orienté, moins le coût d'exécution.

L'Algorithme3 permet de fractionner un hypergraphe en deux hypergraphes suivant un nœud $pivot$. Le premier hypergraphe comporte les hyperarêtes qui utilisent le $pivot$, et le deuxième comporte les autres hyperarêtes. Lorsque le deuxième hypergraphe contient des sommets qui sont utilisés par le premier hypergraphe, on duplique les sommets communs dans le deuxième hypergraphe. La figure 9 montre le résultat d'un exemple de fractionnement d'un hypergraphe via un nœud $pivot$ nj_{28}. Dans cet exemple, on remarque que le nœud nj_{18} étant présent dans les deux hypergraphes, le nœud nj_{33} l'a dupliqué dans l'hypergraphe 1.

5 Expérimentation

Pour valider notre proposition, un outil a été développé en java qui comporte les modules suivants : (1) Un module pour extraire tous les noeuds de jointure, sélection, projection et agrégation à partir d'un ensemble de requêtes introduites sous forme des chaînes SQL. (2) Un module de construction d'un hypergraphe de jointure, où chaque hyperarête de l'hypergraphe représente une requête et relie ses nœuds de jointure. (3) Un module de partitionnement de l'hypergraphe en plusieurs hypergraphes sans couper d'hyperarête. (4) Un module de transformation de chaque hypergraphe en un graphe orienté suivant une fonction qui maximise le bénéfice de réutilisation des nœuds par les requêtes. (5) Un module qui rassemble tous les nœuds de sélection, jointure ordonnées, projection et agrégation pour former le MVPP. (6) Un module d'affichage utilisant le plugin Cytoscape [4], qui permet un affichage visuel du MVPP de toute la charge de requêtes, pour une composante (FIG.11) ou les plans individuels de chaque requête.

Nbr de requête	Hypergraphe	Basique	Programmation 0-1
5	71	82	69
20	88	514	225
30	104	933	510
50	108	2258	781
60	114	4358	1224
499	283	25654	8604

TAB. 2 – *Temps en millisecondes pour la génération du MVPP*

Un autre module a été développé pour simuler les algorithmes de Yang (Yang et al. (1997a)). Ce module permet de : (1) calculer les plans optimaux pour chaque requête suivant une fonction d'évaluation de coût de plans. Dans ces plans individuels les nœuds de projection et de sélection sont en bas. (2) Monter les nœuds de sélection et de jointure en haut. (3) Construire tous les arbres de jointure possible de chaque plan individuel. (4) Déterminer le plan global en suivant la méthode basique qui teste toutes les combinaisons possibles des plan optimaux individuels et sélectionne le plan le moins coûteux. (5) Déterminer le plan global optimal en se basant sur la programmation binaire 0-1 (voir le détail de la méthode programmation binaire 0-1 au début de la section *contribution*). (6) Descendre les nœuds de sélection et de projection dans le plan global généré.

4. http ://www.cytoscape.org

Plan global d'exécution des requêtes

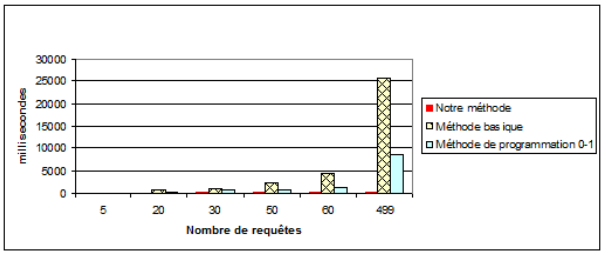

FIG. 10 – *Représentation graphique de temps de génération d'un MVPP*

Une série de tests a été appliquée sur le module de notre approche et de l'approche de Yang (pour les deux algorithmes : la solution basique et la solution de programmation 0-1). Dans chaque test, on change le nombre de requêtes en entrée pour voir le comportement des différents algorithmes. Les résultats obtenus, rassemblés dans le tableau 2, montre que notre algorithme est efficace en terme de calcul et temps de réponse même si la charge de requêtes est grande. En revanche, les algorithmes de Yang ont des temps de réponse qui augmentent exponentiellement avec le nombre de requêtes à traiter. L'efficacité de ces algorithmes est donc limitée lorsque la charge de requête est importante.

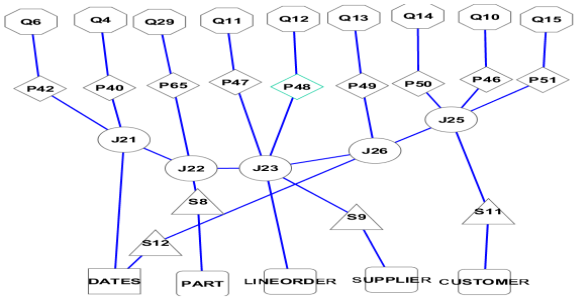

FIG. 11 – *Plan global d'exécution des requêtes d'une composante*

La figure 11 montre une présentation graphique par l'outil *Cytoscape* de quatre plans globaux d'exécution des requêtes pour quatre composantes différentes.

6 Conclusion

Dans ce papier, nous avons présenté une nouvelle vision de génération de plan global d'exécution d'une charge de requêtes en utilisant des algorithmes de la théorie des graphes. L'utilisation de ces algorithmes permet de diviser le travail en plusieurs tâches indépendantes, ce qui ouvre une perspective de programmation parallèle de génération des plans d'exécution. Les résultats expérimentaux ont montré que cette approche est efficace même si elle est appliquée sur une charge de requêtes très importante.

L'approche proposée est une fonction de combinaison des nœuds visant à maximiser le bénéfice de réutilisation de résultats intermédiaires. La génération du MVPP avec cette méthode permet d'avoir une meilleur configuration pour l'optimisation multi requêtes et elle va faciliter la sélection des structures d'optimisations comme les vues matérialisée. Elle permet également le regroupement des requêtes en plusieurs composantes, ce qui va faciliter l'ordonnancement de ces requêtes. Enfin, on peut envisager établir des profils de composantes de requêtes pour déduire directement le plan d'exécution des nouvelles requêtes.

Références

Ahmad, M., A. Aboulnaga, S. Babu, et K. Munagala (2011). Interaction-aware scheduling of report-generation workloads. *VLDB Journal 20*(4), 589–615.

Antoshenkov, G. et M. Ziauddin (1996). Query processing and optimization in oracle rdb. *The VLDB Journal 5*(4), 229–237.

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

ElMasri, R. et S. B. Navathe (1994). *Fundamentals of Database Systems*. Benjamin Cummings, Redwood City, CA.

Gupta, A., S. Sudarshan, et S. Viswanathan (2001). Query scheduling in multi query optimization. In *Proceedings of the International Database Engineering & Applications Symposium (IDEAS)*, Washington, DC, USA, pp. 11–19. IEEE Computer Society.

Gupta, H. (1999). Selection and maintenance of views in a data warehouse. Ph.d. thesis, Stanford University.

Karypis, G., R. Aggarwal, V. Kumar, et S. Shekhar (1997). Multilevel hypergraph partitioning : Application in vlsi domain. In *ACM/IEEE Design Automation Conference (DAC)*, pp. 526–529.

Karypis, G., R. Aggarwal, V. Kumar, et S. Shekhar (1999). Multilevel hypergraph partitioning : applications in vlsi domain. *IEEE Transactions on Very Large Scale Integration Systems 7*(1), 69–79.

Karypis, G. et V. Kumar (1999). Multilevel k-way hypergraph partitioning. In *ACM/IEEE Design Automation Conference (DAC)*, New York, NY, USA, pp. 343–348. ACM.

Kerkad, A., L. Bellatreche, et D. Geniet (2012). Queen-bee : query interaction-aware for buffer allocation and scheduling problem. In *Proceedings of the 14th international conference on Data Warehousing and Knowledge Discovery (DAWAK)*, pp. 156–167.

Le, W., A. Kementsietsidis, S. Duan, et F. Li (2012). Scalable multi-query optimization for sparql. In *ICDE*, pp. 666–677. IEEE.

Sellis, T. et S. Ghosh (1990). On the multiple-query optimization problem. *IEEE Transactions on Knowledge and Data Engineering 2*(2), 262–266.

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

Selvakkumaran, N. et G. Karypis (2006). Multiobjective hypergraph-partitioning algorithms for cut and maximum subdomain-degree minimization. *IEEE Transactions on Computer-*

Aided Design of Integrated Circuits and Systems 25(3), 504–517.

Yang, J., K. Karlapalem, et Q. Li (1997a). Algorithms for materialized view design in data warehousing environment. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, San Francisco, CA, USA, pp. 136–145. Morgan Kaufmann Publishers Inc.

Yang, J., K. Karlapalem, et Q. Li (1997b). A framework for designing materialized views in data warehousing environment. In *ICDCS*, pp. 458.

Summary

In the first generation of databases, the optimizers were designed to optimize individual queries. Once interaction between queries becomes a reality, optimizers propose solutions for multiple queries. Getting this optimization is known as NP-hard problemSellis et Ghosh (1990). In this paper, we deal with this problem using the graph theory that gives efficient solutions in the electronic design automation domain. Our proposal is compared with the existing studies using the HMETIS tool.