ESI : Ecole nationale Supérieure d'Informatique (Algérie)
ISAE-ENSMA : Ecole Nationale Supérieure de Mécanique et d'Aérotechnique (France)

THÈSE

En Cotutelle

pour l'obtention du Grade de

DOCTEUR DE L'ESI & DE L'ISAE-ENSMA

Écoles Doctorales :
Sciences et Technologies de l'Information et de la Communication (STIC, ESI)
Sciences et Ingénierie pour l'Information, Mathématiques (S2IM, ISAE-ENSMA)
Secteur de Recherche : INFORMATIQUE

Présentée par :

Rima BOUCHAKRI

Conception physique statique et dynamique des entrepôts de données

Thèse soutenue le 17 Septembre 2015

Directeurs de thèse : Ladjel BELLATRECHE et Walid-Khaled HIDOUCI

Jury :

Présidente	**Karima BENATCHBA**	Professeur, ESI, Algérie
Examinateurs	**Mohamed MEZGHICHE**	Professeur, UMBB, Algérie
	Mohand-Saïd HACID	Professeur, LIRIS/LYON1, France
	Yamine AÏT AMEUR	Professeur, ENSEEIHT/IRIT, France
	Ladjel BELLATRECHE	Professeur, ISAE/ENSMA, France
	Walid-Khaled HIDOUCI	Professeur, ESI, Algérie

Remerciements

Un grand merci à :

- **Ladjel BELLATRECHE**, mon directeur de thèse, pour m'avoir guidé tout au long de mon travail. Je le remercie pour sa disponibilité, pour ses précieux conseils et orientation, pour sa passion pour la recherche et le domaine scientifique qu'il a su me transmettre.

- **Walid-Khaled HIDOUCI** et **Sofiane MAABOUT** mes directeurs de thèse, pour leur soutien et leur disponibilité.

- Au corps administratif de l'ESI, particulièrement **Si Larabi KHELIFATI** pour sa bienveillance et son aide si précieuse et **Nacera CHERID** pour son écoute.

- Tous les membres du laboratoire LCSI à L'ESI particulièrement les responsables **Touraya TEBIBEL** et **Djamel Eddine ZEGOUR**.

- **A Kamel BOUKHALFA**, collègue de l'USTHB pour son soutien et aide inconditionnel.

- **Emmanuel GROLLEAU** pour avoir bien voulu m'accueillir au sein du laboratoire LIAS qu'il dirige.

- **Zoé FAGET**, pour son aide et contribution à la rédaction de certains articles.

- Tous les membres du laboratoire LIAS à l'ENSMA.

- Tout mes amis(es) sur les trois continents. En Algérie, France et Canada.

- Enfin et surtout à ma **famille** et particulièrement **ma mère** pour avoir toujours cru en moi. Rien n'aurait été possible sans elle.

A ma mère, ma raison d'être...

Table des matières

Table des figures

Introduction Générale

Contexte

Les entrepôts de données contiennent des données importantes qui sont intégrées, historisées et stockées dans des modèles logiques relationnels comme les schémas en étoile ou en flocon de neige [66]. Ces entrepôts sont exploités par des applications décisionnelles pour des fins d'analyse en ligne, à travers des requêtes décisionnelles dites *requêtes de jointures en étoile* très complexes et très gourmandes en temps d'exécution (des heures voir des jours). Ces requêtes contiennent de multiples opérations de sélection, de jointure et d'agrégation et sont complexes car elles renferment des opérations de jointures en étoiles entre table de faits volumineuse et tables dimension avec opérations de sélection sur les dimensions. Dans le but de satisfaire les exigences des décideurs en termes de temps de réponse, il est primordial d'optimiser ces requêtes. Cette optimisation est assurée par l'utilisation des structures d'optimisation sélectionnées lors de la phase de conception physique de l'entrepôt de données, comme la fragmentation horizontale [14], les vues matérialisées [83], l'ordonnancement des requêtes [54], l'indexation [86], la compression [97], etc.

L'indexation est l'une des premières structures d'optimisation proposées dans le contexte des bases de données et qui a montré ses performances [38, 94]. Cependant, les techniques d'indexation utilisées dans les bases de données de type OLTP (On-Line Transaction Processing) ne sont pas réellement adaptées aux environnements des entrepôts des données, car les requêtes de jointures en étoile accèdent à un très grand nombre de n-uplets ce qui conduirait à des index volumineux pas très efficaces [78, 39]. Par conséquent, dans le souci d'optimiser simultanément la sélection et la jointure, les *index de jointure binaires* ont été proposés et supportés par la plupart des SGBD commerciaux. Un index de jointure binaire est construit sur la table de faits suivant un ou plusieurs attributs appartenant à une ou plusieurs tables de dimension. Par conséquent, le nombre d'index possibles peut être très important ce qui rend la tâche de leur sélection difficile. Deux types d'index de jointure binaires existent ; les index simples définis sur un seul attribut et les index multiples définis sur deux ou plusieurs attributs des tables dimensions.

Une autre technique d'optimisation utilisée lors de la phase de conception physique des entrepôts de données est la fragmentation horizontale. Elle permet de répartir les instances d'une table, d'un index ou d'une vue matérialisée en un ensemble de partitions disjointes appelés *fragments horizontaux*[84]. C'est une structure d'optimisation dite non redondante car elle ne nécessite pas un espace de stockage supplémentaire. Elle est supportée par plusieurs SGBD commerciaux (Oracle, DB2, SQL Server, Sybase) et non commerciaux (PostgreSQL et MySQL). Deux types de fragmentation horizontale existent : la fragmentation horizontale primaire et la fragmentation horizontale dérivée. La fragmentation horizontale primaire permet de répartir les tuples d'une table suivent la conjonction de prédicats de sélection définis sur les attributs de la même table. Elle favorise les opérations de sélection portées sur les attributs de fragmentation [29]. La fragmentation horizontale dérivée quant

à elle, permet de fragmenter une table suivant la fragmentation horizontale primaire d'une seconde table, à condition de l'existence de relation père-fils entre les deux tables, ce qui favorise l'optimisation des opérations de jointures entres ces deux tables [47]. La fragmentation horizontale a suscité un grand intérêt dans le domaine de recherche. Premièrement, c'est une technique non redondante qui ne nécessite pas un espace de stockage supplémentaire. Deuxièmement, lors de l'exécution d'une requête, seules les partitions valides pour cette requête sont chargées ce qui réduit considérablement le coût d'exécution de la requête. Une partition valide pour une requête est une partition accessible par cette requête sur au moins un tuple. Troisièmement, elle constitue le point de départ dans la conception des entrepôts de données parallèles et distribués [80].

Vu la nécessité d'optimiser les requêtes de jointures en étoile, plusieurs travaux de recherche proposent des approches de sélection et d'implémentation de ces deux techniques d'optimisation. Ils commencent par formaliser le problème de sélection en un problème d'optimisation mono-objectif où il faux choisir un ensemble d'instances ou structures d'optimisation qui optimise les performances des requêtes sous des contraintes d'optimisations, puis proposent des algorithmes de résolution. D'abords, le problème de sélection des index de jointure binaire dans sa formalisation classique consiste à sélectionner une configuration d'index qui optimisent une charge de requêtes, préalablement connue, sans violer la contrainte d'espace de stockage. Dans le contexte d'entrepôt de données, vu le nombre importants des tables et des attributs concernés par la définition des index, le problème de sélection des index de jointures binaires est difficile et est prouvé NP-Complet. Ainsi, plusieurs algorithmes non exhaustifs ont été développés afin de résoudre ce problème. Ensuite, concernant la fragmentation horizontale d'un entrepôt de données, les travaux proposés commencent par formaliser le problème de sélection d'un schéma de fragmentation en un problème d'optimisation où il faut fragmenter les tables de l'entrepôt afin d'optimiser le coût d'exécution des requêtes sous certaines contraintes, puis ils proposent également des algorithmes de sélection qui répondent aux exigences formalisées.

Problématique et propositions

En analysant les travaux proposés pour sélectionner les techniques d'optimisation, nous remarquons que ces travaux se situent dans le cadre de sélection dite *statique* où les techniques d'optimisation sont sélectionnées lors de la phase de conception physique de l'entrepôt et ne changent plus. Ce type de sélection ne permet pas de faire face à l'évolution de l'entrepôt de données qui comporte deux aspects : l'évolution de charge par l'exécution de nouvelles requêtes décisionnelles et l'évolution de données par l'insertion continuelle de nouvelles données dans les tables de l'entrepôt. L'aspect statique de la sélection présente un handicape quant à l'efficacité d'optimisation des techniques implémentées. Les index de jointures binaires, par exemple, sont définis sur les attributs non-clé des tables dimensions figurant dans les requêtes. L'évolution de l'entrepôt de donnés peut rendre ces index obsolètes et périmés. Pour ce qui est de la fragmentation horizontale si l'entrepôt évolue, rien ne garantit l'efficacité du schéma de fragmentation pour optimiser la nouvelle charge de requêtes.

Nous avons remarqué également que la plupart des travaux de sélection exploite une sélection isolée où une seule technique d'optimisation est étudiée. Avec l'expansion continuelle des entrepôts de données et la complexité grandissante des requêtes, une seule technique ne suffit plus. Effectivement, la sélection d'un schéma d'optimisation est toujours soumise à des limitations est contraintes, comme l'espace de stockage à allouer pour stocker des index par exemple, ou encore la gestion du nombre importants de fragments des tables issus de la fragmentation. Par conséquent, il faut mettre en œuvre une approche de sélection dite jointe qui sélectionne plusieurs techniques d'optimisations afin

d'assurer un meilleur bénéfice tout en respectant les contraintes d'optimisation.

Une autre analyse des travaux de fragmentation et d'indexation nous a mené à faire une troisième constatation. Le problème de sélection d'une technique d'optimisation est toujours formalisé sous forme d'un problème d'optimisation mono-objectif où seul le coût d'exécution de la charge de requêtes est considéré comme un objectif à optimiser. La question que nous nous posons est : existe-t-il d'autres objectifs à optimiser afin de proposer des structures d'optimisation qui non seulement optimisent la charge de requêtes mais répondent également à d'autres exigences ? Notre étude de la sélection incrémentale a révélé que la mise à jour des techniques d'optimisation nécessite un temps d'implémentation et d'utilisation des ressources systèmes. Il est vrai que la sélection incrémentale permet de faire face à l'évolution de l'entrepôt en améliorant continuellement le coût d'exécution des requêtes mais elle engendre ce qu'on appelle *un coût de maintenance* de l'entrepôt de données qu'il faut optimiser également.

Première proposition : Sélection incrémentale isolée

Vu les inconvénients que présente la sélection statique, nous constatons l'intérêt d'étudier la sélection dite *incrémentale* des techniques d'optimisation. La sélection incrémentale effectue la mise à jour du schéma d'optimisation actuel d'un entrepôt de données pour prendre en compte son évolution et les changements survenus, ce qui garantit une continuelle optimisation des requêtes décisionnelles. Afin de faire face à l'évolution de l'entrepôt de données, nous proposons une démarche de sélection incrémentale dite isolée d'une technique d'optimisation. Dans un premier lieu, nous proposons une démarche de sélection incrémentale isolée des index de jointure binaires où nous considérons un entrepôt de données sur lequel est implémenté un ensemble d'index de jointure binaires. Notre démarche capture les changements qui surviennent sur l'entrepôt et effectue la mise à jour des index en ajoutant de nouveaux index bénéfiques et en supprimant les index périmés afin que tous les index implémentés répondent toujours aux exigences d'optimisation. Dans un second lieu, nous proposons une démarche de sélection incrémentale isolée d'un schéma de fragmentation horizontale. Nous considérons un entrepôt de données déjà partitionné. Cette seconde démarche prend en considération l'évolution de l'entrepôt et effectue une mise à jour de son schéma de fragmentation en réorganisant les données entre les différents fragments de l'entrepôt. Cette mise à jour garantie que le nouveau partitionnement de l'entrepôt réponde bien à l'optimisation des requêtes décisionnelles.

Seconde proposition : Sélection incrémentale jointe

Devant la complexité grandissante des requêtes de jointure en étoile, la sélection d'une seule technique d'optimisation devient de moins en moins bénéfique. L'intérêt principale de la sélection jointe et de combiner plusieurs techniques d'optimisation afin de couvrir l'optimisation d'un maximum de requêtes et de respecter au mieux les contraintes d'optimisation. De plus, il existe des techniques qui présentent des similarités ou des différences mais qui sont complémentaires où chaque technique pallie aux manques de l'autre. Récemment, une sélection statique jointe de la fragmentation et de l'indexation a été proposée. Elle exploite les similarités entre les deux techniques pour obtenir un meilleur schéma d'optimisation. Cependant, cette approche reste une approche de sélection statique et écope des inconvénients citées ci-dessus. De plus, il n'existe aucun travail sur la sélection incrémentale jointe de ces deux techniques. Ainsi, nous proposons d'exploiter les similarités entre les index de jointures binaires la fragmentation horizontale pour proposer une approche de sélection incrémentale jointe. Par analogie, nous étudions les similarités et complémentarité des index de jointure binaires

simples et ceux multiples afin d'étudier plus en approfondi leur sélection incrémentale jointe.

Troisième proposition : Optimisation multi-objectif

La sélection incrémentale des technique d'optimisation implique la mise à jour du schéma d'optimisation de l'entrepôt de données ce qui engendre un coût de maintenance. Nous visons donc à proposer une nouvelle formulation du problème de sélection incrémentale en un problème d'optimisation multi-objectif où il faut optimiser deux objectifs : le coût d'exécution des requêtes et le coût de maintenance de l'entrepôt de données. Nous constatons que cet axe de recherche est complètement vierge : il n'existe aucun travail qui traite de la sélection incrémentale et de l'optimisation multi-objectif à la fois dans le contexte d'entrepôt de données. De plus, il n'existe aucun travail qui traite de la sélection multi-objectif (statique ou incrémentale) des index ou de la fragmentation horizontale. Cela a suscité notre grand intérêt pour développer de nouvelles démarches de sélection qui répondent à cette problématique double.

Principales contributions

L'objectif de notre travail est de proposer une nouvelle vision de sélection des techniques d'optimisation qui comporte deux aspects importants : l'optimisation des entrepôts de données dans un cadre incrémentale et l'optimisation multi-objectif. Nous avons étudié deux techniques d'optimisation qui sont les index de jointures binaires et la fragmentation horizontale selon cette nouvelle vision de sélection. Nos travaux nous ont permis d'apporter les contributions suivantes :

1. Sélection incrémentale isolée des index de jointures binaires. Dans cette première sélection proposée, nous avons fait les contributions suivantes :
   - Proposer un nouvel algorithme de sélection des index de jointure binaires basé sur les algorithmes génétiques [18, 25, 21, 20].
   - Proposer une sélection incrémentale des index de jointure binaires qui permet de faire face à l'évolution de l'entrepôt de données et aux changements pouvant survenir [18, 25, 21, 20].
2. Sélection incrémentale isolée d'un schéma de fragmentation horizontale. Sélectionner un schéma de fragmentation est une tâche très complexe. Elle est d'autant plus complexe s'il faut considérer l'aspect incrémentale. Ainsi, nous avons fait les propositions suivantes :
   - Proposer une structure de données flexible pour coder un schéma de fragmentation [24].
   - Proposer une nouvelle vision de fragmentation basée sur la structure de donnée flexible qui peut prendre en charge la fragmentation statique ou incrémentale [19, 24, 7, 23].
   - Proposer un algorithme génétique qui exploite la structure de données pour la sélection statique et incrémentale [24].
   - Proposer une Algèbre dite Algèbre de Fragmentation $\mathcal{AF}$ qui modélise toute les opérations pouvant être appliquées sur la structure de données (un schéma de fragmentation) afin de l'adapter à l'évolution de l'entrepôt [23].
3. Sélection incrémentale jointe de deux techniques d'optimisation. Dans ce cadre, nous avons fait deux propositions
   - Sélection incrémentale jointe des index de jointures binaires simples et des index de jointures binaires multiples.
   - Sélection incrémentale jointe des index de jointures binaires et de la fragmentation horizontale.

4. Sélection incrémentale multi-objectif des techniques d'optimisation. Dans le cadre de cette proposition, nous avons apporté les contributions suivantes :
   - Formaliser le problème de sélection incrémentale d'un schéma de fragmentation en un problème multi-objectif qui vise à optimiser le coût de la charge de requêtes et le coût de maintenance d'un schéma de fragmentation.
   - Proposer un modèle de coût mathématique pour évaluer le coût de maintenance de l'entrepôt de données.
   - Proposer une résolution du problème de sélection multi-objectif.
   - Définir une architecture globale de sélection incrémentale multi-objectif [8, 7].

Organisation de la thèse

Cette thèse est organisée en deux parties principales :

Partie état de l'art

La première partie expose notre état de l'art. Le premier chapitre aborde les principaux concepts sur les index de jointures binaires et la fragmentation horizontale, et présente une analyse et classification des différents travaux de sélections de ces techniques pour optimiser les entrepôts de données. Ce chapitre présente également le positionnement de nos différentes propositions par rapport à l'existant. Le second chapitre aborde l'optimisation multi-objectif du point de vu définitions mathématiques et méthode de résolution ainsi que son application dans le contexte des entrepôts de données puis présente le positionnement de notre proposition dans la littérature.

Partie contributions

Nous exposons dans cette partie six chapitres qui expose nos principales contributions.

Dans le troisième chapitre, nous proposons une approche de sélection incrémentale des index de jointure binaires. Ce chapitre présente le codage utilisé pour codifier les configurations d'index, la méthode d'élagage de l'espace de recherche des index et l'algorithme génétique employé pour la sélection incrémentale des index simples et des index multiples. Le chapitre se termine par une large étude expérimentale sous le SGBD Oracle qui exploite un entrepôt de données réel pour valider les approches proposées et les comparer avec les travaux d'indexation existants.

Le quatrième chapitre expose notre démarche de sélection incrémentale d'un schéma de fragmentation. Il présente la structure de données flexible qui permet de codifier n'importe quel schéma de fragmentation. Ensuite, il expose l'algèbre dite algèbre de fragmentation qui exploite la structure flexible et permet de déterminer toutes les opérations requises pour passer d'un schéma de fragmentation à un autre. Ensuite, le chapitre décrit le Profiling des requêtes mis en œuvre à la base de l'algèbre de fragmentation qui contrôle la refragmentation de l'entrepôt. Après cela, il décrit la nouvelle architecture de sélection incrémentale basée sur le Profiling des requêtes. Finalement, le chapitre expose dans une étude expérimentale le test des approches proposées.

Le cinquième chapitre aborde la sélection jointe des index simples et des index multiples. Il expose une comparaison entre les deux types d'index, leurs principales différences et similarités, l'architecture de l'approche de sélection incrémentale jointe l'étude expérimentale menée sous Oracle 11g.

Le chapitre six aborde une approche de sélection incrémentale jointe des index de jointure binaires de la fragmentation horizontale. Ce chapitre est une continuité de nos travaux précédents sur la

sélection statique jointe de ces deux techniques. La démarche et architecture de cette approche est présentée et est validée avec une étude expérimentale sur Oracle.

Le chapitre sept aborde notre approche d'optimisation incrémentale multi-objectif de la fragmentation horizontale. Il expose la formalisation du problème d'optimisation multi-objectif, le modèle de coût de maintenance, l'approche de résolution du problème et l'architecture de sélection multi-objectif. Le chapitre est conclu par une étude expérimentale théorique et pratique sous Oracle.

Le chapitre huit présente l'architecture de l'outil qui permet d'implémenter toutes les propositions faites et permet de proposer à l'administrateur de l'entrepôt de données une assistance pour optimiser un entrepôt de données.

Le neuvième chapitre présente la conclusion générale de ce travail et esquisse de diverses perspectives.

Publications

Nous listons dans ce qui suit les publications concernant le travail de cette thèse.

Liste des articles portant sur les index de jointure binaires :

1. Rima Bouchakri, Ladjel Bellatreche, Zoé Faget, Sébastien Bress : Sélection statique et incrémentale des Index de Jointure Binaire : concepts, algorithmes, étude de performance. Journal of Decision Systems (JDS'12) Vol. 21(1) : 51-70 (2012) [20].
2. Rima Bouchakri, Ladjel Bellatreche, Khaled-Walid Hidouci : Static and Incremental Selection of Multi-table Indexes for Very Large Join Queries. 16th East-European Conference on Advances in Databases and Information Systems (ADBIS'12) : 43-56, 2012. [21]
3. Rima Bouchakri, Kamel Boukhalfa, Ladjel Bellatreche : Algorithmes de sélection des index de jointure binaires mono et multi-attributs. Ingénierie des Systèmes d'Information (ISI'11) 16(6) : 91-116 (2011) [25].
4. Rima Bouchakri, Ladjel Bellatreche : Sélection statique et incrémentale des index de jointure binaires multiples. 7ème Journée Francophone sur les Entrepôts de Données et l'Analyse en ligne (EDA'11), Revue des Nouvelles Technologies : 171-186, 2011 [18].

Liste des articles portant sur la fragmentation horizontale :

1. Rima Bouchakri, Ladjel Bellatreche, Zoé Faget : Algebra-based Approach for Incremental Data Warehouse Partitioning, 25th International Conference on Database and Expert Systems Applications (DEXA'14), Germany, 2014 [23].
2. Ladjel Bellatreche, Rima Bouchakri, Alfredo Cuzzocrea, Sofian Maabout : Horizontal partitioning of very-large data warehouses under dynamically-changing query workloads via incremental algorithms. 28th Annual ACM Symposium on Applied Computing (SAC'13) : 208-210, 2013 [7].
3. Rima Bouchakri, Ladjel Bellatreche : A Coding Template for Handling Static and Incremental Horizontal Partitioning in Data Warehouses. Journal of Decision Systems (JDS'13) Vol. 22, 2013 [24].

4. Ladjel Bellatreche, Rima Bouchakri, Alfredo Cuzzocrea, Sofian Maabout : Incremental Algorithms for Selecting Horizontal Schemas of Data Warehouses : The Dynamic Case. Conference on Data Management in Grid and P2P Systems (GLOBE'13) : 13-25, 2013 [8].
5. Rima Bouchakri, Ladjel Bellatreche : On Simplifying Integrated Physical Database Design. 15th East-European Conference on Advances in Databases and Information Systems (ADBIS'11) : 43-56, 2011. [17]
6. Rima Bouchakri, Ladjel Bellatreche : Sélection Incrémentale d'un Schéma de Fragmentation Horizontale d'un Entrepôt de Données Relationnel. 8ème Journée Francophone sur les Entrepôts de Données et l'Analyse en ligne (EDA'12), Revue des Nouvelles Technologies : 2-16, 2012 [19].

Articles collaboratifs :

1. Ladjel Bellatreche, Selma Khouri, Ilyès Boukhari, Rima Bouchakri : Using ontologies and requirements for constructing and optimizing data warehouses. International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO12) : 1568-1573, 2012.

Première partie

État de l'art

Chapitre 1

Sélection mono-objectif des techniques d'optimisation

Sommaire

1.1 Introduction

Les requêtes de jointure en étoile comportent des prédicats de sélection définis sur des tables de dimension et des prédicats de jointures entre la table des faits et les tables de dimension. Pour optimiser les sélections et les jointures, les chercheurs ont pensé à employer des techniques d'optimisations proposées dans le contexte des bases de données comme les index, la fragmentation horizontale, la fragmentation verticale et les vues matérialisées. Dans le cadre de ce travail, nous nous intéressons aux deux techniques les plus utilisés dans le contexte d'entreposage de données, à savoir les *index* et la *fragmentation horizontale*.

Les index ont en effet montré leurs performances dans les bases de données traditionnelles, on peut par exemple citer les *arbres B* et leurs variantes [38]. Un index peut être défini sur une seule colonne d'une table, ou sur plusieurs colonnes d'une même table. Il peut être clustérisé ou non clustérisé. Il peut également être défini sur plusieurs tables comme les index de jointure [94]. Avec l'arrivée des entrepôts de données, l'indexation constitue une option importante dans la phase de conception physique [34]. Cependant, les techniques d'indexation utilisées dans les bases de données de type OLTP (On-Line Transaction Processing) ne sont pas réellement adaptées aux environnements des entrepôts des données. En effet la plupart des transactions OLTP accèdent à un petit nombre de n-uplets, et les techniques utilisées (index B^+ par exemple) sont adaptées à ce type de situation. Les requêtes décisionnelles adressées à un entrepôt de données accèdent au contraire à un très grand nombre de n-uplets. Réutiliser les techniques des systèmes OLTP conduirait à des index avec un grand nombre de niveaux qui ne seraient donc pas très efficaces [78]. Afin d'offrir des solutions d'indexation adaptées au contexte des $\mathcal{ED}$, de nouveaux index ont été proposés. Nous pouvons citer : les index binaires, les index de jointure et les index de jointure binaires. (a) Les *index binaires* [30] optimisent les opérations de sélection définies sur des attributs appartenant à des tables de dimension. Ces index sont largement utilisés dans les bases de données XML [69, 48] et dans la recherche d'information [50]. (b) Les *index de jointures en étoile* permettant de stocker le résultat d'exécution d'une jointure en étoile entre plusieurs tables [91]. (c) Les *Index de Jointures Binaires* ($\mathcal{IJB}$) [77] permettant d'optimiser à la fois les jointures en étoile et les opérations de sélections définies sur les tables de dimensions, ce qui fait qu'ils sont bien adaptés pour optimiser les requêtes de jointure en étoile [89]. Vu le bénéfice que les index de jointures binaires apportent aux requêtes décisionnelles, ils sont très souvent employés lors de la phase de conception physique d'entrepôts de données. Par conséquent, plusieurs travaux s'intéressent à développer des processus de sélections de ce type d'index.

Afin d'optimiser les requêtes décisionnelles une autre technique d'optimisation est employée, à savoir la fragmentation horizontale ($\mathcal{FH}$) qui a déjà fait ses preuves dans le contexte des bases de données traditionnelles et d'entreposage de données. La fragmentation horizontale permet de répartir les instances d'une table, d'un index ou d'une vue matérialisée en un ensemble de partitions disjointes appelés *fragments horizontaux* [84]. Initialement, la fragmentation horizontale a été utilisée lors de la conception logique des bases de données relationnelles et orientées objet [64, 52]. Actuellement, elle est considérée comme une technique d'optimisation utilisée au niveau physique. C'est une technique non redondante car elle ne nécessite pas un espace de stockage supplémentaire [12]. Deux types de fragmentation horizontale existent : (a) la fragmentation horizontale primaire ($\mathcal{FHP}$) et (b) la fragmentation horizontale dérivée ($\mathcal{FHD}$) [80].

Nous nous intéressons dans ce travail aux index de jointures binaires et à la fragmentation horizontale car ce sont deux techniques d'optimisation souvent utilisées, lors de la phase de conception des entrepôts de données, pour optimiser les requêtes de jointures en étoiles. En effet, ces deux techniques optimisent les opérations de jointures entre table de faits volumineuse et plusieurs tables dimensions

avec opérations de sélection sur les tables dimensions. Avec l'augmentation continuelle de la taille des entrepôts de données et de la complexités des requêtes décisionnelles, une nouvelle tendance vise à implémenter conjointement les deux techniques au même temps afin de couvrir l'optimisation d'un plus large nombre de requêtes en respectant les contraintes existantes comme l'espace de stockage limité qu'on peut allouer aux index par exemple.

Nous présentons dans ce chapitre un état de l'art portant sur les travaux sur les index de jointures binaires et ceux portant sur la fragmentation horizontale. Ce chapitre est organisé en cinq section. La section 2 présenter un état de l'art sur la sélection des Index de Jointures Binaires et une classification des travaux de recherches. La section 3 décrit un état de l'art et une classification des travaux qui traitent de la sélection d'un schéma de fragmentation horizontale. La section 4 présente les concepts et travaux sur la sélection jointe des techniques d'optimisation. La section 5 conclu le chapitre.

1.2 Sélection Isolée des $\mathcal{IJB}$

Nous présentons dans cette section les Index de Jointures Binaires et leur fonctionnement, la formalisation du problème de sélection des index et sa complexité et les principaux travaux qui proposent des démarche de résolution du problème. Ensuite nous exposons une analyse et classification des travaux existants. Nous présentons deux définitions employés dans le contexte de sélection des index de jointures binaires :

- *Attribut de sélection* : c'est un attribut non-clé d'une table dimension figurant dans la clause WHERE d'une requête et utilisé comme critère de sélection. Supposant une table Clients dont la définition relationnelle est Client(CID, Prénom, Nom, Age, Ville). Soit la requête donnée ci-dessus. L'attribut Ville est appelé attribut de sélection car il permet de sélectionner les clients de la ville d'Alger.
  ```
  SELECT *
  FROM Client
  WHERE Ville='Alger'
  ```
- *Prédicat de sélection* : c'est un prédicat logique défini sur un attribut de sélection. Il est de la forme (A op V_i), où A est un attribut de sélection, V_i une valeur dans le domaine de A et op$\in \{=, <>, <, >, \leq, \geq\}$. Il peut également être de la forme A In $(V_1, \cdots, V_n)$.

1.2.1 Types d'$\mathcal{IJB}$

Un index de jointure binaires permet de pré-calculer les jointures entre une ou plusieurs tables de dimension et la table des faits dans les entrepôts de données modélisés par un schéma en étoile (O'Neil *et al.* 1995, 1997). Il est défini sur la table des faits en utilisant des attributs de sélection appartenant à une ou plusieurs tables de dimension. Il est composé d'un ensemble de vecteurs binaires muni du *RowID* qui permet au SGBD de calculer l'adresse physique de l'enregistrement. Nous distinguons deux types d'$\mathcal{IJB}$: *les $\mathcal{IJB}$ simples* et *les $\mathcal{IJB}$ multiples.*

1.2.1.1 Les $\mathcal{IJB}$ Simples IS

Les $\mathcal{IJB}$ simples, appelés aussi $\mathcal{IJB}$ mono-attributs, sont des index définis sur un seul attribut de sélection. Soit un attribut A appartenant à une table de dimension D. L'attribut A possède m valeurs distinctes $v_1, v_2, \cdots, v_m$. Supposons que la table des faits F est composée de p instances. La construction de l'index de jointure binaire défini sur l'attribut A se fait comme suit :

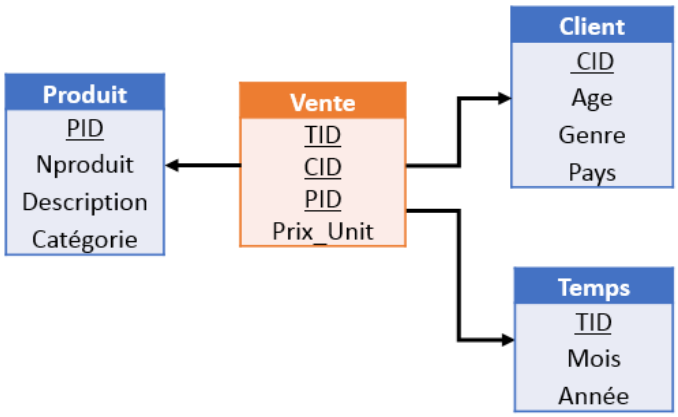


FIGURE 1.1 – Entrepôt de données modélisé en étoile

1. créer m vecteurs binaires (un vecteur pour chaque valeur v_k) composés chacun de p ligne ;
2. le $i^{ème}$ bit du vecteur d'une valeur v_k est mis à 1 si l'instance i de la table des faits est jointe avec l'instance de la table de dimension D tel que la valeur de A de cette instance est égale à v_k. Il est mis à 0 sinon.

La nature binaire des $\mathcal{IJB}$ permet d'améliorer les performances des requêtes en permettant d'appliquer des opérations logiques AND, OR, NOT, etc. Ces opérations permettent de rechercher des n-uplets vérifiant des conjonctions ou des disjonctions de prédicats. Les $\mathcal{IJB}$ sont très bénéfiques pour les requêtes de type *Count(*)* où l'accès unique à l'index binaire permet de répondre à ces requêtes. Notons que la taille d'un $\mathcal{IJB}$ est proportionnelle à la cardinalité des attributs indexés (nombre de valeurs distinctes). Pour comprendre le fonctionnement des $\mathcal{IJB}$ simples, nous présentons l'exemple suivant.

Exemple 1 *Soit l'entrepôt de données modélisé en étoile et composé d'une table des faits Ventes et trois tables de dimension Client, Produit et Temps représenté dans la figure 1.1. Soit la requête Q*1 *suivante :*

```
SELECT count(*)
FROM Ventes V, Produit P
WHERE V.CID=P.CID
AND C.Catégorie='Beaute'
```

Afin d'optimiser le coût d'exécution de Q_1, nous créons un $\mathcal{IJB}$ simple appelé IJB_ Catégorie sur l'attribut Catégorie de la table dimension Clients (figure 1.2(b)). La population des table Ventes et Clients est représenté par la figure 1.2(a). L'implémentation des $\mathcal{IJB}$ est supportée par la plupart des SGBD commerciaux. Ainsi, nous donnons dans ce qui suit la syntaxe Oracle pour la création de l'index IJB_ Catégorie.

```
CREATE BITMAP INDEX IJB_Catégorie
ON Ventes(Produit.Catégorie)
FROM Ventes V, Produit P
WHERE V.CID=P.CID
```

Pour optimiser la requête Q_1, l'optimiser lit le vecteur associé à la valeur 'Beauté'. Puisque que Q_1 est une requête Count(*), aucun accès aux tables n'est requis ; il suffit de calculer le nombre de "1" dans le vecteur résultat (figure 1.2(c).

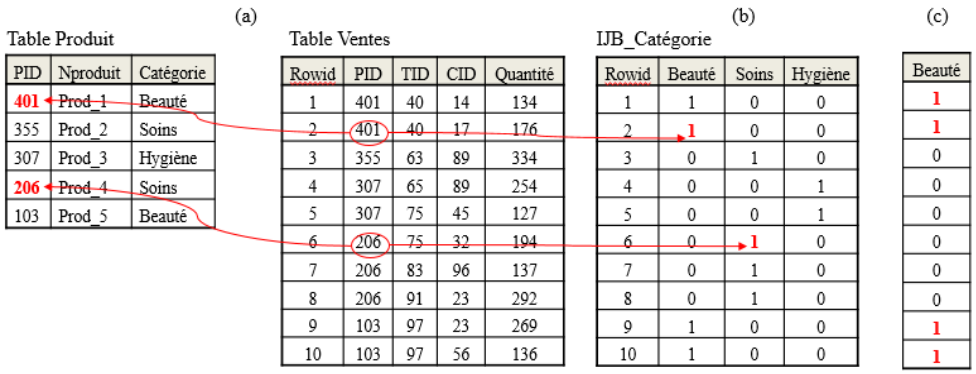


FIGURE 1.2 – Exemple d'index de jointure binaires simple

1.2.1.2 Les $\mathcal{IJB}$ Multiples *IM*

Les $\mathcal{IJB}$ multiples, appelés aussi $\mathcal{IJB}$ multi-attributs, sont définis sur plusieurs attributs issus d'une ou plusieurs tables dimensions. Ce sont donc des $\mathcal{IJB}$ *multi-tables*. Afin de comprendre le fonctionnement des $\mathcal{IJB}$ multiples, supposons la construction d'un $\mathcal{IJB}$ multiple sur un ensemble d'attributs $\{A_1, \cdots, A_n\}$ appartenant aux tables de dimension $\mathcal{D} = \{D_1, \cdots, D_d\}$. L'attribut A_i possède m_i valeurs distinctes $v_1^i, v_1^i, \cdots, v_{m_i}^i$. Supposons que la table des faits F est composée de p instances. La construction de l'index de jointure binaire défini sur les attributs $\{A_1, \cdots, A_n\}$ se fait comme suit :

1. créer m_i vecteurs binaires (un vecteur pour chaque valeur v_k^i) composés chacun de p ligne et cela pour chaque attribut A_i ;
2. le $j^{ème}$ bit du vecteur d'une valeur v_k^i est mis à 1 si l'instance j de la table des faits est jointe avec l'instance de la table de dimension D tel que la valeur de A_i de cette instance est égale à v_k^i. Il est mis à 0 sinon.

Dans l'exemple suivant, nous allons montrer la construction d'un $\mathcal{IJB}$ multiple.

Exemple 2 *Considérons l'entrepôt de données de la figure 1.1. Soit la requête Q_2 suivante :*

```
SELECT avg (quantité)
FROM Ventes V, Clients C, Produits P, Temps T
WHERE V.CID =C.CID AND V.PID =P.PID AND V.TID =T.TID
AND C.Ville= ('Poitiers' OR 'Alger')
AND T.Mois='Février'
AND P.Catégorie IN ('Beaute', 'Soins')
AND C.Genre='M'
```

Pour réduire le coût d'exécution de la requête Q2, un $\mathcal{IJB}$ multiple est crée sur les attributs Ville, Mois, Catégorie et Genre (figure 1.3(a)).

Nous donnons dans ce qui suit la syntaxe Oracle pour la création de l'index.

```
CREATE BITMAP INDEX IJB_VMCG
ON Ventes(Client.Ville, Temps.Mois, Produit.Catégorie, Client.Genre)
FROM Ventes V, Clients C, Produit P, Temps T
WHERE V.CID =C.CID AND V.PID =P.PID AND V.TID =T.TID
```

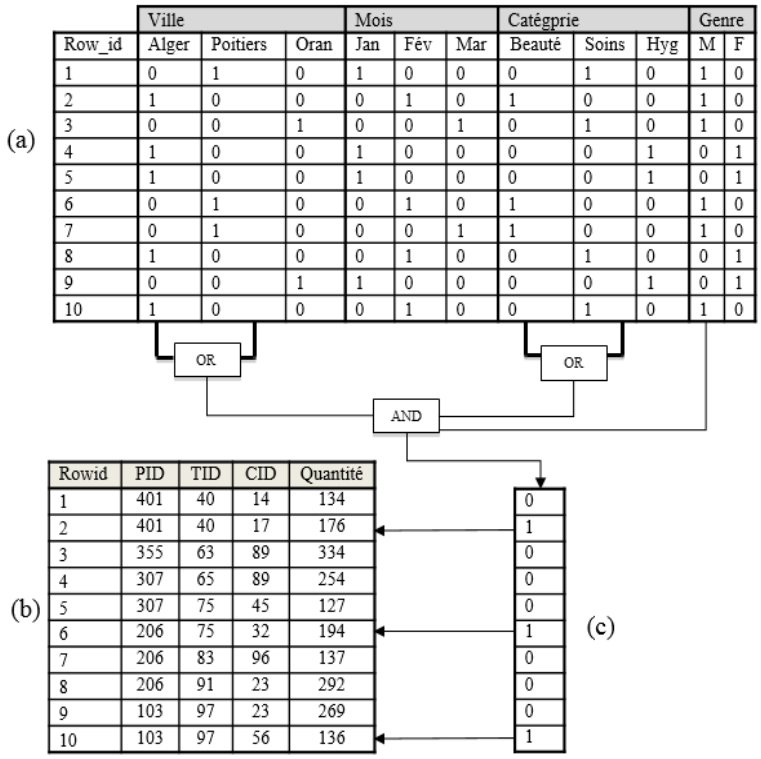


FIGURE 1.3 – Exemple d'index de jointure binaires multiple

Pour optimiser la requête Q_2, des opérations de AND, OR sont appliquées sur les vecteurs du $\mathcal{IJB}$ afin d'obtenir le vecteur résultat présenté dans la figure 1.3(c). Ensuite, l'optimiser accède aux n-uplets de la table de faits en utilisant les Rowid correspondant aux bits mis à 1 dans le vecteur obtenu. Il calcule la moyenne de quantité de ces n-uplets. Cet exemple montre bien l'efficacité des $\mathcal{IJB}$ pour l'optimisation de requêtes de jointures en étoile.

La nature binaire des $\mathcal{IJB}$ permet d'améliorer les performances des requêtes en permettant d'appliquer des opérations logiques AND, OR, NOT, etc. Ces opérations permettent de rechercher des n-uplets vérifiant des conjonctions ou des disjonctions de prédicats. Les $\mathcal{IJB}$ sont très bénéfiques pour les requêtes de type *Count(*)* où l'accès à l'index binaire seulement permet de répondre à ces requêtes. Notons que la taille d'un $\mathcal{IJB}$ est proportionnelle à la cardinalité des attributs indexés (nombre de valeurs distinctes). Pour cette raison, ils sont souvent recommandés pour les attributs de faible cardinalité.

1.2.2 Problème de sélection et travaux existants

Afin d'optimiser les requêtes de jointure en étoile, les index de jointures binaires sont construits sur des attributs non clé issus des tables dimensions. Cependant, au lieu de définir les index sur tout les attributs possibles, on les défini sur un sous ensemble d'attributs non-clés des tables dimensions utilisés par les requêtes appelés *attributs indexables*. Un attribut est indexable s'il est utilisé par un prédicat de sélection figurant dans la clause WHERE d'une requête de jointure en étoile. Cela assure l'optimisation des requêtes tout en excluant les attributs non pertinents.

Le problème de sélection des index dans sa formalisation général consiste à sélectionner un en-

semble d'index réduisant le coût d'exécution d'une charge de requêtes exécutées sur un entrepôt de données, sans violer la contrainte d'espace de stockage alloué pour stocker les index. Ce problème reste cependant difficile à résoudre, particulièrement dans le contexte d'entrepôt de donnés, vu le nombre important d'attributs indexables (plusieurs dizaines ou centaines d'attributs). De plus, les index peuvent être définis sur un ou plusieurs attributs issus d'une ou plusieurs tables. Par conséquent, plusieurs travaux existent dans la littérature proposant des démarches d'indexation des entrepôts de données. Ces travaux concernent généralement les index de jointures binaires qui sont très efficaces pour optimiser les opérations de jointures en étoile entre la table de faits et plusieurs tables dimensions avec opération de sélections sur les dimensions.

Nous présentons dans ce qui suit la formalisation du problème de sélection des $\mathcal{IJB}$, la complexité du problème et les principaux travaux qui proposent des démarches de résolution du problème. Nous terminons par présenter une analyse critique des travaux qui nous à conduite à proposer de nouvelles approches de sélection d'$\mathcal{IJB}$ dans les entrepôts de données.

1.2.2.1 Formalisation

Les $\mathcal{IJB}$ sont sélectionnés et implémentés lors de la phase de conception physique d'un $\mathcal{ED}$ et sont définis sur les attributs indexables. Le problème de sélection des $\mathcal{IJB}$ (PSIJB) est formalisé comme suit [2, 28] :
Étant donné :

- un $\mathcal{ED}$ modélisé par un schéma en étoile ayant d tables de dimension $\mathcal{D} = \{D_1, D_2, \cdots, D_d\}$ et une table des faits $\mathcal{F}$;
- une charge de requêtes $\mathcal{Q} = \{Q_1, Q_2, \cdots, Q_m\}$ à partir de laquelle un ensemble d'attributs indexables $AS = \{A_1, \cdots, A_n\}$ est obtenu ;
- un espace de stockage d'index $\mathcal{S}$.

Le problème PSIJB consiste à trouver une configuration d'index CI (ensemble d'index définis sur AS) tel que :

- Le coût d'exécution de la charge $\mathcal{Q}$ en présence de CI est optimisé.
- L'espace de stockage alloué pour les index de CI ne dépasse pas $\mathcal{S}$ ($Taille(CI) \leq \mathcal{S}$).

Formalisé ainsi, le PSIJB est un problème d'optimisation mono-objectif où il faut minimiser un seul objectif représentant le coût d'exécution de la charge $\mathcal{Q}$ en présence des index CI, noté Cost($\mathcal{Q}$,CI), sous une contrainte $Taille(CI) \leq \mathcal{S}$:

$$\begin{cases} \text{Minimiser Cost}(\mathcal{Q},CI) \\ \text{avec } Taille(CI) \leq \mathcal{S} \end{cases} \tag{1.1}$$

Le PSIJB est connu NP-Complet [42]. De ce fait, il n'existe pas d'algorithme exhaustif proposant une solution optimale en un temps fini. Plusieurs travaux de recherche proposent des solutions proches de la solution optimale en utilisant des heuristiques réduisant la complexité du problème.

1.2.2.2 Complexité

La sélection d'une configuration d'$\mathcal{IJB}$ est généralement une tâche difficile comparée à d'autres types d'index. Cela est dû à plusieurs considérations :

- un $\mathcal{IJB}$ est défini sur les attributs indexables dont le nombre peut être important (plusieurs dizaines ou centaines) dans le contexte d'entrepôt de données, vu le nombre de tables de dimension et le nombre d'attributs non clés de chaque table ;

- un $\mathcal{IJB}$ multiple peut être défini sur un ou plusieurs attributs issus de différentes tables de dimension, ce qui augmente le nombre d'$\mathcal{IJB}$ possible ;
- un attribut indexable peut figurer dans plusieurs $\mathcal{IJB}$ multiples car ces derniers peuvent être non disjoints ;
- la taille d'un $\mathcal{IJB}$ dépend de la cardinalité des attributs indexés. Un attribut de forte cardinalité rend l'index volumineux, donc difficile à stocker et à maintenir ;
- la taille d'un $\mathcal{IJB}$ dépend aussi du nombre d'instance de la tables de faits qui est de l'ordre de plusieurs millions d'instances dans le contexte d'entrepôt de données.

Le problème de sélection d'$\mathcal{IJB}$ est très complexe vu la taille très grande de l'espace de recherche d'une solution optimale. L'espace de recherche représente toutes les configurations possibles de tous les $\mathcal{IJB}$ qu'il est possibles de sélectionner. Nous présentons dans ce qui suit la taille de l'espace de recherche pour les index simples et multiple. Soit $\mathcal{AS} = \{A_1, A_2, \cdots, A_n\}$ un ensemble d'attributs indexables candidats pour la sélection d'une configuration d'$\mathcal{IJB}$.

1. *Taille de l'espace de recherche pour les $\mathcal{IJB}$ simples* : chaque attribut de AS peut donner lieu à l'implémentation d'un $\mathcal{IJB}$ simple. Sur l'ensemble $\mathcal{AS}$, il est possible de définir n $\mathcal{IJB}$ simples distincts. Une configuration d'index contient un sous ensemble des n index. Par conséquent, toutes les configurations d'index possibles constituent une partition de AS en un ensemble de sous groupes. Le nombre de configurations d'index total est donné par la formule suivante :

$$NbConfig_{IS} = 2^n - 1 \tag{1.2}$$

   Si le nombre d'attributs indexables est $n = 20$, le nombre de configurations d'IS possibles est $2^{20} - 1 = 1048575$.

2. *Taille de l'espace de recherche pour les $\mathcal{IJB}$ multiples* : Un $\mathcal{IJB}$ multiple est construit sur un sous ensemble des n attributs indexables. Ainsi, Le nombre total d'$\mathcal{IJB}$ multiples qu'on peut définir représente le nombre de partition de AS en un ensemble de sous-groupes où chaque sous-groupe contient au moins deux attributs. La formule pour calculer le nombre d'index multiples est donné comme suit :

$$Nb_{IM} = 2^n - n - 1 \tag{1.3}$$

   Le nombre de toutes les configurations d'index multiples possibles est donné par l'équation suivante :

$$NbConfig_{IM} = 2^{Nb_{IM}} - 1 = 2^{2^n - n - 1} - 1 \tag{1.4}$$

   Si le nombre d'attributs indexables est $n = 20$, le nombre total d'$\mathcal{IJB}$ est $2^{20} - 20 - 1 = 1048555$. De ce fait, le nombre configurations d'IM possibles est $2^{2^{20}-20-1} - 1 = 2^{1048555} - 1 = 10^{1048555 \times log_{10}(2)} - 1 \sim 10^{315646}$.

Vue cette complexité, et dans le souci d'optimiser simultanément la sélection et la jointure, plusieurs travaux ce sont intéressés à la proposition de démarche de sélection qui emploi des algorithmes moins coûteux.

1.2.2.3 Travaux de Aouiche et al.

Les auteurs dans [2] proposent une démarche de sélection des $\mathcal{IJB}$ simples et multiples. Vu la complexité de l'espace de recherche, les auteurs proposent d'utiliser une technique de DataMining à savoir l'algorithme CLOSE [81] pour la recherche des motifs fréquents fermés afin d'élaguer l'espace de recherche d'index. L'élagage vise à écarter les configurations d'index qui ne sont pas profitable

pour l'optimisation des requêtes. Ensuite, les auteurs emploient un algorithme glouton guidé par modèle de coût pour sélectionner la configuration finale d'$\mathcal{IJB}$. Dans ce qui suit, nous allons définir les motifs fréquents et les motifs fréquents fermés et présenter le principe de l'algorithme CLOSE ensuite nous exposerons le déroulement du processus de sélection proposé. **Motifs fréquents et motifs fréquents fermés :**
Soient $I = i_1, \cdots, i_m$ un ensemble de m items et $B = t_1, \cdots, t_n$ une base de données de n transactions. Chaque transaction est composée d'un sous-ensemble d'items $I' \subseteq I$. Un sous-ensemble I' de taille k est appelé un k-itemset. Une transaction t_i contient un motif I' si et seulement si $I' \subseteq t_i$. Le support d'un motif I' est la proportion de transactions de B qui contiennent I'. Le support est donné par la formule $support(I') = \frac{card(\{t \in B, I' \subseteq t\})}{card(B)}$. Afin de qualifier un motif d'être fréquent ou non, l'utilisateur fixe un seuil minimum pour le support appelé minsup. Si $Support(I') > minsup$ alors I' est un motif fréquent, sinon il est non fréquent. Vu que le nombre de motifs possibles croit exponentiellement (2^m), les auteurs emploient *les motifs fréquents fermés*. Un motif fréquent fermé est un ensemble maximal de motifs présent dans toutes les transactions auxquelles il appartient (ne représente pas un sous ensemble d'un autre motif).

Exemple 3 *Soient les transactions ou requêtes Q1, Q2, Q3, Q4, Q5 et Q6 utilisant cinq motifs qui représentent les attributs indexables, A1, A2, A3, A4 et A5. L'utilisation par les transaction et le support de chaque motif est donné par le tableau 1.1. Si on prend un support minimum minsup=3/6, les motifs A1 et A3 sont fréquents. Le motif A1, A3 est un motif fermé, car il représente l'ensemble maximal d'items communs aux transactions Q1, Q3, Q4 et Q6.*

TABLE 1.1 – Matrice d'usage et de supports des motifs

	A1	A2	A3	A4	A5
Q1	1	0	1	0	1
Q2	0	0	1	1	1
Q3	1	1	1	0	0
Q4	1	0	1	0	0
Q5	0	0	0	1	0
Q6	1	0	1	0	1
Support	4/6	1/6	5/6	2/6	3/6

Algorithme CLOSE : La génération des motifs fréquents est une tâche très complexe. En effet, si le nombre de motifs est grand, le nombre de motifs fréquents peut exploser. Pour réduire cette complexité, les auteurs emploient l'algorithme CLOSE pour générer les motifs fréquents fermés. CLOSE repose sur le principe d'extraction de générateurs d'ensemble de motifs fréquents fermés en calculant les fermetures sur les ensembles de générateurs. Soit un motif fréquent m_i. La fermeture sur l'ensemble $\{m_i\}$ vise a ajouter a cet ensemble les motifs qui sont présents aux même temps que le motif m_i dans un maximum de transaction (requêtes). L'ensemble $\{m_i\}$ est appelé générateur. L'algorithme commence à l'étape k=1 par l'initialisation des générateurs aux singletons 1-itemsets (chaque motif est un générateur). A l'étape k, l'algorithme calcul la fermetures et les supports sur les k-itemsets générateurs représentant un ensemble de motifs fréquents candidats. Il détermine ensuite les ensembles de motifs fréquents fermés selon le seuil minimal du support minsup. Ensuite, les générateurs k+1 sont préparés pour la prochaine itération et les générateur de rangs k sont supprimés. L'algorithme s'arrête lorsque l'ensemble de k-générateurs fréquents est vide.

Générateur	Fermeture	Support
{m1}	{m1, m3}	4/6
{m2}	{m1, m2, m3}	1/6
{m3}	{m3}	5/6
{m4}	{m4}	2/6
{m5}	{m3, m5}	3/6

Suppression Motifs infréquents

générateur	Fermeture	Support
{m1}	{m1, m3}	4/6
{m3}	{m3}	5/6
{m4}	{m4}	2/6
{m5}	{m3, m5}	3/6

Itération k=1

Générateur	Fermeture	Support
{m1, m4}	{m1, m4}	1/6
{m1, m5}	{m1, m5}	2/6
{m3, m4}	{m3, m4, m5}	1/6
{m4, m5}	{m4, m5}	1/6

Suppression Motifs infréquents

générateur	Fermeture	Support
{m1, m5}	{m1, m5}	2/6

Itération k=2

FIGURE 1.4 – Exemple de déroulement de l'algorithme CLOSE

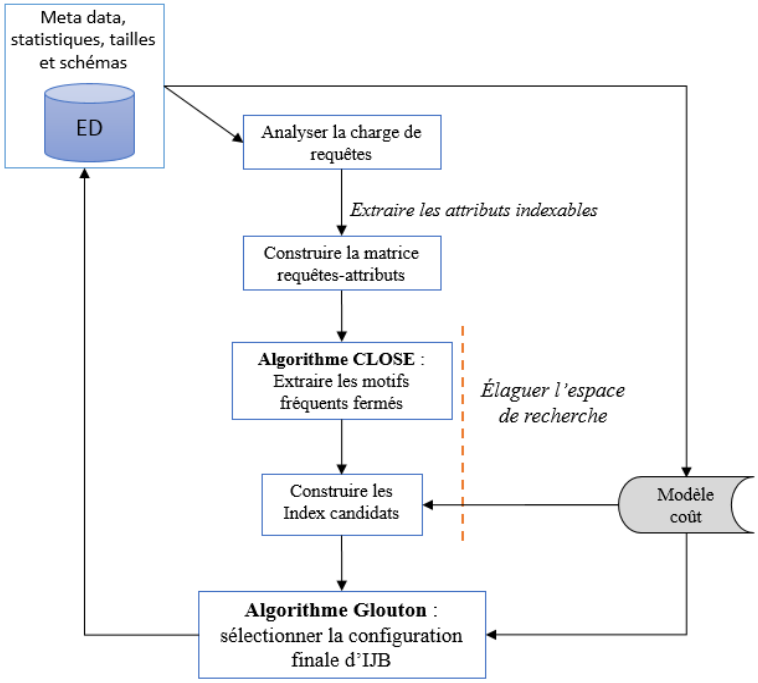


FIGURE 1.5 – Architecture pour la sélection des $\mathcal{IJB}$: Aouiche et al.

Exemple 4 *Considérons le contexte d'extraction décrit dans l'exemple 3. La figure 1.4 illustre les étapes d'exécution de l'algorithme CLOSE avec un support minimal de 2/6.*

Les auteurs proposent un processus de sélection des $\mathcal{IJB}$ simples et multiples qui est exposer dans les étapes suivantes (figure 1.5).

1. **Extraire la charge de requêtes :** La charge de requêtes est extraite à partir du journal de transactions qui est sauvegardé et maintenu automatiquement par le SGBD. Une charge donnée $\mathcal{Q}$ est représentative si elle est accessible grâce à la sauvegarde des opérations réalisées sur le système durant une période donnés. Cette période est fixée par l'administrateur et doit être suffisante pour pouvoir anticiper les requêtes futures.
2. **Extraire les attributs indexables :** Dans le cadre de sélection d'$\mathcal{IJB}$, les transactions re-

présentent les requêtes décisionnelles et les motifs représentent des $\mathcal{IJB}$ simples ou multiples potentiels. Une analyse de la charge de requêtes est effectuée afin d'extraire les attributs indexables figurant dans la clause WHERE. Ensuite, le contexte de recherche des motifs fréquents appelé aussi le contexte d'extraction est déterminé en construisant la matrice requête-attributs comme suit :

$$M(i,j) = \begin{cases} 1 & \text{si } A_j \text{ figure dans la clause WHERE de la requête } Q_i \\ 0 & \text{sinon} \end{cases}$$

3. **Élaguer l'espace de recherche :** Un motif fréquent fermé est un ensemble d'attributs contenu dans une même requête, qui apparait dans un maximum de requêtes et qui n'est pas contenu dans un autre motif. Rechercher les motifs fréquents fermés revient à trouver un ensemble initial d'index qui figurent dans un maximum de requêtes tout en réduisant le nombre de jointure en étoile à effectuer. Sélectionner les index finaux sur cet ensemble permet d'un coté d'élaguer l'espace de recherche des $\mathcal{IJB}$ et d'un autre coté assurer l'optimisation d'un grand nombre de requêtes. Dans un premier lieu, les motifs fréquents fermés sont extraits par exécution de l'algorithme CLOSE. Ensuite, les index candidats sont construits sur les motifs qui permettent de créer un $\mathcal{IJB}$ sur l'entrepôt. Chaque motif va contribuer à la construction d'un $\mathcal{IJB}$ s'il contient les attributs nécessaires : des clés étrangères de la tables des faits, des clés primaires des tables dimensions et les attributs non clés des tables dimensions sur lesquels sont définis les $\mathcal{IJB}$. Un motif qui vérifie ces condition permet de créer la requête SQL d'implémentation des $\mathcal{IJB}$ et qui est donnée comme suit :

```
CREATE BITMAP INDEX <IJB_A1_..._An>
ON TableFait(D1.Ai, D2.Aj, ..., Dd.An)
FROM TableFait, D1, D2, ..., Dd
WHERE TableFait.id1=D1.id1 AND ... AND TableFait.idd=Dd.idd
```

4. **Sélectionner la configuration finale d'index :** à partir de l'ensemble d'index candidats obtenu dans l'étape précédente, un algorithme glouton, guidé par une fonction objectif, est utilisé afin de sélectionner une configuration finale d'index. La fonction objectif est basée sur un modèle de coût qui permet de calculer la taille des index sélectionnés ainsi que le coût d'exécution des requêtes en présence de ces index. L'algorithme glouton démarre en initialisant la configuration finale d'index à l'index qui maximise la fonction objectif. A chaque itération, l'index le plus bénéfique est ajouter à la configuration finale. L'algorithme s'arrête si aucune amélioration de la fonction objectif n'est possible et/ou tous les index ont été sélectionnés et/ou l'espace de stockage disponible est saturé.

1.2.2.4 Travaux de Azefack et al.

Les travaux de Azefack et al. [3] sont une continuité des travaux de Aouiche et al. Ils proposent une approche de sélection d'$\mathcal{IJB}$ dite *dynamique* qui permet de mettre à jour la configuration d'index déjà implémentée sur l'$\mathcal{ED}$ lorsque la charge de requêtes évolue. L'évolution de la charge peut se manifester par l'ajout de nouvelles requêtes ou la suppression de requêtes existantes. Par conséquent, les auteurs proposent d'utiliser **l'algorithme GenMax** de DataMining souvent utilisé dans le contexte de bases de données volumineuse pour la recherche incrémentale des motifs fréquents fermés [53]. A la base de cela, ils définissent un processus de sélection d'$\mathcal{IJB}$ simples et multiples qui est toujours exécuté

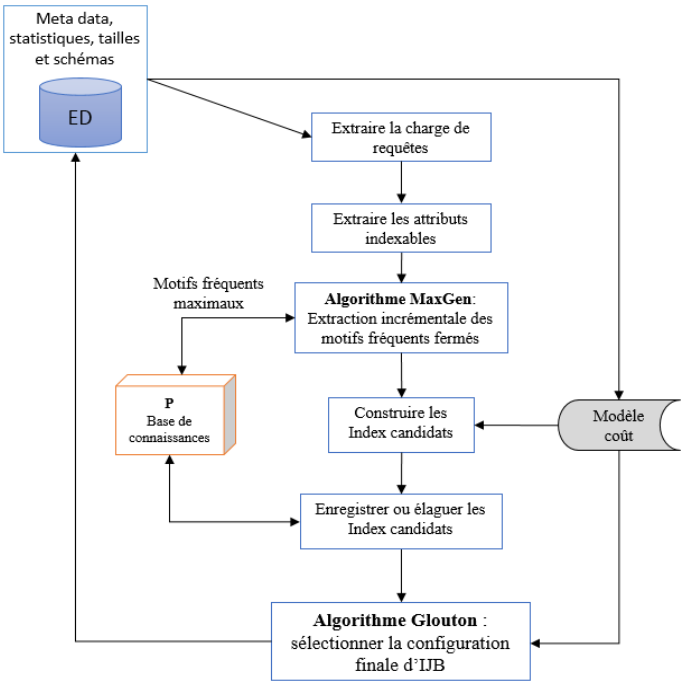


FIGURE 1.6 – Architecture pour la sélection incrémentale des $\mathcal{IJB}$: Azefack et al.

après une période de temps fixée par l'administrateur. Le processus de sélection (figure 1.6) et le déroulement de l'algorithme GenMax sont détaillés dans ce qui suit.

1. **Extraire les attributs indexables :** l'analyse de la charge de requêtes permet d'extraire les attributs indexables. Par la suite, la matrice requêtes-attributs est construite.
2. **Extraction incrémentale des motifs fréquents fermés :** Dans cette extension dynamique des travaux d'Aouiche et al., les auteurs remplaçant l'algorithme CLOSE par l'algorithme GenMax incrémentale pour rechercher les motifs fréquents fermés. Ils se placent dans un contexte où la charge de requêtes $\mathcal{Q}$ peut être très importante et évolue avec le temps. L'algorithme GenMax peut déterminer, dans un temps raisonnable, tous les motifs fréquents fermés dans les grandes bases de données et entrepôts de données. De plus, le nombre de motifs générés est réduit ce qui réduit considérablement l'espace de recherche des index. Le principe incrémental de l'algorithme GenMax est assuré par deux actions : (1) sauvegarder dans une base de connaissance **P** les résultats obtenus lors de précédentes itérations (motifs fréquents, le nombre de requêtes, etc) et (2) scanner la base de données à chaque itération afin de détecter si un changement est survenu sur la charge de requêtes. A chaque itération, la base **P** est mise à jour à partir de la base de données pour stocker le nombre de transactions (requêtes) à l'étape précédente, les motifs fréquents et non fréquents qui ont été créés durant de précédentes itérations, le nombre d'index candidats sélectionnés et la liste des motifs fréquents qui les construisent, etc. Ensuite, l'algorithme calcul la base de données des transactions mises à jour. Soit D une base de données de transactions. Soient d^+ l'ensemble des nouvelles transactions ajoutées et d^- les anciennes transactions supprimées. La base de données des transactions mises à jour est $\triangle = (D \cup d^+) - d^-$. Ainsi, l'extraction des motifs fréquents se fait sur $\triangle$ en utilisant **P** pour minimiser les accès à la base de données.
3. **Générer les index candidats :** L'exécution de l'algorithme GenMax sur la matrice requêtes-

attributs permet d'obtenir trois ensemble d'index :
- un ensemble I^+ de motifs fréquents émergents. Ce sont des motifs in-fréquents dans **P** mais qui sont devenus fréquents dans $\triangle$,
- un ensemble I^- de motifs fréquents qui déclinent. Ce sont des motifs fréquents dans **P** mais qui sont devenus in-fréquents dans $\triangle$,
- un ensemble I^0 de motifs fréquents maintenus. Ce sont des motifs fréquents dans **P** et $\triangle$.

La configuration des index candidats est alors calculée comme suit : $IC = (I \cup I^+) - I^-$. A noter que I^0 n'est pas utilisé pour calculer IC, mais est néanmoins enregistrée dans **P**.

4. **Sélectionner la configuration finale d'index :** un algorithme glouton guidé par modèle de coût permet de sélectionner sur les index candidats IC la configuration d'index finale qui, une fois implémentée sur l'entrepôt, permet d'optimiser la charge de requêtes sans dépasser l'espace de stockage alloué pour les index.
5. **Mise à jour de la configuration d'index :** Pour mettre à jour la configuration actuelle I, il faut créer tous les index émergents $i \in (I' - I)$; supprimer tous les index déclinants $i \in (I - I')$ et réinitialiser I à I'.

1.2.2.5 Travaux de Bellatreche et al.

Bellatreche et al. [16] proposent une approche de sélection des $\mathcal{IJB}$ simples basée sur la génération des motifs fréquents fermés en développent les travaux de Aouiche et al. [2]. En analysant les travaux de Aouiche et al., les auteurs remarquent que le coeur de la sélection proposée est la génération des motifs fréquents fermés qui emploie uniquement la fréquences d'accès des requêtes aux attributs. Par conséquent, les auteurs proposent d'utiliser d'autres paramètres importants comme la taille des tables et des attributs, la longueur des tuples, la taille de la page système et proposent une nouvelle version des algorithmes DataMining appelés DynaClose et DynaCharm qui prennent en compte ces paramètres. Ceci est justifié par le fait que les opérations les plus coûteuses sont les jointures entre la table des faits et les tables de dimension qui dépendent fortement de la taille des tables. En utilisant uniquement la fréquence d'accès comme critère de sélection ou élimination des index, l'algorithme peut éliminer des index sur des attributs non fréquents mais qui appartiennent à des tables de dimension volumineuses.

Exemple 5 *Considérant l'exemple 3. Le motif sélectionné est {A1, A3} et {A2} est écarté. Supposant que l'attribut A2 appartient à une table dimension de 200000 instance et que les attributs A1 et A3 appartiennent quand à eux à une table de 100 instance. Même s'il n'est pas fréquent, définir un index sur A2 apporterait une amélioration considérable au coût d'exécution total de la charge de requêtes.*

Le processus de sélection des index se déroule en trois étapes principales : élagage de l'espace de recherche des index, sélection des index candidats et sélection de la configuration finale d'index.

1. **Élaguer l'espace de recherche :** Les auteurs proposent une amélioration des algorithmes CLOSE et CHARM en proposant une nouvelle version appelée DynaClose et DynaCharm. Ces deux nouveaux algorithmes se basent sur une fonction Fitness comme alternative à l'utilisation du support pour choisir ou écarter un motif. Pour un motif fréquent m_i, une fonction $Fitness(m_i)$ est définie comme suit :

$$Fitness(m_i) = \frac{1}{n} \times (\sum_{j=1}^{n} \alpha_j \times sup_j) \tag{1.5}$$

où n, sup_j représentent respectivement le nombre d'attributs non clés A_j dans m_i et le support de A_j. α_j est un paramètre de pénalisation défini comme suit : $\alpha_j = \frac{|D_j|}{|F|}$, avec $|D_j|et|F|$ représentent respectivement le nombre de pages nécessaires pour stocker la table de dimension D_j et la table des faits F. Notons que la fonction fitness pénalise les motifs fréquents fermés qui sont définis sur des tables de dimension de petite taille. Afin de sélectionner les motifs fréquents, un seuil est calculé comme suit : $Minfit = \frac{minsup}{|F|} \times (\sum_{j=1}^{d} \frac{|D_j|}{|d|})$, avec d le nombre des tables dimensions.

2. **Sélectionner les index candidats** cette étape permet d'éliminer les motifs présentant une anomalie pouvant empêcher la création d'$\mathcal{IJB}$. La purification assure les deux conditions suivantes : (1) Chaque motif ayant tous les attributs non clés définis sur une même table dimension doit avoir uniquement deux attributs clés, une clé dimension et la clé étrangère de fait correspondante. (2) Chaque motif ayant les attributs non clés sur k tables de dimension doit avoir 2k clés : k clés dimension et k clés étrangères de la table de fait correspondantes.
3. **Sélectionner la configuration finale d'index :** les auteurs utilisent un algorithme glouton guidé par un modèle de coût pour la sélection finale d'$\mathcal{IJB}$ simples. A partir des motifs obtenus précédemment sont extraits tout les attributs non clé pour constituer un ensemble d'attributs indexables candidats. A la première itération, l'algorithme commence par sélectionner l'index défini sur l'attribut ayant la plus faible cardinalité. Ensuite, à chaque itération, un nouvel index est ajouté en respectant l'ordre croissant des cardinalité. L'algorithme s'arrête dans le cas où tout les index sont sélectionnés ou l'espace de stockage est saturé.

1.2.2.6 Travaux de Boukhalfa et al.

Les auteurs Boukhalfa et al. [9] ont proposés deux processus de sélection d'$\mathcal{IJB}$, le premier processus sélectionne une configuration d'$\mathcal{IJB}$ simples et le second processus sélectionne un ensemble d'$\mathcal{IJB}$ multiples. Un algorithme glouton est utilisé dans chacune des démarches pour la sélection d'une configuration finale d'index. Nous présentons dans ce qui suit les deux démarches de sélection.

Sélection d'une configuration d'index simples Les $\mathcal{IJB}$ mono-attribut sont intéressants pour des requêtes ayant une seule opération de sélection définie sur un attribut de faible cardinalité. Ce type d'index réduit la taille des $\mathcal{IJB}$ générés. Le déroulement du processus de sélection d'index, illustré dans la figure 1.7 est réalisé en trois étapes :

1. l'identification des attributs indexables : dans la première étape, l'ensemble des requêtes est analysé afin d'extraire les attributs indexables. Ces attributs sont les attributs sur lesquels un prédicat de sélection est défini dans la charge de requêtes. Les attributs indexables candidats sont choisis parmi les attributs indexables de faible et de moyenne cardinalité ;
2. l'initialisation de la configuration : l'algorithme commence par une configuration initiale composée d'un index mono-attribut défini sur l'attribut ayant la plus petite cardinalité noté IJB_{min} ;
3. l'enrichissement de la configuration actuelle par l'ajout de nouveaux index : la configuration initiale est améliorée itérativement par l'ajout d'index définis sur d'autres attributs non encore indexés. L'algorithme s'arrête lorsque l'une des deux conditions suivantes est satisfaite : aucune amélioration n'est possible et l'espace de stockage est consommé.

Sélection d'une configuration d'index multiples Rappelons qu'un $\mathcal{IJB}$ multiple est défini sur plusieurs attributs $\{A_1, A_2, \cdots, A_n\}$, où chaque attribut A_j peut appartenir à n'importe quelle

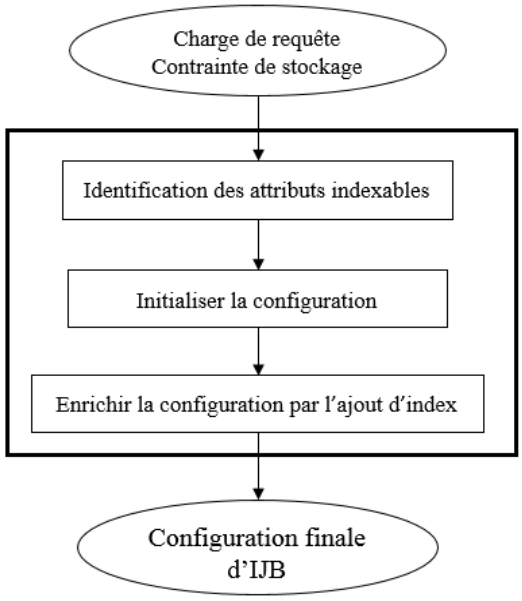


FIGURE 1.7 – approche de sélection des $\mathcal{IJB}$ simples : Boukhalfa et al.

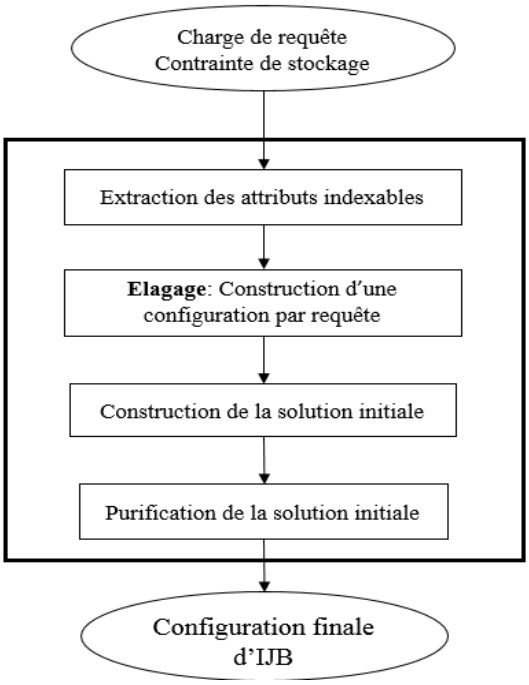


FIGURE 1.8 – approche de sélection des $\mathcal{IJB}$ multiples : Boukhalfa et al.

table de dimension. L'approche proposée (figure 1.8) repose sur quatre étapes : (1) l'identification des attributs indexables, (2) la construction d'une configuration par requête, (3) la construction d'une configuration initiale et (4) la construction d'une configuration finale. Les étapes (2) et (3) constituent la phase d'initialisation tandis que la quatrième étape concerne la phase de purification de la solution.
(1) **Identification des attributs indexables.** Cette étape se fait de la même manière que dans

l'approche de sélection des $\mathcal{IJB}$ mono-attribut.
(2) **Construction d'une configuration par requête.** Dans cette étape, la charge de requêtes $\mathcal{Q}$ est éclatée en m sous-charges. Chaque sous-charge est composée d'une seule requête. Pour chaque requête Q_i est sélectionné l'index qui couvre tous les attributs indexables utilisés par cette requête. Cela est motivé par le fait que cet index pré-calcule toutes les jointures de la requête. Par conséquent, peu ou pas de jointures sont effectuées pour exécuter la requête, ce qui permet de réduire considérablement son coût d'exécution.
(3) **Construction d'une configuration initiale.** Il s'agit de construire une configuration initiale à partir de l'ensemble des index générés lors de l'étape précédente. Cette configuration initiale est générée en effectuant l'union des index obtenus afin que chaque requête puisse être optimisée *via* l'$\mathcal{IJB}$ contenant tous les attributs indexables qu'elle utilise. Notons que le nombre d'index dans la configuration initiale est inférieur ou égal au nombre de requêtes de la charge, car certaines requêtes partagent les mêmes index. De plus, vu l'absence d'étape d'élagage d'attributs, cette configuration est définie sur tout l'ensemble d'attributs indexables.
(4) **Construction d'une configuration finale par purification.** La configuration initiale obtenue dans l'étape précédente ne prend pas en considération la contrainte d'espace qui peut être violée. Si la taille de la configuration initiale est inférieure à la capacité de stockage, alors elle est automatiquement choisie comme configuration finale. Dans le cas contraire, une purification itérative d'attributs est effectuée jusqu'à ce que la contrainte de stockage soit satisfaite. La purification est faite en éliminant certains attributs. Les auteurs ont considéré quatre stratégies d'éliminations : (a) élimination des attributs de forte cardinalité, (b) élimination des attributs appartenant aux tables moins volumineuses, (c) élimination des attributs les moins utilisés par les requêtes, (b) élimination des attributs apportant moins de réduction de coût, cela est motivé par le fait que les trois premières stratégies sont simples mais aucune métrique n'est utilisée. Les auteurs proposent donc d'utiliser un modèle de coût qui évalue le coût d'exécution des requêtes en présence des index et les attributs apportant moins de réduction du coût sont éliminés.

Algorithme d'affinité : Les auteurs proposent un algorithme basé sur les affinités des attributs pour la sélection des $\mathcal{IJB}$ multiples. Cet algorithme est une adaptation de l'algorithme de Navathe *et al.,* ([74][76]) proposé pour la fragmentation verticale. Le principe de base est de regrouper les attributs ayant une grande affinité souvent calculée à partir des fréquences d'accès des requêtes. Ainsi, si un maximum de requêtes utilisent simultanément un ensemble d'attributs de faible cardinalité, ces derniers peuvent former un $\mathcal{IJB}$ qui peut s'avérer bénéfique pour optimiser ces requêtes. Soient la charge de requêtes $\mathcal{Q} = \{Q_1, Q_2, \cdots, Q_m\}$ et l'ensemble des attributs indexables $\{A_1, \cdots, A_n\}$. L'algorithme manipule deux matrices :

- **La matrice d'usage des attributs** contient les requêtes en ligne et les attributs en colonne. Un élément MUA_{ij} $(1 \leq i \leq m; 1 \leq j \leq n)$ est mis à 1 si Q_i référence l'attribut A_j, il est mis à zéro dans le cas contraire. Pour $n = m = 4$, la matrice d'usage est donnée par le tableau 1.2.
- **La matrice d'affinité** est une matrice carrée contenant les attributs indexables en ligne et en colonne. La valeur de l'élément de la ligne i et de la colonne j reporte les valeurs d'affinité définies entre ces attributs. Cette affinité correspond à la somme des fréquences d'accès des requêtes accédant simultanément aux deux attributs.

En utilisant ces deux matrices, deux variantes de l'algorithme existent selon la manière de regrouper les attributs :

1. **Regrouper les attributs par partitionnement binaire :** Dans un premier lieu, la matrice

	A_1	A_2	A_3	A_4	Frq
Q_1	1	0	1	1	45
Q_2	0	1	1	0	5
Q_3	0	1	0	1	75
Q_4	0	0	1	1	3

TABLE 1.2 – Matrice d'usage des attributs

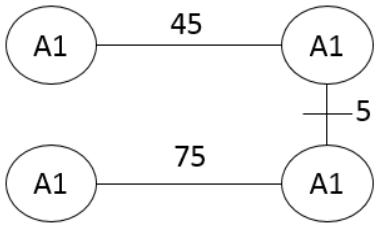


FIGURE 1.9 – Résultat d'exécution du partitionnement graphique

d'affinité ordonnée est calculée. C'est une matrice semi-bloc diagonale obtenue en appliquant l'algorithme BEA [72] qui effectue des permutations de lignes et de colonnes sur la matrice d'affinité. Dans un second lieu, un partitionnement binaire récursif de la matrice d'affinité ordonnée est effectué. Il s'agit de trouver une position sur la diagonale principale afin de définir deux sous-matrices sur cette diagonale. Les deux sous-matrices contiennent des valeurs d'affinités proches donc des attributs fréquemment accédés simultanément. Ce partitionnement peut être appliqué d'une manière récursive sur les sous-matrices obtenues. Chaque sous-matrice finale contient un ensemble d'attributs utilisés pour générer un index de jointure binaire. L'application du regroupement binaire sur la matrice d'usage d'attributs du tableau 1.2 donne le résultat représenté dans le tableau 1.3).

	A_1	A_2	A_3	A_4
A_1	45	45	0	0
A_2	45	53	5	3
A_3	0	5	80	75
A_4	0	3	75	78

TABLE 1.3 – Résultat du partitionnement binaire BEA

2. **Regrouper les attributs par partitionnement graphique :** Le regroupement des attributs est effectuer en générant un graphe complet et étiqueté, appelé *graphe d'affinité des attributs* [76]. Les noeuds de ce graphe représentent les attributs et une arête est la valeur d'affinité. Ce graphe est parcouru pour former des cycles, où chacun contient des attributs ayant une grande affinité. Chaque cycle devient un index de jointure binaire. La complexité de cet algorithme est $\theta(n^2)$, où n représente le nombre d'attributs indexables [76]. A noter que cet algorithme n'utilise pas le modèle de coût et n'est pas contraint par l'espace de stockage. A noter aussi que les deux variantes peuvent générer deux solutions différentes pour le même ensemble d'attributs indexables. L'application du regroupement graphique sur la matrice d'usage d'attributs du

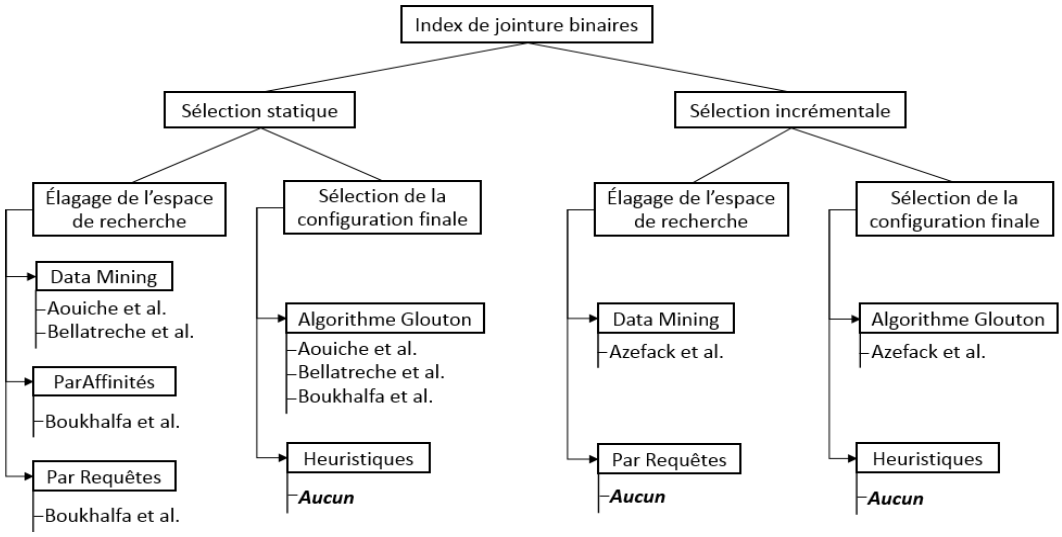


FIGURE 1.10 – Classification des travaux de sélection d'$\mathcal{IJB}$

tableau 1.2 donne le résultat représenté dans la figure 1.9.

1.2.3 Bilan et discussion

En analysant ces travaux de sélection des index de jointures binaires [9, 2, 3, 16], nous avons identifié deux phases importantes dans la sélection des $\mathcal{IJB}$; la sélection de la configuration initiale d'index et la sélection de la configuration finale d'index. *(1) La sélection de la configuration initiale d'index* consiste à déterminer l'espace de recherche des index potentiels à la sélection. Une analyse syntaxique des requêtes est d'abord effectuée afin d'extraire les attributs de sélection candidats à l'indexation, puis une étape d'*élagage de l'espace de recherche* est effectuée. Rappelons que la complexité du problème de sélection d'$\mathcal{IJB}$ est proportionnelle au nombre d'attributs candidats [9]. Dans le contexte d'$\mathcal{ED}$, le nombre d'attributs de sélection est très important (plusieurs dizaines ou centaines d'attributs) ce qui augmente la taille de l'espace de recherche du problème de sélection des index. De ce fait, une phase d'élagage a été proposée afin d'écarter les attributs ou index non pertinents pour la sélection. Cet élagage peut se faire manuellement en se basant sur l'expérience de l'administrateur. Cependant, cette solution n'est pas appropriée surtout si le nombre d'attributs est important. Afin de palier à cette limite, plusieurs travaux proposent *l'élagage automatique* en utilisant une *technique algorithmique* pour écarter les index ou attributs candidats non pertinents. Les techniques de fouille de données [16, 2, 3] et d'élimination d'attributs, en utilisant des critères comme la fréquence, la cardinalité des attributs, etc. [9], sont utilisées dans ce cas. Une fois l'élagage fait, un ensemble d'attributs candidats à l'indexation est établi. *(2) La sélection de la configuration finale d'index* : durant cette phase des algorithmes approximatifs (gloutons, recuit simulé, algorithmes génétiques, etc.) sont proposés afin de sélectionner une configuration finale réduisant le coût d'exécution de requêtes et satisfaisant la contrainte de stockage. Les travaux adoptent les algorithmes gloutons [16, 2, 3, 9].

Nous présentons dans la figure 1.10 une classification des travaux sur la sélection des $\mathcal{IJB}$. Après avoir analysé les principaux travaux sur l'indexation dans le contexte des $\mathcal{ED}$, nous avons identifié cinq principaux verrous :

1. Les plupart des travaux de sélection d'index se situent dans le cadre de sélection dite *statique*,

basée sur la connaissance préalable des requêtes. Or, dans le contexte d'entreposage de données on peut avoir une *évolution d'instance* causée par l'insertion de nouvelles instances dans les tables ou une *évolution de charge* représenté par le changement de fréquences d'exécution des requêtes, l'exécution de nouvelles requêtes, etc. D'un coté, l'évolution d'instance cause une augmentation de la taille des index implémentés sur l'entrepôt ce qui risque de causer une violation de la contrainte d'espace de stockage. D'un autre coté, si les requêtes changent, la configuration d'index implémenté devient obsolète. En conséquence, une optimisation continuelle de la charge de requêtes est primordiale afin d'adapter les index existants à l'évolution de la charge tout en respectant l'espace alloué aux index.

2. La sélection des $\mathcal{IJB}$ multiples représente un enjeu important pour la communauté d'entrepôt de données. D'une part, les $\mathcal{IJB}$ multiples pré-calculent les opérations complexes de jointures en étoiles. D'autre part, l'espace de recherche que les algorithmes doivent parcourir est très complexe. De plus, ils offrent un avantage non négligeable par rapport aux $\mathcal{IJB}$ simples, ils permettent de mieux respecter la contrainte d'espace de stockage. En effet, si deux $\mathcal{IJB}$ simples sont définis sur les attributs A_1 et A_2 respectivement, ils sont plus volumineux qu'un seul $\mathcal{IJB}$ multiple défini sur les deux attributs, puisque chaque index comporte une colonne supplémentaire constituant l'identifiant (*RowID* nécessitant 16 octets dans le SGBD Oracle, par exemple). Ainsi, le développement de techniques d'élagage pertinentes est recommandé pour réaliser la sélection d'$\mathcal{IJB}$ multiples.
3. Peu de travaux de sélection adoptent l'élagage automatique basé sur des requêtes, sachant que ces dernières sont au coeur du processus d'optimisation. En effet, la sélection d'un $\mathcal{IJB}$ qui couvre tout les attributs d'une requête permet de pré-calculer les opérations de jointures en étoiles complexes et couteuses en terme de temps d'exécution. De plus, vu la complexité des requêtes, un $\mathcal{IJB}$ qui couvre tout les attributs d'une requête donnée peut être très volumineux et peut donc être complètement écarté du processus de sélection ce qui compromet l'optimisation des requêtes les plus complexes.
4. Peu de *classes d'algorithmes* ont été explorées pour sélectionner des index, d'où la nécessité de développement d'autres types d'algorithmes.
5. Les travaux de sélection d'index ne font pas la distinction entre les index simples et multiples. Les espaces de recherche sont combinés en un seul espace très complexe ce qui représente un handicap majeur pour les index simples. En effet, l'espace de recherche des index multiples est considérablement plus complexe que celui des index simples. Une autre vision de sélection pourrait être plus bénéfique en considérant chaque type d'index comme une TO à part entière. On pourrait donc étudier la sélection jointe des index simples et des index multiples.

Nos propositions : Compte tenue de l'analyse des travaux d'indexation que nous avons fait, notre objectif est de définir une approche de sélection d'$\mathcal{IJB}$ qui couvre trois aspects :

1. Proposition d'une démarche de sélection incrémentale des $\mathcal{IJB}$ qui permet d'adapter la configuration d'index actuellement implémentée à l'évolution de l'entrepôt.
2. Proposition d'un nouvel algorithme de sélection basé sur les algorithmes génétiques.
3. Proposition d'un nouvel élagage basé sur les requêtes pour la sélection des index multiples.
4. Proposition d'une sélection jointe des index simples et des index multiples, statique puis incrémentale.

1.3 Sélection isolée d'un schéma de Fragmentation Horizontale

La fragmentation horizontale $\mathcal{FH}$ vise à restructurer les données afin de faciliter leur exploitation et interrogation par les requêtes décisionnelles. Elle permet de répartir les instances d'une table, d'un index ou d'une vue matérialisée en un ensemble de partitions disjointes appelés *fragments horizontaux* [84] selon un ensemble d'attributs appelés attributs de fragmentation figurant dans les tables. La fragmentation horizontale est considérée comme une précondition pour la conception de différents types de bases de données : bases de données relationnelles [75], orientées objets [64], relationnelles-objets [52], entrepôts de données [14], etc. Elle est également employée dans différents contextes : bases de données centralisées [80, 84], distribuées [80], parallèles [90], de type grilles (Grid Computing) [96] et plus récemment dans les bases de données de types Cloud [41]. Initialement, la fragmentation horizontale a été utilisée lors de la conception logique des bases de données relationnelles et orientées objet [64]. Actuellement, elle est considérée comme une technique d'optimisation utilisée au niveau physique.

Dans le contexte d'entrepôts de données, deux points importants caractérisent la fragmentation horizontale : (1) elle est considérée comme une technique d'optimisation non redondante car elle ne nécessite pas un espace de stockage supplémentaire [12] et (2) elle est appliquée lors de la création de l'entrepôt de données. Elle est employée dans plusieurs types d'entrepôts de données comme les entrepôts centralisés, distribués et parallèles. Deux types de fragmentation horizontale existent : (a) la fragmentation horizontale primaire ($\mathcal{FHP}$) et (b) la fragmentation horizontale dérivée ($\mathcal{FHD}$) [80]. Avant de définir ces deux types de fragmentation, nous présentons deux définitions utilisées dans les travaux liés à la fragmentation :

- *Attribut de fragmentation* : c'est un attribut de sélection utilisé pour fragmenter la table à laquelle il appartient en fragments disjoints.
- *Domaine d'un attribut* : noté Dom(A), il représente les valeurs distinctes qu'un attribut A peut prendre. Par exemple, le domaine d'un attribut Genre est Dom(Genre)={F,M}.

1.3.1 La $\mathcal{FH}$ et son évolution

Nous allons aborder dans ce qui suit les deux types de fragmentation horizontale. Ensuite, nous abordons la fragmentation horizontale dans les entrepôts de données.

1.3.1.1 Fragmentation primaire et dérivée

La fragmentation horizontale primaire permet de répartir les tuples d'une table suivent la conjonction de prédicats de sélection définis sur les attributs de la même table. Elle favorise les opérations de sélection portées sur les attributs de fragmentation, car pour exécuter une requête qui contient un prédicat de sélection sur un attribut de fragmentation, l'optimiseur ne charge que les fragments concerné par le prédicat de sélection [29]. Elle est réalisée grâce à l'application de l'opération de restriction sur les tuples de la table à fragmenter. Une opération de restriction appliquée sur une table suivant un critère définit un fragment comme suit :

$$Table_i = \sigma_{critere}(Table)$$

La fragmentation horizontale dérivée permet de fragmenter une relation suivant la fragmentation horizontale primaire d'une seconde relation, à condition de l'existence de relation mère-fille entre les deux tables. Elle est réalisée par l'exécution de semi-jointure entre les fragments propriétaires et la table membre à fragmenter [80]. La formule est donnée par :

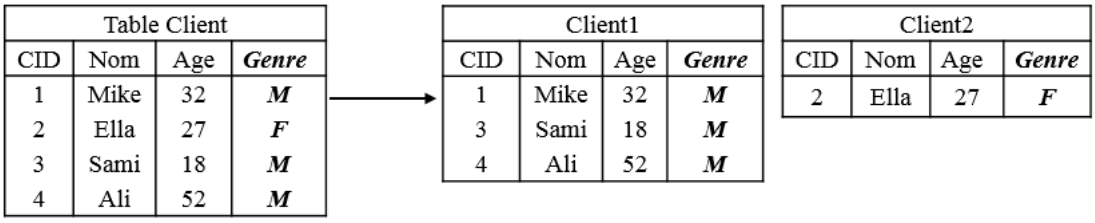


Table Client			
CID	Nom	Age	***Genre***
1	Mike	32	***M***
2	Ella	27	***F***
3	Sami	18	***M***
4	Ali	52	***M***

Client1			
CID	Nom	Age	***Genre***
1	Mike	32	***M***
3	Sami	18	***M***
4	Ali	52	***M***

Client2			
CID	Nom	Age	***Genre***
2	Ella	27	***F***

FIGURE 1.11 – Fragmentation horizontale primaire d'une table Clients sur l'attribut Genre

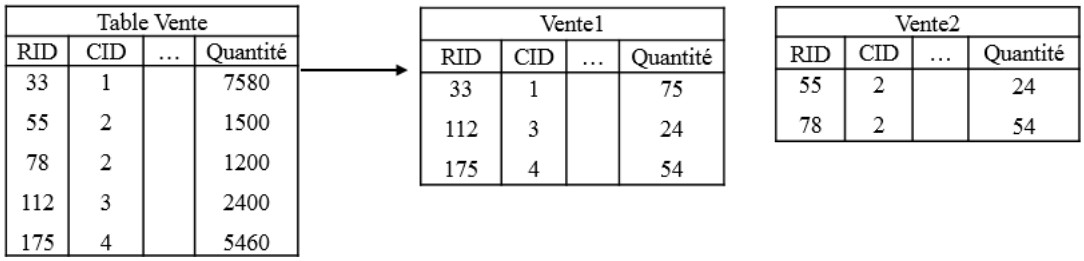


Table Vente			
RID	CID	…	Quantité
33	1		7580
55	2		1500
78	2		1200
112	3		2400
175	4		5460

Vente1			
RID	CID	…	Quantité
33	1		75
112	3		24
175	4		54

Vente2			
RID	CID	…	Quantité
55	2		24
78	2		54

FIGURE 1.12 – Fragmentation horizontale dérivée de la table Ventes suivant Clients

$$TableFille_j = TableFille \ltimes TableMere_i$$

Exemple 6 *Considérons l'entrepôt de données illustré sur la figure 1.1. La table Clients est fragmentée avec une $\mathcal{FH}$ primaire sur l'attribut Genre ce qui permet d'obtenir deux fragments : Client1 qui contient les clients féminins et Client2 contenant les clients masculins (figure 1.11). La description de l'opération de fragmentation par restriction est donnée comme suit :*

- *Client1 = $\sigma_{Genre=F}(Clients)$*
- *Client2 = $\sigma_{Genre=M}(Clients)$*

Vu l'existence d'un lien mère-fille entre les deux tables, la table Ventes est fragmentée par $\mathcal{FH}$ dérivée suivant la $\mathcal{FH}$ primaire de la table Clients (figure 1.12). Cette fragmentation est obtenue en appliquant les semi-jointures suivantes :

- *Vente1 = Ventes $\ltimes$ Client1*
- *Vente2 = Ventes $\ltimes$ Client2*

Par conséquent, le fragment Vente1 contient les ventes des clients féminins et Vente2 les ventes des clients masculins.

1.3.1.2 Fragmentation des entrepôts de données

Afin de fragmenter un entrepôt de données, plusieurs scénarios peuvent être considérés. Cependant, Les auteurs dans [14] montrent que le meilleur scénario est de fragmenter les tables dimensions suivants une fragmentation primaire sur leurs attributs respectifs, ensuite la table de faits est fragmentée par une $\mathcal{FH}$ dérivée suivant la $\mathcal{FH}$ primaire des tables dimensions. Cela permet de répartir l'entrepôt de données en plusieurs sous schémas en étoiles où chaque sous schéma contient un fragment de la table de faits et tous les fragments des tables dimensions qui ont permet de le définir. Ce scénario de $\mathcal{FH}$ permet d'améliorer les opérations de sélections sur les tables dimensions et les opérations de jointures entre la table de faits et les tables dimensions contenues dans les requêtes de jointures en étoile.

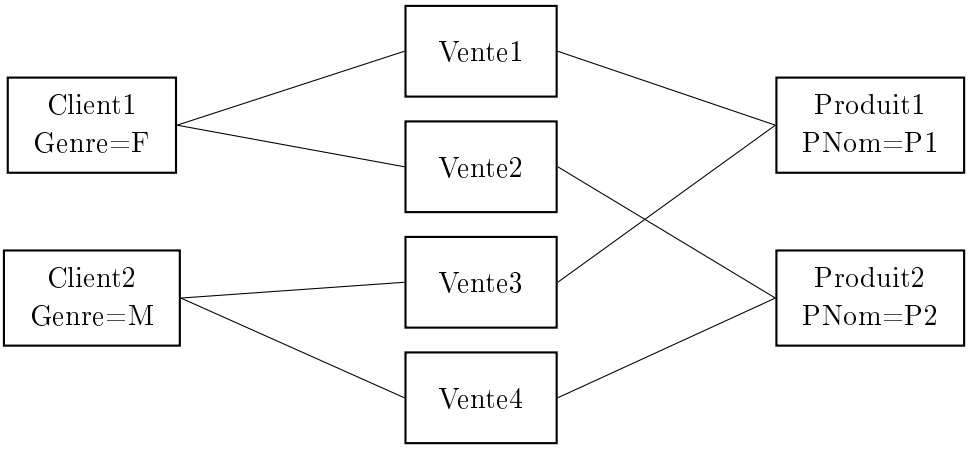


FIGURE 1.13 – Fragmentation de l'entrepôt de données

Exemple 7 *Considérons les tables décrites dans l'exemple 6. Supposons la table dimension Produits fragmentée selon la fragmentation primaire suivante :*

- *Produit1* $= \sigma_{PNom=P1}(Produits)$
- *Produit2* $= \sigma_{Pnom=P2}(Produits)$

La table Ventes est fragmentée en 4 fragments par $\mathcal{FH}$ dérivée suivant la $\mathcal{FH}$ primaire des tables Clients et Produits. La description de cette fragmentation est la suivante :

- *Vente1* $=$ *Ventes* $\ltimes$ *Client1* $\ltimes$ *Produit1*
- *Vente2* $=$ *Ventes* $\ltimes$ *Client1* $\ltimes$ *Produit2*
- *Vente3* $=$ *Ventes* $\ltimes$ *Client2* $\ltimes$ *Produit1*
- *Vente4* $=$ *Ventes* $\ltimes$ *Client2* $\ltimes$ *Produit2*

A l'issu de la fragmentation, l'entrepôt de données est répartis en 4 sous schémas en étoiles comme illustré sur la figure 1.13

1.3.1.3 La fragmentation dans l'industrie

La fragmentation horizontale a suscité beaucoup d'intérêt en industrie. Plusieurs SGBD commerciaux (Oracle, DB2, SQL Server, Sybase) proposent l'implémentation de cette technique d'optimisation. Nous présentons dans ce qui suit l'implémentation de la $\mathcal{FH}$ primaire et dérivée dans le SGBD Oracle.

La fragmentation horizontale primaire dans Oracle : Oracle propose de fragmenter une table, lors de sa création, suivant deux modes : un mode simple où la table est fragmentée suivant un seul de ses attributs et le mode composite où la table peut être partitionnée selon deux de ses attributs. Depuis la version Oracle 8i trois modes simples sont proposés : RANGE, LIST et HASH

1. **RANGE :** appelé aussi mode intervalle, il permet de fragmenter une table suivant la répartition en intervalles disjoints du domaine de l'attribut de fragmentation. Il est souvent utilisé pour des *attributs à forte cardinalité* dont le domaine présente une relation d'ordre entre ses valeurs comme les entiers, réels, dates, etc. comme par exemple l'Age d'un client. Supposant la fragmentation par le mode RANGE de la table Clients suivant l'attribut Age dont le découpage du domaine est Dom(Age) = {[0-20[, [20-40[, [40-80]}. La syntaxe Oracle qui permet d'exécuter cette fragmentation est donnée comme suit :

```
CREATE TABLE CLIENT (CID number(6), Nom varchar2(30), Genre char(1),
 Age number(3), Ville varchar2(50))
PARTITION BY RANGE(Age)
(PARTITION Client1 VALUES LESS THAN (20),
/ PARTITION Client2 VALUES LESS THAN (40),
 PARTITION Client3 VALUES LESS THAN (MAXVALUE));
```

La fragmentation suivant le mode RANGE est très bénéfique pour les requêtes de type intervalle. Par exemple, l'exécution de la requête suivante nécessite uniquement le chargement du fragment Client1.

```
SELECT *
FROM Clients
WHERE Age < 20;
```

2. **LIST :** ce mode permet de fragmenter une table suivant les valeurs distinctes de l'attribut de fragmentation. La syntaxe Oracle de fragmentation de la table Clients suivant l'attribut Genre est donnée par :

```
CREATE TABLE CLIENT (CID number(6), Nom varchar2(30), Genre char(1),
 Age number(3), Ville varchar2(50))
PARTITION BY LIST (Genre)
(PARTITION Client1 VALUES ('F'),
 PARTITION Client2 VALUES ('M'));
```

Ce mode de fragmentation accélère les requêtes avec sélection sur l'attribut Genre comme

```
SELECT *
FROM Clients
WHERE Genre = 'F';
```

3. **HASH :** pour ce mode, il faut donner la clé de la table et le nombre de fragments désirés. Oracle utilise une fonction de hachage interne afin de calculer la valeur de hachage sur chaque tuple de la table et l'attribue au fragment adéquat. Nous donnons dans ce qui suit la syntaxe Oracle pour fragmenter la table Clients par mode HASH :

```
CREATE TABLE CLIENT (CID number(6), Nom varchar2(30), Genre char(1),
 Age number(3), Ville varchar2(50))
PARTITION BY HASH (Id)
PARTITION 4;
```

4. **Mode Composite :** pour fragmenter une table suivant deux attributs, Oracle, depuis sa version 11g, propose la syntaxe SUBPARTITION qui permet de composer entre deux des modes simples présentés ci-dessus. Par exemple : RANGE-RANGE, RANGE-LIST, LIST-RANGE, LIST-HASH, etc. La fragmentation en mode composite RANGE-LIST de la table Clients sur les attributs Age et Genre se fait en deux niveaux. Dans le premier niveau on fragmente par mode RANGE selon l'attribut Age, Dans le second niveau, on fragmente la table partitionnée résultante suivant le mode LIST sur l'attribut Genre. La syntaxe Oracle est donnée comme suit :

```
CREATE TABLE CLIENT (CID number(6), Nom varchar2(30), Genre char(1),
 Age number(3), Ville varchar2(50))
PARTITION BY RANGE (Age)
   SUBPARTITION BY LIST (Genre)
   (SUBPARTITION C-Masculins VALUES ('M'),
    SUBPARTITION C-Féminins VALUES ('F'))
(PARTITION Client1 VALUES LESS THAN (20),
 PARTITION Client2 VALUES LESS THAN (40),
 PARTITION Client3 VALUES LESS THAN (MAXVALUE)));
```

La fragmentation horizontale dérivée dans Oracle : Elle est supportée depuis la version Oracle 11g par le mode REFERENCE. Elle permet de fragmenter une table fille suivant la fragmentation par un des mode RANGE, LIST ou HASH de la table mère en supposant l'existence d'une relation mère-fille entre les deux tables. Considérons la table dimension Clients fragmenté par le mode LIST sur l'attribut Genre et la table de faits Ventes. La syntaxe Oracle qui permet de fragmenter la table Ventes par le mode REFERENCE est donnée par la requête suivante :

```
CREATE TABLE VENTES (TID  number(6), CID  number(6), PID  number(6), PrixUnit Number(8,2)
 CONSTRAINT Client_fk FOREIGN KEY (CID)REFERENCES Client(CID))
PARTITION BY REFERENCE(Client_fk);
```

Gestion de partitions : Afin de gérer les différents fragments ou partitions générés après fragmentations des tables, plusieurs SGBD commerciaux fournissent une syntaxe DDL pour la gestion des partitions. Deux fonctions principales sont disponibles : MERGE PARTITION qui permet de fusionner deux partitions en une seule et SPLIT PARTITION, fonction duale à la première, permet de fragmenter une partition en deux partitions. La syntaxe SQL de ces fonctions est donnée ci-dessous.

```
ALTER TABLE <table_name>
MERGE PARTITIONS <first_partition>, <second_partition>
INTO PARTITION <partition_name> ;

ALTER TABLE <table_name>
SPLIT PARTITION <partition_name> AT <range_definition>
INTO (PARTITION <first_partition>, PARTITION <second_partition>)
```

1.3.2 Problème de sélection et travaux existants

Le problème de sélection d'un schéma de fragmentation est formalisé comme suit :
Étant donné :

- une table T à fragmenter
- une charge de requêtes $Q = \{Q_1, Q_2, \cdots, Q_m\}$

Le problème de sélection d'un schéma de fragmentation consiste à fragmenter la table T en n fragments horizontaux tel que le coût d'exécution de la charge de requête soit optimisé. Nous allons présenter les travaux qui proposent une démarche de fragmentation horizontale dans les entrepôts de données relationnels, centralisés et modélisés en étoile.

1.3.2.1 Travaux de Bellatreche et al.

L'auteur dans [6] propose une approche de fragmentation horizontale des entrepôts de données qui reprend l'algorithme de fragmentation basé sur la complétude et minimalité des prédicats définit dans [80]. Cet algorithme considère une table T et un ensemble de prédicats simples $P = \{p_1, p_2, \cdots, p_n\}$ définis sur les attributs de la table T et extraits à partir de la charge de requêtes à optimiser. L'algorithme produit en sortie des fragments horizontaux de T en exécutant les étapes suivantes :

1. A partir de l'ensemble P, générer un ensemble complet et minimal de prédicats en exécutant l'algorithme COM-MIN présente dans [79]. La minimalité permet d'avoir uniquement des prédicats pertinents, où un prédicat pertinent permet de partitionner une table en deux fragments distincts. La complétude assure qu'il n'y ait aucune perte de données lors du processus de fragmentation tel que chaque tuple de la table peut être identifié par une conjonction de prédicats.

2. Générer l'ensemble des minterms $\mathcal{M} = \{m | m = \wedge_{(1 \leq k \leq n)} p_k*\}$, où p_k* est un prédicat p_k or $\neg p_k$.
3. Éliminer les minterms redundants et contradictoires dans $\mathcal{M}$.
4. Générer les fragments : chaque minterm $m_i \in \mathcal{M}$ permet de décrire un fragment définit par $\sigma_{m_i}(T)$.

Cette approche est simple mais possède une grande complexité. Pour n simples prédicats, l'algorithme COM-MIN génère 2^n minterms.

Exemple 8 *Soit $\{P_1, P_2, P_3, P_4\}$ quatre prédicats définis sur la table Clients et donnés comme suit : $P_1 : A < 40$, $P_2 : A > 20$, $P_3 : Ville = Alger$ et $P_4 : Ville = Tizi$. Sur ces prédicats, on peut définir 16 (2^4) minterms décrit par la table 1.4. Après élimination des minterms redondants et contradictoires,*

Minterms (1 à 8)	Minterms (9 à 16)
$P_1 \wedge P_2 \wedge P_3 \wedge P_4$	$\neg P_1 \wedge P_2 \wedge P_3 \wedge P_4$
$P_1 \wedge P_2 \wedge P_3 \wedge \neg P_4$	$\neg P_1 \wedge P_2 \wedge P_3 \wedge \neg P_4$
$P_1 \wedge P_2 \wedge \neg P_3 \wedge P_4$	$\neg P_1 \wedge P_2 \wedge \neg P_3 \wedge P_4$
$P_1 \wedge P_2 \wedge \neg P_3 \wedge \neg P_4$	$\neg P_1 \wedge P_2 \wedge \neg P_3 \wedge \neg P_4$
$P_1 \wedge \neg P_2 \wedge P_3 \wedge P_4$	$\neg P_1 \wedge \neg P_2 \wedge P_3 \wedge P_4$
$P_1 \wedge \neg P_2 \wedge P_3 \wedge \neg P_4$	$\neg P_1 \wedge \neg P_2 \wedge P_3 \wedge \neg P_4$
$P_1 \wedge \neg P_2 \wedge \neg P_3 \wedge P_4$	$\neg P_1 \wedge \neg P_2 \wedge \neg P_3 \wedge P_4$
$P_1 \wedge \neg P_2 \wedge \neg P_3 \wedge \neg P_4$	$\neg P_1 \wedge \neg P_2 \wedge \neg P_3 \wedge \neg P_4$

TABLE 1.4 – Ensembles de minterms

six fragments sont construits comme présenté dans la table 8.

$F_1 : 20 < A < 40$	$\wedge$	*Ville='Alger'*
$F_2 : 20 < A < 40$	$\wedge$	*Ville='Tizi'*
$F_3 : A \leq 20$	$\wedge$	*Ville='Alger'*
$F_4 : A \leq 20$	$\wedge$	*Ville='Tizi'*
$F_5 : A \geq 40$	$\wedge$	*Ville='Alger'*
$F_6 : A \geq 40$	$\wedge$	*Ville='Tizi'*

TABLE 1.5 – Les fragments finaux de la table Clients

Sur la base du principe de complétude et minimalité des prédicats, l'auteur dans [5] propose l'approche de fragmentation suivantes :

1. Extraire à partir de la charge de requêtes les prédicats de sélection.
2. Répartir l'ensemble des prédicats en sous-ensembles tel que chaque ensemble est défini sur les attributs appartenant à la même table.
3. Éliminer les tables dimensions n'ayant pas de prédicats de sélection.
4. Appliquer l'algorithme de fragmentation basé sur la complétude et minimalité des prédicats définit dans [80]. Le résultat de cette exécution est une fragmentation primaire de chacune des tables dimensions considérée.
5. Fragmenter la table de faits par une fragmentation dérivée selon les fragments des tables dimensions.

Le problème de cette démarche est le nombre important de fragments de la table de faits. En effet, supposant un entrepôts de données contenant d tables de dimension $\mathcal{D} = \{D_1, D_2, \cdots, D_d\}$ et une table des faits $\mathcal{F}$. Supposant k tables dimension fragmentée chacune en m_i fragments, tel que $k \leq d$ et $m_i > 0$. Le nombre total de fragments de la table de faits N est donné par la formule suivante [6] :

$$N = \prod_{i=1}^{k} m_i \tag{1.6}$$

Exemple 9 *Considérons l'entrepôt de données illustré sur la figure 1.1. Les tables dimensions sont partitionnées comme suit :*
Clients partitionnée en 2 sur Genre et en 10 sur Ville, soit un total de 20 fragments.
Temps est fragmentée en 12 fragments sur Mois.
Produits est fragmentée en 6 selon Catégorie.
Le nombre total de fragments de la table de faits est 1440 fragments ($20 \times 12 \times 6$*) donc 1440 sous-schémas en étoile. La gestion d'un tel nombre de fragments par l'administrateur devient difficile.*

Ainsi, l'auteur dans [6] propose de réduire la complexité de la fragmentation en limitant le nombre de table de dimensions à fragmenter. Il formalise ainsi un nouveau problème de sélection des tables de dimension à fragmenter comme suit : Soit :

- une charge de requêtes $Q = \{Q_1, Q_2, \cdots, Q_m\}$,
- d tables de dimension à fragmenter $\mathcal{D} = \{D_1, D_2, \cdots, D_d\}$,
- N le nombre maximum de fragments faits générés.

Le problème consiste à choisir un ensemble de tables dimension tel que une fois ces tables dimensions fragmentées par une $\mathcal{FH}$ primaire et la table des faits fragmentée par une $\mathcal{FH}$ dérivée, le nombre de fragments faits ne dépasse pas N et le coût de la charge Q est optimisé.

Pour résoudre ce problème, l'auteur propose un algorithme glouton. L'algorithme commence par sélectionner une table dimension D et la fragmente par une $\mathcal{FHP}$. Il calcule le nombre de fragments de la table de faits et le coût d'exécution des requêtes. Si le nombre de fragments ne dépasse pas N et le coût est réduit alors la table est sélectionnée, elle est rejetée sinon. Afin d'améliorer l'algorithme et d'éviter la sélection aléatoire de la première table dimension, l'auteur propose trois méthodes de sélection : (1) la table dimension la plus fréquemment utilisée par les requêtes, (2) cette ayant le plus petit nombre de fragments ou (3) celle ayant des attributs de faible cardinalité (attribut Genre). La figure 1.14 illustre l'exécution de l'algorithme glouton.

1.3.2.2 Travaux de Boukhalfa et al.

Les auteurs dans [12, 13, 14, 26] proposent une démarche de fragmentation horizontale des entrepôts de données. Afin de répondre au problème d'explosion du nombre de fragments de la table de faits après fragmentation, ils formalisent le problème comme suit :
Etant donné :

- une charge de requêtes $Q = \{Q_1, Q_2, \cdots, Q_m\}$,
- F une table de fait et $\mathcal{D} = \{D_1, D_2, \cdots, D_d\}$ d tables de dimension,
- W le nombre de fragments maximum de la table de fait imposé par l'administrateur

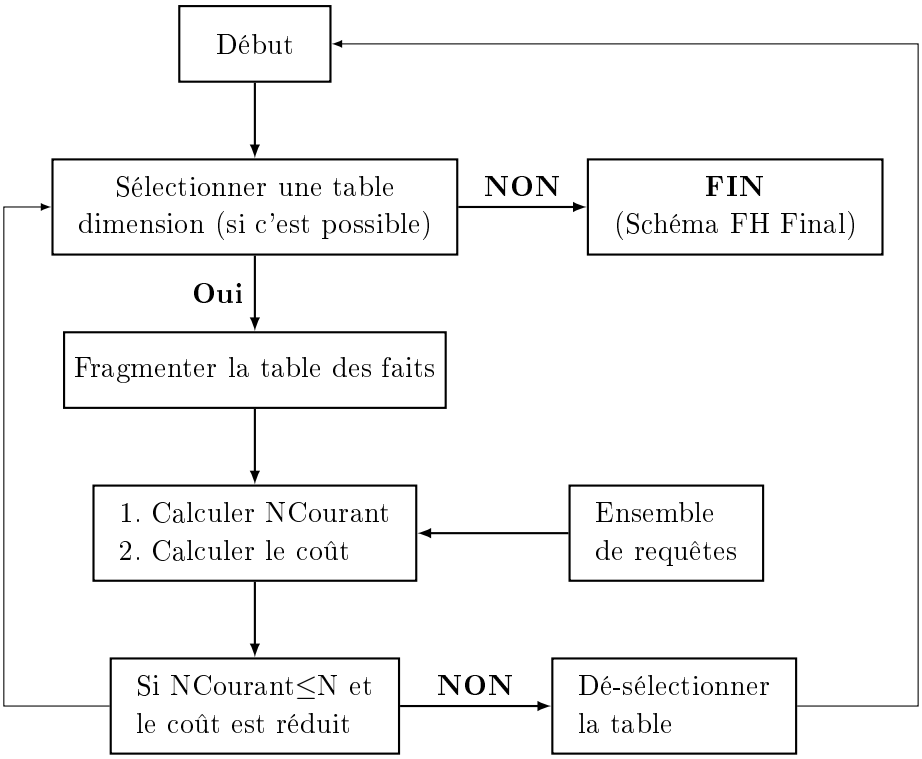


FIGURE 1.14 – Algorithme glouton pour la fragmentation : Bellatreche et al.

Le problème consiste à sélectionner un schéma de fragmentation primaire SF des tables dimensions tel que :

- le coût d'exécution des requêtes soit minimal,
- après fragments dérivée de F suivant SF, le nombre de fragments de F ne dépasse pas W.

Les auteurs dans [14] prouve que ce problème formalisé ainsi est un problème NP-Complet. Ils proposent ainsi une démarche de sélection d'un schéma de fragmentation optimal des tables dimensions qui se base sur des méta-heuristique (Hill Climbing, Recuit Simulé, Algorithmes Génétiques). Cette démarche englobe les étapes suivantes :

Préparer la fragmentation à partir de la charge de requêtes, un ensemble des prédicats de sélection est extrait. Les prédicats de sélection permettent de définir les tables dimensions candidates à la fragmentation. Ce sont les tables contenant un attribut figurant dans au moins prédicat. Ensuite, l'algorithme COM-MIN est exécuté afin de généré l'ensemble de prédicats complet et minimal.

Découper les domaines des attributs A l'issue de l'étape précédente, à chaque table de dimension D_i est attribué un ensemble EP_i de prédicats complet et minimal qui permet de définir les attributs de fragmentation de la table D_i. Pour chaque attribut A_j, il est possible de définir un découpage de son domaine en sous-domaines à partir de l'ensemble EP_i. Par exemple, le prédicat P : Age < 18 permet de découper le domaine de l'attribut Age en deux sous-domaines, [0,18[et [18,80]. Par conséquent, le découpage des domaines en sous-domaines des attributs de fragmentation de D_i, à partir de EP_i, permet de définir une fragmentation horizontale primaire de D_i. Le tableau 1.6 montre un exemple de découpage des domaines des attributs Age, PNom et Ville en sous-domaine.

Age	[0,18[	[18,80]		
PNom	P1	P2	P3	
Ville	Alger	Oran	Blida	Kala

TABLE 1.6 – Découpage des domaines d'attributs en sous-domaines

Coder le schéma de fragmentation Le découpage en sous-domaines des domaines d'attributs de fragmentation permet de définir un schéma de fragmentation primaire des tables dimensions. Il est possible de définir d'autres schémas de fragmentation en regroupant des sous-domaines d'un même attribut dans un seul ensemble. Afin de coder tous les schémas possibles, on représente sous forme d'un tableau de vecteurs le schéma de fragmentation, où chaque vecteur caractérise un attribut et où les cellules du vecteur caractérisent un sous-domaine. Chaque cellule possède un numéro tel que les sous-domaines qui possèdent le même numéro sont regroupés dans le même sous-ensemble. Le tableau 1.7 illustre un exemple de codage. Ce codage signifie que pour l'attribut Ville, les valeurs Alger et Oran sont regroupées dans un sous-ensemble et Blida et Kala dans un autre sous-ensemble. Les valeurs P1, P2 et P3 de l'attribut PNom sont toutes regroupées dans un seul sous-ensemble.

Age	1	2		
PNom	1	1	1	
Ville	1	1	2	2

TABLE 1.7 – Codage d'un schéma de fragmentation

Fragmenter les tables à partir d'un codage Le codage du schéma de fragmentation permet de définir des prédicats de sélection sur les attributs de fragmentation. Pour chaque table dimension, la conjonction des prédicats de sélection définie sur ses attributs permet de donner une fragmentation sur cette table. Un fragment nommé ELSE, défini par la conjonction des négations de tous les prédicats, est ajouté à chaque table dimension, afin d'assurer la complétude. Le codage, présenté dans le tableau 1.7 montre que toutes les valeurs de l'attribut PNom sont regroupées dans un même ensemble. Par conséquent, aucune fragmentation n'est définie sur l'attribut PNom et donc sur la table Produits. Pour la table Clients, sa fragmentation est la suivante :

- Client1 = $\sigma_{(0<Age<18)\wedge(Ville=Alger\vee Ville=Oran)}$(Clients)
- Client2 = $\sigma_{(0<Age<18)\wedge(Ville=Blida\vee Ville=Kala)}$(Clients)
- Client3 = $\sigma_{(18\leq Age<80)\wedge(Ville=Alger\vee Ville=Oran)}$(Clients)
- Client4 = $\sigma_{(18\leq Age<80)\wedge(Ville=Blida\vee Ville=Kala)}$(Clients)

Suivant la fragmentation primaire obtenue des tables dimensions, la fragmentation dérivée de la table de faits est effectuée.

Sélectionner un schéma de fragmentation final Les auteurs proposent dans un premier lieu d'employer l'algorithme d'affinité afin de définir un schéma de fragmentation horizontale primaire des tables dimensions. Cet algorithme a été défini dans [76] pour regrouper les attributs afin de réaliser la fragmentation verticale. L'idée part du principe que les tuples d'une table accédés simultanément doivent appartenir au même fragment horizontal. Vu qu'un fragment horizontal est défini à partir du découpage des domaines des attributs, il faut regrouper les sous-domaines fréquemment accédés ensemble dans le même ensemble. Le regroupement des sous-domaines s'effectuent à la base de leur affinité. L'affinité entre deux sous-domaines est donnée par la somme des fréquences des requêtes utilisant simultanément à ces deux sous-domaines. La sélection d'un schéma de fragmentation basée sur l'algorithme d'affinité est constituée des étapes suivantes :

1. Extraire les sous-domaines utilisés par les requêtes et figurant dans les prédicats de sélection (clause WHERE). Cela donne un ensemble de sous-domaines $\{SD_{1,1}, \cdots, SD_{1,n_1}, \cdots, SD_{n,1}, \cdots, SD_{n,n_n}\}$ de n attributs de fragmentation, où $\{SD_{k,1}, \cdots, SD_{k,n_k}\}$ est l'ensemble des sous-domaines de A_k.
2. Construire la matrice d'usage MUS : elle décrit l'usage des sous-domaines par les requêtes. Pour chaque attribut A_k, on construit une matrice $\text{MUS}^k(m \times n_k)$ où m est le nombre de requêtes et n_k le nombre de sous-domaines de l'attribut A_k. Un élément MUS^k_{ij} $(1 \leq i \leq m; 1 \leq j \leq n_k)$ est mis à 1 si Q_i référence le sous-domaine $SD_{k,j}$, il est mis à zéro sinon. Soit l'attribut Ville de la table Clients avec le découpage en 5 sous-domaines SD_1={Alger}, SD_2={Oran}, SD_3={Blida}, SD_4={Kala} et SD_5={Tizi}. Soit 8 requêtes utilisant des prédicats de sélection définis sur l'attribut Ville. L'utilisation des sous-domaines par les 8 requêtes permet de définir la matrice d'usage décrite dans le tableau 1.8.

	SD_1	SD_2	SD_3	SD_4	SD_5	Fréq
Q_1	1	0	0	0	1	10
Q_2	0	1	1	0	0	50
Q_3	0	0	0	1	0	25
Q_4	0	0	0	0	1	5
Q_5	0	1	1	0	1	45
Q_6	1	0	0	0	1	15
Q_7	1	0	0	0	0	25
Q_8	0	0	0	1	0	15

TABLE 1.8 – Matrice d'usage des sous-domaines de l'attribut Ville

3. Construire la matrice d'affinité des sous-domaines : c'est une matrice carrée $\text{MAS}^k(n_k \times n_k)$ où les lignes et les colonnes représentent les sous-domaines. Chaque élément MAS^k_{ij} représente la somme des fréquences des requêtes accédant simultanément aux deux sous-domaines $SD_{k,i}$ et $SD_{k,j}$. Le tableau 1.9 représente la MAS des sous-domaines de l'attribut Ville.

	SD_1	SD_2	SD_3	SD_4	SD_5
SD_1	50	0	0	0	25
SD_2	0	95	95	0	45
SD_3	0	95	95	0	45
SD_4	0	0	0	40	0
SD_5	25	45	45	0	75

TABLE 1.9 – Matrice d'affinité des sous-domaines de l'attribut Ville

4. Grouper graphiquement les sous-domaines en partitions : l'algorithme de regroupement graphique proposé dans [76] est exécuté sur chaque matrice d'affinité de chaque attribut. L'algorithme génère un graphe complet et étiqueté, appelé graphe d'affinité des sous-domaines, où les nœuds représentent les sous-domaines et une arête la valeur d'affinité. L'algorithme forme des cycles où chacun permet de regrouper les sous-domaines en un ensemble. La figure 1.15 illustre l'exécution du regroupement graphique pour l'attribut Ville. Il permet de définir la répartition

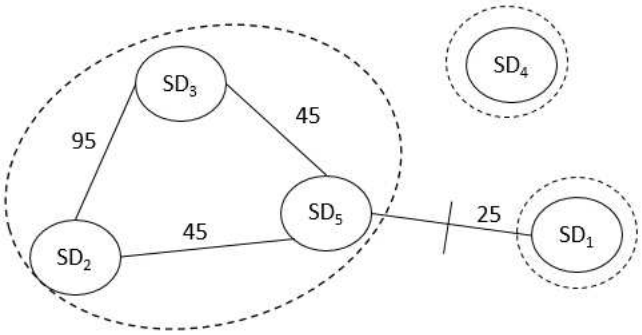


FIGURE 1.15 – Regroupement graphique des sous-domaines de l'attribut Ville

des sous-domaines en trois ensembles comme suit : Ens_1={Alger}, Ens_2={Oran, Blida, Tizi} et Ens_3={Kala}.

5. Générer le schéma de fragmentation : chaque ensemble de sous-domaine se voit attribuer le même numéro pour le codage du schéma de fragmentation. Pour l'attribut Ville, on obtient le codage {1,2,2,3,2}. A l'issu de cette étape, on obtient un tableau numérique représentant le codage du schéma de fragmentation des tables dimensions.

L'avantage de l'algorithme d'affinité est sa simplicité et sa complexité réduite pour la sélection d'un schéma de fragmentation. Cependant, ses inconvénients majeurs et qu'il utilise uniquement la fréquence d'accès comme critère de groupement, ne permet pas de contrôler le nombre de fragments généré et ne donnent aucune garantie sur la qualité de la solution obtenue. Pour cela, les auteurs dans [10, 13] proposent d'utiliser la solution obtenue par l'algorithme d'affinité comme solution initiale pour trois algorithmes : l'algorithme de Hill Climbing, le Recuit Simulé et les Algorithmes génétiques. Le processus de sélection proposé est illustré sur la figure 1.16.

Appliquer une méta-heuristique : La solution initiale obtenue par l'algorithme d'affinité est améliorée grâce à l'application de l'algorithme Hill Climbing. Cet algorithme utilise deux mouvements sur le schéma de fragmentation : Split pour fragmenter et Merge pour fusionner. L'opération Split prend deux sous-domaines ayant le même numéro et attribue à l'un d'eux un numéro différent, ce qui fait que ces deux sous-domaines ne définissent plus le même fragment mais deux fragments différents. Contrairement à cela, l'opération Merge prend deux sous-domaines ayant deux numéros différents et leur attribue le même numéro. En résumé, Merge et Split permettent respectivement de fusionner ou d'éclater des partitions. L'algorithme s'arrête lorsqu'aucune amélioration de la solution n'est possible. Même s'il représente un algorithme moins coûteux en termes d'exécution, le Hill Climbing est confronté au problème d'optimums locaux. Ainsi, les auteurs proposent d'employer l'algorithme du Recuit simulé. Il permet d'améliorer une solution initiale en effectuant un certain nombre de transformations. Pour pallier au problème de l'optimum local, des solutions non avantageuses sont acceptées avec une certaine probabilité qui diminue au cours de l'exécution de l'algorithme. Les deux méta-heuristiques présentées exploitent une seule solution à la fois, ainsi les auteurs proposent un algorithme génétique pour permettre la considération de plusieurs schémas de fragmentations au même temps sous forme d'une population. A chaque itération, l'algorithme génétique considère une population de chromosomes, où chaque chromosome représente le codage d'un schéma de fragmentation possible. Cette population est améliorée en appliquant des opérations génétiques à savoir la sélection, croisement et mutation. A la fin de son exécution, l'algorithme génétique permet de sélectionner un schéma de fragmentation optimal qui minimise le coût de la charge de requêtes et dont le nombre de

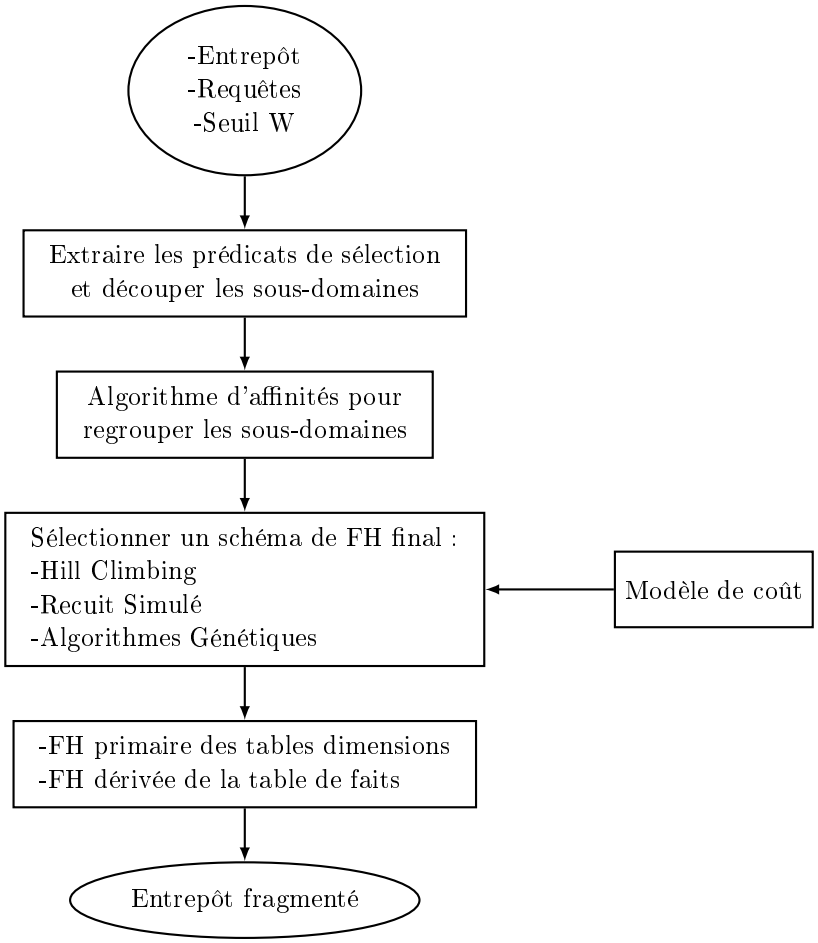


FIGURE 1.16 – Processus de fragmentation : Boukhalfa et al.

fragments de la table des faits ne dépasse pas la contrainte W. Pour les trois algorithmes, un modèle de coût mathématique est employé afin d'évaluer le bénéfice apporté par chaque solution exploitée. Il permet de calculer le coût d'exécution de la charge de requêtes sur un entrepôt fragmenté suivant le schéma de fragmentation décrit par cette solution. Le modèle de coût est décrit dans [14].

1.3.2.3 Travaux de Mahboubi et al.

Dans [56], les auteurs s'intéressent à la fragmentation horizontale des entrepôts de données XML. Pour ce faire, il se base sur l'adaptation des techniques de fragmentation existantes dans les entrepôts de données relationnels sur les entrepôts XML comme les approches basées sur l'algorithme d'affinité. Ensuite, ils présentent une nouvelle méthode de fragmentation qui utilise une technique de fouille de données (Datamining) à savoir la classification de données par k-means. Les deux algorithmes (affinité et fouille de données) visent à répartir en sous-ensembles un ensemble de prédicats de sélection extraits à partir de la charge de requêtes.

1. **Fragmentation basée sur les affinités :** cette démarche est utilisée pour fragmenter un seul graphe dimension. A partir des requêtes, les prédicats de sélections sont extraits. L'algorithme d'affinités est exécuté en construisant la matrice d'usage des prédicats par les requêtes et la

matrice d'affinités des prédicats de sélection. Ensuite, Le regroupement graphique est exécuté ce qui permet d'obtenir des cycles, où chaque cycle est un ensemble de prédicats. A l'issue de l'exécution de l'algorithme d'affinités, une étape de composition des termes de prédicats est effectuée. Tous les cycles obtenus sont évalués pour déterminer les attributs distincts communs à tous les cycles. Ce sont les attributs de fragmentation. Ensuite, on construit une matrice Att($c \times n$) décrivant l'usage de chaque attribut de fragmentation par les prédicats d'un cycle tel que c est le nombre de cycles et n le nombre d'attributs de fragmentation. À partir de Att, les termes de prédicats sont construits. Un terme de prédicat t est constitué d'un ensemble de prédicats qui couvre tous les attributs de fragmentation. Il représente une entrée dans la matrice Att. Si un attribut est manquant dans un terme, on y ajoute un prédicat contenant un attribut. Chaque terme défini alors un fragment horizontale. Un fragment nommé ELSE, défini par la conjonction des négations de tous les prédicats, est ajouté afin d'assurer la complétude.

2. **Fragmentation basée sur la classification k-means des prédicats :** Cette méthode utilise les prédicats de sélection extraits de la charge de requêtes et les classifie par la technique de fouille de données k-means présentée dans [55]. Elle se déroule en trois étapes.
   (a) Codage des prédicats de sélection de la charge de requêtes dans une matrice binaire qui représente le contexte de classification. Elle est similaire à la matrice d'usage des prédicats.
   (b) Classification des prédicats par application de la technique des k-means (47) qui permet de partitionner l'ensemble des prédicats en k classes disjointes. En pratique, les auteurs emploient la classe JavaSimpleKMeans du logiciel Weka[58] pour exécuter l'algorithme k-means.
   (c) Construction des fragments. Cette étape utilise l'ensemble des classes, obtenues dans l'étape précédente, et le document XML qui représente le schéma de l'entrepôt de données pour construire un nouveau document XML donnant un schéma de fragmentation de l'entrepôt.

1.3.2.4 Travaux de Derrar et al.

Les auteurs dans [45] proposent une approche de fragmentation dynamique des entrepôts de données centralisés. Cette approche considère un entrepôt de données fragmenté et en se basant sur un historique récent des accès aux données, l'approche refragmente l'entrepôt de données afin de minimiser le temps de réponse des nouvelles requêtes. Un critère d'évaluation est utilisé pour savoir si oui ou non une refragmentation de l'entrepôt est requise. Ce critère peut être le seuil de minimisation du temps de réponse des requêtes. Si le temps de réponse dépasse le seuil alors une refragmentation est exécutée. Concernant le stockage de l'historique des statistiques, il est fait en construisant des histogrammes. Un histogramme peut être utilisé pour stocker n'importe quel type de données comme l'estimation de la sélectivité des requêtes (nombre de tuples retournés). Soit la requête suivante :

```
SELECT *
FROM R
WHERE R.A < 20
```

Soit un histogramme défini pour estimer la sélectivité de la requête (figure 1.17). Il est construit sur l'attribut A en considérant 4 intervalles sur les valeurs de A I_1=[0,10[, I_2=[10,30[, I_3=[30,40[et I_4=[40,50[. A chaque intervalle [x,y[correspond une fréquence qui est le nombre de tuples sélectionnés par le prédicat $x \leq A < y$. Les intervalles I_1, I_2, I_3 et I_4 ont une sélectivité de 100, 50, 80 et 100 respectivement. Afin d'estimer la cardinalité du prédicat P : A<20, nous remarquons que P couvre

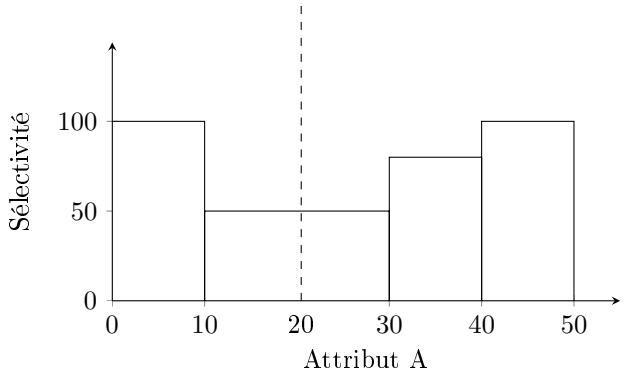


FIGURE 1.17 – Histogramme pour l'estimation de la sélectivité des requêtes

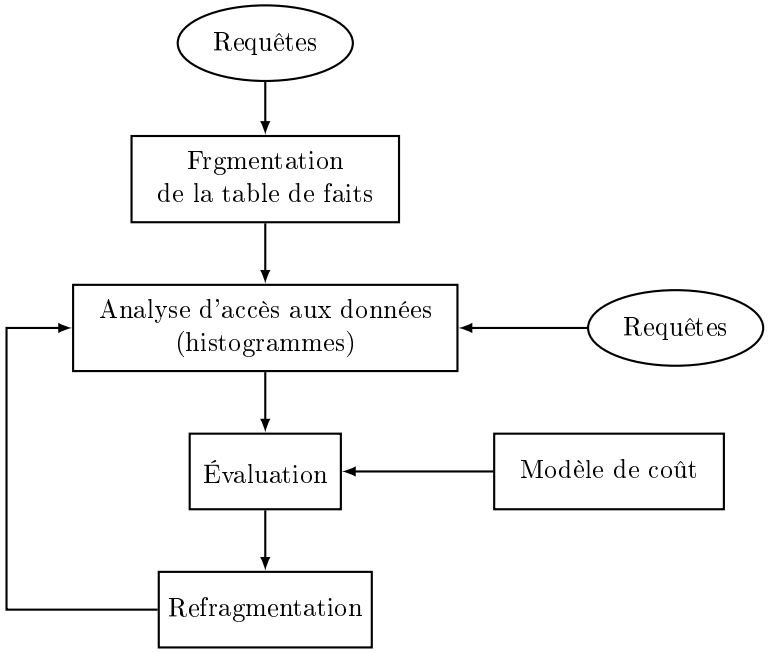


FIGURE 1.18 – Processus de fragmentation : Derrar et al.

complètement l'intervalle I_1 ce qui assure au moins une cardinalité de 100. Aussi, P couvre à 50% I_2 ce qui signifie que 50% des tuples de I_2 satisfont P. Par conséquent, la cardinalité de P :A<20 et estimée à 100+50/2=125 tuples.

Le processus de refragmentation, illustré dans la figure 1.18, est réalisé à travers les opérations suivantes :

1. Fragmenter l'entrepôt de données : seule la table des faits est fragmentée suivant le mode RANGE d'Oracle sur une de ses clés étrangères.
2. Implémenter les histogrammes : Pour chaque clé étrangère de la table de fait un histogramme est construit. Chaque intervalle I de construction de l'histogramme correspond à un fragment horizontal de la table de faits. Pour définir les valeurs de sélectivités des intervalles, une distribution

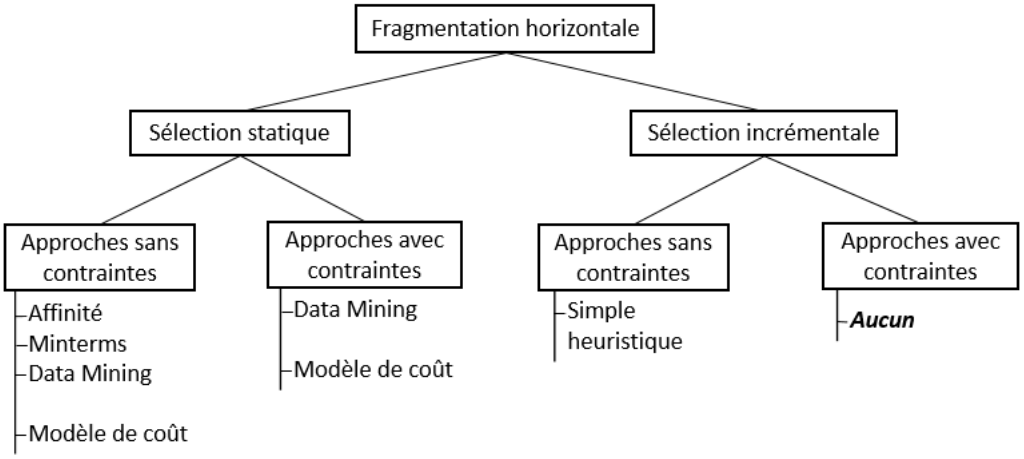


FIGURE 1.19 – Classification et Analyse des algorithmes de fragmentation horizontale

uniforme est utilisée attribuant la même sélectivité a chaque I. L'ensemble des histogrammes est recalculé à chaque refragmentation de la table de faits avec une contrainte sur le nombre maximum d'intervalles dans un histogramme (225 pour le SGBD Oracle).

3. Critère d'évaluation : ce critère représente le coût d'exécution de la charge de requête, il est estimé par le modèle de coût définit dans [26].
4. Refragmentation de la table des faits : à un moment donné, en utilisant les histogrammes et le modèle de coût, le critère est évalué. Si le critère d'évaluation est vérifié (le coût d'exécution des requêtes s'est dégradé) une refragmentation de la table de faits est réalisée suivi d'une mise à jour des histogrammes suivant le nouveau schéma de fragmentation.

1.3.3 Bilan et classification des travaux

En explorant les principaux travaux de fragmentation horizontale des entrepôts de données, nous distinguons quatre principaux types d'algorithmes utilisés : (1) les algorithmes basés sur la complétude et minimalité de prédicats COM-MIN [6], (2) les algorithmes d'affinités [6, 14, 56], (3) les algorithmes guidés par modèle de coût [12, 13, 45] et (4) les algorithmes de Datamining [56].

Une seconde classification permet de répartir les travaux en deux catégories : (1) les algorithmes sans contraintes sur le nombre de fragments faits générés [80, 76, 99, 45]. Dans ces travaux, la fragmentation de l'entrepôt de données est effectuée sans se soucier du nombre de fragments générés. En l'absence de contrôle sur ce nombre, un algorithme de fragmentation donné peut générer un très grand nombre de fragments qui devient difficiles à gérer. (2) Les algorithmes avec contraintes sur le nombre maximum de fragments que l'administrateur veut avoir. Les algorithmes existants sont les algorithmes guidés par modèle de coût [14] et les algorithmes de Datamining [56]. L'analyse et classification de ces travaux, illustrées sur la figure 1.19, nous ont conduit à faire les constatations suivantes :

- L'absence de contrôle sur le nombre de fragments faits généré peut engendrer un très grand nombre de fragments qui devient difficiles à gérer. Il est plus judicieux de formaliser le problème de sélection d'un schéma de fragmentation en un problème d'optimisation qui considère une contrainte d'optimisation sur le nombre de fragments faits générés.

- Peu d'algorithmes étudiés utilisent des structures de données (telles que des tableaux) pour représenter leurs solutions (schéma de fragmentation). Ces structures sont utilisées pour représenter n'importe quel schéma de fragmentation comme dans [14]. L'utilisation des structures de données facilite l'implémentation de plusieurs algorithmes tels que des algorithmes génétiques qui sont basées principalement sur le codage d'un schéma de fragmentation en chromosome.
- La majorité des algorithmes existants sont statiques et ne traitent pas l'aspect dynamique liés aux requêtes et l'entrepôt (ajout de nouvelle requête, changement de la fréquence d'exécution d'une ou plusieurs requêtes). A notre connaissance, seul le travail accompli par [45] propose une approche de sélection incrémentale d'un schéma de fragmentation dans le contexte d'entrepôts de données relationnels centralisés. Cette approche utilise des statistiques récemment stockés dans des histogramme afin de refragmenter l'entrepôt. Cependant, les auteurs proposent la fragmentation de la table de faits sur un seul attribut clé qui pointe une seule table dimension alors qu'il est plus bénéfique de refragmenter la table de faits suivant plusieurs attributs de plusieurs tables dimensions. En raison de l'aspect évolutif de l'entrepôt de données et de la nature ad hoc des requêtes, le développement d'algorithmes incrémentaux pour la fragmentation horizontale devient une nécessité.

Nos propositions : Selon l'analyse que nous venons de faire, nous faisons les propositions suivantes :

- Proposer une structure de données flexible pour coder un schéma de fragmentation.
- Modéliser toutes les opérations pouvant être appliquées sur la structure flexible sous forme d'une Algèbre de Fragmentation. Elle permet de formaliser le processus de passage d'un schéma de fragmentation à un autre.
- Proposer une démarche de fragmentation basée sur la structure de donnée flexible qui peut prendre en charge la fragmentation statique ou incrémentale.
- Proposer un algorithme génétique qui exploite la structure de données pour la sélection statique et incrémentale.

1.4 Sélection jointe des techniques d'optimisation

Nous avons présentés jusque là les travaux qui traitent la sélection des structures d'optimisation pour optimiser les requêtes de jointures en étoiles et cela pour une seule technique d'optimisation donnée (technique d'indexation ou technique de fragmentation). Ces travaux se placent dans un cadre de sélection isolée où chaque technique d'optimisation est sélectionnée seule. Un autre type de sélection appelée *sélection jointe* vise à sélectionner plusieurs instances de plusieurs techniques d'optimisation, qui peuvent être redondantes ou non redondantes, afin d'optimiser une charge de requête. La sélection jointe a suscité un grand intérêt dans la communauté des entrepôts de données pour les raisons suivantes :

1. La sélection jointe permet de couvrir l'optimisation d'un plus grand nombre de requêtes. En effet, les requêtes sont nombreuses de l'ordre de plusieurs dizaines, très complexes et s'exécutent sur un grand volume de données (table de fait de plusieurs Gega ou Tera octets). Donc, sélectionner une seule technique d'optimisation n'est pas suffisant [27].
2. La sélection jointe permet la sélection de techniques redondantes et non redondantes ce qui permet de respecter l'espace de stockage alloué tout en s'assurant d'optimiser les requêtes complexes.
3. Vu la complexité de l'espace de recherche pour les problèmes de sélection, une technique d'optimisation peut être employée afin d'élaguer l'espace de recherche d'une autre TO [89, 12].

Le problème de sélection jointe des techniques d'optimisation (PSJ) peut être formulé en un problème d'optimisation de la manière suivante :
Étant donné :

- un entrepôt de données,
- une charge de m requêtes $Q = \{Q_1, Q_2, \cdots, Q_m\}$ où chaque requête Q_j est munie d'une fréquence d'accès f_j,
- des contraintes d'optimisation $C = \{C_1, C_2, \cdots, C_p\}$ comme l'espace de stockage alloué pour les TOs redondantes comme les index et les vues matérialisées, le temps de maintenance comme le temps nécessaire pour recalculer les vues matérialisées et les index lorsque la population de l'entrepôt de données augmente ou encore le nombre maximum de fragments de la tables de faits générés pour la fragmentation horizontale.

Le PSJ consiste à sélectionner une configuration de structures d'optimisation tels que :

- le coût d'exécution de la charge Q en présence de cette configuration est optimisé,
- les contraintes d'optimisation C sont toutes respectées.

Nous savons déjà que l'espace de recherche d'une seule technique d'optimisation est très complexe. En effet, les problèmes de sélection d'un ensemble d'index ou d'un schéma de fragmentation par exemple sont connus NP-Complet. Par conséquent, la taille de l'espace de recherche du PSJ augment de manière combinatoire. Si les tailles des espaces de recherche pour trois TOs sont respectivement NT_1, NT_2 et NT_3 alors la taille de l'espace de recherche pour le PSJ, défini sur les trois TOs, est $2^{NT_1+NT_2+NT_3}$ [100].

1.4.1 Types de sélection jointe

Il existe deux types de sélections jointes : la sélection dite itérative et la sélection intégrée [100].

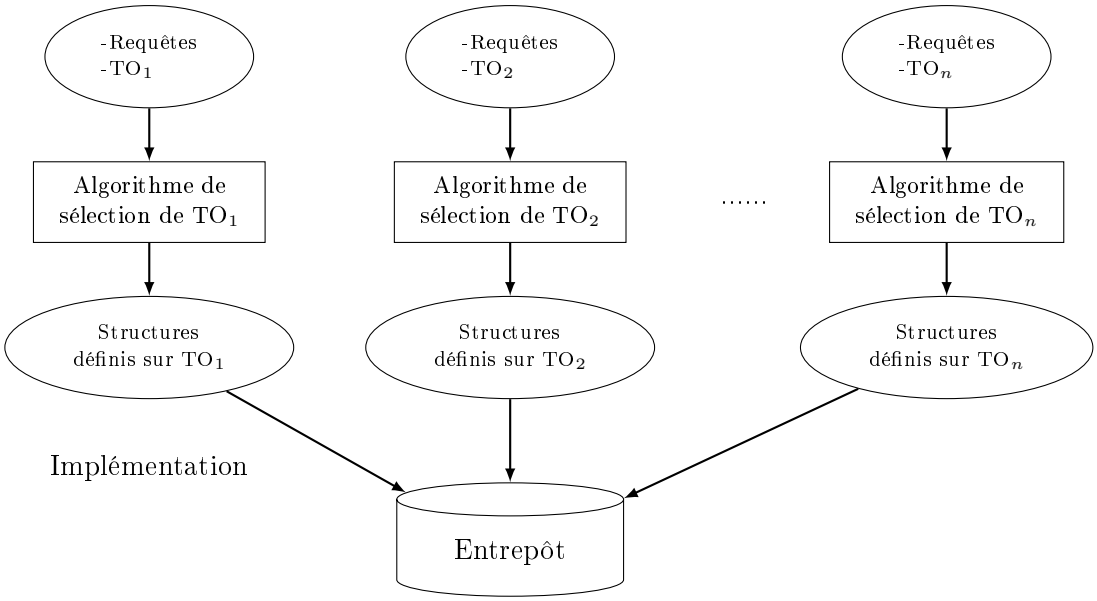


FIGURE 1.20 – Principe de la sélection itérative

1.4.1.1 Sélection itérative

La sélection itérative consiste à sélectionner les techniques d'optimisation les unes après les autres de manière totalement indépendante. Chaque sélection emploi un algorithme de sélection qui lui est propre et permet de sélectionner les structures d'optimisations en considérant toute la charge de requêtes. Le principe de la sélection itérative est illustré sur la figure 1.20.

L'avantage de la sélection itérative est qu'elle considère chaque sélection d'une TO donnée comme une boite noire qui prend en entrée l'entrepôt de données et la charge de requête et retourne un ensemble de structures d'optimisation de cette TO. Par conséquent, il est facile d'ajouter une nouvelle TO en ajoutant une nouvelle boite noire qui implémente sa sélection. Cependant, son principale inconvénient est qu'elle ne prend pas en compte la dépendance qu'il y a entre les différentes TO. Prenant l'exemple d'une vue matérialisée et des index. Un index peut considérablement augmenter le bénéfice d'une vue qu'il indexe car une vue indexée est plus bénéfique pour l'optimisation des requêtes. Également, une vue matérialisée peut rendre un index inutile et inversement. On pourrait donc supprimer les structures inules ce qui optimise considérablement l'espace alloué pour stocker les différentes structures d'optimisation [101].

1.4.1.2 Sélection intégrée

Le principe de la sélection intégrée est d'employer un seul algorithme pour sélectionner au même temps un ensemble de structures d'optimisation sur toutes les techniques d'optimisation considérées. L'algorithme exploite un espace de recherche combiné très complexe qui augmente de manière combinatoire à l'ajout de chaque nouvelle TO. L'avantage de la sélection intégrée est qu'elle permet de prendre en compte les dépendances qui existent entre les TOs conjointement sélectionnées. Par exemple, le Microsoft Tuning Wizard emploi la sélection jointe intégrée des index et des vue matérialisées pour sélectionner les structures d'optimisation optimales tout en prenant en compte les interactions et dépendances entre ces deux TOs [85]. L'inconvénient de la sélection intégrée, en plus

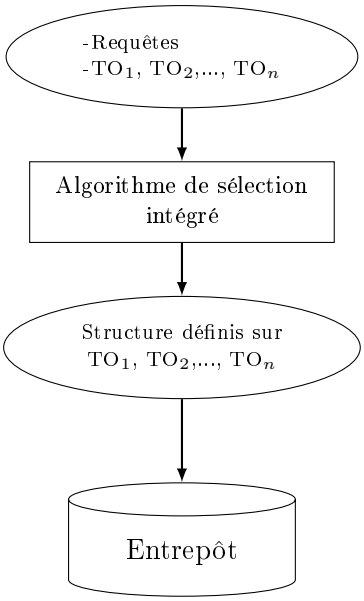


FIGURE 1.21 – Sélection jointe intégrée

de l'espace de recherche complexe, et la difficulté d'ajouter de nouvelles TOs car cela nécessite de changer l'implémentation de l'algorithme de sélection ce qui rend le système de sélection difficilement extensible. La figure 1.21 illustre le principe de la sélection intégrée.

1.4.1.3 Choix du type de sélection jointe

Afin de déterminer le type de sélection jointe (itérative ou intégrée) à employer pour sélectionner un ensemble de techniques d'optimisation, la notion de dépendance entre les TOs à été définie [100]. La dépendance d'une technique d'optimisation a une autre représente l'influence qu'a la seconde technique sur l'implémentation de la première. Il existe deux degrés de dépendance la dépendance faible et la dépendance forte. On dit qu'une technique d'optimisation TO_1 dépend fortement d'une autre technique d'optimisation TO_2 si un changement dans la sélection de TO_2 cause un changement de celle de TO_1. Si cette condition n'est pas vérifiée, on dit que TO_1 dépend faiblement de TO_2. Prenant l'exemple de la fragmentation horizontale et des index de jointures binaires. Lors de la sélection jointe de ces deux techniques, il faut prendre en compte que les index peuvent à leur tour être fragmentés en sous-index de manière horizontale. On dit donc que la technique d'indexation dépend fortement de la technique de fragmentation. Le tableau 1.10 résume les degrés de dépendances entre trois techniques : indexation, fragmentation horizontale et vues matérialisées.

-	Index	Vues M	FH
Index	-	Forte	Forte
Vues M	Forte	-	Faible
FH	Faible	Forte	-

TABLE 1.10 – Dépendances entre trois techniques d'optimisations

En utilisant les dépendances entre les techniques d'optimisation, il est possible de savoir s'il est

préférable d'employer la sélection jointe itérative ou intégrée de la manière suivante [100, 27] :

1. Si deux techniques d'optimisation TO_1 et TO_2 dépendent chacune fortement de l'autre, alors l'approche jointe intégrée est employées. D'après le tableau 1.10, l'indexation et la technique des vues matérialisées sont candidates pour la sélection jointe intégrée.
2. Si une technique TO_1 dépend fortement de TO_2 mais que TO_2 dépend faiblement de TO_1, alors la sélection jointe itérative est employée mais avec un ordre d'exécution précis ; il faut d'abord sélectionner TO_2 puis TO_1. L'indexation et de la fragmentation horizontale illustre bien ce cas. D'après le tableau 1.10, l'indexation dépend fortement de la fragmentation et l'inverse est une dépendance faible. Par conséquent, la sélection de la fragmentation horizontale est exécutée en premier suivi de la sélection des index [27].
3. Si deux techniques TO1 et TO2 sont faiblement dépendantes, alors l'approche itérative est employée dans n'importe quel ordre.

1.4.2 Travaux de sélection jointe des $\mathcal{IJB}$ et de la $\mathcal{FH}$

Nous nous intéressant dans nos travaux aux deux techniques d'optimisations qui permettent d'optimiser les requêtes de jointes en étoiles ; l'indexation et la fragmentation horizontale. Ainsi, nous présentons dans ce qui suit les travaux de sélection jointe de ces deux techniques.

1.4.2.1 Travaux de Stöhr et al

Les auteurs dans [89] proposent une approche de sélection jointe de la fragmentation horizontale et l'indexation par index bitmap, cela dans le contexte d'entrepôts de données parallèles implémentés sur une machine parallèle de type shared disk ayant K disques. Cet entrepôt, modélisé en étoile, est composé d'une table de faits volumineuse et de plusieurs tables de dimension. Les auteurs considèrent une charge de requêtes à optimiser qui permet de définir un ensemble d'index bitmaps, puis de fragmenter à la fois les tables et les index. Pour ce faire, ils définissent une démarche de Fragmentation Hiérarchique Multidimensionnelle (MDHF) qui se base sur les hiérarchies des attributs dans les tables dimensions. Pour la table dimension Temps par exemple, une hiérarchie de ses attributs peut être Année→Trimestre→Mois→Jour, ce qui nécessite la duplication des données de chaque niveau de la hiérarchie suivant les niveaux plus haut (le mois Janvier de l'année 2008 diffère du mois de Janvier de 2009). La démarche, proposée par les auteurs et illustrée sur la figure 1.22, se déroule comme suit :

1. Définir des IJB bitmap encodés. Cette étape commence par définir un index bitmap pour chaque hiérarchie. Ceci est réalisé en attribuant à chaque valeur de chaque attribut d'une hiérarchie un vecteur bitmap. Le problème est l'explosion du nombre de bitmap. Prenant l'exemple de la hiérarchie de la table Temps Année→Trimestre→Mois→Jour. En considérant 6 années et vu que les valeurs des attributs de niveau hiérarchique inférieur sont dupliqués, le nombre de jours différents est 2160 (tableau 1.11). Un index bitmap défini sur l'attribut Jour possède donc 2160 bitmaps. Par conséquent, l'encodage de cet index donne un index avec seulement 17 bitmaps. En effet, le nombre de valeurs de chaque attributs sont codé en binaire. Pour l'attribut Année, les 6 années sont codées sur 3 bits ce qui signifie que l'année 1 à le codage 001, l'année 2 à le codage 010, etc. Un index bitmap 001.10.0110.00000001 permet de codé le jour 2 du mois de Juin du trimestre 2 et de l'année 1.
2. Définir un schéma de fragmentation sur la table de faits et ses index. A partir de chaque hiérarchie, un seul attribut est choisi pour la fragmentation. Le nombre de fragments fait total est le produit des cardinalité de tous les attributs de fragmentation.

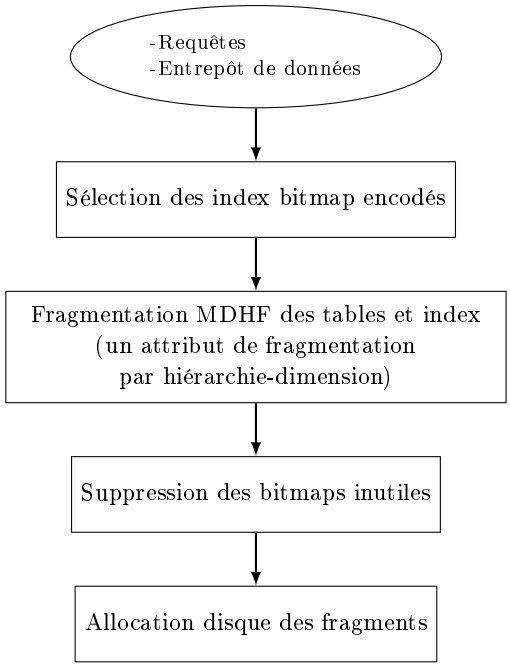


FIGURE 1.22 – Sélection jointe, Stohr et al.

	Année	Trimestre	Mois	Jour	Total Bitmap
Cardinalité Totale	6	8	72	2160	2160
Cardinalité sans hiérarchie	6	4	12	30	
Taille du codage(log2)	3	2	4	8	17
Codage	aaa	tt	mmmm	jjjjjjjj	aaattmmmmjjjjjjjj

TABLE 1.11 – Représentation d'un index bitmap encodé pour une hiérarchie

3. Écarter les bitmaps inutiles. Ce sont les bitmaps définis sur les attributs de fragmentation et leurs parents (attributs plus haut dans la hiérarchie). En effet, une fragmentation de la table Temps sur l'attribut Mois rend les bitmaps définis sur les attributs Mois, Trimestre et Année inutiles. Pour répondre à une requête qui contient le prédicat de sélection "Trimestre=2", il suffit de faire l'union des fragments qui vérifient le prédicat "Mois IN (4, 5, 6)" (le second trimestre contient les mois Avril, Mai et Juin).

Le principale avantage de cette approche est qu'elle permet d'élaguer l'espace de recherche des index grâce au codage et à l'élimination des bitmap inutiles par la fragmentation. Néanmoins, les auteurs se basent sur l'existence systématique de hiérarchie entre les attributs d'une dimension alors que dans la réalité, les dimensions peuvent avoir zéro ou plusieurs hiérarchies. De plus, aucun contrôle n'est effectué sur le nombre de fragments de la table de faits générés représentant le produit des cardinalités des attributs de fragmentation. Dans le cas où tous les attributs de FH choisi sont de basse hiérarchie, aucun index n'est requis, mais le nombre de fragments faits explose (la fragmentation sur l'attribut Jour uniquement génère 2160 fragments).

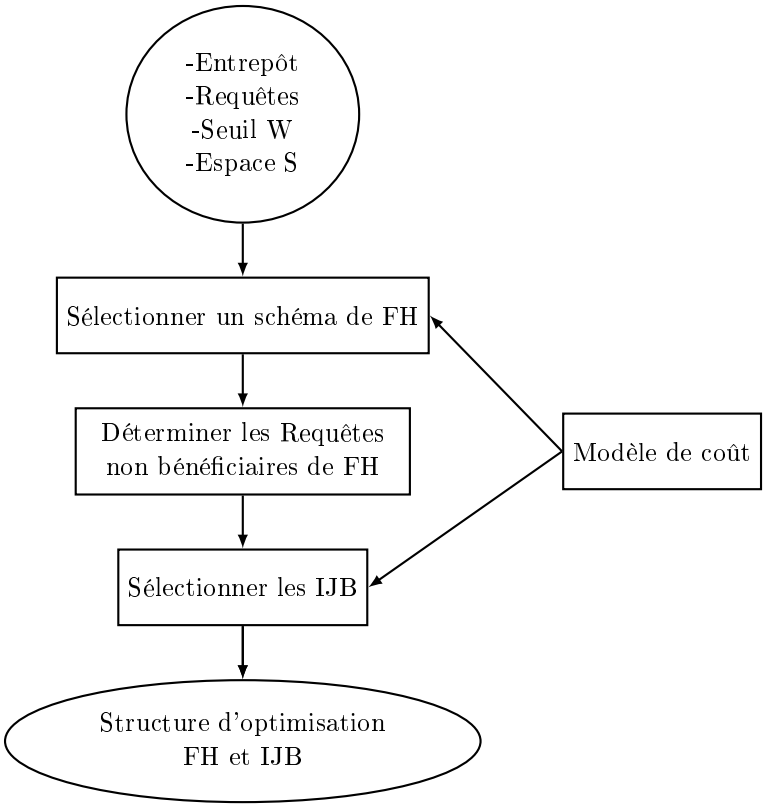


FIGURE 1.23 – Processus de sélection jointe : Boukhalfa et al.

1.4.2.2 Travaux de Boukhalfa et al

Les auteurs dans [26, 27] proposent une démarche de sélection jointe itérative de la fragmentation horizontal et des index de jointure binaires. Vu que la dépendance entre ces deux techniques est forte-faible, les auteurs effectuent la sélection d'un schéma de fragmentation horizontale, ensuite ils sélectionnent les index de jointures binaires. Les auteurs montrent que ces deux techniques représentent plusieurs similitudes :

- Elles permettent d'optimiser les opérations de sélections et jointures que renferment les requêtes décisionnelles.
- Elles permettent de réduire la complexité requêtes en réduisant le volume de données chargées.
- Elles sont définis sur les attributs non-clés des tables dimensions, elles partagent donc la même ressource.

Ces similitudes rendent la sélection jointe de ces deux techniques plus intéressantes car cela permet de couvrir l'optimisation d'avantage de requêtes sans oublier que seule l'indexation nécessite un espace de stockage. Le déroulement de la sélection jointe proposée, illustrée sur la figure 1.23, est le suivant :

1. Sélectionner un schéma de fragmentation. Les auteurs emploient leur approche de sélection isolée d'un schéma de fragmentation par méta-heuristique (Hill Climbing, Recuit Simulé, Algorithmes Génétiques), avec contrôle du nombre de fragments de la tables de faits [13, 14]. Cette approche de fragmentation est présentée dans la section 1.3.2.2.
2. Déterminer les requêtes non bénéficiaires de la fragmentation. Pour ce faire, les auteurs défi-

nissent une métrique représentant le taux d'optimisation d'une requêtes Q_i par un schéma de fragmentation SF, noté $taux(Q_i)$ et donné par la formule suivante :

$$taux(Q_i) = \frac{C[Q_i, SF]}{C[Q_i, \varnothing]}$$

Où $C[Q_i, SF]$ et $C[Q_i, \varnothing]$ sont respectivement le coût d'exécution de Q_i sur un schéma partitionné SF et non partitionné. Ensuite, les auteurs définissent un paramètre de tuning $\lambda \in [0, 1]$ fixé par l'administrateur. Ainsi, si $taux(Q_i) > \lambda$ alors Q_i est bénéficiaire, Q_i est non bénéficiaire sinon.

3. Sélectionner un ensemble d'index de jointure binaires. Sur les requêtes non bénéficiaires de la fragmentation, les auteurs exécutent leur démarche de sélection d'un schéma d'indexation présentée dans [9].

Pour valider leur démarche, les auteurs développent dans [11] un outil d'assistance à l'administration et le tuning des entrepôts de donnée appelé ParAdmin. Cet outil gère trois techniques d'optimisation ; la fragmentation horizontale primaire, la fragmentation horizontale dérivée et les index de jointure binaires. L'outil supporte la sélection isolée et jointe de ces techniques.

Les points forts de la démarche proposée par les auteurs est le contrôle du nombre de fragments faits, l'élagage de l'espace de recherche des $\mathcal{IJB}$ et la prise en compte des requêtes non bénéficiaires de la fragmentation pour la spécification des index ce qui permet de couvrir l'optimisation d'un maximum de requêtes. Cependant, l'approche présente quelques inconvénients. Comme la sélection d'un schéma de $\mathcal{FH}$ est effectué en premier, aucun élagage de son l'espace de recherche n'est effectué et il n'est pas sûr que les attributs laissé pour indexation soient les plus appropriés. Les $\mathcal{IJB}$ définis après fragmentation peuvent s'avérer inutiles pour l'optimisation ; si le volume de données chargé par index est tout aussi volumineux que la table même, l'optimiseur du SGBD jugera préférable d'utiliser la table directement. Aussi, des attributs peuvent être choisis pour la $\mathcal{FH}$ alors que leur utilisation pour définir un $\mathcal{IJB}$ donnerait un meilleur résultat. En effet, à cause de la contrainte W sur le nombre de fragments, les valeurs d'un attribut peuvent se regrouper dans un même domaine, cela obligerait le chargement d'une grande partie de données pour une requête comportant la sélection sur une seule valeur de l'attribut.

1.4.3 Bilan et discussion

L'analyse des travaux de sélection jointe de la fragmentation horizontale et de l'indexation nous a permis de déduire qu'une sélection jointe, permettant un compromis entre la sélection jointe intégrée et la sélection jointe itérative, est requise. En effet, il faut s'inspirer des deux types de sélections jointes et ne garder que les avantages de chacune. En d'autres termes, pour palier au problème de l'explosion combinatoire de la taille de l'espace de recherche pour la sélection intégrée il faut séparer les algorithmes de sélection, mais pour prendre en compte la dépendance entre les techniques d'optimisation, il faut partager l'environnement de sélection entre elles. L'environnement de sélection contient la charge de requêtes, les contraintes d'optimisation si elles sont communes ou encore le temps de maintenance s'il y a lieu. Les techniques redondantes par exemple partagent la même contrainte d'optimisation qui représente l'espace de stockage. Il faut partager cet espace en privilégiant les structures d'optimisation les plus bénéfiques et les moins couteuses en terme de stockage.

Nous remarquons également que l'une des similarités entre l'indexation et la fragmentation est qu'elles sont définies sur le même ensemble d'attributs issus des tables dimensions. Utiliser le même attribut pour définir un index et un schéma de fragmentation peut s'avérer inutile ; les deux structures

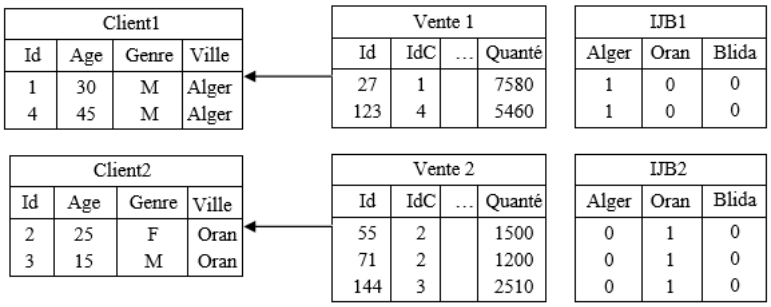


FIGURE 1.24 – Schéma de $\mathcal{FH}$ et $\mathcal{IJB}$ définis sur l'attribut Ville

d'optimisations peuvent avoir le même effet d'optimisation alors qu'il faut respecter aux mieux les contraintes d'optimisation (espace de stockage limité et nombre de fragments faits limité).

Exemple 10 *Considérons l'entrepôt de données de la figure 1.1. On fragmente la table Clients et on crée un $\mathcal{IJB}$ sur le même attribut Ville. La répartition des données est illustrée sur la figure 1.24. On remarque que l'optimisation par une seule technique est suffisante pour les requêtes qui effectuent des opérations de sélection sur l'attribut Ville.*

Enfin, tous les travaux de sélection jointe se situent dans le cadre de sélection statique où l'évolution de l'entrepôt de données n'est pas considérée. A notre connaissance, **il n'existe aucun travail qui traite de la sélection jointe incrémentale**. L'intérêt d'une telle sélection est de combiner plusieurs techniques d'optimisation pour optimiser un maximum de requête tout en adaptant les structures d'optimisation déjà implémentées sur l'entrepôt de données aux changements qui surviennent comme l'évolution de la charge de requêtes.

Nos propositions : Suivant l'analyse que nous venons de faire, nous proposons de réaliser deux sélections incrémentale jointes ; une sélection incrémentale jointe de la fragmentation et de l'indexation et une sélection incrémentale jointe des $\mathcal{IJB}$ simples et des $\mathcal{IJB}$ multiples.

1.5 Conclusion

Nous avons présenté dans ce chapitre un état de l'art sur deux techniques d'optimisation : les index de jointures binaires et la fragmentation horizontale. Nous avons exposé pour chaque technique les travaux existants et avons fait une classification des approches de sélections proposées. Nous avons également présenté les travaux de sélection jointe qui combinent ces deux techniques d'optimisation. L'analyse des travaux nous a conduit à déterminer les manques existants selon trois principaux axes : (1) la plupart des travaux considèrent un contexte statique de sélection où la charge de requêtes est préalablement connue et supposée non évolutive. (2) peu de travaux s'intéressent à combiner les structures d'optimisations pour mieux optimiser les requêtes décisionnelles et il n'existe aucun travail qui traite de la sélection jointe incrémentale. (3) Toutes les approches de sélections se basent sur une optimisation mono-objectif et considèrent un seul objectif à optimiser à savoir les performances des requêtes.

Nous présentons dans le chapitre suivants un état de l'art sur l'optimisation multi-objectif.

Chapitre 2

Optimisation multi-objectifs pour les entrepôts de données

Sommaire

2.1 Introduction

Les travaux de sélection des techniques d'optimisation que nous avons vu jusque-là peuvent être classés selon deux critères ; (1) les approches de sélection isolée où une seule technique d'optimisation est considérée et est sélectionnée et (2) les approches de sélection jointe qui visent à exploiter les similarités entres plusieurs techniques d'optimisation afin de les sélectionner conjointement pour apporter un meilleur bénéfice sur les performances des requêtes. Ces travaux se situent pour la plupart dans le cadre de sélection statique qui ne met pas à jour le schéma d'optimisation implémenté sur l'entrepôt lorsque ce dernier évolue. La plupart de ces travaux commencent par formaliser le problème de sélection en un problème d'optimisation mono-objectif où il faut choisir un ensemble d'instances ou structures d'optimisation (schéma d'optimisation) qui optimisent les performances des requêtes sous des contraintes d'optimisation, puis proposent des algorithmes de résolution. Cependant, l'étude de la sélection incrémentale a révélé que la mise à jour du schéma d'optimisation nécessite un temps d'implémentation et d'utilisation des ressources systèmes. Il est vrai que la sélection incrémentale permet de faire face à l'évolution de l'entrepôt mais elle engendre ce qu'on appelle *un coût de maintenance* de l'entrepôt de données qu'il faut optimiser également. Par conséquent, il faut étudier le problème de sélection multi-objectifs d'un schéma d'optimisation où plusieurs objectifs à optimiser sont considérés. Nous abordons dans ce chapitre l'optimisation multi-objectifs du point de vue concepts, définitions et méthodes de résolution. Ensuite, nous exposons l'état de l'art sur l'optimisation multi-objectifs dans le contexte d'entreposage de données. Ce chapitre est organisé comme suit. La section 2 expose la formalisation du problème d'optimisation multi-objectif. La section 3 aborde les méthodes de résolutions existantes. La section 4 présente les travaux qui se basent sur une formulation multi-objectifs dans le contexte des entrepôts de données. Enfin, la section 5 conclue le chapitre.

2.2 Problème d'optimisation Multi-Objectif

Le problème d'optimisation multi-objectifs PMO consiste à optimiser une fonction objectif fo, représentée sous forme d'un vecteur de plusieurs fonctions objectifs, comme suit :

$$\min_{x \in C}(F(x)) = \min_{x \in C}(f_1(x), f_2(x), \cdots, f_k(x)) \tag{2.1}$$

Où $F(x)$ représente un vecteur de k fonctions objectifs à optimiser avec $k \geq 2$. $x = (x_1, x_2, \cdots, x_k)$ est un vecteur de solutions appelé *solution faisable*. C est l'ensemble des solutions faisables associé avec le vecteur de p contraintes d'inégalités $G(x) = (g_1(x), \cdots, g_p(x))$, et un vecteur de m contraintes d'égalités $H(x) = (h_1(x), \cdots, h_m(x))$ et x^L, x^U représentant respectivement la borne inférieure et supérieure du domaine de recherche. La définition de l'ensemble C est donnée par l'équation suivante :

$$C = \{x \in \mathbb{R}^n / H(x) = 0, G(x) \leq 0 \text{ et } x^L \leq x \leq x^U\} \tag{2.2}$$

A chaque solution faisable x est associé un vecteur $z = F(x) \in \mathbb{R}^k$ appelé *vecteur objectif*. L'ensemble des vecteurs objectifs Z vérifie $F(C) = Z$. En résumé, la définition de la fonction objectif est donnée comme suit :

$$\begin{array}{rccc} F: & \mathbb{R}^n & \longrightarrow & \mathbb{R}^k \\ & x & \longmapsto & z = F(x) \end{array} \tag{2.3}$$

Afin de résoudre le PMO, il faut déterminer la ou les solution(s) faisable(s) $x \in C$ qui permet(tent) de minimiser la fonction objectif et donc de minimiser toutes les fonctions objectifs $\{f_1(x), f_2(x), \cdots, f_k(x)\}$.

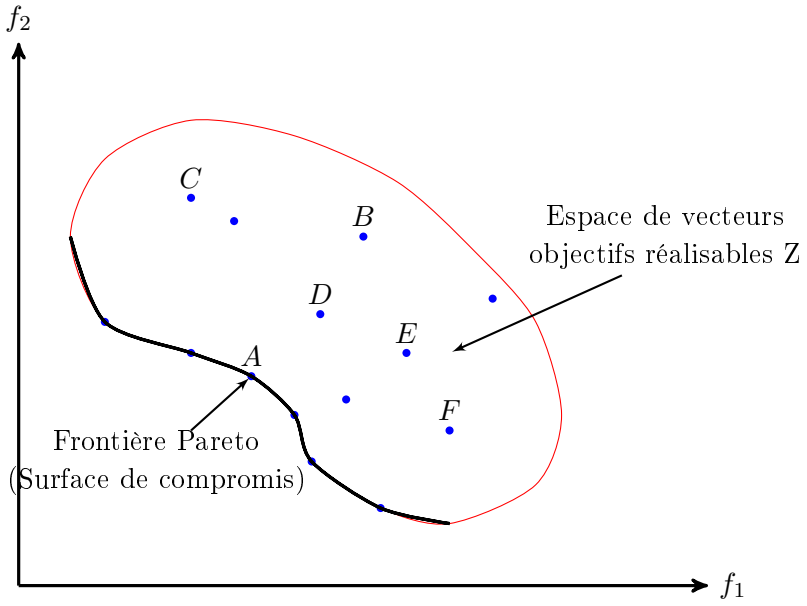


FIGURE 2.1 – Détermination graphique de la Surface de Compromis

Dans l'optimisation multi-objectif, il n'existe pas une unique solution faisable x comme c'est le cas dans l'optimisation mono-objectif. En effet, dans l'optimisation mono-objectif, il est possible d'ordonner les solutions selon l'unique fonction objectif puis de choisir la solution qui minimise cette fonction. Cependant, dans l'optimisation multi-objectif, l'ensemble C peut être partiellement ordonné permettant de trouver l'ensemble de solutions optimales appelées *Optimums de Pareto* dont les vecteurs objectifs correspondants forment la *surface de compromis*. La figure 2.1 illustre un ensemble de vecteurs objectifs et la détermination graphique de la surface de compromis dans un espace à deux objectifs f_1 et f_2. Pour ordonner les solutions faisables de C suivant la fonction objectif, il faut comparer entre leurs vecteurs objectifs correspondants. En mathématique, nous pouvons comparer deux vecteurs x et y appartenant à l'espace $\mathbb{R}^n$ comme suit :

- $x = y \Leftrightarrow \forall i \in [1, n], x_i = y_i$.
- $x < y \Leftrightarrow \forall i \in [1, n], x_i < y_i$
- $x \leq y \Leftrightarrow \forall i \in [1, n], x_i \leq y_i$

En appliquant ce principe sur les solutions illustrées sur la figure 2.1, nous concluons que $F(A) < F(D)$ et $F(D) < F(B)$. Nous pouvons donc ordonner les solutions faisables comme suit : $A < D$ et $D < B$. Cependant, il n'est pas possible de comparer les solutions D et E car $f_1(D) < f_1(E)$ mais $f_2(D) > f_2(E)$. En d'autre termes, $D \not< E$ et $E \not< D$. E et D représentent des solutions indifférentes. Ainsi, une nouvelle comparaison entre vecteurs dite au sens Pareto est introduite.

Définition 1 *Relations au sens Pareto*
En considérant deux solutions réalisables $A, B \in C$, trois relations entre A et B sont possibles au sens Pareto :

1. *$A \prec B$ (A domine B) si et seulement si $F(A) < F(B)$.*
2. *$A \preceq B$ (A domine faiblement B) si et seulement si $F(A) \leq F(B)$.*
3. *$A \sim B$ (A est indifférent par rapport à B) si et seulement si $F(A) \not\leq F(B) \wedge F(B) \not\leq F(A)$.*

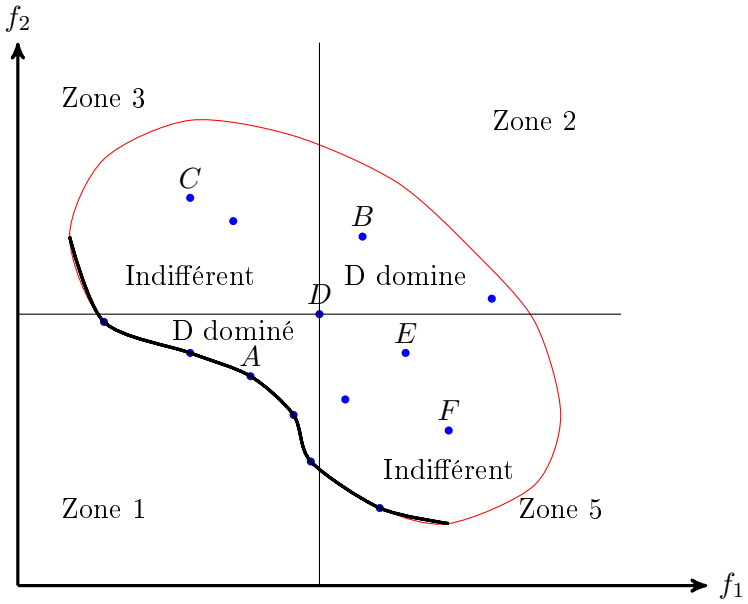


FIGURE 2.2 – Dominance Pareto : Relations entre solutions faisables

Définition 2 *Dominance de Pareto*
On dit qu'une solution faisable $A \in C$ ***Pareto domine*** *une solution faisable* $B \in C$ *si [37] :*
$\forall i \in \{1, 2, \ldots, k\}, f_i(A) \leq f_i(B)$
$\exists j \in \{1, 2, \ldots, k\}, f_j(A) < f_j(B)$

La figure 2.2 illustre quatre zones de relations entre les vecteurs objectifs et le vecteur $F(D)$. La première zone constitue les solutions qui dominent D. La seconde zone représente les solutions dominées par D. La troisième et quatrième zone contiennent les solutions indifférentes par rapport à D. Par exemple, on dit que D Pareto domine B.

Définition 3 *Optimum de Pareto*
Une solution $A \in C$ *est dite* ***Optimum de Pareto****, s'il n'existe aucune autre solution qui la domine, c'est à dire :* $\nexists B \in C$ *tel que :* $B \preceq A$ *ou* $B \prec A$.

Définition 4 *Ensemble d'Optimum de Pareto* P^*
Les optimums de Pareto représentent les solutions qui dominent toutes les autres solutions de l'ensemble C *mais qui ne se dominent pas entre elles. Soit* $P^* \subset C$ *l'ensemble des optimums de Pareto. L'ensemble* P^* *vérifie ce qui suit :*
$\forall A \in P^*, \forall B \in (C - P^*)$ *:* $A \preceq B$ *et*
$\forall A, D \in P^*$ *:* $A \sim D$

Définition 5 *Frontière de Pareto ou surface de compromis* F^*
La frontière de Pareto F^**, appelée aussi surface de compromis, est constituée des vecteurs objectif correspondants aux optimums de Pareto. On note* $F^* = \{F(x) | x \in P^*\}$.

La figure 2.3 illustre la relation entre les Optimums de Pareto. Nous remarquons que ces solutions sont indifférentes entre elles. En effet, pour les solutions A et G, $f_1(A) < f_1(G)$ mais $f_2(A) > f_2(G)$.

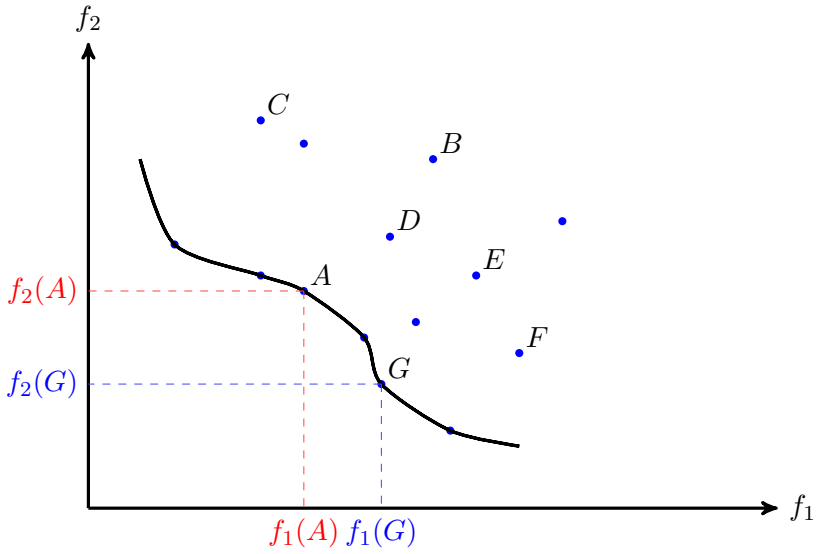


FIGURE 2.3 – Relation entre les Optimums Pareto

La solution Optimum de Pareto est une solution qui ne peut être améliorée sur une fonction objectif sans dégrader au moins une autre fonction objectif [88].

2.3 Approches de résolution du *PMO*

Déterminer l'ensemble des solutions Optimums de Pareto est une première phase dans le processus de résolution d'un *PMO*. La seconde phase consiste à choisir une solution optimale parmi cet ensemble. Un choix qui revient au décideur. Il existe plusieurs méthodes de résolutions d'un *PMO* qui peuvent aider le décideur à trouver la solution optimale. Elles peuvent être réparties en trois catégories [37] :

- **Méthodes à priori** : le décideur détermine les compromis sur les différentes fonctions objectifs qui composent la fonction $F(x)$ en spécifiant le poids de chaque fonction objectif. Le problème devient alors un problème mono-objectif où les fonctions objectifs sont fusionnées en une seule. Ensuite, l'exécution de la méthode d'optimisation est réalisée afin d'obtenir la solution optimale.
- **Méthodes à posteriori** : Le décideur choisit, selon ses préférences et son expérience, une solution parmi les solutions de l'ensemble Pareto optimale obtenu après exécution de l'optimisation.
- **Méthodes progressives** : ces méthodes se basent sur le questionnement continuel du décideur durant l'exécution du processus d'optimisation afin qu'il réoriente la recherche vers des solutions qui satisfont les compromis entre les fonctions objectifs. Le principal problème avec ces méthodes est qu'elles monopolisent l'attention du décideur tout au long de l'optimisation.

Nous proposons la classification des démarches de résolution du *PMO* en deux grandes catégories : Les démarches scalaires dont le principe et de ramener le *PMO* en un problème mono-objectif et les démarches à base de méta-heuristiques qui visent à optimiser simultanément toutes les fonctions objectifs et se basent sur des processus de sélection de l'ensemble des optimums Pareto. Nous présentons ces deux catégories dans ce qui suit.

2.3.1 Les méthodes scalaires

Les méthodes scalaires sont basées sur la transformation du PMO en un problème mono-objectif qui optimise une seule fonction objectif. Afin d'effectuer cette transformation, plusieurs méthodes existent comme la méthode de pondération des fonctions objectif, la méthode du compromis, la méthode dite du "but à atteindre", etc [37], [71].

2.3.1.1 La méthode de pondération des fonctions objectif

La méthode de pondération, initialement introduire par Gass et Saaty en 1955 [51], représente la méthode de résolution la plus évidente appelée aussi *Approche Naïve* [35]. C'est une méthode de résolution a priori du problème multi-objectif. Le principe est de combiner toutes les fonctions d'objectif pour former une seule fonction, à l'aide d'opérations simples telles que la somme pondérée. La somme pondérée est basée sur la somme de toutes les fonctions objectifs en affectant un coefficient de pondération pour chaque fonction. Ces coefficients représentent l'importance relative que le décideur attribue à une fonction donnée. Une seule fonction objectif est exprimée comme la somme des fonctions objectifs pondérées de la manière suivante :

$$F(x) = \sum_{i=1}^{k} w_i \times f_i(x) \tag{2.4}$$

Tel que les poids satisfont la condition :

$$\sum_{i=1}^{k} w_i = 1 \tag{2.5}$$

La formulation du PMO devient alors :

$$\min_{x \in C}(F(x)) = \min_{x \in C}(\sum_{i=1}^{k} w_i \times f_i(x)) \tag{2.6}$$

Avec $C = \{x \in \mathbb{R}^n / H(x) = 0,\ G(x) \leq 0$ et $x^L \leq x \leq x^U\}$. Cette méthode est l'une des premières méthodes employées vu son efficacité et simplicité du point de vue implémentation et algorithmique. Cependant, son principale inconvénient est qu'elle ne garantit pas l'obtenions de la solution Optimum de Pareto pour les espaces non convexes [71].

2.3.1.2 La méthode Keeney-Raiffa

Cette méthode se base sur le même principe que la méthode de pondération où la fonction objectif est obtenue par produit de toutes les fonctions objectif. La fonction objectif résultante est appelée fonction d'utilité de Keeney-Raiffa [65]. La formulation simplifiée de la fonction objectif est donnée comme suit :

$$F(x) = \prod_{i=1}^{k} f(x)^2 + 1 \tag{2.7}$$

La formulation du PMO devient ainsi :

$$\min_{x \in C}(F(x)) = \min_{x \in C}(\prod_{i=1}^{k} f(x)^2 + 1) \tag{2.8}$$

2.3.1.3 La méthode du compromis

Les deux méthodes jusque-là présentées se basent sur le principe de fusionnement de toutes les fonctions objectifs en une seule fonction. Par contre, la méthode du compromis, appelée aussi méthode de la contrainte ϵ [31], considère en priorité un objectif unique à optimiser et utilise un vecteur de contraintes sur les $(k-1)$ autres fonctions objectifs pour transformer le problème en un problème mono-objectif avec contraintes d'inégalité. Ainsi, le problème PMO est donné comme suit, où $\epsilon_i \geq 0$:

$$\begin{cases} \min_{x \in C} F(x) = \min_{x \in C} f_1(x) \\ \text{avec } f_i(x) \leq \epsilon_i, \forall i \in \{2, \cdots, k\} \end{cases} \tag{2.9}$$

2.3.1.4 Les méthodes hybrides

Les méthodes hybrides se basent sur la combinaison de plusieurs méthodes de résolutions existantes en une seule. La méthode hybride la plus utilisé est la méthode de Corley. Elle combine la méthode de pondération des fonctions objectifs et la méthode du compromis comme suit :

$$\begin{cases} \min_{x \in C}(F(x)) = \min_{x \in C}(\sum_{i=1}^{k} w_i \times f_i(x)) \\ \text{avec } f_j(x) \leq \epsilon_j, j \in \{1, \cdots, k\} \end{cases} \tag{2.10}$$

2.3.1.5 La méthode du "but à atteindre"

Cette méthode est appelée également "The goal attaintement method", initialement introduite par Charnes et Cooper [32] puis améliorée dans [33]. Le principe est de choisir un vecteur objectif initial T représentant le but à atteindre et un ensemble de coefficients de pondération w_i, $i \in [1, k]$. Ensuite, on cherche à minimiser l'écart entre les vecteurs objectifs exploités et le vecteur T. Cet écart est appelé α. Le problème PMO devient alors :

$$\begin{cases} \text{Minimiser} & \alpha \\ \text{avec} & f_i(x) - \alpha w_i \leq T_i \\ \text{et} & x \in C \end{cases} \tag{2.11}$$

2.3.1.6 La méthode d'ordonnancement lexicographique

Le principe est d'optimiser les fonctions objectifs de manière isolées les unes après les autres. Le décideur établie un ordre de préférence des fonctions objectif. Ensuite, chaque fonction est optimisée en considérant les contraintes obtenues des précédentes optimisations [35]. La formulation du PMO est donnée comme suit : A l'étape 1, on résout le problème suivant :

$$\min_{x \in C} f_1(x) \tag{2.12}$$

Soit x_1^* la meilleure solution trouvée tel que $f_1(x_1^*) = f_1^*$. Cette dernière devient une contrainte sur la fonction f_1 dans la résolution à l'étape 2, dont le problème est formulé comme suit :

$$\begin{cases} \min_{x \in C} f_2(x) \\ \text{avec } f_1(x) = f_1^* \end{cases} \tag{2.13}$$

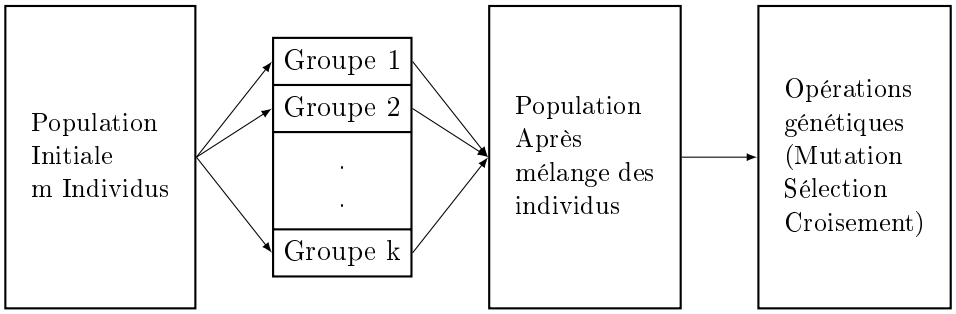


FIGURE 2.4 – Principe de la méthode VEGA

A l'étape k, le problème à résoudre est donné comme suit :

$$\begin{cases} \min_{x \in C} f_k(x) \\ \text{avec } f_1(x) = f_1^* \\ \text{et } f_2(x) = f_2^* \\ \dots \\ \text{et } f_{k-1}(x) = f_{k-1}^* \end{cases} \tag{2.14}$$

2.3.2 Les méthodes à base de méta-heuristiques

Le principe de ces méthodes est de résoudre les problèmes multi-objectifs en utilisant la dominance au sens Pareto. Les fonctions objectifs sont toutes optimisées simultanément afin de converger vers l'ensemble d'optimums de Pareto. Ces méthodes se basent principalement sur l'utilisation de méta-heuristiques comme les Algorithmes Génétiques, le Recuit Simulé ou encore la Recherche Tabou. Nous nous concentrons sur les méthodes qui emploient les algorithmes génétiques car ils sont très bien adaptés à la résolution d'un problème d'optimisation multi-objectifs et de plus en plus d'approches sont développées dans ce sens. Ils sont appelés MOEA (Multi-Objective Evolutionary Algorithms) [37].

2.3.2.1 VEGA (Vector Evaluated Genetic Algorithm)

Cette méthode est la première méta-heuristique multi-objectifs proposée [87] qui n'emploie aucune transformation sur les k fonctions objectifs à optimiser [35]. Elle considère une population de N individus, où N est multiple de k et répartie la population en k groupes, où chaque groupe contient $\frac{N}{k}$ individus. A chaque groupe, on affecte une fonction objectif qui permet d'évaluer chaque individu du groupe. Puis, on attribue un individu au groupe dont il optimise le mieux la fonction objectif correspondante. Ensuite, on applique les principes de génétiques (croisement, mutation, sélection) sur tous les groupes. Les étapes de la méthode VEGA sont résumées dans la figure 2.4.

Le principal inconvénient de cette méthode est qu'elle converge vers des solutions de bénéfice moyen dans tous les objectifs et non pas optimal. En effet, les populations exploitées par l'algorithme génétique sont constituées d'individus qui optimisent à chaque itérations un seul objectif à la fois et non pas tous les objectifs. Malgré l'efficacité et la facilité d'implémentation de cette méthode, elle se comporte comme une approche de pondération des fonctions objectifs dont elle hérite les inconvénients [35].

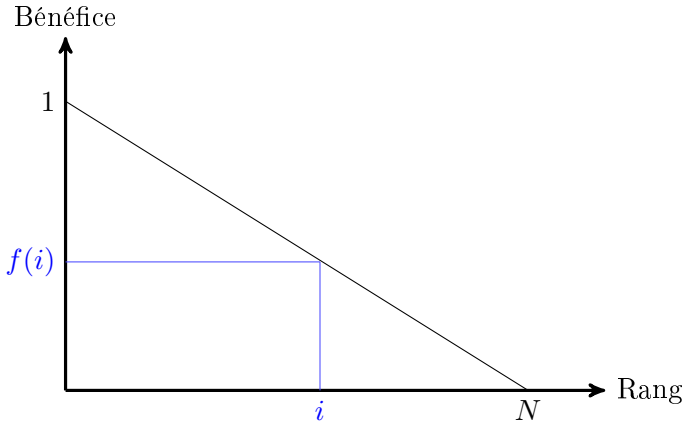


FIGURE 2.5 – Exemple d'une fonction de bénéfice $f(rang)$, méthode MOGA

2.3.2.2 MOGA (Multiple Objective Genetic Algorithm)

Cette méthode, présentée dans [49, 73], utilise la relation de dominance de Pareto pour déterminer le bénéfice d'un individu. Soit une génération de l'algorithme génétique qui exploite une population de N individus (N solutions). Pour chaque individu, on calcule son vecteur objectif et son *rang Pareto*. Le rang Pareto d'un individu représente le nombre d'individus qui le dominent au sens Pareto. Il permet d'établir une relation d'ordre entre les individus pour les classer. Ainsi les individus de rangs inférieurs apportent une meilleure optimisation des fonctions objectifs. On affecte le rang 1 à tous les individus non dominés et on calcul de manière itérative les rangs des autres individus. Prenons par exemple à l'itération t, un individu x_i qui est dominé par $p_i(t)$ individus. On attribue à x_i le rang $rang(x_i, t) = 1 + p_i(t)$. A la base du rang, on définit le bénéfice apporté par chaque individu comme suit :

- Classer les N individus selon un ordre croissant du rang.
- Utiliser une fonction linéaire décroissante $f(rang)$ qui permet d'attribuer un bénéfice aux individus par interpolation du meilleur (rang 1) au pire (rang N). La figure 2.5 illustre un exemple d'une fonction de bénéfice.
- Attribuer aux individus de même rang la moyenne de leurs bénéfices respectifs afin d'uniformiser le bénéfice apporté par les individus de même rang.

Ensuite, les opérations génétiques (sélection, croisement et mutation) sont exécutées afin de générer la nouvelle population de la prochaine génération. Puis, les fonctions objectifs pour chaque individu sont recalculées. Dans l'itération suivante, les individus de la nouvelle population sont une fois encore classés selon leurs rangs afin de déterminer les individus de rang 1. A la fin de l'exécution du processus, l'ensemble d'individus aillant le rang 1 représente l'ensemble d'optimums de Pareto.

Malgré son efficacité et sa facilité de mise en œuvre, le principale chalenge dans la méthode MOGA est de déterminer une fonction de bénéfice efficace qui permet de converger vers l'ensemble des optimums de Paretos [37].

2.3.2.3 NSGA (Non dominated Sorting Genetic Algorithm)

La méthode NSGA, proposée dans [88], emploi le même principe que la méthode MOGA décrite précédemment. La principale différence réside dans le calcul du bénéfice des individus. Soit une

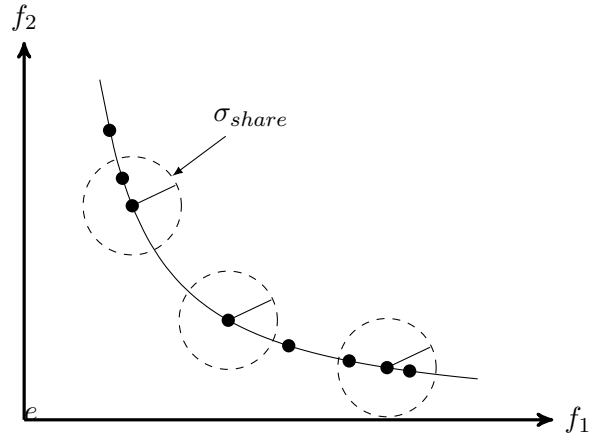


FIGURE 2.6 – Principe de calcul du bénéfice à base de voisinage, méthode NSGA

itération de l'algorithme génétique. On calcule le rang Pareto de tout les individus qu'on classe en S groupes $G_1, \cdots, G_S$, tel que les individus de rang i sont affectés au groupe G_i. Pour chaque groupe G_i, on affecte un bénéfice F_i inversement proportionnel à son rang de Pareto tel que $F_i = \frac{1}{rang_i}$. Ce bénéfice est partagé de manière uniforme sur tous les N_i individus de G_i tel que le bénéfice f_j d'un individu est la moyenne des bénéfices de ses m_j voisins dans G_i, en d'autre termes $f_j = \frac{F_i}{m_j}$.

Il est à noter que la plupart des méthodes basées sur les algorithmes génétique utilisent la notion de voisinage basée sur le calcul de distances entre les individus d'une population. La distance entre l'individu i et l'individu j, notée $d(i,j)$ est le nombre de gènes différents entre les deux, appelée aussi distance de Hamming. Deux individus sont voisins si la distance entre eux est inférieure à un paramètre choisi par le décideur, noté σ_{share} et appelé distance d'influence. Ainsi, le calcul du nombre de voisins m_j d'un individu est donné par la formule suivante :

$$m_i = \sum_{p=1}^{N_i} Sh(d(i,j)) \tag{2.15}$$

$$Sh(d(i,j) = \begin{cases} 1 - (\frac{d(i,j)}{\sigma_{share}})^2 & \text{si } d(i,j) < \sigma_{share} \\ 0 & \text{sinon} \end{cases}$$

La figure 2.6 illustre le principe de voisinage et de calcul du bénéfice. Une fois la classification, l'affectation et le partage du bénéfice effectués, on applique les opérations génétiques afin de générer une nouvelle population et on recalcule les fonctions objectifs pour toute la population. Tout comme l'approche MOGA, à la fin de l'exécution de l'algorithme génétique, les individus aillant un rang Pareto 1 constituent l'ensemble d'optimums de Pareto.

NSGA peut s'avérer inefficace vu la manière dont il classifie les individus [37]. C'est pour cela qu'une nouvelle variante appelée NSGA-II a été proposée. Elle introduit la notion de rang de non-dominance pour évaluer la force de chaque individu à part entière. De plus, pour converger plus rapidement vers les solutions optimales, la NSGA-II conserve les meilleures solutions d'une génération à une autre en utilisant des précédés d'élitisme [44].

2.3.2.4 NPGA (Niched Pareto Genetic Algorithm)

Cette méthode proposée dans [59] se base sur le principe général de la méthode NSGA en modifiant le processus de sélection génétique des solutions optimales et en utilisant un procédé d'élitisme pour les générations futures. Afin de trouver les solutions dominantes et non dominées parmi les individus d'une population (Optimums de Pareto), on commence par choisir deux individus au hasard et on les compare à une sous-population constituée aléatoirement de 10% de la population. Si l'individu 1 est dominé alors on sélectionne l'individu 2 pour la prochaine population et inversement. Si les deux individus sont dominés ou ne sont pas dominés alors on sélectionne pour la prochaine population l'individu avec le meilleur bénéfice et le minimum de voisins suivante le paramètre σ_{share}. Par la suite, on applique les opérations génétiques restantes sur la population actuelle à savoir le croisement et mutation et on termine par une réévaluation des fonctions objectifs.

Cette démarche utilise la dominance de Pareto pour la sélection sur une partie de la population pour chaque génération ce qui la rend très rapide. Cependant, son efficacité dépend de deux paramètres à savoir la distance d'influence σ_{share} et la taille de la sous population choisie.

2.3.2.5 SPEA (Strength Pareto Evolutionary Algorithm)

La méthode SPEA, présentée dans [102], se base sur le principe d'élitisme en sauvegardant un ensemble des solutions non dominées et trouvées dans des générations précédentes. A chaque génération, on ajoute à cet ensemble les individus non dominés. Chaque individu de l'ensemble se voit affecté un rang et un bénéfice calculé comme dans la méthode MOGA ; on affecte à chaque individu un rang représentant le nombre d'individus qui le dominent dans l'ensemble de sauvegarde. De la même manière, le rang et bénéfice de chaque individu dans la population actuelle est calculé à partir des rangs et bénéfices des individus contenus dans l'ensemble de sauvegarde et qui le dominent. Ensuite, les opérations génétiques sont effectuées entre les individus non dominés de l'ensemble sauvegardé et ceux de la population courante puis les fonctions objectifs sont recalculées pour tous les individus.

Le principal inconvénient de la méthode SPEA est l'augmentation continuelle de la taille de l'ensemble de sauvegarde qui pourrait ralentir la convergence vers les solutions optimales. Ainsi, une nouvelle approche appelée SPEA2 a été proposée et se base sur le principe de limitation de la taille de l'ensemble de sauvegarde afin qu'elle ne dépasse pas un certain seuil. De plus, deux autres améliorations ont été apportées dans SPEA2 : le rang de chaque individu est calculé à partir des individus qui le dominent et les d'individus que lui domine, et le principe de voisinage est employé pour déterminer le bénéfice de chaque individu [36].

2.3.2.6 PAES(Pareto Archived Evolution Strategy)

La méthode PAES, présentée dans [67], utilise une stratégie simple de recherche évolutionnaire locale. Elle exploite un seul individu à la fois pour la recherche des solutions optimale et sauvegarde un ensemble d'individus considérés comme des optimums de Pareto temporaires. Au début, le PAES génère aléatoirement un individu et l'ajoute à l'ensemble de sauvegarde qui est initialement vide. Chaque individu en cour d'exploitation est comparé à l'ensemble des optimums temporaires avant d'être muté pour une prochaine génération. Cet individu est considéré comme optimum de Pareto temporaire si et seulement si il n'est dominé par aucun des individus sauvegardés, auquel cas il est supprimé.

2.3.3 Frameworks pour la résolution du *PMO*

Afin de résoudre le *PMO*, plusieurs framework ont été développé principalement avec le langage Orienté-Objet JAVA. La plupart implémentent les méthodes précédemment présentées et basées sur les algorithmes génétiques. Nous citons dans ce qui suit une liste non exhaustive de ces Framworks.

ECJ [1] (Evolutionary Computation research system in JAVA) est une une bibliothèque Java orientée vers la recherche et développée à l'Université George Mason. Elle prend en charge une variété de techniques à base d'évolution tels que les algorithmes génétiques, la programmation génétique, les stratégies d'évolution, la co-évolution, l'optimisation par essaim de particules et l'évolution différentielle. Pour la résolution des *PMO* par algorithme génétiques, l'ECJ se base sur les méthodes NSGA-II et SPEA2. Cette bibliothèque est Open Source et est distribuée sous L'Academic Free License (AFL).

jMetal [2] (Metaheuristic Algorithms in Java) [62] est un framwork qui propose de résoudre les problèmes multi-objectifs à base de méta-heuristiques. Il implémente la méthode NSGA-II, SPEA2, PAES et d'autres méthodes comme PESA-II, OMOPSO, MOCell, etc. Il supporte également plusieurs types de données pour le codage des gènes de chromosomes comme le type binaire, réel, entier et permet des codages de type mixte (réel+binaire, entier+real).

Opt4J [3] [70] est un framwork Open Source JAVA modulaire pour l'optimisation à base méta-heuristique. Il inclut des méthodes d'optimisation multi-objectifs à base d'algorithmes génétiques comme les méthodes SPEA2 et NSGA2, mais aussi l'optimisation par essaim de particules, l'évolution différentielle et l'optimisation par Recuit Simulé. Opt4J offre une interface utilisateur graphique afin de faciliter le paramétrage et la visualisation du processus d'optimisation.

MOEA [4] (Multi-Objective Evolutionary Algorithms) est un framework Open Source pour la résolution des *PMO* à base d'algorithmes génétiques. Il supporte aussi la programmation génétique, les stratégies d'évolution, l'optimisation par essaim de particules et l'évolution différentielle. MOEA fournit les outils nécessaires pour concevoir, développer, exécuter et tester statistiquement les algorithmes d'optimisation. Il implémente 25 différentes approches tirées de l'état de l'art sur l'optimisation multi-objectifs par algorithmes évolutionnaires (NSGA-II, ϵ-MOEA, GDE3 et MOEA/D, etc.) et près de 80 problèmes analytiques. Il intègre les algorithmes du framework jMetal et peut supporter l'intégration de nouveaux problèmes ce qui le rend très utilisé dans le domaine de recherches scientifiques. En effet, il permet de tester de nouvelles méthodes avec une panoplie de méthodes et tests existantes.

2.3.4 Bilan

Dans leurs ouvrage appelé "Optimisation multiobjectif" [37], les auteurs montrent qu'il est très difficile de choisir une méthodes d'optimisation multi-objectifs pour résoudre un *PMO* concret vu le très grand nombre de méthodes existantes. Faut-il choisir une méthode à priori, à posteriori ou progressive ? Faut-il ramener le problème à un problème mono-objectif auquel cas quelle fonction objectif faut-il optimiser en priorité ? Si on emploi l'optimisation multi-objectifs avec dominance de Pareto, quelle algorithme choisir ? Quel Framework implémenter ? Etc. Afin de répondre à ses questions, les auteurs proposent quelques critères de choix résumés dans ce qui suit :

- La qualité de la solution : certains problèmes, comme les problèmes stratégiques, ont un optimum très difficile à trouver car ils exigent une qualité élevée de la solution finale. Pour les

1. http ://cs.gmu.edu/ eclab/projects/ecj/
2. http ://jmetal.sourceforge.net
3. http ://opt4j.sourceforge.net/
4. http ://www.moeaframework.org/

résoudre, les méta-heuristiques sont les plus adaptées. Prenant l'exemple d'un problème d'optimisation d'une structure dans le domaine de la construction. Une solution finale devrait être une structure avec une quantité moindre de matériaux, mais répondant au cahier des charges, qui pourrait faire économiser une somme importante sur le coût de la mise en œuvre.
- La nature de l'espace de recherche : par exemple, l'aspect convexe ou non convexe de la surface de compromis influe sur le choix de la méthode d'optimisation multi-objectif. En effet, si la surface de compromis est non convexe il ne faut pas choisir une méthode de pondération des fonctions objectifs car elle est inefficace pour ce cas [71].
- L'expression des préférences : si le décideur est en mesure d'exprimer ses préférences, on peut choisir une méthode de résolution de type à priori comme l'approche du compromis ϵ, la méthode du "but à atteindre" ou même la méthode d'ordonnancement lexicographique si le décideur exprime un ordre d'importance sur les fonctions objectifs. Sinon, il est préférable de choisir une méthode à posteriori comme celles basées sur les méta-heuristiques qui proposent un ensemble de solutions optimales au décideur. Ce dernier pourra alors choisir une solution qu'il jugera optimale parmi l'ensemble issu de la surface de compromis.

2.4 Travaux existants sur les EDs

Il existe des travaux qui emploient l'optimisation multi-objectifs pour résoudre des problèmes diverses dans les entrepôts de données selon plusieurs contextes : Cloud Computing, entrepôts de données temps réel, conception physique des entrepôts de données. Ils sont donnés dans ce qui suit.

2.4.1 Le Cloud Computing

Le Cloud Computing représente un ensemble de processus qui sont exécutés sur des ordinateurs ou serveurs informatiques distants liés à travers un réseau généralement Internet. Plusieurs clients possèdent chacun plusieurs utilisateurs qui exécutent plusieurs requêtes sur plusieurs serveurs. Ses serveurs peuvent être physiques ou des machines virtuelles et sont alloués à la demande et souvent par tranches selon la puissance, la bande passante, la mémoire et plusieurs d'autres critères de performance. Chaque serveur est alloué pour une période précise suivant un prix ou un forfait. Le Cloud est employé également pour stocker et interroger des bases de données distribuées ce qui a suscité un grand intérêt dans le domaine de recherche académique et industriel. En effet, le système de gestion des base de données Cloud doit gérer l'allocation des ressources pour chaque serveurs, optimiser plusieurs critères comme le prix d'allocation, exécuter les requêtes dans un contexte distribué et obtenir des résultats corrects en un temps raisonnable, etc. En d'autre terme, le système de gestion des bases de données Cloud doit optimiser un problème multi-critères formalisé comme suit : étant donné des contraintes budgétaires et une charge de requêtes, comment allouer les ressources du Cloud (CPU, Mémoire, bande passante du réseau, etc) à des machines virtuelles (serveurs), chacune ayant une partie d'une base de données distribuée, afin d'obtenir la meilleure performance globale des requêtes avec un prix minimum ? Les auteurs dans [46] proposent une approche d'optimisation multi-objectifs à base d'algorithmes évolutionnaires dans les entrepôts de données de type Cloud. Cette approche vise à optimiser les requêtes exécutées sur le Cloud en recherchant les solutions Pareto Optimales qui optimisent deux objectifs : le temps de réponses de requêtes et le coût monétaire. Afin d'exposer la démarche proposée, nous donnons les définitions suivantes :
- Virtual Machine (VM) est un logiciel qui émule l'architecture et les fonctions d'un véritable ordinateur. Le nombre de ses processeurs et de la mémoire principale peut être modifié grâce à

la virtualisation.
- Le temps d'exécution total est la somme entre le temps de calcul du CPU de la VM et le temps de la transmission de données sur le réseau.
- La charge de requêtes Q contient m requêtes $Q = \{Q_1, \cdots, Q_m\}$ à exécuter sur le Cloud.
- Le temps de réponse (RespTime) est le temps écoulé entre l'exécution d'une requête et l'obtention des résultats. Il est calculé avec les paramètres et statistiques utilisés par l'Optimiseur du SGBD.
- Le coût monétaire (MonCost) d'une charge de requêtes exécutée sur un Cloud représente la location des ressources pour exécuter la base de données. Le coût monétaire inclue le coût de stockage de données par les VMs et la somme des coûts de transferts de données.
- Plan d'exécution des requêtes (QP). Chaque requête peut avoir plusieurs plans d'exécution. Cela influe sur le temps d'exécution, le temps de réponse et le coût monétaire d'une requête qui peuvent changer en changeant son plan d'exécution.

Le problème multi-objectifs vise à optimiser la fonction suivante :

$$F(x) = \{RespTime(x), MonCost(x)\} \tag{2.16}$$

Où x est le vecteur solution constitué d'un ensemble de n VMs, m plans d'exécution QPs, m requêtes exécutées sur les n VMs sur un réseau donné.

$$x = \{\{VM_1, \cdots, VM_n\}, \{QP_1, \cdots, QP_m\}, Reseau_i\} \tag{2.17}$$

Pour résoudre ce problème, les auteurs proposent d'utiliser trois différents algorithmes : un algorithme heuristique simple (SHA), un algorithme de type branch-and-bound de recherche des solutions Pareto optimale (MOBB) et un algorithme basé sur la méthode MOGA.

1. **Le SHA** : les machines virtuelles sont classées en fonction de la fréquence des opérations de jointure. La VM classée première est celle où s'exécute la plupart des opérations de jointure. A cette VM est affectée la meilleure configuration (CPU, mémoire, bande passante, etc). Pour les VMs suivantes, on affecte des configurations avec un prix décroissant. Ensuite, les QPs sont évalués sur les VMs. Cet algorithme sert d'environnement de comparaison aux deux autres algorithmes proposés.
2. **Le MOBB** : c'est un algorithme d'optimisation exhaustive. Il énumère toutes les solutions candidates sous forme d'un arbre, où les sous-ensembles de solutions infructueuses sont éliminés en utilisant deux critères d'élimination : la minimisation du temps de réponse et la minimisation du coût monétaire. MOBB commence par une VM et calcul le meilleur temps de réponse et le coût moindre de la charge de requêtes sur cette VM. Pour chaque requête et chaque QP de cette requête, il estime de temps de réponse et le coût monétaire. S'il trouve pour chaque requête de Q un plan d'exécution qui permet de respecter ces deux critères alors la VM actuelle et ajoutée à la solution finale, sinon elle est écartée. Ce traitement est réalisé pour toutes les VMs.
3. **Le MOGA** : le problème multi-objectifs du Cloud est résolu par la méthode MOGA basée sur les algorithmes génétiques. MOGA exploite dans chaque itération une population d'individus ou chromosomes, où chacun correspond à une solution potentielle du PMO. La structure du chromosome, illustrée sur la figure 2.7, est composée des VMs, des QPs de la charge Q et du réseau. Afin de passer à la génération suivante, l'algorithme applique les opérations de croisement et mutation sur le chromosome mais chaque opération est appliquée localement sur la partie des VMs à part et la partie des QPs à part. A la fin de l'exécution, on obtient un ensemble de solutions Pareto Optimales,

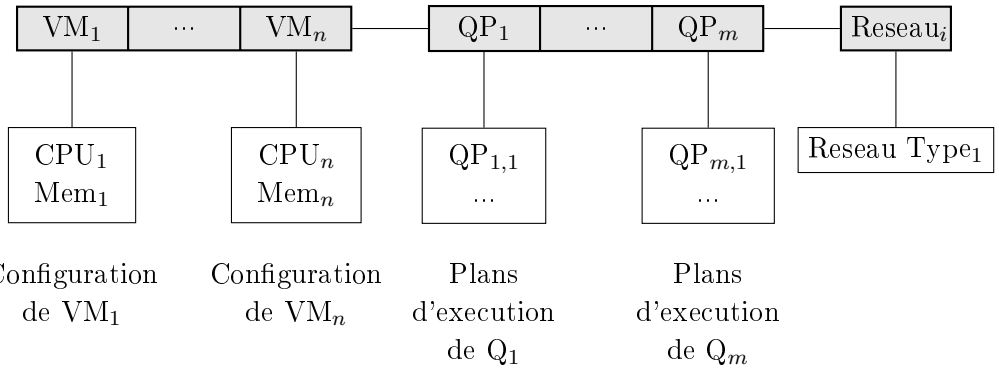


FIGURE 2.7 – Structure du chromosome pour l'application du MOGA sur le Cloud

2.4.2 Les entrepôts de données temps réel

Les entrepôts de données temps réel sont utilisés dans l'informatique décisionnelle afin de répondre à des exigences quotidienne et faire faces aux changements rapides des données. Contrairement aux entrepôts de données traditionnels, où les modifications sont chargées à des périodes précises (fin de journée ou durant le weekend par exemple), les données des entrepôts temps réel sont automatiquement capturées et sauvegardées et sont soumises à des modifications permanentes. Ainsi, lors de l'exécution d'une requête sur un entrepôt temps réel deux options sont possibles selon le critère qu'on souhaite optimiser le plus.

- Exécuter la requête sur des données périmées ou légèrement dépassées afin d'assurer une réponse rapide. Dans ce cas on favorise l'optimisation du temps de réponse de la requête appelée Qualité de Service QoS.
- Attendre la validation de toutes les modifications et mise à jour avant de lancer la prochaine requête. Dans ce cas, on favorise l'optimisation de la qualité de de données de la requête appelée Qualité de Données QoD.

Les auteurs dans [93] propose une approche d'ordonnancement multi-objectifs des requêtes dans les entrepôts de données temps réel. Ils considèrent une charge de requêtes Q composées de deux types de requêtes : les requêtes de lecture seule de type SELECT $q_i \in Q_q$ et les requêtes de modification de type UPDATE, INSERT ou DELETE $u_j \in Q_u$, avec $Q = Q_q \cup Q_u$. Le problème consiste à ordonnancer les modifications u_j et les requêtes q_i, afin d'optimiser deux objectifs qui sont la qualité de service (le temps de réponse) QoS et la qualité de données QoD.

Afin de mesurer la qualité de service QoS, les auteurs évaluent le temps de réponse, qui est généralement le principal objectif pour la plupart des approches d'optimisation. Le temps de réponse de la requête q_i est composé de son temps d'exécution e_{q_i}, le temps d'attente causé par l'exécution de requêtes précédentes, et le temps d'attente causé par l'exécution des mises à jour qui précèdent c_{u_j}. Ainsi, le modèle de la QoS pour une charge de requêtes donnée Q est défini comme suit :

$$QoS(Q) = \sum_{i=0}^{|Q_q|} (|Q_q| - i) \times (e_{q_i} + c_{u_j}) \tag{2.18}$$

Pour évaluer la qualité de données QoD des requêtes, les auteurs utilisent les profits sur la qualité des données apportés par les modifications dont ces requêtes dépendent. Les auteurs considèrent l'entrepôt de données réparti en partitions. On note P_{q_i} les partitions accessibles lors de l'exécution

de la requête q_i. Étant donnée une requête q_i et une modification u_j, on dit que q_i dépend de u_j si u_j est exécutée avant q_i et que les deux accèdent aux mêmes partitions, c'est à dire $P_{q_i} \cap P_{u_j} \neq \emptyset$. Le profit apporté par une u_j, noté p_{u_j}, est exprimé avec un nombre entier positif, où une valeur plus élevée correspond à une augmentation de la qualité des données. Ainsi, si une u_j est exécutée avant une q_i qui dépend d'elle, alors la qualité des données de q_i est améliorée par la valeur du profit p_{u_j}. Ainsi, le modèle de qualité de données d'une charge Q est donné comme suit :

$$QoD(Q) = \sum_{q_i \in Q_q, u_j \in Q_u, P_{q_i} \cap P_{u_j} \neq \emptyset} p_{u_j} \tag{2.19}$$

Une fois les deux objectifs modélisés, les auteurs recherchent à maximiser la qualité de données tout en maximisant la qualité de service, En d'autres termes, maximiser la qualité de données en minimisant le temps de réponse des requêtes. Pour ce faire, le problème est formalisé comme un problème du Sac à Dos pour rechercher les solutions Pareto optimales, tel que chaque solution représente un ordonnancement possible des q_i et u_j. Le problème multi-objectifs est alors ramené à un problème mono-objectif où il faut maximiser la qualité de données avec une contrainte sur le temps de réponse ce qui favorisent les modifications par rapport aux requêtes SELECT. La formulation du problème devient comme suit :

$$\begin{cases} \text{Maximiser} & \sum_{j \in |Q_u|} P_{u_j} u_j \\ \text{avec} & \sum_{j \in |Q_u|} c_j u_j \leq B \end{cases} \tag{2.20}$$

Pour résoudre ce problème, les auteurs proposent d'utiliser un algorithme de programmation dynamique souvent utilisé pour résoudre le problème NP-Complet du Sac à Dos. Le problème prend comme entrée les requêtes de modifications u_j, leurs profits et coûts correspondants ainsi que la contrainte B sur le temps de réponse. L'algorithme permet de sélection une solution optimale représentant un ordonnancement des modifications qui respecte les contraintes du problème.

2.4.3 La conception physique des entrepôts de données

L'optimisation multi-objectifs peut être utilisée pour formuler le problème de sélection d'une technique d'optimisation. En effet, dans les travaux qui se basent sur l'optimisation mono-objectif, seul le coût de la charge de requête est considéré comme objectif à optimiser. Or, pour plusieurs techniques d'optimisation, d'autres objectifs à optimiser rentre en jeu. Par exemple, pour les vues matérialisées, il faut optimiser également le coût de maintenance afin que la mise à jour des structures soit moins couteuse en temps et en ressources. En analysant les travaux existant sur la sélection des vues matérialisées, l'auteur dans [68] déduit que trois variantes du problème de sélection existent :

- Minimiser le coût de la charge de requête sous une contrainte d'espace de stockage.
- Minimiser le coût des requêtes et le coût de maintenance des vues, en considérant une combinaison linéaire des deux coûts afin de sélectionner une solution qui optimise le coût total.
- Minimiser le coût de la charge sous une contrainte du coût de maintenance.

A cet effet, l'auteur dans [68] propose une démarche de sélection statique multi-objectifs des vues matérialisées basée sur les MOEAs. Cette démarche vise à résoudre le problème multi-objectifs formalisé comme suit. Soit un entrepôt de données modélisé en étoile. Le problème de sélection multi-objectifs consiste à sélectionner un ensemble de vues $\{v_1, \cdots, v_m\}$ qui minimisent deux fonctions objectifs : le coût de la charge de requêtes et le coût de maintenance des vues, sous une contrainte d'espace de stockage maximum réservé aux vues.

La matérialisation des vues emploi le principe de treillis. Les nœuds du treillis représentent les vues et un arc de V_i à V_j existe si V_j peut être évaluée en utilisant uniquement V_i. V_i est appelée

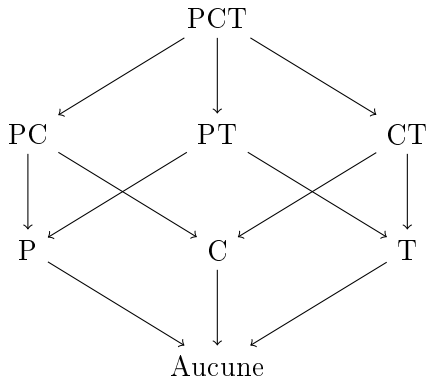


FIGURE 2.8 – Exemple d'un treillis de vues candidates à la matérialisation

alors ancêtre et V_j un descendant. Le treillis permet alors de définir une dépendance entre les vues mais aussi un ordre de matérialisation des vues. Si une vue ancêtre est matérialisée, elle peut être utilisée pour évaluer ses vues descendantes. Nous présentons dans la figure 2.8 un exemple d'un treillis construit sur trois tables dimensions Produits P, Clients C et Temps T. Les vue candidates à la matérialisation sont PCT, PT, PC, CT, P, C, T, Aucune. Pour évaluer le coût d'exécution des requêtes en présence d'un ensemble M de vues matérialisées, on considère pour chaque vue $v \in M$ sa fréquence d'utilisation par les requêtes f_i. Le coût d'exécution d'une charge de requêtes Q en utilisant v avec M requêtes matérialisées, noté $q(v, M)$, est égale au nombre de lignes de la plus petite vue dans M qui est l'ancêtre de v dans le treillis. Le coût total d'exécution de la charge de requêtes Q en présence de toutes les vues matérialisées M, noté $Q(M)$ est donné comme suit :

$$Q(M) = \sum_{v \in M} f_v q(v, M) \tag{2.21}$$

Le coût de maintenance $m(v, M)$ d'une vue matérialisée $v \in M$ est modélisé sur la base d'un coût qui est associé à chaque arête (v_i, v_j) dans le treillis, ce qui représente le coût de maintenance de v_j en utilisant les mises à jour à partir de v_i. Le coût de maintenance $m(v, M)$ de v est la somme des coûts sur le chemin le moins coûteux depuis un ancêtre de v. De plus, chaque vue v a une fréquence de mise à jour notée g_v. Ainsi, le coût total de mise à jour d'un ensemble de vues M est :

$$U(M) = \sum_{v \in M} g_v m(v, M) \tag{2.22}$$

L'optimisation prend en considération la contrainte sur l'espace de stockage des vues matérialisées tel que $S(M) \leq S_{max}$. Pour résumer, l'approche proposée vise à optimiser le PMO suivant :

$$\begin{cases} \text{Maximiser} & Q(M) \\ \text{Maximiser} & U(M) \\ \text{avec} & S(M) \leq S_{max} \end{cases} \tag{2.23}$$

Afin de résoudre le PMO, l'auteur emploi deux méthodes MOEAs à savoir les méthodes MOGA et NPGA considérées comme des méthodes non-élitiste (aucune sauvegarde des individus d'anciennes génération n'est effectuée). Les deux MOEAs exploitent une structure du chromosome qui permet de coder une configuration de vues candidates à la matérialisation et deux fonctions objectifs définis sur le coût d'exécution des requêtes et le coût de maintenances des vues.

2.4.4 Bilan

A notre connaissance, seuls les travaux cités dans [68] traitent de l'optimisation multi-objectifs dans le cadre des $\mathcal{ED}$. Ils proposent une sélection multi-objectifs des techniques d'optimisation à savoir les vues matérialisées, où deux objectifs sont considérés : le coût d'exécution des requête et le cout de maintenance des vues matérialisées. De plus, *aucun travail* n'a été proposée pour l'optimisation multi-objectifs des index et/ou de la fragmentation.

Cependant, dans le contexte de sélection incrémentale, le coût de maintenance des structures d'optimisation est un objectif qu'il faut optimiser. Pour les index, lors de l'évolution de l'entrepôt de données, il faut penser à choisir une nouvelle configuration d'index qui optimise la nouvelle charge requête mais qui optimise aussi le coût de maintenance qui représente le temps d'implémentation des nouveaux index et le temps de suppression des index obsolètes. Pour la fragmentation horizontale, la refragmentation d'un entrepôt déjà partitionné nécessite plusieurs opérations de fusion et/ou éclatement et/ou déplacement des partitions des différentes tables ce qui engendre un temps de maintenance considérable si la refragmentation nécessite beaucoup d'opérations. De ce fait, il faut proposer une nouvelle formulation du problème de sélection incrémentale des structure d'optimisation en un problème multi-objectifs où il faut optimiser deux objectifs : le coût d'exécution de la charge de requête et le coût de maintenance des structures implémentées.

2.5 Conclusion

Dans ce chapitre nous sommes intéressés aux problèmes d'optimisation multi-objectifs PMO. Nous avons présenté la formulation et les définitions mathématiques concernant ce problème. Ensuite, nous avons exposé un état de l'art sur les démarches de résolution du PMO. Nous avons remarqué que dans le contexte d'entrepôt de données, il existe quelques travaux qui formalisent un problème donné sous forme d'un PMO et cela dans le contexte du Cloud Computing et des entrepôts temps réel. Ces travaux visent à optimiser plusieurs objectifs critiques sous des contraintes données. Concernant la conception physique des entrepôts de données, seul le travail présenté dans [68] propose de formaliser le problème de sélection des vues matérialisées en un problème multi-objectifs où il faut optimiser deux objectifs : le coût d'exécution des requêtes et le coût de maintenance des vues. Dans la suite de notre thèse, nous décrivons trois propositions que nous avons effectuée : la sélection isolée incrémentale des techniques d'optimisation, la sélection jointe incrémentale des techniques d'optimisation et enfin la sélection incrémentale multi-objectifs des techniques d'optimisation. Les techniques d'optimisation abordées sont les index de jointures binaires et la fragmentation horizontale.

Deuxième partie

Contributions

Chapitre 1

Sélection incrémentale isolée des IJB

Sommaire

1.1 Introduction

Les *Index de Jointures Binaires* ($\mathcal{IJB}$) [77] permettant d'optimiser à la fois les jointures en étoile et les opérations de sélections définies sur les tables de dimensions, ce qui fait qu'ils sont bien adaptés pour optimiser les requêtes de jointure en étoile [89]. Vu le bénéfice que les index de jointures binaires apportent aux requêtes décisionnelles, ils sont très souvent employés lors de la phase de conception physique d'entrepôts de données. Par conséquent, plusieurs travaux s'intéressent à développer des processus de sélections de ce type d'index. Dans l'industrie, seul le SGBD Oracle implémente les index de jointure binaires.

Le problème de sélection des $\mathcal{IJB}$ dans sa formalisation classique consiste à sélectionner une configuration d'index optimisant une charge de requêtes, préalablement connue, sans violer la contrainte d'espace de stockage. Dans le contexte d'entrepôt de données, vu le nombre important de tables de dimension et d'attributs de sélection concernés par la définition des index, le problème de sélection des $\mathcal{IJB}$ devient difficile, ainsi plusieurs travaux se sont concentrés sur ce problème. Ces travaux sont réalisés en deux phases importantes : une phase de sélection de la configuration initiale qui inclut une étape d'élagage de l'espace de recherche pour la sélection des index multiples afin de réduire la complexité du problème, et une phase de sélection de la configuration finale d'index.

Notre analyse nous a conduit aux constats suivants : (1) La sélection des index multiples est un enjeu important. D'abords, ces index pré-calculent les opérations complexes de jointures en étoiles. De plus, l'espace de recherche que les algorithmes doivent parcourir est très complexe. (2) Peu de travaux emploient les requêtes pour l'élagage de cet espace de recherche. (3) Peu d'algorithmes sont utilisés pour la sélection d'une configuration finale d'index. (4) Les travaux de sélection se situent, pour la plupart, dans le cadre de sélection statique d'index. De ce fait, nous avons fait les propositions suivantes :

- proposer un nouvel algorithme de sélection des index basé sur les algorithmes génétiques [18, 20].
- proposer un modèle de coût pour les index de jointures binaires compressés.
- proposer une nouvelle approche d'élagage de l'espace de recherche des index afin de réduire la complexité du problème de sélection [25, 21, 20].
- proposer une sélection incrémentale des index de jointure binaires qui permet de faire face à l'évolution de l'entrepôt de données et aux changements pouvant survenir [18, 25, 21, 20].

Ce chapitre est organisé en 5 sections. La section 2 décrit notre proposition d'un algorithme de sélection statique des $\mathcal{IJB}$ basé sur les algorithmes génétiques et la description d'une nouvelle démarche d'élagage de l'espace de recherche. Nous exposons dans la section 3 la démarche de sélection incrémentale des $\mathcal{IJB}$. La section 4 représente l'étude expérimentale qui nous à permis de comparer les différentes stratégies de sélection que nous avons développé avec les travaux existants. La section 5 conclu le chapitre.

1.2 Algorithme génétique pour la sélection statique des $\mathcal{IJB}$

Dans cette section, nous présentons notre stratégie de sélection statique d'une configuration d'$\mathcal{IJB}$ simples ou multiples qui repose sur les algorithmes génétiques. Les AG ont été largement utilisés pour la conception physique des bases de données, notamment dans le problème d'optimisation des requêtes de jointure [61], le problème de sélection des vues matérialisées [98] et l'automatisation de la conception physique des bases de données parallèles [82]. Ils ont été utilisés dans l'optimisation de l'opération de jointure [61] et également employé par les Optimiseurs de SGBD comme PostgreSQL.

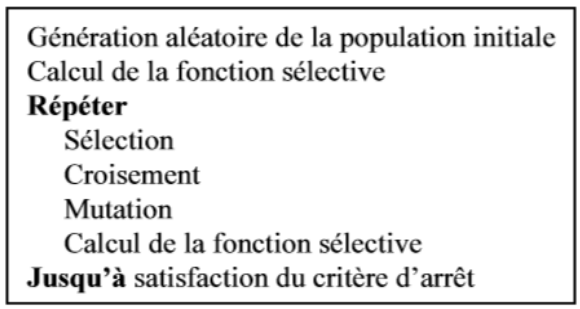


FIGURE 1.1 – La structure d'un AG

Les AG sont des méthodes d'optimisation de fonctions [57]. Ils s'inspirent de l'évolution génétique des espèces. Schématiquement, ils copient de façon extrêmement simplifiée certains comportements des populations naturelles. Ainsi, ces techniques reposent toutes sur l'évolution d'une population de solutions qui, sous l'action de règles précises, optimisent un comportement donné, exprimé sous une fonction dite fonction de sélection ou d'adaptation à l'environnement. Un AG est un algorithme itératif de recherche d'optimum, il manipule une population de taille constante. Cette dernière est formée de candidats appelés individus ou chromosomes. La taille constante de la population entraîne un phénomène de compétition entre les chromosomes. Chaque chromosome représente le codage d'une solution potentielle au problème à résoudre. Il est constitué d'un ensemble d'éléments appelés gènes, pouvant prendre plusieurs valeurs appartenant à un alphabet non forcément numérique [4]. Le but de l'algorithme génétique est de trouver l'individu qui optimise le mieux une fonction objectif donnée appelée aussi fonction sélective.

A chaque itération, appelée génération, est créée une nouvelle population de chromosomes. Cette génération est constituée des chromosomes les mieux "adaptés" à leur environnement tel qu'il est représenté par la fonction sélective. Au fur et à mesure des générations, les chromosomes vont tendre vers l'optimum de la fonction sélective. La création d'une nouvelle population, à partir de la précédente, se fait par application des opérateurs génétiques que sont : la sélection, le croisement et la mutation. Ces opérateurs sont stochastiques. La sélection des meilleurs chromosomes est la première opération dans un AG. Au cours de cette opération, l'AG sélectionne les éléments pertinents qui optimisent le mieux la fonction sélective. Le croisement permet de générer deux individus nouveaux "enfants" à partir de deux individus sélectionnés "parents", tandis que la mutation réalise l'inversion d'un ou plusieurs gènes d'un individu [4]. La figure 1.1 illustre les différentes opérations qui interviennent dans un AG.

Dans le cadre de la résolution du problème de sélection des index de jointures binaires (problème NP-Complet), l'algorithme de résolution doit exploiter un espace de solutions possibles afin de minimiser une fonction objectif sous une contrainte d'espace de stockage. L'espace de solutions possibles représente toutes les configurations d'index possibles où chaque index est défini sur un ou plusieurs attributs indexables. Dans notre cas, l'algorithme de résolution est un AG. Le chromosome représente une configuration d'index possible et la fonction objectif est le coût d'exécution des requêtes qui permet d'estimer la qualité de chaque chromosome. Afin d'évaluer le coût d'exécution des requêtes, nous utilisant un modèle de coût mathématique qui permet de guider l'AG pour trouver la solution optimale (quasi-optimale).

Dans ce qui suit, nous commençons par exposer le codage d'un chromosome, le modèle de coût,

la fonction objectif à optimiser et l'implémentation du AG.

1.2.1 Codage du chromosome pour les index

Afin d'exploiter l'algorithme génétique pour résoudre un problème d'optimisation, il faut définir la structure du chromosome qui représente le codage d'une solution potentielle au problème traité. Nous avons donc défini deux codages ; un codage pour les index simples et un codage pour les index multiples.

1.2.1.1 Codage du chromosome pour les index simples

Ce codage est appelé SI pour *Simple Indexes*. Soit $AS = \{A_1, \cdots, A_n\}$ l'ensemble des attributs indexables. Un index simple est défini sur un seul attribut indexable. Nous proposons de coder le chromosome sous forme d'un tableau de taille n dont les cellules sont binaires, où chaque cellule fait référence à un attribut. Une cellule vaut 1 si un index simple est défini sur l'attribut correspondant. Sinon, elle vaut 0. Le chromosome permet ainsi de représenter une configuration d'index. L'espace de recherche d'index, que ce codage permet de représenté, est constitué des différents chromosomes pouvant être générés. Pour illustrer ce codage, considérons l'ensemble d'attributs indexables suivant : $AS = \{City, Month, Year, Country, Day\}$. Le tableau 1.1 décrit un exemple d'un chromosome dit simple défini sur 5 attributs. Ce chromosome illustre une configuration de trois index simples : $Config_{Ci} = \{I\_City, I\_Country, I\_Day\}$

I_City	I_Month	I_Year	I_Country	I_Day
1	0	0	1	1

TABLE 1.1 – Exemple d'un chromosome pour la sélection d'$\mathcal{IJB}$ simples (SI)

1.2.1.2 Codage du chromosome pour les index multiples

La sélection d'une configuration d'index multiples exploite un espace de recherche très complexe. En effet, pour n attributs indexables, le nombre d'index multiples possibles $Nb_{IM} = 2^n - n - 1$ et la taille de l'espace de recherche est $NbConfig_{IM} = 2^{Nb_{IM}} - 1 = 2^{2^n - n - 1} - 1$ configurations.

Pour la sélection des index multiples, nous avons d'abord pensé à une **solution naïve** qui consiste à généraliser le codage utilisé dans la sélection simple, où un tableau de $2^n - n - 1$ cellules (gènes) est construit. Les gènes du chromosome correspondent chacun à un index multiple et non plus à un attribut indexable (comme pour le codage de SI). Nous avons rapidement réalisé que ce codage est complexe ; puisque la taille du chromosome est très grande, l'espace de recherche devient très complexe. En effet, pour 10 attributs indexables, le codage naïf de chromosome défini sur ces attributs donne lieu à la création d'un tableau binaire de $2^{11} - 11 - 1 = 2036$ cases et la taille de l'espace de recherche est de $2^{2036} - 1$ soit 10^{613} configurations possibles. Nous illustrons à travers le tableau 1.2 le codage naïf du chromosome pour quatre attributs indexables A1, A2, A3 et A4. Le chromosome est de taille $2^4 - 4 - 1 = 11$ cases et l'espace de recherche est de taille $2^{11} - 1 = 2047$.

Vu cette complexité, la simplification du codage naïf est recommandée. Pour ce faire, nous proposons d'utiliser les requêtes pour instancier le codage initial et nous définissons deux codages : codage basé sur les requêtes appelé MIQ et codage amélioré basé sur les requêtes $MIQ*$.

$I\_A_1A_2A_3A_4$	$I\_A_1A_2A_3$	$I\_A_1A_2A_4$	$I\_A_1A_3A_4$	$I\_A_2A_3A_4$	$I\_A_1A_2$	$I\_A_1A_3$	$I\_A_1A_4$	$I\_A_2A_3$	$I\_A_2A_4$	$I\_A_3A_4$
0	1	0	0	1	0	1	1	0	0	1

TABLE 1.2 – Exemple d'un chromosome basé sur le Codage Naïf

Codage basé sur les requêtes MIQ **:** Nous retenons comme index candidats à la sélection uniquement ceux qui couvrent exactement les attributs d'une requête donnée. Ceci réduit considérablement la taille du chromosome dont la taille maximale est égale au nombre de requêtes ; si chaque requête contient une combinaison d'attributs différente des autres requêtes. Nous appelons ce codage MIQ pour *Multiple Indexes Query.*

I_CYP	I_CMD	I_PM
0	1	1

TABLE 1.3 – Codage d'un chromosome basé requêtes MIQ

Exemple 11 *Soit un $\mathcal{ED}$ avec une table de faits Ventes (20 millions de tuples) et trois dimensions Clients, Temps et Produits. Soit une charge de trois requêtes permettant de définir les attributs avec les cardinalités suivantes : City(C :150), Country(T :30), Year(Y :20), PName(P :400), Month(M :12), Day(D :31).*

Q1 : SELECT AVG(PriUnit)
FROM Clients C, Temps T, Produits P, Ventes V
WHERE C.City='Alger' AND T.Year='2014' AND P.PName='PC'
AND C.CID=V.CID AND T.TID=V.TID AND P.PID=V.PID

Q2 : SELECT Count(*)
FROM Clients C, Temps T, Ventes V
WHERE C.City='Oran' AND T.Year='2014' AND T.Day='20'
AND C.CID=V.CID AND T.TID=V.TID

Q3 : SELECT Max(Sold)
FROM Produits P, Temps T, Ventes V
WHERE P.PName='PC' AND T.Month='4'
AND C.CID=V.CID AND T.TID=V.TID AND P.PID=V.PID

Les attributs candidats à l'indexation sont : $AS = \{City, Month, PName, Year, Day\}$ que nous simplifions comme suit : $AS = \{C, M, P, Y, D\}$. La taille du chromosome selon le codage naïf pour représenter cet ensemble d'attributs est $2^5 - 5 - 1 = 26$. Si nous considérons les requêtes pour instancier le chromosome initial, nous aurons un tableau de trois cellules représentant les trois $\mathcal{IJB}$ (tableau 1.3) : I_CYP, I_CYD et I_PM. En conséquence, au lieu de manipuler un tableau de 26 cellules, nous utilisons un tableau de taille 3 qui permet de représenter $2^3 - 1 = 7$ configurations d'index.

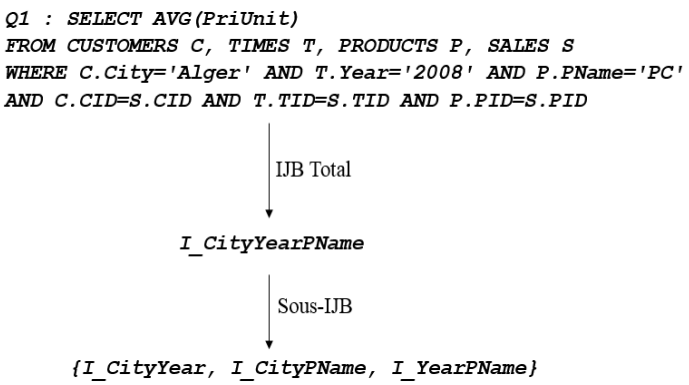


FIGURE 1.2 – Génération des sous $\mathcal{IJB}$ pour une requête

L'inconvénient de cette démarche de sélection guidée par les requêtes est le risque de générer des index volumineux qui pourraient violer l'espace de stockage. Pour illustrer ce problème, supposons que les trois index nécessitent les coûts de stockage suivants : $1.6Go$, $1Go$ et $1.2Go$ respectivement (sans compression). Si la contrainte d'espace de stockage des index est inférieure à $1Go$, aucun index n'est sélectionné. Par conséquence, aucune requête n'est optimisée.

Codage amélioré basé sur les requêtes $MIQ*$: Afin d'améliorer le codage précédent, nous proposons un autre codage appelé MIQ^* pour *Multiple Indexes Query Ameliorated*. Considérons le codage du chromosome MIQ. Le principe est d'ajouter à ce codage et pour chaque requête les sous index multiples pouvant être définis sur ses attributs. Pour une requête donnée, un sous index est défini sur une combinaison possible des attributs indexables qui la constitue. Nous illustrons sur la figure 1.2 un exemple d'une requête, l'index multiple total qui couvre tous ses attributs et les sous index multiples qui peuvent être générés. Le codage final est constitué des $\mathcal{IJB}$ issus du codage MIQ auxquels on ajoute tous les sous index générés. L'intérêt de cette approche est de palier au problème des $\mathcal{IJB}$ volumineux tout en respectant l'élagage par requêtes.

Exemple 12 *Considérons l'$\mathcal{ED}$ et les requêtes définies dans l'exemple 11. Rappelons que les attributs indexables sont $AS = \{C, M, P, Y, D\}$. Nous définissons pour chaque requête l'index total est les sous index comme suit :*

- *Pour Q1, L'index total est I_CYP les Sous Index sont I_CY, I_CP et I_YP.*
- *Pour Q2, L'index total est I_CYD les Sous Index sont I_CY, I_CD et I_YD.*
- *Pour Q3, L'index total est I_PM, aucun Sous Index multiple ne peut être défini.*

Une fois tous les sous index définis sur chaque requête, le chromosome est structuré comme le montre le tableau 1.4. Il est à noter que les doublons sont supprimés. Si la contrainte d'espace de stockage est 1Go, les deux requêtes Q1 et Q2 peuvent être optimisées par l'index I_CY commun aux deux, dont la taille est 710Mo.

I_CYP	I_CY	I_CP	I_YP	I_CYD	I_CD	I_YD	I_PM
0	1	1	0	0	1	0	0

TABLE 1.4 – Codage Amélioré d'un chromosome basé requêtes $MIQ*$

De ce fait, l'optimisation de la charge de requêtes est améliorée. Si l'$\mathcal{IJB}$ défini sur les attributs d'une requête est volumineux, il y a de fortes chances pour qu'un sous index moins volumineux soit pris par le processus de sélection d'index. De plus, à partir du chromosome du tableau 1.4, l'espace de recherche représente $2^8 - 1 = 255$ configurations d'index. Cette nouvelle représentation de l'espace de recherche sur les 5 attributs indexables est un bon compromis entre l'espace total d'index très complexe $2^{2^5-5-1} - 1 = 2^{11} - 1 = 2047$ configurations, et l'espace basé requête (MIQ tableau 1.3) de taille $2^3 - 1 = 7$ configurations, espace qui peut s'avérer non pertinent pour l'optimisation des requêtes.

1.2.2 Modèle de coût mathématique

Afin de guider l'algorithme de sélection pour trouver la solution optimale, plusieurs travaux de recherche ont proposé des modèles de coût pour estimer le coût d'exécution d'une requête. Ces modèles de coûts sont classés en deux catégories : (1) les modèles de coûts mathématiques qui se basent sur des estimations, des statistiques et des formules mathématiques et (2) les modèles de coût qui font appel à l'optimiseur du SGBD. Dans les travaux [26], l'auteur a montré que les modèles de coût mathématiques sont les plus adaptés. En effet, ces derniers se basent sur des fonctions de coût mathématiques faciles à implémenter et rapide en exécution, alors que ceux de la seconde catégorie rendent l'algorithme de sélection dépendant du SGBD, et engendre un temps d'exécution supplémentaire dû aux appels fréquents de l'optimiseur.

Par conséquent, nous utilisons un modèle de coût mathématique qui permet d'évaluer le coût d'exécution de la charge de requêtes, en termes du nombre d'Entrées/Sorties, en présence d'une configuration index. Nous utilisons le modèle de coût présenté dans [2] et défini sur les $\mathcal{IJB}$ simples. Nous avons apporté deux améliorations à ce modèle :

1. **Amélioration 1 :** adapter le modèle de coût pour les index multiples.
2. **Amélioration 2 :** proposer un modèle de coût pour les index compressés. En effet, le modèle de coût présenté dans [2] ne prend pas en compte la compression des index alors que le SGBD Oracle par exemple, stocke tous les index de jointure binaires avec compression ce qui réduit considérablement l'espace de stockage requis.

Le modèle de coût est exposé comme suit. Soit un entrepôt de données modélisé en étoile avec une table de faits F, d tables de dimensions $D_1, \cdots, D_d$ et n attributs dimensions $AS = \{A_1, \cdots, A_n\}$. Nous considérons une charge de p requête $\mathcal{Q} = \{Q1, \cdots, Q_p\}$ à exécuter sur l'entrepôt, l'espace de m configurations d'index possibles $ConfigI = \{Config_{c1}, \cdots, Config_{cm}\}$. Chaque configuration contient N_{ci} index défini sur AS. Afin d'évaluer la qualité d'une configuration d'index $Config_{ci}$, deux coûts sont utilisés : le coût de stockage des index et le coût de la charge de requêtes en présence de $Config_{ci}$. Nous présentons dans ce qui suit le modèle de coût pour les index simples, pour les index multiples et le modèle de coût avec compression.

1. *Modèle de coût pour les index simples :* Pour les index simples, nous employons le modèle de coût présenté dans [2]. L'espace de stockage pour un index simple IJB_j est calculé comme suit :

$$Storage(IJB_j) = (\frac{|A_j|}{8} + 8) \times |F| \tag{1.1}$$

   Où $|A_j|$ et $|F|$ représentent respectivement la cardinalité de l'attribut qui définit l'index IJB_j et la taille de la table de faits F. Le coût d'exécution d'une requête Q_i $(1 \leq i \leq q)$ en présence

de IJB_j est :

$$Cost(Q_i, IJB_j) = log_m(|A_j|) - 1 + \frac{|A_j|}{m-1} + d\frac{||F||}{8PS} + ||F||(1 - e^{-\frac{Nr}{||F||}}) \quad (1.2)$$

où $||F||$, Nr, PS, m et d sont resp. le nombre de pages occupées par la table F, le nombre de tuples accédés par IJB_j , la taille d'une page, l'ordre du B-arbre qui définit l'$\mathcal{IJB}$ et le nombre de vecteurs bitmaps utilisés pour évaluer Q_i.

2. *Modèle de coût pour les index multiples :* Nous avons adapté les formules du modèle de coût pour les index multiples. Soit un index multiple IJB_j défini sur n_j attributs indexables $\{A_1^j, \cdots, A_{n_j}^j\}$. L'espace de stockage requis pour IJB_j est calculé par la formule suivante :

$$Storage(IJB_j) = (\frac{\sum_{k=1}^{n_j} |A_k^j|}{8} + 8) \times |F| \quad (1.3)$$

Le coût d'exécution d'une requête Q_i $(1 \leq i \leq q)$ en présence de IJB_j est :

$$Cost(Q_i, IJB_j) = log_m(\sum_{k=1}^{n_j} |A_k^j|) - 1 + \frac{\sum_{k=1}^{n_j} |A_k^j|}{m-1} + d\frac{||F||}{8PS} + ||F||(1 - e^{-\frac{Nr}{||F||}}) \quad (1.4)$$

3. *Modèle de coût :* Le coût d'exécution de la requête Q_i en présence de la configuration d'index simples ou multiples $Config_{ci}$ est :

$$Cost(Q_i, Config_{ci}) = \sum_{j=1}^{N_{ci}} Cost(Q_i, IJB_j) \quad (1.5)$$

Le coût total d'exécution des q requêtes en présence de $Config_{ci}$ est :

$$Cost(Q, Config_{ci}) = \sum_{i=1}^{q} \sum_{j=1}^{N_{ci}} Cost(Q_i, IJB_j) \quad (1.6)$$

4. *Modèle de coût avec compression :* Dans la définition du modèle de coût avec compression, nous avons constaté que l'accès aux données reste le même, la formule $Cost(Q, Config_{ci})$ reste donc valable. En effet, la principale différence en l'accès aux données avec un index compressé et un index non compressé est le coût de décompression des vecteurs bitmaps de l'index utilisés pour identifier les tuples de la table de faits. Nous considérons négligeable le coût de décompression car c'est une opération qui se déroule en mémoire centrale sur des vecteurs binaires. Cependant, le principal changement est au niveau du calcul de l'espace de stockage $Storage(IJB_j)$. Nous commençons par présenter le principe d'un index simple compressé puis d'un index multiple compressé. Puis à la base de ce principe, nous donnons la nouvelle formulation du calcul de l'espace de stockage.

   Un $\mathcal{IJB}$ simple défini sur un attribut A est un index de vecteurs bitmaps dont le nombre de ligne correspond au nombre de tuples de la table de faits et où chaque ligne est un vecteur binaire dont la longueur correspond à la cardinalité de A. Cette ligne contient une seule case à 1 et toutes les autres cases sont à 0. On peut donc coder ce vecteur ligne en un nombre binaire correspondant à la position verticale de la case qui est à 1. Par exemple, si la cardinalité de A est 15, un vecteur ligne peut être (000000000001000). Les 15 positions possibles du 1 peuvent être

	IJBVille									
Rowid	Alger	Blida	Oran	Paris	Poitiers	Kala	Jijel	Rabat	Fès	Tunis
1	0	0	0	0	1	0	0	0	0	0
2	0	1	0	0	0	0	0	0	0	0
3	0	0	0	0	0	1	0	0	0	0
4	0	0	0	0	0	0	0	0	1	0
5	0	0	0	0	0	0	1	0	0	0
6	0	0	0	0	1	0	0	0	0	0
7	0	1	0	0	0	0	0	0	0	0
8	0	0	1	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	1	0
10	0	0	0	0	0	0	0	0	0	1

IJBVille (Compressé)			
0	1	1	0
1	0	0	1
0	1	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	0	0
0	0	1	0
0	0	0	1

FIGURE 1.3 – Exemple d'un index compressé

codées avec 4 bits de 1 à 5 (de 0001 à 1111). Le vecteur 000000000001000, dont la valeur 1 est dans la 4ème position, est codé donc en 0010. Ainsi, un $\mathcal{IJB}$ défini sur l'attribut de cardinalité 15 contiendra 4 vecteur bitmaps verticaux au lieu de 15. De manière générale, chaque ligne d'un $\mathcal{IJB}$ simple défini sur A peut être codée sur $log_2(|A|)$ bits, où $|A|$ représente la cardinalité de A.

En généralisant cette formule sur les index multiple, nous pouvons coder chaque ligne d'un $\mathcal{IJB}$ multiple, défini sur n_j attributs $\{A_1^j, \cdots, A_{n_j}^j\}$, sur $\sum_{k=1}^{n_j} Log_2(|A_k^j|)$ bits. A la base de cette compression, nous pouvons exprimer la formule de calcul de l'espace de stockage requis pour l'$\mathcal{IJB}$ multiple comme suit :

$$Storage(IJB_j) = (\frac{\sum_{k=1}^{n_j} Log_2(|A_k^j|)}{8} + 8) \times |F| \tag{1.7}$$

Exemple 13 *Soit l'entrepôt présenté dans l'exemple 11. Rappelons que la taille de la table de faits est de 20 millions de tuples. Supposant un $\mathcal{IJB}$ simple défini sur un attribut Ville de cardinalité 10 illustré sur la figure 1.3 gauche. L'index compressé est représenté sur la figure 1.3 droite. Nous calculons l'espace de stockage occupé par chaque index. L'index sans compression donne 176Mo alors que l'index avec compression donne 162Mo. Pour un attribut de cardinalité 300, la compression se fait sur $Log_2(300)$=7 bits ce qui donne un index sans compression de 390Mo contre un index avec compression de taille 171Mo, ce qui réduit de 55% l'espace de stockage.*

1.2.3 Fonction objectif

Le problème de sélection d'index est un problème d'optimisation mono-objectif où il faut optimiser une fonction objectif sous une contrainte d'espace pour le stockage des index. Cette fonction objectif représente le coût d'exécution de la charge de requêtes. Elle est utilisée par l'algorithme génétique pour évaluer chaque chromosome (solution) en cour d'exploitation. En d'autre terme, elle évalue le coût d'exécution de la charge de requêtes en présence des index que le chromosome permet de définir. Ainsi, en utilisant le modèle de coût que nous venons d'exposer, le problème de sélection d'index peut être formalisé comme suit :

$$\begin{cases} \text{Minimiser } Cost(Q, Config_{ci}) \\ \text{avec } Storage(Config_{ci}) \leq S \end{cases} \tag{1.8}$$

Le problème mono-objectif avec contrainte est transformé en un problème mono-objectif sans contrainte comme suit : Soit F(x) une fonction objectif. On s'intéresse à minimiser F(x) sous la contrainte C(x) ce qui représente un problème de minimisation avec contrainte. Soit Pen(x) une fonction pénalité qui pénalise une solution x qui viole la contrainte C. L'utilisation de la fonction de pénalité pour la fonction F(x) permet de transformer le problème en un problème d'optimisation sans contraintes d'une fonction F'(x) définie à partir des fonctions F(x) et Pen(x). La formulation générale d'une fonction objectif est donnée comme suit [2] :

$$F'(x) = \begin{cases} F(x) \times Pen(x) & \text{si Pen(x)>1} \\ F(x) & \text{sinon} \end{cases}$$

En faisant l'analogie avec notre problème de sélection d'index, x représente la configuration d'index $Config_{ci}$, $F(x)$ est le coût de la charge de requêtes en présence des index $Cost(Q, Config_{ci})$ et C représente la contrainte d'espace de stockage des index S. La fonction de pénalité est calculée par la formule suivante :

$$Pen(ConfigI_{ci}) = \frac{Storage(Config_{ci})}{S} \tag{1.9}$$

où $Storage(Config_{ci}) = \sum_{j=1}^{N_{ci}} Storage(IJB_j)$.

Enfin, la fonction objectif est définie comme suit :

$$F'(Config_{ci}) = \begin{cases} Cost(Q, Config_{ci}) \times Pen(Config_{ci}) & \text{si } Pen(Config_{ci})>1 \\ Cost(Q, Config_{ci}) & \text{sinon} \end{cases}$$

1.2.4 Implémentation du AG

Afin d'exploiter l'algorithme génétique, nous avons utilisé une API JAVA nommée JGAP $JGAP$[5] (Java Genetic Algorithms Package). JGAP est un framework Java qui permet de programmer un algorithme génétique. Il implémente les mécanismes génétiques de base qui peuvent être facilement utilisés pour appliquer les principes d'évolution aux solutions d'un problème donnée (mutation, croisement, sélection). JGAP a été conçu pour être très facile à utiliser, tout en étant très modulaire, ce qui permet facilement d'ajouter ou de changer des opérateurs génétiques et autres sous-composants. La figure 1.4 présente le diagramme des classes de l'API. Afin d'exploiter JGAP pour résoudre notre problème, il faut réécrire les deux classes Chromosome et FitnessFunction. La classe Chromosome permet d'implémenter le codage du chromosome selon les index à sélectionner (SI, MIQ ou $MIQ*$). Puisque le chromosome est un tableau binaire, la classe IntegerGene est utilisée pour implémenter les gènes du chromosome. La classe FitnessFunction est réécrite pour implémenter la fonction objectif définie précédemment.

La figure 1.5 illustre notre architecture de sélection statique des $\mathcal{IJB}$ par algorithmes génétiques. Indépendamment du type d'index (simple ou multiple), le déroulement du processus de sélection, décrit dans l'algorithme 1, est le suivant :

1. Extraire les attributs indexables à partir de la charge de requêtes.
2. Coder la configuration d'index en chromosome suivant le codage SI, MIQ ou $MIQ*$.
3. Définir la fonction objectif.

5. http ://jgap.sourceforge.ne

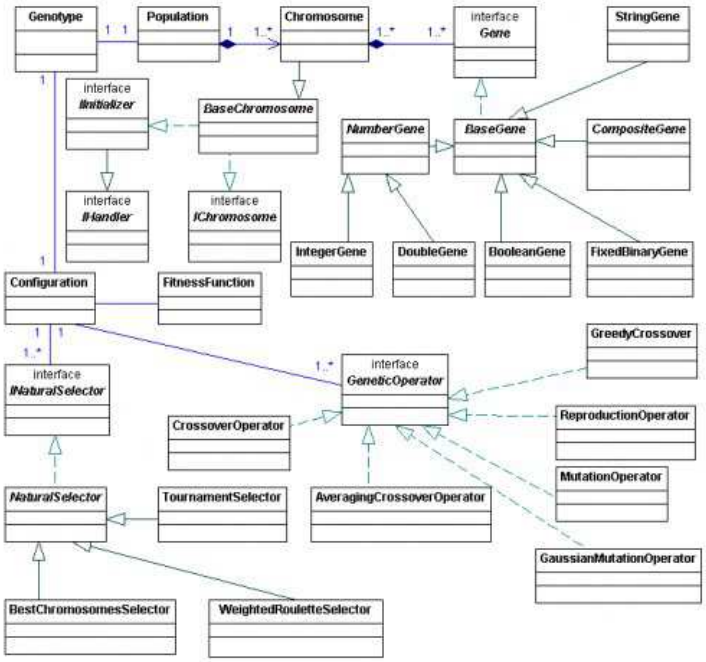


FIGURE 1.4 – Diagramme de classe de l'API JGAP

4. Sélectionner la configuration d'index finale : L'algorithme génétique génère une population initiale qui représente plusieurs chromosomes. A partir de cette population, l'algorithme génétique effectue les opérations génétiques (croisement, mutation et sélection) afin de générer les nouvelles populations. Chaque configuration d'index (chromosome) va être évaluée, par la fonction objectif, afin d'estimer le bénéfice apporté par celle-ci pour l'optimisation de la charge de requêtes. La configuration d'index qui réduit le plus le coût d'exécution de cette charge va être sélectionnée en fin de processus.

1.3 Sélection Incrémentale des $\mathcal{IJB}$ basée sur les AGs

La majorité des travaux qui traitent du problème de sélection d'index se basent sur une sélection statique qui suppose une connaissance préalable de la charge de requêtes obtenue à partir du journal du SGBD. La sélection d'index est réalisée une seule fois lors de la phase de conception physique d'un $\mathcal{ED}$. Par conséquent, elle ne permet pas de faire face aux changements pouvant survenir sur l'entrepôt. Nous présentons dans ce qui suit ces types de changements, puis nous exposons notre approche de sélection incrémentale.

1.3.1 Types d'évolutions d'un entrepôt de données

Un entrepôt de données est en évolution continuelle qui peut se présenter sous trois différents façon :

– *Évolution de schéma* : signifie que le schéma de définition des tables change par l'ajout ou la suppression d'attributs, l'ajout ou la suppression de tables. Dans le contexte d'entreposage de

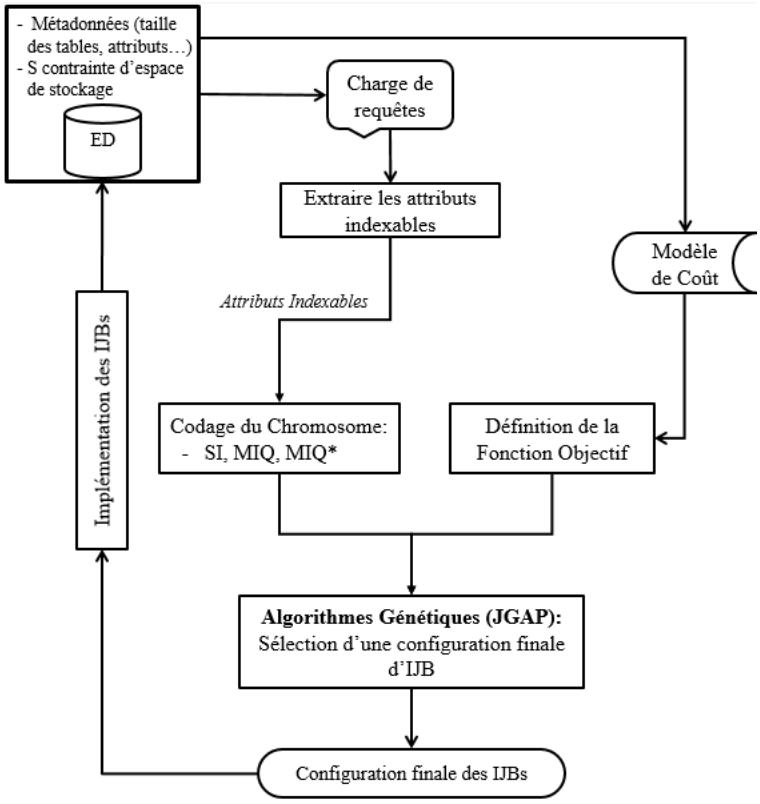


FIGURE 1.5 – Architecture de sélection statique des $\mathcal{IJB}$ par AG

données ce scénario est peut réalisable car le schéma de tables reste le même.

- *Évolution d'instances* : matérialisé par l'insertion continuelle de données au niveau des tables particulièrement au niveau de la table des faits.
- *Évolution de charge* : représente les changements pouvant survenir sur la charge de requêtes décisionnelles par l'ajout ou la suppression de requêtes ou le changement de la fréquence d'accès des requêtes.

L'évolution de la charge de requêtes influe sur la définition de l'ensemble des attributs indexables qui sont utilisés pour implémenter les index. En effet, toutes les démarches de sélection d'index se basent sur l'extraction de cet ensemble à partir de la clause WHERE des requêtes. Donc, de nouveaux attributs peuvent être ajoutés, d'autres attributs peuvent ne plus figurer dans la description des requêtes ou leurs fréquences d'utilisation peuvent augmenter ou diminuer. Par conséquent, les index déjà implémentés deviendront obsolètes et ne répondront plus à l'optimisation de la nouvelle charge de requêtes. Concernant l'évolution d'instances, elle cause une augmentation de la taille des tables particulièrement la table de faits, ce qui augmente la taille des $\mathcal{IJB}$. Dans ce cas, la contrainte d'espace de stockage risque d'être violée si aucun nouvel espace de stockage suffisant n'est alloué pour les index. Afin de répondre à cette problématique, nous avons proposé une nouvelle sélection incrémentale d'index qui vise à garantir une optimisation continuelle de la charge de requêtes par la mise à jour des index implémentés sur l'entrepôt, ce qui permet de faire face à l'évolution de charge et à l'évolution d'instances.

Algorithme de sélection d'$\mathcal{IJB}$

Entrées :

Q : charge de m requêtes.
S : espace de stockage des $\mathcal{IJB}$s.
AS : ensemble d'attributs d'indexation (1, n).
$\mathcal{ED}$: données relatives au modèle de coût (taille des tables, page système, etc.) .

Sortie : Configuration finale d'index C_f.

Notations :

$ChromosomeIJB$: chromosome représentant configuration d'index candidats.
$FonctionIJB$: fonction objectif pour l'AG.
CoderChromosome : Coder le chromosome selon la démarche suivie (SI, MIQ, MIQ^*).
JGAP : API JAVA qui permet d'implémenter l'algorithme génétique.

Début

$ChromosomeIJB \longleftarrow$ CoderChromosome(Q, AS, $SI/MIQ/MIQ^*$) ;
$FonctionIJB \longleftarrow$ CalculFonctionObjectif(AS, S, $\mathcal{ED}$, Q) ;
$C_f \longleftarrow$ JGAP($ChromosomeIJB$, $FonctionIJB$) ;

Fin

Algorithme 1: Algorithme de sélection statique des Index de Jointure Binaires

1.3.2 Notre approche de résolution

Soit un entrepôt de données modélisé en étoile. Lors de la phase de conception physique de l'$\mathcal{ED}$, un ensemble d'$\mathcal{IJB}$ a été sélectionné et implémenté sur l'entrepôt pour optimiser une charge de requêtes $\mathcal{Q}$. Supposons que la charge de requêtes évolue par l'exécution successive et continuelle de nouvelles requêtes. L'arrivée de chaque nouvelle requête Q_i déclenche le processus de sélection incrémentale. Dans un premier lieu, nous avons pensé à refaire tous le processus de sélection d'index défini dans la sélection statique. Une nouvelle sélection d'index par algorithmes génétiques est exécutée sur la nouvelle charge de requêtes $\mathcal{Q} \cup Q_i$. Cette démarche présente deux inconvénients :

- Cette approche ne prend pas en compte la configuration d'index actuellement implémentée sur l'entrepôt. En effet, une approche de sélection est dite incrémentale si elle prend en compte la configuration d'index actuelle et tente de l'améliorer afin que la nouvelle configuration optimise la nouvelle charge de requêtes $\mathcal{Q} \cup Q_i$.
- L'espace de recherche d'index défini sur la charge $\mathcal{Q} \cup Q_i$ est très complexe.
- La sélection d'index peut aboutir à une configuration finale d'index différente de celle déjà implémentée sur l'entrepôt. Par conséquent, l'implémentation de la nouvelle configuration d'index peut être couteuse en termes de temps et de ressources.

Par conséquent, nous proposons d'extraire la configuration d'index actuellement implémentée, d'extraire les index défini sur Q_i puis d'effectuer une nouvelle sélection d'index sur l'union des deux ensembles. Cela réduit considérablement l'espace de recherche des index et le temps d'implémentation. Nous proposons une démarche de sélection incrémentale d'index dont l'architecture globale est illustrée sur la figure 1.6. Les étapes de la sélection sont données dans ce qui suit.

1. Extraire la configuration d'$\mathcal{IJB}$ actuellement implémentée sur l'entrepôt $Config\_IJB$. La liste des index et les attributs sur lesquels chaque index a été construit peuvent être obtenus à partir des méta-données de l'entrepôt. Dans le SGBD Oracle, deux tables permettent d'extraire la description d'un index donné. La table **User_Indexes** contient les index existants et la

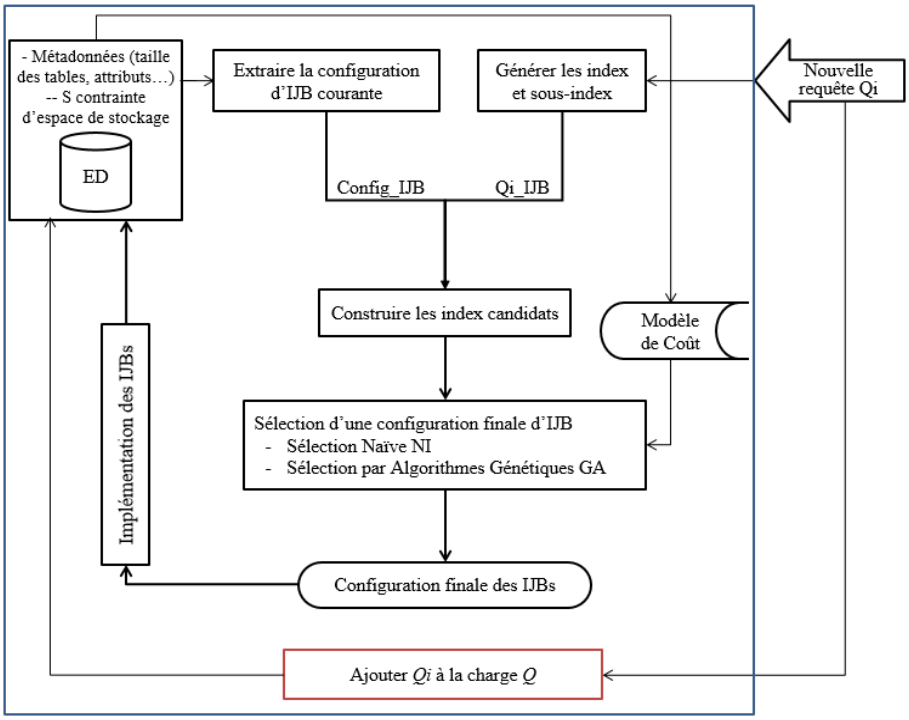


FIGURE 1.6 – Architecture globale de sélection incrémentale des IJB

table **User_Ind_Columns** contient les attributs sur lesquels est défini chaque index. Afin d'extraire les index implémentés, nous exécutons la requête SQL suivante :

```
SELECT Index_Name
FROM User_Indexes
WHERE Index_Type = 'BITMAP'
  AND Join_Index='YES';
```

Pour chaque index obtenu IJB_i, nous exécutons la requête SQL suivante afin d'extraire les attributs sur lesquels il est défini.

```
SELECT Column_Name
FROM User_Ind_Columns
WHERE Index_Name = 'IJBi';
```

L'algorithme 2 résume les étapes d'extraction des index actuellement implémentés sur l'entrepôt.

2. Générer l'ensemble Qi_IJB d'$\mathcal{IJB}$ multiples pouvant être définis sur la requête Q_i selon le codage MIQ^*, à savoir l'$\mathcal{IJB}$ couvrant tous les attributs de la requête et les sous index pouvant être générés sur ces attributs. Pour une requête contenant les attributs A_1, A_2 et A_3 par exemple, l'ensemble Qi_IJB=$\{I\_A_1A_2A_3, I\_A_1A_2, I\_A_1A_3, I\_A_2A_3\}$.
3. A partir des deux précédentes étapes, former une configuration initiale d'index $Config\_IJB \cup Qi\_IJB$.
4. Sur la configuration initiale précédemment obtenue, sélectionner une configuration finale d'index en se basant sur le modèle de coût mathématique.

Algorithme d'extraction de ConfigIJB
Sortie : ConfigIJB : configuration d'index actuellement implémentée sur l'$\mathcal{ED}$.
Notation
getIJBName() : Exécute la requête SELECT Index_Name
FROM User_Indexes
WHERE Index_Type = 'BITMAP'
AND Join_Index='YES';

getIJBAttribut(IJB$_i$) : Exécute la requête SELECT Column_Name
FROM User_Ind_Columns
WHERE Index_Name = IJB$_i$;

Début
$EnsIJB \longleftarrow$ getIJBName();
Pour IJB$_i$ dans $EnsIJB$ **faire**
AttIJB$_i \longleftarrow$ getIJBAttribut(IJB$_i$);
ConfigIJB $\cup \longleftarrow$ {AttIJB$_i$};
Fin Pour
Fin

Algorithme 2: Algorithme qui extrait la configuration d'index actuelle

5. Implémenter la configuration finale d'index C_f. Afin de réduire le temps d'implémentation de C_f, nous implémentons uniquement les nouveaux index ne figurant pas dans l'entrepôt et supprimons de l'entrepôt les index obsolètes non sélectionnés par le processus de sélection. Les nouveaux index sont ceux contenus dans l'ensemble $C_f \setminus Config\_IJB$ et les index obsolètes sont contenus dans l'ensemble $Config\_IJB \setminus C_f$. Dans le SGBD Oracle, l'implémentation des nouveaux index est réalisée par la syntaxe SQL CREATE BITMAP INDEX et la suppression est effectuée par la syntaxe DROP INDEX. La syntaxe SQL pour créer et supprimer un index IJB_A1A2A3 défini sur trois attributs A1, A2 et A3 est donnée comme suit :

```
CREATE BITMAP INDEX IJB_A1A2A3
ON Facts(Dimension1.A1, Dimension2.A2, Dimension2.A3)
FROM Facts F, Dimension1 D1, Dimension2 D2
WHERE F.D1_ID=D1.D1_ID AND F.D2_ID=D2.D2_ID;

DROP INDEX IJB_A1A2A3;
```

Afin de réaliser la sélection incrémentale de la configuration d'index finale, nous avons d'abords proposé une sélection dite *Sélection Incrémentale Naïve NI*, ensuite nous avons voulu adopté notre sélection statique par algorithme génétique pour réaliser une sélection incrémentale que nous appelons GA. Les deux démarches sont détaillées dans ce qui suit.

1.3.2.1 Sélection Incrémentale Naïve NI

Une sélection incrémentale Naïve (NI) démarre d'une configuration d'index initiale $Config\_IJB$. A l'arrivée de chaque nouvelle requête Q_i, nous effectuons les actions suivantes (figure 1.7) :

1. Extraire la configuration courante d'$\mathcal{IJB}$ $Config\_IJB$.
2. Générer l'ensemble Qi_IJB d'$\mathcal{IJB}$ multiples à partir de Q_i.

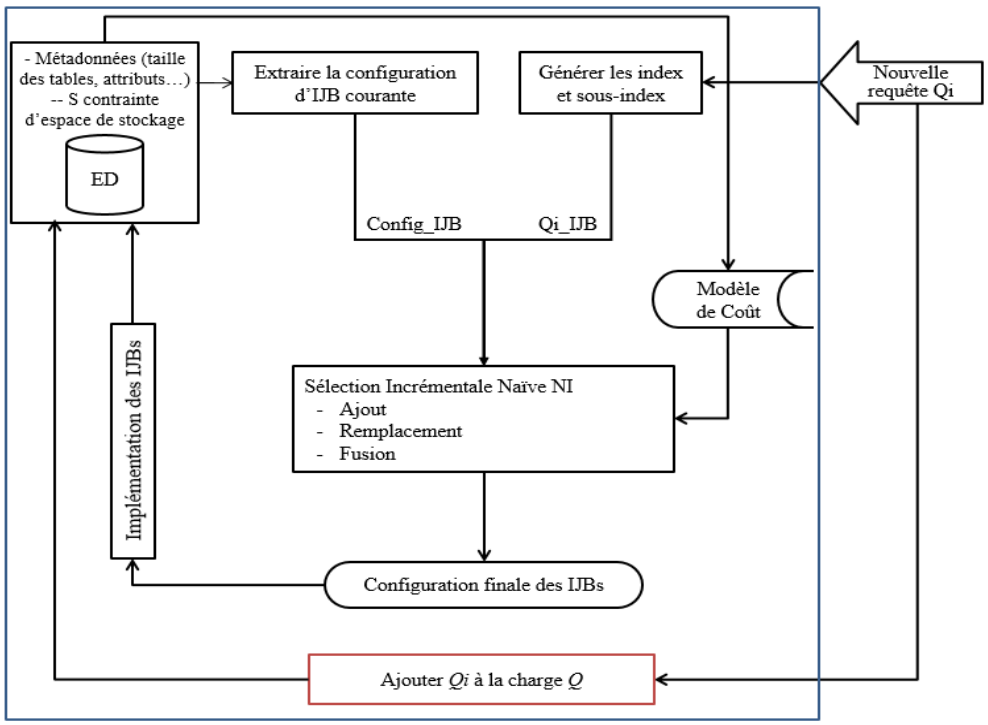


FIGURE 1.7 – Architecture de sélection Incrémentale des IJB Naive NI

3. Ordonner les index de Qi_IJB du plus bénéfique au moins bénéfique à la charge de requêtes $\mathcal{Q} \cup \{Q_i\}$, en se basant sur le modèle de coût.
4. Tant que l'espace de stockage d'index S n'est pas saturé, ajouter à $Config\_IJB$ un sous index de Q_i appelé Qi_IJB_j.
5. Tant que l'espace S est saturé, effectuer une fusion des index de $Config\_IJB$. La fusion est réalisée par la recherche dans $Config\_IJB$ des index qui forment une fusion d'un index de Q_i. Par exemple, la fusion des deux index I_CY et I_YD donne l'$\mathcal{IJB}$ I_CYD.
6. Si aucune fusion n'est possible, remplacer dans $Config\_IJB$ l'index le moins bénéfique pour la charge de requête, par l'index le plus bénéfique de Qi_IJB.

1.3.2.2 Sélection Incrémentale par Algorithmes Génétiques GA

Nous avons voulu adapté notre sélection des $\mathcal{IJB}$ multiples par algorithmes génétiques, présentée dans la section 1.2, dans le cadre d'une sélection incrémentale (GA). La sélection par algorithme génétique adapte le chromosome afin de prendre en compte les nouveaux index générés par la requête en cours de traitement. De ce fait, une nouvelle sélection d'index est réalisée afin de choisir la meilleure configuration d'index pour toute la charge y compris la nouvelle requête L'architecture de notre proposition est illustrée sur la figure 1.8. A l'arrivée de chaque nouvelle requête Q_i, le déroulement du processus de sélection est réalisé comme suit (algorithme 3) :

1. Extraire la configuration courante d'$\mathcal{IJB}$ $Config\_IJB$.
2. Générer l'ensemble Qi_IJB d'$\mathcal{IJB}$ multiples à partir de Q_i.

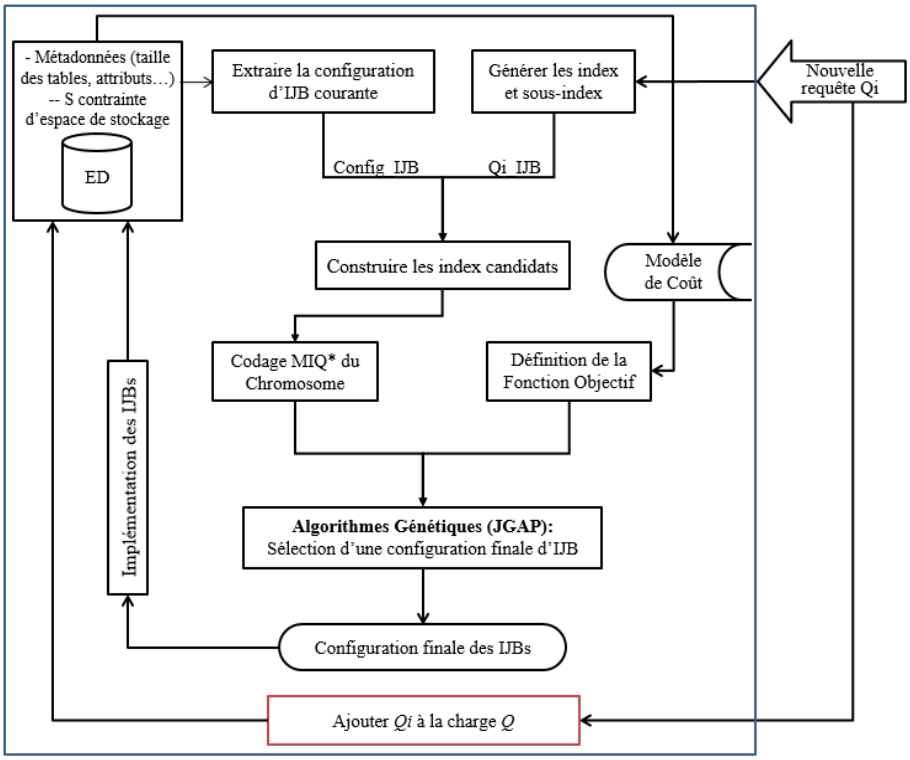


FIGURE 1.8 – Architecture de sélection Incrémentale des IJB par Algorithmes Génétiques GA

Algorithme de sélection incrémentale des IJB par AG
Entrée :
Q : charge de m requêtes.
Q_i : nouvelle requête exécutée.
$\mathcal{ED}$: les données et statistiques utilisées dans le modèle de coût.
S : espace de stockage des IJB.
Sortie : Configuration finale d'index C_f.
Début
Config_IJB ← ExtraireIJBCourant() ;
Qi_IJB ← GenererIJB(Q_i) ;
ConfigInit ← Config_IJB∪Qi_IJB ;
ChromIJB ⟵ CoderChromosomeIJB(ConfigInit, SI/MIQ/MIQ*) ;
FonctionIJB ⟵ CalculFonctionObjectifIJB(S, $\mathcal{ED}$, $Q \cup \{Q_i\}$) ;
C_f ⟵ JGAPIJB(ChromosomeIJB, FonctionIJB) ;
ImplémenterIJB(C_f) ;
Fin

Algorithme 3: Sélection incrémentale d'un schéma d'indexation

3. A partir des deux précédentes étapes, former une configuration initiale d'index $Config\_IJB \cup Qi\_IJB$.
4. Effectuer le codage MIQ^* du chromosome à partir de la configuration initiale d'index.

Table		Nombre de Tuples	Taille d'un enregistrement
Faits	Actvars	24786000	74
Dimensions	Prodlevel	9000	72
	Custlevel	900	24
	Timelevel	24	36
	Chanlevel	9	24

TABLE 1.5 – Description des tables de l'entrepôt

5. Définir la fonction objectif basée sur le modèle de coût.
6. Sélectionner par algorithme génétique guidé par modèle de coût la configuration finale d'index.

1.4 Expérimentation

Nous avons réalisé une série de tests de comparaison sur un entrepôt réel issu du benchmark APB1 [40] sous le SGBD Oracle 11g et une machine Intel Core 2 Duo et une mémoire vive de 2Go. Ces tests visent a comparer les différentes stratégies de sélection statique et sélection incrémentale d'index de jointures binaires, suivant les deux types d'index (simples et multiples). Nous décrivons dans ce qui suit l'environnement de test, les différents tests théoriques effectués à la base d'un modèle de coût mathématique et des tests pratiques sous le SGBD Oracle 11g.

1.4.1 Environnement d'expérimentation

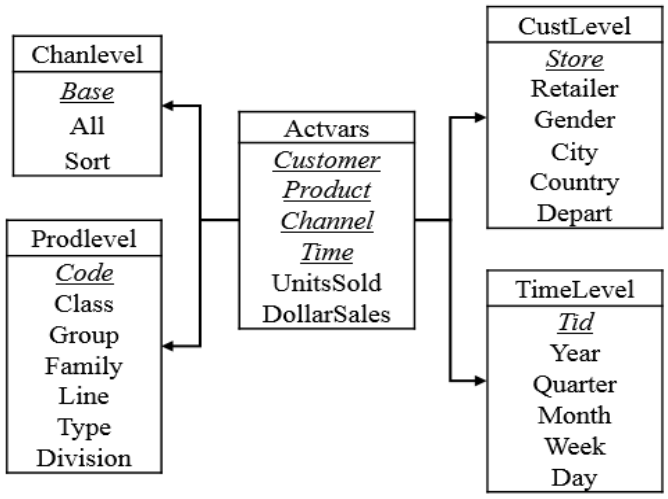


FIGURE 1.9 – Schéma en étoile de l'entrepôt de données issu du benchmark APB1

L'entrepôt de données issu du benchmark APB1 [40] est modélisé par un schéma en étoile représenté sur la figure 1.9. Il est constitué d'une table de faits Actvars et quatre tables de dimension Chanlevel, Custlevel, Prodlevel, Timelevel. La description des tables est représentée sur le tableau 1.5 On exécute sur cet entrepôt une charge de **70 requêtes de jointures en étoile** caractérisées par des

jointures, agrégation, sélection, etc. Ce sont des requêtes mono-bloc non imbriquées qui contiennent **18 attributs** de sélection dont la description et les cardinalités respectives sont données par le tableau 1.6

Attribut	Cardinalités
Country	11
Class	605
Group	300
Family	75
Line	15
Division	4
Year	2
Day	31
Week	52
Month	12
Quarter	4
Retailer	99
City	4
Gender	2
Depart	25
Type	25
Sort	4
Division	4

TABLE 1.6 – Attributs de sélection et cardinalités

1.4.1.1 Création de l'entrepôt

Afin de générer l'entrepôt de données, le benchmark APB1 dispose de trois ressources : (1) un fichier contenant le script SQL de création de la table de faits et des tables de dimensions, (2) un ficher exécutable APB.exe qui permet de générer les fichier de données utilisés pour peupler l'entrepôt et (3) des fichiers de contrôles qui permettent de spécifier les formats des fichiers de données et des tables dans lesquelles ces données seront chargées. D'abords, il faut exécuter le fichier script SQL de création des tables et l'outil APB.exe pour générer les données. Une fois les tables créées, un utilitaire appelé SQLLoader, fourni avec le SGBD Oracle, utilise les fichiers de contrôles pour remplir les différentes tables.

1.4.1.2 Évaluation théorique et pratique d'une approche de sélection

Une fois l'entrepôt généré et peuplé, nous effectuons les différents tests théoriques et pratiques. Dans les tests théoriques nous utilisons un modèle de coût mathématique pour satisfaire deux besoins : (1) pour chaque stratégie de sélection, guider l'algorithme de sélection afin de trouver la configuration finale d'index (optimale ou quasi-optimale) et (2) estimer le coût d'exécution de la charge de requêtes en présence d'une configuration finale d'index et pouvoir ainsi évaluer l'efficacité d'une approche

de sélection. Le coût d'une requête représente le nombre d'Entrées/Sorties système nécessaires pour charger les données en mémoire centrale et exécuter la requête.

Concernant les tests pratiques, ils se déroulent sous le SGBD Oracle 11g. Le but est d'implémenter les solutions sélectionnées par les différentes approches de sélections puis de faire appel à l'Optimiseur Oracle pour calculer le coût d'exécution réel des requêtes sur l'entrepôt de données en présente de la technique d'optimisation sélectionnée. Nous avons développé une classe JAVA appelée ORACLECOST qui fait appel à l'Optimiseur Oracle à travers l'opération EXPLAIN PLAN. Cette opération calcul le coût réel d'exécution d'une requête, en se basant sur des statistiques et stocke les résultats dans une table système d'Oracle appelée PLAN_TABLE. Par la suite, notre classe JAVA ORACLECOST accède à cette table et récupère le coût réel des requêtes. Nous donnant la syntaxe SQL pour effectuer cette procédure. Soit une requête dont la syntaxe est la suivante :

```
SELECT  *
FROM  Table
WHERE Attribut = 'valeur'
```

la requête suivante permet de calculer le coût d'exécution de Q puis de stocker ce coût dans la table PLAN_TABLE dans une colonne COST avec une clé 'Q_Id' :

```
EXPLAIN PLAN  SET STATEMENT_ID = 'Q_Id' FOR
SELECT  *
FROM  Table
WHERE Table.Attribut = 'valeur'
```

Afin d'obtenir le coût de Q à partir de la table système, il suffit d'exécuter la requête suivante

```
SELECT  Cost
FROM  Plan_Table
WHERE STATEMENT_ID = 'Q_Id'
```

1.4.1.3 Facteurs d'expérimentation

Dans la majorité des tests que nous conduisons, nous évaluons deux facteurs ; le coût d'exécution de la charge de requêtes et le nombre de requêtes optimisées. Nous visons souvent à tester à quel point le coût et les requêtes sont optimisées par rapport à une situation sans aucune optimisation. Pour ce faire, nous exprimons deux taux : le taux d'optimisation du coût et le taux de requêtes optimisées. Nous calculons le taux d'optimisation du coût par la formule suivante :

$$\text{Taux d'optimisation du coût} = 1 - \frac{\text{Coût avec optimisation}}{\text{Coût sans optimisation}} \tag{1.10}$$

Par exemple, si le coût d'exécution de la charge de requêtes avec optimisation est égale a 3 Millions d'E/S et que le coût d'exécution sans optimisation est égale a 8 Millions d'E/S alors le taux d'optimisation apporté est 62.5% calculé comme suit : Taux d'optimisation du coût $= 1 - \frac{3}{8} = 62.5\%$
Également, le taux de requêtes optimisées est égales à :

$$\text{Taux de requêtes optimisées} = \frac{\text{Nombre de requêtes optimisées}}{\text{Nombre total des requêtes}} \tag{1.11}$$

Par exemple, pour une charge de 60 requêtes, si le nombre de requêtes optimisée est 50 alors le taux de requêtes optimisées est égale à $\frac{50}{60} = 83\%$

Dans un premier lieu, nous présentons des tests théoriques et pratiques sur les stratégies de sélection statique des index simples et multiples. Ensuite, nous conduisons une série de tests sur la sélection incrémentale des index.

1.4.2 Tests sur la nature des Index

Le but de cette étude expérimentale est de comparer les différentes stratégies de sélection statique d'index simples est multiples, pour savoir quelle est la stratégie qui génère la configuration d'index la plus bénéfique pour la charge de requêtes. Afin de mener à bien nos expérimentations, nous avons réalisés deux types de tests : des tests théoriques basés sur le modèle de coût mathématique défini dans la section 1.2.3, et des tests pratiques réalisés sous Oracle 11g qui calculent le coût d'exécution des requêtes en faisant appel l'Optimiser Oracle. Pour cela nous avons implémenté nos trois stratégies de sélection basées sur les algorithmes génétiques suivantes :

- démarche de sélection d'$\mathcal{IJB}$ simples SI,
- démarche de sélection d'$\mathcal{IJB}$ multiples MIQ,
- démarche de sélection d'$\mathcal{IJB}$ multiples avec codage amélioré MIQ^*

Afin de comparer nos démarches de sélection avec les travaux existants, nous avons implémenté les démarches suivantes :

- démarche de sélection d'$\mathcal{IJB}$ multiples appelée MC basée sur un algorithme glouton avec stratégie d'amélioration basée sur le modèle de coût. Dans les travaux de Boukhalfa et al. [9], les auteurs montrent que la stratégies de purification des attributs la plus bénéfiques et celle basée sur le modèle de coût car elle prend en compte plusieurs paramètres importants (Tailles des tables et des attributs, taille de la page système, etc.),
- démarche de sélection d'$\mathcal{IJB}$ multiples basée sur l'algorithme d'affinité des attributs AF [9].
- démarche de sélection d'$\mathcal{IJB}$ simples et multiples appelée DM qui emploi l'algorithme CLOSE de recherche des motifs fréquents fermés pour la construction d'$\mathcal{IJB}$ et un algorithme glouton pour sélectionner la configuration finale d'index , selon les travaux cités dans [2].

Dans l'étude théorique, nous utilisons le modèle de coût pour guider l'algorithme de sélection et pour estimer la qualité d'une solution sélectionnée par chacune des six démarches cités ci-dessus. Il est à noter que le coût total des 70 requêtes sans optimisation est de 42.5 $millions\ E/S$. La taille totale de tous les index simples avec compression pouvant être défini sur les 18 attributs est de 4Go. La taille totale des index multiples avec codage MIQ pouvant être définis sur les 70 requêtes est de 10Go. Nous rappelons la formule de calcul de l'espace de stockage d'un index IJB multiple définit sur n attributs.

$$Storage(IJB) = (\frac{\sum_{k=1}^{n} Log_2(|A_k|)}{8} + 8) \times |F|$$

1.4.2.1 Tests théoriques

Dans la première expérimentation, nous exécutons les six stratégies de sélection et varions la contrainte d'espace de stockage (S) de 0.5Go à 4Go. Pour chaque stratégie et chaque valeur de S, nous calculons le coût d'exécution de la charge (figure 1.10) et le taux des requêtes optimisées (figure 1.11) en présence des index générés. Cette expérimentation montre que les stratégies SI, $MIQ*$ et DM sont plus bénéfiques que les stratégies MC, MIQ et AF. Dans un premier lieu, la stratégie AF ne prend pas en compte la contrainte d'espace de stockage pour améliorer la configuration d'index générée par l'algorithme d'affinité. Lorsque l'espace de stockage est réduit, la configuration générée par AF est bénéfique, car les autres algorithmes sélectionnent peu d'index, mais quand le quota d'espace

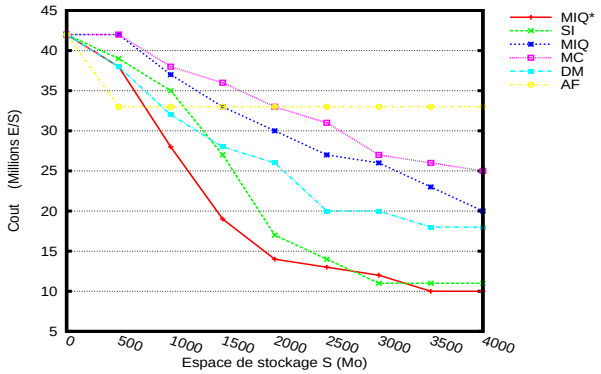


FIGURE 1.10 – Coût d'exécution des requête Vs. espace de stockage S

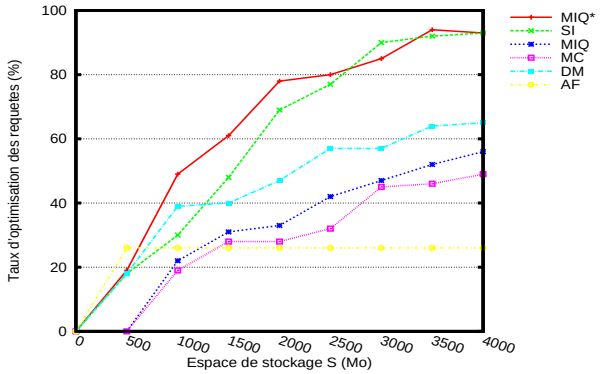


FIGURE 1.11 – Taux de requêtes optimisées Vs. espace de stockage S

augmente, les autres algorithmes permettent de sélectionner chacun une configuration meilleure. Dans un second lieu, les deux approches MC et MIQ emploient la définition des requêtes pour construire des $\mathcal{IJB}$ couvrant tous les attributs de chaque requête, ce qui donne des $\mathcal{IJB}$ très volumineux qui peuvent être écartés par le processus de sélection. Nous remarquons que l'approche MIQ est plus bénéfique que MC car elle emploi un algorithme génétique pour la sélection alors que MC emploi un algorithme glouton. Concernant les démarches SI, DM et MIQ^*, nous comparons leurs résultats comme suit :

1. MIQ^* *vs.* SI : la stratégie MIQ^* donne une meilleure optimisation que SI pour $S < 2.5Go$. Pour $S = 1.5Go$ par exemple, la démarche MIQ^* assure un coût de la charge de 19.5 *millions* E/S avec 61.4% des requêtes optimisées alors que la démarche SI donne un coût de 27.4 *millions* E/S et 48.5% des requêtes sont optimisées. En effet, selon le chromosome de MIQ^*, pour chaque $\mathcal{IJB}$ défini sur tous les attributs d'une requête, des sous index existent avec moins d'attribut et donc moins d'espace de stockage requis. De plus, deux index simples définis sur deux attributs A_1 et A_2 sont plus volumineux qu'un index multiple défini sur les même attributs. De ce fait, il y a davantage d'index pour optimiser chaque requête sans violation de la contrainte d'espace de stockage. A partir des valeurs de $S > 2.5Go$, les deux stratégies sont bénéfiques. Le coût total de la charge de requêtes est réduit à 10.9 *millions* E/S avec un taux de 93% de requêtes optimisées. En effet, davantage d'index sont sélectionnés par l'algorithme génétique pour SI (15 attributs sur 18 sont sélectionnés par l'AG afin de créer 15 $\mathcal{IJB}$ simples) ce qui couvre l'optimisation d'une majorité des requêtes.
2. MIQ^* *vs.* DM : nous remarquons que pour toutes les valeurs de S, notre approche MIQ^* donne de meilleurs résultats que DM. L'approche MIQ^* génère des index qui optimisent la charge de 38.1 *millions* E/S avec 19.2% des requêtes optimisées ($S = 0.5Go$) jusqu'à 10.9 *millions* E/S avec 93% des requêtes optimisées ($S = 4Go$), contre une optimisation qui va de 38.8 *millions* E/S avec 18.2% des requêtes optimisées ($S = 0.5Go$), jusqu'à 18.1 *millions* E/S avec 65% des requêtes optimisées ($S = 4Go$). En effet, la sélection d'une solution finale d'index par algorithmes génétiques reste plus performante que la sélection par algorithme glouton. De plus, l'approche DM emploie uniquement la fréquence d'accès des requêtes pour la sélection de la configuration initiale d'index (configuration à soumettre à l'algorithme glouton), il est donc nécessaire d'utiliser d'autres critères de sélection comme la taille des tables, des tuples, la page système, du buffer, etc. [16]

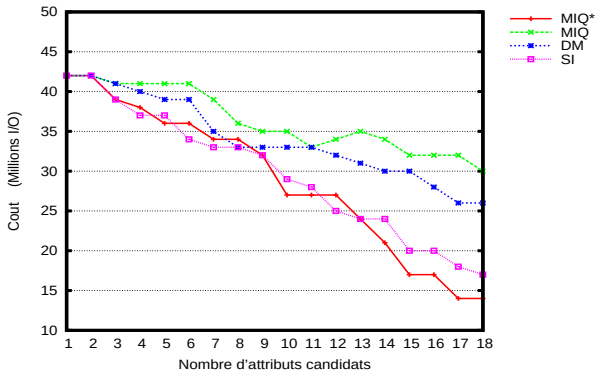


FIGURE 1.12 – Coût d'exécution vs. Nombre d'attributs candidats à la sélection

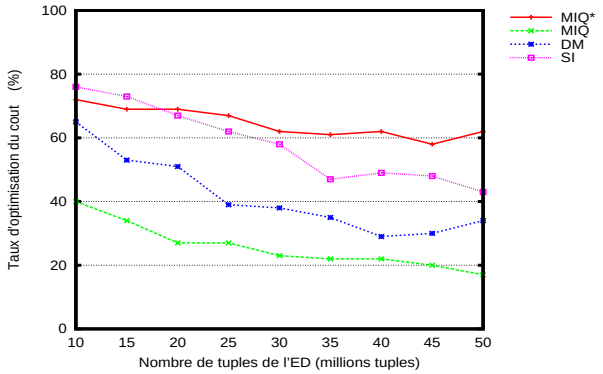


FIGURE 1.13 – Coût d'exécution vs. Taille de l'entrepôt

3. *DM vs. SI* : nous remarquons que la stratégie DM apporte une meilleure optimisation à SI pour $S < 1Go$ particulièrement en ce qui concerne le coût d'exécution des requêtes. Pour $S = 1Go$, l'approche DM assure un coût de 32.1 *millions* E/S avec 39.1% des requêtes optimisées contre un coût de 35.6 *millions* E/S et 33.1% des requêtes optimisées pour SI. En effet, l'emploi des index multiples permet de réduire le nombre de jointures en étoile à précalculer pour les requêtes ce qui réduit leur coût d'exécution. A partir de $S > 1Go$, l'approche SI apporte une meilleure optimisation du fait que cette sélection est guidée par modèle de coût, qu'elle emploi un algorithme génétique et que davantage d'attributs sont sélectionnés par l'AG pour définir la configuration finale d'index.

Afin de tester l'influence du nombre d'attributs indexables candidats à la sélection d'index, nous varions ce nombre de 2 à 18 sous une contrainte d'espace $S = 2.5Go$. Pour chaque nombre d'attributs, nous comparons nos trois stratégies SI, MIQ et MIQ^* avec la stratégie DM. Nous avons choisi DM car c'est la stratégie qui apporte le plus de bénéfice parmi les démarches existantes. La figure 1.12 illustre les résultats en terme de coût d'exécution des requêtes. Les tests montrent que l'augmentation du nombre d'attributs indexables influe positivement sur l'optimisation. En effet, plusieurs index sont sélectionnés et le coût est réduit à 30 *millions* E/S pour MIQ, 26.1 *millions* E/S pour DM, 17.5 *millions* E/S pour SI et 14.2 *millions* E/S pour MIQ^*.

Pour la troisième expérimentation, nous évaluons les performances d'optimisation par $\mathcal{IJB}$ selon l'augmentation de la taille de l'entrepôt de données. L'$\mathcal{ED}$ évolue continuellement ce augmente la complexité de la charge de requêtes. La stratégie d'optimisation doit donc être efficace. Ainsi, nous exécutons la sélection des $\mathcal{IJB}$ avec SI, DM, MIQ et MIQ^*, sous une contrainte d'espace $S = 2.5Go$, et varions le nombre de tuples de la table de faits de 10 millions tuples à 50 millions tuples. Le taux d'optimisation du coût de la charge de requêtes est illustré sur la figure 1.13. Cette figure montre que le taux d'optimisation du coût diminue avec l'augmentation de la taille de l'$\mathcal{ED}$, car les requêtes deviennent de plus en plus complexes et les index de plus en plus volumineux. Nous constatons que les démarches SI et MIQ^* donnent une meilleure optimisation de la charge de requêtes que MIQ et DM. En comparons SI et MIQ^*, nous remarquons que pour une taille d'entrepôt inférieure à 18 millions de tuples, SI est plus bénéfiques. En effet, les index sont moins volumineux compte tenue de la taille des tables. Par conséquent, d'avantage d'index sont sélectionnés dans le processus de sélection pour SI. Mais lorsque le volume de données augmente, l'espace de stockage des index augmente de manière linéaire réduisant la probabilité qu'un index soit sélectionné dans la configuration finale

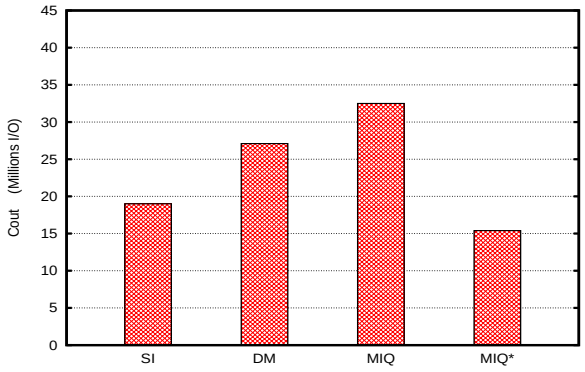


FIGURE 1.14 – Le coût réel des requêtes sous Oracle11g

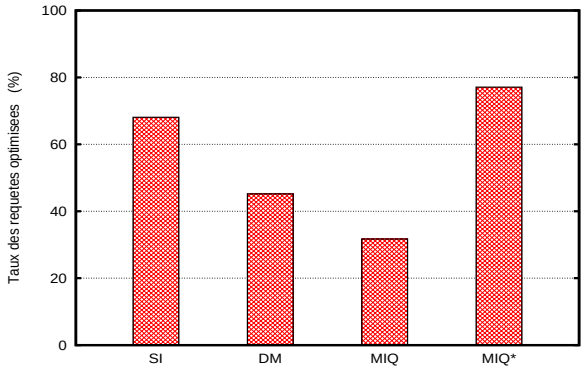


FIGURE 1.15 – Le taux de requêtes optimisées sous Oracle11g

d'index.

1.4.2.2 Tests pratiques sous Oracle 11g

Nous avons effectué des tests pratiques sous le SGBD Oracle 11g avec le banc d'essai APB1 [40], dans le but de tester notre sélection statique des $\mathcal{IJB}$ simples et multiples. Les tests sont réalisés en utilisant la charge des 70 requêtes en étoile. Nous exécutons les quatre stratégies de sélection (SI, DM, MIQ et MIQ^*) avec une contrainte d'espace de stockage $S = 2.5Go$. Chaque stratégie génère une configuration d'$\mathcal{IJB}$ que nous implémentons effectivement sur l'entrepôt de données. Après chaque implémentation, nous calculons le *Coût Réel* des requêtes en utilisant le modèle de coût réel. Le coût réel des requêtes est calculé en utilisant l'Optimiseur Oracle à travers la classe JAVA que nous avons développé ORACLECOST. Les figures 1.14 et 1.15 montrent respectivement le coût réel de la charge de requêtes et le taux des requêtes optimisées. Les tests sous Oracle11g montrent que la sélection basée MIQ^* donne de meilleurs résultats que les trois autres stratégies (19.4 *millions* E/S et 70.1% des requêtes sont optimisées). Nous concluons également que le modèle de coût théorique est proche du modèle réel et estime bien le coût de la charge de requêtes.

Nous présentons dans ce qui suit un tableau récapitulatif qui résume les résultats des tests de comparaisons entre les quatre stratégies (tableau 1.7). L'analyse est réalisée suivant l'algorithme de sélection, le type d'élagage, le type d'index et les performances de chaque approche pour l'optimisation de la charge de requêtes.

1.4.3 Tests sur la sélection Incrémentale

Dans cette étude expérimentale, nous visons à tester les algorithmes de sélection incrémentale d'$\mathcal{IJB}$ que nous avons développé et les comparer avec les travaux existants à savoir les travaux cités dans [3]. Les approches testées sont les suivantes :

- Démarche de sélection incrémentale naïve NI.
- Démarche de sélection incrémentale basée sur algorithmes génétiques GA.
- Démarche de sélection incrémentale basée sur l'algorithme CLOSE de Data Mining que nous appelons DMI, présentée dans les travaux cités dans [3].

Le point de départ pour tester chaque approche incrémentale est une charge de 50 requêtes préalablement optimisée par un ensemble initial d'index sélectionnés et implémentés sur l'entrepôt. L'ensemble

Stratégie	Élagage	Qualité d'élagage	Algorithme de sélection	Qualité de sélection	Type d'IJB	Apport du type d'IJB
AF	Affinité	+	Glouton	-	IM	++
MC	Requête	-	Glouton	+	IS/IM	++
DM	DataMining	+	Glouton	+	IS/IM	++
MIQ	Requête	-	AG	++	IM	++
SI	/	/	AG	++	IS	+
MIQ*	Requête amélioré	++	AG	++	IM	++

TABLE 1.7 – Tableau récapitulatif des tests de sélection statique

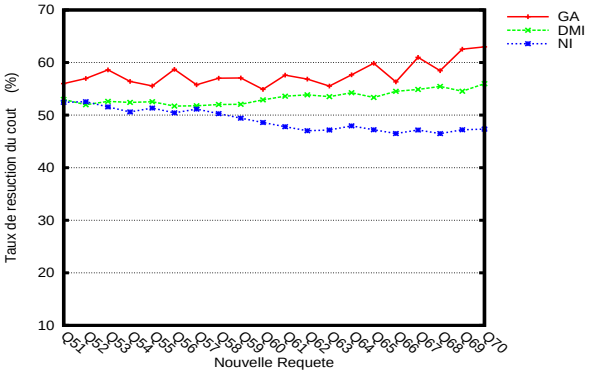


FIGURE 1.16 – Taux d'optimisation du coût d'exécution : NI, DMI et GA

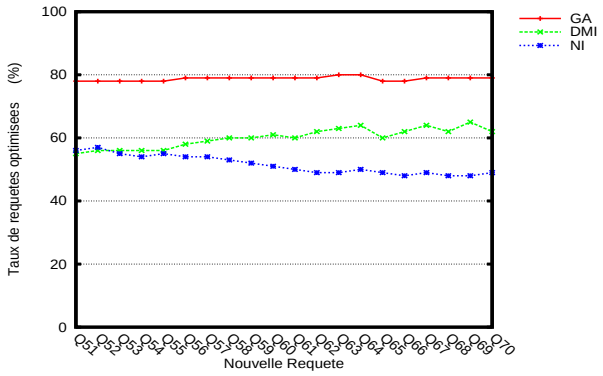


FIGURE 1.17 – Taux de requêtes optimisées : NI, DMI et GA

initial d'index a été sélectionné par notre approche de sélection statique des index multiples par codage $MIQ*$ pour les approches NI et GA. Pour l'approche DMI, l'ensemble initial est sélectionné par l'approche DM citée dans les travaux de [2]. L'étude incrémentale est réalisée avec ajout successif de 20 nouvelles requêtes en considérant une contrainte d'espace de stockage $S = 2.5Go$. Pour chaque requête nouvellement exécutée, nous exécution les trois sélections incrémentales (NI, DMI et GA). Pour chaque sélection et pour chaque nouvelle requête, nous relevons les informations suivantes : (1) le coût d'exécution de toute la charge de requêtes en cours, à partir duquel est calculé le taux d'optimisation du coût et (2) le taux des requêtes ayant été optimisées. Les figures 1.16 et 1.17 montrent respectivement le taux de réduction du coût d'exécution et le taux de requêtes optimisées, selon la sélection incrémentale par NI, DMI et GA.

Nous remarquons que les algorithmes génétiques permettent d'apporter une meilleure optimisation de la charge de requête. En effet, GA permet en moyenne une réduction de 58% du coût total des requêtes avec 80% des requêtes optimisées, contre 53% de réduction de coût et 61% des requêtes optimisées pour DMI et 49% de réduction de coût et 53% des requêtes optimisées pour NI. Contrairement à la sélection NI qui dégrade les performances, la sélection GA apporte une amélioration continue du coût de la charge pour chaque nouvelle requête. Cela est dû aux modifications incrémentales apportées au chromosome, prenant en compte les nouveaux index créés à partir de chaque nouvelle requête exécutée, ce qui permet de couvrir l'optimisation d'un plus grand nombre de requêtes et d'améliorer les requêtes déjà optimisées. De plus, l'emploi de l'algorithme génétique, avec prise en

compte de différents paramètres systèmes dans le modèle de coût, apporte une meilleure optimisation de la charge que DMI. En effet, DMI utilise uniquement la fréquence d'accès des requêtes pour la recherche des motifs fermés, ce qui influe négativement sur l'enrichissement de la base de connaissance qui est au coeur de la sélection incrémentale.

Nous résumons les tests effectués dans un tableau récapitulatif afin de comparer les stratégies de sélection incrémentale (tableau 1.8).

Stratégie	Algorithme de sélection	Qualité de sélection	Type d'IJB	Re-calcul de solution
NI	Naïf	0	IM	Partiel
DMI	Glouton	+	IS/IM	Partiel
GA	Génétique	++	IM	Total

TABLE 1.8 – Tableau récapitulatif des tests sur la sélection incrémentale

1.5 Conclusion

Dans ce chapitre, nous avons abordé le problème de sélection statique et incrémentale des index de jointure binaires. Nous avons proposé une nouvelle approche de sélection statique d'index basée sur les algorithmes génétiques et l'élagage par requêtes. Afin de réaliser la sélection d'index par algorithmes génétiques, nous avons commencé par définir la structure du chromosome qui permet de coder l'espace de recherche des index. Notre étude nous a mené à définir une structure de chromosome pour les index simples (SI) et deux structures pour les index multiples avec élagage par requêtes (une structure basée requêtes MIQ et une autre basée requête améliorée MIQ^*). Nous avons défini une fonction objectif qui permet de guider l'algorithme génétique dans la sélection d'index, puis nous avons détaillé l'algorithme génétique qui permet, indépendamment de la structure du chromosome, de sélectionner des index simples ou multiples en se basant sur l'*API JGAP*. Par la suite, nous avons identifié un nouveau problème qui représente le problème d'évolution de l'entrepôt de données qui rend les index implémentés sur l'entrepôt non efficaces. Par conséquent, nous avons défini une nouvelle architecture de sélection d'index incrémentale basée sur deux méthodes : la sélection incrémentale naïve et la sélection incrémentale avec des algorithmes génétiques.

Enfin, nous avons mené une batterie de tests afin de choisir la démarche de sélection d'index la plus adéquate pour l'optimisation d'une charge de requêtes selon deux axes : *statique* et *incrémentale*. Ces tests ont montré l'intérêt de l'utilisation des index multiples, de la sélection des index par algorithmes génétiques et de l'élagage par *requêtes* et par *requêtes améliorées*. De plus, nous avons comparé notre approche à des travaux de sélection statique et incrémentale existants, et avons ainsi démontré que nos propositions apportent de meilleures performances.

Le chapitre suivant porte sur notre étude approfondie de la fragmentation horizontale des entrepôts de donnes. Comme la sélection incrémentale est un enjeu important dans le cadre d'entreposage de données nous proposons une démarche de sélection incrémentale d'un schéma de fragmentation.

Chapitre 2

Sélection incrémentale isolée de la FH

Sommaire

2.1 Introduction

Dans le chapitre 1, nous avons effectué une analyse des travaux sur la fragmentation horizontale dans le contexte des entrepôts de données. Les algorithmes de sélection d'un schéma de fragmentation horizontale peuvent être classifiés en quatre principales catégories : (1) les algorithmes basés sur la minimalité et complétude de prédicats [80], (2) les algorithmes à base d'affinité des prédicats [99, 14], (3) les algorithmes à base de modèle de coût [12] et (4) les algorithmes à base de DataMining [56]. Une seconde classification vise à répartir les algorithmes de sélection en deux catégories comme suit : *(1) les algorithmes sans contrainte de fragmentation* [80, 6, 99] qui effectuent une sélection d'un schéma de fragmentation sans contrôler le nombre de fragments générés et *(2) les algorithmes avec contrainte de fragmentation* [14, 56] où le nombre de fragments générés est contrôlé et est considéré comme une contrainte d'optimisation défini par l'administrateur de l'entrepôt de données.

Nous avons constaté que la plupart des travaux proposent des approches de fragmentation horizontale primaire mono-table. Aussi, seuls les travaux présentés dans [14] proposent un codage des solutions exploitées par l'algorithme de sélection (schémas de fragmentation) sous forme d'une structure de données (tableau). L'utilisation de structures de données permet le développement de plusieurs algorithmes tels que des algorithmes génétiques qui sont basés principalement sur le codage de solutions. Nous avons constaté également que les différentes approches de sélection d'un schéma de fragmentation se base sur *une sélection statique* et supposent la connaissance préalable des requêtes. Dans la sélection statique, rien ne garantit l'efficacité du schéma de $\mathcal{FH}$ de l'entrepôt après l'évolution de la charge de requêtes (ajout de nouvelles requêtes, changement de la fréquence d'exécution d'une ou plusieurs requêtes). Ainsi, un seul travail de sélection dynamique ou incrémentale d'un schéma de fragmentation a été proposé [45].

Lorsqu'on définit une démarche de sélection incrémentale, il faut prendre en compte deux enjeux importants. Dans un premier lieu, la sélection incrémentale inclue une refragmentation d'un entrepôt déjà partitionné ce qui implique l'emploi de plusieurs opérations d'ajout/suppression d'attributs, ajout/suppression de sous-domaines d'attributs, fusion/éclatement des partitions de l'entrepôt. Il est donc impératif de déterminer l'ensemble des opérations à effectuer lors de l'évolution de la charge. Dans un second lieu, les auteurs dans [45] ont montré que la tâche la plus difficile pour une démarche de sélection incrémentale est la détermination du moment où il faut refragmenter.

Dans ce chapitre, nous proposons une démarche de sélection incrémentale isolée d'un schéma de fragmentation horizontale des entrepôts de données qui vise à répondre aux problématiques citées ci-dessus et qui englobe plusieurs propositions :

- Proposer une structure de données flexible pour coder un schéma de fragmentation [24].
- Proposer une nouvelle vision de fragmentation basée sur la structure de donnée flexible qui peut prendre en charge la fragmentation statique ou incrémentale [24, 8].
- Proposer un algorithme génétique qui exploite la structure de données pour la sélection statique et incrémentale [19, 24].
- Proposer une Algèbre dite Algèbre de Fragmentation $\mathcal{AF}$ qui modélise toute les opérations pouvant être appliquées sur la structure de données afin de s'adapter à l'évolution de charge [23].
- Proposer une sélection incrémentale basée sur le Profiling des requêtes afin de contrôler le déclenchement de la refragmentation de l'entrepôt de données [23].

Ce chapitre est organisé en sept sections. La section 2 présente la sélection statique d'un schéma de fragmentation par Algorithmes Génétiques. Dans la section 3 nous détaillons la sélection incrémentale d'un schéma de $\mathcal{FH}$. La section 4 présente l'Algèbre de Fragmentation. A la base de cette algèbre,

nous élaborons une sélection incrémentale basée sur le Profiling des requêtes expliquée dans la section 5. Dans la section 6 nous effectuons des expérimentations sur les différentes approches développées. Enfin, la section 7 conclue le chapitre.

2.2 Sélection statique d'un schéma de $\mathcal{FH}$ par Algorithmes Génétiques

Le problème de sélection incrémentale d'un schéma de fragmentation est un problème d'optimisation mono-objectif qui vise à optimiser un objectif, qui est le coût d'exécution de la charge de requêtes, avec une contrainte W sur le nombre de fragments faits pouvant être générés. Ce problème est un problème connu NP-Complet. Par conséquent, plusieurs travaux de recherches proposent d'employer des algorithmes non exhaustifs pour le résoudre comme les algorithmes génétiques.

Afin de développer notre sélection incrémentale d'un schéma de fragmentation, nous proposons un algorithme génétique pour la sélection statique d'un schéma de fragmentation qui se basent sur les travaux cités dans [14]. Les auteurs dans [14] montrent que le scénario le plus adéquat pour fragmenter un entrepôt de données est la fragmentation dérivée de la table de faits suivant la fragmentation primaire des tables dimensions. Ce scénario assure l'optimisation des opérations de sélection sur les tables dimensions (fragmentation primaire) et les opérations de jointures entre la table de faits et une ou plusieurs table de dimensions (fragmentation dérivée). Ainsi, fragmenter un $\mathcal{ED}$ revient à trouver un schéma de fragmentation des tables de dimensions. Les auteurs proposent donc plusieurs algorithmes, dont les algorithmes génétiques, qui exploitent un codage du schéma de fragmentation des tables dimensions pour trouver la solution optimale qui minimise une fonction objectif, à savoir le coût d'exécution de la charge de requêtes, sous une contrainte du nombre de fragments faits générés.

Les *AGs* sont des algorithmes itératifs, évolutionnaires qui s'inspirent de la théorie de l'évolution pour résoudre des problèmes d'optimisation [4]. Ils font évoluer un ensemble de solutions à un problème donné, dans le but de trouver la solution optimale. A chaque itération du *AG*, appelée aussi génération, une population de chromosomes (individus) est manipulée. Sur cette population, des opérateurs génétiques (croisement, mutation et sélection) sont appliqués afin de créer une nouvelle population qui optimise d'avantage une fonction objectif. Au fur et à mesure des générations, les individus vont tendre vers l'optimum de la fonction objectif. Chaque chromosome représente une solution potentielle au problème d'optimisation. Dans notre cas, un chromosome est un schéma de $\mathcal{FH}$ primaire des tables dimension. Par conséquent, la difficulté réside dans la définition d'un codage du schéma de $\mathcal{FH}$ qui permet de caractériser le chromosome de l'algorithme génétique et la formulation de la fonction objectif à minimiser qui guide l'algorithme vers la solution finale.

2.2.1 Codage du chromosome

Afin de réaliser la codification du schéma de fragmentation en un chromosome qui peut être exploité par l'algorithme génétique, nous nous basons sur les travaux des auteurs dans [14]. La codification se déroule en deux étapes :

1. *Découpage des domaines des attributs de fragmentation en sous-domaines* : Dans un schéma de fragmentation, chaque fragment horizontal est caractérisé par une conjonction de prédicats définis sur les attributs de fragmentation (attributs de sélections issus des tables dimensions). Ainsi, le découpage en sous-domaines des domaines de ces attributs permet de représenter un schéma

de fragmentation des tables correspondantes. Chaque attribut de fragmentation possède un domaine de valeurs qui peut être fini (valeurs discrètes) ou infini (valeurs numériques ou réelles). On procède donc au découpage du domaine de chaque attribut en plusieurs sous-domaines. Par exemple, l'attribut Genre possède un nombre fini de valeurs et le découpage en sous-domaines est donné par Dom(Genre)={'M','F'}. Par contre, l'attribut Age possède un domaine infini (une plage de valeurs) et un découpage peut être Dom(Age)={'[0,25[','[25,50[','[50,90['}. Un tel découpage peut être défini par l'administrateur de l'$\mathcal{ED}$ ou en utilisant la description des requêtes décisionnelles exécutées sur l'entrepôt. Ainsi, un schéma de $\mathcal{FH}$ peut être représenté sous forme d'un tableau de vecteurs, où chaque vecteur caractérise un attribut et où les cellules du vecteur représentent les différents sous-domaines de l'attribut. Le sous-domaine nommé $Reste_i$ est ajouté pour chaque attribut afin de prendre en compte les valeurs d'attributs existantes dans l'entrepôt, s'il y a lieu, mais ne figurant pas dans le schéma de fragmentation ce qui permet d'assurer la complétude des fragments générés (tableau 2.1).

Genre	M	F			
PNom	P1	P2	P3	$Reste_2$	
Ville	Alger	Oran	Blida	Kala	$Reste_3$

TABLE 2.1 – Tableau regroupant les attributs de fragmentation et leurs sous-domaines

2. *Codage du schéma de fragmentation* : Vu le nombre important des attributs de fragmentation dans le contexte d'entrepôts de données (plusieurs dizaines ou centaines d'attributs), le nombre de fragments des tables particulièrement la table de faits peut exploser. En effet, le nombre de fragments de la table de faits est le produit du nombre de fragments des tables dimensions. Ainsi, plusieurs sous-domaines d'un attribut donné peuvent être fusionnés en un seul ensemble de sous-domaines afin de réduire le nombre total de fragments faits. Le tableau 2.2 illustre un exemple d'un schéma de fragmentation.

Genre	M	F	
PNom	P1, P2	P3	$Reste_2$
Ville	Alger, Oran	Blida, Kala	$Reste_3$

TABLE 2.2 – Exemple d'un schéma de fragmentation

3. *Codage du chromosome* : A partir du schéma de fragmentation, l'encodage du chromosome est réalisé. Ce codage permet d'attribuer à chaque sous-domaine un numéro, les sous-domaines qui possèdent le même numéro sont fusionnés en un seul ensemble. Le tableau 2.3 illustre le codage en chromosome du schéma de fragmentation du tableau 2.2.

Genre	1	2			
PNom	1	1	2	3	
Ville	1	1	2	2	3

TABLE 2.3 – Schéma de fragmentation codé en chromosome

Exemple 14 *Soit un entrepôt de données composé d'une table de faits Ventes et les tables dimensions Clients et Produits. Considérons trois attributs de fragmentation : Genre et Ville de la dimension Clients, et PNom de la table Produits. Les sous-domaines de chaque attribut sont donnés comme suit :*

- *Dom(Genre)={'M','F'}*
- *Dom(PNom)={'P1','P2','P3'}*
- *Dom(Ville)={'Alger','Oran', 'Blida', 'Kala'}*

Les attributs de fragmentation et leurs sous-domaines respectifs sont représentés dans le tableau 2.4. L'encodage d'un schéma de fragmentation sous forme d'un chromosome est représenté dans le tableau 2.4. Pour l'attribut Ville par exemple, le sous-domaine Alger possède la valeur 1, la valeur 2 est attribuée au sous-domaine Oran et la valeur 3 permet de regrouper en un seul ensemble de sous-domaines les valeurs Blida et Kala. Le schéma de fragmentation SF, donne un schéma de $\mathcal{FH}$ avec 8 fragments de la table Clients (2 fragments sur Genre et 4 sur Ville) et 3 fragments sur Produits, soit un total de 24 (8×3) fragments faits.

	F	M			
Genre	1	2			
	P1	P2	P3	$Reste_2$	
PNom	1	2	2	3	
	Alger	Oran	Blida	Kala	$Reste_3$
Ville	1	2	3	3	4

TABLE 2.4 – Codage du schéma de fragmentation en chromosome

2.2.2 Fonction objectif

L'algorithme génétique est employé pour résoudre le problème statique mono-objectif de sélection d'un schéma de fragmentation. Il vise à sélectionner une solution optimale (un schéma de fragmentation SF) qui minimise une fonction objectif sous une contrainte sur le nombre de fragments du SF.

$$\begin{cases} \text{Minimiser} & Cost(Q, SF) \\ \text{avec} & NbFragment(SF) = N \leq W \end{cases} \tag{2.1}$$

Où N est le nombre de fragments de la table de faits générés par SF. Dans un premier lieu, nous allons présenter la formulation du modèle de coût $CostQ(Q, SF)$ qui permet d'évaluer le coût d'exécution de la charge de requêtes Q sur un $\mathcal{ED}$ partitionné suivant SF. Il est employé par l'algorithme génétique afin d'évaluer les solutions ou chromosomes en cours d'exploitation. Nous définissons le modèle de coût en se basant sur les travaux cités dans [14]. Nous considérons ce qui suit :

- Un entrepôt de données modélisé en étoile avec une table de faits $\mathcal{F}$ et d table de dimensions $D = \{D_1, D_2, ..., D_d\}$.
- N sous schémas en étoiles $SF = \{S_1, ..., S_N\}$ issus de la fragmentation de l'entrepôt selon un schéma de fragmentation horizontale SF.
- Une charge de t requêtes $Q = \{Q_1, Q_2, ..., Q_t\}$ à exécuter sur l'entrepôt.

Le modèle de coût calcule le nombre de pages qu'il faut charger pour l'exécution de la requête Q_k sur le schéma SF. Ce coût représente la somme des coûts d'exécution de Q_k sur chaque sous schéma S_i. Dans un sous schéma en étoile S_i, un fragment fait est spécifié par un ensemble de M_i prédicats de sélection PF_j. Un fragment de la dimension D_s est également spécifié par L_s prédicats de sélection PM_j^s. On définit le coût de chargement d'un fragment fait et un fragment dimension par les formules suivantes :

- pour un fragment fait : $\prod_{j=1}^{M_i} Sel(PF_j) \times |F|$,
- pour un fragment dimension D_s : $\prod_{j=1}^{L_s} Sel(PM_j^s) \times |D_s|$,

où $|R|$ et $Sel(P)$ représentent respectivement le nombre de pages nécessaires pour stocker la table R et le facteur de sélectivité du prédicat de sélection P. Le facteur de sélectivité d'un prédicat P noté $Sel(P)$ est une valeur dans $[0, 1[$ représentant le volume de données chargé par une sélection sur P. On suppose que les facteurs de sélectivité sont définis à travers une répartition uniforme (RU).

Le coût d'exécution d'une requête Q_k sur un sous schéma S_i estime le coût de chargement du fragment fait puis la jointure entre ce fragment et les fragments dimensions. On considère le coût de jointure par hachage donné par l'équation suivante :

$$Cost(Q_k, S_i) = \left(3 \times \left(\prod_{j=1}^{M_i} Sel(PF_j) \times |F| + \sum_{s=1}^{d} \prod_{j=1}^{L_s} Sel(PM_j^s) \times |D_s| \right) \right) \tag{2.2}$$

Afin d'estimer le coût d'exécution de Q_k sur un schéma fragmenté, il faut identifier les sous schémas valides pour la requête. Un sous schéma valide est un sous schéma où le fragment fait est accédé par la requête sur au moins un tuple. Soit NS_k le nombre de sous schémas valides pour Q_k. Le coût total d'exécution de Q_k sur tout l'entrepôt fragmenté est :

$$Cost(Q_k, SF) = \sum_{i=1}^{NS_k} Cost(Q_k, S_i) \tag{2.3}$$

Ainsi, le coût d'exécution de toute la charge de requêtes (t requêtes) sur l'entrepôt fragmenté est donné comme suit :

$$Cost(Q, SF) = \sum_{k=1}^{t} Cost(Q_k, SF) \tag{2.4}$$

Une fois le modèle de coût présenté, nous pouvons annoncer la fonction objectif. Le problème de sélection d'un SF est formalisé comme un problème d'optimisation mono-objectif avec contrainte. Il est transformé en un problème d'optimisation d'un objectif sans contraintes. Pour ce faire, nous exprimons une fonction de pénalité qui vise à pénaliser les SF dont le nombre de fragments dépasse la contrainte W. En effet, l'algorithme génétique exploite une population de chromosomes (schéma de fragmentation) dans chaque étape d'exécution. Il peut générer des solutions SF dont le nombre de fragments dépasse W. Par conséquent, ces schémas sont pénalisés par la fonction $Pen(SF)$ formulée comme suit :

$$Pen(SF) = \frac{N}{W} \tag{2.5}$$

Finalement, l'algorithme génétique doit sélectionner un schéma de $\mathcal{FH}$ qui minimise la fonction objectif suivante :

$$F(SF) = \begin{cases} Cost(Q, SF) \times Pen(SF), & \text{Si } Pen(SF) > 1 \\ Cost(Q, SF), & \text{Sinon.} \end{cases} \tag{2.6}$$

2.2.3 Processus de sélection

Afin d'exploiter l'algorithme génétique, nous avons utilisé la même API JAVA *JGAP* (Java Genetic Algorithms Package) présentée dans le chapitre 1. Elle permet d'implémenter l'algorithme génétique. La figure 2.1 illustre notre démarche de sélection statique d'un schéma de $\mathcal{FH}$ par algorithmes génétiques. Le déroulement du processus de sélection est donné comme suit :

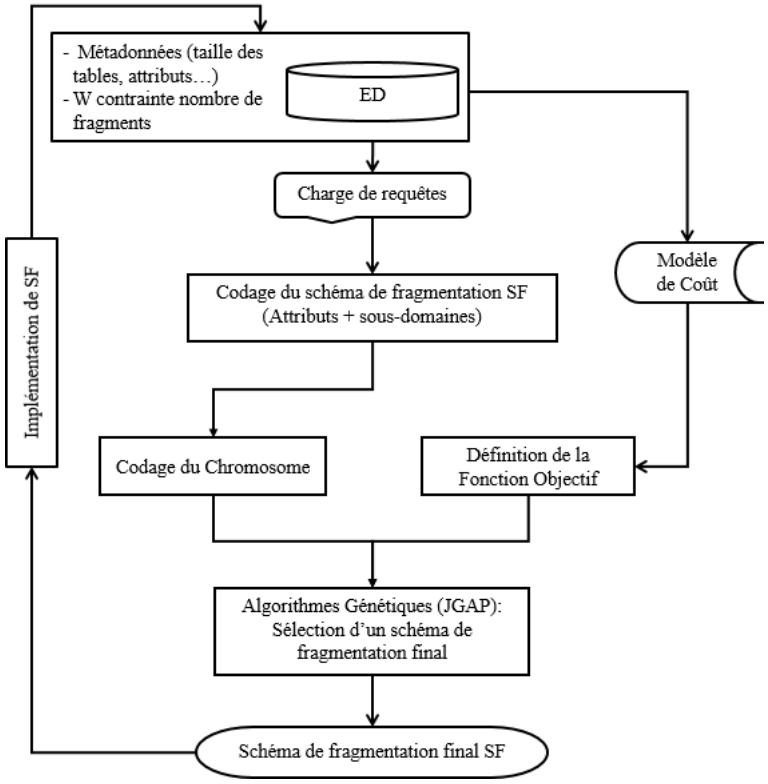


FIGURE 2.1 – Architecture de sélection statique d'un schéma de fragmentation par AG

1. **Analyse de la charge de requêtes :** cette étape permet d'extraire à partir des requêtes les attributs de fragmentation et leurs sous-domaines respectifs. Rappelons qu'un attribut de fragmentation est un attribut non clé d'une table dimension figurant dans la clause WHERE d'une ou plusieurs requêtes.
2. **Codage du schéma de fragmentation en chromosome :** cette étape est décrite dans le paragraphe 2.2.1.
3. **Définition de la fonction objectif :** cette étape est décrite dans le paragraphe 2.2.2.
4. **Sélection du schéma de $\mathcal{FH}$ optimal (quasi-optimal) par AG :** L'algorithme génère une population initiale de chromosomes, puis applique dessus les opérations génétiques (croisement, mutation et sélection) afin de générer les nouvelles populations. Chaque chromosome est évalué par la fonction objectif afin d'estimer le coût d'exécution des requêtes sur un $\mathcal{ED}$ fragmenté selon le schéma de $\mathcal{FH}$ décrit par le chromosome. Le meilleur schéma de fragmentation est sélectionné en fin de processus.
5. **Implémentation sur ED :** Plusieurs SGBD proposent une syntaxe pour créer des tables partitionnée lors la phase de conception physique. En prenant l'exemple du SGBD Oracle, la création d'une table partitionnée est réalisée par la requête CREATE TABLE selon un mode de fragmentation simple ou composite (Range, List, Hash, Range-Range, Range-Hach, Range-List, etc.). Nous donnons dans ce qui suit un exemple de création d'une table Clients partitionné selon le mode composite Range-List.

```
CREATE TABLE CLIENTS (
ClientID NUMBER(5),
Name VARCHAR2(50),
```

Algorithme de sélection d'un schéma de $\mathcal{FH}$
Entrées :
Q : charge de m requêtes
AS : ensemble de n attributs de fragmentation, $AS = \{A_1, \cdots, A_n\}$
SD : l'ensemble des sous-domaines des attributs de fragmentation, $SD = \{SD_1, \cdots, SD_n\}$
$\mathcal{ED}$: données relatives au modèle de coût (taille des tables, page système, etc.)
W : contrainte sur le nombre maximum de fragments faits à générer
Sortie : Schéma de $\mathcal{FH}$ primaire des tables dimensions SF.
Notations :
$Coder\_Chromosome$: coder le chromosome en un schéma de fragmentation
$FonctionFH$: fonction objectif pour l'AG
$Genetic\_FonctionObjectif$: élaborer la fonction objectif à partir du modèle de coût
$JGAP$: API JAVA qui permet d'implémenter l'algorithme génétique
Début
$ChromosomeFH = Coder\_Chromosome(AS, SD)$;
$FonctionFH = Genetic\_FonctionObjectif(Q, AS, W, \mathcal{ED})$;
$SF = JGAP(ChromosomeFH, FonctionFH)$;
Fin

Algorithme 4: Sélection statique d'un schéma de fragmentation par AG

```
Ville VARCHAR2(50),
Age NUMBER(2)
PARTITION BY RANGE(Age)
SUBPARTITION BY LIST(Ville) SUBPARTITION TEMPLATE
(SUBPARTITION CVille1 VALUES ('Alger'),
 SUBPARTITION CVille2 VALUES ('Annaba','Taref'),
 SUBPARTITION CVille3 VALUES ('Oran'),
 SUBPARTITION CVille4 VALUES (DEFAULT))
(PARTITION Clients1 VALUES LESS THAN (20) TABLESPACE TBS1,
 PARTITION Clients2 VALUES LESS THAN (40) TABLE SPACE TBS2,
 PARTITION Clients3 VALUES LESS THAN (MAXVALUE) TABLESPACE TBS3));
```

Nous présentons l'algorithme de sélection d'un schéma de $\mathcal{FH}$ par algorithme génétiques dans l'algorithme 4.

2.3 Sélection incrémentale d'un schéma de $\mathcal{FH}$

Les travaux qui traitent du problème de la sélection d'un schéma de $\mathcal{FH}$ se basent sur une sélection statique et ne permet pas de faire face aux changements qui surviennent sur l'$\mathcal{ED}$. Si un attribut "Ville" n'est plus souvent employé pour interroger l'entrepôt, pourquoi maintenir un schéma de fragmentation défini sur cet attribut, particulièrement si une contrainte sur le nombre maximum de fragments faits à générer a été définie par l'administrateur. Il serait plus judicieux de fusionner les fragments définis sur l'attribut "Ville" et de refragmenter l'$\mathcal{ED}$ suivant un autre attribut plus fréquemment employé par les requêtes décisionnelles. L'exécution d'une nouvelle requête Q_k sur

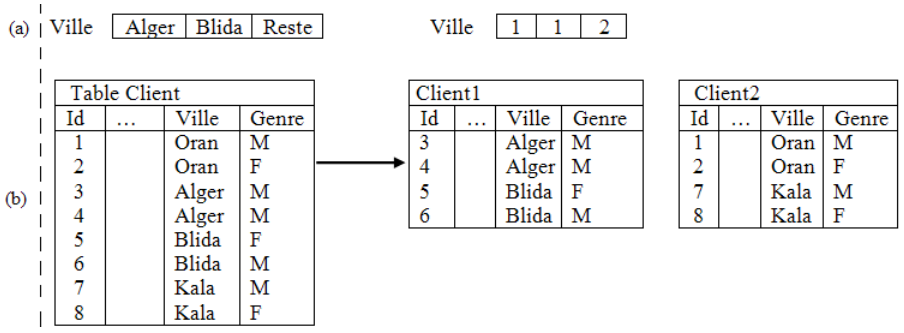


FIGURE 2.2 – La dimension Clients fragmentée

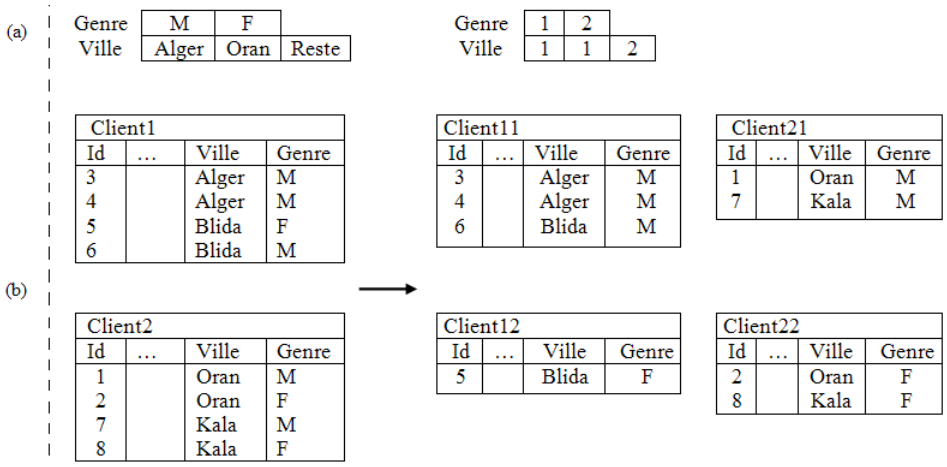


FIGURE 2.3 – La dimension Clients refragmentée selon le nouveau schéma de $\mathcal{FH}$

l'entrepôt de données peut provoquer l'ajout/suppression de nouveaux attributs de fragmentation ou d'éventuelles extensions des domaines des attributs. Cela peut conduire à réaliser des fusions ou éclatements des fragments de l'entrepôt. Sous le SGBD Oracle, il est possible de réadapter un $\mathcal{ED}$ fragmenté selon un nouveau schéma de $\mathcal{FH}$ en utilisant les opérations SPLIT PARTITION ou MERGE PARTITION. L'opération MERGE PARTITION permet de combiner deux fragments en un seul et ainsi diminuer le nombre de fragments générés. L'opération SPLIT PARTITION quant à elle permet d'éclater un fragment en deux et augmenter ainsi le nombre de fragments générés. Nous donnons un exemple de refragmentation d'un $\mathcal{ED}$ après exécution d'une nouvelle requête.

Exemple 15 *Soit un $\mathcal{ED}$ contenant une table de faits Ventes et une dimension Clients fragmentée suivant le schéma illustré dans la figure 2.2.a. Les données de la table Clients ainsi que sa fragmentation sont représentées dans la figure 2.2.b. Supposant l'exécution de la requête suivante :*

```
SELECT AVG(Prix)
FROM Ventes V, Clients C
WHERE V.IdC=C.IdC AND C.Genre = 'F'
```

L'exécution de cette nouvelle requête provoque l'ajout de l'attribut Genre et les sous-domaines $\{F\}$ et $Reste = \{M\}$. Afin de prendre en compte l'optimisation de cette requête, une nouvelle sélection d'un

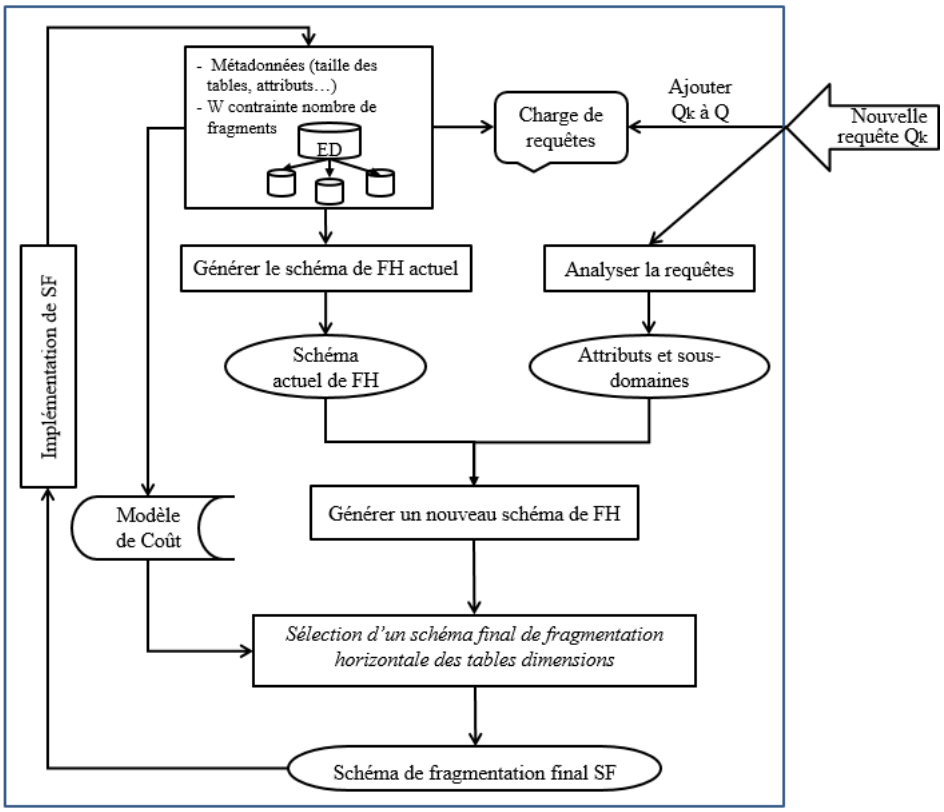


FIGURE 2.4 – Architecture générale de la sélection incrémentale d'un nouveau schéma de $\mathcal{F}\nabla\mathcal{H}$

schéma de $\mathcal{FH}$ est réalisée, cette sélection donne le schéma de $\mathcal{FH}$ de la figure 2.3.a. L'adaptation du nouveau schéma de $\mathcal{FH}$ sur l'$\mathcal{ED}$ nécessite deux opérations SPLIT sur les deux fragments Clients1 et Clients2 selon l'attribut Genre. Le résultat de la nouvelle fragmentation de la table Clients est donné sur la figure 2.3.b.

Nous proposons de mettre en œuvre une démarche de sélection incrémentale d'un schéma de $\mathcal{FH}$ qui se déclenche lorsque la charge de requêtes évolue par l'ajout ou le changement de la fréquence d'exécution d'une ou plusieurs requêtes. Nous présentons dans la figure 2.4 son architecture générale. Lors de l'exécution d'une nouvelle requête Q_k, le processus de sélection se déclenche. Décrit par l'algorithme 5, il est composé des étapes suivantes :

1. Générer le schéma de fragmentation actuel de l'entrepôt ActuelSF.
2. Analyser la nouvelle requête exécutée.
3. Générer un schéma initial InitSF à partir de ActuelS et des informations extraits de la requêtes (attributs et sous-domaines).
4. Sélectionner un schéma FinalSF.
5. Implémenter le FinalSF sur l'entrepôt de données.

Nous détaillons dans les sections suivantes les différentes étapes du processus de sélection.

Algorithme de sélection incrémentale d'un SF
Entrée :

Q : charge de m requêtes

Q_k : nouvelle requête exécutée

D : ensemble de d tables dimensions $\{D_1, \cdots, D_d\}$

AS : ensemble de n attributs non-clés des tables dimensions $AS = \{A_1, \cdots, A_n\}$

$\mathcal{ED}$: les données et statistiques utilisées dans le modèle de coût

W : contrainte sur le nombre maximum de fragments de la table de faits.

Sortie : Schéma final de Fragmentation des tables dimensions FinalSF.
Début

ActuelSF ← GenererActuelSF(D, AS) ;

SchemaQ ← AnalyserQ(Q_k) ;

InitSF ← GenererInitSF(ActuelSF, SchemaQ) ;

FinalSF ← SelectionnerFinalSF(InitSF, $Q \cup \{Q_i\}$, $\mathcal{ED}$, W) ;

ImplémenterFinalSF(FinalSF) ;

Fin

Algorithme 5: Sélection incrémentale d'un schéma de fragmentation

2.3.1 Générer le schéma de fragmentation actuel

La première étape du processus de sélection incrémentale vise à extraire le schéma de fragmentation horizontale actuel des tables dimensions. Pour ce faire, il faut identifier pour chaque dimension les attributs de fragmentation et les valeurs d'attributs qui ont participé à son processus de fragmentation. Il faut donc effectuer l'opération duale à la fragmentation. Considérons d'abords un scénario de fragmentation. Soit un entrepôt de données composé d'une table de faits F et d tables dimensions $\{D_1, \cdots, D_d\}$. Soit une table dimension D_k et un de ses attributs A_i dont le domaine est $Dom(A_i)$. Afin de fragmenter la table D_k suivant A_i, on définit une partition du domaine $Dom(A_i)$ en p sous-ensembles $Ens_1, ..., Ens_p$. La fragmentation sur A_i donne L fragments $\{DFrag_1^k, \cdots, DFrag_L^k\}$ disjoints deux à deux. En effectuant l'opération duale à la fragmentation et en se basant sur cette définition, nous constatons qu'un attribut muni d'un ensemble de sous-domaines $Ens_1, ..., Ens_p$ participe au processus de fragmentation si les p sous-ensembles représentent **une partition** au sens mathématique de l'ensemble $Dom(A_i)$, c'est à dire :

- $Ens_j \neq \varnothing$
- $\bigcup Ens_j = Dom(A_i)$
- $Ens_j \cap Ens_k = \varnothing$ si $j \neq k$

Le prédicat A_i *in* Ens_j caractérise un ou plusieurs fragment(s) de la table D_k.

Exemple 16 *Soit l'attribut Ville de la table dimension Clients et son domaine :*
Dom(Ville)={'Alger','Oran', 'Blida', 'Kala'}. Une partition de l'ensemble Dom(Ville) peut être donnée comme suit :
Partition(Dom(Ville))= {{'Alger'},{'Oran'}, {'Blida', 'Kala'}}. Cette partition donne une fragmentation de la table Clients en trois fragments :

- *Clients1*=$\sigma_{Ville='Alger'}(Clients)$

- *Clients2*=$\sigma_{Ville='Oran'}(Clients)$
- *Clients3*=$\sigma_{Ville='Blida' \vee Ville='Kala'}(Clients)$

Pour construire le schéma de fragmentation actuel de l'$\mathcal{ED}$, nous effectuons les étapes suivantes pour chaque attribut A_i :

1. *Extraire le domaine physique de A_i :* l'ensemble $Dom(A_i)$ contient toutes les valeurs possibles pour l'attribut A_i mais certaines valeurs de A_i peuvent ne pas être physiquement présentes au niveau de la table D_k. Ainsi, nous effectuons l'extraction des valeurs de A_i à partir de D_k pour former le domaine physique de A_i $PhyDom(A_i) = \{V_1, \cdots, V_f\}$. Cet ensemble est construit par l'exécution de la requête SQL suivante :

   ```
   SELECT DISTINCT(Ai)  FROM Dk
   ```

2. *Extraire les sous-ensembles de valeurs $Ens_1, ..., Ens_L$ de A_i :* Cette étape vise à obtenir une première répartition des valeurs de A_i en sous-ensembles. Nous accédons à chaque fragment de D_k : $\{DFrag_1^k, \cdots, DFrag_L^k\}$ et pour chaque fragment $DFrag_m^k$, nous récupérons les valeurs de A_i en exécutant la requêtes SQL suivante :

   ```
   SELECT DISTINCT(Ai)  FROM DFragkm
   ```

   Nous formons les sous-ensembles $Ens_1, ..., Ens_L$, où Ens_m contient les valeurs de A_i dans le fragment $DFrag_m^k$.

3. *Former une partition $Ens = \{Ens_1, ..., Ens_p\}$ de l'ensemble $PhyDom(A_i)$:* nous effectuons une épuration des sous-ensembles $Ens_1, ..., Ens_L$ pour obtenir les sous-ensembles $Ens_1, ..., Ens_p$ qui forment une partition de $PhyDom(A_i)$. L'épuration est effectuée par les opérations suivantes :
   - Éliminer tous les sous-ensembles Ens_j qui vérifient :
     $\exists Ens_k \in Ens$ tel que $Ens_j \subseteq Ens_k$.
   - Fusionner deux à deux tous les sous-ensembles Ens_j et Ens_k tel que :
     $Ens_j \cap Ens_k \neq \varnothing$.

   Ces deux étapes sont ré-exécutées jusqu'à stabilité de l'ensemble Ens. L'algorithme 6 décrit la fonction getPartition() qui permet d'extraire de la table D_k une partition du domaine physique de A_i.

4. *Vérifier si A_i est un attribut de fragmentation :* Selon la partition obtenue dans l'étape précédente $Ens = \{Ens_1, ..., Ens_p\}$, si $p = 1$ cela signifie que $Ens = PhyDom(A_i)$. Ainsi, aucune fragmentation n'est définie sur A_i. Cet attribut est éliminé du processus de construction du schéma de fragmentation actuel. Si $p > 1$, A_i muni des sous-ensembles $Ens_1, ..., Ens_p$ est un attribut de fragmentation.

5. *Construire le schéma de fragmentation actuel ActuelSF :* Un schéma de fragmentation est un tableau de vecteurs où chaque vecteur représente un attribut de fragmentation et chaque case du vecteur contient un ensemble de sous-domaine. L'algorithme 7 décrit la fonction ConstruireSchemaAtt() qui permet de construire un vecteur sur l'attribut A_i muni de ses ensembles de sous-domaines. Pour construire le schéma de fragmentation, il faut déterminer le découpage en sous-domaines des domaines des attributs. Nous constatons qu'il y a deux types d'attributs et la construction des sous-domaines diffère selon le type.

   (a) **Attributs à faibles cardinalités :** ces attributs possèdent un faible nombre de valeurs distinctes et un domaine de valeurs discret. Comme exemple on peut citer les attributs suivants :

```
Fonction getPartition( A_i, D_k : ) : EnsP
    Notation
      getValeurs(A_i, D_k) : Exécute la requête SELECT DISTINCT(Ai) FROM Dk.
    Début
    Pour DFrag_m^1 dans {DFrag_1^k, ··· , DFrag_L^k} faire
        Ens_j ⟵ getValeurs(A_i, DFrag_1^m)
        EnsL ⟵ EnsL ∪ {Ens_j}
    Fin Pour
    EnsP ⟵ ∅
    Tant que (EnsP ≠ EnsL) faire
        EnsP ⟵ EnsL
        Pour Ens_j dans EnsL faire
            Pour Ens_t dans EnsL faire
                Si ((Ens_j ⊆ Ens_t) OR (Ens_j ∩ Ens_t ≠ ∅)) Alors
                    EnsL ⟵ (EnsL − {Ens_j, Ens_t}) ∪ {Ens_j ∪ Ens_t}
                Fin Si
            Fin Pour
        Fin Pour
    Fait
    Retourner EnsP
Fin
```

Algorithme 6: Fonction getPartition

- Dom(Genre)={'M','F'}
- Dom(Ville)={'Alger','Oran', 'Paris', 'Montréal'}
- Dom(Pays)={'Algérie', 'Tunisie', 'Maroc', 'France', 'Canada'}

Si l'attribut A_i est un attribut de faible cardinalité alors chaque valeur de l'ensemble $PhyDom(A_i) = \{V_1, \cdots, V_f\}$ est un sous-domaine. Dans le schéma de fragmentation, les sous-domaines contenus dans Ens_j sont regroupés dans le même ensemble et sont mis dans la même case du vecteur correspondant à A_i.

(b) **Attributs à forte cardinalités :** ce sont des attributs aillant un domaine de valeurs très large souvent représenté sous forme d'un intervalle comme Dom(Age)=[0,90[. Si l'attribut A_i est un attribut à forte cardinalité, les sous-domaines représentent des intervalles de valeurs. Nous construisons les sous-domaines en exécutant les étapes suivantes :

i. Récupérer la valeur minimale de A_i sur chaque sous-ensemble $Ens_1, ..., Ens_p$.

ii. Ordonner ces valeurs par ordre croissant $V_1, \cdots, V_p$.

iii. Remplacer la valeur V_1 par 0 et ajouter V_{p+1} qui représente la valeur maximale de A_i fixée par administrateur. Cela permet de couvrir toutes les valeurs de A_i.

iv. Créer p intervalles $[V_g, V_{g+1}[$ où $1 \leqslant g \leqslant p$. L'ensemble de sous-domaines de l'attribut à forte cardinalité A_i est $SD_i = \bigcup\{[V_g, V_{g+1}[\}$

A l'issue de cette étape, chaque sous-domaine (intervalle) représente un singleton qui va être contenu dans une case du vecteur de A_i.

L'algorithme 8 décrit la fonction GenererActuelSF() qui permet d'extraire à partir d'un entrepôt de données fragmenté le schéma de fragmentation horizontale des tables dimensions.

Fonction ConstruireSchemaAtt(A_i, $EnsP$:) : $Vecteur$
Notations
$Vecteur$: vecteur contenant A_i et une répartition de ses sous-domaines.
FaibleCardinalité(A_i) : retourne vrai si A_i est de faible cardinalité, faut sinon.
MIN(Ens) : retournent la valeur minimale de Ens.
MAX(Ens) : retournent la valeur maximale de Ens.
Début
$Vecteur[0] \longleftarrow A_i$
Si (FaibleCardinalité(A_i)) **Alors**
  Pour j de 1 à $CARD(EnsP)$ **faire**
    $Vecteur[j] \longleftarrow Ens_j$
  Fin Pour
Sinon
  $MinV \longleftarrow \varnothing$
  Pour j de 1 à $CARD(EnsP)$ **faire**
    $MinV \longleftarrow MinV \cup \{MIN(Ens_j)\}$
  Fin Pour
  $j \longleftarrow 1$
  $MinV \longleftarrow (MinV - \{MIN(MinV)\}) \cup \{0\} \cup \{MAX(Dom(A_i))\}$
  Tant que $(MinV \neq \varnothing)$ **faire**
    $Vecteur[j] \longleftarrow]MIN(MinV), MIN(MinV - \{MIN(MinV)\})]$
    $MinV \longleftarrow MinV - \{MIN(MinV), MIN(MinV - \{MIN(MinV)\})\}$
    $j++$
  Fait
Fin Si
Retourner $Vecteur[i]$
Fin

Algorithme 7: Fonction ConstruireSchemaAtt

Exemple 17 *Soit la table dimension fragmentée Clients dont le schéma de données est donné par la figure 2.5. Nous procédons à l'extraction du schéma de fragmentation de la table Clients en exécutant l'algorithme 8. Le déroulement de l'algorithme est donné comme suit :*

1. *Attribut Ville :*
   (a) *PhyDom(Ville)={'Alger','Oran', 'Blida', 'Kala', 'Jijel', 'Annaba'}*
   (b) *Extraire les ensembles* $Ens_1, ..., Ens_L$ *où* $L = 4$ *:*
   - Ens_1 *={'Alger','Oran', 'Blida'},*
   - Ens_2 *={'Oran', 'Blida'},*
   - Ens_3 *={'Kala', 'Jijel'},*
   - Ens_4 *={'Annaba', 'Kala'}*

   (c) *Construire la partition* $Ens = \{Ens_1, ..., Ens_p\}$ *: Nous constatons que* $Ens_2 \subset Ens_1$ *et* $Ens_3 \cap Ens_4$*={'Kala'}. Par conséquent,* Ens_2 *est éliminé et* Ens_3 *et* Ens_4 *sont fusionnés. Au final, Ens={{'Alger','Oran', 'Blida'}, {'Kala', 'Jijel', 'Annaba'} }. Ainsi Ville est un attribut de fragmentation.*
2. *Attribut Métier :*

Fonction GenererActuelSF(D, AS :) : **ActuelSF**
Entrées :
D : ensemble de d tables dimensions $\{D_1, \cdots, D_d\}$
AS : ensemble de n attributs non-clé des tables dimensions $AS = \{A_1, \cdots, A_n\}$
Sortie : ActuelSF : Schéma de fragmentation actuel des tables dimensions.
Notations :
getTable(A_i) : Retourne la table à laquelle A_i appartient.
getValeurs(A_i, D_k) ; Exécute la requête SELECT DISTINCT(Ai) FROM Dk.
Début
Pour A_i dans $AS = \{A_1, \cdots, A_n\}$ **faire**
$D_k \longleftarrow$ getTable(A_i)
PhyDom$\longleftarrow$ getValeurs(A_i, D_k)
$EnsP \longleftarrow$getPartition(A_i, D_k)
Si ($EnsP \neq PhyDom(A_i)$) **Alors**
ActuelSF[i] $\longleftarrow$ConstruireSchemaAtt(A_i, $EnsP$)
Fin Si
Fin Pour
Retourner ActuelSF
Fin

Algorithme 8: Fonction GenererActuelSF() qui extrait le SF actuel de l'$\mathcal{ED}$

Client1

Id	…	Ville	Age	Métier
10		Alger	20	M1
4		Alger	15	M3
6		Oran	24	M3
5		Blida	18	M2

Client2

Id	…	Ville	Age	Métier
1		Oran	30	M1
12		Oran	45	M2
2		Blida	34	M1
8		Blida	32	M3

Client3

Id	…	Ville	Age	Métier
9		Kala	18	M4
3		Kala	15	M3
7		Jijel	23	M5

Client4

Id	…	Ville	Age	Métier
11		Annaba	45	M4
13		Kala	57	M5

FIGURE 2.5 – répartition de données de la table Clients

(a) *PhyDom(Métier)={'M1','M2', 'M3', 'M4', 'M5'}*

(b) *Extraire les ensembles* $Ens_1, ..., Ens_L$ *où* $L = 4$ *:*
- Ens_1 *={'M1','M2', 'M3'},*
- Ens_2 *={'M1','M2', 'M3'},*
- Ens_3 *={'M3', 'M4', 'M5'},*
- Ens_4 *={'M4', 'M5'}*

(c) *Construire la partition* $Ens = \{Ens_1, ..., Ens_p\}$ *: Nous constatons que* $Ens_1 = Ens_2$ *et* $Ens_4 \cap Ens_3$*='M4'. Après élimination et fusion nous obtenons :*
Ens= {Ens_1*={'M1','M2', 'M3'},* Ens_2*={'M3', 'M4', 'M5'}}. Les ensembles* Ens_1 *et*

Ens_2 sont fusionnés car $Ens_4 \cap Ens_3$= 'M3'. Au final nous obtenons : Ens={'M1','M2','M3','M4','M5'}=PhyDom(Métier). Ainsi, l'attribut Métier ne participe pas au processus de fragmentation.

3. *Attribut Age :*
   (a) *PhyDom(Métier)={20,15,24,18,23,30,45,34,32,45,57}*
   (b) *Extraire les ensembles $Ens_1, ..., Ens_L$ où $L = 4$:*
   - *Ens_1 ={20,15,24,18},*
   - *Ens_2 ={30,45,34,32},*
   - *Ens_3 ={18,15,23},*
   - *Ens_4 ={45,57}*

   (c) *Construire la partition $Ens = \{Ens_1, ..., Ens_p\}$: Nous constatons que $Ens_1 \cap Ens_3$={15,18} et $Ens_2 \cap Ens_4$={45}. Nous obtenons après élimination : Ens= {Ens_1={20,15,24,18,23}, Ens_2={30,45,34,32,57}}. Ainsi, l'attribut Age participe au processus de fragmentation.*

Nous obtenons deux attributs de fragmentation : un attribut de faible cardinalité Ville et un attribut de forte cardinalité Age. Pour Age nous procédons à la définition des sous-domaines qui donne $SDAge = \{\{[0, 30[\}, \{[30, 90[\}\}$. La dernière étape consiste à construire le schéma de fragmentation actuel de l'entrepôt de données ActuelSF en regroupant les attributs de fragmentation et les sous-domaines en un seul tableau représenté dans le tableau 2.5. Les sous-domaines appartenant au même ensemble Ens_j sont affectés à la même case. Par exemple, pour l'attribut Ville, les sous-domaines 'Alger','Oran', 'Blida' représente un même ensemble.

Age	[0, 30[	[30, 90[
Ville	Alger, Oran, Blida	Kala, Jijel, Annaba

TABLE 2.5 – Schéma de fragmentation actuel extrait ActuelSF

2.3.2 Analyser la requête nouvellement exécutée

La seconde étape du processus de sélection incrémentale est d'effectuer une analyse syntaxique de la requête nouvellement exécutée Q_k afin d'en extraire les attributs et sous-domaines. Soit l'expression générale d'une requête de jointure en étoile donnée par la requête suivante :

```
SELECT *
FROM F, D1, D2, ..., Dd
WHERE F.ID1=D1.ID1 AND F.ID2=D2.ID2 ... AND F.IDd=Dd.IDd
AND (A1 op V11 OR A1 op V12 ... OR A1 op V1k1)
AND (A2 op V21 OR A2 op V22 ... OR A2 op V1k2) ...
AND (An op Vn1 OR An op Vn1 ... OR An op Vnkn)
[ORDER BY ... ]
[GROUP BY ... ]
[HAVING ... ]
```

Le schéma de fragmentation est défini sur les attributs de fragmentation et leurs sous-domaines figurant dans les prédicats de sélection de la clause WHERE. Un prédicat de sélection est de la forme

A_i op V_{ij} où $1 < i < n$ et $1 < j < k_i$ et op$\in \{=, <, >, <>, \leq, \geq\}$. Chaque Valeur V_{ij} de chaque attribut peut être égale ou être contenue dans un sous-domaine de celui-ci. On extrait donc tous les attributs A_i et valeurs V_{ij} de la requête qu'on sauvegarde dans un tableau à deux dimensions SchémaQ. Chaque ligne représente un attribut A_i et chaque case de la ligne contient une valeur V_{ij}.

2.3.3 Générer un schéma de fragmentation initial

La troisième étape du processus de sélection incrémentale vise à générer un schéma de fragmentation initial qui prend en compte les changements apportés par l'exécution d'une nouvelle requête Q_k. Pour ce faire, nous ajoutons les attributs et valeurs extraits de la requête au schéma de fragmentation ActuelSF. Pour chaque attribut A_i et valeurs V_{ij} extrais de la requête, nous procédons comme suit :

1. Si A_i ne figure pas dans ActuelSF alors l'ajouter et ses valeurs en utilisant la fonction ConstruireSchemaAtt() décrite dans l'algorithme 7. Nous construisons un ensemble $EnsP = \bigcup\{\{V_{ij}\}\}$ représentant une partition des valeurs extraites V_{ij} où chaque valeur est contenue dans un ensemble. Pour assurer la complétude du schéma de fragmentation à construire, nous ajoutons à l'ensemble $EnsP$ l'ensemble $Reste_i = Dom(A_i) - \bigcup\{V_{ij}\}$. $Reste_i$ contient les valeurs de A_i qui ne figurent pas dans la requête.
2. Si A_i figure dans le schéma de fragmentation alors ajouter au schéma les valeurs V_{ij}. L'ajout des valeurs de l'attribut est réalisé de deux manières différentes selon le type de l'attribut.
   - A_i est un attribut de faible cardinalité : Les valeurs V_{ij} qui ne figurent pas dans ActuelSF sont ajoutées au vecteur ActuelSF[i] de A_i. Chaque valeur est considérée comme un sous-domaine et chaque sous-domaine est ajouté dans une case différente. Concernant les valeurs V_{ij} qui existent dans le schéma ActuelSF, nous procédons à l'éclatement de chaque ensemble qui contient une de ces valeurs en deux ensembles sur cette valeur.
   - A_i est un attribut de forte cardinalité : parmi les valeurs V_{ij} nous gardons uniquement les valeurs issues des prédicats A_i op V_{ij} où op$\in \{<, >, \leq, \geq\}$ qui décrivent des intervalles. Pour ces valeurs, détecter à quel intervalle appartient chaque valeur et répartir l'intervalle retrouvé en deux intervalles selon cette valeur. Supposant l'attribut Age et les sous-domaines [0,30[et [30,90[. Si V_{ij}=45 alors les nouveaux sous-domaines sont : [0,30[, [30,45[et [45,90[.

L'algorithme 9 décrit la fonction GenererInitSF() qui permet de construire un schéma de fragmentation initial.

Exemple 18 *Considérons le schéma de fragmentation actuel de l'$\mathcal{ED}$ décrit par le tableau 2.5. Supposant l'exécution de la nouvelle requête Q_k suivante :*

```
SELECT AVG(Prix)
FROM Ventes V, Clients C, Produits P
WHERE V.IdC=C.IdC AND V.IdP=P.IdP
AND C.Ville='Alger'
AND PNom in ('P1', 'P2')
```

La construction du schéma de fragmentation initial InitSF, décrit par le tableau 2.6, est établie comme suit :

1. *Extraire de Q_k les attributs Ville avec la valeur Alger et PNom avec les valeurs P1 et P2.*
2. *Pour l'attribut Ville, éclater l'ensemble {'Alger','Oran', 'Blida'} en deux ensemble {'Alger'} et {'Oran', 'Blida'}.*

Fonction GenererInitSF(ActuelSF, SchemaQ) : **InitSF**
Entrées :
SchemaQ : tableau d'attributs A_i et valeurs V_{ij} extraits de Q_k
Sortie : InitSF : schéma de fragmentation initial des tables dimensions.
Notations :
Exist(A_i, ActuelSF) : Retourne vrai si A_i figure dans le schéma ActuelSF.
addSousDoms(A_i, ValA$_i$) : Construit sur ValA$_i$ les sous-domaines de A_i.
Début
InitSF $\longleftarrow$ ActuelSF ;
Pour A_i dans SchemaQ **faire**
ValA$_i$ $\longleftarrow \bigcup \{V_{ij}\}$
Si (Exist(A_i, ActuelSF)) **Alors**
InitSF[A_i] $\longleftarrow$addSousDoms(A_i, ValA$_i$) ;
Sinon
Reste$_i$ $\longleftarrow$ Dom(A_i) $\setminus$ ValA$_i$;
EnsP$\longleftarrow \bigcup\{\{V_{ij}\}\} \cup \{\text{Reste}_i\}$;
InitSF[max] $\longleftarrow$ConstruireSchemaAtt(A_i, EnsP) ;
Fin Si
Fin Pour
Retourner InitSF
Fin

Algorithme 9: Fonction GenererInitSF qui retourne un SF initial

Age	[0, 30[	[30, 90[	
Ville	Alger	Oran, Blida	Kala, Jijel, Annaba
PNom	P1, P2	*Reste$_i$*	

TABLE 2.6 – Schéma de fragmentation initial InitSF

3. *Pour PNom, ajouter une ligne dans InitSF avec les ensembles {'P1', 'P2'} et {Reste$_i$} où Reste$_i$=Dom(PNom)-{'P1', 'P2'}.*

2.3.3.1 Codage du chromosome

Une fois le schéma de fragmentation initial défini, nous procédons à son codage en un chromosome qui va être exploité par l'algorithme de sélection d'un schéma de $\mathcal{FH}$ final. Le chromosome représente un tableau numérique de vecteurs. Chaque vecteur caractérise un attribut de fragmentation et chaque case du vecteur contient un numéro attribué à un sous-domaine. Les sous-domaines présents dans le même ensemble dans le schéma de $\mathcal{FH}$ obtenu précédemment se verront attribuer le même numéro. L'algorithme 10 décrit le processus de codage du schéma de fragmentation en chromosome.

Exemple 19 *Le codage en chromosome du schéma de fragmentation initial est donné par le tableau 2.7. Le chromosome décrit un schéma de fragmentation où la table Clients est fragmentée en 6 fragments (2 sur Age et 3 sur Ville) et la table Produits est fragmentée en 2 fragments sur PNom soit un total de 12 fragments de la table de faits.*

Algorithme : Codage du schéma de fragmentation en chromosome
Entrées :

InitSF : Schéma de fragmentation initial des tables dimensions.

Sortie : $ChromFH$: Tableau numérique représentant le codage du schéma InitSF.
Notations :

Length(T) : longueur du tableau T.

Début
  Pour i de 1 à $Length(InitSF)$ **faire**
    Pour j de 1 à $Length(InitSF[i])$ **faire**
      $m \longleftarrow 1$
      $SD \longleftarrow InitSF[i,j]$
      Pour SD_k dans SD **faire**
        $Chrom[i,m] \longleftarrow j$
        $m++$
      Fin Pour
    Fin Pour
  Fin Pour
Fin

Algorithme 10: Codage du schéma de fragmentation InitSF en chromosome

	[0, 30[	[30, 90[				
Age	1	2				
	Alger	Oran	Blida	Kala	Jijel	Annaba
Ville	1	2	2	3	3	3
	P1	P2	$Reste_3$			
PNom	1	1	2			

TABLE 2.7 – Codage en chromosome du schéma de fragmentation InitSF

2.3.4 Sélectionner un schéma de fragmentation final

Le schéma de fragmentation initial obtenu précédemment peut donner un nombre de fragments de la table des faits qui dépasse W. De ce fait, nous mettons en œuvre une démarche de sélection incrémentale d'un schéma de fragmentation final (optimal ou quasi-optimal). Dans un premier lieu, nous proposons de faire une sélection incrémentale dite Naïve ($FHNI$) qui se base sur de simples opérations de fusions et d'éclatement. Par la suite, afin de réadapter notre sélection statique d'un schéma de $\mathcal{FH}$ basée sur les AG, nous proposons une sélection incrémentale par algorithme génétique ($FHAG$).

2.3.4.1 Sélection Naïve $FHNI$

Considérons le schéma de fragmentation InitSF et son codage en chromosome ChromFH obtenus précédemment. Si la contrainte W est violée, une opération de fusion doit être réalisée afin de réduire le nombre de fragments. Cela consiste à fusionner deux à deux les ensembles de sous-domaines d'un ou plusieurs attributs. Considérons une table Clients fragmentée selon deux attributs Ville et Age en 6 fragments comme suit :

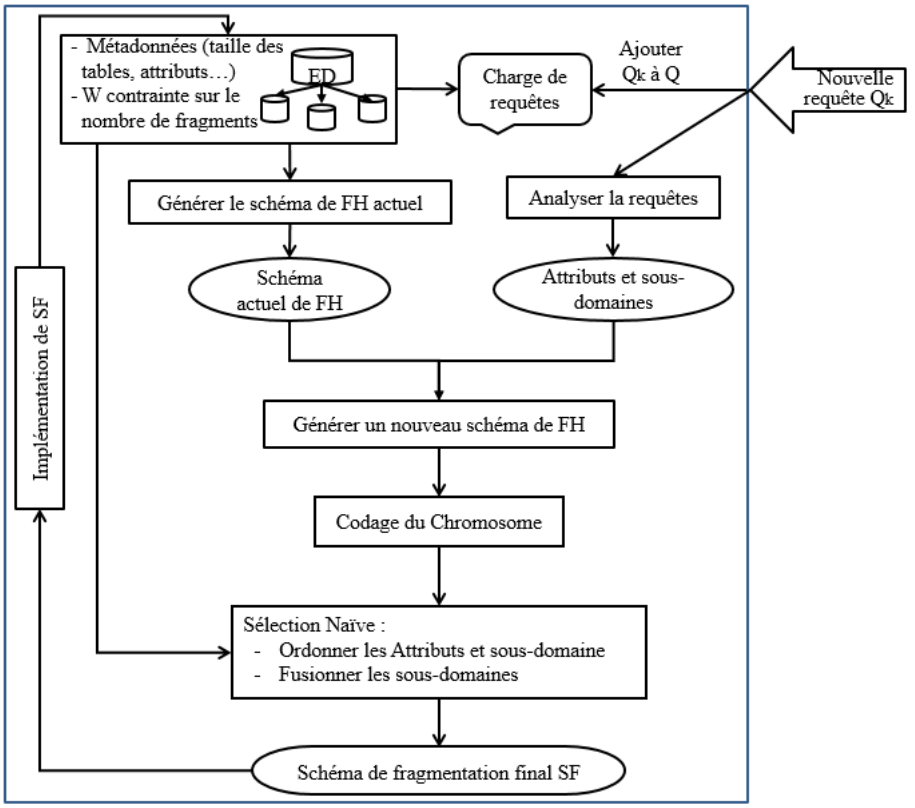


FIGURE 2.6 – Architecture de la sélection incrémentale Naïve $FHNI$

- Clients1=$\sigma_{Ville='Alger' \wedge Age<30}(Clients)$
- Clients2=$\sigma_{Ville='Alger' \wedge Age>=30}(Clients)$
- Clients3=$\sigma_{(Ville='Oran' \vee Ville='Blida') \wedge Age<30}(Clients)$
- Clients4=$\sigma_{(Ville='Oran' \vee Ville='Blida') \wedge Age>30}(Clients)$
- Clients5=$\sigma_{(Ville='Kala' \vee Ville='Annaba' \vee Ville='Jijel') \wedge Age<30}(Clients)$
- Clients6=$\sigma_{(Ville='Kala' \vee Ville='Annaba' \vee Ville='Jijel') \wedge Age>30}(Clients)$

La fusion des ensembles de sous-domaines {'Alger'} et {'Oran', 'Blida'} par exemple aboutie à 4 fragments de La table Clients.

- Clients1=$\sigma_{(Ville='Alger' \vee Ville='Oran' \vee Ville='Blida') \wedge Age<30}(Clients)$
- Clients2=$\sigma_{(Ville='Alger' \vee Ville='Oran' \vee Ville='Blida') \wedge Age>30}(Clients)$
- Clients3=$\sigma_{(Ville='Kala' \vee Ville='Annaba' \vee Ville='Jijel') \wedge Age<30}(Clients)$
- Clients4=$\sigma_{(Ville='Kala' \vee Ville='Annaba' \vee Ville='Jijel') \wedge Age>30}(Clients)$

Nous illustrons sur la figure 2.6 l'architecture de la sélection incrémentale Naïve. Le principe de la sélection $FHNI$ est donné par les étapes suivantes :

1. Ordonner les attributs suivant leurs fréquences d'utilisation par les requêtes, du moins utilisé au plus utilisé.
2. Pour chaque attribut, ordonner les sous-domaines selon leurs fréquences d'utilisation par les requêtes, du moins utilisé au plus utilisé.
3. Fusionner les ensemble de sous-domaines des attributs jusqu'à obtention d'un schéma de $\mathcal{FH}$ qui ne viole pas la contrainte W.

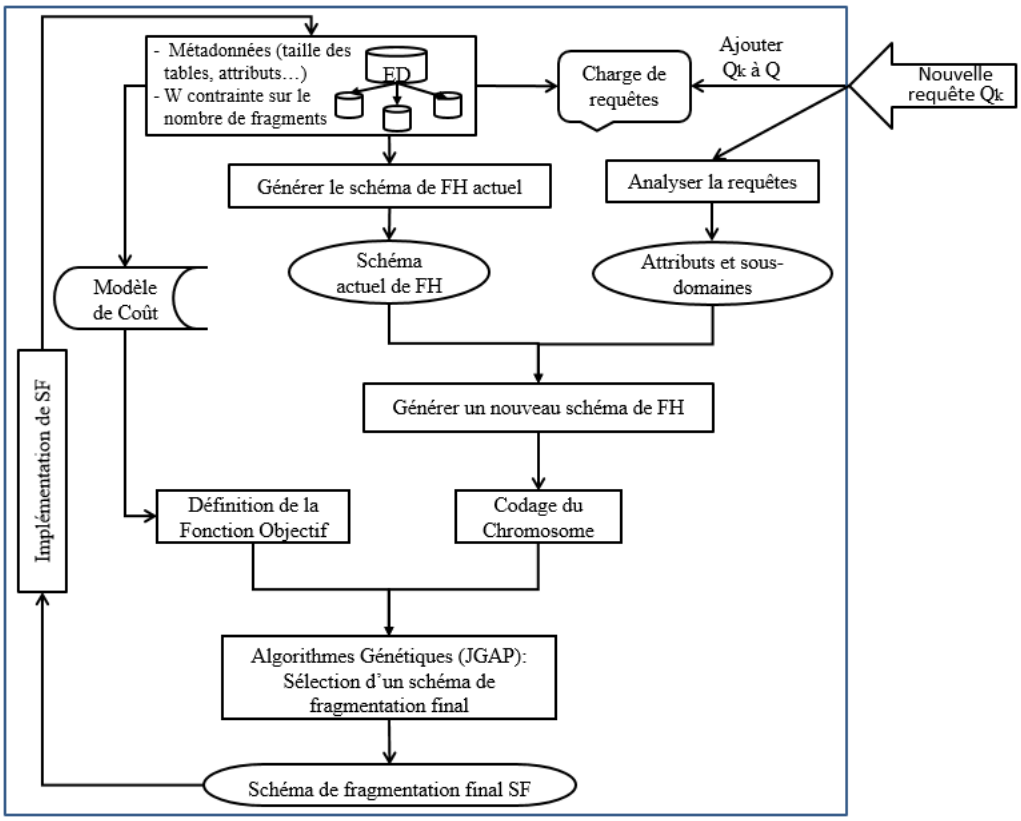


FIGURE 2.7 – Architecture de la sélection incrémentale par algorithmess génétiques $FHAG$

Exemple 20 *Supposant un schéma initial InitSF codé en chromosome ChromFH comme présenté précédemment dans le tableau 2.7. Ce schéma donne deux fragments sur Age, 3 fragments sur Ville et 2 fragments sur PNom soit un total de 12 fragments sur la table Clients. Soit une contrainte* $W = 4$. *Cette contrainte déclenche la* $FHNI$. *Supposant l'ordre des attributs Age, Ville, PNom. Les fusions successives appliquées sur ChromFH sont données par la table 2.8. Le schéma* $\mathcal{FH}$ *résultant possède 4 fragments, deux sur Ville et deux sur PNom.*

Age	1	2					$\Longrightarrow$	**1**	**1**						$\Longrightarrow$	**1**	**1**					
Ville	1	2	2	3	3	3		1	2	2	3	3	3			**1**	**1**	**1**	3	3	3
PNom	1	1	2					1	1	2						1	1	2			

TABLE 2.8 – Sélection incrémentale naïve : fusions successives appliquées sur $ChromFH$

2.3.4.2 Sélection par Algorithmes Génétiques $FHAG$

Nous illustrons sur la figure 2.7 l'architecture de la sélection incrémentale par algorithmes génétiques et dans l'algorithme 11 les étapes du processus de sélection. Considérons le schéma de fragmentation décrit par le chromosome $ChromFH$. Si la contrainte W est violée, nous effectuons une sélection d'un schéma de fragmentation final par algorithme génétique guidé par modèle de coût Cette sélection est décrite dans la section 2.2. L'exécution de notre sélection $FHAG$ sur le schéma de

Algorithme de sélection incrémentale d'un SF par AG
Entrée :
Q : charge de m requêtes.
Q_k : nouvelle requête exécutée.
D : ensemble de d tables dimensions $\{D_1, \cdots, D_d\}$.
AS : ensemble de n attributs non-clé des tables dimensions $AS = \{A_1, \cdots, A_n\}$.
ED : les données et statistiques utilisées dans le modèle de coût.
W : contrainte sur le nombre de fragments de la table de faits.
Sortie : Schéma final de Fragmentation des tables dimensions FinalSF.
Début
ActuelSF ← GenererActuelSF(D, AS) ;
SchemaQ ← AnalyserQ(Q_k) ;
InitSF ← GenererInitSF(ActuelSF, SchemaQ) ;
ChromFH ⟵ CoderChromosomeFH(InitSF) ;
FonctionFH ⟵ CalculFonctionObjectifFH(W, ED, $Q \cup Q_k$) ;
FinalSF ⟵ JGAPFH(ChromFH, FonctionFH) ;
ImplémenterFinalSF(FinalSF) ;
Fin

Algorithme 11: Algorithme de sélection incrémentale d'un schéma de FH par AG

fragmentation, présenté par la table 2.7, avec une contrainte $W = 10$ donne un schéma final décrit par le tableau 2.9. Ce schéma permet de générer 4 fragments sur la table Clients (2 fragments sur Age et 2 fragments sur Ville) et 2 fragments sur PNom de la table Produits, soit un total de 8 fragments faits.

	[0, 30[	[30, 90[				
Age	1	2				
	Alger	Oran	Blida	Kala	Jijel	Annaba
Ville	1	1	1	2	2	2
	P1	P2	$Reste_3$			
PNom	1	1	2			

TABLE 2.9 – Sélection incrémentale par Algorithme Génétique : schéma de fragmentation final

2.3.5 Implémenter le SF final sur l'entrepôt

L'implémentation du nouveau schéma de fragmentation sur un $\mathcal{ED}$ déjà fragmenté nécessite des opérations de fusions et/ou éclatements et/ou déplacements des différents fragments des tables. Dans le SGBD Oracle, on propose une syntaxe pour la fusion, l'éclatement et le déplacement des partitions et cela pour la fragmentation primaire. Concernant la fragmentation dérivée (mode REFERENCES), toute opération de fusion ou éclatement sur la table mère (tables dimensions) est propagée en cascade de manière automatique sur la table fille (table fille).

2.3.5.1 Syntaxe Oracle

La syntaxe Oracle de chaque opération est données dans ce qui suit, où f représente un fragment d'une table.

Split (f, Pr) Cette opération permet l'éclatement d'un fragment f suivant un prédicat Pr. Le prédicat est de la forme *Attribut* op $Valeur$ où op$\in \{=, <, >, <>, \leq, \geq\}$ et $Valeur$ est une valeur dans le domaine de *Attribut*. Cette opération permet d'éclater le fragment f en deux fragments f1 et f2 tel que toutes les instances de f1 vérifient le prédicat "Pr" et toutes les instances de f2 vérifient le prédicat "NOT Pr". La syntaxe SQL générale pour éclater une partition est **ALTER TABLE ... SPLIT PARTITION**. Selon le mode de partitionnement simple (RANGE, LIST ou HASH) ou composite (RANGE-LIST, RANGE-HASH, LIST-HASH, etc.) nous donnons les différentes syntaxes Oracle.

1. **Mode simple RANGE et HASH :** pour une table fragmentée sur un de ses attributs suivant le mode RANGE ou HASH, la syntaxe Oracle générale est donnée comme suit :

```
ALTER TABLE Nom_Table SPLIT PARTITION Nom_Partition AT (Valeur)
INTO
( PARTITION Nom_Partition1 [TABLESPACE Nom_TableSpace],
  PARTITION Nom_Partition2 [TABLESPACE Nom_TableSpace]);
```

2. **Mode simple LIST :** une table fragmentée selon un de ses attributs par LIST, peut être refragmentée selon une ou plusieurs valeur de cet attribut comme suit :

```
ALTER TABLE Nom_Table SPLIT PARTITION Nom_Partition
VALUES (Valeur1, Valeur2...) INTO
( PARTITION Nom_Partition1 [TABLESPACE Nom_TableSpace],
  PARTITION Nom_Partition2 [TABLESPACE Nom_TableSpace]);
```

3. **Mode simple INTERVAL :** un fragment d'une table fragmentée par INTERVAL sur un attribut peut être refragmentée sur une valeur de cet attribut contenue dans ce fragment.

```
ALTER TABLE Nom_Table
SPLIT PARTITION FOR(Valeur_Intervalle1) AT (Valeur_Intervalle2);
```

4. **Mode composite :** la syntaxe Oracle pour refragmenter une table fragmentée par mode composite est similaire à la refragmentation en mode simple, il suffit de remplacer le mot clé PARTITION par SUBPARTITION comme suit :

```
ALTER TABLE Nom_Table SPLIT SUBPARTITION Nom_SUBPartition AT (Valeur)
INTO
( SUBPARTITION Nom_SUBPartition1 [TABLESPACE Nom_TableSpace],
  SUBPARTITION Nom_SUBPartition2 [TABLESPACE Nom_TableSpace]);

 ALTER TABLE Nom_Table SPLIT SUBPARTITION Nom_SUBPartition
 VALUES (Valeur[, Valeur]...) INTO
 ( SUBPARTITION Nom_SUBPartition1 [TABLESPACE Nom_TableSpace],
   SUBPARTITION Nom_SUBPartition2 [TABLESPACE Nom_TableSpace]);
```

Exemple 21 *Supposant trois tables dimension fragmentés Clients, Produits et Temps. Pour chaque table nous donnons un exemple de refragmentation suivant un mode Oracle.*

1. ***Table Clients et mode RANGE :*** *Supposant la table Clients déjà fragmentée suivant le mode RANGE sur un attribut Age comme suit :*
   - *Clients1*=$\sigma_{Age<30}(Clients)$

– *Clients2*=$\sigma_{Age>=30}(Clients)$

Supposons qu'on refragmente la partition Clients2 suivant la valeur Age=45.

– *Clients1*=$\sigma_{Age<30}(Clients)$

– *Clients21*=$\sigma_{Age>=30 \wedge Age<45}(Clients)$

– *Clients22*=$\sigma_{Age>=45}(Clients)$

La requête suivante permet de refragmenter Clients2 en deux Clients21 et Clients22.

```
ALTER TABLE Clients SPLIT PARTITION Clients2 AT (45)
INTO (PARTITION Clients21, PARTITION Clients22)
```

2. ***Table Produits et mode LIST :*** *Supposons maintenant la table Produits fragmentée par mode LIST sur l'attribut PNom représentant le nom du produit muni du domaine Dom(PNom)={'IPhone4','IPhone5','Galaxy3','Galaxy4','HTC','Nexus4','Nexus5'}.*

– *Produit1*=$\sigma_{PNom\ IN\ ('IPhone4','IPhone5')}(Produit)$

– *Produit2*=$\sigma_{PNom\ IN\ ('Galaxy3','Galaxy4','HTC','Nexus4','Nexus5')}(Produit)$

La refragmentation du fragment Produit2 sur la listes des valeurs 'Galaxy3' et 'Galaxy4' donne le schéma suivant :

– *Produit1*=$\sigma_{PNom\ IN\ ('IPhone4','IPhone5')}(Produit)$

– *Produit21*=$\sigma_{PNom\ IN\ ('Galaxy3','Galaxy4')}(Produit)$

– *Produit22*=$\sigma_{PNom\ IN\ ('HTC','Nexus4','Nexus5')}(Produit)$

La requête suivante permet de refragmenter Produit2 en deux Produit21 et Produit22.

```
ALTER TABLE Produits SPLIT PARTITION Produit2 VALUES ('Galaxy3','Galaxy4')
INTO (PARTITION Produit21, PARTITION Produit22)
```

3. ***Table Temps et mode INTERVAL :*** *Supposons la table Temps partitionnée par INTERVAL sur l'attribut Date où chaque Mois défini un fragment. Rappelons la syntaxe de création d'une table Temps partitionnée par INTERVAL.*

```
CREATE TABLE Temps
( IDT NUMBER,
  Time_Date DATE,
  ...
)
PARTITION BY RANGE (Time_Date)
INTERVAL (NUMTOYMINTERVAL(1,'MONTH'))
( PARTITION Temps1 VALUES LESS THAN (TO_DATE('01-02-2014','dd-MON-yyyy')),
  PARTITION Temps2 VALUES LESS THAN (TO_DATE('01-03-2014','dd-MON-yyyy')));
```

Refragmenter un fragment de la table Temps en deux fragments est réalisé par la requête suivante :

```
ALTER TABLE Temps SPLIT PARTITION
FOR(TO_DATE('01-03-2014','dd-MON-yyyy'))
AT (TO_DATE('15-03-2014','dd-MON-yyyy'));
```

Merge (f1, f2) Cette opération permet de fusionner deux fragments f1 et f2 en un seul fragment f2 par exemple. Les données des deux fragments sont toutes fusionnées dans le fragment f2 et le fragment f1 est supprimé. Deux cas se présente selon le mode de fragmentation avec lequel la table a été fragmentée.

1. **Mode simple RANGE et INTERVAL :** afin de pouvoir fusionner deux partitions dans une table fragmentée par le mode RANGE ou INTERVAL, il faut que les deux partitions soit

adjacentes, auquel cas la fusion ne peut être effectuée. Le fragment résultant de l'opération de fusion aura comme limite inférieure et supérieure respectivement la limite inférieure du premier fragment et la limite supérieure du second fragment.

2. **Mode simple LIST et HASH :** si une table est fragmentée suivant l'un de ces modes, n'importe quels deux fragments, adjacents ou pas, peuvent être fusionnés en un seul fragment. En effet, la fragmentation par mode LIST n'assume aucun ordre entre les valeurs de l'attribut de fragmentation. Concernant le mode HASH, lors de la fusion la clé de hachage est recalculée sur le nouveau fragment. Le fragment résultant de la fusion contient toutes les instances des deux fragments fusionnés.

La syntaxe sous Oracle qui permet la fusion deux partitions est la suivante :

```
ALTER TABLE Clients MERGE PARTITIONS Fragment1, Fragment2
INTO PARTITION Fragment2;
```

Exemple 22 *Considérons une table Clients fragmentée par le mode RANGE suivant l'attribut Age.*

- *Clients1*=$\sigma_{Age<30}(Clients)$
- *Clients21*=$\sigma_{Age>=30 \wedge Age<45}(Clients)$
- *Clients22*=$\sigma_{Age>=45}(Clients)$

Seuls les fragments adjacents peuvent être fusionnés. Les fusions possibles sont Merge (Clients1, Clients21) et Merge (Clients21, Clients22). Les fragments Clients1 et Clients22 ne peuvent pas être fusionnés. La syntaxe SQL pour l'opération Merge (Clients21, Clients22) est donnée par :

```
ALTER TABLE Clients MERGE PARTITIONS Clients21, Clients22
INTO PARTITION Clients22;
```

Supposant maintenant que la même table Clients a été fragmentée selon le mode LIST sur l'attribut Ville, dont le domaine est Dom(Ville)={'Alger','Oran', 'Blida', 'Kala', 'Jijel', 'Annaba'}.

- *Clients1*=$\sigma_{Ville='Alger'}(Clients)$
- *Clients2*=$\sigma_{Ville='Oran' \vee Ville='Blida'}(Clients)$
- *Clients3*=$\sigma_{Ville='Kala' \vee Ville='Annaba' \vee Ville='Jijel'}(Clients)$

Trois opérations de fusions sont possibles Merge (Clients1, Clients2), Merge (Clients1, Clients3) et Merge (Clients2, Clients3). La syntaxe SQL pour Merge (Clients1, Clients2) est :

```
ALTER TABLE Clients MERGE PARTITIONS Clients1, Clients2
INTO PARTITION Clients2;
```

Add (f1, $\{V1, V2, \cdots, Vm\}$) Cette opération permet d'étendre la plage de valeurs d'un attribut de fragmentation utilisé pour fragmenter une table dimension suivant le mode LIST avec de nouvelles valeurs. Les valeurs $\{V1, \cdots, Vm\}$ sont ajoutées au fragment f1 à condition de ne pas figurer dans aucune des partitions de la table fragmentée. Pour l'ajout de nouvelles valeurs à une partition on utilise la syntaxe suivante :

```
ALTER TABLE Nom_Table MODIFY PARTITION Nom_Partition
ADD VALUES ('V1', 'V2',..., 'Vm');
```

Exemple 23 *Considérons une table Clients fragmentée par le mode LIST sur l'attribut Ville.*

- *Clients1*=$\sigma_{Ville='Alger'}(Clients)$
- *Clients2*=$\sigma_{Ville='Oran' \vee Ville='Blida'}(Clients)$
- *Clients3*=$\sigma_{Ville='Kala' \vee Ville='Annaba' \vee Ville='Jijel'}(Clients)$

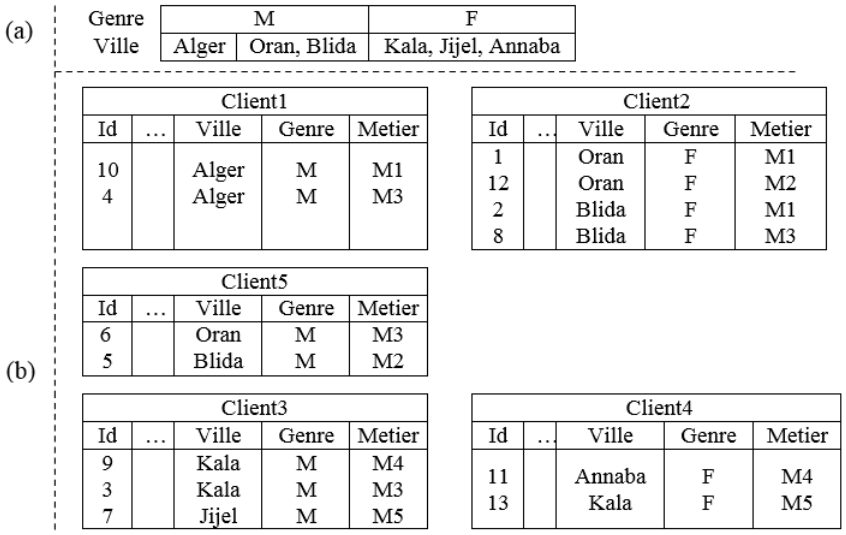


FIGURE 2.8 – Schéma de fragmentation de la table Clients actuel

Nous effectuons l'ajout de deux nouvelles villes 'Msila' et 'Bejaia'. La syntaxe Oracle qui permet d'effectuer cet ajout est donnée par :

```
ALTER TABLE Clients MODIFY PARTITION Clients1
ADD VALUES ('Msila','Bejaia');
```

L'exécution de cette requête donne le schéma de fragmentation suivant :

- *Clients1*=$\sigma_{Ville='Alger' \vee Ville='Msila' \vee Ville='Bejaia'}(Clients)$
- *Clients2*=$\sigma_{Ville='Oran' \vee Ville='Blida'}(Clients)$
- *Clients3*=$\sigma_{Ville='Kala' \vee Ville='Annaba' \vee Ville='Jijel'}(Clients)$

Move (f, TBS) Cette opération permet de déplacer le fragment f vers le tablespace TBS. Cette opération est réalisée automatiquement par le SGBD si une opération de fusion vient d'être effectuée ou si de nouvelles instances sont insérées dans le fragment f et qu'il n'y a plus assez d'espace dans le tablespace actuel. La clause Oracle est donnée comme suit :

```
ALTER TABLE Clients MOVE PARTITION Clients1 TABLESPACE tbs_2;
```

2.3.5.2 Exemple d'application

Exemple 24 *Nous allons présenter dans ce qui suit un example d'emploi des opérations physiques sous le SGBD Oracle afin d'adapter le schéma de fragmentation d'un entrepôt de données à un nouveau schéma sélectionné.*

Supposant un entrepôt de données dont le schéma de fragmentation de la table Clients est donné par la figure 2.8.(a) et le schéma de données est donné par la figure 2.8.(b). Après exécution d'une nouvelle requête Q_k, un nouveau schéma de fragmentation est sélectionné (figure 2.9.(a)). La répartition des données est illustrée sur la figure 2.9.(b). Les opérations physiques nécessaires pour l'implémentation physique du nouveau schéma de fragmentation sur le schéma actuel sont les suivantes :

- *Merge(Clients1, Clients5) : Dans le nouveau schéma de fragmentation, les valeurs Alger, Oran et Blida de l'attribut Ville sont fusionnées dans un seul ensemble ce qui requière une fusion deux à deux de toutes les partitions contenant ces trois valeurs. Dans notre cas les partitions à*

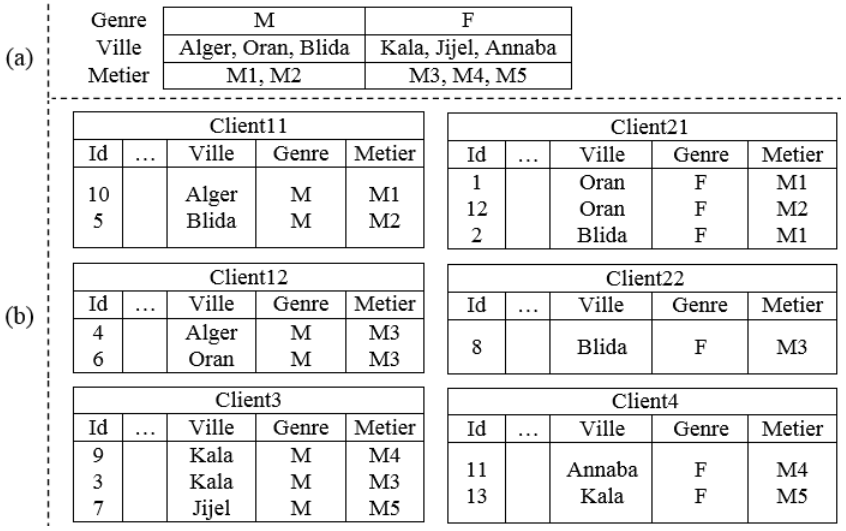


FIGURE 2.9 – Nouveau Schéma de fragmentation de la table Clients

fusionner sont Clients1 et Clients5. Cela donne quatre fragments Clients1, Clients2, Clients3 et Clients4.

- *Split(Clients$_i$, Metier IN ('M1','M2')) : un nouvel attribut Métier a été ajouté au schéma de fragmentation. Cet ajout nécessite la refragmentation de toutes les partitions Clients$_i$ de la table Clients suivant le prédicat (Métier IN ('M1','M2')) afin d'obtenir d'un côté des fragments dont les instances vérifient le prédicat (Métier IN ('M1','M2')) et d'un autre coté des fragments dont les instances vérifient le prédicat (Métier NOT IN ('M1','M2')). Pour réaliser la refragmentation physique des fragments Clients$_i$, il faut identifier les fragments qui nécessitent une refragmentation à savoir Clients1 et Clients2 puis appliquer l'opération Split. En effet, les fragments Clients3 et Clients4 contiennent uniquement des instances qui vérifient le prédicat Métier NOT IN ('M1','M2'). Au final, deux opérations d'éclatement sont nécessaires à savoir : Split(Clients1, Metier IN ('M1','M2')) et Split(Clients2, Metier IN ('M1','M2')) ce qui donne les partitions Clients11, Clients12, Clients21, Clients22, Clients3, et Clients4 représentées par la figure 2.9.(b).*

Les requêtes Oracle à exécuter pour l'implémentation du nouveau schéma de fragmentation sont données comme suit :

1. ```
   ALTER TABLE Clients MERGE PARTITIONS Clients1, Clients5
   INTO PARTITION Clients1;
   ```
2. ```
   ALTER TABLE Clients SPLIT PARTITION Clients1 VALUES ('M1','M2')
   INTO (PARTITION Clients11, PARTITION Clients12)
   ```
3. ```
   ALTER TABLE Clients SPLIT PARTITION Clients2 VALUES ('M1','M2')
   INTO (PARTITION Clients21, PARTITION Clients22)
   ```
   ```

2.4 Algèbre de Fragmentation

Le codage du schéma de fragmentation est au cœur de la sélection d'un schéma de fragmentation après évolution de la charge. Chaque nouvelle requête exécutée sur l'entrepôt de données cause une extension, appelée aussi évolution, ou une réduction du schéma de fragmentation. L'évolution est exprimée par l'ajout/suppression d'un attribut de fragmentation, l'ajout/suppression d'un ensemble de sous-domaines ou l'éclatement/fusion d'un ou plusieurs ensembles de sous-domaines. Afin de formaliser les opérations exactes nécessaires pour faire une évolution ou une réduction d'un schéma, nous avons formalisé une algèbre dite *Algèbre de Fragmentation AF* qui regroupe toutes les opérations qui peuvent être réalisées sur un schéma de fragmentation. Nous allons définir dans ce qui suit l'algèbre de fragmentation et ses opérateurs mais avant cela nous définissons le codage flexible du schéma de fragmentation, l'évolution et la réduction d'un schéma de fragmentation. Nous terminons par l'implémentation de l'algèbre sous le SGBD Oracle.

2.4.1 Codage flexible

Considérons un entrepôt de données $\mathcal{ED}$ avec une table de faits F et d tables de dimension $D = \{D_1, D_2, \cdots, D_d\}$. Un schéma de fragmentation est défini sur les attributs non-clés des tables dimension $AS = \{A_1, \cdots, A_n\}$. Chaque attribut A_i dispose d'un domaine Dom (A_i) partitionné en M_i sous-domaines SD_j^i comme suit : $Dom(A_i) = \{SD_1^i, SD_2^i, \cdots, SD_{m_i}^i\}$. Rappelons qu'en prenant l'exemple des attributs Age et Ville de la table dimension Clients, le partitionnement de leurs domaines respectif est donné comme suit :

- Dom(Age)= {'[0-20[','[20-45[','[45-80['}
- Dom(Ville)= {'Alger', 'Oran', 'Blida', 'Kala', 'Annaba', 'Jijel'}

Un attribut muni d'un de ses sous-domaines forment un prédicat de sélection utilisé pour spécifier le schéma de fragmentation de la table dimension à laquelle appartient cet attribut. Sur cette base, nous définissons une *Structure de données* qui représente *le Schéma de Fragmentation Maximal* SFM des tables de dimension représenté par le tableau 2.10. Rappelons que le nombre de fragments de la table de faits est le produit d'un nombre de fragments des tables dimensions à savoir $\prod_1^n m_i$.

A_1	SD_1^1	SD_2^1	$\cdots$	$SD_{m_1}^1$
A_2	SD_1^2	SD_2^2	$\cdots$	$SD_{m_2}^2$
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.
A_n	SD_1^n	SD_2^n	$\cdots$	$SD_{m_n}^n$

TABLE 2.10 – Schéma de Fragmentation Maximal des tables dimensions SFM

2.4.2 Réduction et évolution d'un SF

Dans le contexte d'entreposage de données, le nombre d'attributs de fragmentation est très élevés (des dizaines ou des centaines d'attributs). En outre, la quantité de données stockées dans un $\mathcal{ED}$ est énorme. En conséquence, le nombre de fragments de la table faits définis par le schéma de fragmentation maximal SFM est très important aussi. Par conséquent, nous pouvons construire un schéma

de fragmentation non maximale par fusion les sous-domaines d'un ou plusieurs attributs en un seul ensemble, ou par l'exclusion de certains attributs du processus de fragmentation. Nous pouvons obtenir un schéma de fragmentation réduit, comme illustré dans le tableau 2.11. Les ensembles $Reste_i$ sont exprimées comme suit :

- $Reste_1 = \{SD_1^1, \cdots, SD_{m_1}^1\} \setminus \{SD_1^1, SD_2^1, SD_3^1, SD_5^1, SD_4^1, SD_6^1\}$
- $Reste_2 = \{SD_1^2, \cdots, SD_{m_2}^2\} \setminus \{SD_1^2, SD_2^2\}$
- $Reste_3 = \{SD_1^3, \cdots, SD_{m_3}^3\} \setminus \{SD_1^3, SD_3^3, SD_5^3, SD_7^3, SD_9^3, SD_8^3\}$
- $Reste_4 = \{SD_1^4, \cdots, SD_{m_4}^4\} \setminus \{SD_1^4, SD_2^4\}$

La transition du schéma SFM au schéma SF est appelé **Réduction du Schéma de Fragmentation** RSF. La RSF comprend les opérations suivantes :

- supprimer les attributs $A_5, \cdots, A_N$,
- fusionner les sous-domaines des attributs A_1, A_2, A_3 et A_4.

A_1	SD_1^1	SD_2^1, SD_3^1, SD_5^1	SD_4^1, SD_6^1	$Reste_1$	
A_2	SD_1^2	SD_2^2	$Reste_2$		
A_3	SD_1^3, SD_3^3	SD_5^3	SD_7^3, SD_9^3	SD_8^3	$Reste_3$
A_4	SD_1^4	SD_2^4	$Reste_4$		

TABLE 2.11 – Réduction du schéma SFM au schéma SF

D'autre part, le schéma de fragmentation peut évoluer par l'ajout de nouveaux attributs fragmentation ou l'éclatement d'un ou plusieurs ensembles de sous-domaines des attributs. Nous présentons dans le tableau 2.12 l'évolution du schéma de fragmentation SF donnée dans le tableau 2.11. La transition du schéma SF vers le schéma SF' est appelé **Évolution du schéma de Fragmentation** ESF. L'ESF est une opération duale à l'opération RSF qui implique les opérations suivantes :

- ajouter l'attribut A_5 et les m_5 sous-domaines $Dom(A_5) = \{SD_1^5, \cdots, SD_{m_5}^5\}$,
- fusionner les sous-domaines de A_5 en deux ensembles $\{SD_1^5, SD_2^5\}$ et $Reste_5 = \{SD_1^5, \cdots, SD_{m_5}^5\} \setminus \{SD_1^5, SD_2^5\}$,
- éclater l'ensemble des sous-domaines de A_3 $\{SD_7^3, SD_9^3\}$ en deux ensembles,
- fusionner les deux ensembles $\{SD_5^3\}$ et $\{SD_7^3\}$ en un seul ensemble $\{SD_5^3, SD_7^3\}$,
- fusionner les deux ensembles $\{SD_8^3\}$ et $\{SD_9^3\}$ en un seul ensemble $\{SD_8^3, SD_9^3\}$.

A_1	SD_1^1	SD_2^1, SD_3^1, SD_5^1	SD_4^1, SD_6^1	$Reste_1$	
A_2	SD_1^2	SD_2^2	$Reste_2$		
A_3	SD_1^3, SD_3^3	SD_5^3, SD_7^3	SD_9^3, SD_8^3	$Reste_3$	
A_4	SD_1^4	SD_2^4	$Reste_4$		
A_5	SD_1^5, SD_2^5	$Reste_5$			

TABLE 2.12 – Évolution du schéma SF au schéma SF'

2.4.3 Opérateurs algébrique

Nous présentons dans ce qui suit la description, la classification et les propriétés des opérateurs de l'algèbre de fragmentation.

2.4.3.1 Description

Afin de réaliser une réduction RSF ou une évolution ESF d'un schéma de fragmentation SF, nous définissons les opérations nécessaires qui constituent l'Algèbre de Fragmentation AF. Considérons un attribut A_i, $1 < i < n$, muni de m_i sous-domaines $SD^i_1, \cdots, SD^i_{m_i}$.

1. $Add\_A(A_i, \{SD^i_{j_1}, \cdots, SD^i_{j_p}\})(SF)$: cet opérateur permet d'ajouter un attribut A_i au schéma de fragmentation SF en incluant l'ensemble de sous-domaines $\{SD^i_{j_1}, \cdots, SD^i_{j_p}\}$. Cela implique la création de l'ensemble $Reste_i = \{SD^i_1, \cdots, SD^i_{m_i}\} \setminus \{SD^i_{j_1}, \cdots, SD^i_{j_p}\}$.
2. $Add\_SD(A_i, \{SD^i_{j_1}, \cdots, SD^i_{j_p}\})(SF)$: ajouter à l'attribut A_i un ensemble de sous-domaines $\{SD^i_{j_1}, \cdots, SD^i_{j_p}\}$ et supprimer ce dernier de l'ensemble $Reste_i$.
3. $Split\_Dom(A_i, \{SD^i_{j_1}, \cdots, SD^i_{j_p}\}, \{SD^i_{k_1}, \cdots, SD^i_{k_s}\})(SF)$: éclater l'ensemble de sous-domaines $\{SD^i_{j_1}, \cdots, SD^i_{j_p}\}$ de l'attribut A_i en deux ensembles $\{SD^i_{k_1}, \cdots, SD^i_{k_s}\}$ et $\{SD^i_{j_1}, \cdots, SD^i_{j_p}\} \setminus \{SD^i_{k_1}, \cdots, SD^i_{k_s}\}$, où $\{k_1, \cdots, k_s\} \subset \{j_1, \cdots, j_p\}$.
4. $Merge\_Dom(A_i, \{SD^i_{j_1}, \cdots, SD^i_{j_p}\}, \{SD^i_{k_1}, \cdots, SD^i_{k_s}\})(SF)$: fusionner les deux ensembles de sous-domaines $\{SD^i_{j_1}, \cdots, SD^i_{j_p}\}$ et $\{SD^i_{k_1}, \cdots, SD^i_{k_s}\}$ en un seul ensemble, où $\{j_1, \cdots, j_p\} \subset [1, m_i]$ et $\{k_1, \cdots, k_s\} \subset [1, m_i]$.
5. $Del\_A(A_i)(SF)$: supprimer l'attribut A_i du schéma de fragmentation SF.
6. $Del\_SD(A_i, \{SD^i_{j_1}, \cdots, SD^i_{j_p}\})(SF)$: supprimer pour l'attribut A_i l'ensemble contenant les sous-domaines $\{SD^i_{j_1}, \cdots, SD^i_{j_p}\}$ et l'inclure dans l'ensemble $Reste_i$.

Afin d'assurer la complétude des opérateurs de l'AF, nous ajoutons le schéma de fragmentation vide $\emptyset$, qui vérifie pour chaque attribut A_i :

1. $Add\_A(A_i, \{SD^i_{j_1}, \cdots, SD^i_{j_p}\})(\emptyset)$: produit un schéma de fragmentation sur l'attribut A_i
2. $Add\_SD(A_i, SD^i_j)(\emptyset) = \emptyset$
3. $Split\_Dom(A_i, \{SD^i_{j_1}, \cdots, SD^i_{j_p}\}, \{SD^i_{k_1}, \cdots, SD^i_{k_s}\})(\emptyset) = \emptyset$
4. $Merge\_Dom(A_i, \{SD^i_{j_1}, \cdots, SD^i_{j_p}\}, \{SD^i_{k_1}, \cdots, SD^i_{k_s}\})(\emptyset) = \emptyset$
5. $Del\_A(A_i)(\emptyset) = \emptyset$
6. $Del\_SD(A_i, \{SD^i_{j_1}, \cdots, SD^i_{j_p}\})(\emptyset) = \emptyset$

En utilisant cette algèbre, nous pouvons exprimer les opérations d'évolution ESF du schéma SF (tableau 2.11) vers le schéma SF' (tableau 2.12) à travers la composition des opérateurs nécessaires comme suit :

$SF' = ESF(SF) =$
$Merge\_Dom(A_3, \{SD^3_8\}, \{SD^3_9\}) \circ Merge\_Dom(A_3, \{SD^3_5\}, \{SD^3_7\}) \circ$
$Split\_Dom(A_3, \{SD^3_7, SD^3_9\}) \circ Add\_A(A_5, \{SD^5_1, SD^5_3\})(SF)$.

2.4.3.2 Classification

Nous pouvons classifier les opérateurs de l'algèbre en deux catégories :

- Opérateurs d'évolution : ce sont les opérateurs requis pour effectuer une évolution ESF à savoir Add_A, Add_SD et $Split\_Dom$.
- Opérateurs de réduction : ce sont les opérateurs requis pour effectuer une réduction RSF à savoir $Del\_A()$, Del_SD et $Merge\_Dom$.

Nous pouvons donner une autre classification des opérateurs comme suit :

- Opérateurs horizontaux : ce sont les opérateurs qui modifient les attributs de fragmentation et qui sont : Add_A et Del_A.
- Opérateurs verticaux : ce sont les opérateurs qui modifient les ensembles de sous-domaines d'attributs et qui sont : $Add\_SD, Split\_Dom, Merge\_Dom$ et Del_SD.

2.4.3.3 Propriétés

Nous donnons maintenant les propriétés des opérateurs précédemment introduits. Ces propriétés vont permettre d'optimiser les opérations d'évolution ou de réduction d'un schéma de fragmentation en précisant l'ordre optimal dans lequel doivent s'exécuter les opérateurs. Cela permet d'ordonner ou réécrire les opérations ou de simplifier des opérateurs qui s'annulent mutuellement.

L'opérateur Inverse Nous introduisons l'opérateur identité $Id(SF)$ représentant l'élément identité de notre algèbre. Cet opérateur laisse inchangé le schéma de fragmentation sur lequel il est appliqué ($Id(SF) = SF$). Les propriétés sont données comme suit :

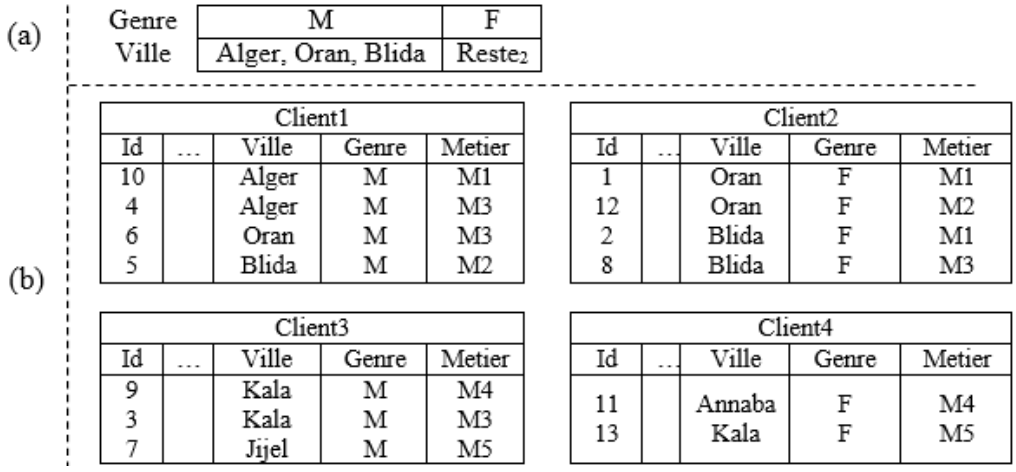
1. Del_A (resp. Add_A) est l'inverse gauche (resp. droit) de l'opérateur Add_A (resp. Del_A). Ces deux opérateurs ne sont pas commutatifs dans le cas général.
   $Del\_A \circ Add\_A = Id$.
2. $Split\_Dom(A_i, Set_1 \cup Set_2, Set_2)$ et $Merge\_Dom(A_i, Set_1, Set_2)$ sont des opérateurs inverses.
3. $Del\_SD(A_i, Set_1)$ et $Add\_SD(A_i, Set_1)$ sont des opérateurs inverses.

Règles d'équivalence

1. les opérateurs impliquant des attributs différents, ou ayant le même attribut mais concernant différents sous-domaines, sont commutatifs.
2. $Merge\_Dom(A_i, Set_1, Set_2) \circ Add\_SD(A_i, Set_2) \circ Add\_SD(A_i, Set_1)$ est équivalent à $Add\_SD(A_i, Set_1 \cup Set_2)$.
3. De façon plus générale, $Merge\_Dom(A_i, Set_1, Set_2, \cdots, Set_b) \circ Add\_SD(A_i, Set_b) \circ \cdots \circ Add\_SD(A_i, Set_2) \circ Add\_SD(A_i, Set_1)$ est équivalent à $Add\_SD(A_i, Set_1 \cup Set_2 \cup \cdots \cup Set_b)$.
4. La suppression d'un ensemble de sous-domaines de A_i est équivalente à la fusion de celui-ci avec l'ensemble $Reste_i$.
   $Del\_SD(A_i, Set_1) = Merge\_Dom(A_i, Set_1, Reste_i)$
5. La suppression d'un attribut est équivalente à fusionner successivement tous ses sous-domaines en un seul ensemble.
   $Del\_A(A_i) = Merge\_Dom(A_i, \{SD_1^i\}, Reste_i) \circ \ldots \circ Merge\_Dom(A_i, \{SD_{m_i}^i\}, Reste_i)$

2.4.4 Implémentation sous Oracle 11g

Supposant un entrepôt de données fragmenté suivant un schéma SF. Lors de la transformation d'un schéma SF vers un schéma SF' par une ESF ou une RSF au niveau logique, il est important de connaitre les opérations physiques qu'il faut réaliser afin d'implémenter effectivement le schéma SF' sur l'$\mathcal{ED}$. Lors de l'évolution de charge, trois opérations physiques peuvent être nécessaires afin d'adapter les changements survenues sur le SF actuel de l'entrepôt à savoir Split, Merge et Move. Ces opérations sont détaillées dans la section 2.3.5. Il est à noter que l'opération Add n'est pas considérée car nous supposons que l'extension des domaines des attributs est réalisée par l'administrateur de



(a)

Genre	M	F
Ville	Alger, Oran, Blida	Reste2

(b)

Client1

Id	...	Ville	Genre	Metier
10		Alger	M	M1
4		Alger	M	M3
6		Oran	M	M3
5		Blida	M	M2

Client2

Id	...	Ville	Genre	Metier
1		Oran	F	M1
12		Oran	F	M2
2		Blida	F	M1
8		Blida	F	M3

Client3

Id	...	Ville	Genre	Metier
9		Kala	M	M4
3		Kala	M	M3
7		Jijel	M	M5

Client4

Id	...	Ville	Genre	Metier
11		Annaba	F	M4
13		Kala	F	M5

FIGURE 2.10 – (a) schéma de fragmentation SFc, (b) table dimension Clients partitionnée

l'$\mathcal{ED}$ et non par les requêtes. Nous allons interpréter l'algèbre de fragmentation au niveau physique. Nous reprenons dans ce qui suit les opérations de l'AF et pour chaque opération nous spécifions les opérations physiques nécessaires à son implémentation. Pour ce faire, nous avons définies deux nouvelles opérations $Identifier\_Frag(Ct)$ et $Fusionnable(f1, f2, A_i)$ données comme suit :

- $Identifier\_Frag(Ct)$: permet d'identifier la ou les fragments des tables dimensions caractérisés par les prédicats spécifiés dans le critère Ct. Soit l'opération $Identifier\_Frag(Ville =' Alger')$. Elle renvoie un ensemble de fragments $\{f1, \cdots, fl\}$ dont les instances vérifient le critère Ville='Alger'.
- $Fusionnable(f1, f2, A_i)$: est une fonction booléenne qui renvoie *Vrai* si les deux fragments $f1$ et $f2$ peuvent être fusionnés uniquement sur l'attribut A_i. Elle renvoie *Faux* sinon. Deux partitions sont dites fusionnables si elles sont identifiées par la même conjonction de prédicats de sélection sauf pour un seul et unique prédicat. Ce dernier est défini sur l'attribut A_i appelé attribut de fusion.

  Exemple 25 *Supposant la table Clients dont le schéma de fragmentation nommé SFc et la répartition des données sont illustrés sur la figure 2.10. Les attributs de fragmentations et leurs domaines respectifs sont donnés comme suit :*
  - Dom(Genre)= {'F','M'}
  - Dom(Ville)= {'Alger', 'Oran', 'Blida', 'Kala', 'Annaba', Jijel'}
  - Dom(Métier)={'M1','M2', 'M3', 'M4', 'M5'}
  Nous appliquons les opérations $Identifier\_Frag(Ct)$ et $Fusionnable(f1, f2, A_i)$ sur les partitions de la table Clients. Nous donnons dans ce qui suit quelques résultats d'application :
  - $Identifier\_Frag(Ville = Blida)$ = {Clients1, Clients2}.
  - $Identifier\_Frag(Ville = Alger)$ = {Clients1}.
  - $Identifier\_Frag(Genre = F)$ = {Clients2, Clients4}.
  - $Fusionnable(Clients1, Clients2, Genre)$ = Vrai.
  - $Fusionnable(Clients1, Clients2, Ville)$ = Faux.
  - $Fusionnable(Clients1, Clients3, Ville)$ = Vrai.
  - $Fusionnable(Clients1, Clients4, Genre)$ = Faux.
  - $Fusionnable(Clients1, Clients4, Ville)$ = Faux.

$Add\_A(A_i, \{SD^i_{j_1}, \cdots, SD^i_{j_p}\})(SF)$
Début
  $Ens\_Frag = Identifier\_Frag(A_i \text{ in } Dom(A_i))$
  Pour Chaque f dans Ens_Part **faire**
    Split(f, A_i in $(SD^i_{j_1}, \cdots, SD^i_{j_p})$)
  Fin Pour
Fin

Algorithme 12: Implémentation de l'opérateur Add_A

Nous représentons dans ce qui suit l'implémentation physique des opérations de l'algèbre :

1. $Add\_A(A_i, \{SD^i_{j_1}, \cdots, SD^i_{j_p}\})(SF)$: ajouter un attribut A_i nécessite de fragmenter tous les fragments de la table dimension à laquelle appartient A_i en deux sous-partitions, où la première sous-partition est caractérisée par le prédicat de sélection A_i in $(SD^i_{j_1}, \cdots, SD^i_{j_p})$ et la seconde sous partition est définie par le prédicat A_i in $Reste_i$. L'algorithme 12 décrit l'implémentation de l'opérateur.

   Exemple 26 *Supposant le schéma de fragmentation SFc de la table Clients présenté dans la figure 2.10.(a). On exécute une évolution du schéma de fragmentation ESF sur SFc :*
   $SFc1 = ESF(SFc) = Add\_A(Job, \{M1, M2\})(SFc)$. *Le nouveau schéma $SFc1$ est donné par la figure 2.11.*
   Nous donnons dans ce qui suit le déroulement de l'algorithme d'évolution :
   $Ens\_Frag = Identifier\_Frag(Job\ in\ Dom(A_i))$
   $= \{Clients1, Clients2, Clients3, Clients4\}$
   $Split(Clients1, Job\ in\ (M1, M2))$
   $Split(Clients2, Job\ in\ (M1, M2))$
   $Split(Clients3, Job\ in\ (M1, M2))$
   $Split(Clients4, Job\ in\ (M1, M2))$

2. $Add\_SD(A_i, \{SD^i_{j_1}, \cdots, SD^i_{j_p}\})$: ajouter un ensemble de sous-domaines nécessite d'identifier le ou les fragments contenant un ou plusieurs sous-domaines parmi l'ensemble $\{SD^i_{j_1}, \cdots, SD^i_{j_p}\}$. Ensuite, chaque fragment identifié est partitionné en deux sous-partitions. La première sous-partition est caractérisée par le prédicat A_i in $(SD^i_{j_1}, \cdots, SD^i_{j_p})$ et la seconde sous-partition est définie par le prédicat A_i in $Reste_i$. L'algorithme 13 décrit l'implémentation de l'opérateur.

   Exemple 27 *Considerons l'évolution de schéma suivante :*
   $SFc2 = ESF(SFc1) = Add\_SD(Ville, \{'Kala'\})(SFc1)$. *Le résultat de l'évolution est donné par la figure 2.12. Les opérations physiques sont données par le déroulement de l'algorithme comme suit :*
   $Ens\_Frag = Identifier\_Frag(Ville =' Kala') = \{Clients3, Clients4\}$
   $Split(Clients3, Ville =' Kala')$
   $Split(Clients4, Ville =' Kala')$

3. $Split\_Dom(A_i, \{SD^i_{j_1}, \cdots, SD^i_{j_p}\}, \{SD^i_{k_1}, \cdots, SD^i_{k_s}\})(SF)$: cette opération permet de partitionner l'ensemble des sous-domaines $\{SD^i_{j_1}, \cdots, SD^i_{j_p}\}$ de l'attribut A_i en deux ensembles de

(a)

Genre	M	F
Ville	Alger, Oran, Blida	$Reste_2$
Metier	M1, M2	$Reste_3$

(b)

Client11

Id	...	Ville	Genre	Metier
10		Alger	M	M1
5		Blida	M	M2

Client21

Id	...	Ville	Genre	Metier
1		Oran	F	M1
12		Oran	F	M2
2		Blida	F	M1

Client12

Id	...	Ville	Genre	Metier
4		Alger	M	M3
6		Oran	M	M3

Client22

Id	...	Ville	Genre	Metier
8		Blida	F	M3

Client3

Id	...	Ville	Genre	Metier
9		Kala	M	M4
3		Kala	M	M3
7		Jijel	M	M5

Client4

Id	...	Ville	Genre	Metier
11		Annaba	F	M4
13		Kala	F	M5

FIGURE 2.11 – (a) $SFc1 : ESF$ sur SFc, (b) Table Clients partitionnée selon $SFc1$

$Add\_SD(A_i, \{SD^i_{j_1}, \cdots, SD^i_{j_p}\})$
Début
  $Ens\_Frag = Identifier\_Frag(A_i \text{ in } (SD^i_{j_1}, \cdots, SD^i_{j_p}))$
  Pour Chaque f dans Ens_Frag **faire**
    Split(f, A_i in $(SD^i_{j_1}, \cdots, SD^i_{j_p})$)
  Fin Pour
Fin

Algorithme 13: Implémentation de l'opérateur Add_SD

$Split\_Dom(A_i, \{SD^i_{j_1}, \cdots, SD^i_{j_p}\}, \{SD^i_{k_1}, \cdots, SD^i_{k_s}\})(SF)$
Début
  $Ens\_Frag = Identifier\_Frag(A_i \text{ in } (SD^i_{k_1}, \cdots, SD^i_{k_s})$
  Pour Chaque f dans Ens_Frag **faire**
    Split(f, A_i in $(SD^i_{k_1}, \cdots, SD^i_{k_s})$)
  Fin Pour
Fin

Algorithme 14: Implémentation de l'opérateur $Split\_Dom$

sous-domaines $\{SD^i_{k_1}, \cdots, SD^i_{k_s}\}$ et $\{SD^i_{j_1}, \cdots, SD^i_{j_p}\} \setminus \{SD^i_{k_1}, \cdots, SD^i_{k_s}\}$ où $\{k_1, \cdots, k_s\} \subset \{j_1, \cdots, j_p\}$. L'éclatement logique d'un ensemble de sous-domaines est exprimé au niveau physique par la fragmentation des partitions, identifiées par le prédicats $(A_i \text{ in } (SD^i_{k_1}, \cdots, SD^i_{k_s}))$, en deux sous partitions comme le montre l'algorithme 14.

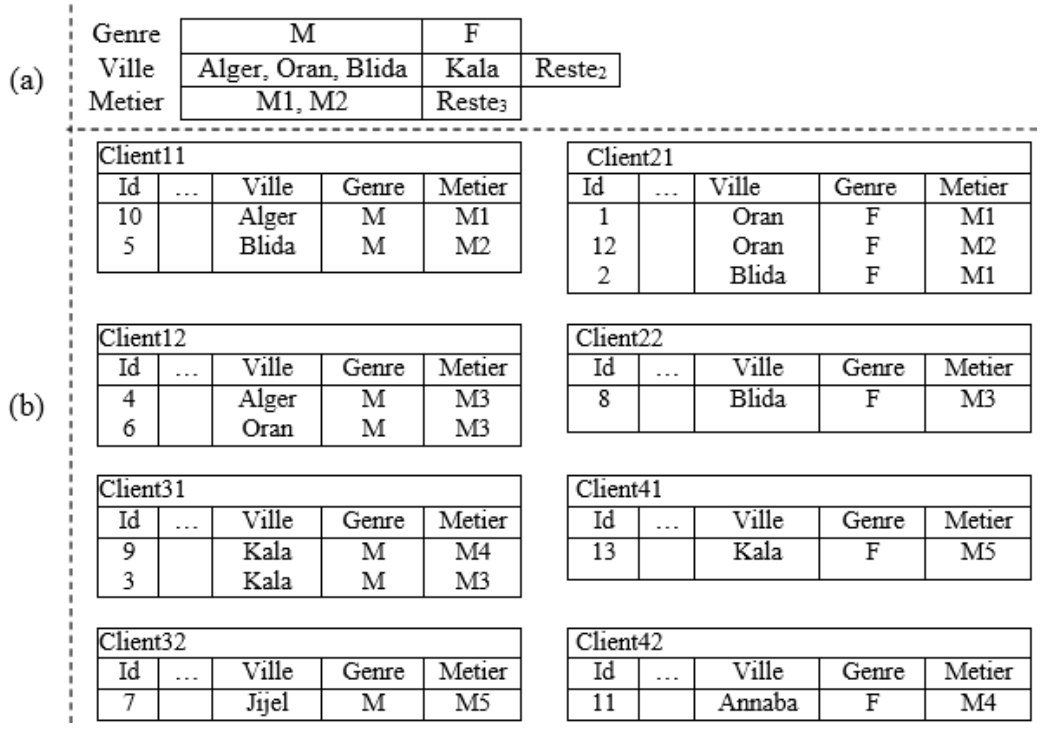


FIGURE 2.12 – (a) $SFc2 : ESF$ sur $SFc1$, (b) Table Clients partitionnée selon $SFc2$

Exemple 28 *Supposant l'évolution suivante :*
$SFc3 = ESF(SFc2) = Split\_Dom(\{'Alger','Oran','Blida'\},\{'Blida'\})(SFc2)$. *Le schéma de fragmentation SFc3 et la répartition des données sont présentés par la figure 2.13. Les opérations physiques à exécuter sont les suivantes :*

$Ens\_Frag = Identifier\_Frag(Ville =' Blida') = \{Clients11, Clients21, Clients22\}$
$Split(Clients11, Ville =' Blida')$
$Split(Clients21, Ville =' Blida')$
$Split(Clients22, Ville =' Blida')$

4. $Merge\_Dom(A_i, \{SD^i_{j_1}, \cdots, SD^i_{j_p}\}, \{SD^i_{k_1}, \cdots, SD^i_{k_s}\})(SF)$: Fusionner les deux ensembles de sous domaines $\{SD^i_{j_1}, \cdots, SD^i_{j_p}\}$ et $\{SD^i_{k_1}, \cdots, SD^i_{k_s}\}$ en un seul ensemble, où $\{j_1, \cdots, j_p\} \subset [1, m_i]$ et $\{k_1, \cdots, k_s\} \subset [1, m_i]$. Cette opération nécessite de fusionner les partitions identifiées par le prédicat $(A_i \text{ in } \{SD^i_{j_1}, \cdots, SD^i_{j_p}\})$ avec celles identifiées par $(A_i \text{ in } \{SD^i_{k_1}, \cdots, SD^i_{k_s}\})$ deux à deux si elles sont fusionnables. L'implémentation de cet opérateur est donnée par l'algorithme 15.

Exemple 29 *Supposant la réduction de schéma suivante :*
$SFc4 = RSF(SFc3) = Merge\_Dom(\{Alger, Oran\}, \{Blida\})(SFc3)$. *Le schéma de fragmentation SFc4 est présenté dans la figure 2.14. Les opérations physiques correspondantes, sont données dans le résultat d'exécution de l'algorithme.*

$Ens\_Frag1 = Identifier\_Frag(Ville \text{ in } (Alger, Oran))$
$= \{Clients111, Clients211, Clients12\}$,
$Ens\_Frag2 = Identifier\_Frag(Ville = Blida)$
$= \{Clients112, Clients212, Clients22\}$,

(a)

Genre	M	F		
Ville	Alger, Oran	Blida	Kala	$Reste_2$
Metier	M1, M2	$Reste_3$		

(b)

Client111

Id	..	Ville	Genre	Metier
10		Alger	M	M1

Client211

Id	...	Ville	Genre	Metier
1		Oran	F	M1
12		Oran	F	M2

Client112

Id	..	Ville	Genre	Metier
5		Blida	M	M2

Client212

Id	...	Ville	Genre	Metier
2		Blida	F	M1

Client12

Id	..	Ville	Genre	Metier
4		Alger	M	M3
6		Oran	M	M3

Client22

Id	...	Ville	Genre	Metier
8		Blida	F	M3

Client31

Id	..	Ville	Genre	Metier
9		Kala	M	M4
3		Kala	M	M3

Client41

Id	...	Ville	Genre	Metier
13		Kala	F	M5

Client32

Id	..	Ville	Genre	Metier
7		Jijel	M	M5

Client42

Id	...	Ville	Genre	Metier
11		Annaba	F	M4

FIGURE 2.13 – (a) $SFc3 : ESF$ sur $SFc2$, (b) Table Clients partitionnée selon $SFc3$

$Merge\_Dom(A_i, \{SD^i_{j_1}, \cdots, SD^i_{j_p}\}, \{SD^i_{k_1}, \cdots, SD^i_{k_s}\})(SF)$
Début
  $Ens\_Frag1 = Identifier\_Frag(A_i \text{ in } (SD^i_{j_1}, \cdots, SD^i_{j_p})$
  $Ens\_Frag2 = Identifier\_Frag(A_i \text{ in } (SD^i_{k_1}, \cdots, SD^i_{k_s})$
  Pour Chaque $f1$ dans Ens_Frag1 **faire**
    Pour Chaque $f2$ dans Ens_Frag2 **faire**
      Si $(Fusionnable(f1, f2, A_i))$ **Alors**
        Merge($f1$,$f2$)
        $Ens\_Frag1 = Ens\_Frag1$ - $\{f1\}$
        $Ens\_Frag2 = Ens\_Frag2$ - $\{f2\}$
      Fin Si
    Fin Pour
  Fin Pour
Fin

Algorithme 15: Implémentation de l'opérateur $Merge\_Dom$

$Fusionnable(Clients111, Clients112, Ville) = Vrai$,
$Merge(Clients111, Clients112)$,
$Ens\_Frag2 = \{Clients211, Clients12\}$,
$Ens\_Frag2 = \{Clients212, Clients22\}$,
$Fusionnable(Clients211, Clients22, Ville) = Faux$,
$Fusionnable(Clients12, Clients212, Ville) = Faux$,
$Fusionnable(Clients12, Clients22, Ville) = Faux$,
$Fusionnable(Clients211, Clients212, Ville) = True$,

(a)

Genre	M	F	
Ville	Alger, Oran, Blida	Kala	$Reste_2$
Metier	M1, M2	$Reste_3$	

(b)

Client11

Id	...	Ville	Genre	Metier
10		Alger	M	M1
5		Blida	M	M2

Client21

Id	...	Ville	Genre	Metier
1		Oran	F	M1
12		Oran	F	M2
2		Blida	F	M1

Client12

Id	...	Ville	Genre	Metier
4		Alger	M	M3
6		Oran	M	M3

Client22

Id	...	Ville	Genre	Metier
8		Blida	F	M3

Client31

Id	...	Ville	Genre	Metier
9		Kala	M	M4
3		Kala	M	M3

Client41

Id	...	Ville	Genre	Metier
13		Kala	F	M5

Client32

Id	...	Ville	Genre	Metier
7		Jijel	M	M5

Client42

Id	...	Ville	Genre	Metier
11		Annaba	F	M4

FIGURE 2.14 – (a) $SFc4$: RSF sur $SFc3$, (b) Table Clients partitionnée selon $SFc4$

$Merge(Clients211, Clients212)$,
$Ens\_Frag1 = \{Clients12\}$, $Ens\_Frag2 = \{Clients22\}$,
$Fusionnable(Clients12, Clients22, Ville) = Faux$

5. $Del\_A(SF, A_i)$: afin de supprimer un attribut d'un schéma de fragmentation, il faut créer un nouveau schéma de fragmentation où tous les sous domaines de l'attribut A_i sont fusionnés en un seul et unique domaine. Pour ce faire, nous exprimons cette opération en utilisant l'opération logique $Merge\_Dom$. Au niveau physique, la suppression de l'attribut nécessite de fusionner toutes les partitions deux à deux suivant cet attribut. L'algorithme 16 implémente la suppression d'un attribut de fragmentation.

Exemple 30 *Supposant la réduction suivante :*
$SFc5 = ESF(SFc4) = Del\_A(Job)(SFc4)$. *Le schéma de fragmentation* $SFc5$ *est présenté dans la figure 2.15. Les opérations physiques à exécuter sont :*
$Ens\_SD = \{\{M1, M2\}, Reste_3\}$
$Merge\_Dom(Job, \{M1, M2\}, Reste_3)$:

$Ens\_Frag1 = Identifier\_Frag(Job\ in\ (M1, M2)) = \{Clients11, Clients21\}$,
$Ens\_Frag2 = Identifier\_Frag(Job\ in\ Reste_3) =$
$\{Clients12, Clients22, Clients31, Clients41, Clients32, Clients42\}$,
$Fusionnable(Clients11, Clients12, Job) = Vrai$,
$Merge(Clients11, Clients12)$,
$Ens\_Frag1 = \{Clients21\}$,
$Ens\_Frag2 = \{Clients22, Clients31, Clients41, Clients32, Clients42\}$,

$Del\_A(SF, A_i)$
Notation
Ens_SD : Ensemble des ensembles de sous-domaines de A_i obtenu à partir schéma SF
Card(Ens) : Cardinalité de l'ensemble Ens
Début
  $Ens\_SD = SF[i]$
  Tant que (Card(Ens_SD)>1) **faire**
    $Ens_1 \longleftarrow Ens\_SD_1$
    $Ens_2 \longleftarrow Ens\_SD_2$
    $Merge\_Dom(A_i, Ens_1, Ens_2)$
    $Ens\_SD \longleftarrow Ens\_SD - \{Ens_1, Ens_2\}$
  Fait
Fin

Algorithme 16: Implémentation de l'opérateur Del_A

(a)

Genre	M	F	
Ville	Alger, Oran, Blida	Kala	$Reste_2$

(b)

Client1

Id	...	Ville	Genre	Metier
10		Alger	M	M1
5		Blida	M	M2
4		Alger	M	M3
6		Oran	M	M3

Client2

Id	...	Ville	Genre	Metier
1		Oran	F	M1
12		Oran	F	M2
2		Blida	F	M1
8		Blida	F	M3

Client31

Id	...	Ville	Genre	Metier
9		Kala	M	M4
3		Kala	M	M3

Client41

Id	...	Ville	Genre	Metier
13		Kala	F	M5

Client32

Id	...	Ville	Genre	Metier
7		Jijel	M	M5

Client42

Id	...	Ville	Genre	Metier
11		Annaba	F	M4

FIGURE 2.15 – (a) $SFc5$: RSF sur $SFc4$, (b) Table Clients partitionnée selon $SFc5$

$Fusionnable(Clients21, Clients22, Job) = Vrai$,
$Merge(Clients21, Clients22)$,
$Ens\_Frag1 = \{\ \}$,
$Ens\_Frag2 = \{Clients31, Clients41, Clients32, Clients42\}$,

6. $Del\_SD(A_i, \{SD^i_{j_1}, \cdots, SD^i_{j_p}\})(SF)$: La suppression de l'ensemble $\{SD^i_{j_1}, \cdots, SD^i_{j_p}\}$ de l'attribut A_i du schéma SF nécessite de l'inclure dans l'ensemble $Reste_i$. Cette opération est équivalente à $Merge\_Dom(A_i, \{SD^i_{j_1}, \cdots, SD^i_{j_p}\}, Reste_i)$.

Exemple 31 *Supposant la réduction suivante :*
$SFc6 = ESF(SFc5) = Del\_SD(Kala) = Merge\_Dom(Ville, \{Kala\}, Reste_2)(SFc5)$. *Le schéma de fragmentation $SFc5$ est présenté dans la figure 2.16. Les opérations physiques à exécuter sont :*

(a)

Genre	M	F
Ville	Alger, Oran, Blida	$Reste_2$

(b)

Client1

Id	...	Ville	Genre	Metier
10		Alger	M	M1
4		Alger	M	M3
6		Oran	M	M3
5		Blida	M	M2

Client2

Id	...	Ville	Genre	Metier
1		Oran	F	M1
12		Oran	F	M2
2		Blida	F	M1
8		Blida	F	M3

Client3

Id	...	Ville	Genre	Metier
9		Kala	M	M4
3		Kala	M	M3
7		Jijel	M	M5

Client4

Id	...	Ville	Genre	Metier
11		Annaba	F	M4
13		Kala	F	M5

FIGURE 2.16 – (a) $SFc6$: RSF vers $SFc5$, (b) Table Clients partitionnée selon $SFc6$

$Ens\_Frag1 = Identifier\_Frag(Ville = Kala) = \{Clients31, Clients41\}$,
$Ens\_Frag2 = Identifier\_Frag(Ville\ in\ Reste_3) = \{Clients32, Clients42\}$,
$Fusionnable(Clients31, Clients32, Ville) = Vrai$,
$Merge(Clients31, Clients32)$,
$Ens\_Frag1 = \{Clients41\}$,
$Ens\_Frag2 = \{Clients42\}$,
$Fusionnable(Clients41, Clients42, Ville) = Vrai$,
$Merge(Clients41, Clients42)$,
$Ens\_Frag1 = \{\ \}$,
$Ens\_Frag2 = \{\ \}$,

2.5 Sélection incrémentale basée sur le Profiling des requêtes

Nous avons proposé deux démarches de sélection incrémentale d'un schéma de fragmentation : sélection incrémentale naïve $FHNI$ et sélection incrémentale basée sur les algorithmes génétiques $FHAG$. Le principal inconvénient de ces démarches est qu'une nouvelle sélection d'un schéma de fragmentation final est déclenchée à l'exécution de chaque nouvelle requête. Or, dans un contexte d'$\mathcal{ED}$, les requêtes décisionnelles présentent des similarités et certaines requêtes ne requièrent pas une refragmentation de l'entrepôt. De plus, l'implémentation d'un nouveau schéma de fragmentation peut être très couteuse en temps et en ressources. Il est donc judicieux d'élaborer une stratégie de contrôle de la refragmentation de l'$\mathcal{ED}$ qui décide s'il est impératif ou non de déclencher un nouveau processus d'optimisation incrémental. Pour ce faire, nous procédons à l'analyse de chaque nouvelle requête exécutée en utilisant notre algèbre de fragmentation AF afin de déterminer les opérations quelle engendre sur le schéma de fragmentation actuel. Cela nous permet de déterminer le profil de la requête. Une fois le profiling effectué, nous procédons à la mise en œuvre d'une nouvelle sélection incrémentale basée sur le profiling des requêtes

2.5.1 Profiling des requêtes basé sur l'Algèbre de Fragmentation

Nous effectuons le Profiling des requêtes décisionnelles en deux étapes. Dans un premier lieu, nous déterminions toutes les opérations algébriques que la requête engendre. Ensuite, suivant la nature des

opérations, nous déterminons le profil de la requête. Pour extraire les opérations algébriques, nous donnons la description générale d'une requête de jointure en étoile comme suit :

```
SELECT *
FROM F, D1, D2, ..., Dd
WHERE F.ID1=D1.ID1 AND F.ID2=D2.ID2 ... AND F.IDd=Dd.IDd
AND (A1 op V11 OR A1 op V12 ... OR A1 op V1k1)
AND (A2 op V21 OR A2 op V22 ... OR A2 op V1k2) ...
AND (An op Vn1 OR An op Vn1 ... OR An op Vnkn)
[ORDER BY ... ]
[GROUP BY ... ]
[HAVING ... ]
```

Un schéma de fragmentation est défini sur les attributs de fragmentation et leurs sous-domaines figurant dans les prédicats de sélection de la forme A_i op V_{ij}, où $1 < i < n$ et $1 < j < k_i$. Chaque valeur V_{ij} peut être égale ou contenue dans un sous-domaines SD^i_j. Par conséquent, l'expression générale d'un prédicat de sélection est donnée par A_i op $\{SD^i_{j_1}, \cdots, SD^i_{j_p}\}$. Considérons une nouvelle requête Q_i exécutée sur un $\mathcal{ED}$ partitionné selon un schéma SF. Nous définissons les opérations algébriques que nous pouvons extraire à partir du prédicat A_i op $\{SD^i_{j_1}, \cdots, SD^i_{j_p}\}$ selon les cas suivants :

- $A_i \notin SF$: l'attribut est ajouté au schéma de fragmentation par l'opération $Add\_A(A_i, \{SD^i_{j_1}, \cdots, SD^i_{j_p}\})(SF)$.
- $A_i \in SF$: nous vérifions si l'ensemble de sous-domaines $\{SD^i_{j_1}, \cdots, SD^i_{j_p}\}$ requièrent des opérations algébriques.
- $\{SD^i_{j_1}, \cdots, SD^i_{j_p}\} \in (SF(i) - \{Reste_i\})$: cet ensemble apparait dans le schéma SF ce qui signifie qu'il participe au processus de fragmentation. aucune opération algébrique n'est requise.
- $\exists \{SD^i_{L_1} \cdots, SD^i_{L_m}\} \in (SF(i) - \{Reste_i\})$ tel que $\{SD^i_{j_1}, \cdots, SD^i_{j_p}\} \subset \{SD^i_{L_1} \cdots, SD^i_{L_m}\}$: l'ensemble $\{SD^i_{L_1} \cdots, SD^i_{L_m}\}$ est éclaté en deux ensembles a travers l'opération $Split\_Dom(A_i, \{SD^i_{L_1}, \cdots, SD^i_{L_m}\}, \{SD^i_{j_1}, \cdots, SD^i_{j_p}\})(SF)$.
- $\exists SubSet_1, \cdots, SubSet_t \in (SF(i) - \{Reste_i\})$ tel que $SubSet_1 \cup SubSet_2 \cup \cdots \cup SubSet_t = \{SD^i_{j_1}, \cdots, SD^i_{j_p}\}$: les t sous-ensembles participent chacun d'entre eux au processus de fragmentation. Ils sont fusionnés comme suit : $Merge\_Dom(A_i, SubSet_1, Merge\_Dom(A_i, SubSet_2, ...Merge\_Dom(A_i, SubSet_{t-1}, SubSet_t)...))(SF)$.
- $\exists \{SD^i_{k_1}, \cdots, SD^i_{k_s}\} \subset \{SD^i_{j_1}, \cdots, SD^i_{j_p}\}$ tel que $\{SD^i_{k_1}, \cdots, SD^i_{k_s}\} \subset Reste_i$: les sous-domaines de l'ensemble $\{SD^i_{k_1}, \cdots, SD^i_{k_s}\}$ ne participent pas au processus de fragmentation. Cet ensemble est ajouté par l'opération $Add\_SD(A_i, \{SD^i_{k_1}, \cdots, SD^i_{k_s}\})(SF)$.
- $\{SD^i_{j_1}, \cdots, SD^i_{j_p}\} \notin (SF(i) - \{Reste_i\})$: en d'autres termes $\{SD^i_{j_1}, \cdots, SD^i_{j_p}\} \subset Reste_i$. L'ensemble est ajouté par l'opération $Add\_SD(A_i, \{SD^i_{j_1}, \cdots, SD^i_{j_p}\})(SF)$.
- Un attribut A_j n'est plus fréquemment utilisé par la charge de requêtes : pour chaque attribut, nous calculons son taux d'utilisation par les requêtes. Si un attribut est utilisé par moins de 20% de la charge, il est supprimé du schéma de fragmentation par l'opération $Del\_A(A_j)(SF)$.
- Un ensemble de sous-domaines $\{SD^j_{R_1}, \cdots, SD^j_{R_h}\}$ n'est plus fréquemment utilisé par la charge de requêtes : si cet ensemble est utilisé par moins de 20% de la charge de requêtes, il est supprimé par l'opération $Del\_SD(A_i, \{SD^j_{R_1}, \cdots, SD^j_{R_h}\})(SF)$.
- Tous les sous-domaines d'un attribut A_j sont fusionnés en un seul ensemble $Reste_j$: ceci se produit si une ou plusieurs opérations $Merge\_Dom()$ et/ou $Del\_SD()$ sont appliquées. Dans ce cas, aucune fragmentation n'est définie sur A_j ce qui requière la suppression de ce dernier par l'opération $Del\_A(A_j)(SF)$.

Une fois toutes les opérations algébriques définies à partir de la requête Q_k, il est possible de savoir si cette requête ne nécessite aucun changement du SF ou si elle va causer une RSF, une ESF ou les deux. De ce fait, nous élaborons quatre profils de requêtes :

1. **Requêtes Évolution :** ce profil regroupe les requêtes qui nécessitent les opérations $Add\_A()$, $Add\_SD()$ et/ou $Split\_Dom()$. Lorsqu'une requête Évolution est exécutée sur l'entrepôt, l'opération ESF est requise. Par conséquent, le nombre de fragments de la table de faits augmente.
2. **Requêtes Réduction :** ce profil décrit les requêtes qui nécessitent les opérations $Del\_A()$, $Del\_SD()$ et/ou $Merge\_Dom()$. Une requête réduction déclenche une RSF du schéma de fragmentation actuel de l'entrepôt. De ce fait, le nombre de fragments de la table de faits diminue.
3. **Requêtes Mixtes :** une requête mixte est une requête Évolution et Réduction au même temps. Elle nécessite les opération $Add\_A()$, $Add\_SD()$, $Split\_Dom()$, $Del\_A()$, $Del\_SD()$ et/ou $Merge\_Dom()$. Par conséquent, le nombre de fragments de la table de faits peut soit augmenter soit diminuer.
4. **Requêtes Neutres :** une requête neutre n'affecte pas le SF actuel. Considérons le prédicat de sélection A_i op $\{SD^i_{j_1}, \cdots, SD^i_{j_p}\}$ d'une requête Q_k. Cette requête est neutre si (1) A_i apparait dans SF et $\{SD^i_{j_1}, \cdots, SD^i_{j_p}\}$ apparait comme un ensemble dans le schéma ou bien (2) si les opérations algébriques extraits n'affectent pas SF. Ce second point se produit si ces opérateurs sont inverses entre eux deux à deux. Nous résumons dans ce qui suit les cas possibles d'opérateurs inverses.
   (a) Del_A et Add_A (resp. Del_SD et Add_SD). Deux opérateurs Del_A (resp. Del_SD) et Add_A (resp. Add_SD) sont obtenues lors de l'analyse de la requête Q_k si l'attribut A_i (resp. l'ensemble $\{SD^i_{j_1}, \cdots, SD^i_{j_p}\}$) n'est plus fréquemment utilisé par la charge de requêtes ce qui requière l'opérateur Del_A (resp. Del_SD) mais un prédicat de sélection est défini sur l'attribut A_i (resp. l'ensemble $\{SD^i_{j_1}, \cdots, SD^i_{j_p}\}$) ce qui nécessite l'opérateur Add_A (resp. Add_SD).
   (b) $Split\_Dom(A_i, Set_1 \cup Set_2, Set_2)$ et $Merge\_Dom(A_i, Set_1, Set_2)$ sont des opérateurs inverses mais ne peuvent être extraits de la même requête. Afin de prouver cette proposition, procédons par absurde. Supposons que les deux opérateurs sont extraits à partir du prédicat A_i op $\{SD^i_{j_1}, \cdots, SD^i_{j_p}\}$ de la même requête. Selon l'analyse faite précédemment, pour obtenir $Split\_Dom(A_i, Set_1 \cup Set_2, Set_2)$, il faut que :
   $Set_2 = \{SD^i_{j_1}, \cdots, SD^i_{j_p}\}$ ..... (1).
   et $Set_2 \subset (Set_1 \cup Set_2)$ .....(2)
   De (1) et (2) on obtient : $Set_1 \neq \emptyset$ .....(2')
   D'un autre coté, l'opération $Merge\_Dom(A_i, Set_1, Set_2)$ est obtenue si :
   $Set_1 \cup Set_2 = \{SD^i_{j_1}, \cdots, SD^i_{j_p}\}$ .....(3)
   De (1) et (3) on obtient : $Set_1 = \emptyset$ .....(3')
   (2') et (3') donne une contradiction ce qui montre par absurde qu'on ne peut extraire d'une même requête les opérations $Split\_Dom(A_i, Set_1 \cup Set_2, Set_2)$ et $Merge\_Dom(A_i, Set_1, Set_2)$.

Exemple 32 *Considérons le schéma de fragmentation SF*1 *donnée par le tableau 2.13 et quatre requêtes. Pour chaque requête nous donnons sa description, son profil et les opérations algébriques requises pour adapter SF1 à l'exécution de chaque requête (tableau 2.14).*

Gender	M	F	
Ville	Alger	Oran	$Reste_2$

TABLE 2.13 – Exemple d'un schéma de fragmentation $SF1$

Requêtes	Opérations Algèbrique	Profil
SELECT * FROM Clients C, Produits P, Ventes V WHERE V.IdC=C.Id And V.IdP=P.Id And C.Ville='Blida' And P.PName='P1'	Add_Dom(Ville, {Blida})(SF1) Add_A(PName, {P1})(SF1)	Evolution
SELECT * FROM Clients C, Ventes V WHERE V.IdC=C.Id And C.Ville='Alger' Or C.Ville='Oran'	Merge_Dom(Ville, {Alger}, {Oran})(SF1)	Reduction
SELECT * FROM Clients C, Produits P, Ventes V WHERE V.IdC=C.Id And V.IdP=P.Id And C.Ville='Alger' Or C.Ville='Oran' And P.PName='P1'	Merge_Dom(Ville, {Alger}, {Oran})(SF1) Add_A(SF1, PName, {P1})	Mixed
SELECT * FROM Clients C, Ventes V WHERE V.IdC=C.Id And C.Ville='Alger'	/	Neutral

TABLE 2.14 – Profils des requêtes et opérations algébriques générées

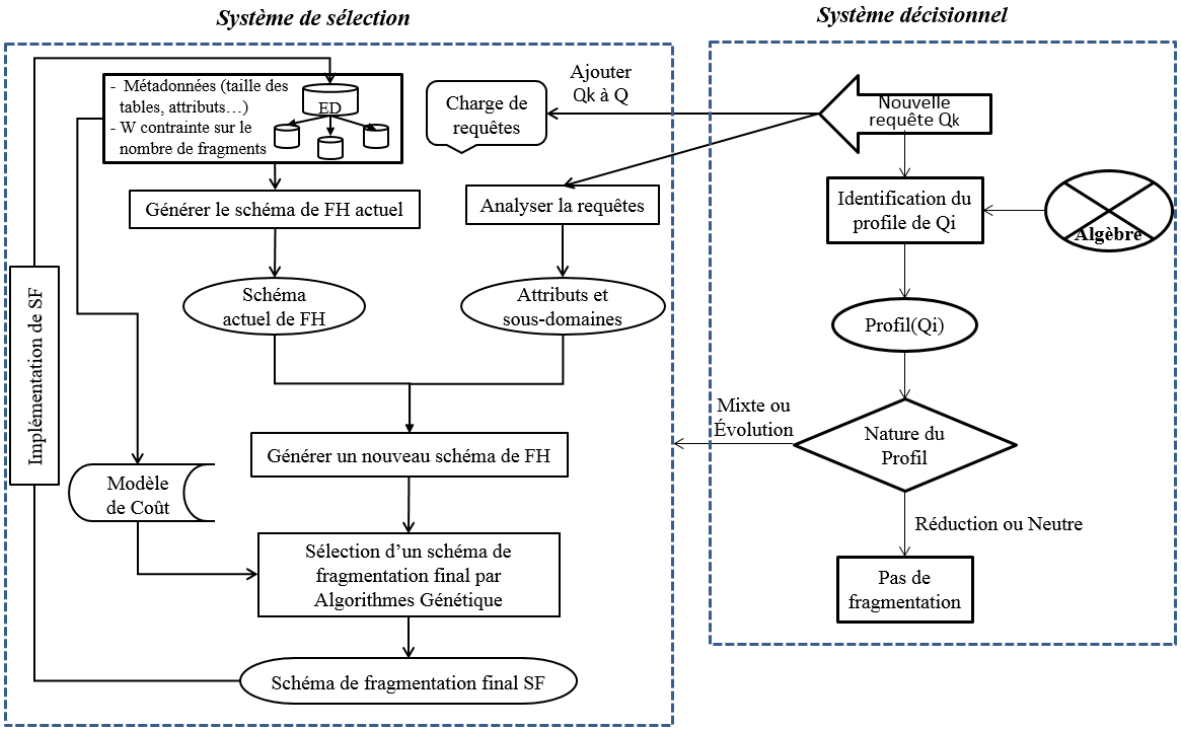


FIGURE 2.17 – Architecture de la sélection incrémentale avec Profiling des requêtes

Algorithme : Sélection incrémentale avec Profiling
Entrée :
Algebre : l'ensemble des opérateurs de l'AF
Q : charge de m requêtes
Q_k : nouvelle requête exécutée
D : ensemble de d tables dimensions $\{D_1, \cdots, D_d\}$
AS : ensemble de n attributs non-clé des tables dimensions $AS = \{A_1, \cdots, A_n\}$
$\mathcal{ED}$: les données et statistiques utilisées dans le modèle de coût
W : contrainte sur le nombre de fragments de la table de faits
Sortie : Schéma final de Fragmentation des tables dimensions FinalSF
Notations :
AnalyseQProfil : retourne le profil d'un requête en se basant sur l'AF
NBfragments : calcul le nombre de fragments de la table des faits généré par un SF
Début
ProfilQ ← AnalyseQProfil(Algèbre, Q_k)
Si (ProfilQ="Neutre") **Alors**
Break ; {*Fin de l'algorithme*}
Fin Si
SF ← GenererActuelSF(D, AS)
$SchemaQ$ ← AnalyserQ(Q_k)
InitSF ← GenererInitSF(SF, $SchemaQ$)
Si (ProfilQ="Évolution" or ProfilQ="Mixte") **Alors**
Si (NBfragments(InitSF) $> W$) **Alors**
FinalSF ← SelectionGA(InitSF, $Q \cup \{Q_i\}$, $\mathcal{ED}$, W)
ImplémenterFinalSF(FinalSF)
Fin Si
Fin Si
End

Algorithme 17: Sélection incrémentale basée sur le Profiling des requêtes

2.5.2 Sélection incrémentale basée sur le Profiling des requêtes

Notre nouvelle proposition est une sélection incrémental d'un schéma de fragmentation basée sur le Profiling des requêtes. Cet proposition utilise la sélection incrémentale basée sur des algorithmes génétiques que nous avons présenté dans la section 2.3. L'architecture de notre proposition est illustrée sur la figure 2.17). Elle est composée de deux systèmes : le système décisionnel et le système de sélection.

1. **Le Système Décisionnel** permet de contrôler le déclenchement du système de sélection lorsqu'une nouvelle requête est exécutée.
2. **Le Système de Sélection** permet de sélectionner un schéma de fragmentation final des tables dimensions en prenant en compte la nouvelle requête exécutée puis d'implémenter physiquement ce schéma sur l'entrepôt.

Le déroulement du processus de sélection est présenté dans l'algorithme 17. Quand une nouvelle requête Q_k est exécutée sur l'$\mathcal{ED}$, le système décisionnel est enclenché. Le profil de la requête est déterminé en fonction de l'AF. Si le profil de la requête est "Neutre" ou "Réduction", aucune nouvelle sélection ou implémentation sur l'$\mathcal{ED}$ n'est nécessaire, car le gain en coût d'exécution des requêtes

sera marginal par rapport au temps nécessaire pour sélectionner et mettre en œuvre un nouveau schéma de fragmentation. Toutefois, si le profil de la requête est "Évolution" ou " Mixte ", le système de sélection est démarré et un nouveau schéma de fragmentation InitSF est calculé. Si le nombre de fragments de la tables des fais que InitSF génère viole la contrainte d'optimisation W, une sélection d'un schéma fragmentation final basée sur les algorithmes génétiques est effectuée. Enfin, le schéma de fragmentation final obtenu est implémenté sur l'$\mathcal{ED}$.

2.6 Expérimentation

Afin de comparer les différentes stratégies de sélection incrémentale d'un schéma de fragmentation, nous avons réalisé des tests de comparaison sur un entrepôt réel issu du benchmark APB1 [40] sous le SGBD Oracle 11g avec une machine Intel Core 2 Duo et une mémoire 2Go. Cet entrepôt est composé d'une table de faits *Actvars*(24 786 000 tuples) et quatre tables de dimension *Prodlevel* (9000 tuples), *Custlevel* (900 tuples), *Timelevel* (24 tuples) et *Chanlevel* (9 tuples). Pour nos tests, nous utilisons une charge de 60 requêtes qui génèrent 18 attributs de sélection *(Line, Day, Week, Country, Depart, Type, Sort, Class, Group, Family, Division, Year, Month, Quarter, Retailer, City, Gender and All)* qui ont respectivement les cardinalités suivantes : 15, 31, 52, 11, 25, 25, 4, 605, 300, 75, 4, 2, 12, 4, 99, 4, 2, 3. Nous effectuons deux parties de tests : dans la première partie, nous comparons les approches de sélection incrémentale d'un schéma de fragmentation que nous avons élaboré à savoir : sélection incrémentale naïve $FHNI$ et sélection incrémentale basée sur les algorithmes génétiques $FHAG$. Dans la seconde partie, nous évaluons l'utilisation du Profiling de requêtes dans la sélection incrémentale et comparons nos propositions avec les travaux existants.

2.6.1 Évaluation des démarches de sélection incrémentale

Afin de comparer entre les démarches $FHNI$ et $FHAG$, nous effectuons des tests théoriques qui se basent sur un modèle de coût mathématique. Ce modèle de coût évalue le coût d'exécution de la charge de requête sur un entrepôt de données partitionné suivant un schéma de fragmentation donné SF. Nos tests se déroulent sur deux phases : d'abords nous réalisons des tests à petite échelle sur une charge de 8 requêtes, par la suite nous menons des tests à grandes échelle sur une charge des 60 requêtes.

2.6.1.1 Tests à petite échelle

Afin de bien comprendre le fonctionnement de la sélection incrémentale, nous avons réalisé une première étude sur une charge de requête de jointure en étoile vide avec une contrainte $W = 40$. L'étude incrémentale est réalisée avec ajout successif de 8 nouvelles requêtes (Tableau 2.15). L'ajout de chaque requête déclenche le processus de sélection d'un SF final. En utilisant le modèle de coût mathématique, nous évaluons chaque SF final en notant le coût d'exécution des requêtes, d'où est calculé le taux d'optimisation du coût (figure 2.18), et le taux des requêtes optimisées (figure 2.19) en utilisant les deux formules citées ci-dessous.

$$\text{Taux d'optimisation du coût} = 1 - \frac{\text{Coût avec optimisation}}{\text{Coût sans optimisation}}$$

$$\text{Taux d'optimisation des requêtes} = \frac{\text{Nombre de requêtes optimisées}}{\text{Nombre total des requêtes}}$$

Requête	Attributs
Q_1	Group
Q_2	Month, Quarter
Q_3	Month, Class
Q_4	City, Gender
Q_5	Month, Year, City, Class
Q_6	Class, Gender
Q_7	City, Gender, Class
Q_8	City, Gender, Group

TABLE 2.15 – Description de la charge des 8 requêtes

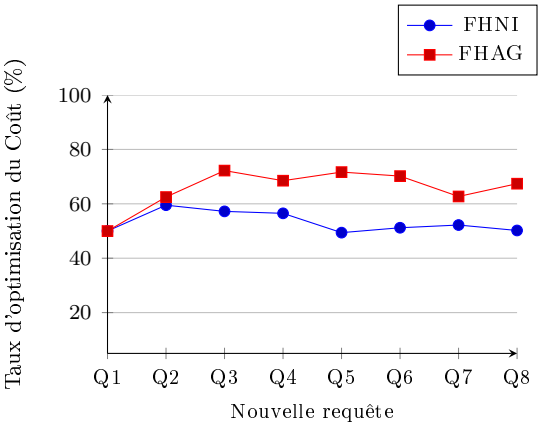


FIGURE 2.18 – Taux d'optimisation du coût d'exécution (cas 8 requêtes)

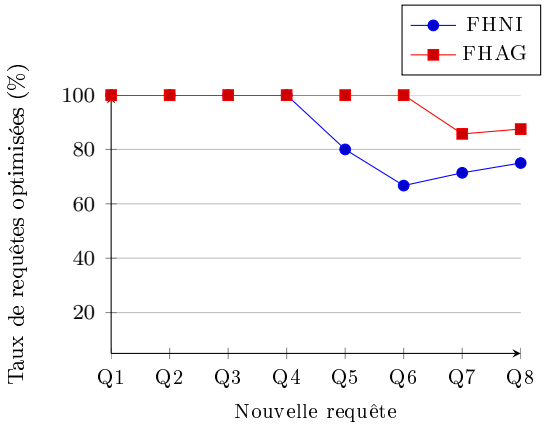


FIGURE 2.19 – Taux de requêtes optimisées (cas 8 requêtes)

Nous remarquons que les meilleurs résultats sont donnés par la démarche $FHAG$. En effet, le coût d'exécution est réduit de 70% pour 96% des requêtes optimisées, alors que pour $FHNI$, le coût d'exécution est réduit de 53% pour 87% des requêtes optimisées. Afin de mieux voir l'efficacité de notre stratégie incrémentale et pour une meilleure comparaison, nous avons effectué des tests à grande échelle.

2.6.1.2 Tests à grande échelle

Considérons une charge de 45 requêtes déjà exécutée sur l'entrepôt de données. Cette charge est optimisée avec un schéma $\mathcal{FH}$ préalablement sélectionné par algorithmes génétiques sous une contrainte $W = 100$. L'étude incrémentale est réalisée avec l'exécution successive de 15 nouvelles requêtes en considérant une contrainte $W = 100$. Pour chaque requête nouvellement exécutée, nous réalisons deux sélections incrémentales : naïve $FHNI$ et basée sur les algorithmes génétiques $FHAG$. Pour chaque sélection ($FHNI$, $FHAG$), et pour chaque nouvelle requête, nous évaluons par modèle de coût mathématique le coût d'exécution de toute la charge de requêtes en cours, à partir duquel est calculé le taux d'optimisation du coût (figure 2.20), et le taux de requêtes optimisées (figure 2.21).

Nous constatons que la démarche de sélection par algorithmes génétiques $FHAG$ apporte de meilleurs résultats que la démarche $FHNI$. En effet, $FHAG$ apporte en moyenne une réduction de

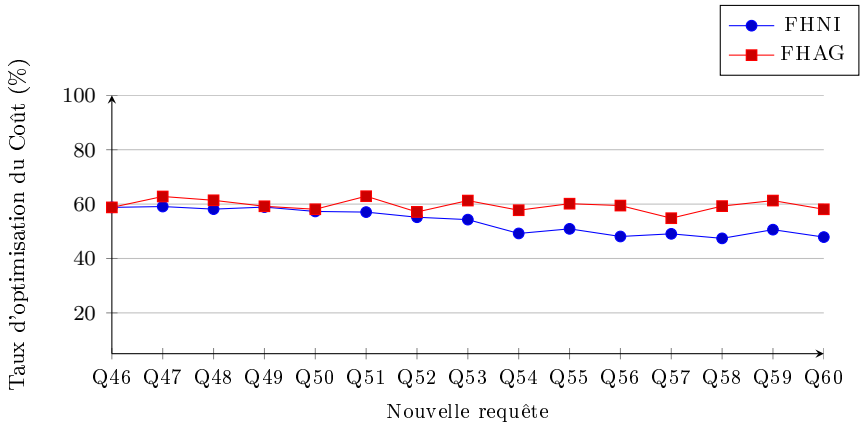


FIGURE 2.20 – Taux d'optimisation du coût : $FHNI$ vs. $FHAG$ (cas 60 requêtes)

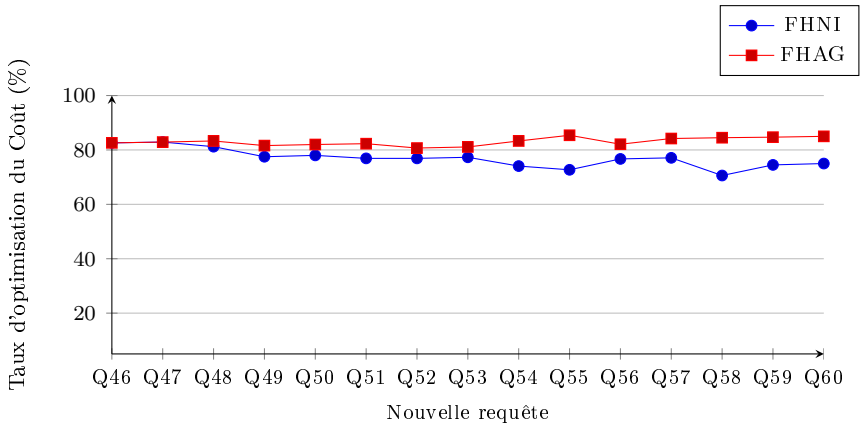


FIGURE 2.21 – Taux de requêtes optimisées : $FHNI$ vs. $FHAG$ (cas 60 requêtes)

60% du coût d'exécution avec 83% en moyen de requêtes optimisées contre 53% de réduction de coût et 77% de requêtes optimisées pour $FHNI$. De plus, $FHNI$ cause une dégradation des performances à fur et à mesure que la charge de requêtes évolue, car son principe de fragmentation incrémentale est basé sur de simples opérations de fusions et éclatements des sous-domaines des attributs. Cela montre bien l'efficacité des algorithmes génétiques à trouver une solution qui optimise continuellement le coût de la charge de requêtes.

2.6.2 Évaluation sous Oracle 11g pour le Profiling des requêtes

Nous avons conduits des expérimentations théoriques et pratiques sous le SGBD Oracle 11g afin d'évaluer l'efficacité de l'intégration du profiling des requêtes basé sur l'algèbre de fragmentation dans le processus de sélection incrémentale. Les tests se déroulent en deux phases : des tests à petites échelles avec 8 requêtes et des tests à grande échelle avec la charge de 60 requêtes. Nous comparons donc les approches suivantes :

- Sélection incrémentale d'un schéma de $\mathcal{FH}$ par algorithmes génétiques $FHAG$.
- Sélection incrémentale d'un schéma de $\mathcal{FH}$ par algorithmes génétiques avec Profiling $AGPQ$.

Elle est composée du système décisionnel et du système de sélection. Lors de l'exécution d'une nouvelle requête, le système décisionnel est déclenché. Si la requête requière une refragmentation de l'entrepôt, un schéma de fragmentation final est sélectionné et implémenté par le système de sélection.

– Sélection incrémentale d'un schéma de fragmentation nommée DD basée sur la conception dynamique d'entrepôts de données distribués proposée dans [92] que nous avons adapté dans un contexte centralisé. Cette approche permet de comparer nos propositions à l'existant.

2.6.2.1 Tests à petites échelle

Requête	Attributs	Profil
Q1	Group	Evolution
Q2	Month, Quarter	Evolution
Q3	Month, Class	Evolution
Q4	City, Gender	Mixed
Q5	Month, Year, City, Class	Neutral
Q6	Class, Gender	Reduction
Q7	City, Gender, Class	Mixed
Q8	City, Gender, Group	Neutral

TABLE 2.16 – Description de la charge de requêtes et profils

Dans cette expérimentation, nous considérons une charge vide à laquelle on ajoute successivement 8 requêtes. La description des requêtes et leurs profils sont donnés dans le tableau 2.16. Sous une contrainte $W = 40$, nous comparons nos approches $AGPQ$ et $FHAG$. Pour la démarche $AGPQ$, les trois premières requêtes Q1, Q2 et Q3 ont un *profil Évolution* et déclenchent le processus de sélection vu que la contrainte W n'est pas encore violée. Les requêtes Q4 et Q7 ont un *profil Mixte* et nécessitent une sélection incrémentale mais les requêtes Q5, Q6 et Q8 ont successivement les profils *Neutre*, *Réduction* et *Neutre* et ne requière pas une refragmentation de l'entrepôt. Pour la démarche $FHAG$, chaque nouvelle requête déclenche le processus de sélection. Pour chaque nouvelle requête et chaque sélection, nous relevons le taux d'optimisation du coût de la charge illustré dans la figure 2.22.

Nous remarquons que le coût d'exécution pour les deux démarches $AGPQ$ et $FHAG$ est globalement le même. En effet, le profiling des requêtes n'influence pas la qualité de la solution choisie par le processus de sélection. En conséquence, nous comparons les deux sélections selon *le temps de Maintenance*. Le temps de maintenance représente le temps nécessaire pour implémenter un SF sur un entrepôt fragmenté sous Oracle 11g. Après l'exécution de chaque requête et pour chaque sélection ($FHAG$ et $AGQP$), nous implémentons sous Oracle 11g le nouveau schéma de fragmentation sélectionné et nous notons le temps de maintenance. Les résultats sont donnés dans la figure 2.23. Pour la sélection $FHAG$, chaque requête nécessite une sélection et la mise en œuvre d'un nouveau schéma de fragmentation, le temps global de maintenance après l'exécution des 8 requêtes est 36 minutes. Pour la sélection $AGPQ$, les requêtes avec les profils *Réduction* et *Neutre* ne déclenchent pas une nouvelle sélection incrémentale, donc aucun changement ne se produit sur l'$\mathcal{ED}$. Le temps global de maintenance est 14 minutes. Par conséquent, la démarche $AGPQ$ réduit le temps global de maintenance par 60%, en comparaison avec la sélection $FHAG$.

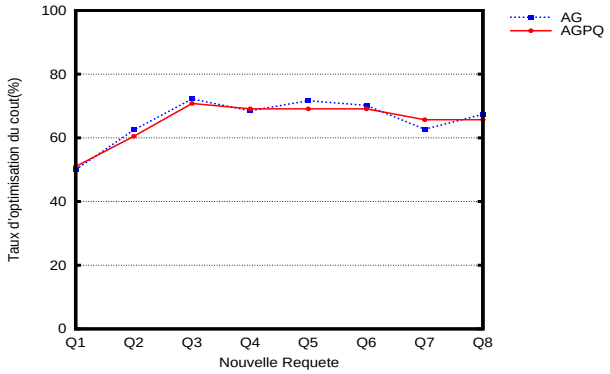


FIGURE 2.22 – Taux l'optimisation du coût (cas 8 requêtes avec Profiling)

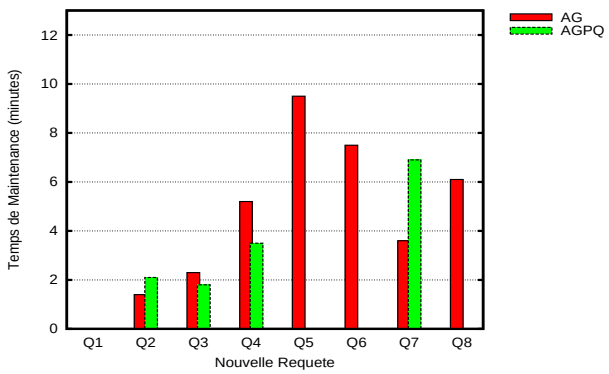


FIGURE 2.23 – Temps de maintenance sous Oracle11g (cas 8 requêtes avec Profiling)

2.6.2.2 Tests à grande échelle

Queries	Profile
Q46, Q47, Q49, Q55, Q56, Q57, Q60	Neutre
Q48, Q50, Q52, Q51,	Mixte
Q58, Q59	Évolution
Q53, Q54	Réduction

TABLE 2.17 – Description de la charge de requêtes et profils

Pour ce test, nous considérons la charge de 45 requêtes préalablement exécutée sur l'entrepôt de données avec $W = 100$ et 15 nouvelles requêtes successivement exécutées sur l'entrepôt dont les profils sont donnés par le tableau 2.17. Nous considérons les trois sélections $FHAG$, $AGPQ$ et DD. Pour chaque sélection et chaque nouvelle requête, nous notons le taux l'optimisation du coût des requêtes exécutées (figure 2.24) et le temps de maintenance sous Oracle11g (figure 2.25). Le temps de maintenance représente le temps nécessaire pour implémenter le nouveau schéma sélectionné par chaque approche et pour chaque requête.

Selon les résultats donnés par la figure 2.24, les coûts de la charge de requêtes obtenus par chacune des trois sélections sont similaires. D'abord l'algorithme utilisé pour la sélection d'un schéma de fragmentation dans les trois sélections se base sur les travaux cités dans [14]. Deuxièmement, le profiling des requêtes n'affecte pas la qualité d'un schéma de fragmentation sélectionné. Cependant, lorsque l'on analyse les résultats de la figure 2.25, nous remarquons que la sélection $AGPQ$ donne un temps de maintenance réduit par rapport aux approches $FHAG$ et DD. En effet, les temps de maintenance globaux de $FHAG$ et DD sont respectivement 320 minutes (5h 20mn) et 339 minutes (5h 39mn), alors que le temps de maintenance globale de $AGPQ$ est de 166 minutes (2h 48mn) ce qui représente une réduction de 52% du temps de maintenance. Contrairement aux sélections $FHAG$ et DD où une nouvelle sélection est déclenchée après l'exécution de chaque requête, pour $AGPQ$ et sur les 15 requêtes, seules 6 requêtes nécessitent une nouvelle sélection incrémentale (profil Mixte et Évolution). Par conséquent, selon le paramètre important qui est le temps de maintenance, l'emploi du profiling des requêtes donne un meilleur rendu par rapport aux approches $FHAG$ et DD. Notons

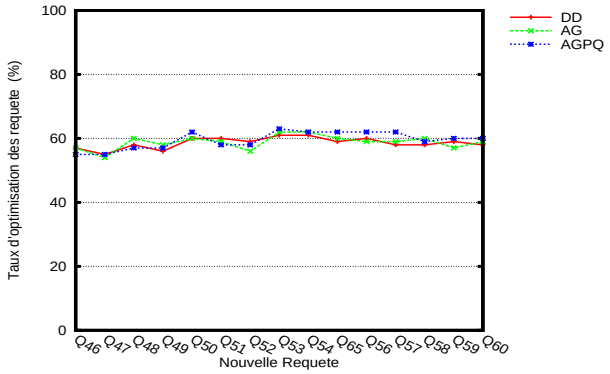


FIGURE 2.24 – Taux l'optimisation du coût (cas 60 requêtes avec Profiling)

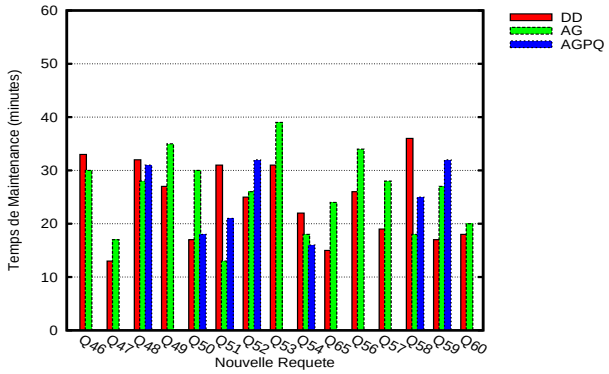


FIGURE 2.25 – Temps de maintenance sous Oracle11g (cas 60 requêtes avec Profiling)

que le temps de maintenance global pour une approche de sélection représente la somme de tous les temps de maintenance sur les 15 requêtes, en d'autre terme le temps de déroulement de l'étude incrémentale sur cette sélection.

2.7 Conclusion

Dans ce chapitre, nous avons défini une sélection incrémentale d'un schéma de fragmentation dans le contexte d'entrepôt de données en étoile. Nous avons proposé une sélection incrémentale d'un schéma de $\mathcal{FH}$ par algorithmes génétiques qui, contrairement à la sélection statique, permet de prendre en considération l'évolution de la charge de requêtes. Deux stratégies de sélection ont été proposées : (1) la sélection naïve $FHNI$ qui permet, à travers des opérations simples, d'adapter le schéma de fragmentation de l'entrepôt aux changements survenus et (2) la sélection par algorithme génétique $FHAG$ effectue la sélection d'un nouveau schéma de fragmentation à la base du schéma courant de l'$\mathcal{ED}$ et des changements survenus. Nos stratégies de sélection incrémentale ont été validées par une étude expérimentale. Les résultats obtenus montrent que la sélection incrémentale $FHAG$ donne une meilleure optimisation de la charge de requêtes grâce à l'emploi des algorithmes génétiques.

Nous avons par la suite formalisé les opérations nécessaires, pour adapter un schéma de fragmentation à l'évolution de charge, sous forme d'une algèbre de fragmentation qui manipule une structure de données flexible codant un schéma de fragmentation. A la base de cette algèbre, nous avons défini un Profiling des requêtes qui permet de déterminer le profil de chaque nouvelle requête et dire si cette requête nécessite effectivement d'altérer le schéma de fragmentation actuel de l'entrepôt. Nous avons donc défini une nouvelle démarche de sélection incrémentale basée sur le Profiling des requêtes et l'algèbre de fragmentation $AGPQ$ et avons montré à travers une étude expérimentale que cette démarche permet de réduire considérablement le temps de maintenance d'un entrepôt fragmenté (temps requis pour sélectionner et implémenter un nouveau SF) lors de l'évolution de charge.

Chapitre 3

Sélection incrémentale jointe des IJB simples et multiples

Sommaire

3.1 Introduction

Dans la littérature, plusieurs travaux proposent des démarches de sélection d'index de jointures binaires. Certains proposent des démarches de sélection d'index simples [16] d'autre définissent des processus de sélection d'index simples et multiples [2, 9, 3]. Les travaux [2, 9, 16] proposent une sélection statique des index alors que les auteurs dans [3] proposent de mettre à jour la configuration des index lorsque la charge de requêtes évolue. Dans ce travail, nous avons proposé un algorithme statique de sélection des index de jointure binaires basé sur les algorithmes génétiques où nous considérons un seul type d'index à la fois. A la base de cet algorithme, nous avons proposé une démarche de sélection incrémentale des index de jointures binaires qui permet de maintenir à jour la configuration d'index implémentée sur l'entrepôt.

Nous avons constaté que les travaux de sélections qui combinent les index simples et multiples [2, 9, 3] ne font pas la distinction entre les deux types index durant le processus de sélection. Les espaces de recherche des deux index sont combinés en un seul espace très complexe ce qui représente un handicap majeur pour les index simples. En effet, l'espace de recherche des index multiples est considérablement plus complexe que celui des index simples ce qui désavantage ces derniers durant le processus de sélection. Il faut donc proposer une nouvelle vision de sélection qui considère chaque type d'index comme une technique d'optimisation TO et permet d'étudier les différences et similarités entre les index simples et multiples et l'avantage de chaque type d'index, dans le but de proposer une sélection jointe. Nous avons aussi remarqué que seuls les travaux dans [3] proposent une approche de sélection incrémentale des deux types d'index de jointures binaires. Nous avons par conséquent fait les propositions suivantes :

1. Étudier la similarité et différence entre les index simples et multiples.
2. Proposer une démarche de sélection statique jointe des index simple et index multiples.
3. Proposer une démarche de sélection incrémentale jointe des index simples et multiples.

La proposition d'une démarche de sélection statique jointe des index simples et index multiples a été effectuée dans le cadre du projet de fin d'étude de Abdelmalek Kessaissia et Tarek Zergat, étudiants à l'école nationale supérieure d'informatique (ESI), encadré par Rima Bouchakri et Dr. Kamel Boukhalfa. Le travail a fait l'objet d'une soutenance à l'issu de laquelle les étudiants ont obtenu le diplôme d'ingénieur en informatique avec la mention Très bien. Ce chapitre contient les sections suivantes. La section 2 explique la démarche de sélection statique jointe des index de jointure binaires simples et multiples. La section 3 présente la démarche de sélection incrémentale jointe des deux types d'index. Dans la section 4, nous présentons les tests expérimentaux théoriques et pratiques effectués sous le SGBD Oracle 11g. La section 5 conclue l'article.

3.2 Sélection statique jointe des index simples et multiples

Les index simples et multiples présentent des avantages et des inconvénients l'un par rapport à l'autre. Ils sont donc complémentaires d'où la nécessité d'étudier la sélection jointe des deux types d'index. Nous présentons dans ce qui suit les avantages d'un type par rapport à l'autre :

1. *Avantage des index simples IS par rapport aux index multiples IM* : vu le nombre important des attributs d'indexations, les IM définis sur ces attributs peuvent être très volumineux et être complètement écartés par le processus de sélection d'index, s'ils violent la contrainte d'espace de stockage. Contrairement à cela, il est possibles de définir des IS qui occupent un espace

moins important que les index IM ce qui augmente la probabilité de les sélectionner dans la configuration finale d'index.

2. *Avantage des index multiples IM par rapport aux index simples IS* : Un index multiple défini sur deux attributs d'une requête est moins volumineux que deux index simples définis sur les deux attributs de la même requête. Effectivement, chaque index comporte une colonne supplémentaire constituant l'identifiant *RowID* nécessitant 8 à 16 octets dans le SGBD Oracle, par exemple. De même, un index multiple couvrant les attributs de toute la requête, pré-calcul ces jointures et l'optimise d'avantage.

Exemple 33 *Soit un $\mathcal{ED}$ avec une table de faits Ventes (20 millions de tuples) et trois tables dimension Clients, Temps et Produits. Soit une charge de requêtes permettant de définir les attributs indexables avec les cardinalités suivantes : Ville(V :150), Pays(P :30), Année(A :20), PNom(N :400), Mois(M :12), Jour(J :31). Ces attributs candidats à l'indexation sont simplifiés comme suit : $AS = \{V, P, A, Pn, M, J\}$. Rappelons que l'espace de stockage requis pour un index IJB_j, définit sur n_j attributs, avec $1 \leq n_j \leq n$ où n est le nombre total d'attributs indexables, est calculé par la formule suivante :*

$$Storage(IJB_j) = (\frac{\sum_{k=1}^{n_j} |A_k|}{8} + 16) \times |F| \tag{3.1}$$

Le tableau 3.1 résume l'espace de stockage requis par quelques exemples d'index simples et multiples. Supposant une contrainte $S = 1go$ représentant l'espace maximum alloué pour le stockage des index.

Index	Taille (go)	Type
I_JMAVPn	1.8	Multiple
I_JMA	0.45	Multiple
I_J	0.38	Simple
I_M	0.33	Simple
I_A	0.35	Simple

TABLE 3.1 – Exemples d'index simples et multiples et leurs tailles

Nous reprenons l'analyse faite précédemment et nous l'appliquons sur l'exemple comme suit :

1. *IS vs. IM : Selon la contrainte $S = 1go$, l'index I_JMAVPn de taille $1.8go$ est toujours écarté par le processus de sélection. Par contre des index simples définis sur les mêmes attributs que constituent cet index (V,P,A,Pn,M,J) peuvent être sélectionnés comme I_J, I_M ou I_A.*
2. *IM vs. IS : l'index multiple I_JMA de taille $0.45go$ est moins volumineux que les trois index simples I_J, I_M et I_A dont la taille globale est de $1.06go$. Ces trois index ne peuvent jamais être sélectionnés au même temps avec $S = 1go$. Ainsi, l'index I_JMA est plus avantageux pour optimiser une requête définie sur ces trois attributs.*

Nous allons présenter dans cette section notre approche de sélection jointe des index simples et des index multiples. Nous commençons par présenter le problème de sélection jointe. Par la suite nous proposons une démarche de sélection qui se base sur les algorithmes génétiques.

3.2.1 Problème de sélection jointe

Nous allons présenter dans ce qui suit le la formalisation du problème de sélection jointe des index simples et des index multiples. Par la suite, nous allons étudier la complexité du problème.

3.2.1.1 Formalisation

Durant la phase de conception de l'entrepôt de données, les index simples et multiples sont implémentés sur l'entrepôt afin d'optimiser les requêtes décisionnelles. Ils sont définis sur les attributs indexables extraits des requêtes. Le problème de sélection jointe des index simples et des index multiples $PSJIJB$ est formalisé comme suit :
Étant donné :

- un $\mathcal{ED}$ modélisé par un schéma en étoile ayant d tables de dimension $\mathcal{D} = \{D_1, D_2, \cdots, D_d\}$ et une table des faits $\mathcal{F}$;
- une charge de requêtes $\mathcal{Q} = \{Q_1, Q_2, \cdots, Q_m\}$ à partir de laquelle un ensemble d'attributs indexables $AS = \{A_1, \cdots, A_n\}$ est obtenu ;
- un espace de stockage d'index $\mathcal{S}$.

Le problème de sélection jointe $PSJIJB$ consiste à sélectionner une configuration jointe d'index simples et multiples CI_f sur l'ensemble AS tel que :

- Le coût d'exécution de la charge $\mathcal{Q}$ en présence de CI_f est optimisé.
- L'espace de stockage alloué pour les index de CI_f ne dépasse pas $\mathcal{S}$ ($Taille(CI) \leq \mathcal{S}$).

Le $PSJIJB$ est un problème de sélection d'index avec contrainte et est connu NP-Complet [42]. De ce fait, il est impératif d'employer des algorithmes non exhaustifs afin de rechercher la solution optimale (quasi-optimale).

3.2.1.2 Complexité

L'espace de recherche pour la sélection jointe des IS et IM représente la combinaison des deux espaces de recherches de chaque type d'index. En considérons l'ensemble des attributs indexables $\mathcal{AS} = \{A_1, A_2, \cdots, A_n\}$, nous rappelons la taille de l'espace de recherche pour chaque type d'index comme suit :

1. *Taille de l'espace de recherche pour les $\mathcal{IJB}$ simples* : en considérant une sélection d'index simples, chaque attribut de AS peut donner lieu à l'implémentation d'un IS ce qui donne n IS possibles. La taille de l'espace de recherches des IS, représentant le nombre total de configurations d'index simples, est donné par la formule suivante :

$$NbConfig_{IS} = 2^n - 1 \tag{3.2}$$

2. *Taille de l'espace de recherche pour les $\mathcal{IJB}$ multiples* : en considérant une sélection d'index multiples, un IM est construit sur un sous-ensemble des n attributs indexables. Donc, le nombre total d'IM qu'on peut définir est donné par la formule $Nb_{IM} = 2^n - n - 1$. Ainsi, le nombre de toutes les configurations d'index multiples possibles est donné par l'équation suivante :

$$NbConfig_{IM} = 2^{Nb_{IM}} - 1 = 2^{2^n - n - 1} - 1 \tag{3.3}$$

L'espace de recherche de la sélection jointe englobe les deux espaces de recherche simple et multiple. Donc pour un ensemble de n attributs candidats à l'indexation, le nombre d'IJB possibles est égale à la somme des index simples (n index) et des index multiples ($2^n - n - 1$ index) ce qui donne :

$$Nb_{IS\&IM} = 2^n - n - 1 + n = 2^n - 1. \tag{3.4}$$

Par conséquent, le nombre de toutes les configurations d'index possibles pour une sélection jointe est donné par la formule suivante :

$$NbConfig_{IS\&IM} = 2^{Nb_{IS\&IM}} - 1 = 2^{2^n - 1} - 1 \tag{3.5}$$

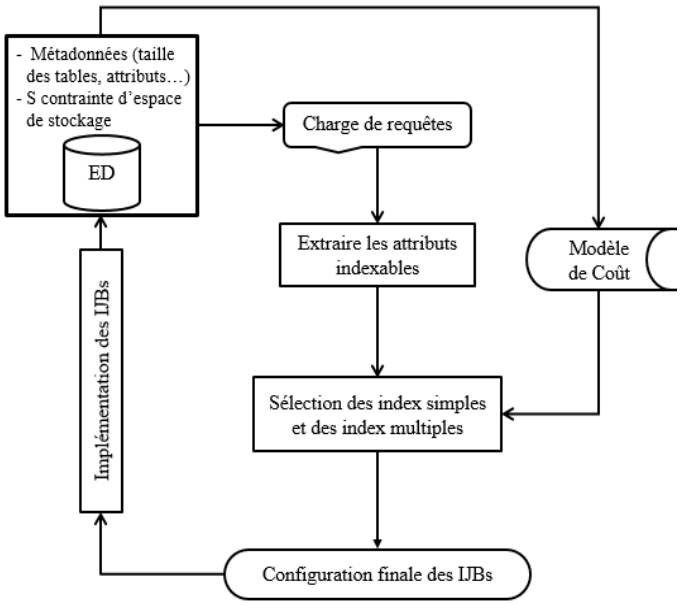


FIGURE 3.1 – Architecture générale de sélection jointe des index simples et multiples

Exemple 34 *Considérons les attributs {Ville, Genre, Mois}={V, G, M}. Le nombre total d'index simples et multiple est* $2^3 - 1 = 7$ *index qui sont : V, G, M, VG, VM, GM, VGM. Le nombre de configurations d'index pour la sélection jointe est* $2^7 - 1 = 127$ *qui sont : {V}, {G}, {V, G}, {VM}, {VG, VM}, {V, GM}, {V, G, GM}, {G, VM, GM}, {V, G, VM, GM, VGM}, etc.*

3.2.2 Notre démarche de sélection jointe

Pour effectuer la sélection des index simples et multiples, nous avons proposée une sélection jointe intégrée dont l'architecture est illustrée sur la figure 3.1. Elle est composée de trois étapes. (1) Extraire les attributs indexables non clé figurant dans la clause WHERE des requêtes analysées. (2) Sélectionner une configuration d'index simples et multiples en employant un modèle de coût. (3) Implémenter la configuration finale d'index obtenue sur l'entrepôt de donnée. L'analyse de cette proposition nous conduit aux constats suivants :

1. la taille de l'espace de recherche pour la sélection jointe et considérablement complexe par rapport à la taille de l'espace de recherche pour la sélection isolée de chaque type index.

   Exemple 35 *Considérons l'exemple 34. D'un côté, le nombre d'index total est égal à 7, par conséquent le nombre de configurations pour la sélection jointe est 127. D'un autre côté, le nombre d'index simples est* $n = 3$*, ce qui fait que le nombre de configurations d'index simples est* $2^3 - 1 = 7$*. Également, le nombre d'index multiples est* $2^3 - 3 - 1 = 4$ *ce qui donne* $2^4 - 1 = 15$ *configurations d'index multiples. Nous remarquons que le nombre de configuration d'un seul espace joint entre les index simples et multiples (127 configuration) est très grand par rapport aux deux espaces des IS et IM (7 et 15).*

2. La sélection jointe sur un espace de recherche joint est très désavantageuse aux IS étant donné que, pour n attributs indexables, la taille de l'espace de recherche pour les IM est très grand

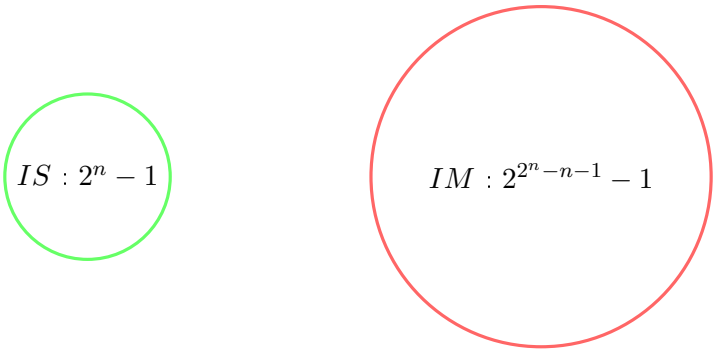


FIGURE 3.2 – Espace de recherche pour IS vs. Espace de recherche pour IM

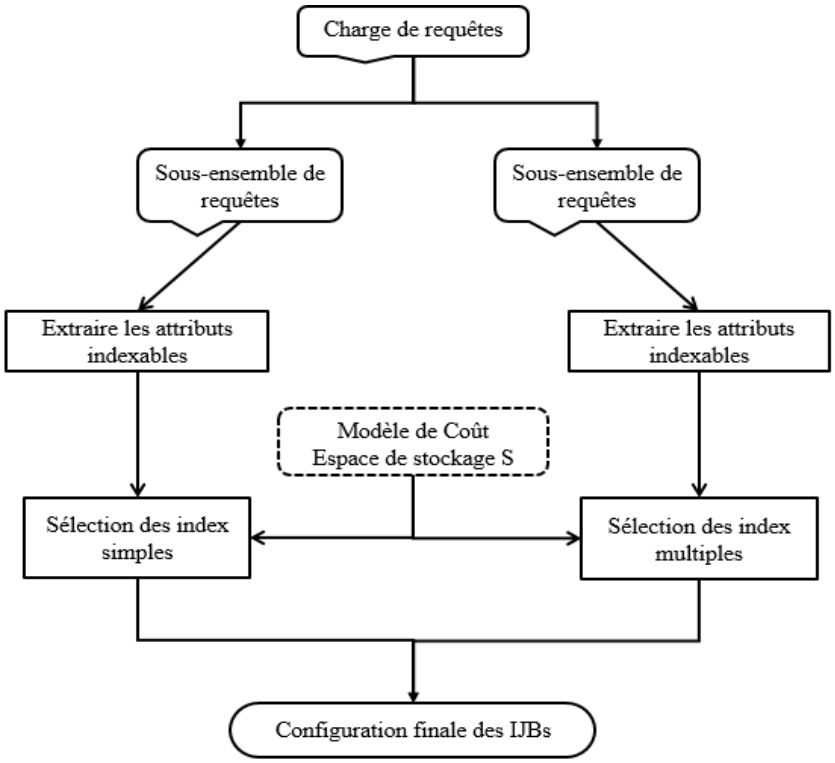


FIGURE 3.3 – Architecture de sélection jointe par séparation de l'espace de recherche

par rapport à la taille de l'espace pour les IS. La combinaison des deux espaces en un seul espace de recherche réduit la probabilité de sélectionner un IS par rapport à un IM. La figure 3.2 illustre la différence en taille entre les deux espaces de recherches.

3. La dépendance entre les IS et IM est faible-faible. En effet, implémenter un type d'index n'influe pas sur la structure des index du second type ; qu'on implémente les index simples puis multiples ou inversement va aboutir au même ensemble d'index final. Ainsi, nous pouvons appliquer la sélection jointe itérative des index simples et des index multiples dans n'importe quel ordre.

Selon l'analyse effectuée ci-dessus, nous adoptons une sélection jointe itérative des deux techniques d'optimisation (index simples et index multiples). Nous séparons les deux espaces de recherche en séparant les deux algorithmes de sélection de IS d'un côté et de IM de l'autre. Chaque algorithme permet de sélectionner un type d'index. Les deux algorithmes de sélection se basent sur le même modèle de coût qui guide la sélection vers la configuration optimale d'index simples d'un côté et multiples de l'autre. Les deux configurations obtenues sont fusionnées en une seule configuration d'index qui sera implémentée sur l'entrepôt. Cependant, comme l'illustre bien la figure 3.3, la séparation des deux sélections de IS et de IM conduit à une concurrence pour les mêmes ressources :

1. Les deux sélections sont concurrentes sur la même charge de requêtes.
2. Les deux sélections utilisent le même modèle de coût, elles sont donc concurrentes sur le même espace de stockage.

Nous allons aborder dans ce qui suit, le problème de concurrence des deux sélections d'index sur les deux ressources : charge de requêtes et espace de stockage. Ensuite, nous présentons les algorithmes de sélections basés sur les algorithmes génétiques.

3.2.3 Problème de partage des requêtes

Après séparation des deux sélections d'index, les index simples et les index multiples sont concurrents sur la même ressource qui est la charge de requêtes. Partager l'ensemble de requêtes entre les index simples les index multiples doit satisfaire le problème global de la sélection jointe cité dans le paragraphe 3.2.1.1. Le problème de partage des requêtes PPR est formalisé comme suit :
Étant donné :

- un $\mathcal{ED}$ modélisé par un schéma en étoile ayant d tables de dimension $\mathcal{D} = \{D_1, D_2, \cdots, D_d\}$ et une table des faits $\mathcal{F}$;
- une charge de requêtes $\mathcal{Q} = \{Q_1, Q_2, \cdots, Q_m\}$ à partir de laquelle un ensemble d'attributs indexables $AS = \{A_1, \cdots, A_n\}$ est obtenu ;
- un espace de stockage d'index $\mathcal{S}$.

Le problème de partage des requêtes PPR consiste à répartir les requêtes en deux classes : ClasseIS et ClasseIM tel que la configuration d'index jointe CI_f, qui représente l'union entre la configuration d'index simples sélectionnée sur ClasseIS (CIS_f) et la configuration d'index multiples sélectionnée sur ClasseIM (CIM_f), vérifie ce qui suit :

- le coût d'exécution de la charge $\mathcal{Q}$ en présence de CI_f est réduit,
- l'espace de stockage alloué pour les index de CI_f ne dépasse pas $\mathcal{S}$ ($Taille(CI) \leq \mathcal{S}$).

Le PPR est un problème de Classification de l'ensemble de requêtes Q en deux classes ClasseIS et ClasseIM largement abordé dans le domaine de Datamining. Une classe peut être vide comme elle peut comporter m requêtes. Le nombre de classifications possibles est égale à 2^m. Énumérer tous les cas de classification possibles et pour chaque cas exécuter deux algorithmes de sélection, un sur chacune des classes, est une tâche impossible. Par conséquent, nous proposons de développer une approche de classifications moins couteuse et moins complexe basée sur la classification en Datamining.

3.2.3.1 Principe générale de classification par Datamining

La classification de données est un problème de Datamining qui vise à extraire des connaissances à partir d'un ensemble de données. Elle peut être supervisée ou non supervisée. La classification supervisée consiste à affecter les individus d'un ensemble à des classes ayant une signification et une interprétation précise. Dans notre problème de partage, nous nous basons sur la classification non

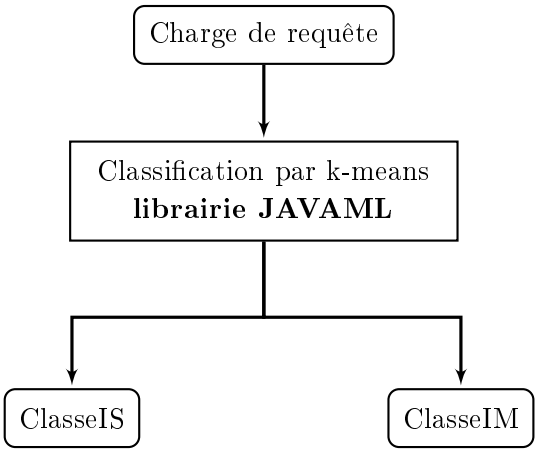


FIGURE 3.4 – Principe de classification des requêtes par k-means

supervisée qui vise à regrouper les individus en classes homogènes telle que les individus similaires appartiennent à la même classe [63]. Parmi les techniques de classification non supervisée les plus simples et les plus utilisées, on trouve l'algorithme k-means [55]. Il partitionne un ensemble de données en k classes non chevauchées. Par conséquent, nous développons une approche de partage de requêtes moins coûteuse qui se base sur la classification non supervisée par k-means. Notre choix s'est porté sur l'algorithme k-means pour les raisons suivantes :

- L'algorithme k-means est un algorithme de classification simple à utiliser et répond bien à nos besoins de classification. Effectivement, nous visons à effectuer une classification non supervisée des requêtes en 2 classes disjointes.
- Deux travaux existent dans la littérature qui emploient l'algorithme k-means dans une démarche d'optimisation des entrepôts de données [56, 17]. Les auteurs dans [56] emploi k-means pour le partitionnement de l'ensemble des prédicats de sélection définis sur les attributs de sélection en sous-ensembles formant ainsi des fragments horizontaux. Dans nos travaux [17] nous avons proposé une démarche de sélection statique combinée des index de jointures binaires et d'un schéma de fragmentation. Nous avons utilisé l'algorithme k-means afin de classifier les attributs de sélection extraits de la charge de requêtes en deux classes : sur la première classe nous effectuons la sélection des index de jointure binaires et sur la seconde classe nous appliquons la sélection d'un schéma de fragmentation. Ces deux travaux permettent de montrer la force et la simplicité de l'emploi de l'algorithme k-means.
- Il existe des Framework JAVA qui permettent d'implémenter l'algorithme k-means, comme la librairie JAVA appelée JAVAML (Java Machine Learning Library) [6], ou encore la classe JavaSimpleKMeans du logiciel Weka défini dans [58].

La figure 3.4 illustre le principe général de classification de la charge de requêtes par k-means. Afin d'être en mesure d'appliquer l'algorithme pour classifier les requêtes, nous définissons deux critères de classification. Ensuite, nous présentons le principe général de l'algorithme k-means et son application à notre problème de sélection.

6. http ://java-ml.sourceforge.net/

3.2.3.2 Critères de classification des requêtes

Afin de décider s'il est préférable d'indexer les attributs d'une requête donnée par IS ou IM, il faut prendre en compte la nature des requêtes qui dépend de deux facteurs importants, à savoir *la cardinalité de la requête* et *le support de chaque requête.*

1. **Cardinalité de la requête CardReq** : la cardinalité d'une requête représente la somme des cardinalités des attributs de sélection figurants dans la clause WHERE de la requête. Les index multiples définit sur des requêtes à forte cardinalité sont volumineux et peuvent ne pas être sélectionnés par le processus de sélection des index s'ils violent la contrainte sur l'espace de stockage. Par conséquent, *les requêtes de faible cardinalité sont plus adaptées pour sélectionner les IM.*
2. **Support des requêtes SupReq** : le support d'une requête représente le nombre d'attributs de sélection figurants dans la clause WHERE. Sur la base de cette définition, nous prenons en compte deux points importants. D'abords, un index multiple qui couvre tous les attributs de sélection de la requête pré-calcule toutes les jointures de la requête. Ensuite, les requêtes dont SupReq=1 ont un seul attribut et définissent donc des IS. Pour cela et afin d'éviter l'affectation des requêtes qui contiennent un seul attribut vers la classe IM, *les requêtes de faible support sont plus adaptées pour sélectionner les IS.*

Pour conclure, nous favorisons pour les IS les requêtes dont la cardinalité est grande et dont le support est petit, contrairement aux IM aux quels on attribue les requêtes ayant une faible cardinalité et un support grand.

3.2.3.3 Déroulement de l'algorithme k-means

Soit à partitionner un ensemble de m données $\mu_1, \cdots, \mu_m$ en k classes $C_1, \cdots, C_k$. Ces données sont représentées sous forme de m points dans un espace $\mathbb{R}^n$. On positionne dans l'espace $\mathbb{R}^n$ k centroïdes $Ct_1, \cdots, Ct_k$ généralement choisis parmi les données elles même. L'algorithme commence avec k classes où une classe contient un centroïde. A chaque itération, chaque donnée est affectée à la classe dont elle est la plus proche en estimant la distance euclidienne entre cette donnée et le centroïde de la classe. La distance euclidienne entre une donnée μ_i dont les coordonnées sont $(\mu_1^i, \cdots, \mu_n^i)$ et un centroïde Ct_j avec les coordonnées $(Ct_1^j, \cdots, Ct_n^j)$ est donnée par la formule suivante :

$$\sqrt{\sum_{p=1}^{n}(\mu_p^i - Ct_p^j)^2} \tag{3.6}$$

Ensuite, k nouveaux centroïdes sont déterminés. Chaque nouveau centroïde Ct_j est calculé à partir des t données $\mu_1, \cdots, \mu_t$ déjà affectées à la classe C_j. Chaque coordonnée Ct_p^j du centroïde Ct_j est calculée par la formule suivante :

$$Ct_p^j = \frac{\sum_{i=1}^{t} \mu_p^i}{t} \tag{3.7}$$

L'algorithme s'arrête lorsqu'on atteint un certain nombre prédéfini d'itérations ou que deux itérations successives conduisent au même résultat. k-means vise ainsi à réduire les distances intra-classes (distances entre les données de chaque classe). Pour appliquer le k-means à notre problème de partage des requêtes, nous considérons que les données à classifier sont les requêtes à partitionner en deux ensembles : ClasseIS et ClasseIM, ainsi k=2. Il faut également représenter les requêtes dans un espace $\mathbb{R}^n$. Pour ce faire, chaque requête est munie des coordonnées (x,y)=(CardReq, SupReq). Enfin, pour le

bon déroulement de l'algorithme k-means nous choisissons deux centroïdes bien définis. Pour ClasseIS nous choisissons la requête avec une cardinalité maximale et un support minimal. Inversement pour ClasseIM, nous choisissons la requête avec une cardinalité minimale et un support maximal. Pour l'implémentation de l'algorithme, nous utilisons la librairie JAVA open source JAVAML. Cette librairie dispose de la classe "javaml.clustering.KMeans" qui permet d'exécuter l'algorithme k-mean et la classe "javaml.distance.EuclideanDistance" utilisée pour déterminer la distance euclidienne comme distance entre les données à classifier.

Exemple 36 *Considérons une charge de 10 requêtes à classifier. Nous donnons dans le tableau 3.2 la cardinalité et le support des 10 requêtes. Afin d'exécuter le k-means, nous choisissons comme centroïde pour la ClasseIS la requête Q8(550, 1) et pour ClasseIM la requête Q2(55, 5). La représentation graphique des requêtes et le résultats de classification par k-means sont illustrés dans la figure 3.5. La figure montre une répartition des requêtes en deux Classe. Les requêtes Q_1, Q_3, Q_4, Q_8, Q_9 vont constituer la ClasseIS pour la sélection des IS. Les requêtes $Q_2, Q_5, Q_6, Q_7, Q_{10}$ vont constituer la ClasseIM pour la sélection des IM.*

	SupReq	CardReq
Q1	1	50
Q2	5	55
Q3	2	480
Q4	1	200
Q5	5	475
Q6	4	110
Q7	3	234
Q8	1	550
Q9	3	600
Q10	3	100

TABLE 3.2 – Support et Cardinalité de 10 requêtes

3.2.4 Problème de partage de l'espace de stockage

La sélection des index simples et celle des index multiples utilisent le même modèle de coût et sont donc concurrentes sur le même espace de stockage. Vu que les deux sélections sont effectuées séparément, il faut répartir l'espace de stockage entre elles. Le problème de partage de l'espace de stockage $PPES$ en deux espaces est formalisé comme suit : Soit :

- un $\mathcal{ED}$ modélisé par un schéma en étoile ayant d tables de dimension $\mathcal{D} = \{D_1, D_2, \cdots, D_d\}$ et une table des faits $\mathcal{F}$;
- une charge de requêtes $\mathcal{Q} = \{Q_1, Q_2, \cdots, Q_m\}$ à partir de laquelle un ensemble d'attributs indexables $AS = \{A_1, \cdots, A_n\}$ est obtenu ;
- un espace de stockage d'index $\mathcal{S}$.

Le $PPES$ consiste à répartir l'espace de stockage S en deux espaces S1 et S2 (S = S1+S2) où S1 est utilisé pour la sélection des IS et S2 pour la sélection des IM, tel que la configuration d'index finale CI_f, composée des deux configurations CIS_f et CIM_f issues des deux sélections précédentes, vérifie ce qui suit :

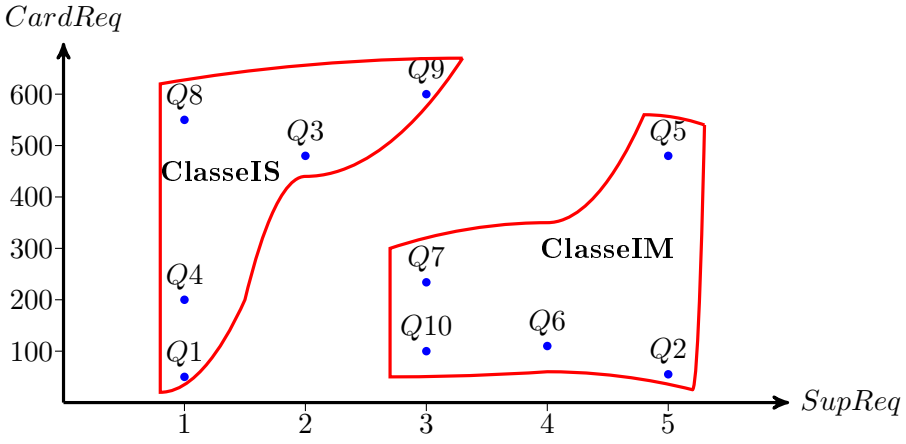


FIGURE 3.5 – Classification par k-means de la charge des 10 requêtes

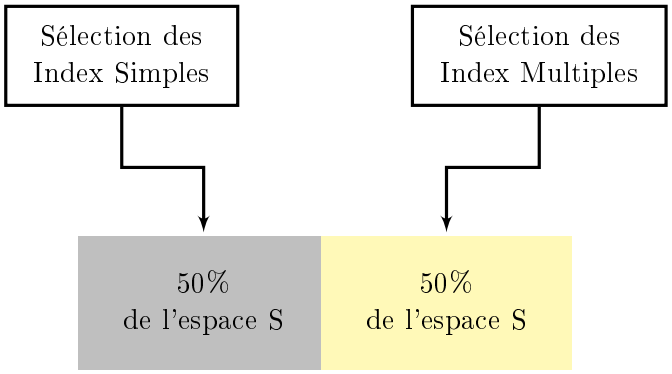


FIGURE 3.6 – Partage naïf de l'espace de stockage

– le coût d'exécution de la charge $\mathcal{Q}$ en présence de CI_f est réduit,
– l'espace de stockage alloué pour les index de CI_f ne dépasse pas $\mathcal{S}$ ($Taille(CI) \leq \mathcal{S}$).

Le problème de partage de l'espace de stockage a déjà été abordé dans le contexte d'optimisation des entrepôts de données. Les auteurs dans [15] proposent une démarche de sélection jointe des index de jointures binaires et des vues matérialisées avec partage de l'espace de stockage entre les deux sélections. La démarche de partage se base sur la mise en œuvre d'un espion pour chaque technique d'optimisation. L'espion des index (resp. des vues) vérifie continuellement si un espace de stockage est disponible au niveau des vues (resp. les index). Si oui, l'espace obtenu est ajouté à l'espace de stockage des index (resp. des vues).

Pour résoudre notre problème de partage de l'espace de stockage entre les index simples et les index multiples, nous proposons dans un premier lieu *un Partage Naïf*. Ensuite, nous proposons *un Partage de l'espace dirigé par la classification des requêtes* qui s'adapte à la nature des requêtes de chaque classe (ClasseIS et ClasseIM) afin de définir l'espace de stockage a allouer pour chaque sélection d'index.

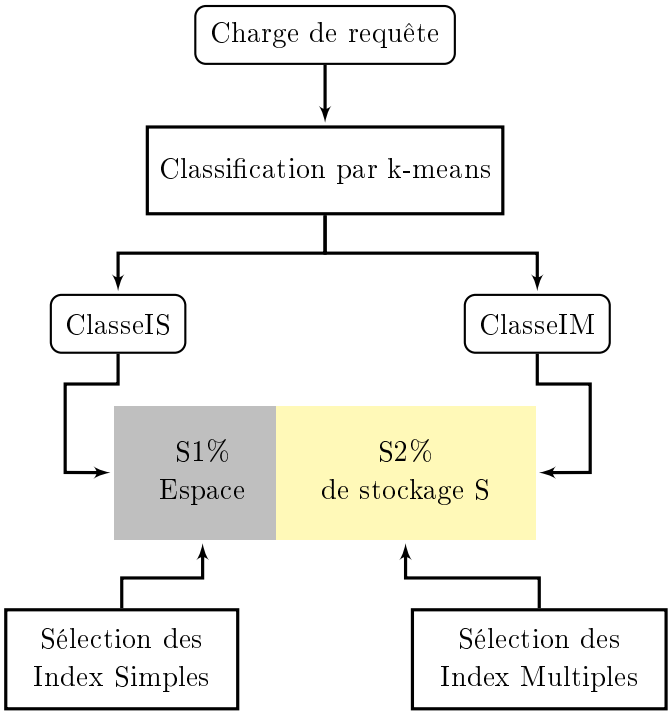


FIGURE 3.7 – Partage de l'espace de stockage dirigé par requêtes

3.2.4.1 Partage naïf de l'espace de stockage

Le partage naïf de l'espace de stockage consiste à partitionner S en deux sous espaces égaux comme suit :

$$S1 = S2 = \frac{S}{2} \tag{3.8}$$

Ce partage signifie que 50% de l'espace de stockage est réservé pour la sélection des IS, et 50% pour la sélection des IM (figure 3.6). L'inconvénient du partage naïf est qu'il est impartial car il ne prend pas en compte le besoin réel en espace de stockage de chaque sélection. En effet, l'espace de stockage employé pour la sélection et l'implémentation des index dépend fortement de la taille de l'espace de recherche pour cette sélection et la nature des requêtes. Ainsi, nous avons défini un partage de l'espace de stockage qui prend en compte ces facteurs.

3.2.4.2 Partage de l'espace de stockage dirigé par requêtes

La taille de l'espace de recherche de chaque sélection d'index est déterminée par le type d'index à sélectionner (simple ou multiple), le nombre d'attributs indexables issus de la charge de requêtes et enfin la cardinalité de ces attributs (la cardinalité d'un attribut influe sur la taille de l'index définit sur cet attribut). On conclut que pour notre sélection jointe, deux facteurs sont importants afin d'effectuer le partage de l'espace de stockage : le résultat du partage des requêtes et la cardinalité des attributs indexables de chaque classe d'index. Ainsi, nous avons défini un critère de partage qui est *la cardinalités de classe.*

La cardinalité d'une classe représente la somme des cardinalités de ses requêtes. En d'autre terme, la somme des cardinalités des attributs indexables issu de cette classe. Soit une classification de

la charge $\mathcal{Q} = \{Q_1, Q_2, \cdots, Q_m\}$ comme suit : ClasseIS=$\{Q_{j_1} \cdots, Q_{j_s}\}$ et ClasseIM=$\{Q_{k_1} \cdots, Q_{k_p}\}$, où $s + p = m$. La cardinalité de ClasseIS est donnée par la formule suivante :

$$CardinaliteClasseIS = \sum_{i=j_1}^{j_s} CardReq_i \tag{3.9}$$

Où $CardReq_i$ est la cardinalité de la requête Q_i. Également, la cardinalité de ClasseIM est calculée comme suit :

$$CardinaliteClasseIM = \sum_{i'=k_1}^{k_p} CardReq_{i'} \tag{3.10}$$

A la base de cette définition, nous définissons l'espace de stockage pour une classe donnée comme la proportion de la cardinalité de cette classe par rapport à la cardinalité totale de la charge de requêtes (figure 3.7). Nous calculons dans ce qui suit l'espace alloué pour les IS et les IM :

L'espace alloué pour les IS=$\frac{(CardinalitClasseIS)}{(CardinalitClasseIS + CardinalitClasseIM)}$× S

L'espace alloué pour les IM =$\frac{(CardinalitClasseIM)}{(CardinalitClasseIS + CardinalitClasseIM)}$× S

Exemple 37 *Considérons la charge des 10 requêtes décrites dans l'exemple 36. La cardinalité de chaque classe est calculée comme suit :*

- *CardinalitéClasseIS = CardReq1+CardReq3+CardReq4+CardReq8+CardReq9 = 50+480+200+550+600=1880*
- *CardinalitéClasseIM = CardReq2+CardReq6+CardReq10+CardReq7+CardReq5 = 55+110+100+234+475=974*

Enfin l'espace de stockage alloué pour chaque classe est calculé comme suit :

- *L'espace alloué pour les IS =1880/2854 × S=66% × S*
- *L'espace alloué pour les IM =974/2854 × S=34% × S*

3.2.5 Algorithmes de sélection

Afin de réaliser la sélection des index simples et la sélection des index multiples, nous utilisons notre démarche de sélection statique des index de jointure binaires définie dans le chapitre 1 et basée sur les algorithmes génétiques. Afin d'employer les algorithmes génétiques, nous avons défini une fonction objectif, basée sur un modèle de coût mathématique, et un codage d'une configuration d'index en chromosome et cela pour les index simples et les index multiples. Notre démarche de sélection jointe des index simples et multiples est illustrée sur la figure 3.8. Le processus de sélection, décrit par l'algorithme 18 est réalisé en plusieurs étapes comme suit :

1. Extraire la charge de requêtes.
2. Classifier par k-means la charge de requêtes en deux ensembles ClasseIS et ClasseIM.
3. Partager l'espace de stockage S en deux espaces S1 et S2 en se basant sur la classification précédemment obtenue.
4. Sélectionner par algorithmes génétiques une configuration finale d'index simples CIS_f sur la classes de requêtes ClasseIS sous la contrainte d'espace de stockage S1.

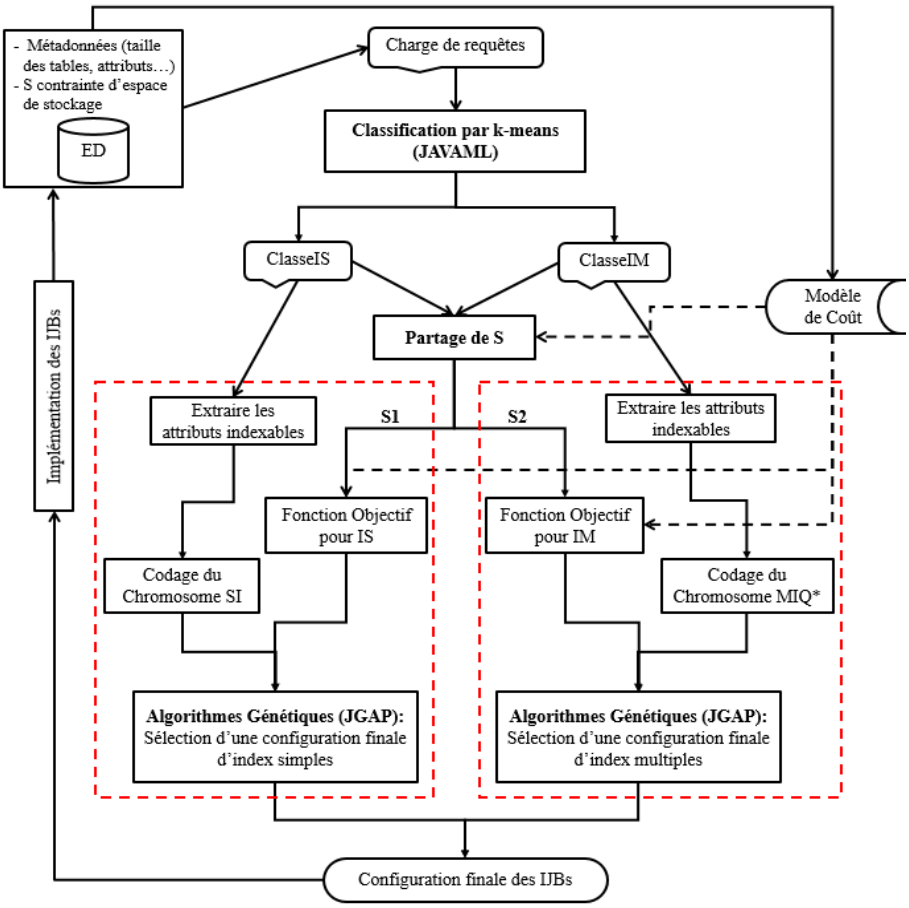


FIGURE 3.8 – Architecture de sélection jointe des index simples et multiples basée sur les AG

5. Sélectionner par algorithmes génétiques une configuration finale d'index multiples CIM_f sur la classes de requêtes ClasseIM sous la contrainte d'espace de stockage S2.
6. Implémenter sur l'entrepôt de données la configuration finale d'index $CI_f = CIS_f \cup CIM_f$.

Nous résumons dans les paragraphes suivants les deux démarches de sélections d'index.

3.2.5.1 Sélection des index simples

Pour la sélection des index simples, nous choisissons le codage du chromosome SI. Soit $AS = \{A_1, \cdots, A_n\}$ l'ensemble des attributs indexables. Le chromosome représente un tableau de taille n dont les cellules sont binaires, où chaque cellule fait référence à un attribut. Une cellule vaut 1 si un index simple est défini sur l'attribut correspondant. Sinon, elle vaut 0. Supposant l'ensemble des attributs indexables $AS = \{Ville, Mois, Anne, Pays, Jour\}$. Le tableau 3.3 décrit le codage en chromosome d'une configuration de trois index simples
$Config_{Ci} = \{I\_City, I\_Country, I\_Day\}$.
Pour guider l'algorithme génétique dans la sélection d'une configuration d'index simples $Config_{ci}$

Algorithme de sélection jointe d'IS et IM
Entrées :

Q : charge de m requêtes

S : espace de stockage des $\mathcal{IJB}$

$\mathcal{ED}$: données relatives au modèle de coût (taille des tables, page système, etc.)

Sortie : Configuration finale d'index CI_f. **Notations :**

KMeans : algorithme k-means implémenté à travers la librairie JAVAML.

PartagerS : répartir l'espace de stockage des index entre les deux sélections.

JGAP : API JAVA qui permet d'implémenter l'algorithme génétique.

$ChromIS$, $ChromIM$: chromosome pour la configuration d'index simples resp. multiples.

CoderChromosome : Coder le chromosome selon le codage SI ou MIQ^*.

$FitnessIS$, $FitnessIM$: fonction objectif pour l'AG.

CalculFitness : établir la fonction objectif.

Début

KMeans(Q, ClasseIS, ClasseIM) ;

PartagerS(ClasseIS, ClasseIM, S, S1, S2) ;

$ChromIS \longleftarrow$ CoderChromosome(ClasseIS, SI) ;

$ChromIM \longleftarrow$ CoderChromosome(ClasseIM, MIQ^*) ;

$FonctionIS \longleftarrow$ CalculFitness(S1, $\mathcal{ED}$, ClasseIS) ;

$FonctionIM \longleftarrow$ CalculFitness(S2, $\mathcal{ED}$, ClasseIM) ;

$CI_f \longleftarrow$ JGAP($ChromIS$, $FonctionIS$) $\cup$ JGAP($ChromIM$, $FonctionIM$) ;

Fin

Algorithme 18: Algorithme de sélection statique jointe des Index simples et multiples

I_Ville	I_Mois	I_Année	I_Pays	I_Jour
1	0	0	1	1

TABLE 3.3 – Codage de chromosome pour les index simples SI

contenant N_{ci} index, la fonction objectif a été définie comme suit :

$$F'(Config_{ci}) = \begin{cases} Cost(Q, Config_{ci}) \times Pen(Config_{ci}) & \text{si } Pen(Config_{ci}) > 1 \\ Cost(Q, Config_{ci}) & \text{sinon} \end{cases}$$

avec :

$$Pen(ConfigI_{ci}) = \frac{storage(Config_{ci})}{S1} = \frac{(\frac{|A_j|}{8} + 16) \times |F|}{S1} \tag{3.11}$$

où $|A_j|$ et $|F|$ représentent respectivement la cardinalité de l'attribut A_j et la taille de la table de fait F, et :

$$Cost(Q, Config_{ci}) = \sum_{i=1}^{q} \sum_{j=1}^{N_{ci}} log_m(|A_j|) - 1 + \frac{|A_j|}{m-1} + d\frac{||F||}{8PS} + ||F||(1 - e^{-\frac{N_r}{||F||}}) \tag{3.12}$$

où $||F||$, Nr, PS, m et d sont resp. Le nombre de pages occupées par la table F, le nombre de tuples accédés par le jème index, la taille d'une page, l'ordre du B-arbre qui définit l'index simples et le nombre de vecteurs bitmaps utilisés pour évaluer Q_i.

3.2.5.2 Sélection des index multiples

Pour la sélection des index multiples, plusieurs codages de chromosomes ont été définis. Nous choisissons le codage le plus bénéfique à savoir le codage amélioré basé sur les requêtes MIQ^*. Dans un premier lieu, nous effectuons un élagage de l'espace de recherche des index multiples dirigé par les requêtes. Nous retenons comme index candidats à la sélection uniquement ceux qui couvrent exactement les attributs de chaque requête. Afin de palier au problème de génération d'index multiples volumineux qui pourraient violer l'espace de stockage, nous ajoutons pour chaque requête les sous-index multiples pouvant être définis sur ses attributs. Par exemple, pour une requête contenant les attributs $\{Ville, Mois, Anne\}$, l'index multiple total est I_VMA et les sous-index sont I_VM, I_VA et I_MA. Une fois les index et sous-index multiples définis, nous construisons le chromosome sous forme d'un tableau binaire où chaque case référence un index. Si une case est à 1, l'index correspondant est sélectionné, 0 sinon. Le tableau 3.4 montre un exemple d'un chromosome par codage MIQ^* sur un ensemble $AS = \{Ville, Mois, Anne, Pays, Jour\}$=$\{V, M, A, P, J\}$. Le codage du chromosome donne une configuration de quatre index multiples $Config_{Ci} = \{I\_VM, I\_VA, I\_AP, I\_PJ\}$.

I_VMA	I_VM	I_VA	I_MA	I_APJ	I_AP	I_AJ	I_PJ
0	1	1	0	0	1	0	1

TABLE 3.4 – Codage de chromosome pour les index multiples $MIQ*$

La fonction objectif qui permet de guider l'algorithme de sélection a été définie comme suit :

$$F'(Config_{ci}) = \begin{cases} Cost(Q, Config_{ci}) \times Pen(Config_{ci}) & \text{si } Pen(Config_{ci})>1 \\ Cost(Q, Config_{ci}) & \text{sinon} \end{cases}$$

avec :

$$Pen(ConfigI_{ci}) = \frac{storage(Config_{ci})}{S2} = \frac{(\frac{\sum_{k=1}^{n_j} |A_k|}{8} + 16) \times |F|}{S2} \quad (3.13)$$

et :

$$Cost(Q, Config_{ci}) = \sum_{i=1}^{q} \sum_{j=1}^{N_{ci}} log_m(\sum_{k=1}^{n_j} |A_k|) - 1 + \frac{\sum_{k=1}^{n_j} |A_k|}{m-1} + d\frac{||F||}{8PS} + ||F||(1 - e^{-\frac{Nr}{||F||}}) \quad (3.14)$$

3.3 Sélection incrémentale jointe des index simples et multiples

La sélection incrémentale jointe des index simples et multiples vise à mettre à jour les index actuellement implémentés sur l'entrepôt lorsque celui-ci évolue par évolution de charge et évolution d'instances. L'évolution de charge peut causer l'ajout et/ou suppression des attributs d'indexation ou encore le changement de leurs fréquences d'utilisation. Quant à l'évolution d'instances, elle cause l'augmentation de la taille des tables particulièrement la table de faits, ce qui augmente la taille des $\mathcal{IJB}$. Par conséquent, nous proposons une démarche de sélection jointe des index simples et multiples afin de continuellement mettre à jour les index implémentés sur l'entrepôt.

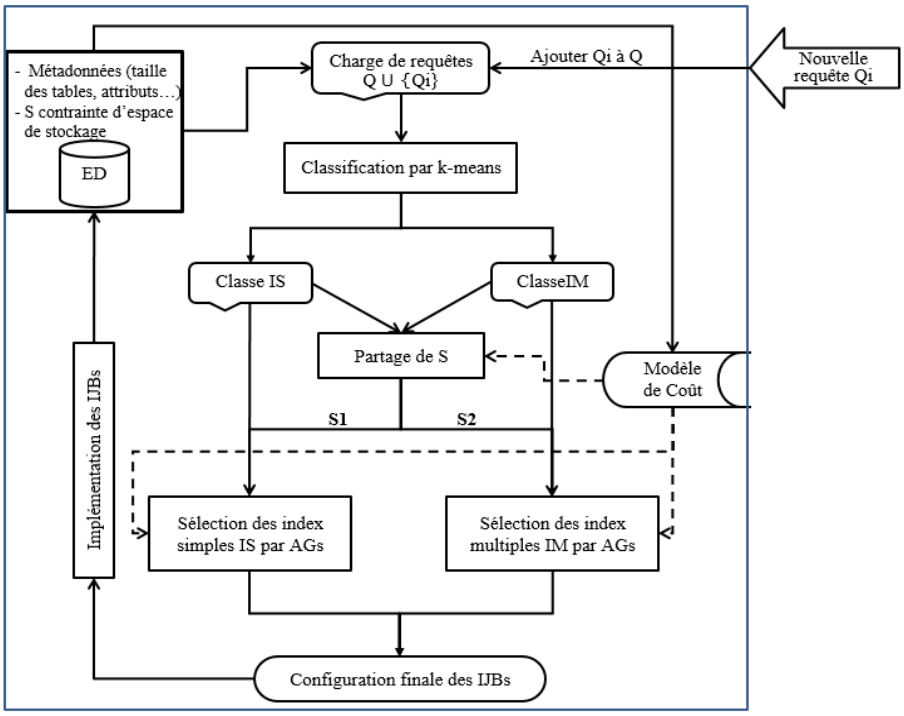


FIGURE 3.9 – Processus naïf de sélection incrémentale jointe des index simples et multiples

Soit un entrepôt de données modélisé en étoile. Lors de la phase de conception physique de l'$\mathcal{ED}$, nous exécutons notre démarche de sélection statique jointe des IM et IS qui permet de sélectionner et implémenter une configuration finale d'index $CI_f = CIS_f \cup CIM_f$ pour optimiser une charge de requêtes $\mathcal{Q}$. CIS_f représente une configuration finale d'index simples et CIM_f une configuration finale d'index multiples. A l'issue de l'exécution de notre démarche, nous obtenons une classification de la charge Q en deux classes ClasseIS et ClasseIM et un partage de l'espace de stockage S en deux espaces S1 et S2. Supposons que la charge de requêtes évolue par l'exécution successive et continuelle de nouvelles requêtes. L'arrivée de chaque nouvelle requête Q_i déclenche le processus de sélection incrémentale jointe. Nous avons développé deux approches de sélections incrémentale jointe que nous détaillons dans ce qui suit.

3.3.1 Sélection naïve

Dans un premier lieu, nous proposons de ré-exécuter toute la démarche de sélection statique jointe décrite dans la section 3.2 : la classification des requêtes, le partage de l'espace de stockage, le déclenchement des deux sélections pour les IS et les IM et l'implémentation de la configuration finale d'index. Cette démarche, appelée démarche de sélection incrémentale jointe naïve, est illustrée sur la figure 3.9. Après analyse de cette démarche nous avons fait les constatations suivantes.

- Refaire le partage de toutes requêtes est inutile, il suffit juste de savoir à quelle classe appartient la nouvelle requête Q_i.
- Il n'est pas nécessaire d'effectuer les deux sélections d'index simples et multiples. En effet, la classification de la requête Q_i modifie une seule classe à la fois. Ainsi, la configuration d'index définie sur l'autre classe n'est pas altérée. Il suffit donc de réaliser une nouvelle sélection d'index

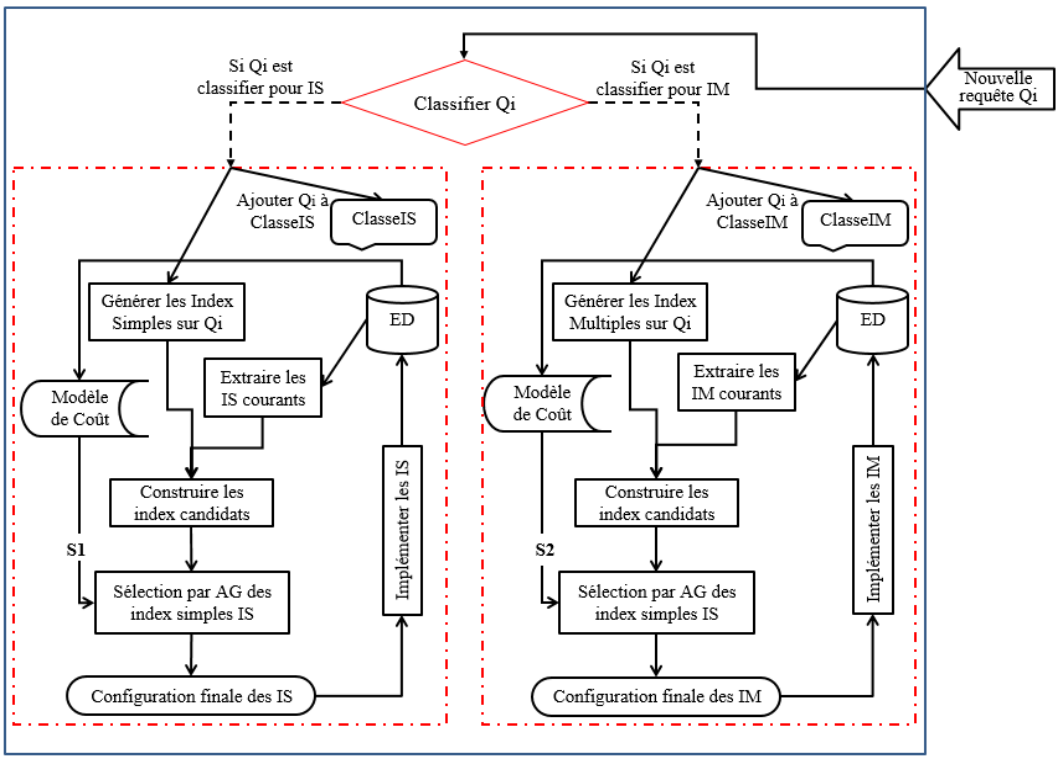


FIGURE 3.10 – Architecture de sélection incrémentale jointe des index simples et multiples

sur la classe à laquelle va appartenir Q_i. La réexécution d'une seule sélection optimise le temps de mise à jour de la configuration d'index implémentée sur l'entrepôt. Selon le type de sélection, nous mettons à jour soit les index simples soit les index multiples et non les deux.

- L'aspect incrémental de notre démarche signifie qu'il faut lancer une nouvelle sélection sur une configuration d'index candidats et non pas sur la classe de requête. Par conséquent, nous effectuons l'extraction des index définis sur Q_i et des index (soit simples soit multiples) actuellement implémentés sur l'$\mathcal{ED}$ et qui représentent la configuration optimale d'index sélectionnée par une précédente exécution du processus de sélection. L'union des deux ensembles constitue une configuration d'index candidats. Cela permet d'un côté d'optimiser le temps d'exécution du processus de sélection puisque l'espace de recherche est considérablement réduit. D'un autre côté, le temps nécessaire pour mettre à jour la configuration d'index implémentée sur l'entrepôt est également réduit.
- Il est impératif de garder le même partage de l'espace de stockage. Du moment qu'une seule sélection est effectuée, une seule configuration d'index est modifiée. On ne peut donc pas changer le partage de l'espace de stockage afin que la condition S1+S2=S soit toujours vérifiée.

3.3.2 Notre sélection incrémentale jointe

Après l'analyse faite sur la sélection naïve, nous proposons une démarche de sélection incrémentale jointe dont l'architecture globale est illustrée sur la figure 3.10. L'idée principale est de choisir la sélection à ré-exécuter selon la nature de la nouvelle requête Q_i. Les étapes du processus de sélection sont données dans ce qui suit.

Fonction ClassifierQ(Q_i :) : **TypeClasse**
Notation
getCardReq(Q_i) : retourne la cardinalité de la requête Q_i
getSupReq(Q_i) : retourne le support de la requête Q_i
getCentroide : renvoit le centroïde d'une classe
DistanceEuclidienne(x, y) : calcul la distance euclidienne entre x et y
Début
$CoordQ_i \longleftarrow$ (getCardReq(Q_i), getSupReq(Q_i)) ;
$Ct_{IS} \longleftarrow$ getCentroide(ClasseIS) ;
$Ct_{IM} \longleftarrow$ getCentroide(ClasseIM) ;
$DE_{IS} \longleftarrow$ DistanceEuclidienne($CoordQ_i$, Ct_{IS}) ;
$DE_{IM} \longleftarrow$ DistanceEuclidienne($CoordQ_i$, Ct_{IM}) ;
Si ($DE_{IS} \leq DE_{IM}$) **Alors**
ClasseIS $\longleftarrow$ ClasseIS $\cup$ {Q_i} ; TypeClasse $\longleftarrow$ IS ;
Sinon
ClasseIM $\longleftarrow$ ClasseIM $\cup$ {Q_i} ; TypeClasse $\longleftarrow$ IM ;
Fin Si
Retourner TypeClasse
Fin

Algorithme 19: Fonction ClassifierQ

1. Classifier Q_i : cette étape vise à déterminer si la nouvelle requête Q_i nécessite de mettre à jour les index simples ou les index multiples. Pour ce faire, nous utilisons l'étude réalisée lors du partage des requêtes par k-means. Nous calculons pour Q_i les critères de classification à savoir la cardinalité CardReq et le support SupReq. Ensuite, nous déterminons les centroïdes de chaque classe ClasseIS et ClasseIM. Puis, nous calculons la distance euclidienne entre Q_i et chaque centroïde. La requête Q_i sera classifiée pour les IS si la distance euclidienne entre elle et le centroïde de ClasseIS est la plus petite, elle sera classifiée pour les IM sinon. L'algorithme 19 résume le processus de classification de la requête Q_i.
2. Extraire la configuration d'index courante : selon le résultat de classification de Q_i, on extrait soit les index simples soit les index multiples. Dans le SGBD Oracle, deux tables permettent d'extraire la description d'un index donné. La table **User_Indexes** contient les index existants et la table **User_Ind_Columns** contient les attributs sur lesquels est défini chaque index. Afin d'extraire les IJB implémentés, nous exécutons la requête SQL suivante :

```
SELECT Index_Name
FROM User_Indexes
WHERE Index_Type = 'BITMAP'
  AND Join_Index='YES';
```

Pour chaque index obtenu IJB_i, nous exécutons la requête SQL suivante afin d'extraire les attributs sur lesquels il est défini.

```
SELECT Column_Name
FROM User_Ind_Columns
WHERE Index_Name = 'IJBi';
```

Nous combinons ses deux requêtes afin de définir une requête qui permet d'extraire uniquement les index simples comme suit :

```
SELECT UI.Index_Name, Count(UIC.Column_Name)
FROM User_Ind_Columns UIC, User_Indexes UI
WHERE UI.Index_Name=UIC.Index_Name AND Index_Type = 'BITMAP'
AND Join_Index='YES'
GROUP BY UI.Index_Name
HAVING Count(UIC.Column_Name)=1;
```

Pour obtenir uniquement les index multiples, nous exécutons la requête suivante :

```
SELECT UI.Index_Name, Count(UIC.Column_Name)
FROM User_Ind_Columns UIC, User_Indexes UI
WHERE UI.Index_Name=UIC.Index_Name AND Index_Type = 'BITMAP'
AND Join_Index='YES'
GROUP BY UI.Index_Name
HAVING Count(UIC.Column_Name)>1;
```

3. Extraire les attributs indexables à partir de Q_i. Si Q_i est classifiée pour les IS alors on définit sur chaque attribut un index simple. Sinon, on définit un index multiple couvrant tous ses attributs puis, suivant le codage MIQ^*, on génère les sous-index.
4. Regrouper la configuration d'index courante et les index construits sur la requête pour créer la configuration des index candidats.
5. Si la configuration des index candidats viole la contrainte d'espace de stockage (S1 pour la sélection des IS et S2 pour la sélection des IM), une sélection d'une configuration finale d'index guidée par modèle de coût est exécutée. Nous employons notre sélection basée sur les algorithmes génétique présentée dans la section précédente.
6. Implémenter les index sélectionnés : puisqu'il s'agit d'une ré-indexation de l'entrepôt, et dans le but de réduire le temps d'implémentation de la configuration finale d'index, nous implémentons uniquement les nouveaux index ne figurant pas dans l'entrepôt et supprimons de l'entrepôt les index obsolètes non sélectionnés par le processus de sélection. Également, si la sélection déclenchée est la sélection des IS (resp. IM), alors la mise à jour va être effectuée uniquement sur les index simples (resp. multiples), aucun index multiple (resp. simple) ne sera affecté. Dans le SGBD Oracle, l'implémentation des nouveaux index est réalisée par la syntaxe SQL CREATE BITMAP INDEX et la suppression est effectuée par la syntaxe DROP INDEX. Nous rappelons la syntaxe SQL pour créer et supprimer un index IJB_A1A2A3 défini sur trois attributs A1, A2 et A3 comme suit :

```
CREATE BITMAP INDEX IJB_A1A2A3
ON Facts(Dimension1.A1, Dimension2.A2, Dimension2.A3)
FROM Facts F, Dimension1 D1, Dimension2 D2
WHERE F.D1_ID=D1.D1_ID AND F.D2_ID=D2.D2_ID;
```

```
DROP INDEX IJB_A1A2A3;
```

3.3.3 Partage dynamique de l'espace de stockage

Dans notre démarche incrémentale, le partage de l'espace de stockage reste **statique**. Cependant, lors de l'exécution d'une nouvelle requête, de nouveaux index sont ajoutés ce qui augmente la taille de

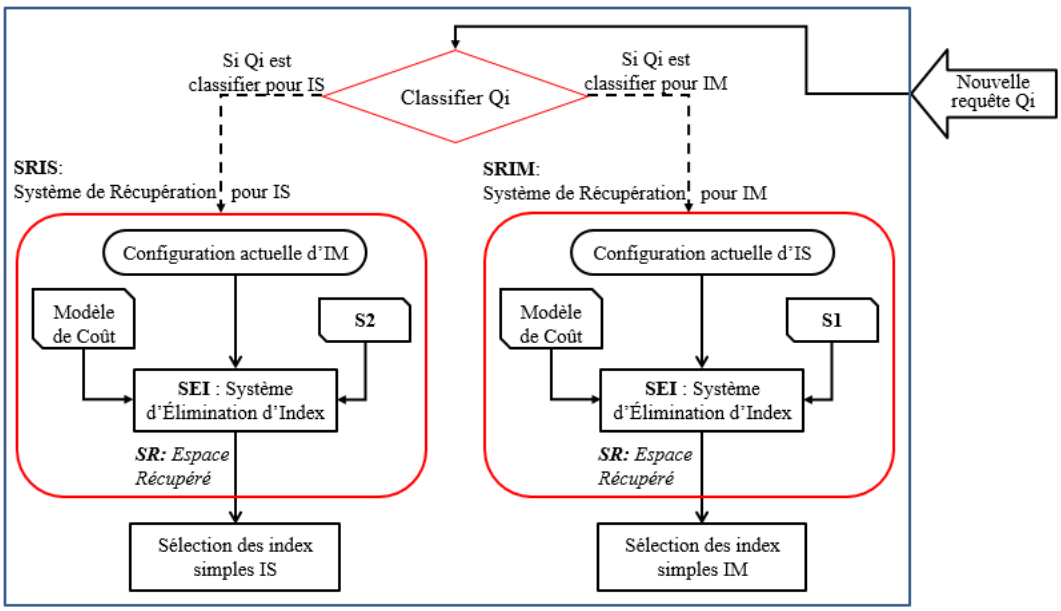


FIGURE 3.11 – Système de Récupération de l'espace de stockage

l'espace de stockage nécessaire pour stocker tous les index. Par conséquent, nous proposons d'effectuer un partage dynamique de l'espace de stockage entre les deux sélections d'index. L'idée générale est de créer un système de récupération de l'espace de stockage entre la configuration d'index simples et la configuration d'index multiples actuellement implémentés sur l'$\mathcal{ED}$. Lorsqu'une sélection des index se déclenche, tel que si A=IS alors B=IM et inversement, le système de récupération recherche de l'espace disponible au niveau de la configuration d'index B. Cet espace est ajouté à l'espace de stockage pour la sélection des index A. Nous illustrons sur la figure 3.11 le système de récupération de l'espace de stockage entre les IS et les IM. Les principaux composants qui constituent le système de récupération sont : SPIS, SPIM, SEI, S1, S2 et le modèle de coût où chaque composant réalise une tâche donnée.

1. Le **SRIS** représente le système de récupération d'espace de stockage pour la sélection des index simples.
2. Le **SRIM** est le système de récupération d'espace de stockage pour la sélection des index multiples.
3. Le Système d'Élimination d'Index **SEI** permet de supprimer l'IM (resp. IS) le moins bénéfique pour l'optimisation du coût de la nouvelle charge de requête $\mathcal{Q}\cup\{Q_i\}$, en se basant sur un modèle de coût. Le SEI produit l'espace de stockage récupéré SR et l'attribue à la sélection des IS (resp. IM).

Le système de récupération d'espace est employé dans la sélection incrémentale jointe des index simples et multiples. A l'exécution de la nouvelle requête Q_i, sa classification est réalisée. Si Q_i est classifiée pour les IS (resp. IM), le SRIS (resp. SRIM) se déclenche. La configuration d'IM (resp. IS) actuelle est extraite et est employée par le SEI afin de récupérer de l'espace de stockage. Ce dernier attribue l'espace récupéré ER à la sélection des IS (resp. IM). Une nouvelle sélection des IS (resp. IM) est donc exécutée avec une contrainte d'espace de stockage S1+ER (resp. S2+ER). Les

deux espaces de stockages S1 et S2 sont redéfinis comme suit :
S1←S1+ER, S2←S2-ER (resp. S1←S1-ER, S2←S2+ER)

3.4 Expérimentation

Nous avons mené des expérimentations sur un entrepôt de données réel issu du benchmark AP1 [40] afin de tester nos démarches de sélections jointes des index simples et multiples, que ce soit dans un contexte statique ou incrémentale. Nous avons conduits les tests sur une machine Intel Core 2 Duo avec une mémoire vive de 3Go. Nous considérons une charge de **60 requêtes de jointures en étoile** qui contiennent **12 attributs** de sélection *(Country, Depart, Class, Group, Family, Year, Month, Quarter, Retailer, City, Gender et All)* avec les cardinalités : 11, 25, 605, 300, 75, 2, 12, 4, 99, 4, 2, 3 respectivement. Le coût d'exécution de la charge des 60 requêtes sans aucune optimisation est de 39.5 Millions d'E/S. Pour cette étude, nous avons employé le modèle de coût mathématique pour les index **sans compression**. La taille de tous les index simples pouvant être implémentés sur l'entrepôt est de 7Go. Nous commençons notre étude expérimentale par le contexte statique puis nous passons au contexte incrémentale.

3.4.1 Évaluation de la sélection statique jointe

Dans la première évaluation, nous effectuons quatre tests sur la sélection statique. Le premier test vise à évaluer l'intérêt de la sélection statique jointe des $\mathcal{IJB}$ simples et multiples par rapport la sélection statique isolée. Dans le second test, nous testons l'influence de la contrainte de l'espace de stockage sur l'optimisation de la charge de requêtes. Dans le troisième test, nous comparons différentes stratégies de partage de l'espace de stockage entre la sélection des IS et celle des IM. Enfin, le quatrième test vise à situer notre approche proposée par rapport aux travaux existants. Les tests sont effectués selon deux dimensions ; une dimension théorique qui emploie un modèle de coût mathématique pour évaluer les solutions sélectionnées et une dimension pratique qui implémente les solutions sélectionnées puis utilise l'Optimiseur Oracle pour calculer le coût d'exécution des requêtes.

3.4.1.1 Test1 : variation du type de sélection d'index

Nous avons testé l'optimisation de l'entrepôt de donnée selon trois différents types de sélection d'index : sélection isolée des index simples (IS isolée), sélection isolée des index multiples (IM isolée) et sélection jointe des index simples et multiples (Jointe ISIM). Nous exécutons les trois sélections avec une contrainte d'espace de stockage S=5000 Mo. Chaque sélection génère une configuration d'index. Nous estimons par modèle de coût le coût d'exécution de la charge des 60 requêtes en présence de chaque configuration d'index. Les résultats sont donnés sur la figure 3.12. Nous représentons le coût total de la charge de requêtes sans optimisation qui est 39.5 Millions d'E/S.

Nous remarquons que la meilleure réduction du coût d'exécution est obtenue par la sélection jointe des index simples et multiples. Le coût passe de 39.5 millions E/S à 17.2 millions E/S. En effet, combiner les deux sélections d'index permet d'obtenir des index simples et multiples qui couvrent d'avantage l'optimisation des requêtes, où chaque type d'index palis aux manques de l'autre. D'un côté, les index multiples définis sur un ensemble d'attributs optimisent l'espace de stockage par rapport aux index simples définis sur les mêmes attributs car chaque index contient la colonne RowID dont la taille est 16 octets. D'un autre côté, des index multiples définis sur un grand nombre d'attributs sont très volumineux et sont écartés par le processus de sélection. De plus, notre sélection jointe

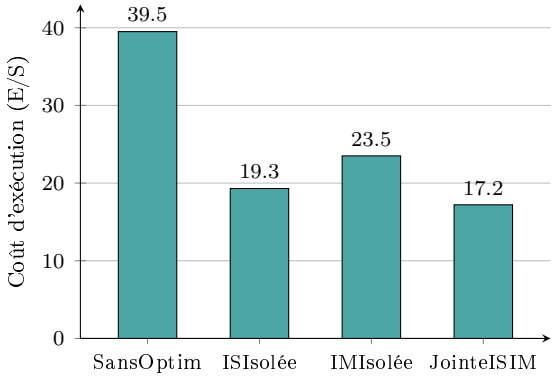


FIGURE 3.12 – Coût d'exécution vs. variation du type de sélection (théorique)

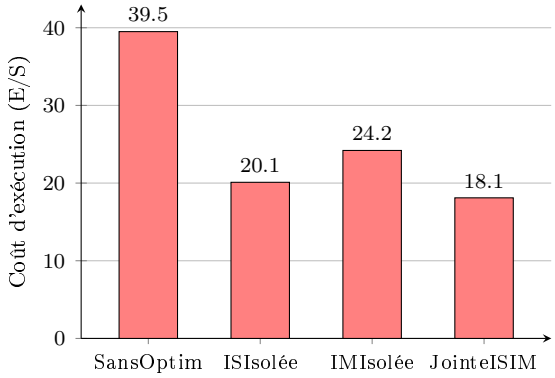


FIGURE 3.13 – Coût d'exécution vs. variation du type de sélection (Oracle 11g)

permet de choisir pour l'indexation simple et l'indexation multiple les requêtes les plus adéquates par le procédé de classification basé sur k-means. En comparant les deux sélections isolées, nous constatons que la sélection des index simples (le coût est de 19.3 millions E/S) et plus avantageuse que celle des index multiples (le coût est de 23.5 millions E/S). Cela est dû à la nature de la charge de requête. Étant donné que 45% des requêtes sont mono-attributs, elles contiennent chacune un seul attribut candidat à l'indexation. Ces requêtes sont optimisées par des index simples.

Afin de valider nos trois type de sélection (IS Isolée et IM Isolée et Jointe ISIM), nous réalisons des tests pratiques sous Oracle 11g. Nous exécutons les trois stratégies de sélection, guidée par modèle de coût théorique avec une contrainte d'espace de stockage S = 5000 Mo. Chaque stratégie génère une configuration d'$\mathcal{IJB}$ que nous implémentons sur l'entrepôt de données. Après chaque implémentation, nous calculons le coût Réel des requêtes. Pour ce faire nous utilisons notre classe ORACLECOST qui fait appel à l'Optimiseur Oracle à travers l'opération EXPLAIN PLAN. Cette classe JAVA accède à la table système d'Oracle PLAN_TABLE et récupère le coût des requêtes. Les coûts réels des requêtes pour chaque type de sélection sont illustrés sur la figure 3.13. Ces résultats montrent que la sélection jointe (18.1 million d'E/S) donne de meilleurs résultats que les deux sélections isolées (20.1 et 24.2 million d'E/S pour la sélection Isolée IS et Isolée IM resp.). Nous concluons également que le modèle de coût théorique que nous avons utilisé estime bien le coût de la charge de requêtes en présence des index.

3.4.1.2 Test2 : Variation de l'espace de stockage S

Nous avons voulu étudier l'influence de la contrainte d'espace de stockage S sur l'optimisation de la charge de requêtes. Pour ce faire, nous varions S entre 1000Mo et 5000Mo et pour chaque valeur, nous effectuons deux sélections isolées pour les IS et les IM et une sélection jointe. Nous relevons le coût d'exécution de la charge de requêtes estimé par modèle de coût. Les résultats sont illustrés sur la figure 3.14.

Nous remarquons que pour toutes les valeurs de S, la sélection jointe donne une meilleure optimisation que les sélections isolées. En effet, pour S=5000Mo, la sélection jointe réduit le coût de 39.5.6 million d'E/S à 17.2 million d'E/S, pour la sélection isolée des IS, le coût est réduit à 19.3 million d'E/S et pour la sélection isolée des IM, le coût est réduit à 23.5 million d'E/S. Ces résultats montrent bien l'efficacité du partage des requêtes par k-means entre les index simples et les index

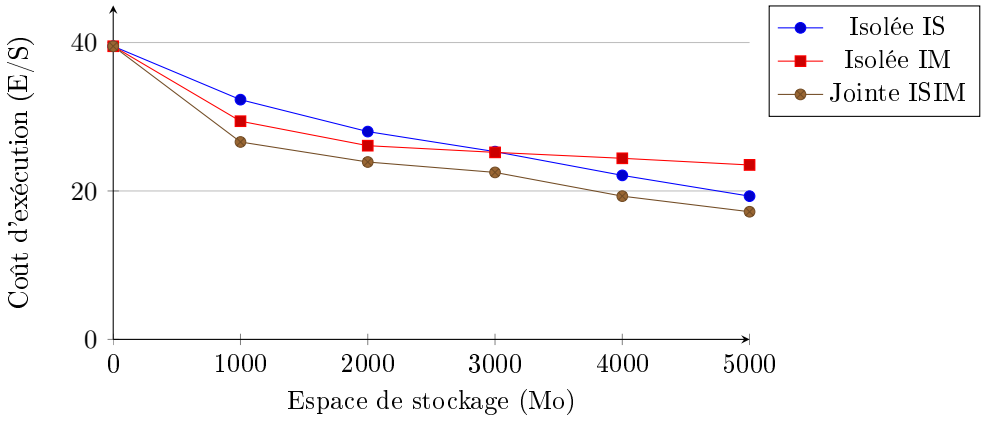


FIGURE 3.14 – Effet de la variation de l'espace de stockage S sur les démarches de sélection

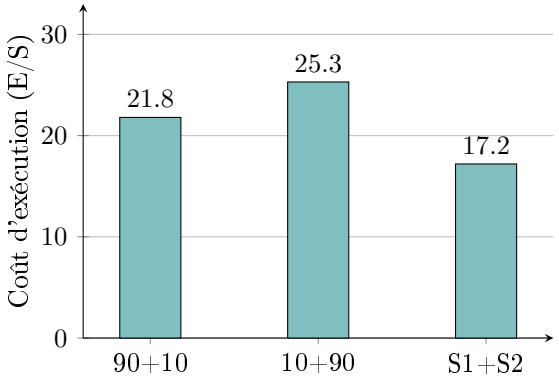


FIGURE 3.15 – Coût d'exécution vs. partage de l'espace de stockage (théorique)

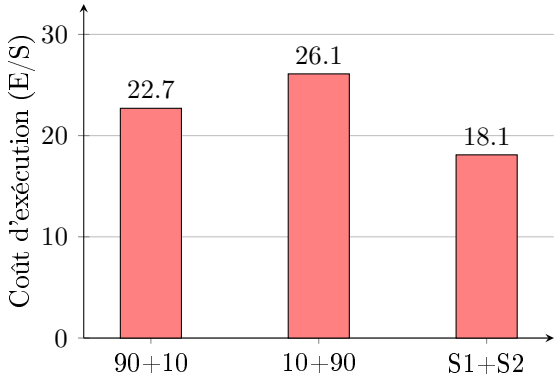


FIGURE 3.16 – Coût d'exécution vs. partage de l'espace de stockage (Oracle11g)

multiples qui choisit pour chaque type index les requêtes les plus adéquates. Nous remarquons également que la sélection isolée des IM est plus efficace que la sélection isolée des IS pour S<3000Mo. Au-delà de cette valeur de S les tendances se renversent car d'avantage d'IS sont sélectionnés ce qui couvre l'optimisation de plus de requêtes.

3.4.1.3 Test3 : Évaluation de l'approche de partage de S

Nous avons voulu étudier l'influence de partage d'espace de stockage S entre la sélection des IS et celle des IM sur l'optimisation de la charge de requêtes. Nous avons testé trois stratégies de partage avec une contrainte S=5000Mo. Dans la première stratégie 90%+10%, nous allouons 90% de l'espace de stockage S pour la sélection des IS (4500Mo) et 10% pour la sélection des IM (500Mo). Dans la seconde stratégie 10%+90%, nous allouons 10% de l'espace de stockage S pour la sélection des IS (500Mo) et 90% pour la sélection des IM (4500Mo). La troisième stratégie adopte notre partage de l'espace de stockage (S=S1+S2) basé sur les cardinalité des classes de requêtes ClasseIS et ClasseIM que nous avons présenté dans la section 3.2.4. Pour chaque stratégie, nous exécutons la sélection jointe des index simples et multiples et notons le coût d'exécution de la charge de requêtes en présence des index sélectionnés. Les résultats sont illustrés sur la figure 3.15.

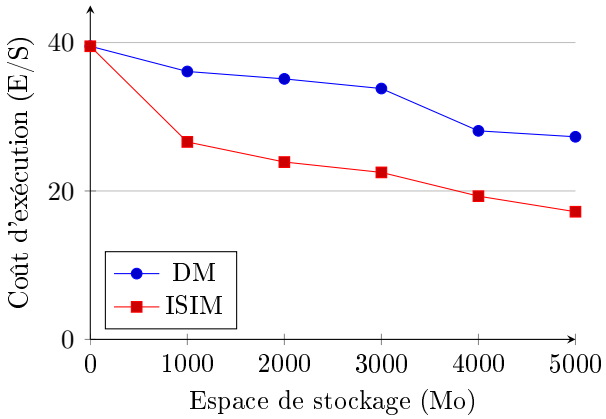


FIGURE 3.17 – Coût d'exécution et variation de S : *DM* vs. *ISIM*

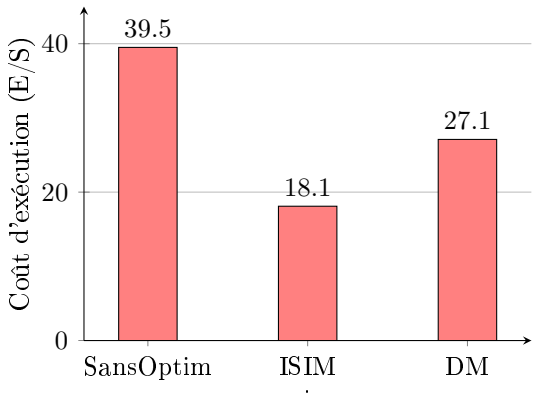


FIGURE 3.18 – Coût d'exécution réel sous Oracle 11g : *DM* vs. *ISIM*

En analysant les résultats de ce test, nous remarquons que notre partage S=S1+S2 apporte une meilleure optimisation pour la charge de requête (17.2 million d'E/S) que les deux autres stratégies de partage. En comparons ces deux derniers, nous constatons que pour la stratégie 90%+10%, le coût d'exécution de la charge (21.8 million d'E/S) est presque égale au coût d'exécution obtenu dans la sélection isolée des index simples pour S=4500Mo (test 2, figure 3.14). Effectivement, l'espace réservé aux index multiples volumineux est insuffisant pour sélectionner une configuration d'*IM* bénéfique. La même conclusion est faite pour la stratégie 10%+90%. Cela explique bien le bénéfice apporté par la stratégie 90%+10% par rapport à la seconde stratégie (10%+90%) car les *IS* sélectionnés s'avèrent plus bénéfiques pour l'optimisation de la charge de requête.

Afin de valider le partage de l'espace de stockage que nous avons proposé, nous réalisons le même test présenté sur la figure 3.15 sous Oracle 11g. Nous exécutons la sélection jointe des *IS* et *IM* suivant les trois stratégies de partage avec une contrainte d'espace de stockage S = 5000 Mo. Pour chaque sélection, nous implémentons la configuration d'index obtenue sur l'entrepôt, puis à travers la classe JAVA ORACLECOST développée, nous calculons le coût réel de la charge de requêtes. Les résultats sont donnés sur la figure 3.16. Les tests sous Oracle11g montrent que notre partage donne une meilleure optimisation du coût des requêtes (18.1 million d'E/S) que les deux autres partages (90%+10% donne 22.7 millions d'E/S et 10%+90% donne 26.1 millions d'E/S).

3.4.1.4 Test4 : Comparaison de notre approche avec l'existant

Afin de situer notre approche de sélection jointe des index simples et multiples avec l'existant, nous avons comparé notre approche avec l'approche *DM* citée dans [2]. Les auteurs proposent une démarche de sélection des $\mathcal{IJB}$ simples et multiples basée sur le DataMining. L'algorithme CLOSE pour la recherche des motifs fréquents fermés est utilisé pour élaguer l'espace de recherche des index. Ensuite, un algorithme glouton guidé par modèle de coût est exécuté pour sélectionner la configuration finale d'$\mathcal{IJB}$.

Le test que nous avons effectué se déroule comme suit : nous varions l'espace de stockage S de 1000Mo à 5000Mo et pour chaque valeur de S, nous exécutons *ISIM* et *DM*. A l'issu de chaque sélection et pour chaque valeur de S, nous notons le coût d'exécution de la charge de requêtes en présence des index sélectionnés. Les résultats sont illustrés sur la figure 3.17. Les deux approches peuvent être comparées selon trois dimensions : le type d'index, le type d'élagage de l'espace de

recherche et l'algorithme de sélection. Selon le type d'index, les deux approches sélectionnent des index simples et des index multiples ce qui couvre l'optimisation d'un maximum de requêtes. Selon le types d'élagage de l'espace de stockage, la démarche DM se base uniquement sur la fréquence d'utilisation des attributs par les requêtes alors que dans le contexte d'entrepôt de données il est impératif d'employer plus de critères comme la taille des tables, des attributs, de la base systèmes, etc. Nous utilisons une approche d'élagage différente qui se base sur la description de la requêtes (codage MIQ^*). Selon l'algorithme de sélection, l'approche DM emploi un algorithme glouton alors que nous employons un algorithme génétique plus performant pour rechercher la solution optimale. Cela explique bien le résultat obtenu dans ce test. Effectivement, notre approche apporte une meilleure optimisation de la charge de requêtes par rapport à DM sur toutes les valeurs de S. Notre approche réduit le coût d'exécution des requêtes de 39.5 million d'E/S à 17.2 million d'E/S pour S=5000Mo, où DM apporte une réduction jusqu'à 27.3 million d'E/S pour S=5000Mo

Afin de valider les deux approches de sélections d'index, nous effectuons un test réel sous le SGBD Oracle11g. Nous exécutons les deux sélections sous une contrainte S=5000Mo puis implémentons les index obtenus. La figure 3.18 illustre les résultats obtenus où notre approche apporte une meilleure optimisation de la charge de requête que l'approche existante DM.

3.4.2 Évaluation de la sélection incrémentale jointe

Nous avons effectué des tests sur la sélection incrémentale jointe avec une machine Intel CORE i5 ayant une capacité mémoire de 6Go. Le test est effectué sur une charge de 60 requêtes qui génèrent 18 attributs de sélection *(Line, Day, Week, Country, Depart, Type, Sort, Class, Group, Family, Division, Year, Month, Quarter, Retailer, City, Gender and All)* qui ont respectivement les cardinalités suivantes : 15, 31, 52, 11, 25, 25, 4, 605, 300, 75, 4, 2, 12, 4, 99, 4, 2, 3. Nous supposons une charge de 45 requêtes optimisées avec un ensemble d'$\mathcal{IJB}$ simples et multiples sélectionnés préalablement par notre sélection statique jointe. Cette dernière donne lieu à une classification de la charge de requêtes en deux classe ClasseIS et ClasseIM et un partage de l'espace de stockage S en S1 pour le stockage des IS et S2 pour les IM. L'étude incrémentale est réalisée avec ajout successif de 15 nouvelles requêtes en considérant une contrainte d'espace de stockage $S = 3Go$. Nous implémentons les deux démarches que nous avons proposées ; la sélection incrémentale jointe naïve $SJIN$ et la sélection incrémentale jointe avec classification des requêtes $SJIC$

- Le principe général de notre approche $SJIC$ consiste à classifier chaque nouvelle requête exécutée afin de l'affecter soit à ClasseIS soit à ClasseIM. Selon la nature de la requête, nous exécutons soit la sélection des index simples soit celle des index multiples. Ainsi, la configuration d'index implémentée sur l'entrepôt est mise à jour selon un seul type d'index. Nous assurons un partage dynamique de l'espace de stockage S entre les deux sélections afin qu'il soit adapté à l'évolution de la charge requêtes.
- Le principe de l'approche $SJIN$ est de ré-exécuter toute la démarche de sélection statique jointe décrite dans la section 3.2 : la classification des requêtes, le partage de l'espace de stockage, le déclenchement des deux sélections pour les IS et les IM et l'implémentation de la configuration finale d'index.

Nous effectuons un test théorique basé sur modèle de coût mathématique puis un test pratique sous Oracle 11g afin d'évaluer les deux approches de sélection. Ensuite, nous comparons notre approche $SJIC$ avec les travaux existants. Le tableau 3.5 résume la description des 15 nouvelles requêtes et donne pour chacune ses attributs de sélection, le type d'index pour lequel elle est classifiée et le nombre d'index pouvant être définis sur ses attributs pour les deux approches $SJIC$ et $SJIN$.

Req	Attributs	SJIC		SJIN
		IS/IM	Total IJB	Total IJB
Q46	Group	IS	1	1
Q47	Group, Familly	IS	2	3
Q48	Retailer	IS	1	1
Q49	Sort, Quarter, Type, All	IM	11	15
Q50	Week	IS	1	1
Q51	Month, City, Retailer	IM	4	7
Q52	Line, City, Year, Sort, Gender	IM	26	31
Q53	Month, Quarter, Gender	IM	4	7
Q54	Group, Type	IS	2	3
Q55	Division, Family, Type	IM	4	7
Q56	Class	IS	1	1
Q57	Depart, Type, Line, City	IM	11	15
Q58	Sort, Country, Type	IM	4	7
Q59	Family, Retailer, Group	IS	3	7
Q60	Day, Week, Month, Year	IM	11	15

TABLE 3.5 – Description des nouvelles requêtes : sélection incrémentale jointe IS&IM

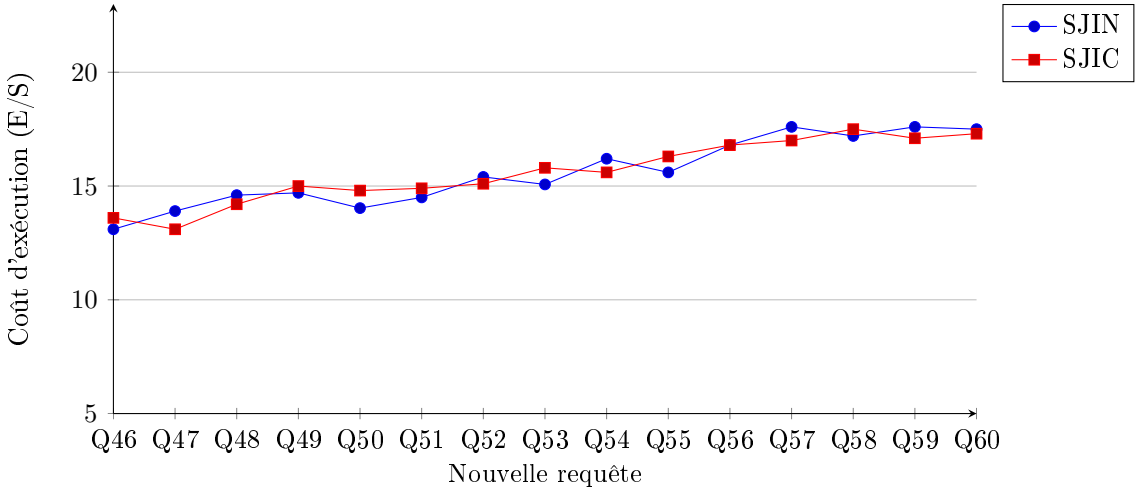


FIGURE 3.19 – Comparaison entre sélections incrémentale jointe : $SJIN$ vs. $SJIC$

3.4.2.1 Test1 : Évaluation Théorique

L'évaluation théorique se déroule comme suit : nous faisons varier la contrainte sur l'espace de stockage S entre 1000Mo et 5000Mo et pour chaque valeur de S, nous exécutons les deux sélections $SJIN$ et $SJIC$. Nous estimons par modèle de coût le coût de la charge de requêtes en présence de chaque configuration d'index sélectionnée. Les résultats sont illustrés sur la figure 3.19.

Les résultats obtenus montrent une augmentation continuelle du coût d'exécution de la charge de requêtes due à l'ajout successif de nouvelles requêtes. Nous remarquons également que les deux sélections $SJIN$ et $SJIC$ apportent la même optimisation du coût d'exécution. Afin d'expliquer cette similitude, nous analysons les deux sélections selon les trois étapes qu'ils les constituent : classification des requêtes, algorithme de sélection et partage de l'espace de stockage S.

1. Classification : L'approche $SJIN$ se base sur une classification de toute la charge de requêtes

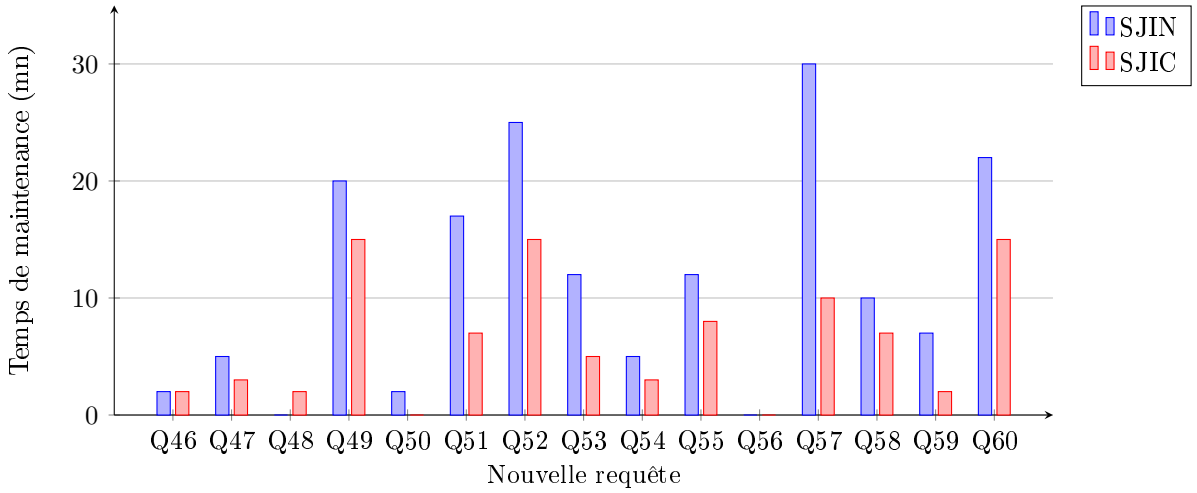


FIGURE 3.20 – Temps de maintenance des index sous Oracle 11g : $SJIN$ vs. $SJIC$

alors que $SJIC$ effectue uniquement la classification de la nouvelle requête exécutée Q_i. Cependant, les deux opérations aboutissent au même résultat de classification puisque l'unique changement est issu de Q_i. On déduit donc qu'une seule classe de requête évolue.

2. Sélection : A l'issu de la classification, nous obtenons les mêmes classes de requêtes. La démarche $SJIC$ effectue une nouvelle sélection d'une configuration finale d'index sur une configuration d'index candidate représentant l'union entre la configuration d'index actuellement implémentée sur l'entrepôt et les index extrait de Q_i. La démarche $SJIN$ effectuent deux sélections d'index sur les deux classes de requêtes, ce qui au final donne un changement significatif uniquement par rapport à la classe à laquelle appartient Q_i.
3. Partage de S : Pour la démarche $SJIN$, le partage de S et ré-exécuté selon la nouvelle classification de la charge de requêtes. La cardinalité de la classe à laquelle appartient Q_i augmente ce qui augmente l'espace de stockage réservé aux index correspondants. Pour $SJIC$, le système de récupération obtient de l'espace de stockage pour les index qui correspondent à la classe qui à évoluée. Donc pour les deux sélections, l'espace de stockage reversé pour les index dont la classe a évolué augmente et l'autre espace diminue.

D'après cette analyse et la similitude des résultats d'optimisation, nous avons montré l'efficacité de notre démarche proposé $SJIC$; la classification de la nouvelle requête exécutée, l'exécution de l'algorithme de sélection et le partage dynamique de l'espace de stockage.

3.4.2.2 Test2 : Évaluation Pratique sous Oracle 11g

Vu la similitude des résultats obtenus dans le test théorique, nous avons effectué un nouveau test sous le SGBD Oracle afin d'évaluer le temps de maintenance des index actuellement implémentée sur l'$\mathcal{ED}$. Le temps de maintenance représente le temps nécessaire pour implémenter les nouveaux index et le temps de supprimer les index obsolètes. Les nouveaux index sont les index sélectionnés par le processus de sélection mais non présents dans l'entrepôt. Les index obsolètes sont non sélectionnés mais actuellement implémentés sur l'entrepôt. Durant le déroulement du test précédent, pour chaque nouvelle requête et chaque sélection, nous implémentons la configuration d'index finale et nous re-

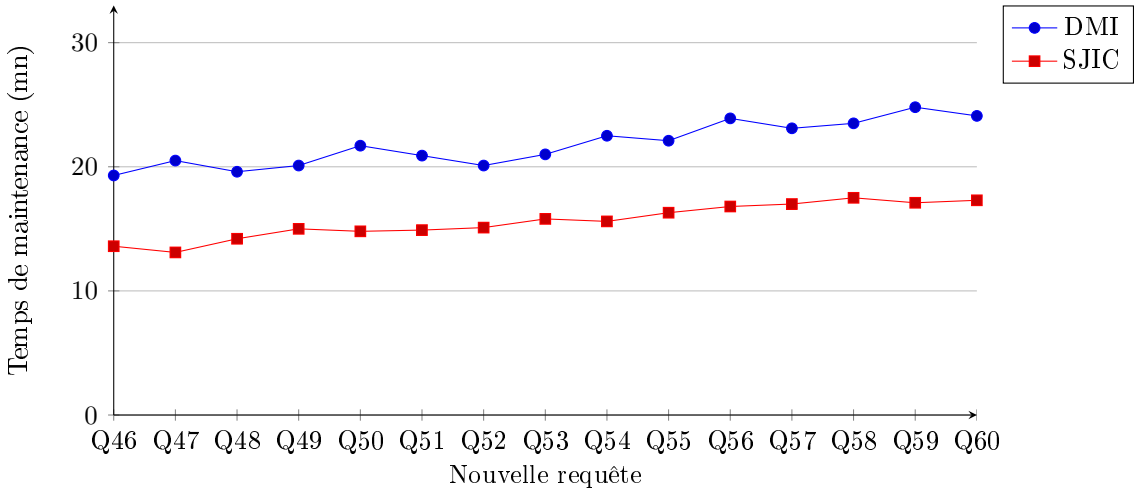


FIGURE 3.21 – Coût d'exécution des requêtes : DMI vs. $SJIC$

levons le temps de maintenance. Les résultats sont illustrés sur la figure 3.20. Notons que le temps d'implémentation d'un index (simple ou multiple) est en moyenne entre 2mn et 3mn. Le temps de suppression d'un index est entre 1s et 2s.

Nous remarquons que le temps de maintenance apporté par notre approche $SJIC$ est considérablement réduit par rapport à l'approche $SJIN$. Le temps de maintenance global pour tout le test est de 169mn (2h et 49mn) pour $SJIN$ et de 92mn (1h et 32mn) pour l'approche $SJIC$ soit une réduction de près de 45%. La principale différence entre les deux approches $SJIN$ et $SJIC$ est le temps nécessaire pour la mise à jour des index implémentés. D'une part, l'approche $SJIN$ met à jour tous les index implémentés alors que $SJIC$ met à jour soit les index simples soit les index multiples. De plus, les deux sélections exploitent deux espaces de recherches différents. Effectivement, dans $SJIN$ les algorithmes de sélections sont lancés sur toute la charge de requêtes alors que dans $SJIC$ la sélection est effectuée sur un ensemble d'index candidats (les index implémentés et les index issus de Q_i). De plus, sur les requêtes Q46, Q47, Q48, Q50, Q54, Q56 et Q59, l'approche $SJIC$ apporte un temps de maintenance entre 0 et 3mn car ces requêtes sont classifiées pour les index simples ce qui donne un nombre d'index possibles réduit.

3.4.2.3 Test3 : Comparaison de notre sélection avec l'existant

Dans la littérature, seuls les travaux cités dans [3] proposent une sélection incrémentale jointe des index simples et des index multiples. Ces travaux emploient l'algorithme CLOSE de Datamining pour effectuer l'élagage de l'espace de recherche et un algorithme glouton, guidé par modèle de coût, pour sélectionner et implémenter la configuration finale d'index. Une base de connaissance qui garde trace de toutes les anciennes exécutions du processus de sélections est employée afin de ré-indexer l'entrepôt de données. Nous comparons cette approche que nous nommons DMI (sélection incrémentale avec Datamining) à notre approche $SJIC$. Nous considérons l'exécution successive de 15 requêtes. Pour chaque nouvelle requête, nous exécutons les deux approches DMI et $SJIC$ avec un espace de stockage pour les index S=5000Mo. Les résultats sont illustrés sur la figure 3.21.

Nous remarquons que notre approche de sélection incrémentale jointe $SJIC$ apporte une meilleure optimisation du coût de la charge de requêtes que la démarche DMI et cela en tout temps. $SJIC$

apporte un coût d'exécution qui évolue de 13.6 millions E/S jusqu'à 17.3 millions E/S alors que DMI apporte un coût d'exécution de 19.3 millions E/S jusqu'à 24.1 millions E/S. Ces résultats peuvent être expliqués par plusieurs raisons. D'abords, notre démarche emploi les algorithmes génétiques pour la sélection alors que DMI utilise un algorithme glouton. De plus, l'emploi de l'algorithme génétique, qui utilise les paramètres systèmes dans le modèle de coût, apporte une meilleure optimisation de la charge que DMI. En effet, DMI utilise uniquement la fréquence d'accès des requêtes pour la recherche des motifs fermés, ce qui influe négativement sur l'enrichissement de la base de connaissance qui est au coeur de la sélection incrémentale pour DMI.

3.5 Conclusion

Dans ce chapitre, nous avons étudié le problème de sélection jointe des index de jointure binaires simples et multiples qui est un problème NP-Complet. A l'issu de notre étude, nous avons proposé une démarche de sélection statique jointe. Nous avons montré que l'espace de recherche de la sélection jointe est très complexe. De plus, l'espace de recherche des index simples est très "petit" relativement à celui des index multiples, ce qui réduit la probabilité de sélectionner un index simple par rapport aux index multiples. Par conséquent, nous avons proposé de séparer les deux espaces de recherche ce qui induit à la concurrence des deux sélections d'index aux mêmes ressources : charge de requêtes et espace de stockage S. Cette concurrence nous a conduit à définir deux nouveaux problèmes d'optimisation combinatoire : le problème de classification des requêtes en deux classes et le problème de partage de l'espace de stockage en deux espaces. Afin de résoudre le problème de classification des requêtes, nous avons adopté l'algorithme k-means vu sa simplicité d'utilisation et d'implémentation et son efficacité pour répondre à nos besoins. Pour le partage de l'espace de stockage, nous avons proposé une démarche guidée par le résultat de classification des requêtes car la taille de chaque classe de requêtes influe sur l'espace de recherche ce qui influe sur l'espace nécessaire pour stocker les index.

Afin de répondre aux problèmes d'évolution de charge et d'instance dans les entrepôts de données, nous avons développé une approche de sélection incrémentale qui permet de mettre à jour les index implémentés sur l'entrepôt. Nous avons remarqué que lors de l'évolution de charge, il est requis de mettre à jour soit les index simples soit les index multiples ce qui réduit considérablement le temps de maintenance (mise à jour) des index actuellement implémentés sur l'entrepôt. Également, nous avons développé un système de récupération d'espace de stockage qui permet de garder un équilibre du partage de l'espace de stockage entre les index simples et multiples.

Nous avons terminé par une étude expérimentale constituée de tests théoriques à la base d'un modèle de coût mathématique et de tests pratiques sous le SGBD Oracle 11g. Nous avons aussi comparé nos approches avec les travaux existants dans la littérature. Notre sélection statique jointe apporte une meilleure réduction du coût d'exécution de la charge de requêtes par rapport aux approches qui combinent les deux espaces de recherches des index simples et multiples ce qui montre l'efficacité du partage de la charge de requêtes et du partage de l'espace de stockage. Nous avons fait le même constat pour la sélection incrémentale jointe. De plus, nous avons montré l'efficacité du système de récupération de l'espace de stockage qui équilibre bien l'espace de stockage entre les deux types d'index quand la charge de requête évolue.

Chapitre 4

Sélection incrémentale jointe des IJB et de la FH

Sommaire

4.1 Introduction

La sélection incrémentale d'un schéma d'optimisation devient primordiale afin d'optimiser la charge des requêtes décisionnelles en mettant à jour continuellement le schéma d'optimisation implémenté sur l'entrepôt. Néanmoins, l'utilisation de la sélection isolée d'une seule technique d'optimisation (TO) est de moins en moins efficace. En effet, la sélection jointe de plusieurs techniques d'optimisation permet de couvrir l'optimisation d'un nombre plus large de requêtes en exploitant la similarité entre ces TOs tout en palliant aux manques ou contraintes liées à chacune d'entre elles. En prenant l'exemple de la fragmentation horizontale et des index de jointures binaires, les deux techniques présentent des similarités ce qui rend leur sélection jointe très intéressante [27]. Effectivement, les deux techniques optimisent le même type d'opérations ; sélection sur les tables dimensions avec jointures en étoiles entre table de faites et tables dimension. Cependant, elles sont sélectionnées sous des contraintes d'optimisation ; espace de stockage pour les index et nombre maximum de fragments de la table de faits pour la fragmentation, donc une seule technique ne couvre plus l'optimisation de toute la charge de requêtes.

A notre connaissance, **il n'existe aucun travail qui traite de la sélection incrémentale jointe en général et la sélection incrémentale jointe des $\mathcal{IJB}$ et de la $\mathcal{FH}$ en particulier**. De ce fait, nous proposons une nouvelle sélection qui possède deux principales caractéristiques. Elle est jointe car elle combine la sélection de l'indexation et la fragmentation et elle est incrémentale car elle met à jour les schémas d'optimisations implémentés sur l'entrepôt pour prendre en compte l'évolution de ce dernier.

Ce chapitre est réparti en cinq sections. Dans la section 2 nous abordons notre sélection jointe statique des deux TOs qui va servir de points de départ pour notre proposition. La section 3 décrit notre approche de sélection incrémentale jointe. La section 4 contient une étude expérimentale qui permet d'évaluer l'approche proposée. Enfin, la section 5 conclu le chapitre.

4.2 Sélection jointe statique des $\mathcal{IJB}$ et de la $\mathcal{FH}$

Nous avons constaté que peu de travaux se sont intéressés à la mise en œuvre d'une démarche de sélection jointe statique des $\mathcal{IJB}$ et de la $\mathcal{FH}$. Les auteurs dans [89] proposent d'indexer un entrepôt de données puis définissent une démarche de fragmentation hiérarchique des tables et des index. La démarche permet de réduire le nombre d'attributs de fragmentation en choisissant un seul attribut par hiérarchie de dimension, mais risque de générer un nombre considérable de fragments faits. Concernant les travaux dans [12, 27], la démarche de sélection jointe commence par définir un schéma de fragmentation à partir de la charge de requêtes, puis définie des index de jointures binaires sur les requêtes non bénéficiaires de la fragmentation. L'inconvénient dans cette proposition est l'espace de recherche très complexe de la fragmentation qui est défini sur tous les attributs de sélection issus de toute la charge de requêtes. De plus, la fragmentation peut utiliser certains attributs de sélection plus adéquats pour l'indexation et vice versa.

L'analyse de ces travaux nous a permis de constater que les deux techniques d'optimisation sont concurrentes sur la même ressource à savoir les attributs de sélection issus des tables dimension. Dans le contexte d'entrepôts de données, les attributs dimensions servent à effectuer des analyses décisionnelles et leur nombre peut être de l'ordre de plusieurs centaines d'attributs. Par conséquent, définir un schéma d'optimisation sur un tel ensemble d'attributs devient une tâche difficile. Nous commençons par introduire le problème de sélection jointe statique, nous exposons ensuite notre

démarche proposée.

4.2.1 Formalisation du problème

Le problème de sélection jointe statique de la fragmentation et de l'indexation PSJSFI est formalisé comme suit :
Étant donné :

- un $\mathcal{ED}$ modélisé par un schéma en étoile ayant d tables de dimension $\mathcal{D} = \{D_1, D_2, \cdots, D_d\}$ et une table des faits $\mathcal{F}$;
- une charge de requêtes $\mathcal{Q} = \{Q_1, Q_2, \cdots, Q_m\}$ à partir de laquelle un ensemble d'attributs de sélection $AS = \{A_1, \cdots, A_n\}$ est obtenu ;
- un espace de stockage d'index $\mathcal{S}$;
- un nombre de fragments faits à générer $\mathcal{W}$.

Le PSJSFI consiste à sélectionner un schéma de fragmentation SF et une configuration d'index CI_f sur l'ensemble AS tel que :

- Le coût d'exécution de la charge $\mathcal{Q}$, sur l'$\mathcal{ED}$ partitionné suivant SF et indexé par CI_f, est optimisé.
- L'espace de stockage alloué pour les index de CI_f ne dépasse pas $\mathcal{S}$ ($Taille(CI) \leq \mathcal{S}$).
- le nombre de fragments de la table de faits générés N ne dépasse pas $\mathcal{W}$ ($N \leq \mathcal{W}$)

Le PSJSFI est un problème complexe composé de deux problèmes de sélection prouvés NP-Complet ; le problème de sélection isolée des index [42] et le problème de sélection isolée d'un schéma de fragmentation [14]. Par conséquent, l'espace de recherche du PSJSFI combine de manière exponentielle les deux espaces de recherches des deux problèmes NP-Complet, et sa complexité augmente donc de manière exponentielle également [100]. Afin de résoudre ce problème, nous employons un compromis entre la sélection jointe itérative et intégrée qui, en se basant de la dépendance faible-forte entre l'indexation et la fragmentation, sépare les deux sélections des TOs tout en prenant en compte que les deux techniques partage le même environnement (l'ensemble des attributs de sélection).

4.2.2 Approche de sélection

Nous exposons notre approche de sélection jointe statique que nous avons présenté dans [22]. Notre approche vise à élaguer l'espace de recherche de la fragmentation horizontale, élaguer l'espace de recherche de l'indexation et étudier la concurrence entre les deux techniques d'optimisation sur les attributs de sélection. A la base de cette étude, l'élagage de l'espace de recherche de chaque technique est effectué en partageant les attributs de sélection entre les deux TOs. La figure 4.1 illustre l'architecture de sélection jointe statique de la fragmentation et de l'indexation avec partage des attributs de sélection. Le déroulement du processus de sélection est effectué selon les étapes suivantes :

1. Extraire les attributs de sélection à partir de l'analyse syntaxique de la charge de requêtes.
2. Partager les attributs de sélection entre les deux TOs en utilisant la classification par l'algorithme k-means. Le partage à priori des attributs de sélection entre les deux TOs réduit l'espace de recherche des deux problèmes de sélection.
3. Sélectionner un schéma de fragmentation sur les attributs classifiés pour la fragmentation dans l'étape précédente. Cette sélection se base sur l'approche de fragmentation horizontale des entrepôts de données, avec contrôle du nombre de fragments faits générés, présentée dans [12, 27].

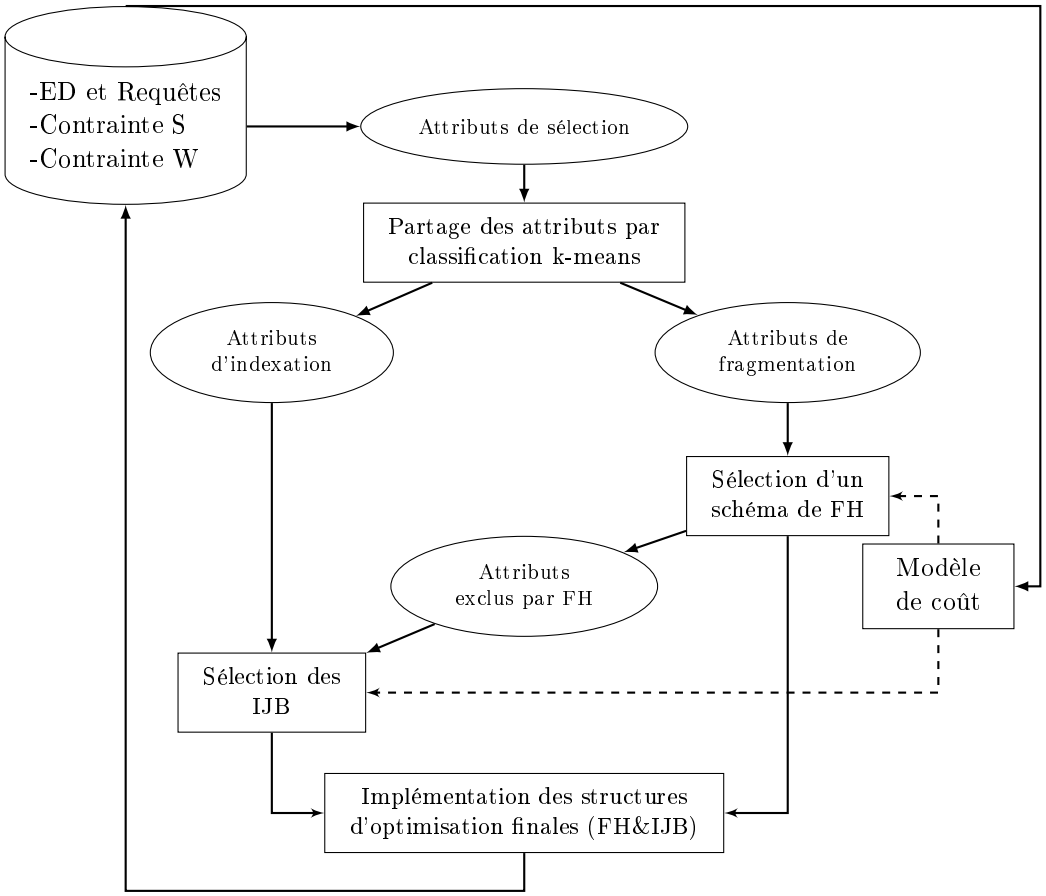


FIGURE 4.1 – Sélection jointe statique de la $\mathcal{FH}$ et des $\mathcal{IJB}$

4. Sélectionner les index de jointure binaires sur les attributs classifiés pour l'indexation et ceux écartés par le processus fragmentation. Le processus d'indexation employé est décrit dans le chapitre 1.
5. Implémenter les structures d'optimisation finales. Les tables dimensions sont fragmentées selon le schéma de $\mathcal{FH}$ primaire sélectionné, suivi de la $\mathcal{FH}$ dérivée de la table de faits. Ensuite, les $\mathcal{IJB}$ sont implémentés en utilisant le mot clé LOCAL du SGBD Oracle 11g. La création d'un $\mathcal{IJB}$ de type LOCAL, permet de créer un sous-$\mathcal{IJB}$ sur chaque fragment fait. Ce procédé est équivalent à la création des $\mathcal{IJB}$ globaux sur toute la table de faits puis de les fragmenter suivant le schéma de fragmentation de cette même table. La syntaxe Oracle de création d'un index local est donnée comme suit :

```
CREATE BITMAP INDEX IJB1
ON F(A1, A2,..., An)
FROM F, D1, D2,..., Dp
WHERE F.D1_ID=D1.D1_ID AND F.D2_ID=D2.D2_ID...
AND F.Dp_ID=Dp.Dp_ID LOCAL;
```

L'étape cruciale dans l'approche proposée est la répartition des attributs de sélection entre les deux TOs. Ainsi, nous allons la détailler dans ce qui suit.

4.2.2.1 Partage des attributs de sélection

Le partage des attributs de sélection entre $\mathcal{FH}$ et $\mathcal{IJB}$ vise à attribuer à chaque technique les attributs les plus bénéfiques. Ce sont les attributs qui donnent les structures d'optimisation qui optimisent au mieux la charge de requêtes. Pour ce faire, nous avons effectué une analyse des TOs afin de savoir comment répartir les attributs de sélection entre les deux. Nous présentons la formalisation du problème de partage des attributs puis nous présentons la démarche de partage.

(1) Formalisation du problème de partage : Le problème de partage des attributs de sélections PPAS entres la fragmentation et l'indexation est donné comme suit :
Étant donné :

- un $\mathcal{ED}$ modélisé par un schéma en étoile ayant d tables de dimension $\mathcal{D} = \{D_1, D_2, \cdots, D_d\}$ et une table des faits $\mathcal{F}$;
- une charge de requêtes $\mathcal{Q} = \{Q_1, Q_2, \cdots, Q_m\}$ à partir de laquelle un ensemble d'attributs de sélection $AS = \{A_1, \cdots, A_n\}$ est obtenu ;
- un espace de stockage d'index $\mathcal{S}$;
- un nombre de fragments faits à générer $\mathcal{W}$.

Le PPAS consiste à répartir les attributs de sélection entre la fragmentation et l'indexation tel que : (1) La sélection des deux techniques, définies chacune sur son ensemble d'attributs, optimise le coût d'exécution des requêtes. (2) L'espace réservée aux index ne dépasse pas $\mathcal{S}$. (3) Le nombre de fragments faits ne dépasse pas $\mathcal{W}$.

Le PPAS est un problème de classification d'un ensemble de n attributs en deux classes distinctes largement étudié dans le domaine du Datamining. Le nombre de classifications possibles est 2^n, tel que : une classe peut être vide, et au maximum contenir n attributs. Énumérer tous les cas possibles de classification et pour chaque cas exécuter deux algorithmes de sélection reste impossible vue la complexité identifiée par l'équation précédente. Ainsi, nous avons employé l'algorithme de classification k-means qui se base sur des critères de partage énoncées ci-dessous.

(2) Critères de partage : Nous avons analysé les deux techniques d'optimisation et avons conclus ce qui suit :

- *Analyse de la fragmentation.* Le but de la $\mathcal{FH}$ est de réduire le volume de données chargé par l'optimiseur pour exécuter une requête donnée. Cependant, si on considère un attribut de forte cardinalité (attributs avec un domaine de valeur très large comme l'attribut Age), la contrainte W sur le nombre de fragments faits maximum influe sur la répartition du domaine de l'attribut en ensemble de sous-domaines. Par conséquent, plusieurs sous-domaines de l'attribut se retrouveront dans le même ensemble, contraignant ainsi le chargement d'un grand volume de données pour la sélection sur une seule valeur. Prenant l'exemple d'un attribut Ville avec 50 valeurs distinctes. Supposant une contrainte W qui répartie le domaine de l'attribut Ville en 5 sous-ensemble contenant 10 villes chacun. La sélection sur le prédicat Ville='Alger' contraint le chargement de tous les tuples de l'entrepôt qui vérifient Ville='Alger' et Ville = au 9 autres valeurs du même sous-ensemble. Néanmoins, nous constatons un avantage important de la $\mathcal{FH}$ relativement à l'indexation ; un attribut candidat à l'indexation est soit sélectionné soit non sélectionné (une sélection binaire) alors qu'un attribut peut faire partie du schéma de $\mathcal{FH}$ mais avec un nombre de fragments réduit. En résumé, on dit que la $\mathcal{FH}$ est plus flexible que l'indexation.

– *Analyse de l'indexation.* Pour des attributs de faible cardinalité comme Genre (Dom(Genre)={'F', 'M'}), le volume de données chargé par un $\mathcal{IJB}$ défini sur ce type d'attributs, lors de l'exécution d'une requête, peut être tout aussi volumineux que celui chargé lors de l'accès direct à l'entrepôt. Dans ce cas, l'optimiseur jugera l'index inutile et utilisera directement les tables. Concernant les attributs de forte cardinalité (attribut Age par exemple), ces derniers peuvent définir des index volumineux qui peuvent être écartés par le processus de sélection à cause de la contrainte S sur l'espace de stockage. Néanmoins, l'indexation présente un avantage sur la $\mathcal{FH}$; les index définis sur les attributs de moyenne cardinalité sont plus bénéfiques car ils évitent de charger des données supplémentaires comme c'est le cas pour la $\mathcal{FH}$. Pour l'exemple d'un attribut Ville avec 50 valeurs distinctes. La sélection sur le prédicat Ville='Alger' avec présence d'un $\mathcal{IJB}$ défini sur Ville permet de charger uniquement les tuples de l'entrepôt qui vérifient Ville='Alger'.

A partir de cette analyse, nous avons déterminé trois critères de partage qui sont les suivants.

1. **Cardinalité de l'attribut Card :** les attributs de forte cardinalité sont adaptés à la $\mathcal{FH}$ tandis que les attributs de faible cardinalité sont candidats à l'indexation.
2. **Facteur de sélectivité d'un attribut FS :** nous avons constaté que le volume de données chargé, lors de l'exécution des requêtes en présence des structures d'optimisation, est un critère important. En effet, les expérimentations ont montré qu'un index défini sur un attribut est bénéfique si les facteurs de sélectivité de ses prédicats sont faibles. Le facteur de sélectivité d'un prédicat est une valeur entre 0 et 1 représentant la proportion de données chargées par une opération de sélection sur ce prédicat. Ainsi, nous définissons un facteur de sélectivité d'un attribut A (FS(A)), muni de k (k>0) prédicats de sélection P_1, ..., P_k, par la moyenne des facteurs de sélectivité de ses prédicats.
$$FS(A) = \frac{\sum_{i=1}^{n} Sel(P_i)}{Card(A)}$$
Nous favorisons donc pour l'indexation les attributs dont le facteur de sélectivité est réduit
3. **Fréquence d'utilisation d'un attribut Frc :** vu son caractère non redondant et sa flexibilité par rapport à l'indexation, on privilégie les attributs fréquemment utilisés par les requêtes pour la $\mathcal{FH}$.

Pour résumer, nous favorisons, pour la $\mathcal{FH}$, les attributs dont la fréquence d'utilisation, le facteur de sélectivité et la cardinalité sont grands.

(2) Principe de partage par k-means Le principe de l'algorithme de classification k-means est présenté dans la section 3.2.3. Pour exécuter l'algorithme k-means, nous avons considéré un espace $\mathbb{R}^2$ dans lequel sont représentés les attributs de sélection munis des coordonnées (i, Poids(A_i)). Le poids d'un attribut Poids(A_i) est calculé par la formule suite :

$$\text{Poids}(A_i)=\text{Frc}_i+\text{FS}_i+\text{Card}_i$$

Où Frc_i, FS_i et Card_i représentent respectivement la fréquence, le facteur de sélectivité et la cardinalité de l'attribut A_i tous normalisés. Chaque critère est normalisé afin d'obtenir un poids cohérent. La normalisation vise à centrer et réduire un échantillon de données suivant la moyenne et l'écart type de celui-ci, cet échantillon suivra alors une loi normale centrée réduire de moyenne 1 et d'écart type 0. En effet, les critères ont des échelles différentes, la cardinalité d'un attribut est une valeur numérique qui peut être de l'ordre de plusieurs dizaines ou centaine alors que le facteur de sélectivité est une

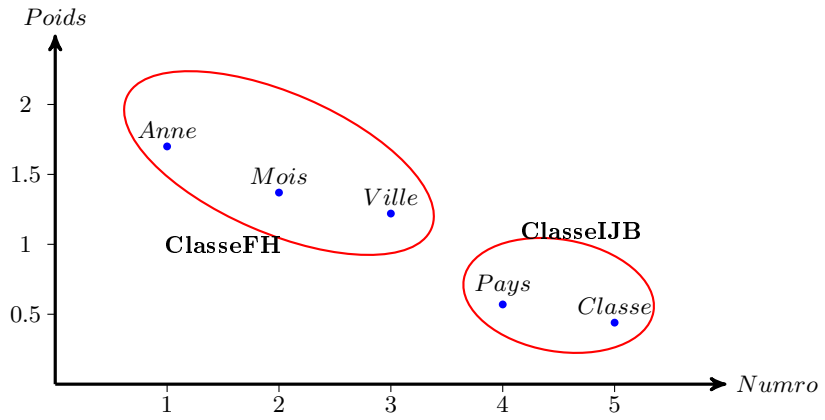


FIGURE 4.2 – Classification par k-means des attributs de sélection

valeur entre 0 et 1. Vu que nous favorisons pour la $\mathcal{FH}$ les attributs dont la fréquence d'utilisation, le facteur de sélectivité et la cardinalité sont grands. Cela signifie que les attributs à fort poids sont candidats à la $\mathcal{FH}$.

Exemple 38 *Considérons cinq attributs de sélection Année, Mois, Ville, Pays et Classe. Les poids normalisés de ces attributs sont donnés par le tableau 4.1. La figure 4.2 représente le partage des cinq attributs Année, Mois, Ville, Pays et Classe. Les attributs Année, Mois et Ville constituent ClasseFH et les attributs Pays et Classe composent ClasseIJB.*

Attribut	Année	Mois	Ville	Pays	Classe
Coordonnées	(1, 1.7)	(2, 1.37)	(3, 1.22)	(4, 0.57)	(5, 0.44)

TABLE 4.1 – Partage par k-means des attributs de sélections

4.3 Sélection incrémentale jointe des $\mathcal{IJB}$ et de la $\mathcal{FH}$

En considérant un $\mathcal{ED}$ optimisé conjointement par $\mathcal{FH}$ et $\mathcal{IJB}$, celui-ci peut évoluer par évolution de charge et/ou évolution d'instance. L'évolution de charge influe sur les structures d'optimisations implémentées sur l'entrepôt. L'évolution cause l'ajout et/ou la suppression d'attributs de sélection, l'ajout et/ou la suppression de sous-domaines d'attributs ce qui influe sur le schéma de $\mathcal{FH}$ d'un côté et sur la définition des $\mathcal{IJB}$ de l'autre. Quant à l'évolution d'instances, elle cause l'augmentation de la taille des tables particulièrement la table de fait, ce qui augmente la taille des $\mathcal{IJB}$. Cette évolution peut rendre obsolètes les structures d'optimisation déjà implémentées sur l'$\mathcal{ED}$. Par conséquent, nous proposons une démarche de sélection incrémentale jointe des $\mathcal{IJB}$ et de la $\mathcal{FH}$ qui vise à mettre à jour les structures d'optimisation implémentées lorsque l'entrepôt évolue.

4.3.1 Scénarios de sélection incrémentale jointe

Lorsque la charge de requêtes évolue, les structures d'optimisation implémentées peuvent changer selon trois scénarios possibles.

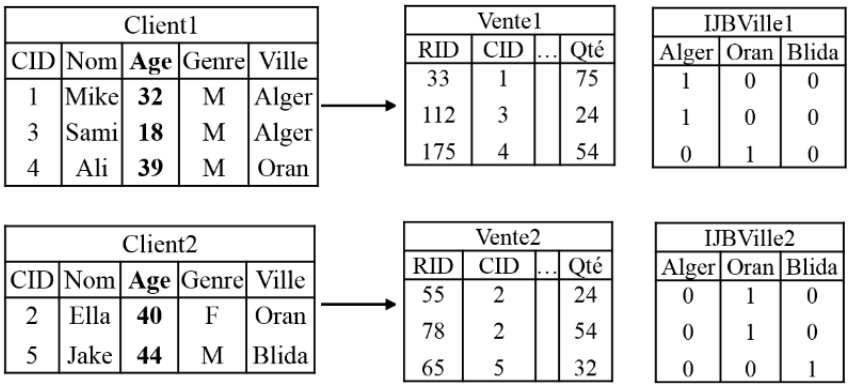


FIGURE 4.3 – Représentation des données de l'entrepôt avant évolution

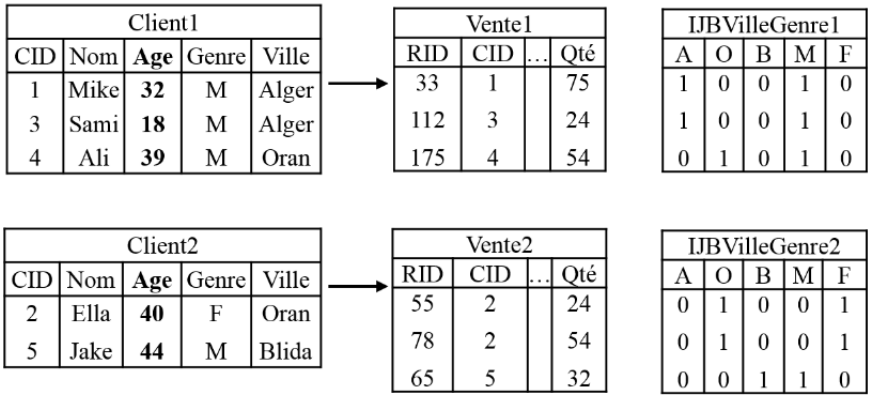


FIGURE 4.4 – Fragmentation sur Age et indexation sur Ville et Genre : Scénario 1

4.3.1.1 Scénario 1 : Seule l'indexation change

A l'exécution de nouvelles requêtes, celles-ci peuvent nécessiter de mettre à jour uniquement la configuration d'index implémentée sur l'entrepôt. Puisque la dépendance de la fragmentation envers l'indexation est faible, le changement des index n'influe pas sur le schéma de fragmentation actuel de l'entrepôt. Seule une nouvelle sélection d'un schéma d'indexation est requise.

Exemple 39 *Considérons un $\mathcal{ED}$ avec une table de faits Ventes et une table dimensions Clients. Supposant le schéma de l'entrepôt fragmenté et indexé comme le présente la figure 4.3. La sélection jointe de la fragmentation et de l'indexation a donné lieu à la création d'un $\mathcal{IJB}$ sur l'attribut Ville, appelé IJBVille, et à la définition d'un schéma de fragmentation sur l'attribut Age comme suit :*

- *Client1=$\sigma_{0<Age<40}(Clients)$*
- *Client2=$\sigma_{Age\geqslant 40}(Clients)$*

Supposons l'exécution de la nouvelle requête Q_i suivante :

```
SELECT AVG(Prix)
FROM Ventes V, Clients C
WHERE V.IdC=C.IdC
AND C.Genre = 'F' AND Ville='Alger'
```

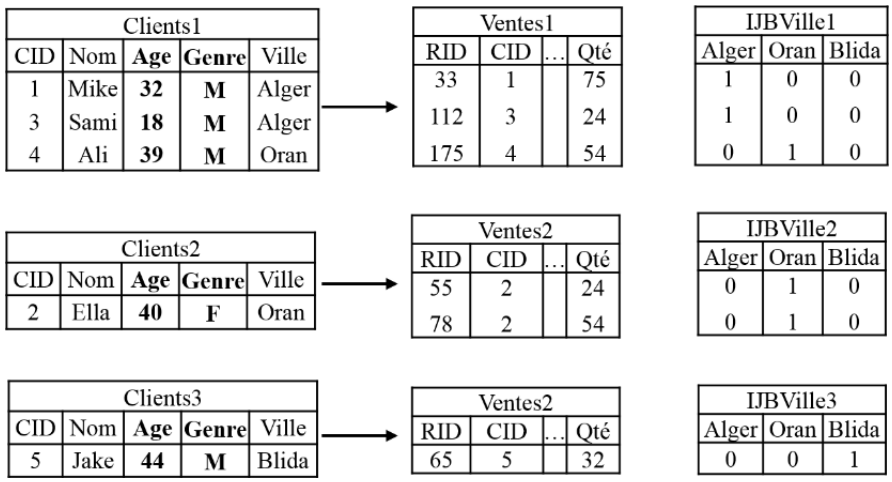


FIGURE 4.5 – Fragmentation sur Age et Genre et indexation sur Ville : Scénario 2

Supposant que la requête Q_i engendre la suppression de l'index IJBVille et l'implémentation d'un nouvel $\mathcal{IJB}$ sur les attributs Genre et Ville, appelé IJBVilleGenre. La description des données est illustrée sur la figure 4.4. En comparant les deux figures (figure 4.3 et figure 4.4), nous constatons que seuls les index ont changés et sont partitionnés puisque crées en utilisant la syntaxe LOCAL d'Oracle. Le schéma de fragmentation lui n'est pas altéré.

4.3.1.2 Scénario 2 : Seule la fragmentation change

L'exécution de nouvelles requêtes peut altérer le schéma de fragmentation au niveau des attributs et/ou de la répartition en sous-ensembles des sous-domaines. Par conséquent, une nouvelle exécution de l'algorithme de sélection d'un nouveau schéma de fragmentation est requise. Vu que l'indexation dépend fortement de la fragmentation, la configuration d'index est altérée également du point de vue répartition horizontale des données. Il est donc primordial de reconstruire ces index sans changer leur définition.

Exemple 40 *Considérons l'exemple 39 et un schéma de l'entrepôt fragmenté et indexé comme le présente la figure 4.3. Supposons que l'exécution de la nouvelle requête Q_i cause l'ajout au schéma de fragmentation de l'attribut Genre avec une répartition de son domaine {{F}, {M}} ce qui donne une fragmentation primaire de la table Clients comme suit :*

- *Client1*$=\sigma_{(0<Age<40)\wedge(Genre=M)}(Clients)$
- *Client2*$=\sigma_{(Age\geqslant 40\wedge(Genre=F)}(Clients)$
- *Client3*$=\sigma_{(Age\geqslant 40\wedge(Genre=M)}(Clients)$
- *Client4*$=\sigma_{(0<Age<40)\wedge(Genre=F)}(Clients)=\varnothing$

La nouvelle description des données est illustrée sur la figure 4.5. En comparant les deux figures (figure 4.3 et figure 4.5), nous constatons que le seul changement survenu au niveau de l'indexation est une nouvelle répartition de l'index IJBVille suivant le nouveau schéma de fragmentation.

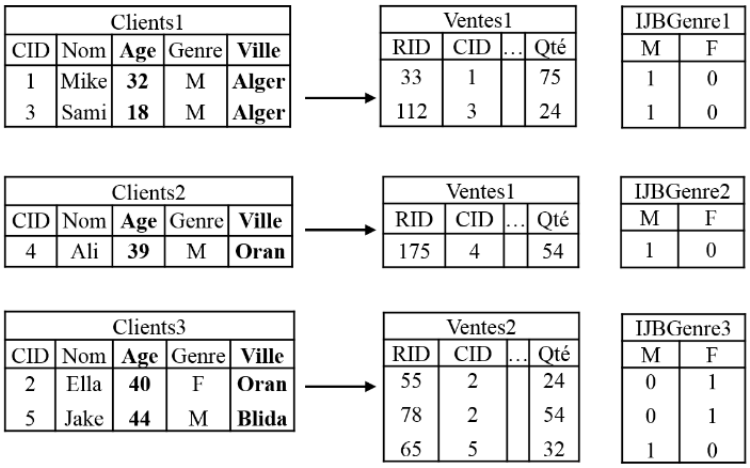


FIGURE 4.6 – Fragmentation sur Age et Ville et indexation sur Genre : Scénario 3

4.3.1.3 Scénario 3 : La fragmentation et indexation changent

Les nouvelles requêtes exécutées peuvent engendrer le changement du schéma de fragmentation et des $\mathcal{IJB}$ implémentés sur l'entrepôt. Dans ce cas, il est impératif d'exécuter une nouvelle sélection jointe d'un schéma de fragmentation et d'un schéma d'indexation.

Exemple 41 *Considérons l'exemple 39 et un schéma de l'entrepôt fragmenté et indexé comme le présente la figure 4.3. Supposons que l'exécution de la nouvelle requête Q_i cause la suppression de l'index IJBVille, la création de l'index IJBGenre et l'ajout de l'attribut Ville au schéma de fragmentation, avec une répartition de son domaine {{Alger}, {Oran, Blida}}, ce qui donne une fragmentation primaire de la table Clients comme suit :*

- *Client1*=$\sigma_{(0<Age<40)\wedge(Ville=Alger)}(Clients)$
- *Client2*=$\sigma_{(0<Age<40)\wedge(Ville\ IN\ (Oran,Blida))}(Clients)$
- *Client3*=$\sigma_{(Age\geqslant 40\wedge(Ville\ IN\ (Oran,Blida))}(Clients)$
- *Client4*=$\sigma_{(Age\geqslant 40\wedge(Ville=Alger)}(Clients)=\varnothing$

La nouvelle description des données est illustrée sur la figure 4.6.

Le facteur qui détermine le scénario qui se réalisera est la classification des attributs de sélections extraits de la nouvelle requête afin de savoir à quelle classe (ClasseFH ou ClasseIJB) appartiendra chaque attribut. En d'autres termes, la classification permet de déterminer si ces attributs sont candidats à l'indexation (scénario 1), à la fragmentation (scénario 2) ou les deux (scénario 3).

4.3.2 Démarche de sélection incrémentale jointe

Afin de mettre en oeuvre une approche de sélection incrémentale jointe des deux TOs, nous avons commencé par une approche dite Naïve dont le principe et de ré-exécuter toute la sélection jointe statique exposée précédemment lorsque la charge de requêtes évolue. Cependant, cette vision de sélection ne respecte pas le principe **incrémentale**. Une approche de sélection est dite incrémentale si elle prend en compte l'existant et tente de l'adapter et de l'améliorer afin de faire face aux changements survenues. De manière plus détaillée, nous exposons les principaux handicapes de la sélection incrémentale jointe naïve :

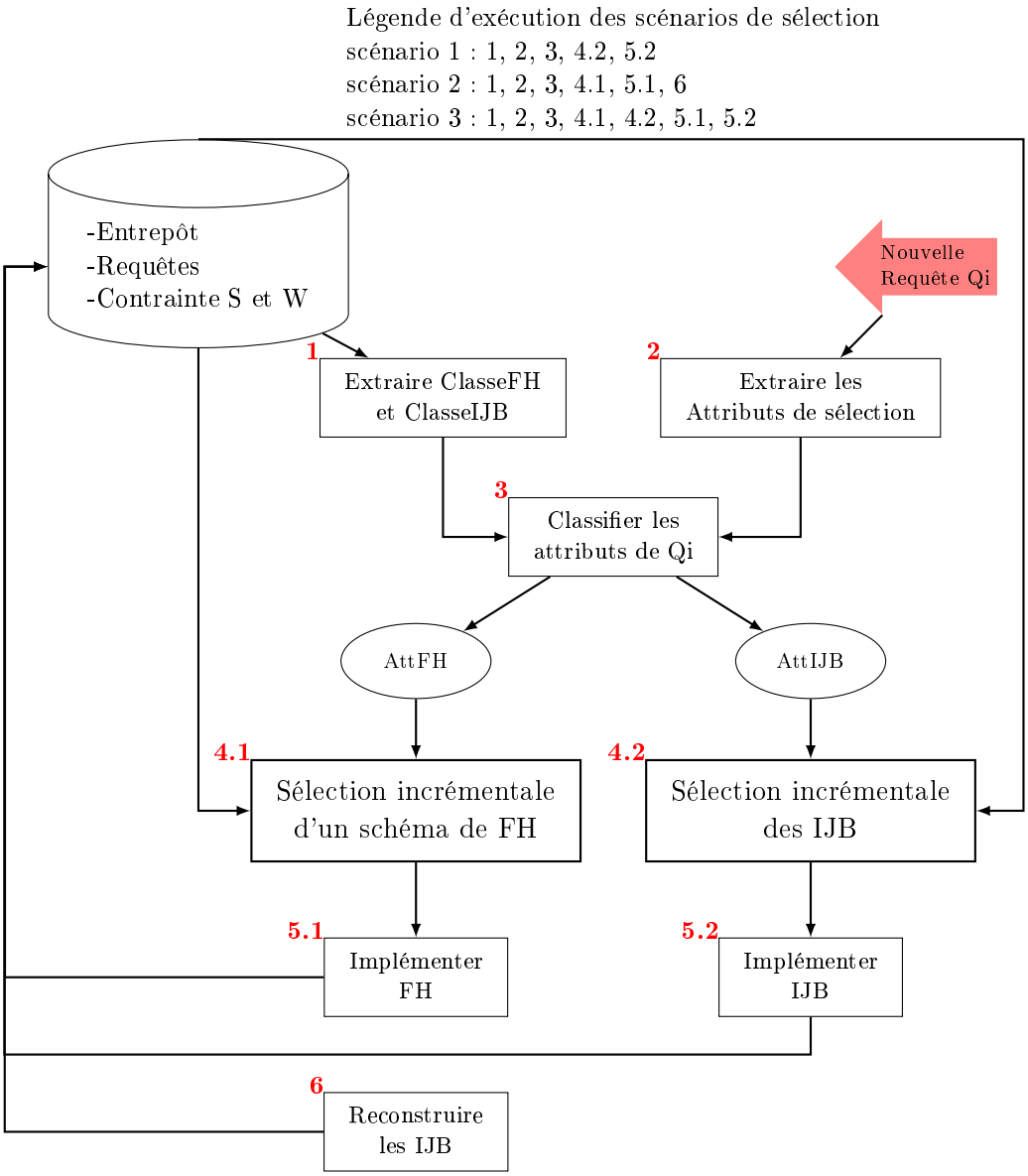


FIGURE 4.7 – Architecture de sélection incrémentale jointe de FH et IJB

- Refaire la classification des attributs peut aboutir à une classification d'attributs complètement différents des deux classes existantes. Par conséquent, après lancement des algorithmes de sélection, la configuration des structures d'optimisation finales (schéma de fragmentation et configuration d'index) peut être complètement différente que celle déjà implémentée sur l'entrepôt. Cela engendre un coût de maintenance élevé lors de son implémentation.
- Lancer systématiquement deux nouvelles sélections peut s'avérer inutile si on est en présence du scénario 1 ou 2.
- Lancer les algorithmes de sélections sur une classe d'attributs (ClasseFH ou ClasseIJB) engendre un espace de recherche grand.
- Ne pas prendre en compte les structures déjà implémentées sur l'entrepôt ne respecter pas le

Algorithme : Sélection incrémentale jointe de FH et IJB
Entrée
$\mathcal{Q}$: charge de requêtes.
Q_i : nouvelle requête exécutée.
D : ensemble des tables dimensions.
Sortie : Entrepôt optimisé.
Notation
AnalyseQ : analyse syntaxique de la charge $\mathcal{Q}$.
Début
AS⟵ AnalyseQ($\mathcal{Q}$) ;
ClasseIJB⟵ getClassIJB() ;
ClasseFH⟵ getClasseFH(D, AS) ;
ClassifierAttributsQi(ClasseFH, ClasseIJB, Q_i, AttFH, AttIJB) ;

Si (AttFH $\neq \varnothing$ et AttIJB $\neq \varnothing$) **Alors**
  SelectionIncrémentaleFH() ;
  SelectionIncrémentaleIJB() ;
Fin Si
Si (AttFH $\neq \varnothing$ et AttIJB $= \varnothing$) **Alors**
  SelectionIncrémentaleFH() ;
  ReconstruireIJB() ;
Fin Si
Si (AttFH $= \varnothing$ et AttIJB $\neq \varnothing$) **Alors**
  SelectionIncrémentaleIJB() ;
Fin Si

Algorithme 20: Algorithme de sélection incrémentale jointe de $\mathcal{FH}$ et $\mathcal{IJB}$

principe incrémentale. Il faut extraire les structures implémentées (schéma de fragmentation et/ou configuration d'index) et construire une configuration initiale de structures d'optimisation à partir de l'existant et de la nouvelle requête, puis lancer l'algorithme de sélection. Cela simplifie l'espace de recherche, le temps d'exécution de l'algorithme de sélection et le temps d'implémentation des structures d'optimisation finales.

A partir de cette analyse, nous exposons une nouvelle sélection incrémentale jointe des deux TOs. L'architecture de l'approche que nous proposons est illustrée sur la figure 4.7. Nous considérons un entrepôt de données optimisé par un schéma de fragmentation et un schéma d'indexation préalablement sélectionnés et implémentés par notre sélection jointe statique avec partage des attributs de sélection. Lorsqu'une nouvelle requête est exécutée, la sélection incrémentale jointe se déclenche et exécute les opérations suivantes (algorithme 20).

1. Extraire les attributs de sélection à partir de la requête.
2. Extraire les deux classes d'attributs ClasseFH et ClasseIJB à partir de l'entrepôt.
3. Classifier les attributs de la requêtes afin de déterminer le scénario à exécuter et pour quelle TO ils sont candidats.
4. Selon le scénario désigné (1, 2 ou 3), exécuter la sélection adéquate.
   – Exécuter la sélection incrémentale d'un schéma de fragmentation (scénario 2 et 3).

```
Fonction getClasseIJB() : ClasseIJB
    Sortie : ClasssIJB : Ensemble d'attributs sur lesquels sont défini les index implémentés.
    Notation
      ExecuteReq(req) : exécuter la requête SQL req.
    Début
      EnsIJB ⟵ ExecuteReq(SELECT Index_Name
                          FROM User_Indexes
                          WHERE Index_Type = 'BITMAP'
                          AND Join_Index='YES';);
      Pour IJB_i dans EnsIJB faire
          ClasseIJB ∪ ⟵ ExecuteReq(SELECT Column_Name
                                   FROM User_Ind_Columns
                                   WHERE Index_Name = IJB_i ;);
      Fin Pour
    Retourner ClasseIJB
Fin
```

Algorithme 21: Fonction getClasseIJB()

- Exécuter la sélection incrémentale d'un schéma d'indexation (scénario 1 et 3).
- Reconstruire les index (scénario 2).

4.3.2.1 Extraire ClasseFH et ClasseIJB

Afin de savoir à quelle classe appartient les attributs issus de la nouvelle requête exécutée, il faut extraire les attributs sur lesquelles sont définis les index (ClasseIJB) et ceux sur lesquels est défini le schéma de fragmentation (ClasseFH). Pour ClasseIJB, nous employons les métadonnées à savoir les tables **User_Indexes** et **User_Ind_Columns**. L'algorithme 21 expose la fonction getClasseIJB qui permet d'extraire et de construire les attributs d'indexation actuels.

Afin d'extraire les attributs de fragmentation, nous utilisons la fonction getPartition(), présenté dans le chapitre 2, section 2.3.1. Nous partons du principe qu'un découpage du domaine d'un attribut en sous-ensembles de sous-domaines permet de définir un schéma de fragmentation. Ainsi, pour savoir si un attribut à participé au processus de fragmentation d'une table, on exécute la fonction getPartition(). Cette fonction permet dans un premier lieu d'obtenir le domaine physique d'un attribut $\text{PhyDom}(A_i)$ contenant les valeurs physiquement présentes dans l'entrepôt. Puis, à partir de la table dimension auquel appartient l'attribut, la fonction extrait une première répartition en sous-ensembles du domaine physique où chaque sous-ensemble contient les valeurs de l'attribut appartenant au même fragment. Ces sous-ensemble sont épurés afin d'obtenir une partition de l'ensemble $\text{PhyDom}(A_i)$. Rappelons que les sous-ensembles $\text{Ens}_1, \cdots, \text{Ens}_p$ constituent une partition de $\text{PhyDom}(A_i)$ si les conditions suivantes sont vérifiées.

- $Ens_j \neq \varnothing$
- $\bigcup Ens_j = \text{PhyDom}(A_i)$
- $Ens_j \cap Ens_k = \varnothing$ si $j \neq k$

Si cette partition contient au moins 2 ensembles, alors l'attribut participe au processus de sélection. L'attribut n'est pas un attribut de fragmentation sinon. L'algorithme 22 expose la fonction

Fonction getClasseFH(D, AS) : **ClasseFH**
Entrées :
D : ensemble de d tables dimensions $\{D_1, \cdots, D_d\}$
AS : ensemble de n attributs non-clé des tables dimensions $AS = \{A_1, \cdots, A_n\}$
Sortie : ClasseFH : Ensemble d'attributs sur lesquels est défini le SF actuel.
Notations :
getTable(A_i) : Retourne la table dimension à laquelle appartient A_i.
ExecuteReq(req) : exécuter la requête SQL req.
PhyDom : le domaine de valeurs d'un attribut présentes dans l'entrepôt.
Début
Pour A_i dans $AS = \{A_1, \cdots, A_n\}$ **faire**
$D_k \longleftarrow$ getTable(A_i) ;
$EnsP \longleftarrow$getPartition(A_i, D_k) ;
$PhyDom(A_i) \longleftarrow$ ExecuteReq(SELECT DISTINCT(Ai) FROM Dk ;) ;
Si ($EnsP \neq PhyDom(A_i)$) **Alors**
ClasseFH $\cup \longleftarrow A_i$
Fin Si
Fin Pour
Retourner ClasseFH
Fin

Algorithme 22: Fonction getClasseFH()

getClasseFH qui extrait la classe des attributs de fragmentation à partir de l'entrepôt partitionné.

4.3.2.2 Classifier les attributs de la nouvelle requête

Afin de savoir s'il faut mettre à jour le schéma de fragmentation, le schéma d'indexation ou les deux, il faut connaître la nature des attributs issus de la nouvelle requête exécutée. Notant que les attributs extraits de la nouvelle requête Q_i sont soient nouveaux (interrogés pour la première fois par une requête) soient existants (déjà interrogés par au moins une requête et sont présents soit dans ClasseFH soit dans ClasseIJB). Dans les deux cas, on effectue la classification d'un attribut de sélection A_j issu de Q_i afin de savoir s'il faut l'affecter à ClasseIJB ou à ClasseFH, en employant le principe de classification des attributs par k-means.

Dans un premier lieu, le poids de l'attribut est calculé en sommant les trois critères de partage normalisés Frc_j, FS_j et $Card_j$ représentent respectivement la fréquence d'utilisation par les requêtes, le facteur de sélectivité et la cardinalité de l'attribut. Seule la cardinalité de l'attribut ne change pas. Cependant, La fréquence d'utilisation par les requêtes change à cause de l'évolution de charge et le facteur de sélectivité change également du fait de l'évolution d'instances. Ensuite, nous calculons les centroïdes de chaque classe ClasseFH et ClasseIJB. Puis, nous déterminons la distance euclidienne entre A_j et chaque centroïde. A_j sera classifié pour l'indexation si la distance euclidienne entre lui et le centroïde de ClasseIJB est la plus petite, il sera classifié pour la fragmentation sinon. Il est à noter que si l'attribut existe déjà, il est d'abords supprimé de la classe à laquelle il appartient, puis sa classification et réalisée. Nous rappelons que le calcul d'un centroïde Ct_k avec les coordonnées

Procédure ClassifierAttributsQi(ClasseFH, ClasseIJB, Q_i, AttFH, AttIJB)
Sortie :
AttFH : ensemble des attributs de Q_i classifiés pour la fragmentation.
AttIJB : ensemble des attributs de Q_i classifiés pour l'indexation.
Notation
getAttributs(Q_i) : extrait les attributs de sélection à partir de Q_i
getFrc(A_j) : retourne la fréquence d'utilisation de A_j
getFS(A_j) : retourne le facteur de sélectivité de A_j
getCard(A_j) : retourne la cardinalité de A_j
getCentroide : renvoit le centroïde d'une classe
DistanceEuclidienne(x, y) : calcul la distance euclidienne entre x et y
Début
AttQi⟵ getAttributs(Q_i) ;
Pour A_j dans AttQi **faire**
ClasseFH ⟵ ClasseFH \ $\{A_j\}$;
ClasseIJB ⟵ ClasseIJB \ $\{A_j\}$;
Poids$_j$ ⟵ getFrc(A_j) + getFS(A_j) + getCard(A_j) ;
$CoordA_j$ ⟵ (j, Poids$_j$) ;

Ct_{FH} ⟵ getCentroide(ClasseFH) ;
Ct_{IJB} ⟵ getCentroide(ClasseIJB) ;
DE_{FH} ⟵ DistanceEuclidienne($CoordA_j$, Ct_{FH}) ;
DE_{IJB} ⟵ DistanceEuclidienne($CoordA_j$, Ct_{IJB}) ;

Si ($DE_{FH} \leq DE_{IJB}$) **Alors**
AttFH ⟵ AttFH ∪ $\{A_j\}$;
ClasseFH ⟵ ClasseFH ∪ $\{A_j\}$;
Sinon
AttIJB ⟵ AttIJB ∪ $\{A_j\}$;
ClasseIJB ⟵ ClasseIJB ∪ $\{A_j\}$;
Fin Si
Fin Pour
Fin

Algorithme 23: Procédure ClassifierAttributsQi() : Classification des attributs de Q_i

(Ct_1^k, Ct_2^k) d'une classe de t attributs $A_1, \cdots, A_t$, où chaque attribut A_j est muni des coordonnées (A_1^j, A_2^j), est donné par la formule suivante :

$$Ct_p^k = \frac{\sum_{j=1}^{t} A_p^j}{t}$$

La distance euclidienne entre A_j et un centroïde Ct_k est donnée par la formule suivante :

$$\sqrt{\sum_{p=1}^{2}(A_p^j - Ct_p^k)^2}$$

L'algorithme 23 résume le processus de classification des attributs par la procédure ClassifierAttributsQi().

4.3.2.3 Exécuter un scénario de sélection incrémentale

A l'issu de l'étape de classification des attributs de Q_i, nous obtenons deux ensembles ; AttIJB et AttFH. Selon que ces ensembles soient vides ou non, nous pouvons exécuter un scénario de sélection incrémentale jointe.

Exécuter le Scénario 1 lorsque AttIJB $\neq \varnothing$ et AttFH $= \varnothing$. Ainsi, nous exécutons notre approche de sélection incrémentale isolée d'un schéma d'indexation présentée dans le chapitre 1. Nous rappelons les principales étapes qui composent cette approche.

1. Extraire la configuration d'index ConfigIJB actuellement implémentée sur l'entrepôt en utilisant les tables des métadonnées de l'entrepôt **User_Indexes** et **User_Ind_Columns**.
2. Générer l'ensemble d'index QiIJB pouvant être défini sur la requête Q_i. Ces d'$\mathcal{IJB}$ peuvent être simples ou multiples. On peut dont utiliser le codage d'index IS (index simples), $MIQ*$ (index multiples) ou les deux.
3. A partir des deux précédentes étapes, former une configuration initiale d'index ConfigIJB $\cup$ QiIJB. La configuration initiale est codée en chromosome selon le type d'index choisi.
4. Sélectionner une configuration finale d'index en utilisant un algorithme génétique guidé par un modèle de coût mathématique. Le modèle de coût permet de développer la fonction objectif de l'algorithme génétique.
5. Implémenter la configuration finale d'index C_f, en implémentant uniquement les nouveaux index ne figurant pas dans l'entrepôt et en supprimant de l'entrepôt les index obsolètes.

Exécuter le Scénario 2 lorsque AttIJB $= \varnothing$ et AttFH $\neq \varnothing$. Ainsi, nous exécutons notre approche de sélection incrémentale isolée d'un schéma de fragmentation présentée dans le chapitre 2. Nous rappelons les principales étapes qui composent cette approche.

1. Générer le schéma de $\mathcal{FH}$ actuel des tables dimensions ActuelSF. Pour ce faire, il faut identifier pour chaque dimension les attributs de fragmentation et les valeurs d'attributs qui ont participé à son processus de fragmentation. Ensuite, en exécutant la fonction GenererActuelSF(), nous construisons le schéma actuel de l'entrepôt. Rappelons qu'un schéma de fragmentation est un tableau de vecteurs où chaque vecteur représente un attribut de fragmentation et chaque case de vecteur contient un ensemble de sous-domaine comme le montre le tableau 4.2.

Age	[0, 25[	[45, 90[
Ville	Alger, Oran, Blida	Kala, Jijel, Annaba

TABLE 4.2 – Exemple d'un schéma de fragmentation actuel d'un $\mathcal{ED}$

2. Analyser la nouvelle requête exécutée pour extraire les attributs de sélection et leurs sous-domaines respectifs.
3. Générer un schéma de fragmentation initial appelé InitSF représentant la solution initiale à soumettre à l'algorithme de sélection. InitSF est construit en ajoutant au schéma ActuelSF les attributs et valeurs extraits de la nouvelle requête.
4. Exécuter l'algorithme génétique pour sélectionner un schéma de fragmentation final FinalSF. L'algorithme génétique exploite un chromosome qui représente un codage du schéma InitSF et est guidé par une fonction objectif définie à partir d'un modèle de coût mathématique.

5. Implémenter le schéma FinalSF sur l'entrepôt de données en employant les opérations SPLIT PARITION et MERGE PARRTITION sous le SGBD Oracle.

Nous avons constaté que lorsque le schéma de fragmentation des tables de l'entrepôt évolue, il faut également mettre à jour le schéma de fragmentation des index, en d'autre terme, *reconstruire les index*. Pour ce faire, deux syntaxes Oracle existent. La première syntaxe vise à recalculer les index et la seconde syntaxe vise à supprimer puis à recréer les index. Les syntaxes sont présentées comme suit :

- Recalculer l'index. Cette opération est effectuée par la syntaxe REBUILD. L'opération REBUILD est utilisée pour réorganiser ou modifier les caractéristiques de stockage d'un index existant. Le principe est de créer un nouvel index à partir de son ancienne version puis de supprimer cette dernière. Cela signifie clairement que cette opération nécessite deux fois l'espace de stockage réservé à l'index en cour de recalcule (l'ancienne et la nouvelle version). La requête SQL sous Oracle pour la reconstruction d'un index est donnée comme suit :
  ```
  ALTER INDEX nom_index REBUILD;
  ```
  L'utilisation de cette syntaxe présente deux problèmes. D'abords, le recalcule des index nécessite deux fois plus d'espace de stockage. De plus, il n'est pas possible de reconstruire un index partitionné. Oracle exige la reconstruction de l'index partition par partition ce qui est très fastidieux. La syntaxe est la suivante :
  ```
  ALTER INDEX nom_index REBUILD PARTITION part_x;
  ```
- Supprimer et recréer l'index. Vu les inconvénients que présente la syntaxe REBUILD, nous proposons une seconde solution pour mettre à jour l'index qui consiste à exécuter l'opération DROP INDEX pour supprimer l'index suivie de l'opération CREATE INDEX de manière locale pour obtenir une nouvelle version de l'index. Cependant, la requête CREATE INDEX nécessite de connaitre les attributs et tables dimensions qui participent à la construction de chaque index. Par conséquent, avant de supprimer un index, nous récupérons la requête SQL de sa création à partir des métadonnées de l'entrepôt. Ceci en exécutant la fonction GET_DDL() du package DBMS_METADATA comme suit :
  ```
  SELECT  DBMS_METADATA.GET_DDL('INDEX','nom_index')
  FROM DUAL;
  ```
  Le résultat d'exécution de cette requête est le script de création de l'index nom_index. Après suppression de l'index, il suffit d'exécuter ce script pour le recréer. L'algorithme 24 présence le processus de reconstruction des index de jointure binaires

Exécuter le Scénario 3 lorsque AttIJB $\neq \varnothing$ et AttFH $\neq \varnothing$. Il faut mettre à jour le schéma de fragmentation en exécutant notre approche de sélection incrémentale isolée d'un schéma de fragmentation présentée dans le chapitre 2 qui se conclue par l'implémentation du schéma de fragmentation sélectionné, puis nous exécutons la sélection incrémentale isolée d'un schéma d'indexation présentée dans le chapitre 1 et nous implémentons de manière locale la configuration d'index sélectionnée.

4.4 Expérimentation

Pour évaluer notre approche de sélection incrémentale jointe des deux TOs ; la fragmentation et l'indexation, nous effectuons des tests expérimentaux avec une machine Intel CORE i5 ayant une capacité mémoire de 6Go sur un entrepôt réel issu du benchmark APB1 [40] sous le SGBD Oracle 11g. L'entrepôt est composé d'une table de faits *Actvars*(24 786 000 tuples) et quatre tables de dimension

```
Procédure ReconstruireIJB()
    Notation
      ExecuteReq(req) : exécuter la requête SQL req.
      Début
      EnsIJB ⟵ ExecuteReq(SELECT Index_Name
                          FROM User_Indexes
                          WHERE Index_Type = 'BITMAP'
                          AND Join_Index='YES';);
      Pour IJB_i dans EnsIJB faire
        ReqIJB ⟵ ExecuteReq( select DBMS_METADATA.GET_DDL('INDEX','IJB_i')
                             from DUAL;);
        ExecuteReq(DROP INDEX IJB_i;);
        ExecuteReq(ReqIJB;);
      Fin Pour
Fin
```

Algorithme 24: Fonction ReconstruireIJB()

Prodlevel (9000 tuples), *Custlevel* (900 tuples), *Timelevel* (24 tuples) et *Chanlevel* (9 tuples). Pour nos tests, nous utilisons une charge de 60 requêtes qui génèrent 18 attributs de sélection *(Line, Day, Week, Country, Depart, Type, Sort, Class, Group, Family, Division, Year, Month, Quarter, Retailer, City, Gender and All)* qui ont respectivement les cardinalités suivantes : 15, 31, 52, 11, 25, 25, 4, 605, 300, 75, 4, 2, 12, 4, 99, 4, 2, 3. Le coût de la charge sans optimisation est de 39.5 millions E/S.

Vu que dans la littérature, aucun travail ne traite de la sélection incrémentale jointe de la fragmentation et de l'indexation, nous proposons de comparer notre approche avec les deux approches de sélection isolés présentées dans les chapitres précédents. En résumé, nous comparons les trois approches suivantes :

- Sélection incrémentale jointe de $\mathcal{FH}$ et $\mathcal{IJB}$ que nous appelons $SIJFI$. Nous avons choisi d'implémenter les $\mathcal{IJB}$ multiples avec le codage $MIQ*$.
- Sélection incrémentale isolée de $\mathcal{FH}$ que nous appelons $SIFH$.
- Sélection incrémentale isolée des $\mathcal{IJB}$ que nous appelons $SIJB$.

Afin de réaliser notre étude expérimentale, nous exécutons les trois étapes suivantes :

1. Etape 1 : exécuter $SIJFI$. Nous considérons un entrepôt de données déjà fragmenté et indexé suite à l'exécution de la sélection jointe statique de $\mathcal{FH}$ et $\mathcal{IJB}$ sur une charge de 50 requêtes. L'étude incrémentale est réalisée avec ajout successif de 10 nouvelles requêtes en considérant une contrainte $W = 100$ sur le nombre maximum des fragments faits et un espace de stockage des index $S = 2.5Go$. L'exécution de chaque nouvelle requête déclenche une $SIJFI$ qui permet de mettre à jour le schéma de fragmentation et/ou le schéma d'indexation. Le temps d'exécution de la sélection jointe statique avec implémentation des structures est de 66mn.

   A l'issu de la sélection jointe statique sur la charge des 50 requêtes, nous obtenons le résultat de classification des attributs de sélection suivant :
   - ClasseFH = {Gender, Month, Year, All, Quarter, Type, Line, City}.
   - ClasseIJB = {Family, Division, Class, Retailer, Group}.

   Nous constatons que 5 attributs ne sont pas utilisés par la charge de requêtes et qui sont Day, Week, Country, Depart, Sort. Ces attributs figurent dans les 10 nouvelles requêtes qui vont être

exécutées. Le tableau 4.3 résume pour chaque nouvelle requête exécutée ses attributs de sélection et la technique d'optimisation mise à jour, en d'autres termes le scénario d'optimisation qui s'est exécuté. A la fin du déroulement du test, la nouvelle répartition des deux classes d'attributs est la suivante :
- ClasseFH = {Gender, Month, Year, All, Quarter, Group, Type, Line, Division}.
- ClasseIJB = {Family, Class, Retailer, Day, Week, Country, Sort, Depart, City}.

Req	Attributs	TO mise à jour	Commentaire
Q50	All, Quarter, Type	FH	-
Q51	Group	FH	Group passe à ClasseFH
Q52	Day*, Week*	IJB	*Nouvel attribut à ClasseIJB
Q53	Gender, Month, Class	FH et IJB	-
Q54	Day, Type, Gender, Division	FH et IJB	City passe à ClasseIJB
Q55	Division	IJB	-
Q56	Family, Division, Group, City	FH et IJB	Division passe à ClasseFH
Q57	Country*, Depart*	IJB	*Nouvel attribut à ClasseIJB
Q58	Sort*, Depart	IJB	*Nouvel attribut à ClasseIJB
Q59	Month	FH	-
Q60	Day, Week, Month, Year	FH et IJB	-

TABLE 4.3 – Description des nouvelles requêtes, attributs et les TOs mises à jour : $SIJFI$

Nous constatons que chaque nouvel attribut est classifié pour l'indexation car selon les critères de classification, on favorise pour la fragmentation les attributs les plus fréquemment utilisés par la charge de requête vu son caractère non redondant et sa tolérance face à l'indexation. En effet, la sélection des index est binaire ; un attribut est soit sélectionné dans la configuration finale d'index soit écarté, alors que pour la sélection d'un schéma de fragmentation, un attribut peut être sélectionné avec un nombre réduit de fragments. Nous remarquons également que certains attributs migrent d'une classe à une autre. En effet, pour chaque nouvelle requête, une classification de ses attributs est effectuée et si ces attributs existent déjà dans une classe d'attributs ils peuvent être classifiés pour une autre. D'une part, les attributs Group et Division migrent vers ClasseFH car leur poids augmente vu l'augmentation de leurs fréquence d'utilisation par les requêtes (critère Frc du poids). D'autre part, l'attribut City est affecté à ClasseIJB après l'exécution de la requête Q54 car sa fréquence d'utilisation par les requêtes diminue.

2. Etape 2 : exécuter $SIFH$. Nous considérons un entrepôt de données déjà fragmenté par l'exécution de la sélection statique isolée de $\mathcal{FH}$ sur une charge de 50 requêtes. Pour l'étude incrémentale, on exécute successivement les 10 nouvelles requêtes en considérant une contrainte $W = 100$.
3. Etape 3 : exécuter $SIJB$. Nous considérons un entrepôt de données déjà indexé par l'exécution de la sélection statique isolée des $\mathcal{IJB}$ sur une charge de 50 requêtes. Pour l'étude incrémentale, on exécute successivement les 10 nouvelles requêtes en considérant une contrainte $S = 2.5Go$.

Pour chacune des trois étapes d'expérimentation, et pour chaque nouvelle requête exécutée, nous relevons trois informations : (1) le coût d'exécution de la charge de requête évoluée en présence du nouveau schéma d'optimisation implémenté. (2) Le taux de requêtes optimisées par les nouvelles structures sélectionnées. (3) Le temps de maintenance représentant le temps d'implémentation du nouveau schéma d'optimisation sélectionné. Notons que la charge de requêtes évoluée représente la charge des 50 requêtes plus les nouvelles requêtes exécutées jusque-là.

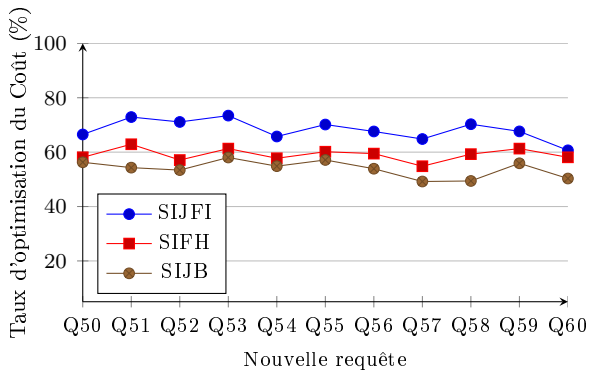


FIGURE 4.8 – Taux d'optimisation du coût : $SIJFI$, $SIFH$, $SIJB$

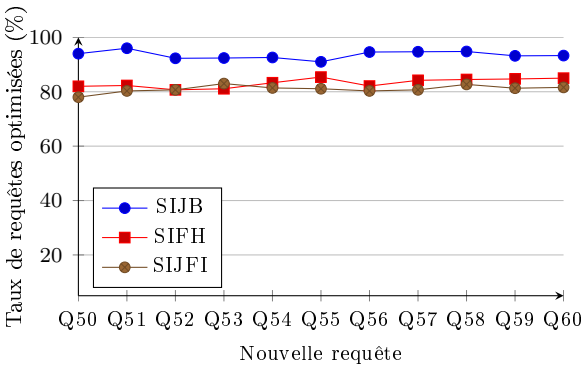


FIGURE 4.9 – Taux de requêtes optimisées : $SIJFI$, $SIFH$, $SIJB$

4.4.1 Test 1 : Évaluation théorique

Le but de ce test est d'évaluer le coût d'exécution de la charge de requêtes et le taux de requêtes optimisées pour chacune des approches de sélection ($SIJFI$, $SIFH$ et $SIJB$), afin de comparer les schémas d'optimisations sélectionnés pour chaque approche et après exécution de chaque nouvelle requête. Le coût d'exécution des requêtes est évalué par modèle de coût mathématique. Les résultats de ce test sont résumés dans la figure 4.8.

La figure 4.8 montre que l'approche $SIJFI$ réduit le coût d'exécution des requêtes de 68% en moyenne avec en moyenne 94% des requêtes optimisées alors que $SIFH$ donne un taux d'optimisation du coût de 60% avec 84% des requêtes optimisées en moyenne et $SIJB$ un taux de 54% avec 81% des requêtes optimisées en moyenne. L'analyse des résultats montre que la sélection jointe de la fragmentation et de l'indexation $SIJFI$ donne une meilleure optimisation que les approches de sélection isolée car elle permet de couvrir l'optimisation d'avantage de requêtes (94%) et choisit à travers la classification, les attributs les plus bénéfiques pour chaque technique d'optimisation.

4.4.2 Test 2 : Évaluation pratique sous Oracle 11g

Dans le second test, nous comparons les trois approches de sélection incrémentale selon le temps de maintenance de l'entrepôt de données sous le SGBD Oracle 11g. En d'autres termes, le temps d'implémentation des structures d'optimisations sélectionnées par chaque sélection et pour chaque nouvelle requête. Nous donnons quelques statistiques sur l'étude effectuée. Le temps de maintenance d'un schéma de fragmentation est le temps d'exécution des opérations SPLIT, MERGE et MOVE où chacune dure en moyenne 1mn à 3mn. Rappelons que l'exécution d'une opération sur la table de dimension engendre sa propagation sur la table de faits. Le temps de maintenance d'une configuration d'index représente le temps d'implémentation des nouveaux index et le temps de suppression des index obsolètes. Pour une contrainte $S = 2.5Go$, le nombre d'index sélectionné est en moyenne 13 index. L'implémentation d'un index dure en moyenne 1mn à 2mn et le temps de suppression en moyenne 1s. Le temps de reconstruction de 13 index est en moyenne de 18mn.

Les résultats de l'étude expérimentale sont illustrés sur la figure 4.10. L'analyse des résultats est répartie en trois catégories selon le résultat de classifications des attributs des nouvelles requêtes par l'approche $SIJFI$.

1. Attributs classifiés pour l'indexation (Q52, Q55, Q57 et Q58). Pour ces requêtes, l'approche

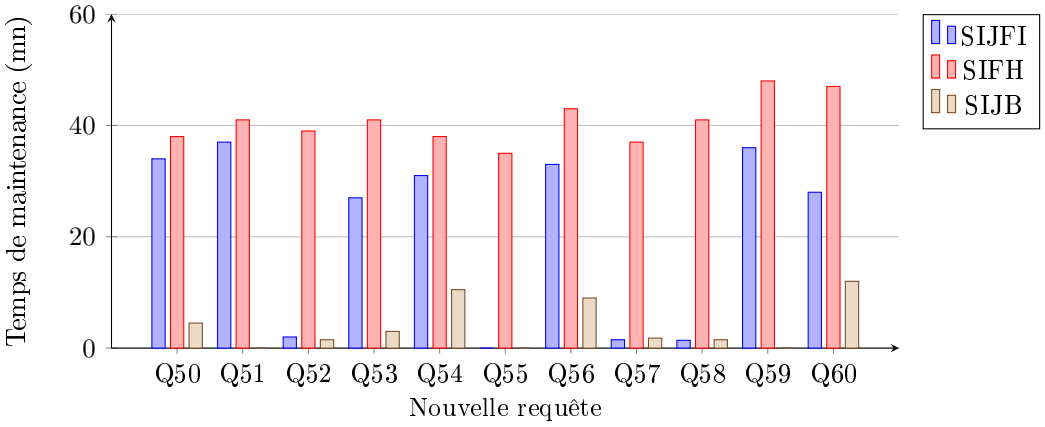


FIGURE 4.10 – Temps de maintenance de l'entrepôt sous Oracle 11g : $SIJFI$, $SIFH$ et $SIJB$

$SIJFI$ déclenche une nouvelle sélection des index ce qui est similaire à l'approche de sélection isolée $SIJB$. L'approche $SIJFI$ donne les temps de maintenance respectifs 2mn, 0mn, 1.5mn et 1.4mn, $SIJB$ donne les temps de maintenance resp. 1.5mn, 0mn, 1.8mn et 1.5mn et $SIFH$ donne resp. 39mn, 35mn, 37mn et 41mn. Nous constatons que les approches $SIJFI$ et $SIJB$ apportent un temps de maintenance énormément réduit par rapport à l'approche $SIFH$. Effectivement, la mise à jour d'une configuration d'index (suppression, création d'index) est une opération moins complexe que la mise à jour d'un schéma de fragmentation des tables qui peut nécessiter plusieurs opérations de fusions et/ou éclatements et/ou déplacements de fragments. De plus, l'approche $SIFH$ effectue une reconstruction systématique de toute la configuration d'index à chacune de ses exécution.

2. Attributs classifiés pour la fragmentation (Q50, Q51 et Q59). L'approche $SIJFI$ classifie les attributs de sélection pour la fragmentation. Ainsi, seule une nouvelle sélection d'un schéma de fragmentation est requise, ce qui est similaire à l'approche isolée $SIFH$. Les deux approches effectuent une sélection d'un nouveau schéma de fragmentation suivi d'une reconstruction des index. Cependant, le temps de maintenance du schéma de fragmentation engendré par $SIJFI$ (36mn en moyenne) est réduit par rapport au temps de maintenance engendré par l'approche isolée $SIFH$ (42mn en moyenne). Cela est due à l'élagage de l'espace de recherche de la fragmentation pour $SIJFI$. L'élagage permet de réduit la complexité du processus de sélection pour $SIJFI$ et génère un nouveau schéma de fragmentation qui nécessite moins d'opérations de mise à jour que pour l'approche isolée $SIFH$. Quant à l'indexation isolée $SIJB$, elle génère un temps de maintenance moyen de 1.5mn ce qui appuie notre conclusion sur la complexité de mettre à jour un schéma de fragmentation par rapport à un schéma d'indexation.
3. Attributs classifiés pour l'indexation et la fragmentation (Q53 Q54 Q56 Q60). Pour ces requêtes, l'approche jointe $SIJFI$ effectue les deux sélections et met donc à jour le schéma de fragmentation et la configuration d'index, ce qui donne un temps de maintenance moyen de 29mn, alors que les approches isolées donnent 42mn et 8.5mn pour l'approche $SIFH$ et $SIJB$ respectivement.

Nous remarquons que pour l'approche $SIJB$, les requêtes Q51, Q55 et Q59 donne un temps de maintenance 0. La raison est que ces requêtes sont mono-attributs et ne peuvent définirent un index

multiple, donc aucun changement n'est effectué sur l'entrepôt.

4.5 Conclusion

Nous avons proposé et développé dans ce chapitre une approche de sélection incrémentale jointe qui sélectionne conjointement des structures d'optimisation définies sur deux techniques d'optimisation ; la fragmentation horizontale et les index de jointure binaires. Nous avons commencé par présenter nos travaux antérieurs sur la sélection jointe statique de la fragmentation et de l'indexation. Nous avons utilisé la dépendance faible-forte entre l'indexation et la fragmentation afin de séparer leurs processus de sélection tout en prenant en compte leur concurrence sur la même ressource ; les attributs de sélection issus des requêtes de jointures en étoile. La sélection jointe statique se base sur le partage des attributs de sélection, entre les deux techniques, afin de choisir pour chaque technique les attributs les plus adéquat susceptibles de générer les structures d'optimisation les plus bénéfiques pour la charge de requêtes. Sur cette logique, nous avons développé une approche de sélection incrémentale jointe qui prend en compte l'évolution de la charge de requêtes. Les attributs de chaque nouvelle requête sont classifiés pour les affecter soit à l'indexation soit à la fragmentation soit aux deux. Cette classification déclenche un scénario d'optimisation qui met en scène les sélections isolées incrémentale de $\mathcal{FH}$ et $\mathcal{IJB}$ que nous avons proposé dans les chapitres précédents.

Nous avons appuyé notre proposition par une étude expérimentale qui permet de comparer les deux types de sélection incrémentale ; isolée et jointe. Nous avons constaté que la sélection incrémentale jointe est plus bénéfique que la sélection incrémentale isolée, car elle permet de couvrir l'optimisation d'avantage de requêtes et choisit pour chaque technique d'optimisation les attributs les plus intéressants qui permettent de sélectionner des structures d'optimisation plus bénéfiques.

Chapitre 5

Sélection Multi-Objectif de la FH

Sommaire

5.1 Introduction

Dans les démarches d'optimisation des requêtes de jointures en étoile que nous avons proposé jusque-là, plus précisément dans le contexte incrémentale, la mise à jour des structures d'optimisation déjà implémentées sur l'entrepôt de données requière un coût de maintenance considérable. Le coût de maintenance représente le temps et les ressources requis pour mettre à jour les structures d'optimisation implémentées sur l'entrepôt de données afin que celles-ci s'adaptent à l'évolution de l'entrepôt. Pour la fragmentation horizontale par exemple, lorsque l'entrepôt de donnée évolue, il faut adapter le schéma de fragmentation des tables aux changements survenus, ce qui nécessite plusieurs opérations de fusion et/ou éclatement et/ou déplacement des fragments des différentes tables. Par conséquent, il est impératif de sélectionner les structures d'optimisation qui optimisent la charge de requêtes tout en optimisant le coût de maintenance. En d'autre terme, il faut mettre en œuvre une démarche de sélection qui optimise deux objectifs : le coût de la charge de requêtes et le coût de maintenance.

Pour optimiser plusieurs objectifs, il faut formaliser le problème en un problème d'optimisation multi-objectif. L'optimisation multi-objectif est employée dans plusieurs domaines depuis plusieurs décennies, c'est ainsi qu'il existe une multitude de travaux qui proposent des approches et méthodes de résolutions et d'automatisation pour ce problème. Cependant, **dans le contexte d'entrepôt de données, seuls les travaux dans [68] proposent une formulation multi-objectif d'un problème de sélection statiques des vues matérialisées, aucun travail n'est proposé dans le contexte incrémentale**. Ainsi, dans ce chapitre nous faisons les propositions suivantes :

- Formaliser le problème de sélection incrémentale d'un schéma de fragmentation en un problème multi-objectif qui vise à optimiser le coût de la charge de requêtes et le coût de maintenance d'un schéma de fragmentation. En effet, nos tests expérimentaux menés jusque-là ont montré que par rapport aux index de jointures binaires, la fragmentation horizontale engendre un coût de maintenance considérable qu'il faut réduire.
- Proposer un modèle de coût mathématique pour évaluer le coût de maintenance d'un schéma de fragmentation implémenté sur un entrepôt déjà fragmenté. La formalisation du modèle de coût emploie l'algèbre de fragmentation que nous avons défini dans le chapitre 2.
- Proposer une résolution du problème de sélection incrémentale multi-objectif.
- Définir une architecture globale de sélection incrémentale d'un schéma de fragmentation qui prend en compte l'optimisation du coût de maintenance et qui intègre le Profiling des requêtes afin de réduire d'avantage le temps de maintenance de l'entrepôt de données [8, 7].

Ce chapitre est organisé comme suit. Dans la section 2 nous formalisons le problème de sélection multi-objectif de la fragmentation horizontale. La section 3 expose le modèle de coût de maintenance. La section 4 présente notre démarche de résolution du problème. La section 5 expose l'étude expérimentale que nous avons effectué. Enfin, la section 6 conclue le chapitre.

5.2 Formalisation du problème multi-objectif

Afin de réaliser la sélection incrémentale d'un nouveau schéma de fragmentation, nous avons proposé dans le chapitre 2 deux approches à savoir la sélection incrémentale naïve $FHNI$ et la sélection incrémentale par algorithmes génétiques $FHAG$. Le principal problème avec ces approches est que le nouveau schéma de $\mathcal{FH}$ est sélectionné sans prendre en compte son coût d'implémentation sur l'entrepôt. Par conséquent, un nouveau schéma de $\mathcal{FH}$ sélectionné peut considérablement améliorer les

performances d'exécution des requêtes mais peut être très coûteux du point de vu maintenance. Par conséquent, nous proposons une approche de refragmentation incrémentale des entrepôts de données qui optimise deux objectifs : le coût d'exécution de la charge de requêtes et le coût de maintenance du schéma de fragmentation. Le problème de sélection incrémentale multi-objectif d'un schéma de fragmentation $PMOFH$ est formalisé comme suit :
Étant donné :

- un $\mathcal{ED}$ fragmenté suivant un schéma SF et modélisé par un schéma en étoile ayant d tables de dimension $\mathcal{D} = \{D_1, D_2, \cdots, D_d\}$ et une table des faits $\mathcal{F}$,
- une charge de requêtes $Q = \{Q_1, Q_2, ..., Q_t\}$,
- une nouvelle requête Qi qui s'exécute sur l'$\mathcal{ED}$,
- un seuil W représentant le nombre maximum de fragments de la table de faits dans le schéma final SF'.

Le $PMOFH$ consiste à générer un schéma SF' de fragmentation primaire des tables dimensions tel que :

- une fois la table de faits fragmentée par une FH dérivée suivant le schéma SF', le nombre de fragments faits ne dépasse pas W,
- le coût total d'exécution des requêtes $Q \cup \{Q_i\}$ sur l'ED fragmenté avec SF' est réduit,
- **le coût de maintenance de l'entrepôt est réduit (passage de SF à SF').**

Formalisé ainsi, le $PMOFH$ est un problème d'optimisation multi-objectif où il faut minimiser deux objectifs : le coût d'exécution des requêtes et le coût de maintenance de l'$\mathcal{ED}$ sous une contrainte W sur le nombre de fragments faits générés. Le problème de sélection mono-objectif d'un schéma de fragmentation est prouvé NP-Complet [14]. Ainsi, l'ajout d'un nouvel objectif à optimiser ne réduit pas la complexité du problème. Donc le $PMOFH$ est également NP-Complet.

Nous avons présenté dans le chapitre 2 le modèle de coût pour évaluer le coût d'exécution d'une charge de requêtes sur un entrepôt partitionné que nous rappelons dans ce qui suit. Considérons l'entrepôt fragmenté en N sous schémas en étoiles $SF = S_1, ..., S_N$. Le coût d'exécution d'une charge de requêtes représente la somme des coûts d'exécution de chaque $Q_k \in Q$ sur chaque sous schéma $S_i \in SF$. On définit le coût de chargement d'un fragment fait et un fragment dimension par les formules suivantes :

- pour un fragment fait : $\prod_{j=1}^{M_i} Sel(PF_j) \times |F|$,
- pour un fragment dimension D_s :$\prod_{j=1}^{L_s} Sel(PM_j^s) \times |D_s|$,

où $|R|$ et $Sel(P)$ représentent respectivement le nombre de pages nécessaires pour stocker la table R et le facteur de sélectivité du prédicat de sélection P. Le coût d'exécution $CostE(Q, SF)$ de la charge de requêtes Q sur l'entrepôt fragmenté suivant le schéma SF est :

$$CostE(Q, SF) = \sum_{k=1}^{t} \sum_{j=1}^{NS_k} \left(3 \times \left(\prod_{j=1}^{M_i} Sel(PF_j) \times |F| + \sum_{s=1}^{d} \prod_{j=1}^{L_s} Sel(PM_j^s) \times |D_s| \right) \right) \tag{5.1}$$

Avec $NS_k \leq N$ le nombre de sous schémas valides pour chaque $Q_k \in Q$. Nous annonçons dans ce qui suit, le modèle de coût de maintenance d'un schéma de fragmentation.

5.3 Modèle de coût de maintenance basé sur l'Algèbre de Fragmentation

Nous définissons le coût de maintenance pour un nouveau schéma de fragmentation SF' comme le coût d'implémentation de SF' sur un entrepôt partitionné suivant un schéma actuel SF. Pour calculer le coût de maintenance, nous effectuons les trois étapes suivantes :

1. Utiliser l'Algèbre de Fragmentation AF, définie dans le chapitre 2, pour déterminer les opérations algébriques nécessaires pour passer du schéma SF vers SF'.
2. Déterminer les opérations physiques correspondantes à chacune des opérations algébriques trouvées précédemment.
3. Évaluer le coût de chaque opération physique afin de déterminer le coût total de maintenance.

Nous allons détailler dans ce qui suit chaque étape du processus de calcul du coût de maintenance.

5.3.1 Déterminer les opérations algébriques

Nous avons formalisé les opérations nécessaires pour passer d'un schéma actuel SF vers un nouveau schéma SF' en algèbre dont les opérations algébriques sont résumées comme suit :

- $Add\_A(A_i, \{SD^i_{j_1}, \cdots, SD^i_{j_p}\})(SF)$.
- $Add\_SD(A_i, \{SD^i_{j_1}, \cdots, SD^i_{j_p}\})(SF)$.
- $Split\_Dom(A_i, \{SD^i_{j_1}, \cdots, SD^i_{j_p}\}, \{SD^i_{k_1}, \cdots, SD^i_{k_s}\})(SF)$.
- $Merge\_Dom(A_i, \{SD^i_{j_1}, \cdots, SD^i_{j_p}\}, \{SD^i_{k_1}, \cdots, SD^i_{k_s}\})(SF)$.
- $Del\_A(A_i)(SF)$.
- $Del\_SD(A_i, \{SD^i_{j_1}, \cdots, SD^i_{j_p}\})(SF)$.

Nous présentons dans ce qui suit un exemple où nous montrons comment déterminer les opérations algébriques nécessaires pour passer d'un schéma de fragmentation à un autre.

Exemple 42 *Soit un entrepôt de données composé d'une table de faits Ventes et une dimension Clients. Considérons trois attributs de fragmentation : Genre, Ville et Métier. L'entrepôt de données est fragmenté suivant un schéma de fragmentation SF donné par le tableau 5.1 gauche ce qui donne une fragmentation de la table Clients en quatre fragments comme suit :*

- $Clients1 = \sigma_{(Genre=M) \wedge (Ville \text{ in } (Alger, Oran, Blida))}(Clients)$
- $Clients2 = \sigma_{(Genre=F) \wedge (Ville \text{ in } (Alger, Oran, Blida))}(Clients)$
- $Clients3 = \sigma_{(Genre=M) \wedge (Ville=Kala)}(Clients)$
- $Clients4 = \sigma_{(Genre=F) \wedge (Ville=Kala)}(Clients)$

Les opérations algébriques nécessaires pour passer du schéma SF (tableau 5.1 gauche) vers un nouveau schéma SF′ (tableau 5.1 droit) sont données par la formule suivante :
$SF' = Merge\_Dom(Ville, \{Blida\}, \{Kala\}) \circ Split\_Dom(Ville, \{Blida\}) \circ$
$Add\_A(Metier, \{M1, M2\})(SF)$.

5.3.2 Déterminer les opérations physiques

Nous avons exposé, dans la chapitre 2 de la partie II, les trois opérations physiques utilisées afin d'implémenter un nouveau SF sur un $\mathcal{ED}$ partitionné sous le SGBD Oracle, à savoir Split, Merge et Move. Nous les résumons comme suit :

- Split (P, Ct) : Supposant une table Clients sur laquelle on effectue l'opération Split(Clients, Ville='Alger'). La syntaxe Oracle est donnée comme suit :

SF

Genre	M	F
Ville	Alger,Oran,Blida	Kala

$\Longrightarrow$

SF'

Genre	M	F
Ville	Alger,Oran	Blida,Kala
Métier	M1,M2	M3,M4,M5

TABLE 5.1 – Schéma de fragmentation SF actuel et nouveau schéma SF'

```
ALTER TABLE Clients
SPLIT PARTITION Ville VALUES ('Alger')
INTO ( PARTITION Clients1, PARTITION Clients2 );
```

- Merge (P1, P2) : La syntaxe sous Oracle qui permet la fusion de deux partitions est la suivante :
```
ALTER TABLE Clients
MERGE PARTITIONS Clients1, Clients2 INTO PARTITION Clients2;
```

- Move (P1, TBS) : déplacer le fragment P1 vers le tablespace TBS, s'il n'y a plus d'espace dans le tablespace en cour. La clause Oracle est donnée comme suit :
```
ALTER TABLE Clients
    MOVE PARTITION Clients1 TABLESPACE tbs_2;
```

A la base de ces opérations, nous avons interprété l'algèbre de fragmentation au niveau physique. Nous avons repris les opérations de l'AF et pour chaque opération nous avons spécifié les opérations physiques nécessaires à son implémentation. L'implémentation de toute l'AF est détaillée dans la section 2.4.4.

Exemple 43 *Considérons l'exemple 42. Le passage de SF vers SF' nécessite trois opérations algébriques dont nous donnons l'implémentation physique comme suit :*

- *$Add\_A(Metier, \{M1, M2\})$: l'exécution de l'algorithme d'implémentation donne :*
  Ens_ Part = Identify_ Part(Métier in (M1, M2)) = {Clients1, Clients2},
  Split (Clients1, Métier in (M1, M2)) = {Clients11, Clients12},
  Split (Clients2, Métier in (M1, M2)) = {Clients21, Clients22}.

- *$Split\_Dom(Ville, \{Blida\})$ les opérations physiques sont les suivantes :*
  Ens_ Part = Identify_ Part(Ville = Blida) = {Clients11, Clients21, Clients22},
  Split (Clients11, (Ville = Blida)) = {Clients111, Clients112},
  Split (Clients21, (Ville = Blida)) = {Clients211, Clients212},
  Split (Clients22, (Ville = Blida)) = {Clients22}.

- *$Merge\_Dom(Ville, \{Blida\}, \{Kala\})$: l'exécution de l'algorithme d'implémentation donne :*
  Ens_ Part1 = Identify_ Part(Ville = Kala) = {Clients111, Clients211, Clients12},
  Ens_ Part2 = Identify_ Part(Ville = Blida) = {Clients112, Clients212, Clients22},
  Mergeable(Clients111, Clients112, Ville) = True ,
  Merge (Clients111, Clients112),
  Ens_ Part2 = {Clients211, Clients12}, Ens_ Part2 = {Clients212, Clients22},
  Mergeable(Clients211, Clients22, Ville) = False,
  Mergeable(Clients12, Clients212, Ville) = False,

Mergeable(Clients12, Clients22, Ville) = False,
Mergeable(Clients211, Clients212, Ville) = True,
Merge (Clients211, Clients212),
Ens_Part1 = {Clients12}, Ens_Part2 = {Clients22},
Mergeable(Clients12, Clients22, Ville) = False.

5.3.3 Modèle de coût de maintenance

Afin de spécifier le coût d'implémentation d'un nouveau SF, il faut évaluer le coût des opérations Split, Merge et Move. Les coûts des opérations, représentant le nombre d'entrées/sorties système, sont donnés respectivement comme suit :

1. *Coût d'une opération Split* : l'exécution d'une opération Split sur un fragment dimension, engendre l'éclatement d'un fragment FD de la table dimension D en deux fragments FD1 et FD2. Ensuite, tous les fragments faits FF_i correspondant au fragment FD doivent à leurs tour être éclatés en deux fragments $FF1_i$ et $FF2_i$. Un fragment fait FF_i correspond à un fragment dimension FD si tous les tuples de FF_i peuvent être joints avec les tuples de FD. Ainsi le coût d'une opération Split, noté CSplit, est donné par

$$CSplit = |FD| + |FD1| + |FD2| + \sum_{i=1}^{NbFF} (|FF_i| + |FF1_i| + |FF2_i|) \tag{5.2}$$

   Où |R| représente la taille de la table R en termes de pages systèmes et $NbFF$ le nombre de fragments fais correspondant au fragment dimension FD

2. *Coût d'une opération Merge* : l'opération Merge signifie la fusion de deux fragments dimension FD1 et FD2 en un seul fragment FD. Chaque deux fragments faits $FF1_i$ et $FF2_i$, correspondant resp. à FD1 et FD2, sont joints aussi en un seul fragment fait FF_i. L'opération de fusion peut être considérée comme une opération d'union des deux fragments. De ce fait, le coût de l'opération Merge, noté CMerge, est donné comme suit :

$$CMerge = |FD1| + |FD2| + |FD1 \cup FD2| + \sum_{i=1}^{NbFF} (|FF1_i| + |FF2_i| + |FF1_i \cup FF2_i|) \tag{5.3}$$

3. *Coût d'une opération Move* : Après fusion de deux fragments, le fragment résultant F peut être déplacé vers un autre tablespace si l'espace sur le tablespace en cour est insuffisant. Le fragment est réécrit sur le disque dur à l'emplacement prévu. Ainsi, le coût d'une opération Move, notée CMove, est donné par la formule suivante :

$$CMove = |FD| + \sum_{i=1}^{NbFF} |FF_i| \tag{5.4}$$

   Ainsi, le coût de maintenance CostM(SF) représente la somme des coûts de toutes les opérations de fusion, éclatement et déplacement nécessaires pour implémenter le nouveau SF. Il est donné par la formule suivante :

$$CostM(SF) = \sum_{i=1}^{Nb_{Merge}} CMerge_i + \sum_{i=1}^{Nb_{Move}} CMove_i + \sum_{i=1}^{Nb_{Split}} CSplit_i \tag{5.5}$$

Où, Nb_{Merge}, Nb_{Split} et Nb_{Move} représentent respectivement le nombre de fusions, éclatements et déplacements nécessaires pour implémenter le nouveau SF sur l'$\mathcal{ED}$. Considérons le pire des cas où chaque fusion nécessite un déplacement, le coût de maintenance est alors donné comme suit :

$$CostM(SF) = \sum_{i=1}^{Nb_{Merge}} (CMerge_i + CMove_i) + \sum_{i=1}^{Nb_{Split}} CSplit_i \quad (5.6)$$

Exemple 44 *Considérons l'exemple 43. Pour simplifier les notations dans l'exemple, nous notons Clientsi par CLi. A chaque fragment de la table Clients CLi correspond un fragment de la table des faits Ventes noté Vi. Les opérations physiques pour passer de SF vers SF' sont données comme suit :*
Split1 (CL1, Métier in (M1, M2)) = {CL11, CL12}
Split2 (CL2, Métier in (M1, M2)) = {CL21, CL22}
Split3 (CL11, (Ville = Blida) = {CL111, CL112}
Split4 (CL21, (Ville = Blida) = {CL211, CL212}
Split5 (CL22, (Ville = Blida) = {CL22}
Merge1 (CL111, CL112) = {CL11}
Merge2 (CL211, CL212) = {CL21}
En considérant le pire des cas, deux opérations Move sont nécessaires :
Move1 (CL11)
Move2 (CL21)
Nous calculons le coût de maintenance pour le nouveau schéma SF' comme suit :
$CostM(SF') = \sum_{i=1}^{2} (CMerge_i + CMove_i) + \sum_{i=1}^{5} CSplit_i$
$= CMerge1 + CMove1 + CMerge2 + CMove2 + CSplit1 + CSplit2 + CSplit3 + CSplit4 + CSplit5.$
La table de faits Vente est fragmentée par une fragmentation dérivée suivant uniquement la table Clients. A chaque fragment CLi correspond alors un seul fragment faits Vi. Donc les coûts de chaque opération est donnée comme suit :
$CMerge1 = |CL111| + |CL112| + |CL111 \cup CL112| + |V111| + |V112| + |V111 \cup V112|$
$CMove1 = |CL11| + |V11|$
$CMerge2 = |CL211| + |CL212| + |CL211 \cup CL212| + |V211| + |V212| + |V211 \cup V212|$
$CMove2 = |CL21| + |V21|$
$CSplit1 = |CL1| + |CL11| + |CL12| + |V1| + |V11| + |V12|$
$CSplit2 = |CL2| + |CL21| + CL22| + |V2| + |V21| + V22|$
$CSplit3 = |CL11| + |CL111| + CL112| + |V11| + |V111| + V112|$
$CSplit4 = |CL21| + |CL211| + CL212| + |V21| + |V211| + V212|$
$CSplit5 = |CL22| + |V22|$

5.4 Notre démarche de résolution du $PMOFH$

Afin de résoudre le $PMOFH$, nous proposons de l'amener à un problème mono-objectif en utilisant la méthode de compromis. Notre choix s'est porté sur cette méthode pour deux raisons :

1. La fonction objectif qu'il faut optimiser en priorité est le coût total d'exécution des requêtes sur l'entrepôt fragmenté.
2. Dans la formulation du problème, nous avons une contrainte W sur le nombre de fragments faits générés.

Par conséquent, nous introduisons une contrainte sur le coût de maintenance pour un nouveau schéma de fragmentation noté R. La contrainte R représente le coût maximum d'opérations d'éclatement, de fusion et de déplacement permises pour l'implémentation du nouveau schéma de fragmentation sélectionné. Ainsi, la nouvelle formulation du problème de sélection d'un schéma de fragmentation $PMOFH'$ est donnée comme suit :
Étant donné :

- un $\mathcal{ED}$ fragmenté suivant un schéma SF, modélisé par un schéma en étoile ayant d tables de dimension $\mathcal{D} = \{D_1, D_2, \cdots, D_d\}$ et une table des faits $\mathcal{F}$,
- une charge de requêtes $Q = \{Q_1, Q_2, ..., Q_t\}$,
- une nouvelle requête Qi qui s'exécute sur l'$\mathcal{ED}$,
- un seuil W représentant le nombre maximum de fragments de la table faits dans le schéma final SF',
- **une contrainte R sur le coût de maintenance de SF'.**

Le $PMOFH'$ consiste à générer un schéma SF' de fragmentation primaire des tables dimensions, défini sur l'ensemble AS, tel que :

- une fois la table de faits fragmentée par une FH dérivée suivant le schéma SF', le nombre de fragments faits ne dépasse pas W,
- le coût total d'exécution des requêtes $Q \cup \{Q_i\}$ sur l'ED fragmenté avec SF' est optimisé,
- **le coût de maintenance de l'entrepôt ne dépasse pas R.**

Formalisé ainsi, le $PSIFH'$ est un problème mono-objectif qui consiste à minimiser le coût d'exécution des requêtes sous les deux contraintes W et R.

$$\begin{cases} \text{Minimiser} & CostQ(Q, SF') \\ \text{avec} & CostM(SF') \leq R \\ \text{et} & NbFragment(SF') \leq W \end{cases} \tag{5.7}$$

5.4.1 Sélection incrémentale avec optimisation du coût de maintenance *CM*

Afin de résoudre le $PMOFH'$, nous proposons une approche de sélection incrémentale d'un SF avec optimisation du coût de maintenance basée sur les algorithmes génétiques que nous appelons *CM*. L'architecture globale de notre approche est présentée sur la figure 5.1.

Le déroulement du processus de sélection, illustré sur la figure 5.1, est similaire au processus de sélection incrémentale détaillé dans la section 2.2. Rappelons que les étapes du processus de sélection sont les suivantes :

1. Générer le schéma de fragmentation actuel de l'$\mathcal{ED}$ SF.
2. Analyser la nouvelle requête exécutée Q_i.
3. Générer un nouveau schéma de fragmentation SF_1 en ajoutant les attributs et sous-domaines issus de la requêtes Q_i au schéma actuel SF.
4. Sélectionner par AG un schéma SF' final qui optimise le coût de la charge de requête et le coût de maintenance.
5. Implémenter le SF' sur l'entrepôt de données.

Au niveau de la sélection d'un SF' final, nous employons un algorithme génétique guidé par une fonction objectif qui permet d'évaluer les différentes solutions exploitées par l'AG. Nous présentons dans ce qui suit le codage du chromosome et la formulation de la fonction objectif pour l'AG.

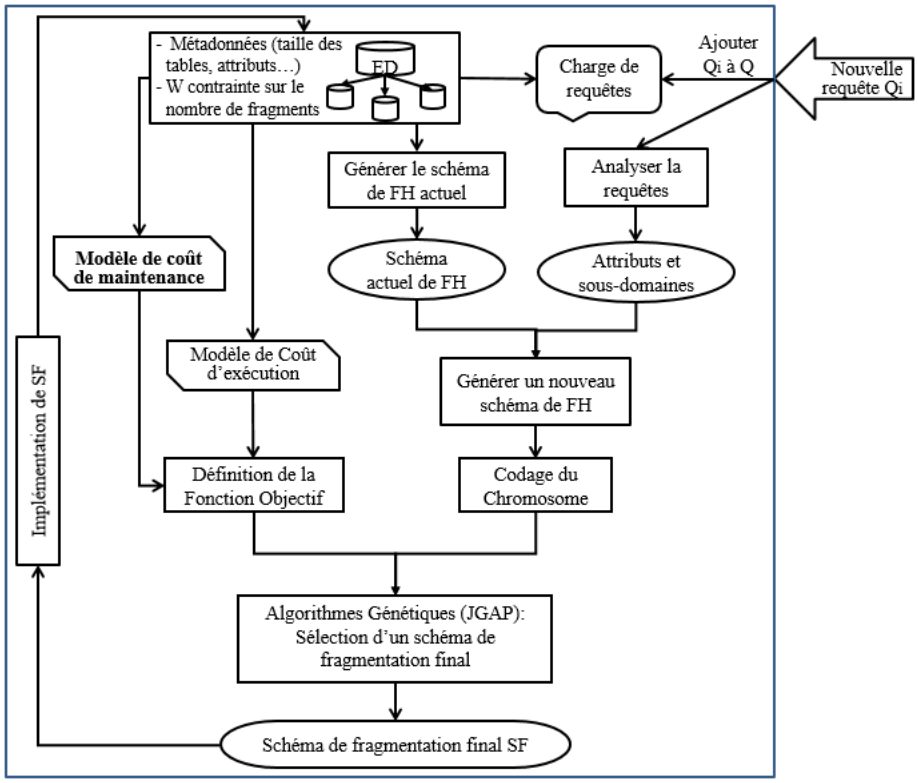


FIGURE 5.1 – Architecture de la sélection incrémentale avec coût de maintenance *CM*

5.4.1.1 Codage du chromosome pour la démarche *CM*

L'encodage d'un schéma de fragmentation sous forme d'un chromosome est représenté dans le tableau 5.2. Ce codage permet d'attribuer à chaque sous-domaine un numéro, les sous-domaines qui possèdent le même numéro sont fusionnés en un seul ensemble.

	M	F			
Genre	1	2			
	Alger	Oran	Blida	Kala	$Reste_2$
Ville	1	2	3	3	4
	P1	P2	P3	$Reste_3$	
PNom	1	2	2	3	

TABLE 5.2 – Codage du schéma de fragmentation en chromosome : démarche *CM*

5.4.1.2 Fonction objectif pour la démarche *CM*

Le problème d'optimisation mono-objectif vise à optimiser un objectif, qui est le coût de la charge de requêtes, avec deux contraintes. Étant donnée un schéma SF et une charge de requêtes Q, le

problème est formalisé comme suit :

$$\begin{cases} \text{Minimiser} & CostQ(Q, SF) \\ \text{avec} & CostM(SF) \leq R \\ \text{et} & NbFragment(SF) = N \leq W \end{cases} \tag{5.8}$$

Nous proposons d'amener ce problème à un problème d'optimisation d'un objectif sans contraintes. Pour ce faire, nous exprimons deux fonctions de pénalités. La première fonction vise à pénaliser les SF dont le nombre de fragments dépasse la contrainte W :

$$Pen1(SF) = \frac{N}{W} \tag{5.9}$$

La seconde fonction de pénalité pénalise les schémas de fragmentation SF dont le coût de maintenance dépasse R :

$$Pen2(SF) = \frac{CostM(SF)}{R} \tag{5.10}$$

Nous appliquons la première pénalité afin d'obtenir une première formulation de la fonction objectif comme suit :

$$F(SF) = \begin{cases} CostQ(Q, SF) \times Pen1(SF), & \text{Si } Pen1(SF) > 1 \\ CostQ(Q, SF), & \text{Sinon.} \end{cases} \tag{5.11}$$

Nous appliquons ensuite la seconde pénalité afin d'obtenir la formulation finale de la fonction objectif comme suit :

$$FC(SF) = \begin{cases} F(SF) \times Pen2(SF), & \text{Si } Pen2(SF) > 1 \\ F(SF), & \text{Sinon.} \end{cases} \tag{5.12}$$

5.4.2 Intégration du Profiling des requêtes dans l'approche *CM*

Nous proposons une nouvelle approche de sélection incrémentale qui combine deux approches que nous avons développé et qui sont la sélection incrémentale avec optimisation du coût de maintenance *CM* et la sélection incrémentale avec Profiling des requêtes *ProfilQ* développée dans le chapitre 2, section 2.5. L'architecture globale de cette nouvelle approche est illustrée sur la figure 5.2. Elle est composée de deux systèmes ; le **Système de Sélection** qui permet de sélectionner un schéma de fragmentation final en optimisant le coût de maintenance, puis d'implémenter physiquement ce schéma sur l'entrepôt et le **Système Décisionnel** qui permet de contrôler le déclenchement du système de sélection lorsqu'une nouvelle requête est exécutée. Le déroulement du processus de sélection est donné comme suit :

- Le système décisionnel analyse la requête nouvellement exécutée afin de déterminer son profil. Si le profil de la requête est "Neutre" ou "Réduction", aucune nouvelle sélection n'est nécessaire. Si le profil de la requête est "Évolution" ou " Mixte", le système de sélection est démarré et effectue les opérations suivantes :
  - ⋆ Générer le schéma de fragmentation actuel de l'$\mathcal{ED}$.
  - ⋆ Analyser la nouvelle requête exécutée Q_i.
  - ⋆ Générer un nouveau schéma de fragmentation SF_1.
  - ⋆ Sélectionner par *AG* un schéma SF' final qui optimise le coût de la charge de requête et le coût de maintenance.
  - ⋆ Implémenter le SF' sur l'entrepôt de données.

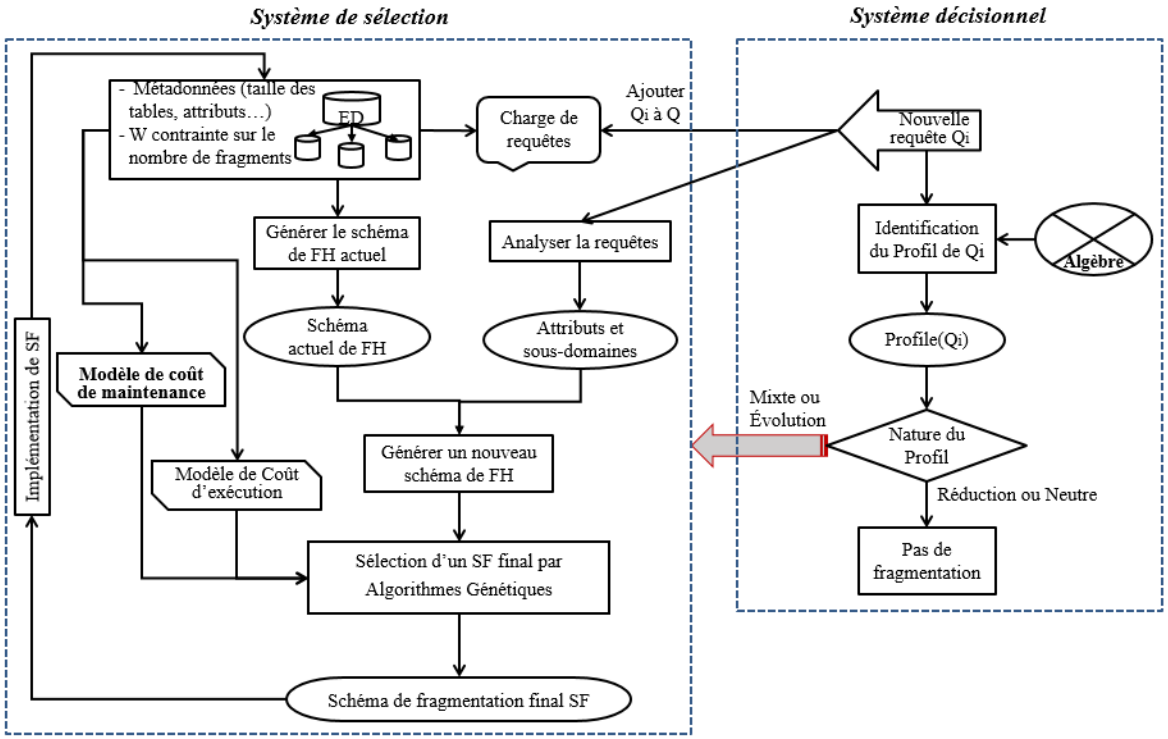


FIGURE 5.2 – Architecture de la sélection incrémentale *CM+ProfilQ*

5.5 Étude expérimentale

Afin d'évaluer nos propositions, nous avons réalisé des tests de comparaison sur un entrepôt réel issu du benchmark APB1 [40] sous le SGBD Oracle 11g avec une machine Intel Core 2 Duo et une mémoire 2Go. Cet entrepôt est composé d'une table de faits *Actvars*(24 786 000 tuples) et quatre tables de dimension *Prodlevel* (9000 tuples), *Custlevel* (900 tuples), *Timelevel* (24 tuples) et *Chanlevel* (9 tuples). Pour nos tests, nous utilisons une charge de 60 requêtes qui génèrent 18 attributs de fragmentation *(Line, Day, Week, Country, Depart, Type, Sort, Class, Group, Family, Division, Year, Month, Quarter, Retailer, City, Gender and All)* qui ont respectivement les cardinalités suivantes : 15, 31, 52, 11, 25, 25, 4, 605, 300, 75, 4, 2, 12, 4, 99, 4, 2, 3.

A notre connaissance, aucun travail ne traitre de la sélection incrémentale multi-objectif. Ainsi, nous avons effectué des comparaisons entres les différentes approches que nous avons développé. Nous effectuons deux parties de tests : (1) des tests théoriques pour évaluer nos approches de sélection incrémentale, et (2) des tests pratiques sous le SGBD Oracle qui vise à évaluer le coût de maintenance engendré par nos approches de sélection incrémentale. Il est à noter que le coût d'exécution de la charge des 60 requêtes sans aucune optimisation est de 42.5 Millions d'E/S.

5.5.1 Évaluation Théorique

Nous avons effectué trois tests théoriques en considérant une contrainte $W = 100$. Ces tests visent à comparer entre deux approches que nous avons proposées.

1. Sélection incrémentale multi-objectif d'un schéma de fragmentation, notée CM

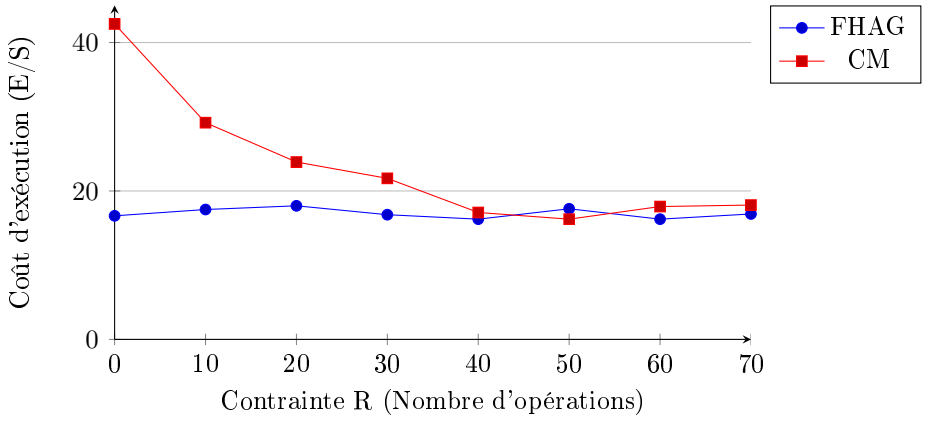


FIGURE 5.3 – Effet de la variation de la contrainte R sur les démarches $FHAG$ et CM

2. Sélection incrémentale d'un schéma de fragmentation par algorithmes génétiques $FHAG$, présentée dans le chapitre 2.

5.5.1.1 Test 1 : Variation de la contrainte R

Le premier test théorique vise à étudier l'influence de la contrainte de maintenance R sur les approches $FHAG$ et CM. Pour ce faire, nous considérons un schéma de fragmentation initial qui servira de base de comparaison et de calcul du coût de maintenance. Puis, nous faisons varier la contrainte R sur le coût de maintenance. Pour simplifier, nous définissons la contrainte R comme le nombre d'opérations de fusions, éclatements et déplacements nécessaires pour implémenter un nouveau schéma de fragmentation. La contrainte R est donc variée de 0 à 70 opérations et à chaque palier de 10 opérations, nous exécutons les deux approches CM et $FHAG$ sur la charge des 60 requêtes. Pour chaque valeur de R et chaque approche, nous relevons le coût d'exécution de la charge de requêtes en présence de la structure d'optimisation sélectionnée. Les résultats sont donnés sur la figure 5.3.

Nous constatons que la variation de la contrainte R n'influe pas sur la démarche $FHAG$ car celle-ci considère uniquement le coût d'exécution de la charge de requêtes comme objectif à optimiser. Cependant, la contrainte R influe sur la qualité des solutions sélectionnées par CM. D'abords, aucune optimisation n'est apportée par CM pour R=0 car aucun schéma de fragmentation n'est sélectionné. Pour des valeurs de R entre 0 et 40 opérations, la démarche $FHAG$ donne une meilleure optimisation du coût d'exécution par rapport à CM. Pour ces valeurs de R de bonnes solutions qui optimisent bien le coût des requêtes sont pénalisées dans la démarche CM, car elles engendrent un coût de maintenance élevé par rapport au seuil R. A partir d'un seuil R=40, les deux approches donnent les mêmes résultats d'optimisation car dans CM de moins en moins de solutions sont pénalisées avec l'augmentation de R.

5.5.1.2 Test 2 : Coût d'exécution

Dans le second test, nous effectuons une évaluation incrémentale avec une contrainte W=100 et R=30. Nous considérons une charge de 45 requêtes déjà exécutées sur l'entrepôt de données. Cette charge est optimisée avec un schéma $\mathcal{FH}$ préalablement sélectionné et implémenté. Ensuite,

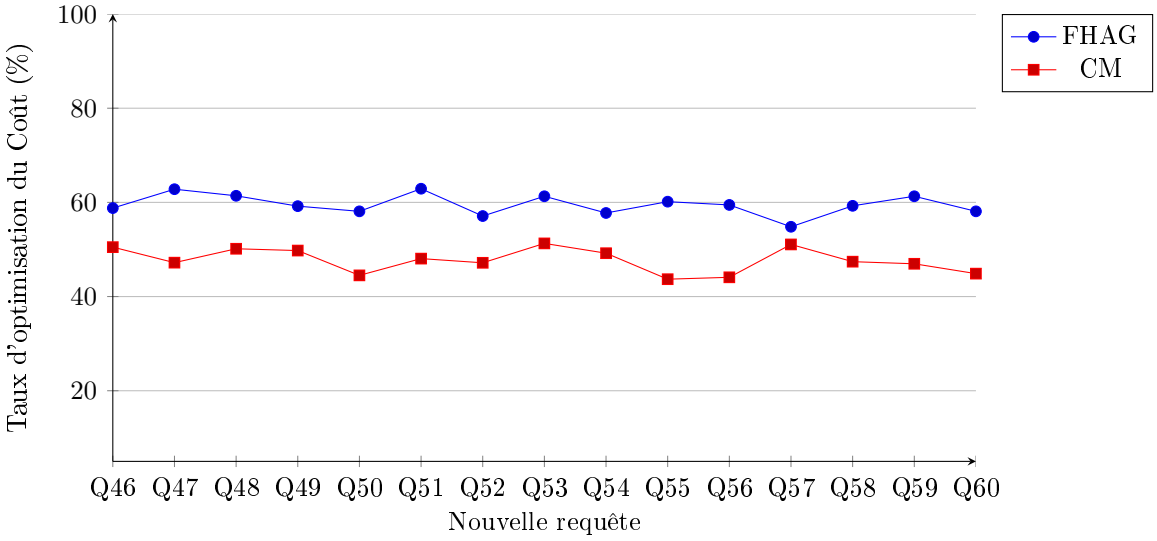


FIGURE 5.4 – Taux de réduction du coût d'exécution : $FHAG$ vs. CM

15 nouvelles requêtes sont exécutées successivement sur l'entrepôt. Lors de l'exécution de chaque nouvelle requête Q_i, nous effectuons la sélection d'un nouveau schéma de fragmentation suivant les deux démarches $FHAG$ et CM. Pour chaque requête et chaque démarche, nous relevons le taux d'optimisation du coût d'exécution de la charge de requêtes en présente du schéma de fragmentation sélectionné (figure 5.4).

En analysant les résultats qu'illustre la figure 5.4, nous constatons que la démarche $FHAG$ apporte une meilleure optimisation du coût que la démarche CM. En effet, $FHAG$ apporte en moyenne une réduction de 59% du coût d'exécution des requêtes et CM donne une amélioration de 50% en moyenne. En effet, dans la démarche de CM, les solutions coûteuses en terme de coût de maintenance sont pénalisées avec une contrainte $R = 30$. Ainsi, dans CM de bonnes solutions, qui permettent d'améliorer les performances des requêtes, peuvent être écartées par le processus de sélection.

5.5.1.3 Test 3 : Coût de maintenance

Dans le troisième test, nous évaluons les deux démarches de sélection selon le coût de maintenance des schémas de fragmentation que chacune des deux démarches sélectionne. Nous avons relevé, durant la réalisation du test 2, le coût de maintenance de chaque schéma de fragmentation final sélectionné par les deux démarches $FHAG$ et CM (contrainte R=30). Le résultat est illustré sur la figure 5.5. Cette figure montre clairement que les solutions finales sélectionnées par la démarche $FHAG$ nécessitent plus d'opérations que les solutions apportées par la démarche CM, une différence allant jusqu'à 30% d'opérations supplémentaires pour $FHAG$. De ce fait, en prenant en compte les deux paramètres importants à savoir : l'optimisation de la charge de requêtes et le coût de maintenance du schéma sélectionné, l'approche CM permet d'apporter les meilleurs résultats d'optimisation multi-objectif.

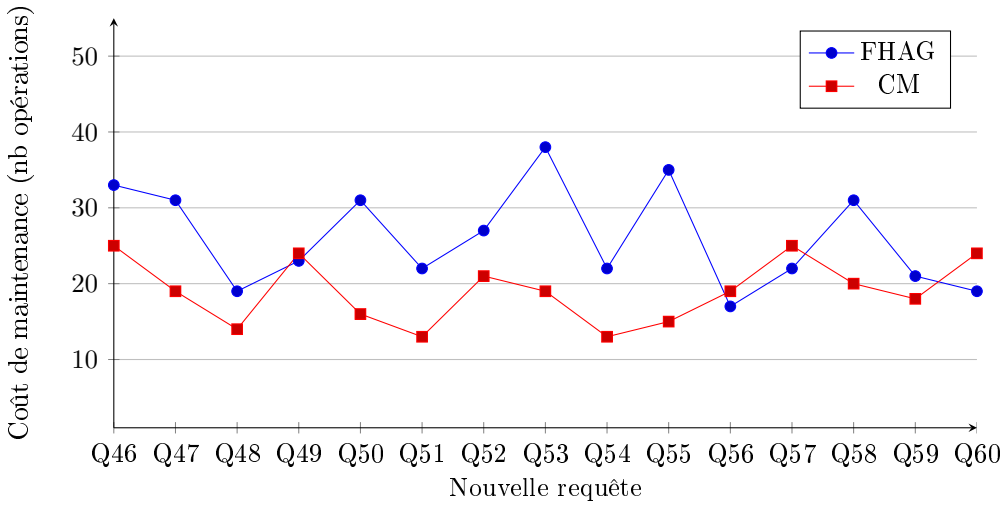


FIGURE 5.5 – Coût de maintenance : $FHAG$ vs. CM

5.5.2 Évaluation pratique sous Oracle 11g

Afin de valider les approches que nous avons implémenté, nous effectuons un test pratique sous le SGBD Oracle qui vise à comparer le temps d'implémentation des schémas de fragmentation sélectionnés par quatre démarches de sélection incrémentale.

- $FHAG$: Sélection incrémentale basée sur les AG.
- CM : Sélection incrémentale basée sur les AG avec optimisation du coût de maintenance.
- $ProfilQ$: Sélection incrémentale basée sur les AG avec Profiling des requêtes.
- CM+$ProfilQ$: Sélection incrémentale basée sur les AG avec Profiling des requêtes et optimisation du coût de maintenance.

Nous considérons l'environnement de sélection incrémentale précédent avec W=100 et R=30. Nous exécutons les quatre approches de sélection et pour chaque approches, nous relevons le temps total d'implémentation effective de tous les schémas de fragmentation sélectionnés après exécution des 15 requêtes sur l'entrepôt de données. Nous pouvons considérer ce temps comme le temps de déroulement du test pour chaque sélection. Les résultats sont donnés par la figure 5.6.

Nous constatons que la démarche $FHAG$ donne le temps d'implémentation le plus lent par rapport aux autres approches (5h21 heures) vu qu'elle ne considère aucune optimisation du coût de maintenance d'un schéma de fragmentation. De plus, cette approche se déclenche après l'exécution de chaque nouvelle requête. En comparant les deux approches CM et $ProfilQ$, nous remarquons que l'approche $ProfliQ$ engendre un temps globale d'implémentation réduit (2h48 heures) par rapport à CM (3h54 heures). Contrairement à la sélection CM qui altère le schéma de fragmentation de l'entrepôt après l'exécution de chaque nouvelle requête parmi les 15 requêtes, pour $ProfliQ$ seules 6 requêtes nécessitent une nouvelle sélection incrémentale car elles ont un profil Mixte et Évolution. Les 9 autres requêtes ont un profil Neutre et Réduction qui ne nécessite aucune nouvelle sélection. Enfin, nous remarquons que la démarche CM+$ProfilQ$ apporte les meilleurs résultats à savoir un temps d'implémentation de 2h06. Non seulement, cette sélection se déclenche uniquement pour 6 requêtes sur 15 mais en plus, elle optimise le coût de maintenance de chaque nouveau schéma de fragmentation sélectionné ce qui réduit considérablement le temps d'implémentation total.

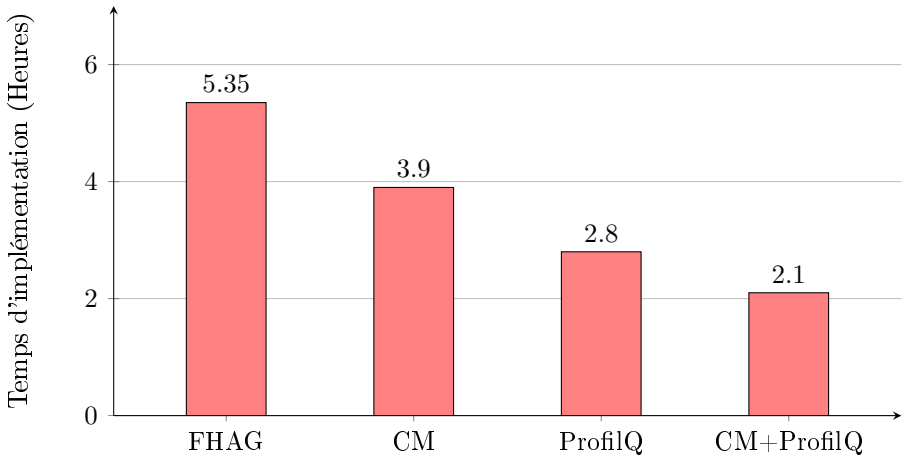


FIGURE 5.6 – Temps d'implémentation des SF : $FHAG$, CM, $ProfilQ$, $CM + ProfilQ$

5.6 Conclusion

Dans ce chapitre nous avons proposé une nouvelle formulation du problème de sélection incrémentale d'un schéma de fragmentation en un problème multi-objectif qui vise à optimiser deux objectifs à savoir le coût d'exécution des requêtes et le coût de maintenance du nouveau schéma de fragmentation sélectionné. En effet, lorsque la charge de requêtes évolue, il faut adapter le schéma de fragmentation actuel de l'entrepôt ce qui nécessite d'effectuer plusieurs opérations de fusions, éclatements et déplacements des différents fragments. Sélectionner un nouveau schéma de fragmentation qui optimise uniquement les performances de la charge évolutive n'est pas suffisant ; il faut également penser à optimiser le coût de maintenance de l'entrepôt de données pour l'adapter à l'évolution de charge.

Afin de résoudre le PMO sur la sélection incrémentale d'un schéma de fragmentation, nous avons utilisé la méthode du compromis qui vise à transformer le PMO en un problème mono-objectif. Cela nous a conduit à proposer une nouvelle approche de sélection incrémentale d'un schéma de fragmentation avec optimisation du coût de maintenance. Nous avons proposé un modèle de coût de maintenance et pour ce faire, nous nous sommes basés sur l'algèbre de fragmentation afin de déterminer toutes les opérations de passage d'un schéma de fragmentation vers un nouveau schéma. Puis chaque opération est évaluée afin de déterminer le coût de maintenance total. Ensuite, nous avons combiné cette proposition avec nos travaux précédents où nous avons intégré le principe de Profiling des requêtes afin de mieux réduire le coût de maintenance de l'entrepôt de données.

Nous avons conduit une étude expérimentale sous le SGBD Oracle 11g afin de tester toutes les approches proposées. D'abords, nous avons montré une fois de plus l'efficacité de notre algèbre de fragmentation qui est utilisée dans détermination du modèle de coût de maintenance et le Profiling des requêtes. De plus, nous avons conclu que combiner le principe d'optimisation du coût de maintenance et le principe de Profiling des requêtes permet à la fois de répondre aux exigences des décideurs quant à la performance des requêtes exécutées mais aussi permet au concepteur de l'entrepôt de données d'optimiser le coût de maintenance de l'entrepôt lorsque celui-ci évolue.

Chapitre 6

AdminInc : Outil pour la conception physique dynamique

Sommaire

6.1 Introduction

L'administrateur des entrepôts de données a la tâche difficile de choisir un scénario d'optimisation des requêtes décisionnelles lors de la conception physique de l'entrepôt de données. Il doit choisir entre plusieurs techniques d'optimisation comme les index, vues, fragmentation, gestion du buffer, ordonnancement des requêtes, etc. et entre plusieurs algorithmes de sélection afin de gérer une taille gigantesque de données (plusieurs Go ou To). Les auteurs dans [26] ont montré que parmi les responsabilités de l'administrateur, comme le codage, le recouvrement, la surveillance du système, la mise à niveau du système, etc, la tâche de conception physique consomme 17% du temps d'administration. Par conséquent, plusieurs outils existent pour assister l'administrateur dans l'étape de conception physique comme Oracle Tuning Advisor [43] pour l'administration et le tuning des entrepôts de données, DB2 Design Advisor [95, 100, 60], Microsoft Database Tuning Advisor[1] et l'outil ParAdmin proposé par [11] dans le domaine de la recherche.

La tâche d'administration des $\mathcal{ED}$ est d'avantage plus complexe dans le contexte incrémentale car il faut maintenir le schéma d'optimisation implémenté sur l'entrepôt de manière continuelle afin de répondre aux nouvelles exigences des décideurs. Par conséquent, l'attention de l'administrateur est continuellement sollicité pour cette tâche. La question qui se pose est : quand faut-il mettre à jours les structures d'optimisation implémentées ? Dans le cadre de la sélection incrémentale de la fragmentation horizontale par exemple, les auteurs dans [45] montent qu'il est difficile de définir le moment où il faut refragmenter un entrepôt de données. En plus de cette difficulté, et lors de notre étude de la sélection incrémentale, nous constatons que pour mettre à jour un schéma d'optimisation de l'entrepôt, il faut tenir compte de plusieurs facteurs comme la disponibilité des ressources, la nécessité d'une mise à jour, le coût de maintenance vs. le gain de performances, etc.

Ainsi, nous proposons un outil pour la conception physique des $\mathcal{ED}$ dans le cadre incrémentale, appelé AdminInc. AdminInc implémente les approches que nous avons proposées. L'administrateur effectue le paramétrage de l'outil afin de choisir la démarche de sélection souhaitée ; IJB ou FH, sélection isolée ou jointe, sélection mono ou multi-objectif, optimisation du coût de maintenance par Profiling de requêtes, par modèle de coût de maintenance ou les deux. AdminInc analyse continuellement le journal du SGBD afin de détecter des changements au niveau des requêtes. Lors de l'évolution de charge, AdminInc effectue la mise à jour du schéma d'optimisation selon le paramétrage effectué. L'outil est développé sous l'environnement Windows 7 avec le langage JAVA sous l'EDI Eclipse et utilise l'API JGAP pour implémenter l'algorithme génétique.

Ce chapitre est organisé comme suit. Dans la section 2 nous présentons la conception et réalisation de notre outil AdminInc. La section 3 conclue le chapitre.

6.2 Conception et réalisation de AdminInc

Nous présentons dans cette section la description de notre outil d'assistance à l'administration des $\mathcal{ED}$ dans le cadre incrémentale AdminInc. Cet outil permet de maintenir continuellement le schéma de fragmentation et/ou indexation d'un entrepôt et de le mettre à jour lorsqu'une nouvelle requête est exécutée. Dans un premier lieu, nous présentons l'analyse fonctionnelle de l'outil qui comporte la description des techniques d'optimisation, les modes de sélection et les algorithmes de sélection. Dans un second lieu, nous exposons la conception de AdminInc et ses fonctionnalités. Finalement, nous présentons un exemple d'utilisation de l'outil connecté à Oracle 11g avec le benchmarck APB-1.

6.2.1 Analyse fonctionnelle

Notre outil doit répondre à plusieurs exigences et permettre à l'administrateur d'effectuer le bon paramétrage afin de réaliser le maintien des structures d'optimisation implémentées sur un entrepôt de données. L'administrateur doit paramétrer l'outil selon les aspects présentés dans ce qui suit.

6.2.1.1 Les techniques d'optimisations

L'outil permet d'implémenter deux techniques d'optimisation. La fragmentation horizontale et les index de jointures binaires. Le scénario de fragmentation horizontale dans les entrepôts données consiste à effectuer une $\mathcal{FH}$ primaire des tables dimensions suivie par une $\mathcal{FH}$ dérivée de la table de faits [14]. Ce scénario de fragmentation engendre un nombre de fragments important de la table de fait qui est égale au produits des nombres de fragments respectifs des tables de dimension. L'outil permet de fixer le nombre de fragments faits à générer au maximum et d'effectuer la sélection incrémentale d'un schéma de $\mathcal{FH}$ comme montré dans le chapitre 2. Concernant les index de jointures binaires, l'outil permet d'implémenter les index simples mono-attributs et les index multiples multi-attributs. Il permet de fixer l'espace de stockage alloué aux index. La sélection incrémentale des index est réalisée comme présenté dans le chapitre 1.

6.2.1.2 Le mode de sélection

AdminInc permet de choisir entre une sélection isolée ou une sélection jointe. Pour la sélection isolée, l'outil permet d'optimiser l'entrepôt avec une $\mathcal{FH}$ ou avec les index de jointures binaire et de fixer pour chaque technique les contraintes d'optimisation. Concernant la sélection jointe, il est possible de sélectionner deux techniques d'optimisation ; fragmentation et index simples, fragmentation et index multiple, index simples et index multiples. La sélection jointe de la fragmentation et de l'indexation est présentée dans le chapitre 4. Quant à la sélection jointe des index simples et des index multiples, elle est présentée dans le chapitre 3.

6.2.1.3 Les algorithmes de sélection

Pour la sélection des structures d'optimisation finales, nous avons employé un algorithme génétique implémenté à travers le framework JGAP développé en JAVA.

6.2.1.4 Les objectifs à optimiser

Dans le contexte d'optimisation incrémentale, un coût de maintenance de l'entrepôt est engendré représentant le coût d'exploitation des ressources pour maintenir et mettre à jour le schéma d'optimisation actuellement implémenté selon l'évolution de l'entrepôt. AdminInc donne la possibilité d'inclure le coût de maintenance dans le processus de d'optimisation afin de réaliser une optimisation multi-objectif des requêtes décisionnelles. L'administrateur peut fixer le coût de maintenance maximum qu'un nouveau schéma d'optimisation peut engendrer.

6.2.1.5 Le contrôle de la sélection incrémentale

La principale difficulté dans le cadre de sélection incrémentale est de décider quand es ce qu'il faut mettre à jour le schéma d'optimisation de l'entrepôt. Par conséquent, notre outil capture de manière continuelle les changements effectués sur la charge de requêtes afin d'exécuter une stratégie

d'optimisation. Dans le SGBD Oracle, les requêtes exécutées sur l'entrepôt sont stockées dans le journal des transactions, plus exactement dans une table appelée v$sql. La requête SQL suivante permet de récupérer et d'ordonner les requêtes de la plus récente à la plus ancienne requête exécutée.

```
SELECT *
FROM v$sql
WHERE sql_text LIKE '%Actvars%'
AND parsing_schema_name='DW'
ORDER BY first_load_time DESC;
```

Où Actvars est le nom de la table de faits et DW le nom du schéma de l'entrepôt de données. Lorsqu'un changement au niveau du journal des transactions est détecté, l'outil exécute une sélection incrémentale selon le paramétrage effectué (technique, mode de sélection, contraintes d'optimisation, objectifs). Cependant, certains changements au niveau des requêtes peuvent ne pas nécessiter une mise à jour du schéma d'optimisation. Ainsi, AdminInc permet d'inclure le Profiling des requêtes dans le processus de sélection. Il permet de visualiser la nouvelle requête exécutée, son profil, les opérations algébriques qu'elle engendre et décide s'il est impératif ou non de lancer un nouveau processus de sélection incrémentale.

6.2.2 Conception et réalisation

Nous présentons sur la figure 6.1 l'architecture globale de notre outil AdminInc. L'outil permet à l'administrateur de visualiser les différentes tables et attributs de l'entrepôt et les requêtes exécutées. Il permet d'extraire le schéma d'optimisation actuel ; schéma de fragmentation des tables et/ou schéma d'indexation. L'administrateur peut aussi paramétrer l'outil pour spécifier la ou les techniques à sélectionner, le type de sélection isolée ou jointe, les contraintes d'optimisation à savoir S (espace de stockage des index), W (nombre de fragments faits maximum) et R (coût maximum de maintenance), le type d'optimisation mono ou multi-objectif et l'utilisation du Profiling des requêtes.

L'outil est composé de quatre ongles, Visualiser, Système Décisionnel, Système de Sélection et Optimisation inrémentale. Ils sont détaillés dans ce qui suit.

6.2.2.1 Visualisation

Comme le montre la figure 6.2, AdminInc permet de se connecter à un schéma d'entrepôt de données, en l'occurrence le schéma "dw". Il permet d'extraire les tables, leurs tailles et attributs respectifs et permet d'identifier la table de faits, dans cet exemple "Actvars". La charge de requêtes actuellement optimisée peut être extraite et affichée. L'outil permet également d'extraire le schéma d'optimisation de l'entrepôt (figure 6.3) ; schéma de fragmentation et schéma d'indexation s'il y a lieu. L'outil montre pour un schéma d'indexation le type d'index implémenté (simple ou multiple ou les deux), les attributs indexables, les index, la taille de chacun en Mo et la taille de l'espace de stockage occupé par tous les index. Pour la fragmentation, on peut visualiser le schéma de fragmentation ; les attributs et pour chaque attribut la répartition de son domaine en ensembles de sous-domaines, ainsi que le nombre de fragments de la table de faits. A travers le menu et le bouton Classification, il est possible de voir le résultat de classification des attributs pour la sélection jointe de $\mathcal{FH}$ et $\mathcal{IJB}$ et la classification des requêtes pour la sélection jointe des index simples et multiples.

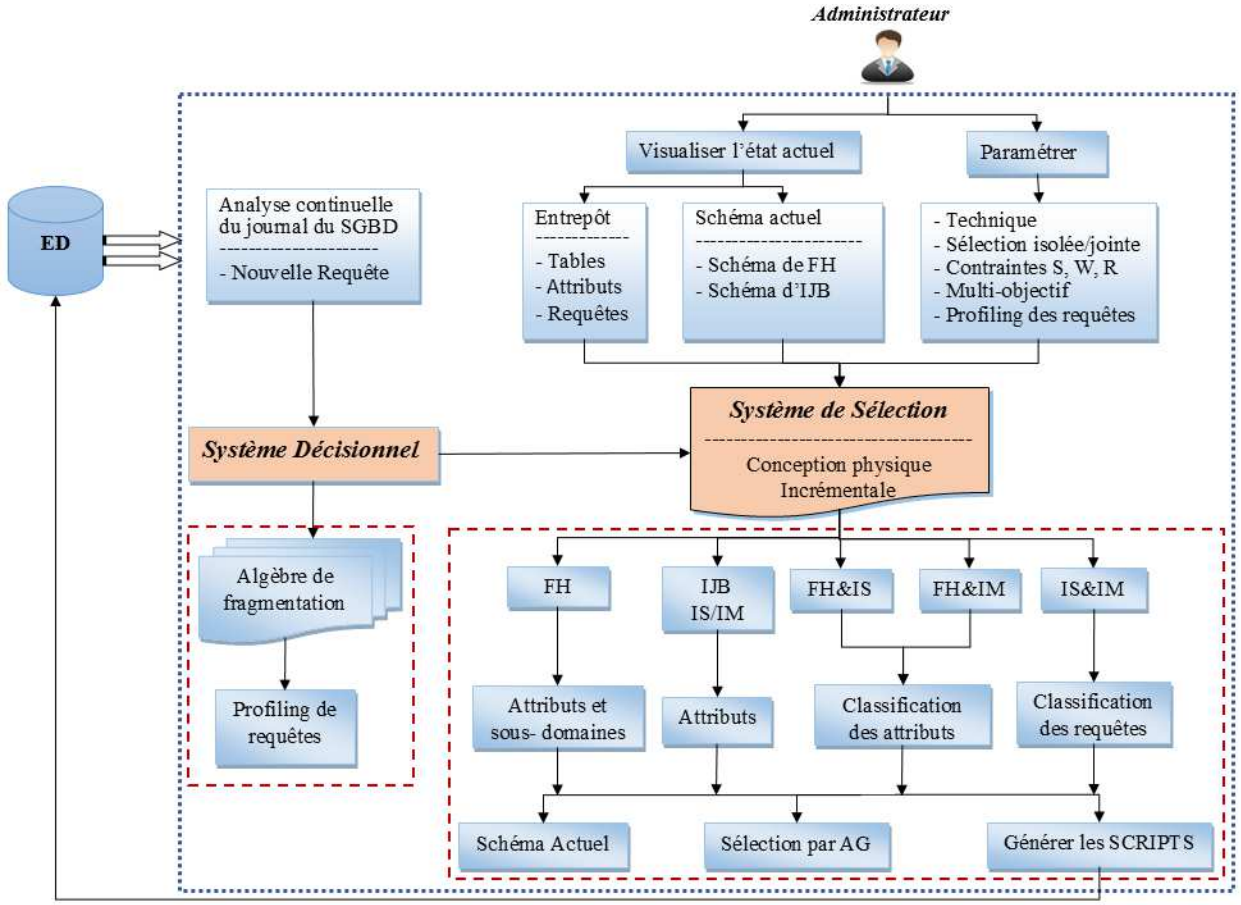


FIGURE 6.1 – Architecture globale de l'outil AdminInc

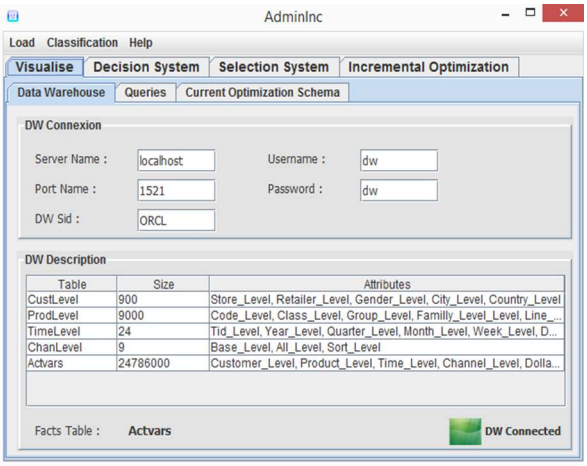


FIGURE 6.2 – Connexion à l'entrepôt de données et visualisation des tables

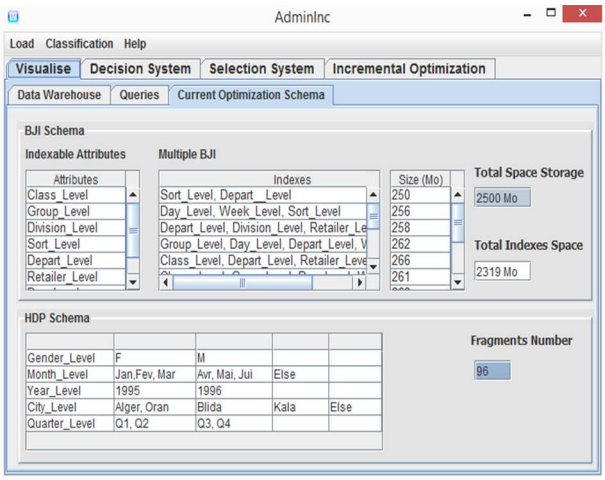


FIGURE 6.3 – Schéma d'optimisation actuel de l'entrepôt

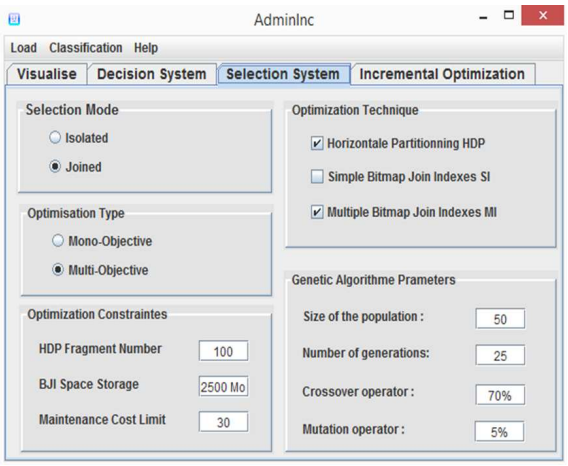


FIGURE 6.4 – Paramétrage du Système de Sélection

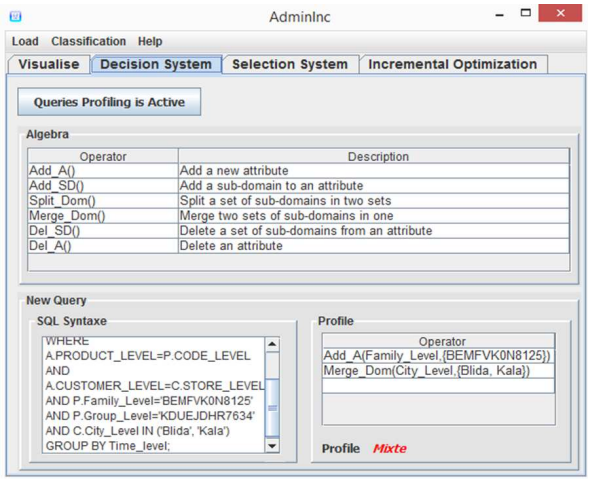


FIGURE 6.5 – Activation du Système Décisionnel, Algèbre et Profil de requête

6.2.2.2 Système de Sélection

Dans l'onglet Selection System, l'administrateur peut paramétrer l'outil afin de choisir une stratégie d'optimisation comme le montre la figure 6.4. Il choisit le mode de sélection isolée ou jointe, la technique d'optimisation à employer à savoir la fragmentation, les index simples et/ou les index multiples et le type d'optimisation mono ou multi-objectif. Selon le choix fait jusque-là, l'administrateur peut fixer les contraintes d'optimisation qui représentent le nombre maximum de fragments faits pour la fragmentation, l'espace de stockage des index et le coût de maintenance maximum dans le cadre de la sélection multi-objectif. Enfin, l'outil implémente un algorithme génétique pour chaque processus de sélection dont il est possible de fixer les paramètres à savoir taille de la population, nombre de génération, taux de croisement et taux de mutation.

6.2.2.3 Système Décisionnel

L'onglet Decision System permet d'activer le système décisionnel qui contrôle le lancement du système de sélection (figure 6.5). AdminInc expose l'algèbre de fragmentation à travers la description de ses opérateurs. La dernière requête exécutée est affichée ainsi que son profil et les opérateurs algébriques qu'elle a engendré sur le schéma de fragmentation actuel de l'entrepôt de données.

6.2.2.4 Optimisation incrémentale

L'onglet Incremental Optimization présente les résultats d'optimisation incrémentale. AdminInc est exécuté continuellement en arrière-plan et analyse de façon périodique le journal de transaction du SGBD. Lorsque une nouvelle requête est exécutée, par exemple à travers l'utilitaire SQLPlus* d'Oracle (figure 6.6), l'outil capture ce changement et selon la configuration du système de sélection, effectue la mise à jour du schéma d'optimisation. Selon figure 6.4, le paramétrage du système de sélection donne lieu à une sélection incrémentale jointe de la fragmentation et des index multiples où l'optimisation est multi-objectif. L'exécution de la sélection incrémentale donne lieu à un nouveau schéma d'optimisation illustré sur la figure 6.7. La nouvelle requête exécutée est donnée comme suit :

```
SELECT Time_level,count(*)
```

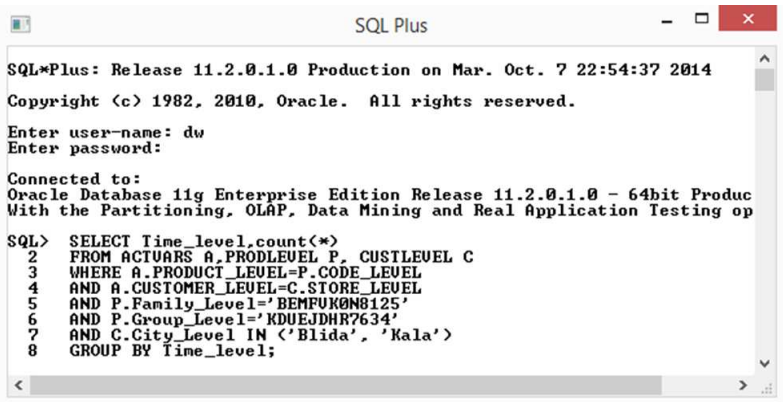


FIGURE 6.6 – Exécution d'une nouvelle requête sous Oracle 11g (SQLPlus*)

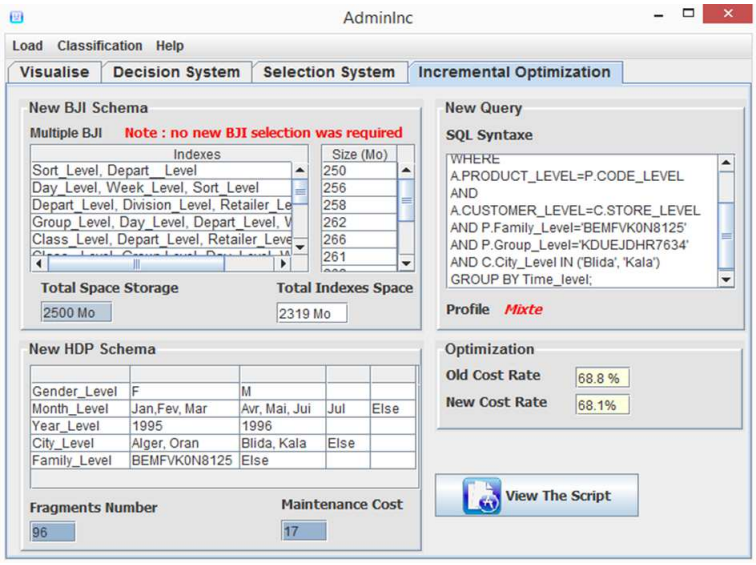


FIGURE 6.7 – Résultat de l'optimisation incrémentale après exécution de la nouvelle requête

```
FROM ACTVARS A,PRODLEVEL P,CUSTLEVEL C
WHERE A.PRODUCT_LEVEL=P.CODE_LEVEL
AND A.CUSTOMER_LEVEL=C.STORE_LEVEL
AND P.Family_Level='BEMFVK0N8125'
AND P.Group_Level='KDUEJDHR7634'
AND C.City_Level IN ('Blida', 'Kala')
GROUP BY Time_level;
```

Le profil de la requête est Mixte car elle engendre les opérations algébriques Add_A(Family_Level,BEMFVK0N8125) vu que l'attributs Family_Level ne figure pas sur le schéma actuel de l'entrepôt et Merge_Dom(City_Level,Blida, Kala) car les deux valeurs sont présentes dans deux ensembles différents sur le même schéma (voir figure 6.3). La sélection jointe effectue la classification des attributs de la requête comme suit : Family_Level et City_Level pour la fragmentation et Group_Level pour l'indexation. Seule une nouvelle sélection incrémentale d'un schéma de fragmentation est requise car aucun index multiple ne peut être défini sur un seul attribut (voir figure

6.7).

6.3 Conclusion

Le contexte de sélection incrémentale des entrepôts de données a donné lieu à deux nouvelles problématiques à savoir le déclenchement du processus de sélection et l'analyse de la nécessité d'une nouvelle sélection ainsi que l'optimisation du coût de maintenance de l'entrepôt. Par conséquent, nous avons développé un outil d'assistance à l'administration des entrepôts de données dans le cadre incrémentale. L'administrateur paramètre une fois l'outil afin de choisir un scénario d'optimisation. L'outil s'exécute en arrière-plan de manière continuelle et permet de capturer les changements survenus sur l'entrepôt de données. Quand une nouvelle requête s'exécute, le système décisionnel décide s'il faut ou non altérer le schéma d'optimisation actuel. Si oui, un nouveau schéma d'optimisation est sélectionné et est implémenté sur l'entrepôt. AdminInc fonctionne de manière indépendante et réduit donc le temps alloué par l'administrateur à la tâche de conception physique de l'entrepôt de données.

Troisième partie

Conclusion et perspectives

Conclusion et perspectives

Conclusion

Nous présentons dans ce chapitre un bilan de ce travail et propositions que nous avons élaboré ainsi qu'un ensemble de perspectives. Concernant les principales propositions de cette thèse, elles découlent de trois importantes constatations.

La première constatation est que la plupart des travaux de sélection des techniques d'optimisation traitent d'une sélection statique. L'hypothèse forte considérée est que les requêtes décisionnelles qu'il faut optimiser sont préalablement connues. Les structures d'optimisation sont sélectionnées durant la phase de conception physique de l'entrepôt de données et ne changent plus. Cependant, l'entrepôt de données est soumis à une continuelle évolution de charge (exécution de nouvelles requêtes, changement de leurs fréquences d'exécution) et évolution de données (insertion continuelles de nouveaux tuples dans la tables de faits) ce qui nécessite une mise à jour continuelle des structures implémentées, devenues obsolètes, pour les adapter à l'évolution de l'entrepôt.

La seconde constatation que nous avons faite et que la plupart des approches de sélection sont proposées dans un cadre isolé où une seule technique d'optimisation est considérée. Devant la complexité des requêtes qui augmente jour après jour, la sélection d'une seule technique d'optimisation n'est plus suffisante. Il faut combiner plusieurs techniques d'optimisation afin de couvrir l'optimisation d'un maximum de requêtes et de respecter au mieux les contraintes d'optimisation, dans une sélection dite sélection jointe. Cette dernière vise à exploiter les similarités entre les techniques d'optimisation pour sélectionner un schéma d'optimisation plus efficace.

La troisième constatation est que tous les travaux qui traitent de l'indexation et/ou de la fragmentation horizontale formalisent le problème d'optimisation en un problème mono objectif ou seules les performances des requêtes sont optimisées. Cependant, dans un contexte d'entrepôt évolutif, la mise à jour des structures d'optimisation engendre un coût de maintenance de l'entrepôt de données qu'il faut prendre en compte dans le processus d'optimisation.

Pour répondre à ces trois problématiques, nous avons développé les propositions suivantes :

Sélection incrémentale isolée des index de jointure binaires

Dans l'état de l'art sur les index de jointure binaires, nous avons présenté une analyse et classification des principaux travaux de recherches qui traitent de la sélection de ce type d'index. Nous avons conclu que ces travaux sont réalisés en deux phases importantes : une phase de sélection de la configuration initiale, qui inclut une étape d'élagage de l'espace de recherche afin de réduire la complexité du problème, et une phase de sélection de la configuration finale d'index. Nous avons conclus que peu de travaux emploient les requêtes pour l'élagage de l'espace de recherche, peu d'algorithme sont utilisés pour la sélection de la configuration finale et ces travaux se situent, pour la plupart,

dans le cadre de sélection statique d'index. De ce fait, nous avons proposé une nouvelle approche de sélection incrémentale d'index de jointure binaire basée sur les algorithmes génétiques et l'élagage par requêtes. Afin de réaliser la sélection d'index par algorithmes génétiques, nous avons commencé par définir la structure du chromosome qui permet de coder l'espace de recherche des index. Nous avons défini une structure de chromosome pour les index simples (index de jointure binaires définis sur un seul attribut) et deux structures pour les index multiples (index de jointure binaires définis sur deux ou plusieurs attributs) avec élagage par requêtes. Nous avons défini le modèle de coût mathématique qui permet de formuler la fonction objectif puis avons proposé un modèle de coût pour les index compressés. La fonction objectif permet de guider l'algorithme génétique dans la sélection des index. Ensuite, nous avons détaillé l'algorithme génétique qui permet, indépendamment de la structure du chromosome, de sélectionner des index simples ou multiples. Nous avons terminé par la réalisation de tests réels comparatifs avec les approches proposées dans la littérature, sous le SGBD Oracle11g

Sélection incrémentale isolée d'un schéma de fragmentation horizontale

Nous avons proposé une sélection incrémentale d'un schéma de fragmentation horizontale par algorithmes génétiques qui permet de prendre en considération l'évolution de l'entrepôt. Nous avons commencé par proposer une structure de données basée sur un codage flexible de n'importe quel schéma de fragmentation. Ensuite, nous avons proposé une algèbre de fragmentation qui permet de modéliser toutes les opérations possibles pouvant être effectuées sur la structure de données. En d'autre terme, l'algèbre permet de déterminer les opérations nécessaires pour mettre à jour un schéma de fragmentation. Grâce à notre algèbre, nous avons défini le Profiling des requêtes. Selon le profil d'une nouvelle requête exécutée, on détermine si une refragmentation de l'entrepôt est nécessaire. Après cela, nous avons proposé une architecture de la sélection incrémentale d'un schéma de fragmentation composée de deux systèmes, un système de sélection et un système décisionnel qui emploi l'algèbre de fragmentation pour contrôler le déclenchement du système de sélection. Enfin, nous avons mené une étude expérimentale dans le cadre du SGBD Oracle 11g pour montrer l'efficacité de nos propositions.

Sélection incrémentale jointe des index simples et des index multiples

Nous avons étudié le problème de sélection jointe des index de jointure binaires simples et des index de jointure binaires multiples. Nous avons vu que les deux types d'index combinés permettent de couvrir l'optimisation d'un plus large nombre de requêtes et chaque type d'index présente un avantage par rapport à l'autre. Nous avons montré que l'espace de recherche de la sélection jointe est très complexe. De plus, la probabilité de sélectionner un index simple par rapport aux index multiples est réduite. Par conséquent, nous avons proposé de séparer les deux espaces de recherche ce qui induit à la concurrence des deux sélections d'index aux mêmes ressources : charge de requêtes et espace de stockage. Cette concurrence nous a conduit à définir deux nouveaux problèmes d'optimisation combinatoire : le problème de classification des requêtes en deux classes et le problème de partage de l'espace de stockage en deux espaces. Afin de résoudre le problème de classification des requêtes, nous avons adopté l'algorithme de classification k-means. Pour le partage de l'espace de stockage, nous avons proposé une démarche guidée par le résultat de classification des requêtes. Ensuite, nous avons développé une approche de sélection incrémentale jointe et y avons intégré un système de récupération d'espace de stockage qui permet de garder un équilibre d'espace entre les index simples et multiples. Nous avons terminé par une étude expérimentale constituée de tests théoriques et tests pratiques

sous le SGBD Oracle 11g.

Sélection incrémentale jointe de l'indexation et de la fragmentation

A la base de nos précédents travaux sur la sélection statique jointe des index de jointure binaires et de la fragmentation horizontale, nous avons proposé une approche de sélection incrémentale jointe de ces deux techniques. Nous avons présenté le principe de la sélection statique jointe ; extraction des attributs de sélection à partir des requêtes, classification par k-means des attributs en deux classes, une classe pour les index et une autre pour la fragmentation, lancement de l'algorithme de sélection de chaque technique sur sa classe, implémentation du schéma d'optimisation final. Ensuite, nous avons exposé notre démarche de sélection incrémentale jointe. Lorsqu'une nouvelle requête est exécutée, ses attributs sont extraits et classifiés. Suivant le résultat de classification, nous avons élaboré trois scénarios d'optimisation. Si les attributs sont tous classifiés pour l'indexation, alors une nouvelle sélection incrémentale des index est effectuée. S'ils sont classifiés pour la fragmentation, on effectue une nouvelle sélection incrémentale d'un schéma de fragmentation avec reconstruction des index. Si les attributs sont classifiés pour les deux techniques alors on exécute les deux sélections incrémentale. Cette approche permet de mettre à jour le schéma d'optimisation selon le besoin apporté par la requête. Nous avons mené des tests sous Oracle et avons conclus que la sélection jointe apporte une meilleure optimisation que la sélection isolée.

Sélection incrémentale multi-objectif

Nous avons proposé une approche de sélection incrémentale multi-objectif d'un schéma de fragmentation où nous considérons deux objectifs à optimiser ; le coût d'exécution des requêtes et le coût de maintenance de l'entrepôt de données. Nous avons commencé par présenter un état de l'art sur l'optimisation multi-objectif en général et l'optimisation multi-objectif dans le contexte d'entrepôt de données en particulier. Ensuite, nous avons exposé notre approche développée. Pour ce faire, nous avons formulé le problème de sélection en un problème multi-objectif PMO et avons proposé une démarche de résolution du PMO. Nous avons proposé un modèle de coût de maintenance en employant l'algèbre de fragmentation. Par la suite, nous avons combiné cette proposition avec nos travaux précédents comme le principe de Profiling des requêtes. Nous avons conduit une étude expérimentale sous le SGBD Oracle 11g afin de tester toutes les approches proposées. D'abords, nous avons montré l'efficacité de notre algèbre de fragmentation. De plus, nous avons conclu que combiner le principe d'optimisation multi-objectif et le principe de Profiling des requêtes permet à la fois de répondre aux exigences des décideurs quant à la performance des requêtes exécutées mais aussi permet à l'administrateur de faire face à l'évolution de l'entrepôt de données.

Outil pour la conception physique dynamique

Nous avons proposé un outil pour la conception physique des $\mathcal{ED}$ dans le cadre incrémentale, appelé AdminInc. Le principal apport de notre outil est qu'il réduit considérablement l'intervention de l'administrateur dans la conception physique dynamique des entrepôts de données. En effet, AdminInc s'exécute en arrière-plan de manière continuelle et indépendante et permet de capturer les changements survenus sur l'entrepôt de données. Quand une nouvelle requête s'exécute, le système décisionnel décide s'il faut ou non altérer le schéma d'optimisation actuel. Si oui, un nouveau schéma d'optimisation est sélectionné et est implémenté sur l'entrepôt. Notre outil permet de décider quand faut-il mettre à jour le schéma d'optimisation de l'entrepôt, permet d'optimiser son coût de

maintenance et allège le temps alloué par l'administrateur pour administrer et tuner l'entrepôt de données.

Perspectives

Les travaux présentés dans ce manuscrit laissent envisager de nombreuses perspectives. Nous présentons celles qui nous paraissent être les plus intéressantes.

Nos travaux de sélection incrémentales ce sont centrées sur deux techniques d'optimisation ; les index de jointures binaires et la fragmentation horizontale. Il serait intéressant d'étudier la sélection incrémentale d'autres techniques d'optimisation comme les vues matérialisées. Les vues matérialisées permettent d'accélérer les traitements et matérialiser des requêtes ou sous-résultats des requêtes. Ces vues sont continuellement mises à jours lorsque les données des tables changent. Il faut également prévoir la mise à jours de ces vues lorsque la charge de requêtes évolue.

Nous avons étudié la sélection incrémentale dans le contexte d'entrepôt de données centralisé modélisés en étoile. Il serait judicieux d'adapter les approches proposées dans le cadre d'entrepôt de données parallèles ou distribués particulièrement dans le cadre de la fragmentation horizontale. Il faut développer une approche de réallocation des fragments sur les différents nœux lorsque le schéma de fragmentation de l'entrepôt évolue.

Nous avons proposé une modélisation multi-objectif du problème de sélection d'un schéma de fragmentation. On peut généraliser cette formalisation pour n'importe quelle structure d'optimisation comme par exemple le problème de sélection des vues matérialisées où il faut optimiser deux objectifs ; le coût de la charge de requêtes et le coût de maintenance des vues dans un cadre incrémentale.

Nous avons proposé une structure de donnée et une algèbre pour la fragmentation horizontale. Il faut étudier la possibilité de proposer une structure d'optimisation et une algèbre pour la sélection d'autres techniques d'optimisation.

Quatrième partie

Annexes

Annexe A : Requêtes

Requête 1
SELECT Division,Count(*)
FROM Actvars A, Prodlevel P
WHERE A.Product=P.Code AND P.Group='S7jweujryiwn'
GROUP BY Division;
Requête 2
SELECT Count(*)
FROM Actvars A, Prodlevel P,Timelevel T, Custlevel C
WHERE A.Product=P.Code AND A.Customer=C.Store
AND A.Time=T.Tid AND P.Group='S7jweujryiwn'
AND P.Family='Cknh6pcpi21c';
Requête 3
SELECT Count(*)
FROM Actvars A,Prodlevel P, Custlevel C
WHERE A.Product=P.Code AND A.Customer=C.Store
AND P.Class='Df2i75i10ks8' AND C.Depart='B'
AND C.City='Alger';
Requête 4
SELECT Time,Avg(Unitssold)
FROM Actvars A,Timelevel T
WHERE A.Time=T.Tid AND T.Week='3'
AND (T.Quarter='Q1' Or T.Quarter='Q2')
GROUP BY Time;
Requête 5
SELECT Division,Count(*)
FROM Actvars A,Prodlevel P
WHERE A.Product=P.Code AND P.Group='S7jweujryiwn'
AND P.Type='B'
GROUP BY Division;
Requête 6
SELECT Max(Unitssold)
FROM Actvars A,Timelevel T
WHERE A.Time=T.Tid AND T.Month=12 AND T.Day='7';
Requête 7
SELECT Base, Count(*)
FROM Actvars A,Chanlevel H,Timelevel T,Prodlevel P
WHERE A.Channel=Ch.Base AND A.Product=P.Code
AND A.Time=T.Tid AND H.Sort='E' AND T.Quarter='Q2'
AND P.Type='B' AND H.All='Efghijklmnop'
GROUP BY Base;
Requête 8
SELECT Avg(Unitssold)
FROM Actvars A,Timelevel T
WHERE A.Time=T.Tid AND T.Week='6';
Requête 9
SELECT Product,Count(*)
FROM Actvars A,Timelevel T
WHERE A.Time=T.Tid AND T.Day=7 AND T.Month='3'
AND T.Week='1' AND T.Year='1996'
GROUP BY Product;
Requête 10
SELECT Max(Dollarcost)
FROM Actvars A, Prodlevel P
WHEREA.Product=P.Code AND P.Family='Pi9do0ygkyul'
AND P.Division='Zqthjt5jzmmu' AND P.Type='F';
Requête 11
SELECT Division,Count(*)
FROM Actvars A,Prodlevel P
WHERE A.Product=P.Code AND P.Type='B'
AND P.Group='S7jweujryiwn'
GROUP BY Division
Requête 12
SELECT Group,Count(*)
FROM Actvars A, Prodlevel P
WHERE A.Product=P.Code AND P.Class='P70j55l4hybv'
GROUP BY Group;
Requête 13
SELECT Avg(Unitssold)
FROM Actvars A,Prodlevel P, Timelevel T
WHERE A.Product=P.Code AND A.Time=T.Tid
AND P.Class='Df2i75i10ks8' AND P.Group='Xn2p90mnmf1o'
AND P.Type='B' AND T.Week='6' AND T.Day='11'
Requête 14
SELECT Retailer,Avg(Unitssold)
FROM Actvars A,Prodlevel P,Custlevel C,Chanlevel Ch
WHERE A.Product=P.Code AND A.Customer=C.Store AND
A.Channel=Ch.Base AND P.Type='R' AND C.Country='Algerie'
AND Ch.Sort='E'
GROUP BY Retailer;
Requête 15
SELECT Family,Count(*)
FROM Actvars A, Prodlevel P
WHERE A.Product=P.Code AND P.Type='R'
GROUP BY Family;
Requête 16
SELECT Store, Count(*)
FROM Actvars A,Timelevel T, Custlevel C
WHERE A.Time=T.Tid AND A.Customer=C.Store
AND T.Year='1995' AND C.Retailer='Y2kxn3oc7ryr'
AND C.Country='Maroc'
GROUP BY Store;
Requête 17
SELECT Count(*)
FROM Actvars A,Prodlevel P
WHERE A.Product=P.Code AND P.Type='R'
AND P.Family='Lyr3m3t4lolm'
Requête 18
SELECT Avg(UNITSSOLD)
FROM Actvars A,Custlevel C,Timelevel T
WHERE A.Customer=C.Store AND A.City=T.Channel
AND T.Month='1' AND T.Quarter='Q3' AND C.Gender='M';
Requête 19
SELECT sum(Dollarcost)
FROM Actvars A, Chanlevel H,Timelevel T
WHERE A.Channel=H.Base AND A.City=T.Channel
AND T.Month='1' AND H.All='ABCDEFGHIJKL';

Requête 20
SELECT Customer, Product, Sum(Dollarcost)
FROM Actvars A, Custlevel C,Prodlevel P
WHERE A.Customer=C.Store AND A.Product=P.Code AND
P.Family='Ytu811t81e1g' AND C.Retailer='Hc93hunb0rb8'
AND P.Group='Xn2p90mnmf1o'
GROUP BY Customer, Product ;
Requête 21
SELECT avg(Dollarcost)
FROM Actvars A,Custlevel C
WHERE A.Customer=C.Store AND C.City='Paris' ;
Requête 22
SELECT Time,Avg(Unitssold)
FROM Actvars A,Timelevel T
WHERE A.Time=T.Tid AND T.Week='3'
AND (T.Quarter='Q1' Or T.Quarter='Q2')
GROUP BY Time ;
Requête 23
SELECT Year, Division, Max(Unitssold)
FROM Actvars A, Custlevel C,Timelevel T ,Prodlevel P, Chanlevel Ch
WHERE A.Customer=C.Store AND A.Product=P.Code AND
A.Time=T.Tid AND A.Channel=Ch.Base AND Ch.Sort='B'
AND C.Depart='A' AND C.Gender='F' AND T.Year='1995'
AND C.City='Poitiers' AND P.Line='Qqpe0mi3wdhn'
GROUP BY Year, Division ;
Requête 24
SELECT Month,Count(*)
FROM Actvars A,Timelevel T
WHERE A.Time=T.Tid AND T.Quarter='Q3'
GROUP BY Month ;
Requête 25
SELECT Tid, Avg(Unitssold)
FROM Actvars A,Timelevel T
WHERE A.Time=T.Tid AND T.Year='1996'
GROUP BY Tid ;
Requête 26
SELECT Division,Avg(Unitssold)
FROM Actvars A,Timelevel T, Prodlevel P
WHERE .Time=T.Tid AND A.Product=P.Code
AND T.Month='7'
GROUP BY Division ;
Requête 27
SELECT Month, Avg(Unitssold)
FROM Actvars A,Timelevel T, Custlevel C
WHERE A.Time=T.Tid AND A.Customer=C.Store
AND (C.Country='Algerie' OR C.City='Tunis')
GROUP BY Month ;
Requête 28
SELECT City,Count(*)
FROM Actvars A,Custlevel C
WHERE A.Customer=C.Store AND C.Depart='A' AND
GROUP BY City
Requête 29
SELECT Max(Unitssold)
FROM Actvars A,Chanlevel Ch
WHERE A.Channel=Ch.Base AND Ch.All='Efghijklmnop'
Requête 30
SELECT Count(*)
FROM Actvars A,Chanlevel Ch,Timelevel T
WHERE A.Channel=Ch.Base AND A.Time=T.Tid
AND (T.Quarter='Q1' Or T.Quarter='Q2')
AND T.Year='1995' AND Ch.All='Bcdefghijklm'
Requête 31
SELECT Count(*)
FROM ACTVARS A, Chanlevel H
WHERE A.Channel=H.Base AND H.All='ABCDEFGHIJKL' ;
Requête 32
SELECT Year,Month, Max(Unitssold)
FROM Actvars A,Prodlevel P ,Timelevel T
WHERE A.Product=P.Code AND A.Time=T.Tid
AND (T.Month='1' Or T.Month='2') AND
P.Division='Zqthjt5jzmmu' AND T.Week='7'
GROUP BY Year,Month ;
Requête 33
SELECT Count(*)
FROM Actvars A,Custlevel C
WHERE A.Customer=C.Store AND C.City='Alger'
AND C.Retailer='Y2kxn3oc7ryr' ;
Requête 34
SELECT Class, Month, Min(Unitssold)
FROM Actvars A,Custlevel C, Prodlevel P, Timelevel T
WHERE A.Product=P.Code AND A.Customer=C.Store AND
A.Time=T.Tid AND C.City='Dijon' ANDC.Depart='A' AND
P.Type='A' AND P.Line='Qqpe0mi3wdhn'
GROUP BY Class, Month ;
Requête 35
SELECT Year,Sum(Dollarcost)
FROM Actvars A,Custlevel C ,Timelevel T
WHERE A.Customer=C.Store AND A.Time=T.Tid
AND T.Day='7' AND T.Week='7'
GROUP BY Year ;
Requête 36
SELECT Base, Count(*)
FROM Actvars A,Chanlevel Ch
WHERE A.Channel=Ch.Base AND Ch.Sort='E'
GROUP BY Base ;
Requête 37
SELECT Channel,Time, Sum(Dollarcost)
FROM Actvars A,Chanlevel Ch
WHERE A.Channel=Ch.Base AND Ch.All='Bcdefghijklm' AND
Ch.Sort='B'
GROUP BY Channel, Time ;
Requête 38
SELECT Product, Sum(Dollarcost)
FROM Actvars A,Timelevel T, Prodlevel P
WHERE A.Time=T.Tid AND A.Product=P.Code
AND (T.Month='1' Or T.Month='2') AND P.Type='R'
AND T.Week='5'
GROUP BY Product ;
Requête 39
SELECT sum(Dollarcost)
FROM Actvars A,Custlevel C,Prodlevel P,Timelevel T
WHERE A.Customer=C.Store AND A.Product=P.Code AND
P.LINE='MJ1F1U1EG009' AND C.Gender='F' ;
Requête 40
SELECT City,Count(*)
FROM Actvars A,Chanlevel H
WHERE A.Channel=H.Base AND H.All='ABCDEFGHIJKL'
GROUP BY City ;
Requête 41
SELECT Year, Max(Unitssold)
FROM Actvars A,Prodlevel P, Custlevel C, Timelevel T
WHERE A.Time=T.Tid AND A.Customer=C.Store
AND A.Product=P.Code AND P.Division='Zqthjt5jzmmu' AND
C.Country='France' AND C.Depart='B' AND T.Day='7'
GROUP BY Year
Requête 42
SELECT sum(DOLLARSALES)
FROM Actvars A, Prodlevel P,Timelevel T

```
WHERE A.Product=P.Code AND A.City=T.Channel
AND T.QUARTER='Q2' AND T.Year='1996'
AND P.DIVISION='P65YQPVI7E1B';
```
Requête 43
```
SELECT sum(UNITSSOLD)
FROM Actvars A, Custlevel C,Timelevel T
WHERE A.Customer=C.Store AND A.City=T.Channel
AND T.QUARTER='Q1' AND C.Gender='F';
```
Requête 44
```
SELECT Max(UNITSSOLD)
FROM Actvars A, Custlevel C,Timelevel T
WHERE A.Customer=C.Store AND A.City=T.Channel AND
T.QUARTER='Q1' AND C.Gender='F';
```
Requête 45
```
SELECT Count(*)
FROM Actvars A, Custlevel C,Timelevel T
WHERE A.Customer=C.Store AND A.City=T.Channel AND
T.Month='12' AND C.City='Dijon';
```
Requête 46
```
SELECT max(Dollarcost), min(UNITSSOLD)
FROM Actvars A, Custlevel C, Timelevel T
WHERE A.Customer=C.Store AND A.City=T.Channel AND
T.Month='12' AND C.City='Alger';
```
Requête 47
```
SELECT sum(Dollarcost), Avg(UNITSSOLD)
FROM Actvars A, Custlevel C,Timelevel T
WHERE A.Customer=C.Store AND A.City=T.Channel AND
T.Month='12' AND C.City='Oran';
```
Requête 48
```
SELECT Avg(UNITSSOLD)
FROM Actvars A,Chanlevel H,Prodlevel P,Timelevel T
WHERE A.Product=P.Code AND A.Channel=H.Base
AND A.City=T.Channel AND T.Month='1'
AND H.All='ABCDEFGHIJKL';
```
Requête 49
```
SELECT Min(UNITSSOLD)
FROM Actvars A,Custlevel C,Timelevel T
WHERE A.Customer=C.Store AND A.City=T.Channel AND
T.Month='1' AND C.Gender='OIZGIUUM6VZ8';
```
Requête 50
```
SELECT Time, Max(Unitssold)
FROM Actvars A,Custlevel C,Timelevel T
WHERE A.Customer=C.Store AND A.Time=T.Tid
AND C.City='Oran' AND (T.Month='1' Or T.Month='2') AND
C.Retailer='Zstv6mycbs7u'
GROUP BY Time;
```
Requête 51
```
SELECT Count(*)
FROM Actvars A,Chanlevel H,CustlevelC,Prodlevel P,TimlevelT
WHERE A.Customer=C.Store AND A.Product=P.Code AND
A.Channel=H.Base AND A.City=T.Channel AND T.Year='1995'
AND T.Month='5' AND C.Gender='M';
```
Requête 52
```
SELECT Line,Sum(Dollarcost)
FROM Actvars A,Prodlevel P, Chanlevel H
WHERE A.Product=P.Code AND A.Channel=H.Base AND
P.Class='Osvspq7bsiic' AND H.All='Efghijklmnop'
AND P.Type='B' AND P.Group='Xn2p90mnmf1o'
GROUP BY Line;
```
Requête 53
```
SELECT Month, Avg(UNITSSOLD)
FROM Actvars A,Prodlevel P,Timelevel T
WHERE A.Product=P.Code AND A.City=T.Channel
AND T.Year='1995' AND P.GROUP='DDYH5IKY9PYQ'
GROUP BY Month;
```
Requête 54
```
SELECT Month
FROM Actvars A,Custlevel C,Timelevel T,Prodlevel P
WHERE A.Customer=C.Store AND C.City='Dijon'
AND C.Retailer='OIZGIUUM6VZ8' AND C.Gender='M';
```
Requête 55
```
SELECT Product
FROM Actvars A,Timelevel T
WHERE A.City=T.Channel AND T.Year='1995'
GROUP BY Product;
```
Requête 56
```
SELECT Count(*)
FROM Actvars A,Custlevel C
WHERE A.Customer=C.Store AND C.Gender='F';
```
Requête 57
```
SELECT Month,Sum(Dollarcost)
FROM Actvars A,Custlevel C ,Timelevel T
WHERE A.Customer=C.Store AND A.City=T.Channel AND
C.City='Dijon'
GROUP BY Month;
```
Requête 58
```
SELECT City,avg(Dollarcost)
FROM Actvars A,Custlevel C
WHERE A.Customer=C.Store AND C.City='Paris'
GROUP BY City;
```
Requête 59
```
SELECT Product, Sum(Dollarcost)
FROM Actvars A,Timelevel T, Prodlevel P, Chanlevel H
WHERE A.Time=T.Tid AND A.Product=P.Code AND
A.Channel=H.Base AND T.Year='1996' AND H.Sort='A' AND
H.All='Efghijklmnop' AND P.Class='Kybelcz9a2h1'
AND P.Family='Wryojud9s31o'
GROUP BY Product;
```
Requête 60
```
SELECT Month
FROM Actvars A,Custlevel C,Timelevel T,Prodlevel P
WHERE A.Customer=C.Store AND C.Gender='F'
AND C.Retailer='OIZGIUUM6VZ8' AND C.City='Paris';
```

Annexe B : Liste des publications

Cet annexe présente la liste des publications réalisées dans le cadre de cette thèse. La liste des publications est présentée en deux parties. Chaque partie correspondant aux travaux qui s'articulent autour d'une technique d'optimisation étudiée et comporte la liste des articles acceptés (leur résumé et leur principale contribution).

Liste des articles relatifs à la première technique d'optimisation

Rima Bouchakri, Ladjel Bellatreche : On Simplifying Integrated Physical Database Design. 15th East-European Conference on Advances in Databases and Information Systems (ADBIS'11) : 43-56, 2011. [17]
Résumé : This paper deals with the problem of integrated physical database design involving two optimization techniques : horizontal data partitioning (HDP) and bitmap join indexes (BJI). These techniques compete for the same resource representing selection attributes. This competition incurs attribute interchangeability phenomena, where same attribute(s) may be used to select either HDP or BJI schemes. Existing studies dealing with integrated physical database design problem not consider this competition. We propose to study its contribution on simplifying the complexity of our problem. Instead of tackling it in an integrated way, we propose to start by assigning to each technique its own attributes and then it launches its own selection algorithm. This assignment is done using the K-Means method. Our design is compared with the state of the art work using APB1 benchmark. The results show that an interchangeability attribute-aware database designer can improve significantly query performance within the less space budget.
Contribution de l'article : proposer une sélection jointe de la fragmentation horizontale et des index de jointures binaires, et proposer un algorithme génétique pour la sélection les index mono-attributs.

Rima Bouchakri, Ladjel Bellatreche : Sélection statique et incrémentale des index de jointure binaires multiples. 7ème Journée Francophone sur les Entrepôts de Données et l'Analyse en ligne (EDA'11), Revue des Nouvelles Technologies : 171-186, 2011 [18].
Résumé : Les index de jointure binaires ont montré leur intérêt dans la réduction des coûts d'exécution des requêtes décisionnelles définies sur un schéma relationnel en étoile. Leur sélection reste cependant difficile vu le vaste et complexe espace de recherche à explorer. Peu d'algorithmes de sélection des index de jointure existent, contrairement à la sélection des index définis sur une seule table qui a connu un intérêt particulier auprès de la communauté des bases de données traditionnelles. La principale particularité de ces algorithmes est qu'ils sont statiques et supposent la connaissance préalable des requêtes. Dans cet article, nous présentons une démarche de sélection des index de

jointures binaires définis sur plusieurs attributs appartenant à des tables de dimension en utilisant des algorithmes génétiques. Ces derniers sont utilisés dans le cadre statique et incrémental qui prévoit l'adaptation des index sélectionnés à l'arrivée de nouvelles requêtes. Nous concluons nos travaux par une étude expérimentale démontrant l'intérêt de la sélection des index de jointure binaires multiple, de l'élagage de l'espace de recherche et de l'efficacité des algorithmes génétiques dans les cas statique ou incrémentale.
Contribution de l'article : proposer une sélection incrémentale des index de jointures binaires basée sur les algorithmes génétiques.

Rima Bouchakri, Kamel Boukhalfa, Ladjel Bellatreche : Algorithmes de sélection des index de jointure binaires mono et multi-attributs. Ingénierie des Systèmes d'Information (ISI'11) 16(6) : 91-116 (2011) [25].
Résumé : La conception physique des entrepôts de données relationnels est basée essentiellement sur l'utilisation des index afin d'optimiser les requêtes de jointure en étoile connues pour leur complexité. Les index de jointure binaires sont adaptés pour réduire le coût d'exécution de ces requêtes. Ils sont définis sur la table des faits en utilisant un ou plusieurs attributs de table(s) de dimension. Sélectionner une configuration d'index réduisant le coût d'exécution d'un ensemble de requêtes en présence d'une contrainte de stockage est une tâche difficile. Dans cet article, une classification des principaux travaux effectués dans ce domaine est proposée. Nous formalisons le problème de sélection des index de jointure binaires. Nous présentons par la suite un ensemble d'algorithmes de sélection des index de jointure binaires mono et mult-iattributs. Cette multitude d'algorithmes permet aux administrateurs de choisir l'algorithme qui leur convient. Une comparaison des algorithmes proposés est présentée en utilisant un modèle de coût théorique et une validation réelle sur le SGBD Oracle10G.
Contribution de l'article : proposer une multitude d'algorithmes de sélection des index de jointure binaires et les comparer avec les travaux existants.

Rima Bouchakri, Ladjel Bellatreche, Khaled-Walid Hidouci : Static and Incremental Selection of Multi-table Indexes for Very Large Join Queries. 16th East-European Conference on Advances in Databases and Information Systems (ADBIS'12) : 43-56, 2012. [21]
Résumé : Multi-table indexes boost the performance of extremely large databases by reducing the cost of joins involving several tables. The bitmap join indexes (BJI) are one of the most popular examples of this category of indexes. They are well adapted for point and range queries. Note that the selection of multi-table indexes is more difficult than the mono-table indexes, considered as the pioneer of database optimisation problems. The few studies dealing with the BJI selection problem in the context of relational data warehouses have three main limitations : (i) they consider BJI defined on only two tables (a fact table and a dimension table) by the use of one or several attributes of that dimension table, (ii) they use simple greedy algorithms to pick the right indexes and (iii) their algorithms are static. In this paper, we propose genetic algorithms for selecting BJI using a large number of attributes belonging to n (≥ 2) dimension tables in the static and incremental ways. Intensive experiments are conducted to show the efficiency of our proposal.
Contribution de l'article : proposer une sélection incrémentale des index de jointures binaire mono et multi-attributs basée sur les algorithmes génétiques et l'élagage de l'espace de recherche dirigé par les requêtes avec une validation des algorithmes sous le SGBD Oracle 11g.

Rima Bouchakri, Ladjel Bellatreche, Zoé Faget, Sébastien Bress : Sélection statique et incrémentale des Index de Jointure Binaire : concepts, algorithmes, étude de performance. Journal of Decision Systems (JDS'12) Vol. 21(1) : 51-70 (2012) [20].
Résumé : Afin de réduire le temps d'exécution des requêtes décisionnelles, l'administrateur a la possibilité de sélectionner des index de jointure binaires (IJB). Cette sélection demeure une tâche difficile vue la complexité de l'espace de recherche à parcourir. De ce fait, un grand intérêt est porté à la mise en oeuvre d'algorithmes de sélection. Cependant, ces algorithmes sont statiques. Dans cet article, nous centrons nos travaux sur la sélection des index de jointures binaires définis sur plusieurs attributs appartenant à des tables de dimension en utilisant des algorithmes génétiques. Nous présentons deux types d'algorithmes : des algorithmes de sélection statiques et des algorithmes de sélection incrémentales qui prévoient l'adaptation des index sélectionnés à l'arrivée de nouvelles requêtes. Nous concluons nos travaux par une étude expérimentale démontrant l'apport de notre sélection des index de jointure binaires en comparaison avec les travaux de sélection statiques et incrémentales existants.
Contribution de l'article : cet article est une extension en papier journal de la précédente proposition, suite à la sélection des meilleurs articles proposés et leur réévaluation, avec classification des travaux de recherches et positionnement de notre proposition dans la littérature.

Liste des articles relatifs à la seconde technique d'optimisation

Rima Bouchakri, Ladjel Bellatreche : Sélection Incrémentale d'un Schéma de Fragmentation Horizontale d'un Entrepôt de Données Relationnel. 8ème Journée Francophone sur les Entrepôts de Données et l'Analyse en ligne (EDA'12), Revue des Nouvelles Technologies : 2-16, 2012 [19].
Résumé : La fragmentation horizontale permet de réduire la complexité des requêtes décisionnelles exécutées sur un entrepôt de données relationnel. Elle se base sur le principe de réorganisation des données qui ne nécessite pas un espace de stockage supplémentaire. Cependant, sélectionner un schéma de fragmentation horizontale d'un entrepôt n'est guère une tâche facile, vu l'espace de recherche très complexe à exploiter. Les algorithmes existants sélectionnent un schéma de fragmentation lors de la phase de conception physique d'un entrepôt, afin d'optimiser une charge de requêtes préalablement connue. Ces algorithmes ne prendre pas en considération les changements au niveau des requêtes. Dans cet article, nous proposons d'effectuer une sélection d'un schéma de fragmentation dite incrémentale basée sur les algorithmes génétiques. Notre approche permet l'optimisation de l'exécution de la charge de requêtes décisionnelles et l'adaptation du schéma de fragmentation aux changements de la charge. Nous réalisons une étude expérimentale qui montre l'intérêt de la sélection incrémentale d'un schéma de fragmentation.
Contribution de l'article : proposer une sélection incrémentale de la fragmentation horizontale dans le contexte des entrepôts de données basée sur les algorithme génétiques.

Ladjel Bellatreche, Rima Bouchakri, Alfredo Cuzzocrea, Sofian Maabout : Horizontal partitioning of very-large data warehouses under dynamically-changing query workloads via incremental algorithms. 28th Annual ACM Symposium on Applied Computing (SAC'13) : 208-210, 2013 [7].
Résumé : With the explosion of the size of data warehousing applications, the horizontal data partitioning is well adapted to reduce the cost of complex OLAP queries and the warehouse manageability. It is considered as a non redundant optimization technique. Selecting a fragmentation schema for a

given data warehouse is NP-hard problem. Several studies exist and propose heuristics to select near optimal solutions. Most of these heuristics are static, since they assume the existence of a priori known set of queries. Note that in real life applications, queries may change dynamically and fragmentation heuristics need to integrate these changes. In this paper, we propose an incremental selection of fragmentation schemes using on genetic algorithms. Intensive experiments are conducted to validate our proposal.
Contribution de l'article : proposer deux démarches de sélection incrémentale d'un schéma de fragmentation avec évaluations et comparaisons.

Ladjel Bellatreche, Rima Bouchakri, Alfredo Cuzzocrea, Sofian Maabout : Incremental Algorithms for Selecting Horizontal Schemas of Data Warehouses : The Dynamic Case. Conference on Data Management in Grid and P2P Systems (GLOBE'13) : 13-25, 2013 [8].
Résumé : Looking at the problem of effectively and efficiently partitioning data warehouses, most of state-of-the-art approaches, which are very often heuristic-based, are static, since they assume the existence of an a-priori known set of queries. Contrary to this, in real-life applications, queries may change dynamically and fragmentation heuristics need to integrate these changes. Following this main consideration, in this paper we propose and experimentally assess an incremental approach for selecting data warehouse fragmentation schemes using genetic algorithms.
Contribution de l'article : proposer une sélection incrémentale de la fragmentaiton horizontale avec optimisation du coût de maintenance causé par la refragmentation des entrepôts de données. Cet article est une extension en papier long de la précédente proposition.

Rima Bouchakri, Ladjel Bellatreche : A Coding Template for Handling Static and Incremental Horizontal Partitioning in Data Warehouses. Journal of Decision Systems (JDS'13) Vol. 22, 2013 [24].
Résumé : Today, data feeding warehouses comes from devices with high speed and may affect the selected optimization techniques such as horizontal data partitioning (HDP). HDP helps reducing the cost of complex OLAP queries and facilitates the warehouse manageability. Selecting a partitioning schema for a given data warehouse is a NP-hard problem. Several studies exist and propose heuristics to select near optimal solutions. Most of these heuristics are static, because they assume the existence of an a priori known set of queries. However, in real life applications queries may change dynamically and the partitioning heuristics need to integrate these changes. An easy and naive way to deal with this problem is to make the heuristics incremental by making efforts on implementation and coding levels. In this paper, we propose a new vision for handling the incremental selection of a HDP in a relational data warehouse. Our main efforts are concentrated on having a template coding that supports efficiently the changes of queries. First, we present a rich state of art regarding the concepts of HDP (types, modes, algorithms). Second, we present a flexible coding to manage the static partitioning, where a genetic algorithm is proposed. Third, we extend our genetic algorithm to deal with the incremental aspects of queries. Finally, we conduct intensive experiments to validate our proposal.
Contribution de l'article : présenter une classification des travaux de fragmentation horizontale des entrepôts de données et positionner nos travaux par rapport aux travaux existants, puis proposer une nouvelle vision de sélection statique et incrémentale qui exploite un codage flexible d'un schéma de fragmentation. Cet article comporte une extensions des deux propositions précédentes.

Rima Bouchakri, Ladjel Bellatreche, Zoé Faget : Algebra-based Approach for Incremental Data Warehouse Partitioning, 25th International Conference on Database and Expert Systems Applications (DEXA'14), Germany, 2014 [23].

Résumé : Horizontal Data Partitioning is an optimization technique well suited to optimize star-join queries in Relational Data Warehouses. Most works focus on a static selection of a fragmentation schema. However, due to the evolution of data warehouses and the ad hoc nature of queries, the development of incremental algorithms for fragmentation schema selection has become a necessity. In this work, we present a Partitioning Algebra containing all operators needed to update a schema when a new query arrives. To identify queries which should trigger a schema update, we introduce the notion of query profiling.

Contribution de l'article : proposer une algèbre de fragmentation qui exploite le codage flexible d'un schéma de fragmentation puis l'emploie pour définir un Profiling des requêtes décisionnelles qui permet de contrôler la refragmentation des entrepôts de données.

Glossaire

ED : Entrepôt de Données
FH : Fragmentation Horizontale
FHP : Fragmentation Horizontale Primaire.
FHD : Fragmentation Horizontale Dérivée.
SD : Sous Domaine d'un attributs.
AS : Attributs de Sélection issus des tables dimensions et figurants dans les requêtes .
SF : Schéma de Fragmentation.
IJB : Index de Jointure Binaire.
IS : Index de jointure binaire Simple.
IM : Index de jointure binaire multiple.
Q : Charge de requêtes.
RowID : Identificateur de ligne dans un SGBD relationnel.
W : Le nombre de fragments maximum de la table de faits que l'administrateur doit fixer.
S : Espace de stockage des index.
PMO : Problème d'optimisation Multi-Objectif.
AG : Algorithme Génétique.
AF : Algèbre de Fragmentation.
PQ : Profiling des reQuêtes.
DM : DataMining (fouille de données).
SGBD : Système de Gestion de Bases de Données
SQL : Structured Query Language.
OLAP : OnLine Analytical Processing.
OLTP : OnLine Transactional Processing.

Bibliographie

[1] S. Agrawal, S. Chaudhuri, L. Kollar, A. Marathe, V. Narasayya, and M. Syamala. Database tuning advisor for microsoft sql server 2005. *vldb (VLDB'05)*, pages 1110–1121, 2004.

[2] K. Aouiche, O. Boussaid, and F. Bentayeb. Automatic selection of bitmap join indexes in datawarehouses. *7th International Conference on Data Warehousing and Knowledge Discovery (DAWAK'05)*, pages 64–73, 2005.

[3] S. Azefack, K. Aouiche, and J. Darmont. Dynamic index selection in data warehouses. *4th International Conference on Innovations in Information Technology (IIT'07), Dubai*, 2007.

[4] T. Bäck. *Evolutionnary algorithms in theory and practice*. Oxford University Press, New York, 1995.

[5] L. Bellatreche. Une méthodologie pour la fragmentation dans les entrepôts des données. *in 18th Congrès d'Informatique Des Organisations et Systèmes d'Information et de Décision (INFORSID'00), Lyon*, pages 245–260, May 2000.

[6] L. Bellatreche. Utilisation des vues matérialisées, des index et de la fragmentation dans la conception logique et physique d'un entrepôts de données. Phd. thesis, Blaise Pascal University, France, 2000.

[7] L. Bellatreche, R. Bouchakri, A. Cuzzocrea, and S. Maabout. Horizontal partitioning of very-large data warehouses under dynamically-changing query workloads via incremental algorithms. *28th Symposium On Applied Computing (SAC'13)*, pages 208–210, 2013.

[8] L. Bellatreche, R. Bouchakri, A. Cuzzocrea, and S. Maabout. Incremental algorithms for selecting horizontal schemas of data warehouses : The dynamic case. *7th International Conference on Data Management in Cloud, Grid and P2P Systems (Globe'13)*, pages 13–25, 2013.

[9] L. Bellatreche and K. Boukhalfa. Yet another algorithm for selecting bitmap join indexes. *In International Conference on Data Warehousing and Knowledge Discovery (DaWaK'2010)*, pages 105–116, 2010.

[10] L. Bellatreche, K. Boukhalfa, and H. I. Abdalla. Saga : A combination of genetic and simulated annealing algorithms for physical data warehouse design. *23rd British National Conference on Databases (BNCOD'06)*, pages 212–219, July 2006.

[11] L. Bellatreche, K. Boukhalfa, and S. Caffiau. Paradmin : Un outil d'assistance à l'administration et tuning d'un entrepôt de données. *4ème Journée Francophone sur les Entrepôts de données et l'Analyse en ligne (EDA'08), Revue des Nouvelles Technologies de l'Information, RNTI-B-4*, pages 99–114, 2008.

[12] L. Bellatreche, K. Boukhalfa, and M. Mohania. Pruning search space of physical database design. *18th International Conference On Database and Expert Systems Applications (DEXA'07)*, pages 479–488, 2007.

[13] L. Bellatreche, K. Boukhalfa, and P. Richard. Data partitioning in data warehouses : Hardness study, heuristics and oracle validation. *International Conference on Data Warehousing and Knowledge Discovery (DaWaK'2008)*, pages 87–96, 2008.

[14] L. Bellatreche, K. Boukhalfa, and P. Richard. Referential horizontal partitioning selection problem in data warehouses : Hardness study and selection algorithms. *International Journal of Data Warehousing and Mining (IJDWM'09)*, 5(4) :1–23, 2009.

[15] L. Bellatreche, K. Karlapalem, and M. Schneider. On efficient storage space distribution among materialized views and indices in data warehousing environments. *Proceedings of the International Conference on Information and Knowledge Management (ACM CIKM'2000)*, pages 397–404, 2000.

[16] L. Bellatreche, R. Missaoui, H. Necir, and H. Drias. A data mining approach for selecting bitmap join indices. *Journal of Computing Science and Engineering (JCSE'08)*, 2(1) :206–223, January 2008.

[17] R. Bouchakri and L. Bellatreche. On simplifying integrated physical database design. *15th East-European Conference on Advances in Da-*

tabases and Information Systems (ADBIS'11), pages 333–346, 2011.

[18] R. Bouchakri and L. Bellatreche. Sélection statique et incrémentale des index de jointure binaires multiples. *7ème Journée Francophone sur les Entrepôts de données et l'Analyse en ligne (EDA'11), Revue des Nouvelles Technologies RNTI-B-7*, pages 171–186, 2011.

[19] R. Bouchakri and L. Bellatreche. Sélection incrémentale d'un schéma de fragmentation horizontale d'un entrepôt de données relationnel. *8ème Journée Francophone sur les Entrepôts de données et l'Analyse en ligne (EDA'12), Revue des Nouvelles Technologies RNTI-B-8*, pages 2–16, 2012.

[20] R. Bouchakri and L. Bellatreche. Sélection statique et incrémentale des index de jointure binaire : concepts, algorithmes, étude de performance. *Journal of Decision Systems (JDS'12)*, 21(1) :51–70, 2012.

[21] R. Bouchakri and L. Bellatreche. Static and incremental selection of multi-table indexes for very large join queries. *16th East-European Conference on Advances in Databases and Information Systems (ADBIS'12)*, 2012.

[22] R. Bouchakri, L. Bellatreche, and K. Boukhalfa. Une approche par k-means de sélection multiple de structures d'optimisation dans les entrepôts de données. *6ème Journée Francophone sur les Entrepôts de données et l'Analyse en ligne (EDA'10), Revue des Nouvelles Technologies RNTI-B-6*, 2010.

[23] R. Bouchakri, L. Bellatreche, and Z. Faget. Algebra-based approach for incremental data warehouse partitioning. *25th International Conference on Database and Expert Systems Applications (DEXA'14), Germany*, 2014.

[24] R. Bouchakri, L. Bellatreche, Z. Faget, and S. Breß. A coding template for handling static and incremental horizontal partitioning in data warehouses. *Journal of Decision Systems (JDS'14)*, 23(4) :481–498, 2014.

[25] R. Bouchakri, K. Boukhalfa, and L. Bellatreche. Algorithmes de sélection des index de jointure binaires mono et multi-attributs. *Ingénierie des Systèmes d'Information (ISI'11)*, 16(6) :91–116, 2011.

[26] K. Boukhalfa. De la conception physique aux outils d'administration et de tuning des entrepôts de données. Phd. thesis, Poitiers University, France, 2009.

[27] K. Boukhalfa, L. Bellatreche, and Z. Alimazighi. Hp&bji : A combined selection of data partitioning and join indexes for improving olap performance. *Annals of Information Systems, Special Issue on new trends in data warehousing and data analysis, Springer (AoIS'08)*, 3 :179–2001, November 2008.

[28] K. Boukhalfa, L. Bellatreche, and B. Ziani. Index de jointure binaires : Stratégies de sélection et étude de performances. *6éme Journée Francophone sur les Entrepôts de données et l'Analyse en ligne (EDA'10), Revue des Nouvelles Technologies RNTI-B-6*, 2010.

[29] S. Ceri, M. Negri, and G. Pelagatti. Horizontal data partitioning in database design. *Proceedings of the ACM SIGMOD International Conference on Management of Data. SIGPLAN Notices (SIGMOD'82)*, pages 128–136, 1982.

[30] C. Y. Chan and Y. E. Ioannidis. Bitmap index design and evaluation. *sigmod (SIGMOD'82)*, pages 355–366, June 1998.

[31] V. Chankong and Y. Haimes. *Multiobjective Decision Making Theory and Methodology*. Elsevier Science, New York, 1983.

[32] A. Charnes, W. W. Cooper, and R. Ferguson. Optimal estimation of executive compensation by linear programming. *Management science*, 1 :138–151, 1955.

[33] A. Charnes, W. W. Cooper, and R. Ferguson. Goal programming and multiple objectives optimisations. *European Journal of Operational Research*, 1 :39–54, 1977.

[34] S. Chaudhuri and V. Narasayya. Self-tuning database systems : A decade of progress. *vldb (VLDB'07)*, pages 3–14, September 2007.

[35] C. Coello. An updated survey of gabased multiobjective optimization techniques. *Technical Report : Lania Laboratory, Veracruz, Mexico*, 1998.

[36] C. Coello, G. B. Lamont, and D. van Veldhuizen. Evolutionary algorithms for solving multiobjective problems. *Genetic and Evolutionary Computation, Springer US*, 2007.

[37] Y. Collett and P. Siarry. Optimisation multiobjectif. *Groupe Eyrolles*, 2002.

[38] D. Comer. The ubiquitous b-tree. *ACM Computing Surveys*, 11(2) :121–137, 1979.

[39] I. Corporation. Informix-online extended parallel server and informix-universal server : A new generation of decision-support indexing for enterprise data warehouses. *White Paper*, 1997.

[40] O. Council. Apb-1 olap benchmark, release ii. *http ://www.olapcouncil.org/research/bmarkly.htm*, 1998.

[41] C. Curino, E. Jones, R.Popa, N. Malviya, E. Wu, S.Madden, H. Balakrishnan, and N. Zeldovich. Relational cloud : a database service for

the cloud. *Fifth Biennial Conference on Innovative Data Systems Research (CIDR'11), USA*, pages 235–240, 2011.

[42] C. D. The dificulty of optimum index selection. *ACM Transactions on Database Systems (TODS)*, 3(4) :440–445, march 1978.

[43] B. Dageville, D. Das, K. Dias, K. Yagoub, M. Zait, and M. Ziauddin. Automatic sql tuning in oracle 10g. *vldb (VLDB'04)*, pages 1098–1109, 2004.

[44] K. Deb, A. Pratap, S. Agrawal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm : Nsga-ii. *IEEE Transactions on Evolutionary Computation*, 6(2) :182–197, 2002.

[45] H. Derrar, M. Ahmed-Nacer, and O. Boussaid. Dynamic distributed data warehouse design. *Journal of Intelligent Information and Database Systems*, 7(1), 2013.

[46] T. Dokeroglu, S. A. Sert, and M. S. Cinar. Evolutionary multiobjective query workload optimization of cloud data warehouses. *The Scientific World Journal*, 2014.

[47] G. Eadon, E. I. Chong, S. Shankar, A. Raghavan, J. Srinivasan, and S. Das. Supporting table partitioning by reference in oracle. *sigmod (SIGMOD'82)*, pages 1111–1122, 2008.

[48] N. El-Tazi and H. V. Jagadish. Bpi : Xml query evaluation using bitmapped path indices. *EDBT/ICDT Workshops*, pages 55–64, 2009.

[49] C. M. Fonseca and P. J. Fleming. Genetic algorithms for multiobjective optimization : Formulation, discussion and generalization. *Proceedings of the Fifth International Conference on Genetic Algorithms, USA (ICGA'93)*, pages 416–423, 1993.

[50] K. Fujioka, Y. Uematsu, and M. Onizuka. Application of bitmap index to information retrieval. *Proceedings of the 17th International Conference on World Wide Web (WWW'08)*, pages 1109–1110, 2008.

[51] S. Gass and T. Saaty. The computational algorithm for the parametric objective function. *Naval Research Logistics Quarterly*, 2(1-2) :39–45, 1955.

[52] V. Gopalkrishnan, Q. Li, and K. Karlapalem. Efficient query processing with associated horizontal class partitioning in an object relational data warehousing environment. *Proceedings of 2nd International Workshop on Design and Management of Data Warehouses (DMDW'00)*, pages 1–9, June 2000.

[53] K. Gouda and M. Zaki. Genmax : An efficient algorithm for mining maximal frequent itemsets. *Data Mining and Knowledge Discovery, Vol. 11, Springer, Heidelberg*, pages 1–20, 2005.

[54] A. Gupta, S. Sudarshan, and S. Viswanathan. Query scheduling in multi query optimization. *International Database Engineering & Applications Symposium, (IDEAS'01), Grenoble, France*, pages 11–19, 2001.

[55] J. A. Hartigan and M. A. Wong. Algorithm as 136 : A k-means clustering algorithm. *Journal of the Royal Statistical Society, Vol. 28, No. 1*, pages 100–108, 1979.

[56] H.Mahboubi and J. Darmont. Data mining-based fragmentation of xml data warehouses. *ACM 11th International Workshop on Data Warehousing and OLAP (DOLAP'08)*, pages 9–16, 2008.

[57] J. H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, Michigan, 1975.

[58] G. Holmes, A. Donkin, and I. H. Witten. Weka : a machine learning workbench. *Second Australia and New Zealand Conference on Intelligent Information Systems, Brisbane, Australia*, pages 357–361, 1994.

[59] J. Horn, N. Nafpliotis, and D. Goldberg.

[60] IBM. Db2 product family. *Available at http ://www-306.ibm.com/software/data/db2/.*

[61] Y. Ioannidis and Y. Kang. Randomized algorithms algorithms for optimizing large join queries. *sigmod (SIGMOD'90)*, pages 9–22, 1990.

[62] A. N. J. Durillo. jmetal : A java framework for multi-objective optimization. *Advances in Engineering Software*, 42 :760 – 771, 2011.

[63] A. Jain, M. Murty, and P. Flynn. Data clustering : A review. *ACM Computing Surveys*, 31(3), 1999.

[64] K. Karlapalem and Q. Li. A framework for class partitioning in object-oriented databases. *Distributed and Parallel Databases Journal*, 8(3) :333–366, 2000.

[65] R. Keeney and H. Raiffa. Decisions with multiple objectives preferences and value tradeoffs. *Cambridge University Press*, 1993.

[66] R. Kimball and K. Strehlo. Why decision support fails and how to fix it. *SIGMOD Record*, 24(3) :92–97, September 1995.

[67] J. Knowles and D. Corne. The pareto archived evolution strategy : a new baseline algorithm for pareto multiobjective optimisation. *Proceedings of the 1999 Congress on Evolutionary Computation (CEC'99)*, 1 :98–105, 1999.

[68] M. Lawrence. Multiobjective genetic algorithms for materialized view selection in olap data warehouses. *The Genetic and Evolutionary Computation Conference (GECCO'06)*, pages 699–706, 2006.

[69] K. H. Lee and B. Moon. Bitmap indexes for relational xml twig query processing. *Proceedings of the 18th ACM Conference on Information and Knowledge Management (CIKM'09)*, pages 465–474, 2009.

[70] M. Lukasiewycz, M. Glaß, F. Reimann, and J. Teich. Opt4j : A modular framework for meta-heuristic optimization. *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*, pages 1723–1730, 2011.

[71] R. Marler and J. Arora. Survey of multi-objective optimization methods for engineering. *Struct. Multidisc. Optim. 26*, 26 :369–395, 2004.

[72] W. T. McCormick, P. J. Schweitzer, and T. W. White. Problem decomposition and data reorganization by a clustering technique. *Operation Research*, 20(5) :993–1009, September 1972.

[73] T. Murata and H. Ishibuchi. Moga : multi-objective genetic algorithms. *In Proceedings of the 2nd IEEE International Conference on Evolutionary Computation*, 1 :289–294, 1995.

[74] S. Navathe, S. Ceri, G. Wiederhold, and D. J. Vertical partitioning algorithms for database design. *ACM Transaction on Database Systems*, 9(4) :681–710, December 1984.

[75] S. Navathe, K. Karlapalem, and M. Ra. A mixed partitioning methodology for distributed database design. *Journal of Computer and Software Engineering*, 3(4) :395–426, 1995.

[76] S. Navathe and M. Ra. Vertical partitioning for database design : a graphical algorithm. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 440–450, 1989.

[77] P. O'Neil and G. Graefe. Multi-table joins through bitmapped join indices. *SIGMOD Record*, 24(3) :8–11, September 1995.

[78] P. O'Neil and D. Quass. Improved query performance with variant indexes. *sigmod (SIGMOD'97)*, pages 38–49, May 1997.

[79] M. T. Özsu and P. Valduriez. Distributed database systems : Where are we now ? *IEEE COMPUTER*, 24(8) :68–78, August 1991.

[80] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems : Second Edition.* Prentice Hall, 1999.

[81] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Discovering frequent closed itemsets. *International Conference on Database Theory (ICDT'99)*, pages 398–416, 1999.

[82] J. Rao, C. Zhang, G. Lohman, and N. Megiddo. Automating physical database design in a parallel database. *sigmod (SIGMOD'02)*, pages 558–569, June 2002.

[83] K. Ross, D. Srivastava, and S. Sudarshan. Materialized view maintenance and integrity constraint checking : Trading space for time. *sigmod (SIGMOD'96)*, pages 447–458, June 1996.

[84] A. Sanjay, V. R. Narasayya, and B. Yang. Integrating vertical and horizontal partitioning into automated physical database design. *sigmod (SIGMOD'04)*, pages 359–370, June 2004.

[85] A. Sanjay, C. Surajit, and V. R. Narasayya. Automated selection of materialized views and indexes in microsoft sql server. *vldb (VLDB'00)*, pages 496–505, September 2000.

[86] S. Sarawagi. Indexing olap data. *IEEE Data Engineering Bulletin*, 20(1) :36–43, March 1997.

[87] J. D. Schaffer. Multiple objective optimization with vector evaluated genetic algorithms. *Proceedings of the 1st International Conference on Genetic Algorithms*, pages 93–100, 1985.

[88] N. Srinivas and K. Deb. Multiobjective optimization using non-dominated sorting in genetic algorithms. *Journal of Evolutionary Computation*, 2(8) :221, 1995.

[89] T. Stöhr, H. Märtens, and E. Rahm. Multi-dimensional database allocation for parallel data warehouses. *vldb (VLDB'00)*, pages 273–284, 2000.

[90] T. Stöhr and E. Rahm. Warlock : A data allocation tool for parallel warehouses. *Proceedings of the International Conference on Very Large Databases*, pages 721–722, 2001.

[91] R. B. Systems. Star schema processing for complex queries. *White Paper*, July 1997.

[92] K. Tekaya. Dynamic distributed data warehouse design. *IRMA International Conference*, 2007.

[93] M. Thiele, A. Bader, and W. Lehner. Multi-objective scheduling for real-time data warehouses. *Computer Science - R&D*, 24(3) :137–151, 2009.

[94] P. Valduriez. Join indices. *ACM Transactions on Database Systems*, 12(2) :218–246, June 1987.

[95] G. Valentin, M. Zuliani, D. C. Zilio, G. M. Lohman, and A. Skelley. Db2 advisor : An optimizer smart enough to recommend its own indexes. *Proceedings of the 17th International Conference on Data Engineering ICDE'00*, pages 101–110, 2000.

[96] P. Wehrle, M. Miquel, and A. Tchounikine. A model for distributing and querying a data warehouse on a computing grid. *11th International Conference on Parallel and Distributed Systems (ICPADS 2005)*, pages 203–209, July 2005.

[97] K. Wu, E. Otoo, and A. Shoshani. An efficient compression scheme for bitmap indices. *In Proceedings of the ACM Transactions on Database Systems (TODS'06)*, 2006.

[98] C. Zhang and J. Yang. Genetic algorithm for materialized view selection in data warehouse environments. *Proceeding of the International Conference on Data Warehousing and Knowledge Discovery (DAWAK'99)*, pages 116–125, September 1999.

[99] Y. Zhang and M.-E. Orlowska. On fragmentation for distributed database design. *Information Sciences*, 1(3) :117–132, 1994.

[100] D. Zilio, J. Rao, S. Lightstone, G. Lohman, A. Storm, C. Garcia-Arellano, and S. Fadden. Db2 design advisor : Integrated automatic physical database design. *Proceedings of the Thirtieth International Conference on Very Large Data Bases (VLDB '04)*, 30 :1087–1097, 2004.

[101] D. Zilio, C. Zuzarte, S. Lightstone, W. Ma, G. Lohman, R. Cochrane, H. Pirahesh, L. Colby, J. Gryz, E. Alton, and G. Valentin. Recommending materialized views and indexes with the ibm db2 design advisor. *Proceedings of the 1st International Conference on Autonomic Computing (ICAC'04)*, pages 180–187, 2004.

[102] E. Zitzler and L. Thiele. Multiobjective evolutionary algorithms : a comparative case study and the strength pareto approach. *Evolutionary Computation, IEEE Transactions on*, 3(4) :257–271, 1999.

Résumé

Les entrepôts de données permettent le stockage et la consolidation, en une seule localité, d'une quantité gigantesque d'information pour être interrogée par des requêtes décisionnelles complexes dites requêtes de jointures en étoiles. Afin d'optimiser ses requêtes, plusieurs travaux emploient des techniques d'optimisations comme les index de jointure binaires et la fragmentation horizontale durant la phase de conception physique d'un entrepôt de données. Cependant, ces travaux proposent des algorithmes *statiques* qui sélectionnent ces techniques de manière *isolée* et s'intéressent à l'optimisation d'*un seul objectif* à savoir les performances des requêtes. Notre principale contribution dans cette thèse est de proposer une nouvelle vision de sélection des techniques d'optimisation. Notre première contribution est une sélection incrémentale qui permet de mettre à jour de manière continuelle le schéma d'optimisation implémenté sur l'ED, ce qui assure l'optimisation continuelle des requêtes décisionnelles. Notre seconde contribution est une sélection incrémentale jointe qui combine deux techniques d'optimisation pour couvrir l'optimisation d'un maximum de requêtes et respecter au mieux les contraintes d'optimisation liées à chacune de ces techniques. A l'issu de ces propositions, nous avons constaté que la sélection incrémentale engendre un coût de maintenance de l'ED. Ainsi, notre troisième proposition est une formulation et résolution du problème multi-objectif de sélection des techniques d'optimisation où il faut optimiser deux objectifs : la performance des requêtes et le coût de maintenance de l'ED.

Mots-clés: Entrepôt de données, fragmentation horizontale, index de jointure binaire, sélection incrémentale, sélection jointe, sélection multi-objectif

Abstract

Data Warehouses store into a single location a huge amount of data. They are interrogated by complex decisional queries called star join queries. To optimize such queries, several works propose algorithms for selecting optimization techniques such as Binary Join Indexes and Horizontal Partitioning during the DW physical design. However, these works propose *static* algorithms, select optimization techniques in and *isolated* way and focus on optimizing a single objective which is the query performance. Our main contribution in this thesis is to propose a new vision of optimization techniques selection. Our first contribution is an incremental selection that updates continuously the optimization scheme implemented on the DW, to ensure the continual optimization of queries. To deal with queries complexity increase, our second contribution is a join incremental selection of two optimization techniques which covers the optimization of a maximum number of queries and respects the optimisation constraints. Finally, we note that the incremental selection generates a maintenance cost to update the optimization schemes. Thus, our third proposition is to formulate and resolve a multi-objective selection problem of optimization techniques where we have two objectives to optimize : queries performance and maintenance cost of the DW.

Keywords: Data Warehouse, horizontal partitioning, binary join indexes, combined selection, incremental selection, multi-objective selection