ENSMA : Ecole Nationale Supérieure de Mécanique et d'Aérotechnique
LIAS : Laboratoire d'Informatique et d'Automatique pour les Systèmes

THESE

pour l'obtention du Grade de

DOCTEUR DE L'ÉCOLE NATIONALE SUPÉRIEURE DE MÉCANIQUE ET D'AÉROTECHNIQUE

(Diplôme National — Arrêté du 7 août 2006)

Ecole Doctorale : Science et Ingénierie pour l'Information, Mathématiques
Secteur de Recherche : INFORMATIQUE ET APPLICATIONS

Présentée par :

Cheikh SALMI

****

Toward a Description and Modeling Inputs of Mathematical Cost Models Application: Data Warehousing

---

Vers une Description et une Modélisation des Entrées des Modèles de Coût Mathématiques : Applications aux Entrepôts de Données

****

Directeurs de Thèse : **Ladjel BELLATRECHE**
Co-directeur : **Jalil BOUKHOBZA**

Soutenue le 29 mars 2017
devant la Commission d'Examen

JURY

Rapporteurs :	**CLAUDE GODARD**	Professeur, LORIA, Université de Lorraine
	KOKOU YETONGNON	Professeur, LE2I, Université de Bourgogne
Examinateurs :	**DJAMEL BENSLIMANE**	Professeur, Université Lyon 1
	MOHAMED MEZGHICHE	Professeur, Université de Boumerdès, Algérie
	Ladjel BELLATRECHE	Professeur, ISAE-ENSMA, Poitiers, France
	Jalil BOUKHOBZA	Maitre de conférences, Université de Bretagne Occidentale, LabSTICC

Remerciements

Ce fut un plaisir pour moi de travailler avec tous les membres merveilleux dans le laboratoire LIAS , à l'école nationale supérieure de mécanique et d'aérotechnique (ENSMA) de Poitiers. Tout d'abord, je tiens à remercier Ladjel BELLATRECHE, Mon directeur de thèse et Jalil BOUKHOBZA mon co-directeur de thèse. Leurs idées et leurs soutiens extraordinaire ont eu une influence majeure sur cette thèse. Ils ont passé beaucoup de temps à m'aider ainsi que toutes les autres personnes dans notre laboratoire. Je remercie tous les doctorants du LIAS, Selma Khouri et Yassine Ouhamou pour leurs aides. Mes remerciements vont également aux directeurs du laboratoire : à Yamine AIT-AMEUR pour ses qualités humaines hors du commun, son écoute et son dévouement, et à Emmanuel GROLLEAU pour son aide et sa sympathie.

À tout le personnel du laboratoire, particulièrement Claudine RAULT et Stéphane JEAN, Mickael BARON et Frédéric CARREAU pour leur disponibilité, leurs aides et leur bonne humeur.

Je dédie cette thèse à : Mes chers défunts et regrettés ***père*** *et frère* ***Ibrahim****. Mon père, qui de son vivant, n'a jamais cesser de m'encourager dans mes études. Qu'il trouve ici le résultat de longues années de sacrifices et de privations pour m'aider à avancer dans la vie.*

Ibrahim: tu seras toujours , notre exemple de valeurs nobles, d'éducation, de courage, de persévérance, de patiente et de générosité.

Que dieux vous accueille dans son vaste paradis et que vous trouvez ici mon affectueux souvenir !

Á ma mère, qui a oeuvré pour ma réussite, de par son amour, son soutien, tous les sacrifices consentis et ses précieux conseils, pour toute son assistance et sa présence dans ma vie, reçois à travers ce travail aussi modeste soit-il, l'expression de mes sentiments et de mon éternelle gratitude.

Table des matières

Chapitre 1 Contexte de l'étude **1**

1 Introduction 3

2 Phase en amont 3

2.1 Bilan sur la phase en amont 9

3 Phase en aval 10

3.1 Choix de structures d'optimisation 12

3.2 Modes de sélection des structures d'optimisation 12

3.2.1 La sélection isolée des schémas d'optimisation 12

3.2.2 La sélection multiple des schémas d'optimisation 13

3.3 Développement des algorithmes de sélection 13

3.4 Validation de schémas d'optimisation 13

3.4.1 Mesure de la qualité des schémas sélectionnés 14

4 Bilan sur la phase en aval 17

5 Problématique et Contributions 17

6 Organisation de la thèse 19

Chapitre 2 Évolution dans le Monde des Bases de Données **21**

1 Introduction 23

2 Évolution des bases de données 25

2.1 Naissance des bases de données 25
2.2 Les premières générations de bases de données 26
2.3 Le modèle relationnel 26
2.4 Évolution des modèles conceptuels 32
2.5 Les approches de conception 32
3 Évolution de Plate-formes 34
4 Évolution des modèles de stockage des données 35
4.1 Stockage orienté ligne 36
4.2 Stockage orienté colonne 37
5 Évolution des supports de stockage 38
5.1 Les Bandes Magnétiques 39
5.2 Les disques durs 39
5.2.1 Les performances des HDD 40
5.3 Les disques à base de mémoire flash: SSD 41
5.3.1 Organisation des disques SSD 41
5.3.2 Opération d'entrée/sortie 42
5.3.3 La couche de traduction d'adresses FTL 44
5.3.4 Traduction d'adresse 44
5.3.5 Répartition de l'usure et ramasse-miettes 46
5.4 Les disques hybrides 47
5.4.1 Cloud Storage 47
6 Évolution des Requêtes 48
6.1 Optimisation Isolée de Requêtes 49
6.2 Optimisation Multiple de Requêtes 50
6.3 Charge de requêtes mixtes (OLTP/OLAP) 50
7 Évolution des Structures d'Optimisation 51
8 Les techniques d'optimisation 51

8.1 Les techniques d'optimisation matérielles 52
8.2 Les techniques d'optimisation logicielles 52
9 Conclusion 60

Chapitre 3 Description des Entrées des Modèles de Coût **63**

1 Introduction 65
2 Évolution des modèles de coût 66
3 Catégorisation des Paramètres de Modèles de Coût 69
3.1 Paramètres de l'entrepôt de données 70
3.1.1 Paramètres liés aux requêtes 71
3.2 Paramètres liés aux Structures d'Optimisation 73
3.3 Paramètres liés à l'architecture de déploiement 73
3.4 Paramètres liés aux supports de stockage 75
3.5 Paramètres liés au modèle de stockage physique 75
3.6 Une ontologie pour le support de stockage 75
3.7 Processus de construction d'ontologie 78
4 Méta modèle pour le modèle de coût 82
5 Instanciation du modèle de coût générique 85
6 Études de Cas 86
6.1 Importance de la gestion du cache et l'ordonnancement des requêtes . . 87
6.2 Formalisation du problème 88
6.3 Le modèle de coût utilisé 89
6.4 Résolution du problème 90
6.4.1 L'algorithme queen-bee 90
6.4.2 Descente des projections 91
7 Expérimentation 91
7.1 Résultats des simulations 91
7.2 Validation des résultats 91

8 Conclusion 93

Chapitre 4 Extension du cache du SGBD par la mémoire flash **95**

1 Introduction 97

2 État de l'art 98

2.1 Bilan 102

3 Vers un SGBD avec un système de stockage hybride 103

3.1 Le module d'apprentissage 105

3.1.1 Algorithmes de résolution de notre problème 106

3.2 Cache dynamique 106

3.3 Ordonnanceur heuristique des requêtes 110

4 Exécution des requêtes dans le cache hybride 112

4.1 Approche naïve 112

4.2 Approche basées sur l'éligibilité au cache 114

4.2.1 Coût d'exécution d'un noeud 114

4.2.2 Éligibilité d'un noeud pour le cache 114

4.2.3 Mise en cache des noeuds 115

4.3 Combinaison de l'approche d'éligibilité avec le mode statique 115

5 Étude expérimentale 116

5.1 Star Schema Benchmark 116

5.2 Charge de requêtes et environnement physique 117

5.3 Test des algorithmes 118

5.4 Validation sous Oracle 11g 120

6 Conclusion 120

Chapitre 5 Conclusion Générale et perspectives **123**

1 Conclusion 125

1.1 Identification des dimensions 126

1.2 Méta modélisation des dimensions 126
1.3 Une ontologie de supports de stockage 126
1.4 Hybridation du cache 127
2 Perspectives 128
2.1 Validation de l'ontologie 128
2.2 Une ontologie par dimension 128
2.3 Substitution de supports de stockage 129
2.4 Développement d'un outil pour les modèles de coût 129
2.5 Développement d'un suite logicielle pour la conception globale des entrepôts de données 129
2.6 Etude de passage à l'échelle et retour d'expérience des concepteurs . . 130
2.7 Vers un système persistant des modèles de coût 130

Annexe A Annexe : Charge de requêtes **131**

Annexes **131**

Annexe B Annexe : Ontologie des supports de stockages **135**

Table des figures **147**

Liste des tableaux **151**

Glossaire **153**

Chapitre 1

Contexte de l'étude

Sommaire

1 Introduction 3
2 Phase en amont 3
3 Phase en aval 10
4 Bilan sur la phase en aval 17
5 Problématique et Contributions 17
6 Organisation de la thèse 19

1 Introduction

Comme l'ont déjà annoncé plusieurs spécialistes, la troisième *guerre mondiale* sera celle de la donnée. Celui qui la *possède* et sait l'analyser en *temps voulu*, gagnerait la guerre. Les données sont partout, dans les sources de données, dans les réseaux sociaux, dans les entreprises locales et étendues, sous des formes hétérogènes, autonomes et évolutives. La course pour gagner cette guerre a motivé les organismes étatiques et privés à collecter ce déluge de données, les nettoyer et les stocker pour des fins d'analyses (dans le but d'augmenter leur compétitivité, leur stabilité, etc.). Selon une étude menée par the *Economist Intelligence Unit* en 2014, 68% des entreprises qui ont systématiquement recours à une analyse de données dans leurs prises de décision voient leurs bénéfices augmenter.

Les besoins d'analyse des entreprises peuvent être réalisés grâce à quatre éléments importants: **(i)** l'évolution technologique qui couvre les plateformes de déploiement (p. ex. les machines parallèles, les grilles de calcul, Cloud) dotées de supports de stockage avancés (p. ex., les mémoires flash) et des processeurs de calcul puissants (p. ex., les processeurs graphiques (GPU)). **(ii)** Les nouveaux paradigmes de programmation comme *Map-Reduce* et *Spark* [**?**]. **(iii)** L'ingénierie Dirigée par les Modèles (IDM) qui a mis à disposition des concepteurs et développeurs des outils, concepts et langages pour créer et transformer des modèles hétérogènes. L'IDM propose également l'utilisation systématique des méta-modèles, des modèles et des processus de tissage de conception suffisamment précis et formels, pour être interprétés ou transformés par des machines [**?**]. **(iv)** La maturité de la technologie des entrepôts de données (Figure 1.1), car elle a prouvé ses succès pour plusieurs entreprises importantes comme *Walmart* [**?**] et *Coca Cola* [1]. Rappelons qu'un entrepôt de données intègre les informations en provenance d'un nombre important de sources données *réparties*, *évolutives* et *hétérogènes* pour des besoins d'analyse de type OLAP (On-Line Analytical Processing ou Analyse en ligne).

Un entrepôt de données est *l'élément central* de n'importe quel système décisionnel, car il stocke les données. Sa construction et son exploitation est une tâche complexe. Elle se décompose en deux principales phases: une dite *en amont* comportant la construction de l'entrepôt et une autre dite *en aval* dédiée à son exploitation (Figure 1.2).

2 Phase en amont

Quatre étapes (tâches) principales caractérisent la phase de construction de l'entrepôt de données : **(1)** la définition du *schéma global* à partir des besoins fonctionnels identifiés par les décideurs, **(2)** la mise en correspondance entre les schémas locaux des sources et le schéma global de l'entrepôt de données, **(3)** la modélisation logique et **(4)** le processus ETL (Extraction, Transformation, Loading).

1. `http://www.mrweb.com/drno/news12772.htm`



FIGURE 1.1 – Les efforts d'établissement des correspondances entre les schémas locaux et global

1. **La première tâche** consiste à définir (ou construire) le schéma global de l'entrepôt de données en exploitant les besoins fonctionnels exprimés par les décideurs. Plusieurs formalismes sont utilisés pour définir le schéma, comme le modèle E/A (Entité Association) [**?**], le modèle UML, les modèles ontologiques (p. ex., OWL). Deux approches existent pour construire le schéma global d'un entrepôt de données: une approche ascendante (connue sous le nom Global as View (GaV)) dans laquelle les concepteurs utilisent les sources pour produire le schéma global. Dans la deuxième approche (Local as View (LaV)), on suppose l'existence d'un schéma global à partir duquel on décrit les sources. L'ajout d'une nouvelle source dans l'approche LaV est plus facile, car elle n'impacte pas le schéma global. En revanche, la construction des réponses à des requêtes est plus complexe comparée à une approche GaV qui consiste simplement à remplacer les prédicats du schéma global de la requête par leur définition [**?**].
2. **La deuxième tâche** consiste à définir les correspondances qui existent entre le schéma global et les schémas locaux. Ces correspondances dépendent de l'approche utilisée pour construire le schéma global (GaV, LaV). Établir ces correspondances demande un travail considérable [**?**]. Pour le simplifier, des efforts de description des sources de données ont été menés. Au lieu de travailler sur les instances des sources de données, le recours à leurs modèles est nécessaire pour rendre le processus d'élaboration des correspondances facile. Les premières solutions qui ont été définies dans le contexte de l'intégration des données ont considéré les schémas conceptuels des sources et de l'entrepôt de données. Ces efforts de description ont évolué avec la vulgarisation des ontologies de domaine et des bases de connaissances. Les concepteurs des entrepôts de données ont largement exploité leurs caractéristiques pour définir soit les schémas locaux ou le schéma global de l'entrepôt cible. Une ontologie est "*une spécification explicite et formelle d'une conceptualisation faisant l'objet d'un consensus*" [**?**]. Dans cette conceptualisation, le monde réel est appré-

hendé à travers des concepts représentés par des classes et des propriétés. Une ontologie conceptuelle regroupe ainsi les définitions d'un ensemble structuré de concepts. Ces définitions sont *traitables* par machine et *partagées* par les utilisateurs de tout système ou application utilisant une ou plusieurs ontologies. Elles doivent, en plus, être explicites, c'est-à-dire que toute la connaissance nécessaire à leur compréhension doit être fournie. Les ontologies, comme les modèles conceptuels, conceptualisent l'univers du discours au moyen de classes associées à des propriétés, et hiérarchisées par des relations de *subsomption*. Les ontologies contribuent à l'automatisation du processus de construction de l'entrepôt, car elles peuvent jouer le rôle d'un schéma global et contribuent à automatiser les correspondances en résolvant les conflits sémantiques et syntaxiques existants entre les sources [**?**].

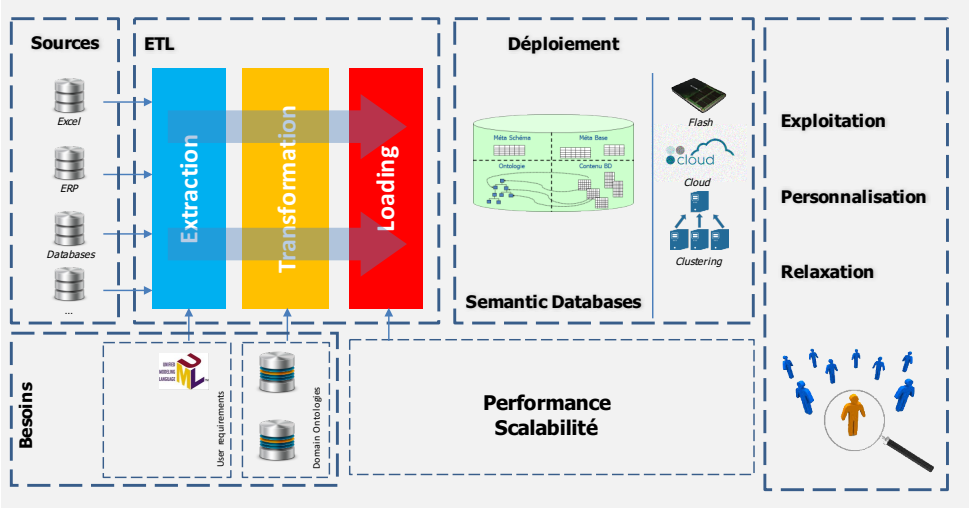


FIGURE 1.2 – Cycle de vie de conception d'un entrepôt de données

Les ontologies présentent cependant quatre différences par rapport aux modèles conceptuels. Ces différences sont à la base de la contribution que peuvent apporter les ontologies dans la problématique des bases de données en général, et les entrepôts de données en particulier [**?**].

— **Objectif de modélisation**. Une ontologie *décrit les concepts d'un domaine* d'un point de vue assez général, indépendamment de chaque application précise et de tout système dans lequel elle est susceptible d'être utilisée. À l'opposé, un modèle conceptuel de données *prescrit l'information* qui doit être représentée dans un système informatique particulier pour faire face à un besoin applicatif donné. Les éléments de la connaissance du domaine représentés par une ontologie et pertinents pour un système particulier pourront donc être extraits de l'ontologie par le concepteur du système sans que celui-ci ait besoin de les redécouvrir. Ainsi par exemple, la norme IEC 61360-4 (IEC 61360-4 1998) constitue un dictionnaire formel et consensuel de la plupart des catégories de *composants électroniques existants*, ainsi que leurs propriétés et les relations qui les relient. Ce domaine constitue donc une description de référence du

domaine des *composants électroniques* dont un concepteur d'un système de stockage (une base de données ou un entrepôt de données) peut aisément extraire, en interaction avec les utilisateurs, les entités, propriétés et relations devant être représentées au sein d'une base de données d'ingénierie d'une entreprise de conception électronique. Cette ontologie est décrite dans la Figure 1.3 (elle contient 190 classes 1026 propriétés).

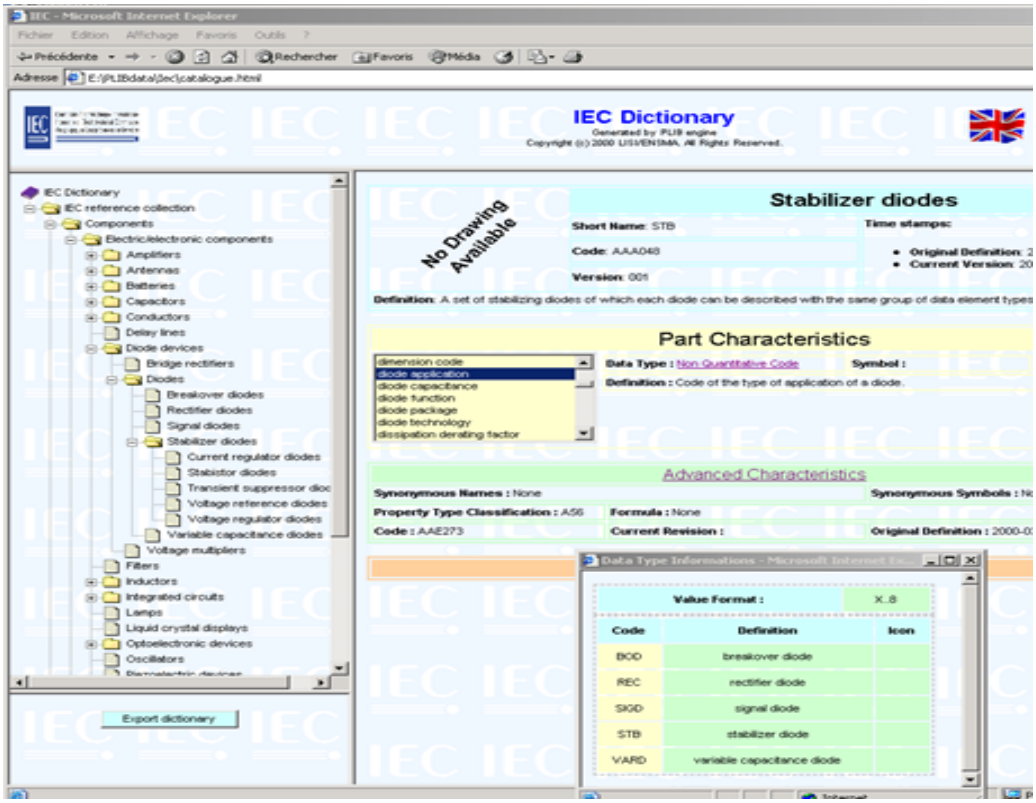


FIGURE 1.3 – Un exemple de l'ontologie IEC

— **Atomicité des concepts**. À la différence d'un modèle conceptuel de données, où chaque concept ne prend son sens que dans le contexte du modèle dans lequel il est défini, dans une ontologie, chaque concept est identifié et défini explicitement de façon individuelle et constitue une unité élémentaire de connaissance. Un modèle conceptuel de données peut donc extraire d'une ontologie seulement les concepts (classes et propriétés) pertinents pour son objectif applicatif. Il peut également, sous réserve de respecter leur sémantique (par exemple, subsomption), les organiser de façon assez différente de leur organisation dans l'ontologie, la référence au concept ontologique permettant de définir avec précision la signification de l'entité référençante.
— **Consensualité**. Une ontologie de domaine représente les concepts sous une forme consensuelle pour une communauté. L'exploitation de l'ontologie tant dans une phase de conception, par les concepteurs, qu'ultérieurement dans une phase d'exploitation, par un utilisateur, permet un accès naturel et ergonomique aux informations du do-

maine dès lors que le concepteur et l'utilisateur relèvent de la communauté visée par l'ontologie. De même, l'intégration sémantique de tous les systèmes basés sur une même ontologie pourra facilement être réalisée si les références à l'ontologie sont explicitées.

— **Non-canonicité des informations représentées**. A la différence des modèles conceptuels qui utilisent un langage minimal et non redondant pour décrire les informations d'un domaine, les ontologies utilisent, en plus des concepts primitifs, des concepts définis qui fournissent donc des accès alternatifs à la même information. La représentation de ces concepts non canoniques sous forme de vues sur la base de données permet d'étendre encore l'ergonomie d'accès à l'information.

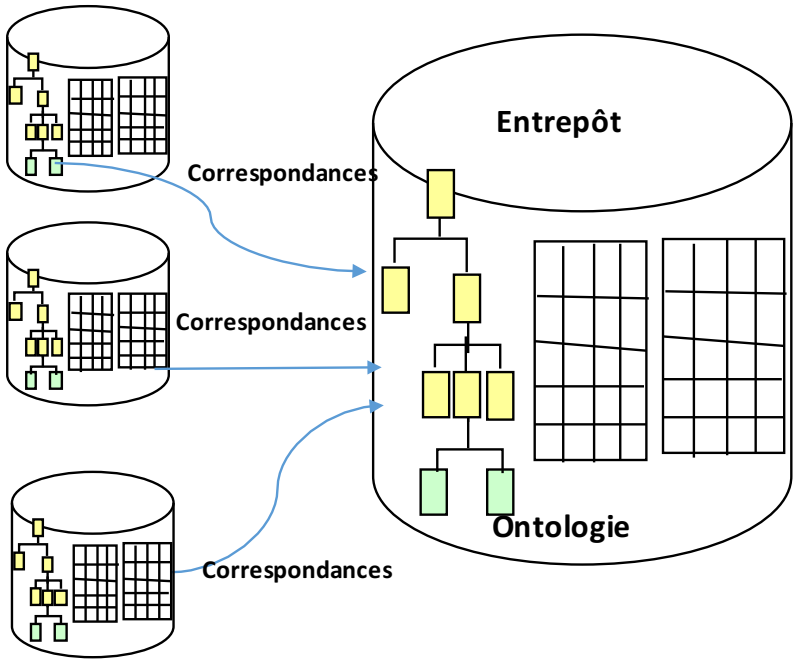


FIGURE 1.4 – Correspondances entre les ontologies locales et globale

Plusieurs approches de construction d'entrepôts de données à partir d'une ontologie de domaine existent [**?**]. Nous proposons de les classer en trois principales catégories selon l'utilisation de l'ontologie de domaine: (a) *les approches homogènes*, *les approches de type LaV sur l'ontologie de domaine*, et *les approches fédérées*. Dans la première catégorie, chaque source de données est définie comme un fragment de l'ontologie globale de l'entrepôt de données (cf. fragmentation dans les bases de données réparties homogènes [**?**]). Le traitement de requêtes dans une telle approche est bien maîtrisé dans le monde des bases de données repartie. Par contre les sources ont peu d'autonomie, car elles sont construites par des opérateurs de base de l'algèbre relationnelle définis sur les classes de l'ontologie de domaine (la projection et la sélection). Dans la deuxième catégorie, LaV sur ontologie, chaque source est représentée commune une vue sur l'ontologie globale. Cette architecture permet l'autonomie sémantique; mais le problème de réécri-

ture de requête reste posé [**?**]. Finalement, la troisième architecture suppose que chaque source possède sa propre ontologie, dite locale. Chaque ontologie locale est mise en relation avec l'ontologie du domaine (partagée). Cette catégorie offre plus d'autonomie aux sources et évite la réécriture de requêtes [**?**].

Un autre avantage considérable lié à l'utilisation des ontologies dans la construction des entrepôts de données est : **(i)** la définition des processus d'ETL au niveau ontologique. ETL correspond au processus d'extraction, transformation et chargement des données sources dans un entrepôt de données sémantique dont le schéma correspond à une partie de l'ontologie globale. Cela permet d'ignorer les spécificités d'implémentation de chaque source. **(ii)** La gestion de l'évolution des sources. Cela signifie que l'adjonction d'une nouvelle source est plus facile si les concepts et les propriétés de cette dernières sont déjà décrits dans l'ontologie de domaine. Dans ce cas, il faut juste adapter les correspondances entre les schémas.

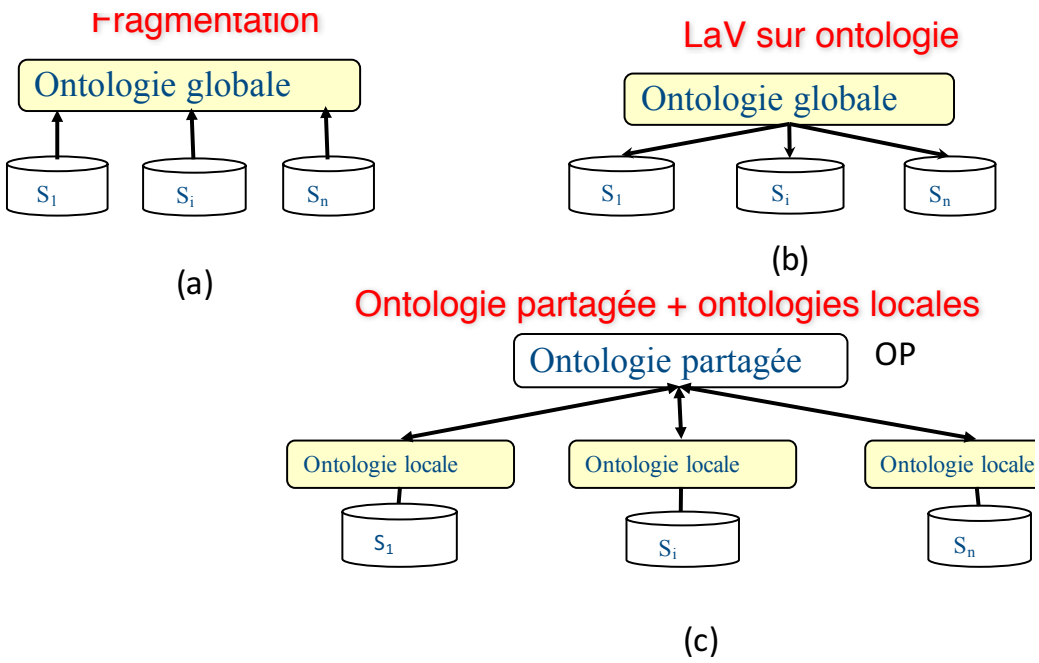


FIGURE 1.5 – Différentes architectures pour la modélisation conceptuelle d'un entrepôt de données

3. **La troisième étape** de la phase en amont caractérisant la construction de l'entrepôt de données est la phase logique consiste à concevoir le modèle logique cible de l'entrepôt de données répondant aux besoins des utilisateurs, tout en tenant compte des spécificités des données des sources. La conception du modèle logique passe par les tâches suivantes?:
   — la traduction du modèle conceptuel multidimensionnel en un modèle logique;
   — l'identification des sources de données candidates pour alimenter le schéma de l'entrepôt et pour préparer la prochaine étape ETL;
   — la sélection des sources candidates pour alimenter le modèle de l'entrepôt, enfin

— le mapping entre les données des sources et le schéma logique cible de l'entrepôt.

4. La phase d'ETL: comme son nom l'indique, cette tâche consiste à extraire les données à partir des sources sélectionnées et à effectuer les transformations nécessaires pour assurer le chargement des données des sources au niveau du schéma cible de l'entrepôt. Les données sont extraites à partir des sources de données hétérogènes. Elles sont ensuite propagées dans une zone de stockage temporaire: Data Staging Area (DSA), où auront lieu leur transformation, homogénéisation et nettoyage. Enfin, les données sont chargées dans l'entrepôt de données cible [**?**].

2.1 Bilan sur la phase en amont

Les efforts de description des sources de données ont largement contribuer à l'élaboration des correspondances entre les schémas locaux et global de l'entrepôt de données. La technologie d'IDM peut contribuer à établir ces correspondances. Un autre avantage de cette description est la gestion de *l'évolution des sources de données* en cas d'ajout. Si ces dernières référencent le schéma global, les concepteurs peuvent facilement définir les correspondances nécessaires comme l'indique la Figure 1.4. Il est important de rappeler que l'IDM a contribué au processus de construction d'un entrepôt de données [**?**]. La méta modélisation dans le contexte des entrepôts de données a été initiée en 2003 par Poole [**?**] en proposant une approche basée sur les différents paquetages du *Common Warehouse Metamodel* (CWM) [2]. *CWM* est un standard de l'OMG (Object Management Group) pour les techniques liées aux entrepôts de données. La particularité de ce standard est qu'il couvre le cycle de vie de construction d'un entrepôt de données. Il offre un méta modèle décrivant les méta-données aussi bien métiers que techniques que nous trouvons dans les tâches de construction d'un entrepôt de données. Les Métamodèles du CWM définissent l'interopérabilité entre les différentes tâches de la phase de construction d'un entrepôt de données. Dans [**?**, **?**], les auteurs proposent une démarche de méta modélisation des deux tâches de construction d'un entrepôt: conceptuelle et logique. Trois niveaux des modèles indépendants des plateformes (**PIM**) ont été proposés. Le premier niveau représente le modèle conceptuel (**PIM1**). Il permet de décrire la structure multidimensionnelle de l'entrepôt et de spécifier les traitements ETL associés. Le second niveau (**PIM2**) présente différentes alternatives de modèles logiques déduits automatiquement à partir du modèle conceptuel. Enfin, les différents modèles logiques sont traduits de manière automatique en un ensemble de modèles de plateformes d'implantation physiques (**PSMs**). Ces travaux ont essayé de couvrir en largeur les trois phases de conception d'entrepôt de données: conceptuelle, logique et physique, sans une étude en profondeur sur chaque phase.

Les ontologies offrent également un niveau de description plus élevé comme le montre la Figure 1.6. En conséquence, elles peuvent être combinées par des schémas globaux pour réduire l'hétérogénéité et raisonner. Ces intérêts ont poussé la communauté scientifique à développer

2. `http://www.omg.org/spec/CWM/`

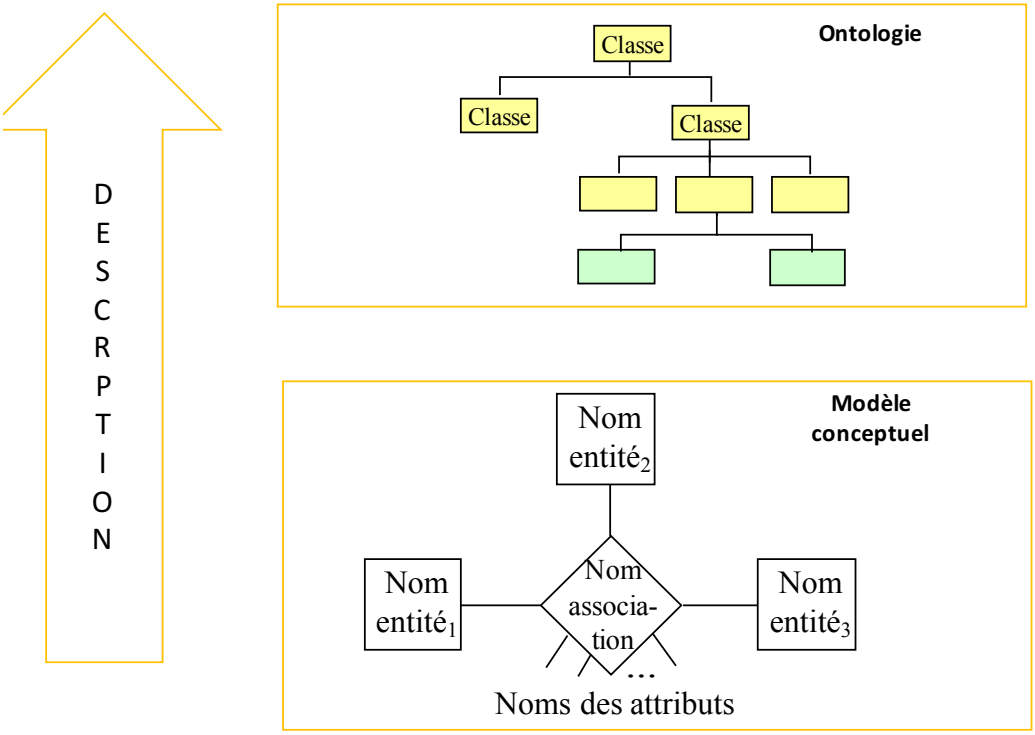


FIGURE 1.6 – L'évolution du niveau de la description

des bases de connaissances importantes, comme le cas de l'ontologie *YaGo* (*Yet Another Great Ontology*) décrivant plus de 2 millions d'entités, des personnes, des organisations, des villes et contenant plus de 20 millions de faits sur ces entités. Ses données proviennent de *Wikipédia* et sont structurées à l'aide de *WordNet*. De par sa taille et ses multi-thématiques, *YAGO* est difficile à utiliser directement pour des applications particulières. Pour faciliter son utilisation par des sources locales, des procédures d'extraction et de partitionnement ont été développés [**?**]. Si toutes les sources de données sont construites à partir d'une telle base de connaissances, leur intégration sera facile.

Cette expérience dans l'utilisation des schémas globaux et les ontologies peut être reproduite dans la phase en aval pour que les efforts de description soient généralisés à l'ensemble de phases de cycle de vie de conception des entrepôts de données.

3 Phase en aval

La phase en aval est souvent associée à l'image de qualité de l'entrepôt de données, car une fois construit, il doit être exploité d'une manière *efficace par les décideurs*. Cette efficacité dépend fortement des besoins non fonctionnels (p. ex., temps de réponse des requêtes, le coût de stockage, la consommation énergétique, etc.) fixés par les utilisateurs/décideurs. Notons que l'exploitation d'un entrepôt de données est similaire à celle d'une base de données

centralisée développée dans le cadre des applications OLTP (Online Transaction Processing), mais en même temps complexes vu les caractéristiques des requêtes OLAP et les exigences des décideurs sur le temps de réponse de requêtes. Cette situation complique les tâches des administrateurs des entrepôts de données.

Rappelons que dans les bases de données traditionnelles, la tâche d'un administrateur était principalement concentrée sur (a) la gestion des utilisateurs et (b) la sélection d'un nombre restreint de structures d'optimisation comme les index mono tables ou les différentes implémentations de l'opération de jointure (boucles imbriquées, tri fusion, hash, etc.). Dans les applications décisionnelles, la conception physique est devenue un enjeu important, comme l'indique Chaudhuri dans son article intitulé *Self-Tuning Database Systems: A Decade of Progress* qui a eu le prix de 10 Year Best Paper Award à la conférence VLDB'2007 : "*The first generation of relational execution engines were relatively simple, targeted at OLTP, making index selection less of a problem. The importance of physical design was amplified as query optimizers became sophisticated to cope with complex decision support queries.*".

Dans le contexte de l'optimisation des accès dans les entrepôts de données, les tâches des administrateurs ont été multipliées. En addition des tâches traditionnelles de nouvelles ont été ajoutées comme (Figure 1.7): (1) *le choix des structures d'optimisation*, (2) le *choix de leur mode de sélection*, (3) le *développement des algorithmes de sélection* et (4) la *validation et déploiement des solutions d'optimisation*.

Figure 1.7 récapitule l'ensemble des entrées et les tâches de cette phase.

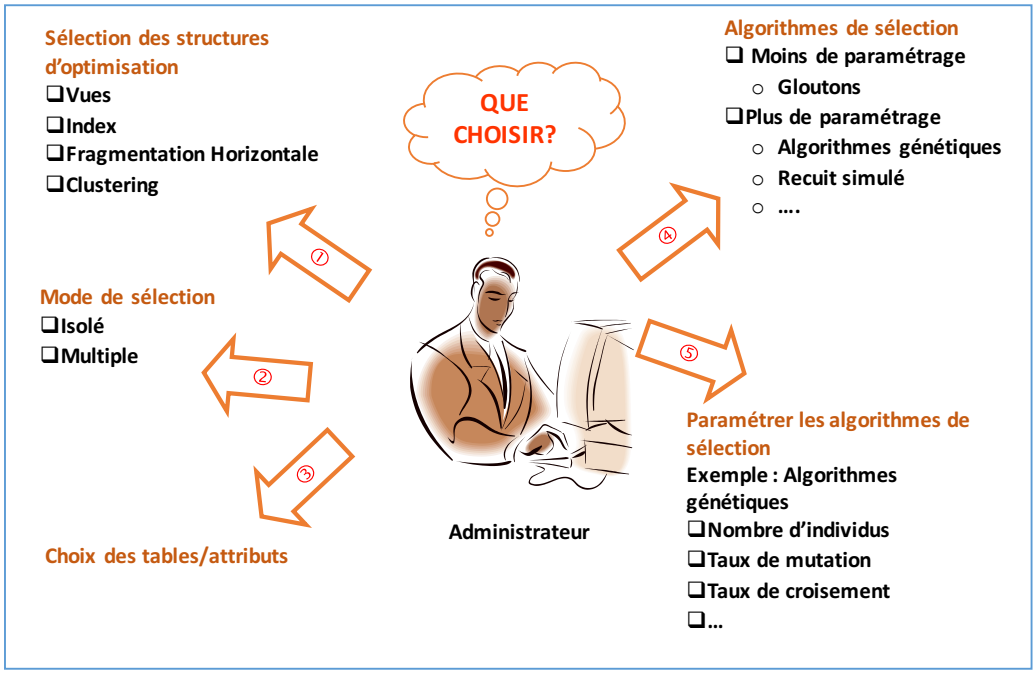


FIGURE 1.7 – L'évolution du niveau de la description

3.1 Choix de structures d'optimisation

Une large panoplie des structures d'optimisation a été proposée et la plupart supportée par SGBD commerciaux et académiques. Nous pouvons ainsi citer les *vues matérialisées*, les *index avancés*, la *fragmentation*, l'*allocation*, le *traitement parallèle*, le *groupement*, la *compression*, etc. Certaines structures sont l'héritage des bases de données traditionnelles ; comme la fragmentation, l'allocation, certains types d'index, le regroupement, etc. Notons que chaque structure a ses avantages, ses limites et certaines ont de fortes similarités. Ces structures peuvent être classées en deux catégories: [**?**, **?**] : les *structures redondantes* et les *structures non redondantes*. Les structures redondantes optimisent les requêtes, mais exigent des coûts de stockage et de maintenance. Les vues matérialisées [**?**], les index [**?**], la fragmentation verticale [3] [**?**, **?**] sont trois principaux exemples de cette catégorie. Les structures non redondantes ne nécessitent ni coût de stockage, ni coût de maintenance. Deux exemples de cette catégorie sont la fragmentation horizontale [**?**, **?**] et le traitement parallèle (dans le cas où l'allocation est faite sans réplication) [**?**].

3.2 Modes de sélection des structures d'optimisation

La sélection et la gestion des structures d'optimisation sont au coeur de la conception physique. La sélection d'une structure d'optimisation donne un schéma contenant une ou plusieurs instances de cette structure. Pour satisfaire sa charge de requêtes, l'administrateur peut choisir une ou plusieurs structures. Dans le cas où il choisit une seule structure, sa sélection est appelée une sélection isolée. Dans le cas où il choisit plusieurs structures, leur sélection est appelée sélection multiple.

3.2.1 La sélection isolée des schémas d'optimisation

Cette sélection a connu beaucoup d'intérêt de la part des communautés de bases et entrepôts de données [**?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**]. Elle peut concerner une structure redondante ou non redondante. La plupart des problèmes de sélection d'un schéma d'optimisation ont été reconnus comme des problèmes NP-Complet. Dans les bases de données traditionnelles, cette sélection a souvent concerné les index mono attributs [**?**], la fragmentation horizontale et verticale [**?**, **?**, **?**], l'allocation [**?**, **?**], etc. Dans le contexte des entrepôts de données, en plus des structures traditionnelles, elle concerne les vues matérialisées [**?**], les index binaires et multi tables [**?**], la compression des index binaires, etc.

La sélection isolée n'est pas toujours suffisante pour optimiser toute la charge de requêtes du fait que chaque structure d'optimisation est bien adaptée pour un profil particulier de requêtes, d'où le recours à la sélection multiple [**?**, **?**].

3. Dans la fragmentation verticale, la clé de la table fragmentée est dupliquée sur tous les fragments. Pour cela, elle est considérée comme une structure redondante

3.2.2 La sélection multiple des schémas d'optimisation

Comme la sélection isolée, elle peut concerner des structures redondantes, non redondantes ou les deux. Elle est plus complexe que la sélection isolée vu la complexité de son espace de recherche englobant ceux des structures concernées. En plus de cette complexité, une autre s'ajoute qui concerne la gestion des similarités (ou dépendances) entre certaines structures. Prenons l'exemple de deux structures d'optimisation qui sont les vues matérialisées et des index. Elles génèrent deux schémas d'optimisation redondants, partageant la même ressource qui est l'espace de stockage et nécessitent des mises à jour régulières. Une vue matérialisée relationnelle peut être indexée afin d'augmenter ses performances et vice versa. En conséquence, la présence d'un index peut rendre une vue matérialisée plus avantageuse. Cette similarité pourrait influencer la sélection de leurs schémas [**?**, **?**].

3.3 Développement des algorithmes de sélection

Le problème de sélection de structures d'optimisation est complexe, d'où la nécessité de développer des algorithmes offrant des solutions quasi optimales, en allant des algorithmes simples comme les gloutons aux algorithmes complexes comme les génétiques et le recuit simulé. Les algorithmes proposés doivent prendre en compte la ou les structures d'optimisation utilisées et leur mode de sélection.

3.4 Validation de schémas d'optimisation

Afin de quantifier la qualité des solutions proposées par chaque algorithme, la présence d'une métrique est nécessaire. Deux types de métrique sont possibles : (i) mathématique et (ii) validation réelle sur le système de base de données cible implémentant le système d'intégration. La validation réelle est difficile à mettre en place vue sa complexité, car elle exige que la base de données soit déployée et opérationnelle. La validation mathématique est souvent réalisée par le développement des modèles de coût simulant l'exécution de la charge de requêtes sur le SGBD cible déployé sur une plateforme donnée. Un modèle de coût estime le coût d'exécution d'une charge de requêtes. Cette estimation peut concerner le nombre d'entrées sorties nécessaires pour exécuter cette charge, le temps CPU, l'énergie consommée, etc. Notons que les optimiseurs de requêtes (Oracle, SQL Server, etc.) proposent des optimisations basées sur des modèles de coût (*Cost-based Optimisation*). Dans le cas, où le concepteur est satisfait de ses structures d'optimisation, il peut générer des scripts correspondant aux schémas d'optimisation qui peuvent attaquer le SGBD cible implémentant l'entrepôt de données.

Le développement de ces modèles de coût doit prendre en compte les caractéristiques logiques, physiques et matérielles du SGBD utilisé ainsi que la plateforme de déploiement.

Parmi les consommateurs des modèles de coût est la phase physique qui associée au pro-

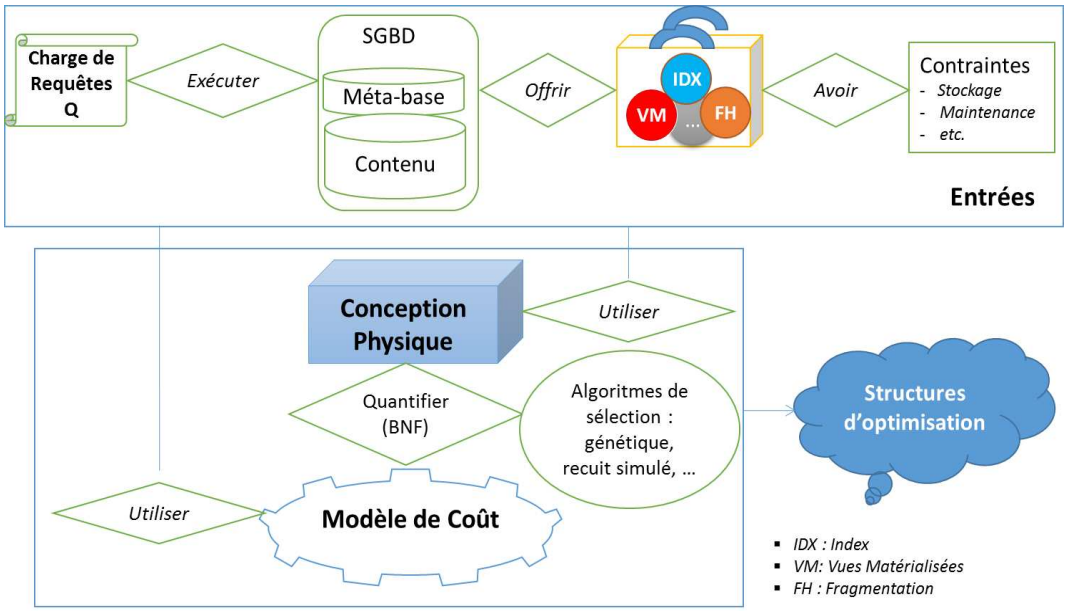


FIGURE 1.8 – Les éléments de la phase en Aval

blème de de la conception physique qui peut être défini comme suit :
Etant donnés:

- un entrepôt de données ($\mathcal{ED}$), stocké sur un support de stockage d'un Système de Gestion de Base Données $\mathcal{SGBD}_{\mathcal{ED}}$ et déployé sur une plateforme matérielle $\mathcal{P}_{\mathcal{ED}}$;
- une stratégie de gestion de buffer (support de stockage);
- une charge de requêtes $\mathcal{Q}$ définie sur cet entrepôt $\mathcal{ED}$;
- un ensemble de contraintes $\mathcal{ED}$ liées aux structures d'optimisation à sélectionner
- un ensemble de besoins non-fonctionnels $\mathcal{BNF}$ que la sélection des structures doit satisfaire;

L'objectif de la conception physique est de satisfaire l'ensemble de besoins non-fonctionnels tout en respectant les contraintes. Toute instance de ce problème est connue comme NP-complet [**?**]. Une instance est obtenue en considérant une variation une ou plusieurs entrées de ce problème. En analysant les travaux existants, nous trouvons un nombre important d'algorithmes qui traitent une instance du problème.

3.4.1 Mesure de la qualité des schémas sélectionnés

La qualité de la solution de tout algorithme résolvant une instance de ce problème est quantifiée par un modèle de coût mathématique. Vu son intérêt majeur, la communauté de base de données y a consacré des efforts considérables depuis les années 80. En faisant une recherche sur *googlescholar* et en tapant "*cost model database query*", nous avons trouvé environ 241 400 articles en 2016, ce qui montre l'intérêt de la communauté sur cette thématique. Les modèles de coût suivent l'évolution de bases de données à la fois logique, physique et matérielle.

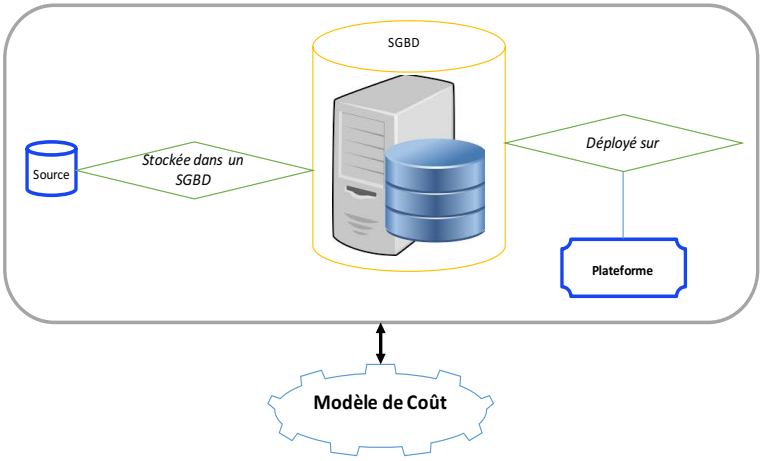


FIGURE 1.9 – Les paramètres d'un modèle de coût pour une seule instance de base de données

Les paramétrés utilisés par un modèle de coût donné peuvent appartenir à cinq classes décrivant les objets du monde base de données, à savoir: les données (le schéma de la base de données et ses instances), les requêtes, les structures d'optimisation (supportées par le SGBD cible), le support de stockage (disque, flash, etc.) et la plateforme de déploiement (centralisée, distribuée, etc.). Contrairement à la phase de construction de l'entrepôt de données, où des efforts considérables de description du domaine ont été déployés, la phase physique pêche énormément. Prenons l'exemple des modèles de coût qui représentent l'une des composantes principales de cette conception, ses paramètres ne sont pas structurés. Ils sont dans éparpillés dans un fichier. Une des conséquences de cette situation est la naissance d'une vision locale de développement de modèle de coût. Autrement dit, un modèle pour chaque base de données déployée (Figure 1.9). Cette vision a été largement développée au sein de notre laboratoire LIAS de l'ISAE-ENSMA (`www.lias-lab.fr`). Nous avons eu une grande expérience dans le développement des modèles de coût dans la plupart de générations de bases de données: traditionnelles [**?**], orientée objet [**?**], entrepôt de données centralisé [**?**] et sur cloud [**?**] et sémantique [**?**]. Souvent les paramètres des modèles de coût mathématiques ne sont pas structurés.

Les efforts de description du domaine de la conception physique permettra de généraliser les modèles de coût et les reproduire pour des nouvelles instances de bases de données. Pour ce faire des efforts de modélisation et *d'ontologisation* des paramètres des modèles de coût en classes pertinentes est nécessaire (Figure 1.11). Le domaine de la conception physique offre les possibilités de description, car ses entrées sont bien identifiées comme le montre la Figure 1.10.

Prenons l'exemple des mémoires flash qui sont considérées comme une solution alternative aux disques durs pour le stockage des données. Les deux supports partagent certains paramètres, mais avec des valeurs différentes. On peut prendre l'exemple du temps nécessaire pour écrire sur une page (sur disque/flash), ou le temps de lire une page (de disque/flash). Un effort de méta-modélisation contribuera à représenter ces paramètres une seule fois. Ces supports diffèrent du

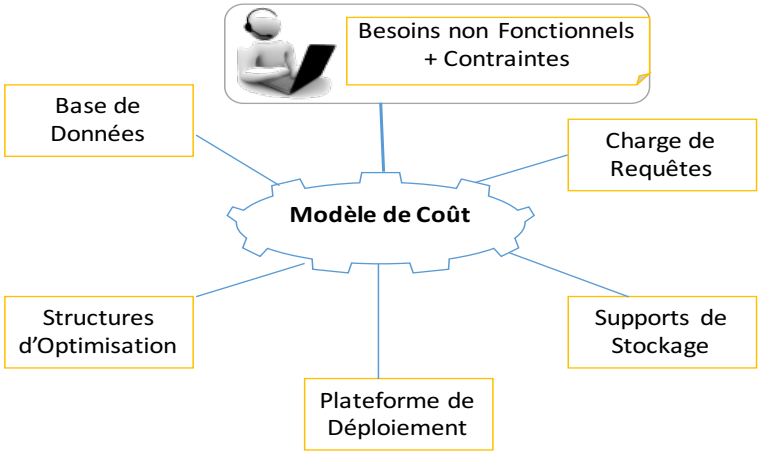


FIGURE 1.10 – Les dimensions d'un modèle de coût

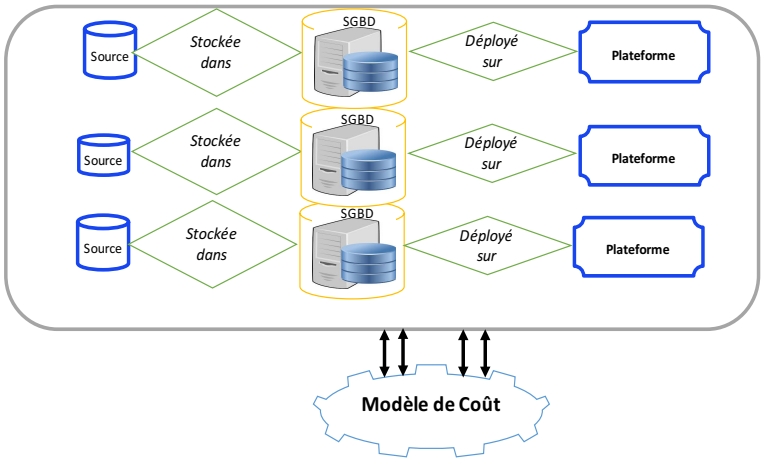


FIGURE 1.11 – Les paramètres d'un modèle de coût pour plusieurs instances de bases de données

leur fonctionnement. Les mémoires flash possèdent trois types d'opération bas niveau : la lecture ou l'écriture d'une page (typiquement 528, 2112 ou 4224 octets), et l'effacement d'un bloc (typiquement 32 ou 64 pages). De plus, des mécanismes de correction d'erreurs et de répartition de l'usure sont nécessaires pour améliorer la durée de vie de la mémoire. Pour que cette technologie soit interchangeable avec les disques durs, l'interface d'accès est restreinte aux opérations de lecture et d'écriture. Cette interface est fournie par une sur-couche logicielle fréquemment intégrée à la mémoire flash : la Flash Translation Layer (FTL), qui permet également de simplifier la gestion de la mémoire en prenant en charge les codes de correction d'erreurs, la répartition de l'usure et les effacements. La généralisation des modèles de coût permettra de dériver des nouveaux modèles mathématiques pour les nouveaux supports de stockage.

4 Bilan sur la phase en aval

Le débat que nous souhaitons lancer à travers cette thèse est le suivant : *est-il possible de bénéficier des efforts menés dans la phase en amont en termes d'utilisation massive des ontologies et de la méta modélisation à la phase en aval pour la résolution de problème de la conception physique et la définition des modèles de coût?*.

Rappelons que ces efforts ont été menés dans la phase de construction d'un entrepôt de données en utilisant le standard *CWM*. En hiérarchisant les niveaux d'abstraction des modèles, et l'utilisation des techniques de méta modélisation augmente la productivité en maximisant la compatibilité entre les systèmes. Dans cette thèse, nous proposons une utilisation conjointe de la méta modélisation et les ontologies pour la phase physique. Cette dernière est vue comme un tunnel de toutes les étapes de cycle de vie de conception d'entrepôts de données. Les efforts de modélisation concernent les entrées d'un modèle de coût, et l'ontologie représente dans notre cas les supports de stockage.

5 Problématique et Contributions

La phase de construction d'entrepôts de données a vu un développement considérable de techniques de modélisation/méta-modélisation et d'ontologies. Cela est du à la maîtrise des différentes tâches de cette phase. Dans le monde informatique, toute personne a déjà construit un modèle entité association ou un modèle de classes UML. Cette maîtrise concerne également les modèles logiques de bases de données. Nous pouvons facilement générer un méta modèle décrivant une base de données relationnelle d'une manière générique. Par contre la phase physique est plus complexe, car elle nécessite des connaissances approfondies sur l'ensemble d'entités du monde de base de données et elle utilise des modèles de coût mathématiques pour quantifier les solutions qu'elle propose. Rappelons les efforts considérables des éditeurs commerciaux et académiques pour développer des outils (connus sous le nom d'advisor) pour assister les administrateurs dans leurs tâches en leur recommandant des structures d'optimisation pour une charge de requêtes donnée. Nous pouvons citer par exemple: **Design Advisor pour IBM DB2** [**?**] qui recommande quatre structures d'optimisations (les indexes, la fragmentation horizontale, les vues matérialisées et le groupement de données). Quant à **Database Tuning Advisor pour Microsoft SQL Server** [**?**], il recommande trois structures d'optimisation (indexes, fragmentation horizontale et les vues matérialisées). **SQL Access Advisor pour Oracle** [**?**] qui permet de donner des recommandations les vues matérialisées les index (binaires et B-arbre). Mais ces outils n'offrent pas la possibilité de visualiser les modèles de coût développés. Pour les concepteurs, ces derniers sont des boîtes noires. Certains SGBD open source comme PostgeSQL offrent aux concepteurs la possibilité de voir les modèles de coût utilisés. Les paramètres et les formules de ces derniers sont *éparpillés* dans le code source souvent développé en langage C; ce qui rend leur exploitation difficile.

Les objectifs de ce travail sont:

1. Expliciter et catégoriser l'ensemble de paramètres utilisés dans la phase physique. Le fait que cette phase arrive à la fin de cycle de vie, elle nécessite des informations liées à toutes les autres phases: les besoins non fonctionnels, la phase conceptuelle, la phase logique, la phase de déploiement. Cette explicitation doit passer par une analyse approfondie de toutes les phases de cycle de vie de conception des entrepôts de données.
2. Rappelons que la quantification des solutions proposées par cette phase passe par des modèles de coût mathématiques qui peuvent être vus comme une fonction mathématique ayant des paramètres que nous avons ci-dessous cités. Afin de faciliter leur développement et leur utilisation, des efforts de méta-modélisation sont nécessaires. Un autre point concerne la similarité qui peut exister entre les différentes entrées de cette phase. Nous pouvons citer l'exemple des supports de stockage qui sont hiérarchiques (flash → disque dur → bande magnétique). Cette hiérarchisation permet d'augmenter les mécanismes de spécialisation/généralisation entre les supports. La majorité des modèles de coût existants concerne des bases de données avec une architecture traditionnelle mémoire-disque dur. Due à la similarité des disques durs et les mémoires flash, les efforts considérables qui ont été déployés pour développer des modèles de coût pour l'architecture traditionnelle peuvent être réutiliser pour le cas de l'archiecture (mémoire flash-disque dur).
3. Traiter des requêtes ayant une forte interaction comme dans le cas des entrepôts de données relationnels nécessite des optimisations avancées pour réduire leur coût d'exécution. Traditionnellement leur optimisation passe par le processus de matérialisation (sur le disque dur) de leurs noeuds communs (représentant leurs résultats intermédiaires). Vu les similarités entre les disques et les mémoires flashs, ces dernières peuvent combiner avec le cache du SGBD. Cela permettra de stocker un nombre important de noeuds au niveau du cache pour faire profiter d'autres requêtes de ces noeuds. Pour réussir cette association, le développement d'un système de décision gérant les deux mémoires à l'aide de différents modèles de coût et des politiques de gestion de ces caches.

Nos contributions à travers ce travail, sont quadruples. Premièrement, nous avons établi un état des lieux de deux phénomènes en bases de données: la *diversité* qui a touché toutes les phases de conception et d'exploitation des bases de données et l'*évolution logicielle et matérielle*. Cette situation nous a poussé à méta modéliser les différentes entrées de la phase couche physique. Une ontologie de domaine des supports de stockage a également été construite. Cette construction a été motivée par la présence de fortes similarités entre les supports de stockage. Ce travail a donné lieu à la publication suivante:

— *Ladjel Bellatreche, Salmi Cheikh, Sebastian Breß, Amira Kerkad, Ahcène Boukorca, Jalil Boukhobza: How to exploit the device diversity and database interaction to propose a generic cost model? IDEAS 2013: 142-147*

La deuxième contribution consiste à simplifier le processus de développement de modèles de coût mathématiques. Cette simplification est due à la présence de tous les méta modèles

et l'ontologie des entrées. Un concepteur de modèle de coût peut juste piocher les paramètres qui répondent à ses besoins non fonctionnels. Cette contribution a donné lieu à la publication suivante :

— *Ladjel Bellatreche, Sebastian Breß, Amira Kerkad, Ahcène Boukorca, Cheikh Salmi, The Generalized Physical Design Problem in Data Warehousing Environment: Towards a Generic Cost Model, Proceedings of the 31th IEEE International Convention MIPRO (Mipro2013), edited by IEEE, 2013*

Notre troisième contribution porte sur le processus d'instanciation de nos méta modèles pour répondre à des problèmes traditionnels de la phase physique comme le problème de la gestion de cache et le problème de l'ordonnancent de requêtes décisionnelles.

Finalement, notre quatrième contribution concerne l'extension du cache traditionnel par une mémoire flash afin d'augmenter sa capacité pour qu'il puisse stocker un nombre important de résultats intermédiaires des requêtes. Cette hybridation est motivée par les similarités entre les supports de stockage. Cette contribution a donné lieu à la publication suivante :

— *Cheikh Salmi, Abdelhakim Nacef, Ladjel Bellatreche, Jalil Boukhobza: What can Emerging Hardware do for your DBMS Buffer? ACM DOLAP 2014: 91-94*

6 Organisation de la thèse

Cette thèse est organisée en quatre chapitres et une annexe, comme illustré par la Figure 1.12.

Le **deuxième chapitre** présente un état des lieux sur les différentes évolutions dans le monde des bases de données. Ces évolutions concernent six dimensions: (1) les bases de données, (2) les reqûetes, (3) les supports de stockage, (4) les modèles de stockage, (5) les structures d'optimisation et (6) les plateformes de déploiement. Une analyse approfondie de chaque dimension est menée permettant d'identifier les paramètres pertinents caractérisant chaque dimension. Vu la diversité des dimensions, des efforts de synthèses importants sont effectués pour rendre la lecture de ce manuscrit plus agréable.

Le **troisième chapitre** présente notre première et deuxième contributions. La première concerne la méta modélisation des l'ensemble des entrées des modèles de coût utilisés dans la phase physique des entrepôts de données. Cette contribution est le fruit de l'analyse approfondie effectuée dans le Chapitre 2. Nous nous sommes inspirés des travaux de description et de méta modélisation établis dans la phase de construction des entrepôts de données. Cette modélisation nous offre une vue globale et fragmentée en même temps sur l'ensemble des paramètres utilisés par les modèles de coût mathématiques qui représente le noyau de la phase de conception physique. La deuxième contribution concerne la définition d'une ontologie de domaine dédiée aux supports de stockage, en exploitant les similarités entre les différents supports de stockage utilisés dans le contexte des bases de données. Cette ontologie permet d'expliciter les différentes propriétés. En conséquence, ils peuvent être partagés par les développeurs et les concepteurs.

Deux exemples d'instanciation de nos modèles sont également présentés. La nouvelle vision de modèles de coût est utilisée dans le contexte de problème d'ordonnancement de requêtes.

Le **quatrième chapitre** est consacré l'hybridation des supports de stockage traditionnels et avancés (les mémoires flashs) pour augmenter la capacité de stockage du cache. Cette hybridation est motivée par la création de notre ontologie de supports de stockage. Cela permettra de cacher un nombre considérable de résultats intermédiaires de requêtes décisionnelles connues par leur forte interaction (elles partagent des opérations). Cette hybridation est utilisée dans le cadre du problème d'ordonnancement de requêtes combiné au problème du cache. Des résultats expérimentaux sont conduits pour évaluer la qualité et la faisabilité de notre approche.

Enfin, dans le **dernier chapitre**, nous présentons les conclusions générales de cette thèse ainsi que les diverses perspectives.

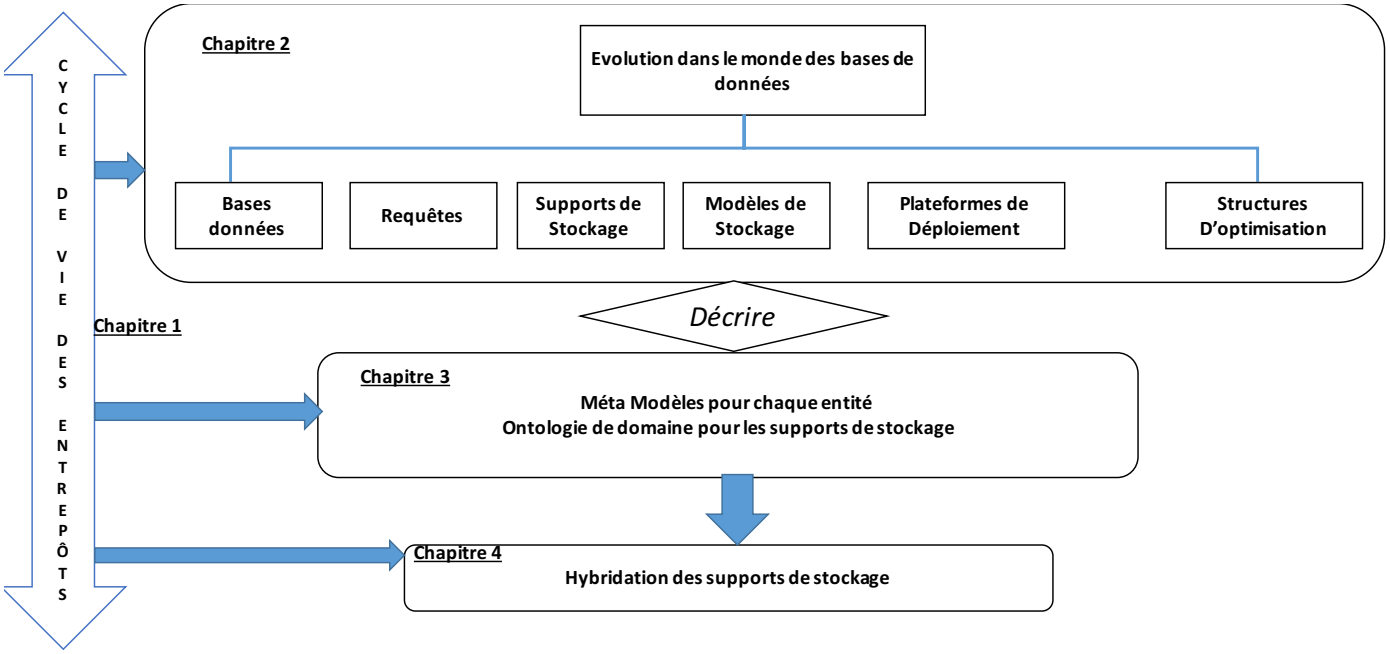


FIGURE 1.12 – Organisation de la thèse

Chapitre 2

Évolution dans le Monde des Bases de Données

Sommaire

1	Introduction	23
2	Évolution des bases de données	25
3	Évolution de Plate-formes	34
4	Évolution des modèles de stockage des données	35
5	Évolution des supports de stockage	38
6	Évolution des Requêtes	48
7	Évolution des Structures d'Optimisation	51
8	Les techniques d'optimisation	51
9	Conclusion	60

1 Introduction

La technologie des bases de données a subi des évolutions spectaculaires durant les deux dernières décennies. Ces évolutions concernent des aspects méthodologiques (p. ex., le cycle de vie de conception), logiciels (p. ex., modèles de représentation et de description des données, langages de manipulation des données, etc.) et matériels (p. ex., structures d'optimisation, systèmes de stockage de données (p. ex. flash), et les plateformes de déploiement, le matériel émargeant, etc.).

En ce qui concerne l'évolution méthodologique, les entrepôts de données ont révolutionné le cycle de vie classique de conception de bases de données traditionnelles. Ce dernier comporte cinq phases principales: (1) l'analyse des besoins, (2) la modélisation conceptuelle, (3) la modélisation logique, (4) la phase de déploiement et (5) la modélisation physique. La Table 2.1 présente brièvement les différents modèles, formalismes et structures d'optimisation utilisés dans les phases de cycle de vie de conception des bases de données traditionnelles.

Phases	**Formalismes et Structures d'Optimisation**
Analyse de besoins	cas d'utilisation d'UML
Phase conceptuelle	Entité-Association, UML
Phase logique	Modèle Relationnel, Orienté-Objet, Objet Relationnel
Phase de déploiement	plateforme centralisée avec des matériels traditionnels
Phase physique	Index

TABLE 2.1 – Les formalismes et structures d'optimisation utilisés dans la conception des bases de données traditionnelles

Le cycle traditionnel a subi des évolutions intra phases et ajout de nouvelles phases pour répondre aux besoins d'intégration, de consolidation et de performance exigées par les entrepôts de données. La phase ETL (Extraction, Transformation, Chargement) qui permet d'extraire des données des sources, les nettoyer, les transformer et enfin de les charger dans l'entrepôt de données a été ajoutée au cycle traditionnel vu son importance, plusieurs outils commerciaux (e.g., IBM Information Server InfoSphere, SAS Data Integration Studio Oracle Warehouse Builder (OWB), Sap BusinessObjects Data Integration, etc.) et open source (e.g., Talend Open Studio, Pentaho Data Integration (ex Kettle), Clover ETL, etc.) ont été développés. Une autre évolution concerne l'amplification de la phase physique [**?**] avec l'extension de plusieurs structures d'optimisation qui n'existaient pas dans les bases de données traditionnelles comme les vues matérialisées, les index de jointure binaires, la compression, etc.

La phase de déploiement a également subi une évolution spectaculaire avec l'apparition des nouvelles plateformes robustes et scalables pour stocker de grandes masses de données (les grappes de machines, les machines parallèles, le Cloud, etc.). Ce progrès technologique a contribué à la baisse du coût de stockage. En 2004, le stockage de 1 Go était inférieur à 1$ (Figure 2.1). Une autre évolution concerne le matériel émergeant (emerging hardware), qui a contribué au passage des disques durs traditionnels aux SSD et du CPU au GPU comme le

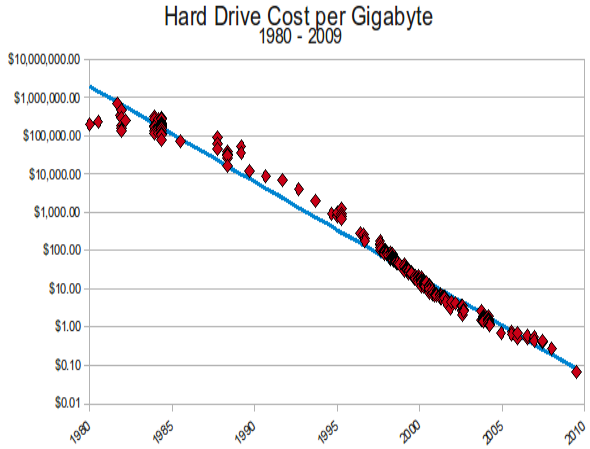


FIGURE 2.1 – Evolution du prix de stockage

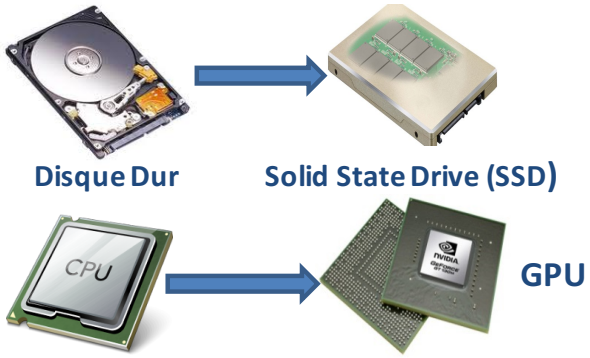


FIGURE 2.2 – Matériel émargeant

montre la Figure 2.2.

La combinaison des progrès logiciels et matériels peut être mise à profit pour le processus d'optimisation de traitement de requêtes complexes des entrepôts de données tout en prenant en compte leur hétérogénéité. Dans ce chapitre, nous présentons les différentes évolutions qui concernent les données, les systèmes de stockage, les plate-formes de déploiement et le matériel émergeant.

Les évolutions importantes dans le nouveau cycle de vie de conception des bases de données avancées sont décrites dans la Table 2.2. Elles sont soulignées pour les distinguer.

Dans les sections suivantes, nous détaillons l'ensemble des évolutions que nous avons citées. Un intérêt particulier est donné aux entrepôts de données qui représentent un exemple des bases de données avancées.

Phases	Formalismes et Structures d'Optimisation
Analyse de besoins	Réseaux de Pétri, modèles de buts
Phase conceptuelle	Entité-Association, UML, Entité-association OLAP
Phase logique	Modèle Relationnel, Orienté-Objet, Objet Relationnel schéma en étoile, flocon de neige
Phase de déploiement	plateformes parallèles avec des matériels émergeants
Phase physique	Index, vues matérialisées, le partitionnement, l'ordonnancement de requêtes, etc.

TABLE 2.2 – Les formalismes et structures d'optimisation utilisés dans la conception des bases de données traditionnelles

2 Évolution des bases de données

L'histoire des bases de données a commencé par le stockage des données sur des fichiers plats. Les fichiers furent le premier moyen pour l'enregistrement de données sur un support de stockage non volatile. Ce sont des collections de données structurées ou non. Ce modèle de stockage nécessite un programme utilisateur pour définir la structure du fichier, gérer les données et coder l'équivalent des requêtes $\mathcal{SQL}$. Ce modèle de représentation présente les inconvénients suivants :

— La lourdeur d'accès aux données : un programme est toujours nécessaire pour accéder aux données. Aussi, si la structure du fichier change, il faut aussi modifier le programme.
— l'accès non optimisé aux données : le temps de réponse à une requête dépend des compétences du programmeur.
— les données non partagées : chaque programme maintient ses propres données (un programme ignore les données qui lui sont utiles lorsqu'elles sont manipulées par un autre programme)
— la redondance des données : les mêmes données peuvent être conservées par différents programmes, ce qui entraîne une perte d'espace de stockage.
— le manque de sécurité : si les fichiers de données sont accédés d'une manière concurrente par plusieurs utilisateurs, il est difficile de garantir l'intégrité des données.

2.1 Naissance des bases de données

Les fichiers étaient stockés sur des bandes magnétiques caractérisées par un accès séquentiel aux données. La lecture d'un petit fragment de données nécessite le parcours intégral des fichiers. Quelques années plus tard, la technologie de stockage sur les disques à accès direct est devenue mature. Ceci a largement contribué à la genèse des bases de données. Une base de données étant définie comme une représentation unique des informations indépendamment de toute application ou programme.

2.2 Les premières générations de bases de données

Les trois niveaux de conception définis pour concevoir toute base de données sont : le niveau conceptuel, logique et physique. L'évolution des bases de données qui a permis l'établissement de ce cycle s'est faite de manière verticale où les premiers modèles de données apparus ont été des modèles physiques, suivi des modèles logique (le relationnel étant le modèle le plus connu), ensuite des modèles conceptuels. Nous nous concentrons dans ce qui suit sur les modèles implémentés au sein des bases de données, à savoir les modèles logiques et physiques.

Suite à la définition des critères que doit satisfaire un $\mathcal{SGBD}$ par le CODASYL (Conference on Data Systems Languages), le premier modèle de base de données baptisé modèle hiérarchique, a vu le jour. Il modélise et organise les données de façon hiérarchique sous la forme d'un ensemble d'arbres. Tout enregistrement possède un et un seul enregistrement propriétaire ou racine. L'accès aux données dans une base de données hiérarchique se fait par des programmes de bas niveau qui permettent le déplacement dans la base en utilisant un langage de navigation. Cette navigation est faite à partir des données racines vers les données souhaitées [**?**] . L'inconvénient de ce modèle est la difficulté de représenter les données non hiérarchiques comme les enregistrements ayant plusieurs racines. Il faut ajouter à cela le fait que le programmeur doit connaître la représentation physique de la base de données. Le modèle réseau est apparu pour pallier les lacunes des données hiérarchiques et supporter les enregistrements plusieurs à plusieurs.

Une base de données dans ce modèle est représentée sous forme d'un graphe orienté. Les données sont représentées sous forme d'ensembles appelés *set*. Chaque set possède plusieurs entrées où chaque entrée spécifie les tuples qui sont en relation avec le tuple de cette entrée [**?**, **?**]. L'inconvénient de ce modèle est la difficulté d'accès aux données; en effet le programmeur est obligés de naviguer le long du chemins d'accès pour atteindre une donnée cible, ayant comme conséquence la difficulté de maintenance de la base et des programmes. Un autre inconvénient, comme pour le modèle hiérarchique, réside aussi dans la connaissance des détails de représentation physique de la base de données. Le concept de "modèle de donnée" a été introduit avec l'apparition de ces deux modèles, même si le terme modèle de données a été introduit et défini plus tard par Edgar Codd pour son modèle relationnel.

2.3 Le modèle relationnel

Le modèle relationnel proposé par Codd E. F. [**?**] est un modèle simple et basé sur une théorie mathématique rigoureuse. Les données sont représentées sous forme de tables. Une table est une relation au sens mathématique, i.e. un sous-ensemble du produit cartésien de plusieurs domaines. Une base de données est un ensemble de tables reliées par des attributs en commun. Ce modèle a une partie structurelle et aussi une partie manipulation, qui a donné naissance à $\mathcal{SQL}$, un langage de base de données descriptif, facile et qui peut être utilisé même par des non-informaticiens. Plusieurs types de bases de données ont été développées autour de ce modèle qui

a été adopté par la plupart des éditeurs de $\mathcal{SGBD}$ commerciaux. Le modèle relationnel continue à jouer un rôle principal dans le monde de stockage de l'information grâce à sa simplicité et son adéquation à différents cas d'utilisation. Le modèle relationnel peut être implémenté dans plusieurs types de bases de données, que nous pouvons classer selon les catégories suivantes:

Bases de données transactionnelles

Les bases de données transactionnelles ou en lignes (Online Transaction Processing) sont utilisées par les entreprises dans leurs tâches journalières (*day-to-day applications*). Elles se caractérisent par un haut débit transactionnel et un taux d'insertion et de mise à jour important. Leurs données sont dynamiques et accédées concurremment par plusieurs utilisateurs. Ce type de bases de données nécessite une disponibilité totale, ce qui implique l'implémentation de mécanisme fiables pour les reprises après les pannes.

Bases de données décisionnelles

Ces bases de données connues sous le nom d'entrepôts de données ($\mathcal{ED}$) sont utilisées pour la prise de décision en termes de stratégie de l'organisation et la génération de rapports d'activités. Les données de ces bases sont recueillies, nettoyées et intégrées à partir de bases de données transactionnelles ou tout autre type de source.

Les entrepôts de données sont devenus depuis les années 90 un moyen stratégique de l'entreprise et une composante incontournable pour la prise de décision dans les entreprises et les organisations.

Définition 1

Un entrepôt de données tel que le définit Bill Inmon [?] est une collection de données orientées sujet, non volatiles, intégrées, historisées et utilisées pour supporter un processus d'aide à la décision.

Définition 2

Selon Ralph Kimball [?], un entrepôt de données doit être conçu pour être compréhensible (par l'analyseur). Dans ce but, Kimball a proposé la méthodologie de modélisation dimensionnelle (Dimentional Modeling ou Kimball Modeling en anglais), qui est devenue par la suite un standard dans le domaine décisionnel [?].

Nous établissons une comparaison entre une base de données classique et un entrepôt de données comme le montre le tableau 2.3. Cette comparaison révèle les aspects analytiques et les besoins en performance liés aux entrepôts.

Selon Ralph Kimball [?], la représentation multidimensionnelle est la plus adaptée pour le processus d'analyse. Un tel modèle permet d'analyser les données sous plusieurs axes, et d'y

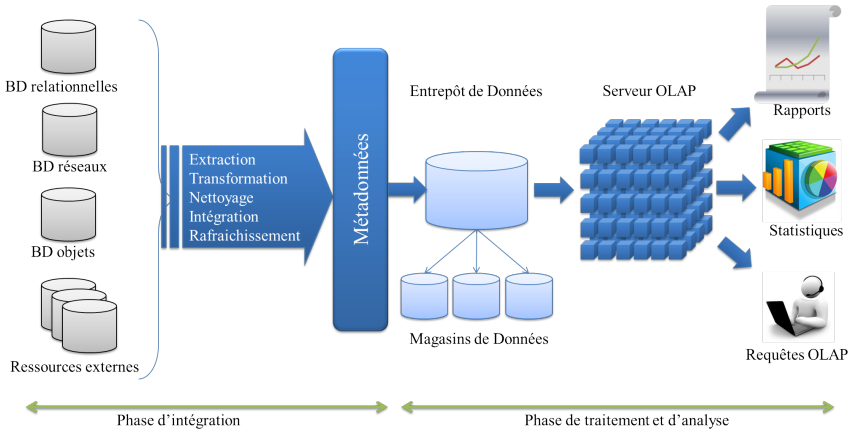


FIGURE 2.3 – Architecture d'un entrepôt de données

Caractéristiques	Base de Données	Entrepôt de Données
Opération	Gestion courante	Analyse et aide à la décision
Modèle	Souvent Entité-Relation	Schéma en étoile, flocon de neige ou en constellation
Normalisation	Fréquente	Rare
Données	Courantes, brutes	Historiques, agrégées
Mise à jour	Immédiate	Différée
Perception	Bidimensionnelle	Multidimensionnelle
Requêtes	Lecture et écriture	Lecture et rafraîchissement
Volume	Des Giga Octets	Jusqu'à quelques Zéta-octets

TABLE 2.3 – Comparaison entre les bases de données classiques et les entrepôts de données

Temps / Ville / Produit	Trimestre1			Trimestre2			Trimestre3			Trimestre4			Total		
	Po	Pa	L	Po	Pa	L	Po	Pa	L	Po	Pa	L	Po	Pa	L
Tablette	22	68	60	18	72	34	22	62	35	33	32	62	95	234	191
Smart phone	88	120	90	80	120	90	56	175	59	81	57	39	305	472	278
Clavier	6	49	30	16	42	20	34	43	24	18	16	59	74	130	133
Total	116	237	180	114	234	144	112	280	118	132	105	160	474	836	602

Po: Poitiers; Pa: Paris; L: Lyon

FIGURE 2.4 – Représentation MOLAP par un tableau Multidimensionnel

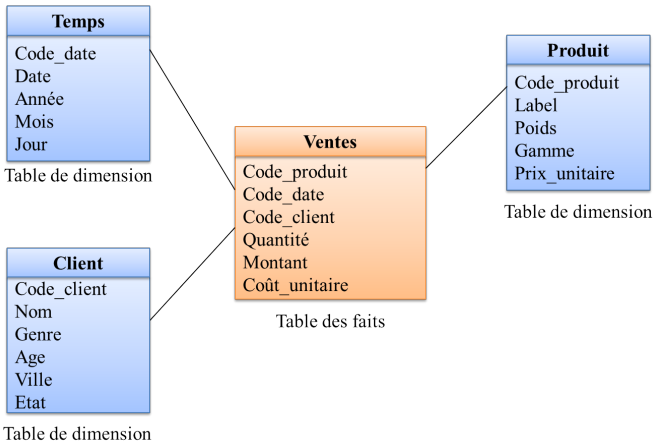


FIGURE 2.5 – Exemple d'un entrepôt de données modélisé par un schéma en étoile

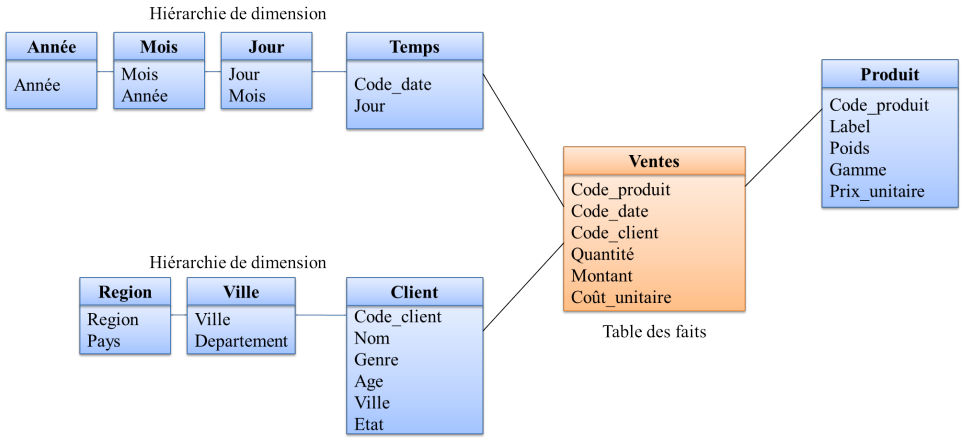


FIGURE 2.6 – Exemple d'un entrepôt de données modélisé par un schéma en flocon de neige

faciliter l'accès contrairement au cas normalisé souvent employé dans les bases de données traditionnelles. Ce modèle contient deux types de concepts:

1. un fait qui représente une fonction numérique qui peut être évaluée en tout point du modèle multidimensionnel en agrégeant les données correspondant à ce point mesure d'activité (critère d'analyse);
2. une dimension présente le point de vue selon lequel on veut voir les données décrites par un ensemble d'attributs ⟶ un axe de l'analyse. Par exemple, nous pouvons trouver les commandes, les achats, les réclamations, les produits, les clients, etc.

Un entrepôt de données peut être modélisé de deux manières différentes : (1) le modèle *multidimensionnel* MOLAP (*Multidimensional OnLine Analytical Processing*), où un tableau multidimensionnel à n dimensions est utilisé pour représenter les données (cf. figure 2.4), et (2) le modèle *Relationnel* ROLAP (*Relational OnLine Analytical Processing*), qui offre plusieurs types de schémas plus adaptés aux besoins multidimensionnels, analytiques et décision-

nels comparé au modèle normalisé en 3-ième Forme Normale (3FN).

Le modèle ROLAP est le plus utilisé pour modéliser un entrepôt de données. Dans ce modèle, si l'entrepôt est représenté par une table de faits et plusieurs dimensions, le schéma obtenu est dit *schéma en étoile* (cf. figure 2.5). Cependant, dans certains cas, les dimensions sont munies d'hiérarchies permettant de passer d'un niveau moins détaillé au plus détaillé. Par exemple, la dimension *Temps* contient l'attribut *année* qui peut être détaillé par les valeurs de l'attributs *mois*, qui lui-même peut être détaillé par les valeurs de l'attribut *jour* : année ⇒ mois ⇒ jour.

Dans le cas où les dimensions comportent des hiérarchies, le schéma est dit en *flocon de neige* comme le montre l'exemple dans la figure 2.6. Un autre type de schéma existe mais qui est moins répandu que les deux premiers : c'est le schéma en constellation. Dans ce type de schéma, il existe plusieurs tables de faits qui ne sont pas concernées par le même ensemble de dimensions. Parmi les schémas que nous venons de présenter, le modèle en étoile est le plus répandu dans les entrepôts de données.

Bases de données de type NoSQL

Plusieurs organisations collectant et manipulant de grandes quantités de données non-structurées (textes, emails, multimédia, etc.), spécialement les organisations du Web 2.0 (Google, Amazon, etc) où l'utilisateur est devenu producteur d'informations et pas uniquement simple consommateur, génèrent de plus en plus de quantités de données. Plusieurs exemples peuvent être cités pour illustrer cette évolution comme le site de vente aux enchères eBay dont les données dépassent les 6.5 Pera Octet, ou le réseau social Facebook dont la quantité de données a atteint les 2.5 PB. Pour traiter et gérer ces données, ces organisations se tournent actuellement vers des modèles de stockage plus adéquats que le modèle relationnel, qui sont les modèles NoSql (Not Only SQL).

Le mouvement NoSql a été lancé en 2007 par les deux BDs Dynamo et BigTable développées respectivement, par les compagnies Amazon et Google et qui ont inspiré par la suite plusieurs applications NoSql [**?**]. NoSql désigne *Not Only SQL*, ce qui signifie que ces modèles ne s'opposent pas au modèle relationnel, mais doivent être vu comme une alternative possible pour certains types d'applications. On pourrait penser que la notion de bases de données non relationnelles est connue depuis quelque temps déjà. Plusieurs modèles non relationnels ont été proposés dès les années 60 : les bases de données hiérarchiques, les bases de données de graphe, les bases de données objet. La différence est que le mouvement NoSql survient dans un tout autre contexte (Internet, Bigdata, etc.). La différence entre le modèle relationnel et le modèle NoSql se présente dans ces quelques points :

— **Le schéma de données** : alors qu'une base de données relationnelle nécessite la connaissance de la structure des données *à priori* (un schéma prédéfini), les modèles NoSql permettent de modéliser et de faire évoluer le schéma de données à postériori, comme

c'est le cas dans plusieurs applications réelles [4].

— **Langage d'interrogation** : les SGBD relationnels utilisent le langage d'interrogation SQL. Une des critiques souvent reprochée à ce langage est qu'il nécessite la connaissance du schéma de données pour pouvoir formuler une requête. De plus, le langage SQL est destiné à répondre à plusieurs types de requêtes complexes (jointures, union, etc) qui ne sont pas utilisées dans la majorité des applications du web 2.0. Ces applications du web 2.0 utilisent un certain type de requêtes qui, dans l'ensemble, représentent des requêtes simples (évitant les jointures) et accèdent à un petit nombre d'enregistrements, comme les requêtes de : mise à jour des mails, des profils utilisateurs, les posts, les wikis, les publicités, etc. La plupart des bases de données NoSql proposent leur propre langage d'interrogation, mais présentent des capacités d'interrogations limitées.

— **Support des contraintes *ACID* pour les transactions**: les contraintes ACID représentent les quatre principaux attributs d'une transaction de données. Pour rappel, une transaction est un ensemble d'instructions SQL permettant d'effectuer des modifications sur la base tout en gardant la cohérence des données.

La plupart des systèmes NoSql (basé sur un modèle NoSql) sont développés pour tourner dans des environnements clusters distribués (généralement dans une architecture orientée web service), où la scalabilité et la tolérance aux pannes sont deux contraintes très importantes à supporter. Pour répondre à ces deux contraintes, ces systèmes effectuent des compromis entre les propriétés ACID, la gestion des transactions, et les possibilités d'interrogation. Ils proposent leur propre modèle de données (flexible) et leur propre langage d'interrogation. Ces nouveaux modèles de stockage doivent donc répondre aux trois caractéristiques principales :

— **Flexibilité du schéma de stockage** : Ces modèles requièrent des schémas suffisamment flexibles où le nombre d'attributs est variable, où les valeurs nulles sont tolérées et où plusieurs champs sont optionnels (listes de contacts, adresses, listes de livres favoris, etc).

— **Scalabilité des données** : même si la scalabilité des SGBD relationnels a fait l'objet de plusieurs recherches, notamment l'utilisation de partitionnement efficace pour l'équilibrage de charge, de nombreux progrès restent à faire. Les systèmes NoSql sont conçus de manière à répartir les traitements sur plusieurs noeuds afin d'assurer une bonne scalabilité de données.

— **Grande tolérance aux pannes** : la disponibilité des données doit toujours être assurée même en cas de panne d'une ou plusieurs machines. Alors que les technologies de réplication disponibles dans les SGBDR sont limitées et avantagent généralement la cohérence des données à leur disponibilité (une application ACID préfère mettre les données incohérentes indisponibles jusqu'à ce que leur cohérence soit vérifiée), une base de données NoSql effectue un compromis entre la cohérence des données et leur disponibilité.

4. `http://guide.couchdb.org/editions/1/fr/why.html`

2.4 Évolution des modèles conceptuels

Depuis les années 70, les modèles conceptuels ont émergé suite à la proposition du modèle Entité/Association par Dr. *Peter Chen* [**?**]. Plusieurs autres modèles conceptuels, comme les modèles orientés objet ou les modèles sémantiques, ont complété le cycle de conception des bases de données afin de fournir un modèle de données proche de l'utilisateur final et complètement indépendant de toute technique d'implémentation. Cette vision de la conception des bases de données distinguant les différents niveaux d'abstraction a été unifiée dans l'architecture ANSI/SPARC [**?**] qui est actuellement admise dans la communauté des bases de données. Cette architecture distingue le niveau conceptuel, du niveau externe représentant les vues des utilisateurs sur les données, du niveau interne spécifiant la représentation des données telles qu'elle sera implémentée dans le système final.

On note que l'avènement de la technologie des ontologies a fait étendre le cycle de conception des bases de données au niveau conceptuel, logique et physique de manière horizontale (au sein de chaque phase). Les ontologies constituent donc un modèle de données conceptuel, à partir duquel le concepteur peut extraire le modèle conceptuel spécifique à une base de données particulière relevant du domaine de l'ontologie. Plusieurs approches ont ensuite été proposées afin de traduire un modèle ontologique en représentations logiques et physiques. Pour une traduction vers le modèle relationnel par exemple, un modèle ontologique peut être traduit selon l'approche binaire, verticale ou horizontale [**?**]. Au niveau physique, l'architecture même de la base de données a évolué en une architecture permettant le stockage du modèle de données et aussi du modèle ontologique. Cette nouvelle architecture est appelée "base de données à base ontologique" (BDBO).

Plus récemment encore, les modèles de données se sont encore diversifiés par l'émergence des bases de données " In-Memory " et aussi par l'avènement du *Big Data* et du *Cloud*, qui ont fait émerger les bases de données noSQL nécessitant des modèles de données moins rigides que le modèle relationnel, et. De nouveaux modèles de données logiques et physiques ont ainsi fait leur apparition comme les modèles *key-value*, les modèles orientés graphe ou les modèles orientés document. Le choix du modèle de données le plus adéquat dépend du type de l'application, des contraintes imposées par les besoins techniques (contraintes de performances, énergétiques, etc.) et aussi de la charge de requêtes des utilisateurs finaux.

2.5 Les approches de conception

Afin de fournir au concepteur le moyen de générer le schéma final de sa base de données à partir des besoins des utilisateurs, plusieurs approches de conception ont été proposées. Les deux premières générations d'approches de conception sont les suivantes :

— l'approche dirigée par le modèle de Codd (dirigée par les dépendances fonctionnelles) : cette approche prend comme entrée une relation universelle contenant toutes les propriétés pertinentes pour l'application à concevoir. La relation universelle est composée

de tous les attributs identifiés lors de l'étude de l'existant. Cette approche se base sur un ensemble de dépendances fonctionnelles définies entre les attributs de la relation universelle, et une décomposition progressive de cette dernière. La décomposition se base sur des règles de normalisation de la relation universelle.

— l'approche dirigée par dirigée par le modèle de Chen : contrairement à l'approche précédente qui ignore totalement le niveau conceptuel, cette deuxième approche prend comme entrée un modèle conceptuel (notamment le modèle E/A ou un autre modèle comme le diagramme de classes UML). L'approche effectue ensuite la génération du modèle logique et la normalisation de ce modèle. La plupart des processus de conception qui ont suivi s'inspirent cette approche, et se basent sur l'établissement de règles de traduction pour le passage du niveau conceptuel au niveau logique et niveau physique.

Plusieurs autres approches de conception ont suivi, soutenues par des outils de conception proposés par la communauté scientifique et par la communauté industrielle comme AMC Designer [5] ou Rational Rose [6] d'IBM. Ces outils sont enseignés dans les parcours universitaires et sont utilisés par les ingénieurs dans leur travail de conception quotidien. Actuellement, les outils de génie logiciel permettent de générer une base de données selon un système de gestion cible (Oracle, SQL Server, Mysql, etc.).

Plus récemment, des approches offrant davantage d'automatisation au processus de conception ont été proposées, issues notamment des approches dirigées par les modèles, comme l'approche MDA qui a été standardisée par l'*Object Management Group* (OMG). Ces approches sont issues de la communauté du génie logiciel, mais ont été adoptées pour la conception des bases de données. MDA se base sur la définition de modèles, de leur abstraction en métamodèles et des règles de transformation entre les modèles. MDA procède selon la vision de séparer la spécification d'un système des détails d'implémentation du système selon les capacités de la plate-forme d'implémentation [**?**]. MDA se base sur trois niveaux : CIM, PSM, PIM [**?**] :

— Le modèle CIM (computation independent model) fournit une vue du système indépendamment des détails de sa structure d'implémentation. CIM est aussi appelé modèle du domaine, car il fournit le vocabulaire des praticiens du domaine du système à concevoir. Certaines approches proposent d'utiliser une ontologie de domaine comme modèle CIM car elle fournit ce vocabulaire et cette spécification du domaine [**?**]

— Le modèle PIM (platform independent model) fournit une vue indépendante de sa plateforme d'implémentation. Ce modèle définit donc une plateforme générique de telle sorte que le système à concevoir puisse être utilisé sur plusieurs plate-formes/technologies du même type.

— Le modèle PSM (platform specific model) fournit une vue du système spécifique à une plate-forme définie. PSM permet de rajouter les détails d'implémentation d'une plateforme ou technologie spécifique aux spécifications fournies par le modèle PIM.

5. http://help.sap.com/poweramc
6. http://www-03.ibm.com/software/products/fr/enterprise

Des règles de transformation sont ensuite établie entre les trois (CIM vers PIM ensuite vers PSM). Généralement ces règles sont définies entre les méta-modèles de chaque modèle, afin de permettre de transformer leurs modèles instances. Par exemple, le concepteur peut établir des correspondances (mappings) entre le méta-modèle E/A (le CIM dans ce cas) et le méta-modèle relationnel (le PIM), afin de transformer n'importe quel modèle E/A en u modèle relationnel.

La disposition de données sur les supports (modèles de stockage interne) est très importante pour accroître les performances et les capacités de la base de données. Nous détaillons dans ce qui suit les modèles de stockage disponibles.

3 Évolution de Plate-formes

D'un point de vue architecturale, les bases de données relationnelles sont utilisées sur d'autres systèmes informatiques autre que le centralisé. Les architectures les plus répandues sont l'architecture distribuée et parallèle. Les bases de données distribuées consistent à répartir des parties des données sur plusieurs sites dans un réseau. Chaque site possède son propre $\mathcal{SGBD}$ qui fonctionne indépendamment des autres. Les utilisateurs ont accès à la partie de la base de données pertinente à leurs tâches qui se trouve à leur emplacement ou à un emplacement proche. Les bases de données parallèles sont développées pour des fins de performances. Cette évolution a fait naître un cycle de vie de déploiement de bases de données avancées sur des plate-formes. Ce dernier comporte de cinq étapes principales : (1) le choix de l'architecture matérielle, (2) la fragmentation de la base de données, (3) l'allocation des fragments, (4) la réplication des fragments et (5) l'équilibrage de charges [**?**]. Le choix d'une architecture matérielle destinée à supporter une base de données volumineuse est guidé principalement par le souci d'atteindre le meilleur rapport prix/performances, l'extensibilité et la disponibilité des données [**?**]. Actuellement, les architectures parallèles sont disponibles sous plusieurs formats tels que les Multi-Processeurs Symétriques (SMP), les Clusters, les Machines Massivement Parallèles (MMP). Ces architectures sont classifiées selon les critères suivants: partage de la mémoire (shared-memory), partage des disques (shared disks), sans aucun partage (shared nothing) et partage partiel des ressources (shared-something). Cette classification a donné lieu à trois classes d'architecture: *shared-memory, shared-disk, shared-nothing* et *shared-something*. La fragmentation consiste à décomposer horizontalement les objets d'accès (tables, vues matérialisées, index) en un ensemble de partitions disjointes. Cette fragmentation est dirigée par les prédicats utilisés par les requêtes. L'allocation des fragments est une phase importante pour atteindre la haute performance des architectures parallèles. Elle consiste à placer des fragments d'une manière optimale sur l'ensemble des noeuds de la machine parallèle. Plusieurs techniques d'allocation ont été proposées. On peut citer: le placement circulaire (round-robin) qui place les tuples en fonction de leur numéro d'ordre sur les noeuds. le placement par hachage (*hash placement*) qui distribue l'ensemble des tuples en utilisant une fonction de hachage définie sur un ensemble d'attributs. Le placement par intervalle (*range partitionning*) distribue les tuples

d'une table en fonction de la valeur d'un ou de plusieurs attributs, formant alors une valeur unique dite clé, par rapport à un ordre total de l'espace des clés.

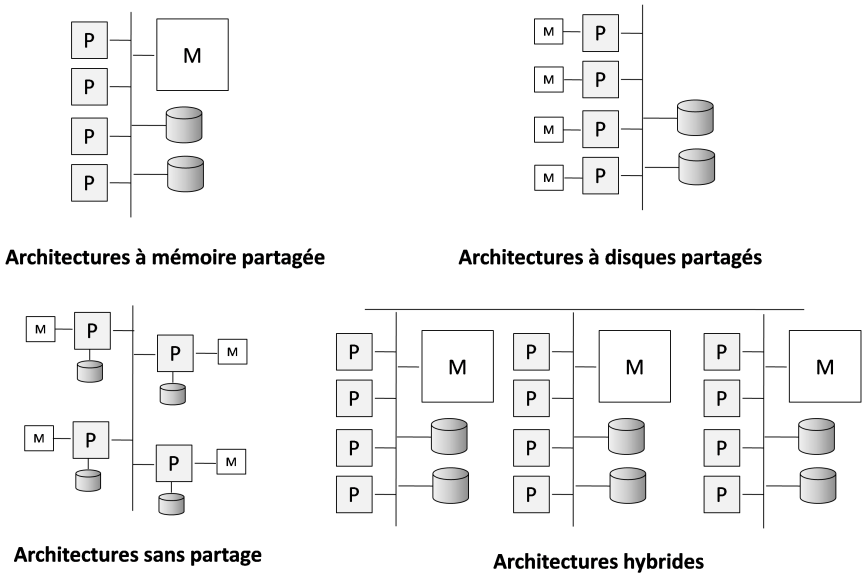


FIGURE 2.7 – *Architetcures matérielles usuelles*

La réplication est une technique qui consiste à créer et à maintenir des données et des ressources dupliquées dans un système distribué. Chaque copie est nommée réplique (*replica*). Son rôle principal est d'augmenter la fiabilité et la disponibilité des données ainsi que les performances d'accès. Elle consiste à placer plusieurs copies de l'objet sur différents noeuds. Elle fournit une grande disponibilité des données et garantit la tolérance aux pannes en dépit de la défaillance des noeuds. Elle améliore également les performances d'accès en augmentant la localité de référence.

Une fois que les données sont fragmentées, allouées et répliquées sur les noeuds de traitement, une stratégie de traitement doit être définie. Le traitement de requêtes consiste à les réécrire en utilisant le schéma de fragmentation puis à placer les sous-requêtes générées sur les noeuds de traitement. Le principal enjeu est d'allouer la charge de requêtes d'une manière équilibrée.

4 Évolution des modèles de stockage des données

Deux modèles de stockage interne pour les bases de données relationnelles ont vu le jour. Le stockage en ligne et en colonne.

4.1 Stockage orienté ligne

Dans ce modèle, le $\mathcal{SGBD}$ stocke les données sous forme de **lignes** comme illustré dans la figure 2.8. Les lignes ou encore les **tuples** sont composés de données élémentaires appelées **attributs**. Les lignes sont stockées dans des blocs de disque appelés **pages**. Enfin, ces pages, qui sont en général de taille de 8 ko, sont stockées dans des **fichiers**.

$$attributs \in tuples \in pages \in fichiers$$

Chaque ligne (tuple) d'une table possède un identifiant unique appelé ROWID. Ce dernier est utilisé en interne par le système pour se référer aux données. Le RowID est composé de l'identifiant de la page contenant ce tuple, ainsi que de sa position au sein de cette page.

$$RowID =< Id_page, \#_emplacement >$$

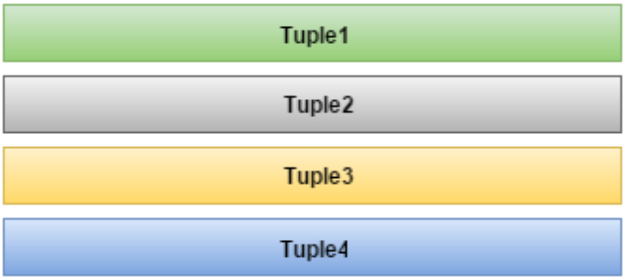


FIGURE 2.8 – stockage par lignes

Contrairement à la clé primaire d'un tuple qui représente un pointeur *logique*, le RowID est en fait un pointeur *physique* indépendant de la clé primaire et qui est souvent un entier de taille 64 ou 128 bits dans la plupart des $\mathcal{SGBD}$ modernes.

RowID	ProduitID	Designation	Marque	Prix $
001	15	clavier	dell	25
002	20	souris	dell	15
...	...	...	...	...
...	...	...	...	...
003	25	écran	dell	50
004	30	manette de jeu	dell	60

TABLE 2.4 – Stockage orienté ligne

Les systèmes de stockage orientés ligne sont conçus pour retourner efficacement (meilleur temps de réponse) les données relatives à un tuple entier. Cela correspond à beaucoup de cas de figures ou l'utilisateur tente de récupérer toutes les informations sur un objet bien précis ou un petit ensemble d'objets. En stockant toutes les données élémentaires du même objet sur

le même bloc ou sur des blocs contigus, l'accès est nettement plus rapide en termes de temps d'accès disque.

Une large base de données occupe un nombre important de blocs sur le support de stockage. Les systèmes de stockage orientés ligne ne sont pas efficaces lorsqu'il s'agit d'exécuter des requêtes qui nécessitent le parcours d'une large base de données car cela nécessite plusieurs opérations de disque. Par exemple, pour trouver tous les produits du tableau 2.4 qui ont des prix entre 15 et 60, le $\mathcal{SGBD}$ devra balayer l'ensemble de données à la recherche des tuples correspondants. Ce qui peut induire un nombre important d'opérations sur la mémoire secondaire.

Caractéristiques des systèmes orientés lignes

Pour résumer, les systèmes de stockages orientés lignes sont caractérisés par:

— un stockage de tous les attributs du même tuple sur des zones voisines (octet par octet sur le support de stockage);
— une utilisation de blocs de petite taille (4KO ou 8KO selon le $\mathcal{SGBD}$);
— une utilisation d'un exécuteur et d'un optimiseur de requête classique (à la manière du système $\mathcal{R}$ [**?**]);
— une écriture et suppression d'un tuple en une seule opération physique;
— une meilleure adaptation pour les systèmes transactionnels ($\mathcal{OLTP}$) ;
— une optimisation pour les opérations d'écriture;
— une utilisation des index pour optimiser les accès aux données.

4.2 Stockage orienté colonne

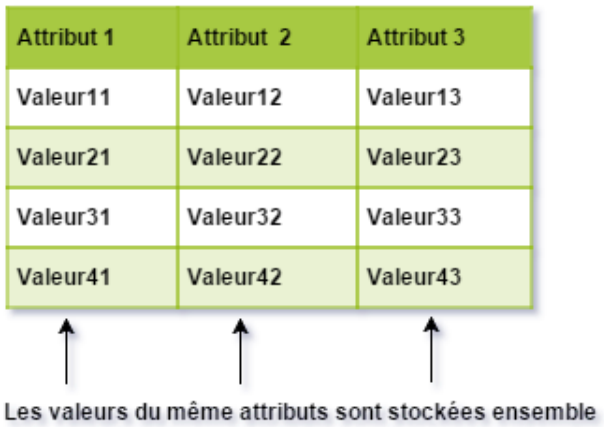


FIGURE 2.9 – stockage par colonnes

Un SGBD orienté colonne sérialise et stocke les données par colonne comme l'illustre la

Zone1	001:15, 002:20, 003:25, 004: 30
Zone2	001:clavier, 002:souris, 003:Ecran, 004: manette de jeux
Zone3	001, 002, 003, 004: Dell
Zone4	001:25, 002:15, 003:50, 004: 60

FIGURE 2.10 – Exemple stockage par colonnes

Figure 2.9. Plus précieusement, les valeurs d'un attributs sont stockées ensemble sur le même fichier ou sur la même zone sur le support de stockage (voir la Figure 2.10).

Les $\mathcal{SGBD}$ orientés colonnes sont mieux adaptés pour les requêtes analytiques qui font un parcours intense d'une large base de données en faisant des agrégations sur un ensemble réduit de colonnes. L'idée de développement de ces systèmes est la technique de la fragmentation verticale dans les bases de données relationnelles. Il existe beaucoup de $\mathcal{SGBD}$ commerciaux orientés colonnes dont la plupart sont inspirés de prototypes académiques comme C-store, Monetdb et VectorWise [**?**] [7], [8]. Contrairement aux $\mathcal{SGBD}$ orienté lignes, les systèmes orientés colonnes ne lisent que les colonnes référencées dans la requête, ce qui permet de lire plus de données lors du même accès au support de stockage.

Caractéristiques des Systèmes orientés Colonnes

Les systèmes de stockages orientés colonnes sont caractérisés par :

— une compression des données pour optimiser l'utilisation de l'espace de stockage;
— une utilisation de blocs de grande taille (minimum 64KO);
— un tri des données avant leur stockage plutôt que d'utiliser des index;
— une optimisation pour les lectures;
— une meilleure adaptation pour les systèmes décisionnels ($\mathcal{OLAP}$) .

5 Évolution des supports de stockage

Comme nous l'avons déjà indiqué dans l'introduction générale, les supports de stockage sont hiérarchisés. Nous trouvons trois principaux types de stockage: les bandes magnétiques, les disques durs et les mémoires flashs, que nous détaillons dans les sections suivantes.

7. db.csail.mit.edu/projects/cstore/
8. https://www.monetdb.org

5.1 Les Bandes Magnétiques

La bande magnétique est un support de stockage qui a été inventée pour les enregistrements sonores en 1928. Puis, elle a été utilisée pour le stockage des données sur les ordinateurs centraux (*main-frames*) dans les années 1950. La bande magnétique est un support de stockage à accès séquentiel, cette propriété physique la rend inutile pour les applications qui nécessite un accès direct aux données avec un temps de réponse acceptable tels que les bases de données transactionnelles.

Pour des types d'applications comme l'archivage de données, l'accès direct et le temps de réponse ne sont pas souvent les critères les plus recherchés. L'archivage de volume très important de données nécessite un support de stockage robuste et de faible coût. Dans des applications pareils à l'archivage de donnée, le stockage sur bande magnétique prend tout son intérêt.

Dans le domaine des bases de données, les sauvegardes (data backups) physiques sont des copies des fichiers physiques utilisés dans le stockage et la récupération d'une base de données. Ces fichiers incluent les fichiers de données, les fichiers de contrôle et de journalisation archivés. En plus des bonnes caractéristiques des bandes magnétiques pour l'archivage citées avant, sa durée de vie qui peut aller jusqu'à 20 ans la rend un bon candidat pour la sauvegarde et la reprise après pannes des bases de données.

5.2 Les disques durs

Le disque dur $\mathcal{HDD}$ a joué un rôle important dans le développement des bases de données grâce à leur propriété d'accès direct aux données. Les disques durs ont fait leur apparition la première fois en 1956 dans les laboratoires d'IBM à San Jose (Californie). Le premier $\mathcal{HDD}$ avait une capacité de 5 MO sous forme de 50 disques superposés de 24 pouces de diamètre. Le développement de la mécatronique a donné naissance à des disques dur de 100 GO battis seulement sur un seul disque de 3 pouce et demi. Depuis, les $\mathcal{HDD}$ ont connu un développement rapide, ce qui a récemment donné naissance à des disques de capacité de 10 TO comme le modèle *Ultrastar He10* [9] à l'hélium. Les avantages des $\mathcal{HDD}$ sont leur prix moins élevé par rapport à leurs concurrents comme les $\mathcal{SSD}$ (*Solid-State Drive*, voir la section 5.3).

Physiquement, un disque dur est composé de plusieurs plateaux, concentriques et superposés. Chaque face d'un plateau est recouverte d'oxyde magnétique. Les faces sont constituées de pistes, elles même divisées en secteurs de taille généralement de 512 octets. Le plus souvent, il y a autant de tête de lecture/écriture qu'il y a de face. Enfin, les pistes de même numéro sur tous les plateaux constituent un cylindre. Ainsi, la donnée pourra être adressée par l'ensemble (cylindre, tête, secteur). La figure 2.11 illustre la structure physique d'un disque dur.

9. http://www.hgst.com

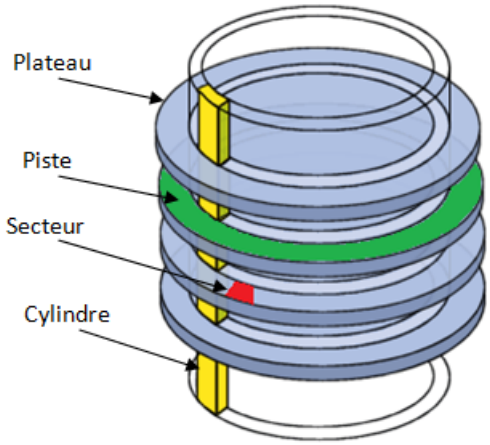


FIGURE 2.11 – Structure physique des disques durs

5.2.1 Les performances des HDD

Les performances d'un disque dur représentent un facteur déterminant qui influence le temps de réponse globale d'un système. Un disque dur lent peut ralentir un processeur rapide plus que tout autre composant d'un système informatique. La vitesse des disques durs actuels est de l'ordre de 7200 à 15000 $\mathcal{RPM}$ (tours par minutes). La vitesse effective d'un disque dur est déterminée par un certain nombre de facteurs. La vitesse de rotation qui représente la vitesse à laquelle les plateaux tournent est un élément essentiel de la performance du disque dur car elle influe directement sur la latence et le taux de transfert du disque.

Un autre facteur important est la latence mécanique qui est mesurée en millisecondes. Cette latence comprenne à la fois:

— le temps de recherche qui est une mesure qui définit le temps nécessaire pour qu'une tête de lecture / écriture trouve la piste recherchée sur le disque.

— La latence relationnelle est le temps moyen pour déterminer le bon secteur qui contient la donnée recherchée.

Ainsi, le temps d'accès moyen d'un disque dur est le temps moyen que met le disque entre le moment où il a reçu l'ordre d'une opération de lecture/écriture de données et le moment de terminaison de l'opération. Une autre métrique intéressante pour mesurer la performance des disques durs est le taux de transfert, débit ou encore la bande passante. Elle doit représenter la quantité de données pouvant être lues ou écrites sur le disque par unité de temps à condition que le transfert doit être entre le disque et la mémoire centrale.

Enfin, on note que les disques dur sont doté de mémoire cache qui garde les données les plus récemment utilisées, ce qui permet d'améliorer les performances.

5.3 Les disques à base de mémoire flash: SSD

Un disque $\mathcal{SSD}$ (*Solid State Drive*) est un support de stockage de données à base de mémoire flash. Les bits sont stockés dans les cellules qui peuvent être composées de transistors à grille flottante.

Selon le mode de connexion en série ou parallèle des cellules [**?**], on peut distinguer deux type de mémoire flash $\mathcal{NAND}$ et $\mathcal{NOR}$ (voir la figure 2.12):

1. $\mathcal{NOR}$: les cellules sont connectées en parallèle, ce qui les rend rapides en lecture. La mémoire flash NOR est utilisée pour stocker du code exécutable.
2. $\mathcal{NAND}$: les cellules sont connectées en série. Ce mode offre une meilleure densité de stockage et un accès rapide en écriture.

Une cellule peut être de type
— $\mathcal{SLC}$(Single level cell): les transistors peuvent stocker un seul bit;
— $\mathcal{MLC}$ (Multi-level cell): stockage de deux bits.
Contrairement au $\mathcal{HDD}$ qui contiennent des composantes mécaniques(tête de lecture/écriture, disques rotationnels), les $\mathcal{SSD}$ stockent les données selon un procédé purement électronique. Un $\mathcal{SSD}$ peut être utilisé simplement comme un $\mathcal{HDD}$ via des interfaces ordinaires (ATA, SATA et SCSI). Dans le reste de ce chapitre nous ne discutons que la mémoire flash $\mathcal{NAND}$ qui est la mieux adaptée pour le stockage de données et qui est la solution choisie par tous les fabricants des disques $\mathcal{SSD}$.

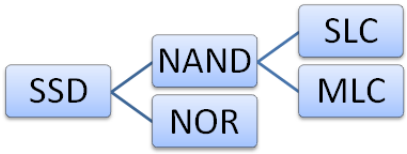


FIGURE 2.12 – Les types du *SSD*

5.3.1 Organisation des disques SSD

Un disque $\mathcal{SSD}$ est composé principalement d'un contrôleur et d'un ensemble de puces flash qui stockent les données (voir la figure 2.13). Le processeur reçoit les commandes d'entrée/sortie via l'interface hôte. Le contrôleur joue le rôle d'interface entre le système hôte et les puces flash. Le disque $\mathcal{SSD}$ intègre également une mémoire interne ($\mathcal{SRAM}$) utilisée pour stocker des informations sur la $\mathcal{FTL}$ (voir section 5.3.3). Un disque $\mathcal{SSD}$ peut aussi intégrer une $\mathcal{DRAM}$ comme un cache.

Un package flash est un ensemble de puces. Les puces sont formées de blocs eux même composés de pages flash. Un bloc contient un nombre de pages multiple de 32 (généralement 64 pages). Les pages sont de taille égale à un multiple de 2 (2KO, 4KO, 8KO, etc.) [**?**]. Une

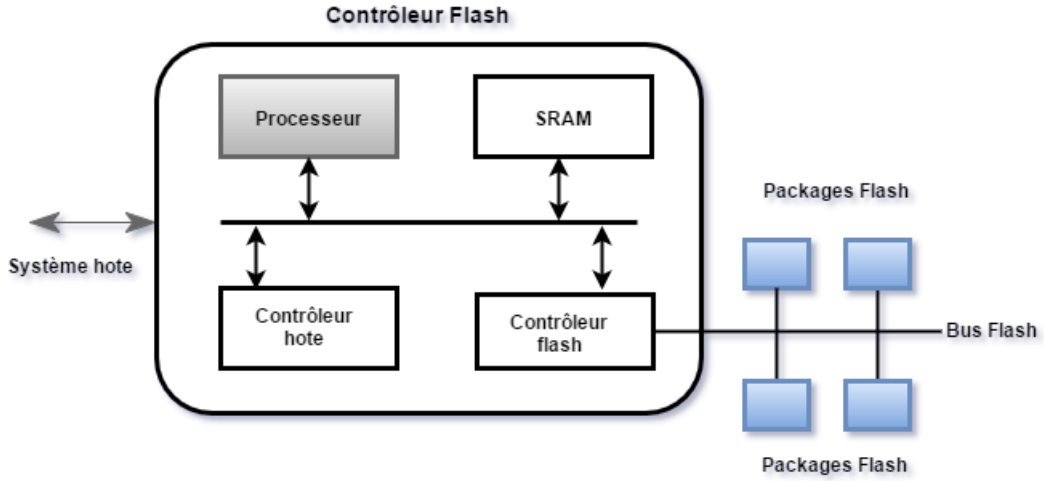


FIGURE 2.13 – Architecture d'un disque SSD

page flash est la plus petite unité adressable. Elle est divisée en deux régions distinctes, une région pour le stockage de données appelée BI (IN-Band) et une autre pour le stockage de métadonnées utilisées par la $\mathcal{FTL}$ (voir la section 5.3.3) appelée Out-of-band ($\mathcal{OOB}$). La région $\mathcal{OOB}$ contient des informations comme: (1) le code de correction d'erreur (Error Correction Code) pour vérifier la cohérence des données, (2) le numéro de page logique correspondant à des données stockées dans la zone de données, et (3) l'état des pages. Chaque page de la mémoire flash peut être dans l'un des trois états suivants: (i) valide, (ii) invalide, et (iii) libre / effacé. Lorsqu'il n'y a pas de données écrites sur une page, elle est à l'état libre.

5.3.2 Opération d'entrée/sortie

Contrairement au $\mathcal{HDD}$ qui offre deux opérations de lecture et d'écriture, les contraintes technologiques de la mémoire flash imposent (mémoire de type EEPROM) trois opérations différentes : lecture, écriture et effacement. Les opérations de lecture et écritures s'effectuent au niveau de la page. L'effacement s'applique sur un bloc entier. L'écriture d'une page p_i qui se trouve sur un bloc B_l nécessite (1) la copie de toutes les pages de B_l vers un nouveau bloc libre B_m en prenant en considération les mises à jour de la page p_i (2) invalider le bloc B_l (3) mettre à jour l'adressage pour que B_m devient le nouveau bloc . Le bloc B_l sera effacé pour une éventuelle nouvelle utilisation. Ce mode de fonctionnement de l'opération d'écriture est appelé effacement avant écriture (*erase before write*).

Contrairement à un disque $\mathcal{HDD}$, les opérations de lecture et d'écriture sont très asymétriques ; effacer et écrire une page flash est beaucoup plus long que sa lecture [**?**].

Disque			
Type	SSD	SSD	HDD
Type cellule	MLC	MLC	–
Interface	SATA 2.0	SATA 2.0	SATA 3.0
Capacité	65GO	80GO	4TO
Pages par block	128	128	–
Taile page	4K	4K	–
Taille bloc	512KO	512KO	–
Lecture séquentielle(MB/s)	100	254	185
Ecriture séquentielle(MB/s)	92	78	185
4Ko lecture séquentielle(MB/s)	17	23.6	0.54
4Ko lecture aléatoiree(MB/s)	5.5	11.2	0.85
4Ko lecture séquentielle(KIOPS)	4	6	0.14
4Ko lecture aléatoire(KIOPS)	1.5	2.8	0.22
Métrique	KIOPS: 1000 IO/s; MB/S: Mégaoctets par seconde		

TABLE 2.5 – Comparaison des caractéristiques des SSD et HDD

Les Avantages des disques SSD

Les disques $\mathcal{SSD}$ et les supports de stockage basés sur les mémoires flash sont de plus en plus utilisés pour le stockages de données, allant des téléphones mobiles aux systèmes de stockage à l'échelle d'une entreprise (data centers), en raison d'un certain nombre d'avantages à savoir:

1. Intégration facile dans un système de stockage
2. Poids plus réduit
3. Temps d'accès plus court
4. Faible consommation d'énergie
5. Grande résistance aux vibrations et aux chocs

Les inconvénients des SSD

Aujourd'hui, on assiste à une baisse continue des prix des disques $\mathcal{SSD}$ alors que les annonces se multiplient du côté des grandes capacités (des disques de 2TO et 4TO sont déjà sur le marché). On peut conclure que le prix et la capacité des disques $\mathcal{SSD}$ ne seront plus un vrai obstacle dans les années à venir. L'inconvénient majeur pour les fabricants de ces disques est leur durée de vie. Pour les utilisateurs finaux, cela ne pose pas de problème car les disques sont garantis à fonctionner sans erreurs pendant une certaine période de temps. Les cellules flash sont fabriquées par un procédé technologique appelé grille flottante. Les données sont représentées par la présence ou l'absence de charges électriques dans la grille flottante. Après plusieurs cycles d'effacement de bloc, des charges électriques peuvent rester emprisonnées dans l'oxyde tunnel de cellules ce qui rend le bloc inutilisable. Ce phénomène qualifie les mémoires flash de support de stockage à durée de vie limité. Pour pallier cette lacune , les disques à base de

mémoires flash intègrent des mécanismes correction d'erreurs et de gestion des effacements de bloc. La $\mathcal{FTL}$ (flash translation layer) qui fera l'objet de discussion dans la section 5.3.3 représente le module responsable de la gestion de ces mécanismes.

5.3.3 La couche de traduction d'adresses FTL

Un avantage principal des disques $\mathcal{SSD}$ est leur intégration facile dans un système de stockage classique en utilisant des interfaces ordinaires (IDE, SATA, etc.). Ceci est garantie par la $\mathcal{FTL}$. La $\mathcal{FTL}$ joue le rôle d'intermédiaire entre le pilote de périphérique et la mémoire flash. Sans la présence de la couche $\mathcal{FTL}$, il sera nécessaire de faire recours à des systèmes de gestion spécifiques qui gère le support de stockage flash [**?**]. La $\mathcal{FTL}$ abstrait le mode de fonctionnement complexe de la flash en la présentant comme un périphérique bloc ordinaire. La $\mathcal{FTL}$ est un ensemble d'applicatifs embarquée à l'intérieur des couches matérielles du support flash. Les fonctions principales de la $\mathcal{FTL}$ sont:

— La traduction d'adresses physique/logique
— Ramasse-miettes: recyclages des blocs non valides (*garbage collection*)
— Réparation d'usure des blocs (*wear-leveling*).

5.3.4 Traduction d'adresse

Comme mentionné déjà, le rôle principale de la $\mathcal{FTL}$ est la traduction d'adresses logique reçus sur les interfaces hôtes en adresses physique. Cette traduction d'adresses est faite via des tables de correspondance adresse logique/adresse physique (LPN/PPN). La table de correspondance est stockée sur la mémoire flash et est chargée dans la $\mathcal{SRAM}$ à chaque démarrage du disque $\mathcal{SSD}$. Il existe trois type de traduction d'adresse:

Traduction d'adresse par page

Elle consiste à faire correspondre chaque page logique à une page physique [**?**]. Dans l'exemple de la figure 2.14, la requête d'écriture reçue s'adresse à la page logique 4. La $\mathcal{FTL}$ détermine l'adresse physique 6, grâce à la table de traduction. Ce mode de traduction est très flexible mais nécessite un espace mémoire important. Pour chaque page de la mémoire flash, il doit y exister une entrée dans la table.

Traduction d'adresse par bloc

Dans ce mode de traduction, la $\mathcal{FTL}$ associe à chaque bloc logique un bloc physique dans la table de traduction. Cela améliore considérablement l'espace requis pour le stockage de la table. Pour une mémoire flash composée de blocs contenant chacun n pages, la table de traduction est réduite d'un facteur n. La figure 2.15 montre un exemple de traduction par bloc. La

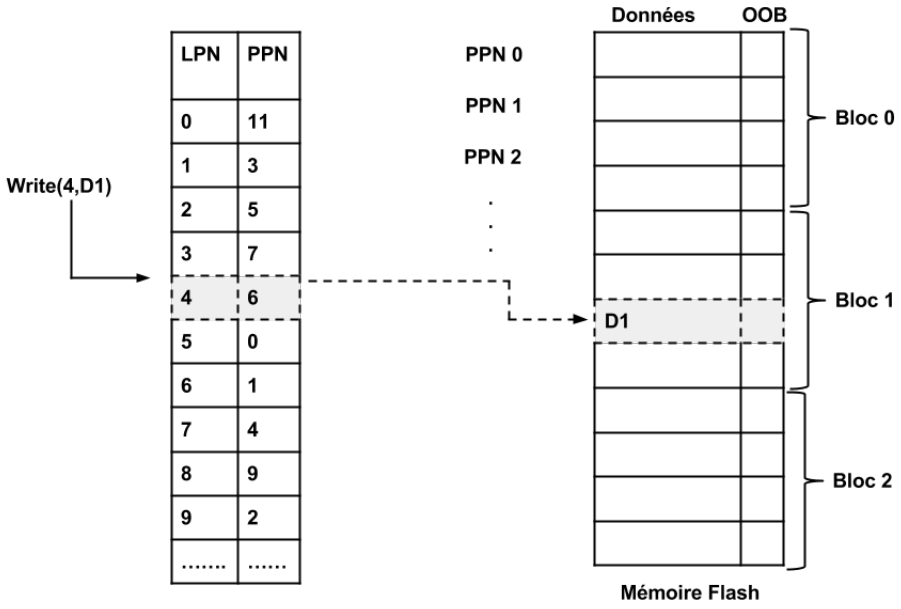


FIGURE 2.14 – Exemple de traduction d'adresse par page

requête demande d'écrire une donnée dans la page d'adresse logique 7. La $\mathcal{FTL}$ détermine tout d'abord l'adresse du bloc logique correspondant $\mathcal{LBN}$ qui est le résultat de la division entière de l'adresse logique de la page (envoyé par le système hôte) par le nombre de pages dans le bloc ($LBN = 7\,div\,4$), puis le déplacement dans le bloc (offset) qui est le reste de cette division entière. Dans le mode de traduction par bloc, le déplacement d'une page dans un bloc flash est toujours fixe. L'inconvénient ici est que l'écriture d'une seule page nécessite aussi l'écriture de toutes les pages du même bloc.

Traduction d'adresse hybride

L'objectif de cette approche est de remédier aux problèmes engendrés par la traduction par page et par bloc. Deux traductions sont utilisées. La première consiste à traduire les adresses de blocs logiques en adresses de bloc physiques et la deuxième fait la traduction des adresses pour repérer les pages à l'intérieur des blocs physiques. L'adresse du bloc physique est calculée de la même façon que la traduction par bloc. Le déplacement (offset) est représenté par l'adresse logique de requête hôte. La figure 2.16 décrit l'exemple d'une requête d'écriture de la page logique 7. La première étape consiste à calculer l'adresse du bloc logique $LBN = 7\,mod\,4$ (4 est le nombre de pages par bloc physique), ensuite l'adresse du bloc physique est déterminé à partir de table de correspondance (mapping). La donnée peut être écrite sur n'importe quelle

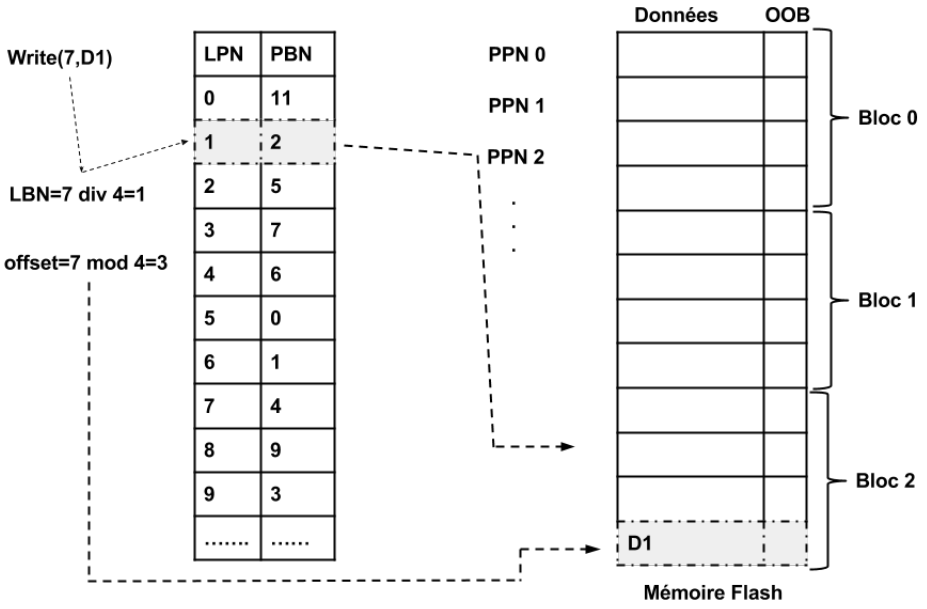


FIGURE 2.15 – Exemple de traduction d'adresse par bloc

page libre *pl* du bloc physique. L'offset qui égal à l'adresse logique de la page requise par le système hôte est inscrit dans la zone $\mathcal{OOB}$ de la page *pl* sélectionnée.

Traduction par blocs journalisés

La plupart des implémentations des $\mathcal{FTL}$ [**?**, **?**, **?**] à base de traduction hybride utilisent une approche similaire aux systèmes de fichiers journalisés. La mémoire flash est divisée en deux zones. Une zone pour les blocs de données adressés par bloc et une petite zone pour les blocs journaux(logs) adressés par page [**?**]. Lorsqu'une requête de mise à jour d'une page d'un bloc de données arrive, elle est redirigée vers un bloc journal. Les pages sont écrites séquentiellement sur les blocs-journaux. Lorsqu'un bloc-journal est saturé, il est fusionné avec le bloc de données associé au même bloc logique pour récupérer de l'espace. Cette opération est réalisée par un mécanisme de ramasse-miette (*garbage collection*) implémenté par la $\mathcal{FTL}$.

5.3.5 Répartition de l'usure et ramasse-miettes

Les mémoire flash sont des supports de stockage à durée de vie limitée. Les cellules flash peuvent se voir détériorées après un certain nombre de cycles d'écriture/effacement. L'objectif de la répartition de l'usure(wear levelling) est de répartir les opérations d'effacement sur tous les

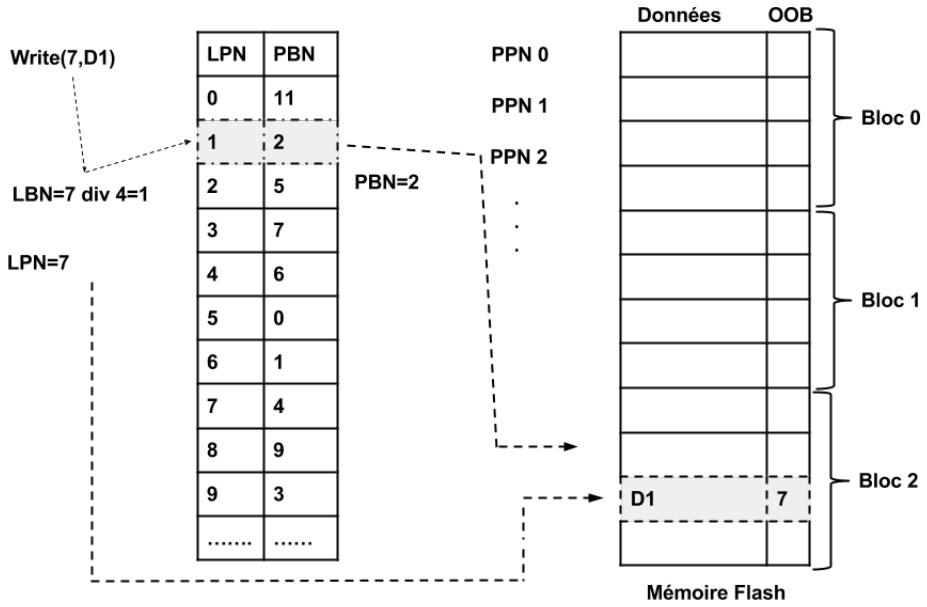


FIGURE 2.16 – traduction hybride d'adresse

blocs de la mémoire flash. Ce processus est souvent associé à un autre service de la $\mathcal{FTL}$ qui est le ramasse-miettes. Le mécanisme de ramasse-miettes assure que les pages invalides peuvent être utilisées par des commandes d'écriture future. Pour une bonne répartition de l'usure, les blocs ayant le moins nombre d'effacements sont souvent sélectionnés. Pour minimiser le temps de latence des opérations d'écriture le ramasse-miettes est lancé régulièrement comme une tâche de fond au moment où il n'y a pas d'activité d'entrée/sortie.

5.4 Les disques hybrides

5.4.1 Cloud Storage

Depuis ces dernières années, un nouveau concept est apparu, appelé **Cloud Computing**. Il consiste à déporter sur des serveurs distants des traitements et des stockages informatiques. Dans la langue française différentes traductions du terme sont données : informatique virtuelle, informatique dans le nuage, ou encore informatique dématérialisée. Le *National Institute of Standards and Technology* (*NIST*) définit le *cloud computing* comme " un modèle permettant un accès facile et à la demande, via le réseau, à un pool partagé de ressources informatiques configurables (par exemple, réseaux, serveurs, stockage, applications et services) qui peuvent être rapidement mises à disposition des utilisateurs ou libérées avec un minimum d'effort de

gestion ou d'interaction avec les services du fournisseur [**?**].

Ce qui nous mène à parler du *Cloud Computing* c'est sa partie de stockage, appelée **Cloud Storage**, ou stockage en nuage. Le *Cloud Storage* est un modèle de réseau de stockage en ligne où les données sont stockées dans des pools de stockage virtualisés qui sont généralement hébergés par des tiers. Les sociétés d'hébergement fonctionnent en grands centres de données, et les personnes qui ont besoin d'héberger leurs données peuvent acheter ou louer la capacité de stockage. Physiquement, la ressource de stockage peut s'étendre sur plusieurs serveurs. Actuellement, les nouvelles versions des $\mathcal{SGBD}$ prennent en charge le principe du Cloud. A titre d'exemple, la dernière version d'Oracle s'appelle 12c, " **c** " pour **c**loud.

Le *Cloud Storage* est un marché très récent. Le Gartner Group a publié des estimations de revenus, au niveau mondial en milliards de dollars, pour les trois services : Cloud Computing, Cloud Storage et la sauvegarde via le Cloud Storage, avec des prévision pour l'année 2015 [10] (voir le tableau 2.6).

Service	2010	2011	2015	Taux de croissance moyen
Cloud Computing	2,20	3,40	15,50	47,8 %
Cloud Storage	0,12	0,27	2,88	89,5 %
Sauvegarde	0,46	0,56	1,21	21,6 %

TABLE 2.6 – Evolution de revenus des services *Cloud* 2010-2015

A cet effet, certains chercheurs se sont intéressés dans l'optimisation du stockage des base de données dans le cadre du Cloud. Les travaux de Zhang et al. [**?**] abordent le problème de traitement par les centres de données (Data center) des charges de travail. Leur travail consiste à évaluer les coûts du service Cloud Storage pour satisfaire les clients (moindre coût possible). Les paramètres suivants sont pris en considération : les coûts initiaux des différents périphériques d'E/S (disque dur et SSD), les capacités de stockage, les performances et les coûts énergétiques. L'ensemble de ces coût est appelé TOC (pour Total Operating Cost).

Les auteurs proposent une solution heuristique à ce problème, appelé $\mathcal{DOT}$ pour *Data layout Optimized to reduce the TOC*. La méthode DOT utilise le modèle de coût de l'optimiseur de requêtes du $\mathcal{SGBD}$ pour estimer le temps des E / S des différents périphériques de stockage.

6 Évolution des Requêtes

Dans cette section, nous présentons une évolution des requêtes à travers des générations de bases de données et les approches d'optimisation.

10. Forecast Analysis: External Controller-Based Disk Storage, Worldwide, 2011-2015, 1Q11 Update, 16 March 2011

6.1 Optimisation Isolée de Requêtes

Dans la première génération de bases de données, les requêtes de type OLTP sont exécutées et optimisées d'une manière isolée. Rappelons que chaque requête dans l'univers relationnel peut être représentée par un arbre algébrique. Les feuilles de l'arbre représentent les tables de la requête. La racine de l'arbre représente la table résultat. Tous les autres noeuds de l'arbre sont des opérateurs de l'algèbre relationnelle. Plus précisément, un arbre algébrique correspond à une décomposition de la requête en opérateurs élémentaires avec introduction de tables intermédiaires. Le processus d'optimisation de requête consiste à passer d'une requête exprimée dans un langage source (le SQL dans le modèle relationnel) à une requête exprimée par un ensemble d'opérations plus élémentaires exprimées dans un langage cible (opérations dites « de bas niveau »). Ces opérations élémentaires dépendent particulièrement : de la requête SQL initiale, des contraintes d'intégrité, des structures d'optimisation utilisées (e.g., index), des statistiques sur la base de données, les algorithmes d'implémentation des opérations élémentaires (jointure par boucles imbriquées, jointure par hashage, etc.). Un plan d'exécution d'une requête est une suite possible d'opérations de bas niveau qui permettent de réaliser cette requête. Il peut exister plusieurs plans d'exécution pour une requête donnée qui donnent tous le même résultat mais avec des performances variables. Le problème de génération du plan d'exécution optimal est NP-complet [**?**]. Dans la première génération des optimiseurs de requêtes, une approche naïve d'optimisation dirigée par des règles (connue sous le nom *rule-based approach*) a été proposée . Ces règles sont appliquées sur un plan de requête. Parmi parmi celles-ci, nous trouvons: la descente des sélections (*Push Down Selections*) qui permet de réduire la taille des relations et des résultats intermédiaires manipulés.

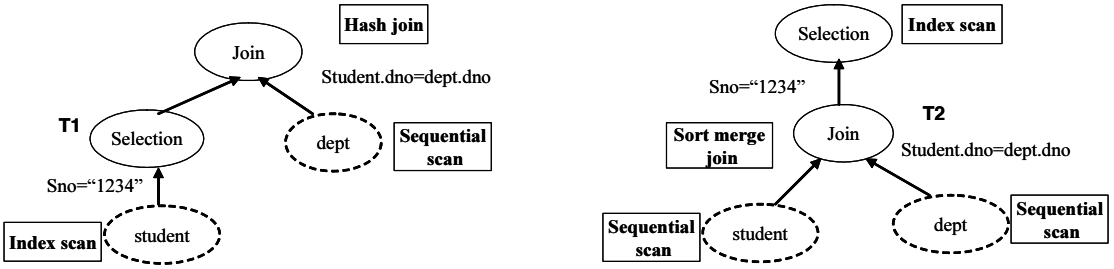


FIGURE 2.17 – Exemples de plan d'exécution de requêtes

Avec l'apparition des grosses bases de données impliquant un nombre important de tables, la deuxième génération des optimiseurs a vu le jour. Elle utilise une approche dirigée par des modèles de coût (connue sous le nom *cost-based approach*). Cette génération remet en cause les optimiseurs basés sur des règles. Pour ce faire, elle propose d'utiliser des modèles de coût mathématiques estimant le coût de chaque opération élémentaire d'un plan d'exécution d'une requête. Le coût d'un plan d'exécution est évalué en cumulant le coût des opérations élémentaires, de proche en proche selon l'ordre défini par le plan d'exécution, jusqu'au obtenir le coût total du plan. Souvent tout modèle de coût est développé en fonction des composantes principales d'un $\mathcal{SGBD}$, d'où sa décomposition en trois parties : (1) le coût des entrées-sorties (IO)

pour lire et écrire entre la mémoire et le support de stockage comme les disques, (2) le coût CPU et (3) le coût de communication sur le réseau COM (si les données sont réparties sur le réseau). Ce dernier est généralement exprimé en fonction de la quantité totale des données transmises. Il dépend de la nature de la plate-forme de déploiement.

L'optimisation isolée quel que soit l'approche d'optimisation utilisée a été utilisée dans toutes les générations de bases de données: bases de données traditionnelles, orientée objet, les entrepôts de données, bases de données sémantiques, bases de données de graphes.

6.2 Optimisation Multiple de Requêtes

En 1988, Timos Sellis a souligné l'importance de considérer l'interaction entre les requêtes lors de l'optimisation d'une charge [**?**]. Suite à ce travail, l'optimisation de requêtes a pris une nouvelle tournure, où l'optimisation ne considère plus les requêtes d'une manière isolée, mais la totalité de la charge. Cette approche est connue sous le nom de l'optimisation multi-requêtes (Multi-Query Optimization (MQO)), et a été intégrée dans une grande partie des travaux de la conception physique des bases de données avancées. Comme son nom l'indique, l'optimisation multiple concerne un ensemble de requêtes ayant une forte interaction. Les applications d'entreposage de données ont favorisé cette interaction entre les requêtes, car les requêtes de jointure passent systématiquement par la table centrale qui est la table de faits. Au lieu de travailler sur un seul plan d'exécution, un plan global unifiant les plans individuels est défini. La difficulté principale liée à cette génération d'optimiseur est double : (i) comment obtenir le plan global contenant les expressions communes et (ii) comment quantifier sa qualité. Ce problème est connu comme NP-difficile, surtout lorsqu'il y a une charge de requêtes très importantes à optimiser. Les travaux existants traitant ce problème proposent des heuristiques combinant des plans individuels et à l'aide de modèle de coût sélectionnent les meilleurs plans.

L'une des principales caractéristiques de ce problème de recherche est qu'il a été abordé dans tous les langages de requêtes associées à toutes les générations de bases de données: les bases de données traditionnelles (RDB-SQL) [**?**], bases de données orientées objet (BDOO - OQL) [**?**], les bases de données sémantiques (BDS - - SPARQL)[**?**], les bases de données distribuées [**?**], les bases de données de flux (STDB - CQL) [**?**], les entrepôts de données (SQL OLAP) [**?**], etc. (comme le montre la figure 2.18).

6.3 Charge de requêtes mixtes (OLTP/OLAP)

Comme nous l'avons déjà mentionné, l'optimisation multiple de requêtes concerne à la fois des requêtes transactionnelles et les requêtes OLAP. Les optimiseurs de requêtes des SGBD relationnelles restent des infrastructures principales pour le traitement transactionnel en ligne (OLTP), le traitement analytique en ligne (OLAP) et les charges OLTP/OLAP mixtes. Hyper [11]

11. `http://www.hyper-db.de/`

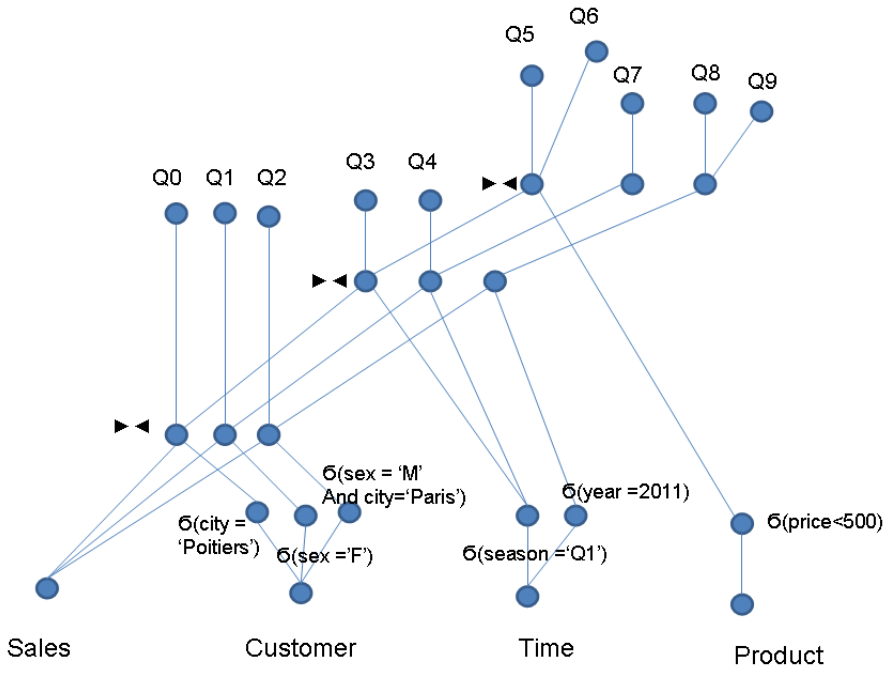


FIGURE 2.18 – Plan unifié de requêtes

est un exemple d'un SGBD dédié aux charges mixtes de type OLTP et OLAP.

7 Évolution des Structures d'Optimisation

Deux caractéristiques principales d'un entrepôt de données sont: la complexité de ses requêtes et le volume de ses données (par exemple, l'entrepôt de données de la chaîne américaine Wal-Mart comprend 40 téraoctets de données). Accéder à ce volume impose des opérations de jointures et d'agrégation très coûteuses, d'où la nécessite d'optimiser les accès.

8 Les techniques d'optimisation

Les techniques d'optimisation ont également suivi les différentes évolutions. Les techniques d'optimisation utilisées dans le contexte des bases de données traditionnelles ne sont pas bien adaptées aux environnements des entrepôts de données. En effet la plupart des transactions OTLP accèdent à un petit nombre de n-uplets, les techniques utilisées comme les index B 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 (ce type de requêtes est encore appelé requêtes d'intervalle). Réutiliser les techniques des systèmes OLTP conduirait à des index avec un grand nombre de niveaux qui ne seraient donc pas très efficaces [**?**]. Cette situation a

motivé les chercheurs et les industriels d'en proposer d'autres techniques d'optimisation. Deux classes principales composent ces techniques: (i) les techniques matérielles et les (ii) techniques logicielles.

8.1 Les techniques d'optimisation matérielles

Comme leur nom indique, ces techniques sont souvent offertes par les plateformes de déploiement et leurs systèmes de stockage. On peut citer par exemple les optimisations suivantes:

- L'utilisation de supports de stockage plus rapides (comme les mémoire flash, ...) [**?**];
- L'adoption du traitement parallèle en utilisant des processeurs supplémentaires notamment des processeurs graphique pour accélérer les requêtes [**?**];
- L'amélioration des performances du réseau dans le cas des machines parallèles [**?**]. Le traitement parallèle est une technique d'optimisation qui consiste à bénéficier du multiprocesseur (plusieurs $\mathcal{CPU}$) ou multiprocessus (multithreading). Beaucoup de travaux de recherches sont effectués dans le cadre de l'optimisation des bases de données dans un environnement parallèle [**?**, **?**, **?**, **?**]. L'un des avantages de traitement parallèle est de paralléliser l'exécution des requêtes:
  - parallélisme intra-requête: cela consiste à diviser une requête en plusieurs sous requêtes qui seront exécutées en parallèle
  - parallélisme inter-requête: exécuter plusieurs requêtes différentes en parallèle.
- L'augmentation de la taille des caches à tous les niveaux ;
- La fragmentation matérielle consiste à tirer profil du système de stockage des données. Par exemple, l'utilisation de dispositifs $\mathcal{RAID}$ (Redundant Array of Independent Disks) permet d'allouer des données sur plusieurs unités de disques. Ces dernières sont considérées comme un seul disque appelé grappe. Une bonne utilisation de cette technologie peut améliorer considérablement les performances de traitement de requêtes vu le nombre important de tête de lecture/écriture qui fonctionnent en même temps.

8.2 Les techniques d'optimisation logicielles

Les techniques d'optimisation logicielles sont souvent sélectionnées dans la phase physique du cycle de vie de conception des bases de données avancées. L'optimisation physique est le résultat d'une bonne conception de la base de données, d'une analyse approfondie des requêtes et le choix des techniques d'optimisation appropriées.

Les techniques d'optimisation sont classées en deux grandes catégories [**?**]. Des techniques redondantes et des techniques non redondantes (voir la figure 2.19). Les techniques redondantes, nécessitent un espace de stockage des structures utilisées ce qui induit un coût de mise à jour supplémentaire pour la synchronisation de toutes les copies de la même donnée. Les techniques non redondantes, ne nécessitent pas une duplication de données. Cette propriété de redondance

n'est en aucun cas le critère décisif pour le choix de la technique d'optimisation, cela est dû à la complexité élevée du problème de la conception physique qui intègre beaucoup de contraintes et de la forte interaction entre les différentes techniques d'optimisation.

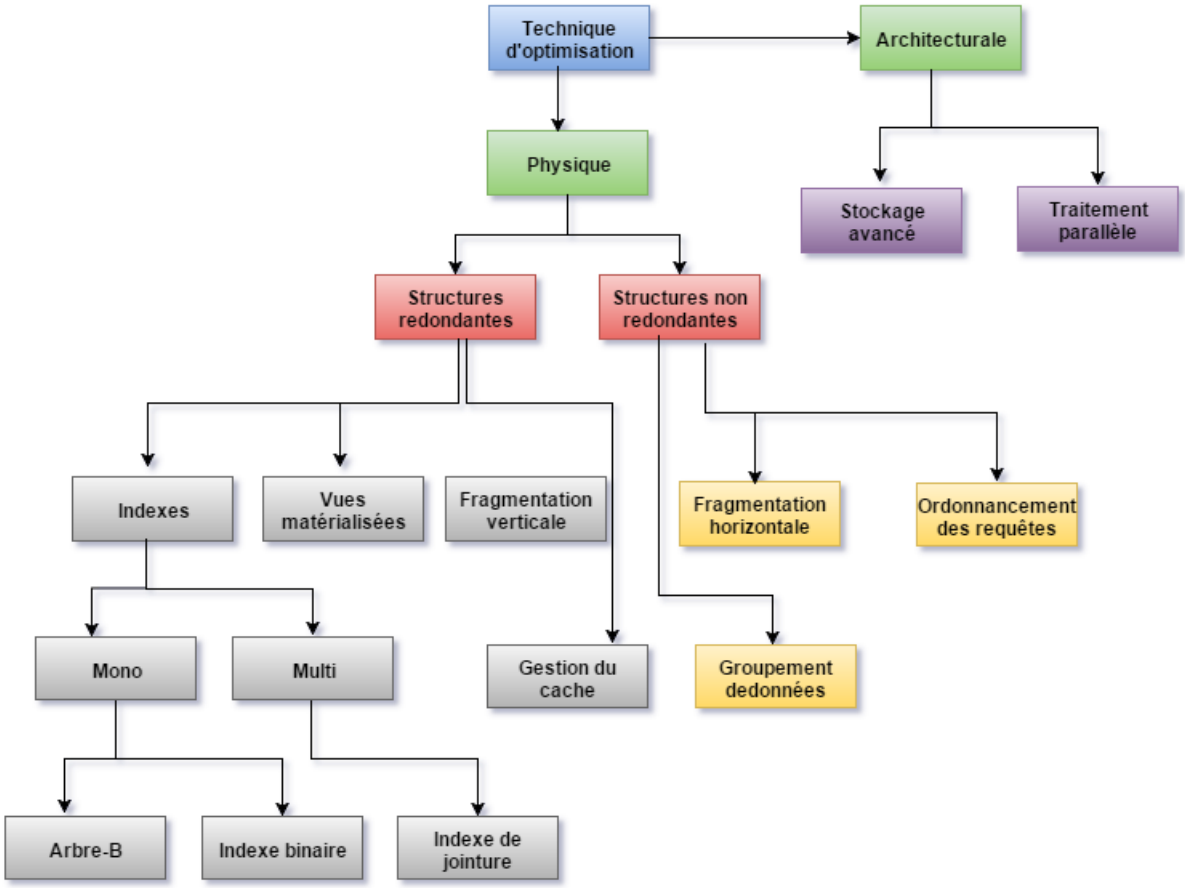


FIGURE 2.19 – Les techniques d'optimisation

Vu la diversité des techniques d'optimisation, nous nous concentrons sur deux techniques redondantes, à savoir les index, les vues matérialisées et la fragmentation verticale, et une technique non redondante qui est la fragmentation horizontale.

Un index est un objet complémentaire à la base de données permettant d'indexer certaines colonnes dans le but d'améliorer l'accès aux données par le $\mathcal{SGBD}$, au même titre qu'un index dans un livre permet souvent d'économiser du temps lorsqu'on recherche une partie spécifique dans ce dernier. En absence de l'index utilisable, une requête portant sur une table nécessitera le balayage total de la table. Toutefois les index sont très gourmands en espace de stockage. Choisir les attributs d'une base de données à indexer est un problème difficile [**?**] connu sous le nom du problème de sélection des index. Traditionnellement, il est formalisé comme suit:

Étant donnés :

— $Q = \{q_1, q_2, ..., q_n\}$, une charge de requêtes où chaque requête q_i possède une fréquence d'accès f_i, $(1 \leq i \leq n)$

— un schéma d'une base de données/entrepôt de données;
— un espace de stockage S ;

Le problème de sélection des index consiste à trouver une meilleure configuration d'index (CI), permettant d'optimiser la charge de requête Q, c'est-à-dire réduire le coût total d'exécution des requêtes tout en respectant la contrainte de stockage ($Taille(CI) \leq S$).

Une large panoplie d'algorithmes basés sur les méta-heuristiques a été proposée pour la résolution de ce problème [**?**].

Il existe plusieurs types d'index : les index en B-arbres [**?**], les index binaires [**?**], les index binaires de jointure [**?**], les index binaires codés par intervalle (range bitmap index) [**?**], les index de projection [**?**], les index binaires dynamiques [**?**], les index de jointure [**?**], etc. Nous détaillons que deux types d'index utilisés dans le contexte des entrepôts de données.

— Les index binaires sont très différents des index classiques comme les B-arbre. Ils sont destinés à être utilisés pour indexer des colonnes de valeurs cardinalité faible. Une colonne de cardinalité faible signifie simplement que la colonne a relativement peu de valeurs distinctes. Par exemple, une colonne, sexe qui ne dispose que de des deux valeurs masculin et féminin comme valeurs possibles est considéré comme de faible cardinalité, car il y a seulement deux valeurs distinctes pour la colonne. Un index binairemap peut être assimilé à une matrice dont les les colonnes représentent les tuples de la tables à indexer. Chaque colonne de cette matrice représente un tuple de la table à indexer. Chaque valeur dans la colonne index correspond à une valeur possible de l'attribut indexé. Si l'intersection d'une colonne j avec une ligne i est égal à 1 cela signifie que j ième tuple correspond à la $i$ème valeur de l'attribut indexé. Un index binaire est crée par la l'instruction SQL suivante:
CREATE BITMAP INDEX <nom de l'index> **ON** <nom de la table> (<attribut indexé>);

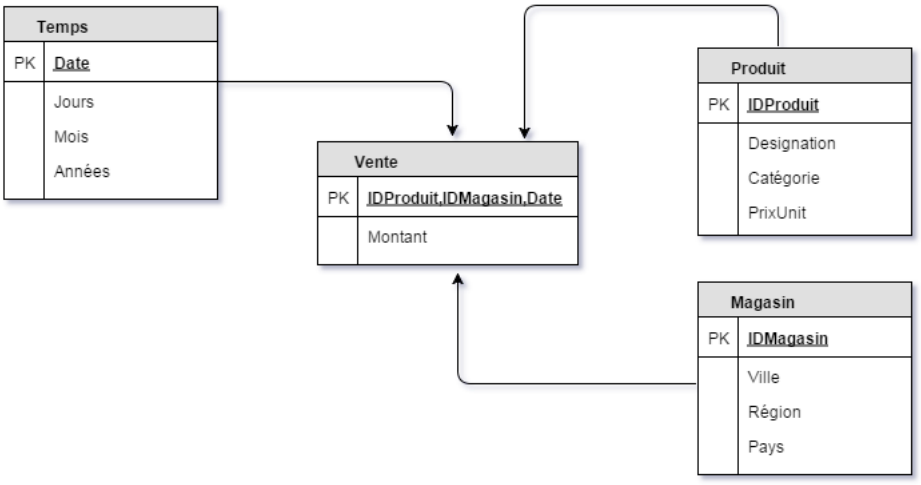


FIGURE 2.20 – Entrepôt de données pour les ventes

Exemple 1

Supposons que la table Magasin de la figure 2.20. Elle comporte 8 enregistrements (comme montré dans le tableau 2.7). Sur cette table, nous pouvons créer un index binaire sur l'attribut région qui possèdent quatre valeurs différentes (sud, nord, est et ouest) à l'aide de la commande suivante:

```
Create bitmap index region_idx  On Magasin(region)
```

*Cette commande crée un index binaire comme le montre la Table 2.7. Si un utilisateur chercher les magasins de la région ouest, il suffit de sélectionner les n-uplets dont le bit correspondant à la valeur **1** pour la 4ième colonnes, à savoir les n-uplets 1 et 8.*

		Bitmap			
IDtuple	Région	Nord	Sud	Ouest	Est
1	O	0	0	1	0
2	S	0	1	0	0
3	N	1	0	0	0
4	S	0	1	0	0
5	N	1	0	0	0
6	E	0	0	0	1
7	E	0	0	0	1
8	O	0	0	1	0

TABLE 2.7 – Index bitmap construit sur la base de region pour la table magasin

— Un index de jointure binaire est un index binaire impliquant plusieurs tables. Il permet de réduire d'une manière significative le stockage des données et de pré-calculer les jointures à l'avance. Pour chaque valeur dans une colonne (attribut indexable) d'une table, l'index stocke les identifiants des n-upets des lignes correspondant à un ou plusieurs autres tables. Dans le contexte des entrepôts de données, il est souvent défini sur la table des faits d'un schéma en étoile [**?**] tout en stockant les identifiants des n-upets de cette table avec les colonnes indexées des tables de dimensions. Un index de jointure binaire peut être représenté par une matrice dont le nombre de lignes correspondent au nombre de n-uplets de la table de faits et les colonnes correspondent au nombre de valeurs distinctes de l'attribut d'index des tables de dimensions. Un élément $index[i, j^{D_{kj}}] = 1$ si le ième tuple de la table de fait joint avec un tuple de la table de dimension et sa valeur est égale à la j ième valeurs de l'attribut indexable de la table de dimension D_{kj}. Dans le cas contraire l'élément $index[i, j^{D_{kj}}] = 1$ de la matrice d'index est égale à 0. La particularité de ces index est qu'ils optimisent à la fois les jointures et les sélections. Pour les créer, il suffit d'écrire la commande suivante:

```
CREATE BITMAP INDEX <nom de l'index>
ON <Table des Faits> (<Tables de Dimensions (Liste d'Attributs))
FROM  TABLES
WHERE <clause de jointure entre les tables>;
```

Exemple 2

Considérons un entrepôt de données dont le schéma est décrit par la figure 2.20. Sur cet

entrepôt une requête OLAP est définie comme suit:

```
SELECT region, count(*)
FROM Vente V,  Magasin M
WHERE V.IDMagasin=M.IDMgasin
AND Region = "Nord"
GROUP BY region
```

Pour exécuter cette requête, une jointure est nécessaire entre la table des faits Ventes et la table de dimension Magasin. Cette jointure peut être coûteuse. Supposons qu'un index de jointure binaire entre les deux tables existe via l'attribut Région (Tableau 2.8).

```
CREATE BITMAP INDEX VenteMagasin_region_idx
ON Vente (Magasin.region)
FROM Vente, Magasin
WHERE Vente.IDMagasin=Magasin.IDMagasin;
```

Avec cet index, il suffit d'accéder à la colonne Nord et compter le nombre de ***1****.*

Produit		
TID	IDProduit	Categorie
7	P1	C4
8	P2	C3
5	P3	C2
4	P4	C1
8	P5	C1
9	P6	C3

Ventes	
TID	IDProduit
50	P5
51	P3
52	P6
53	P2
54	P5
55	P6
56	P3
57	P2
58	P2
59	P5

Index Bitmap de jointure				
TID	C1	C2	C3	C4
50	1	0	0	0
51	0	1	0	0
52	0	0	1	0
53	0	0	1	0
54	1	0	0	0
55	0	0	1	0
56	0	1	0	0
57	0	0	1	0
58	0	0	1	0
59	1	0	0	0

TABLE 2.8 – Index bitmap de jointure construit sur la base de catégorie du Produit sur la table Vente

Les vues matérialisées. Une vue ordinaire stocke le résultat d'une requête en mémoire centrale comme une table virtuelle. Une vue matérialisée est une vue stockée d'une façon permanente sur un support de stockage non volatile. Les vues matérialisées sont très utilisées dans le cadre des entrepôts de données. Elles permettent d'extraire des sous ensembles de données, de faire des agrégations et de stocker les résultats sous forme de cubes de données. Dans la présence de vues matérialisées, les requêtes peuvent accéder indirectement aux données de l'entrepôt via le mécanisme de réécriture de requêtes en sollicitant ces vues. Les vues matérialisées nécessitent un coût supplémentaire pour leur maintenance. Une bonne conception des vues peut améliorer considérablement le temps d'accès des requêtes utilisateurs [**?**, **?**, **?**, **?**, **?**]

La syntaxe de création d'une vue matérialisée est la suivante :

```
Create materialized view [options]
AS requête;
```

Les paramètres d'options permettent de spécifier le mode de maintenance des vues matérialisées après une mise à jour des tables de références de base. Ce mode de maintenance peut être immédiat, différé ou périodique. La maintenance des vues matérialisées peut être aussi faite d'une façon totale (refresh complete) ou incrémentale (option refresh fast). L'avantage des vues matérialisées réside dans le fait qu'elles peuvent être utilisées avec d'autres techniques d'optimisation comme la fragmentation ou les index.

La fragmentation est une technique d'optimisation de performance et de simplification de la maintenance. Elle consiste à diviser un ensemble de données en plusieurs fragments de façon à ce que la combinaison de ces fragments, produit l'intégralité des données source, sans perte ou rajout d'information. Dans le contexte relationnel, ce sont les tables (relations) qui sont partitionnées. Si une grande table est fragmentée en plus petites tables individuelles, les requêtes qui n'accèdent qu'à une fraction de données peuvent être exécutées plus rapidement, car la quantité de données à analyser est amoindrie. Pour une base de données relationnelle, la fragmentation peut intervenir à deux niveaux:

1. durant la conception logique pour le cas des bases de données traditionnelles [**?**];
2. durant la conception physique. Cela est dû à l'explosion du volume des entrepôts de données [**?**].

Deux types de fragmentation existent: la fragmentation horizontale et la fragmentation verticale, que nous allons détailler dans les sections suivantes.

La fragmentation verticale est une technique qui a été initialement proposée pour la distribution des données dans le cadre des bases de données réparties [**?**, **?**, **?**, **?**]. Elle consiste à diviser un schéma de données S en un sous-ensemble de schémas S_i ($i = 1..n$ où n représente le le nombre de sous-schémas) . Les sous-schémas sont obtenus par des opérations de projection selon un ou plusieurs attributs a_j ($j = 1..m$ où m est le nombre d'attributs). L'objectif principal de la fragmentation verticale est de réduire la taille des données chargées en mémoire centrale pour répondre à une requête en regroupant les attributs fréquemment utilisés ensemble dans le même fragment. La fragmentation verticale, crée des fragments avec des tuples de longueur réduite ce qui permet de lire un nombre minime de page disque pour exécuter une requête. La figure 2.21 montre un exemple d'une fragmentation verticale de la table produit de l'entrepôt décrit dans la Figure 2.20.

Pour préserver la propriété de reconstruction discutée dans les paragraphes suivants, il est nécessaire de dupliquer l'attribut clé primaire de la table fragmentée pour pouvoir la reconstruire à partir de ces fragments. La fragmentation verticale est pénalisante dans les cas suivants:

- pour les requêtes qui accèdent à plusieurs fragments de la même table à cause des jointures supplémentaires à effectuer.
- pour les tables de fait si le nombre de dimensions est élevé ce qui revient à dupliquer $(d * f + 1)$ clés où d est le nombre de dimension et f est le nombre de fragments.
- **La fragmentation horizontale** divise une table en plusieurs sous tables appelées fragments horizontaux. Chaque sous table possède le même schéma que la table de base

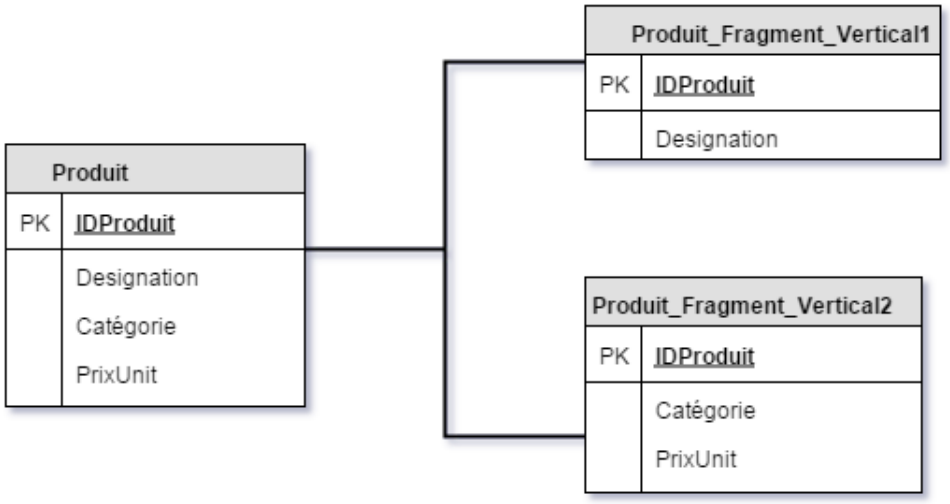


FIGURE 2.21 – Exemple de fragmentation verticale de la table produit

mais ne contient qu'un sous ensemble de tuples. Un fragment horizontal est défini par un **minterm** qui est une conjonction de prédicats simples. Un prédicat simple est de la forme **attribut** θ **valeur** avec $\theta \in \{<, >, <=, >=, =, \text{IN}, \text{NOT IN}\}$ et *valeur* correspond à une valeur ou un ensemble de valeurs prises par l'attribut.

La fragmentation horizontale a été largement étudiée dans le contexte des bases de données traditionnelles et les entrepôts de données [**?**, **?**, **?**, **?**]. Ces travaux de recherches ont beaucoup inspirés les éditeurs de $\mathcal{SGBD}$ commerciaux (Oracle, SQL serveur, etc.) et académiques (PostgreSQL) qui ont proposé des langages de définition de partitions de tables. Plusieurs modes de base de fragmentation existent et sont supportés par ces SGBD :

— La fragmentation par intervalle (Range) définit sur la base de séquences de valeur (intervalles) inclus dans le domaine de définition de l'attribut de fragmentation;
— La fragmentation par hachage (Hash) décompose la table selon une fonction de hachage utilisateur ou fournie par le système, cette fonction est appliquée sur les valeurs des attributs de fragmentation;
— La fragmentation par liste (List) décompose une table selon les listes de valeur d'un attribut;
— La fragmentation composée (Range-Hash et Range-List, etc.).

Deux types de fragmentation horizontale existent : la fragmentation primaire et la fragmentation dérivée. La fragmentation horizontale primaire d'une table est effectuée sur la base de ces propres attributs. La Figure 2.22 représente un exemple de fragmentation horizontale de la table magasin (voir la figure 2.20) sur la base de l'attribut de dimension région. Le premier fragment contient les magasins de la région Poitou-charantes tandis que le deuxième contient tous les magasins de la région Bretagne.

Nous rappelons que les valeurs des clés de la fragmentation sont issues des prédicats simples définis par les requêtes utilisateur les plus pertinentes. La fragmentation horizontale dérivée

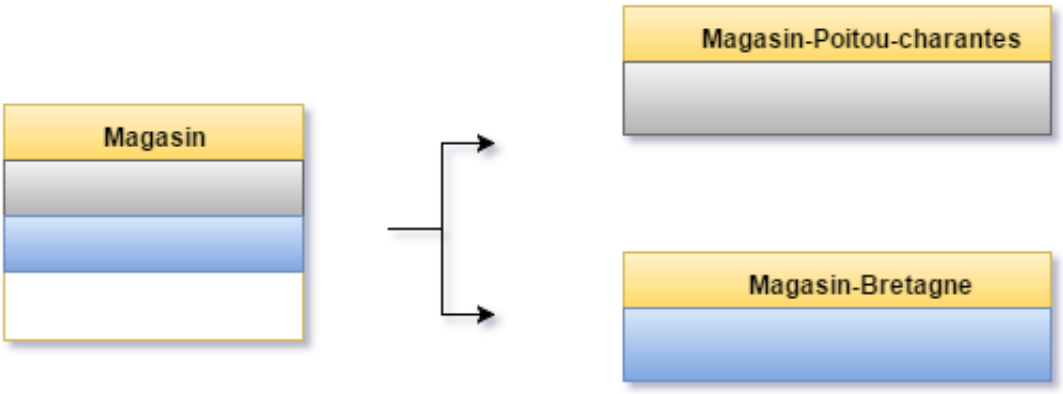


FIGURE 2.22 – Exemple de fragmentation horizontale primaire de la table Magasin

($\mathcal{FHD}$) quant à elle consiste à partitionner une table en utilisant les attributs définis sur une ou plusieurs autres tables. En d'autres termes, cette technique permet de fragmenter une table en fonction de la façon dont une autre table est fragmentée. Naturellement, la $\mathcal{FHD}$ n'est intéressante que dans le cas des tables reliées par une contrainte d'intégrité référentielle. Par conséquent, le stockage à proximité des fragments correspondants des deux tables permettra d'accélérer l'opération de jointure entre les deux tables.

la figure 2.22 montre un exemple de fragmentation horizontale de la table vente en utilisant les fragments horizontaux de la table « Magasin » (voir la figure 2.20).

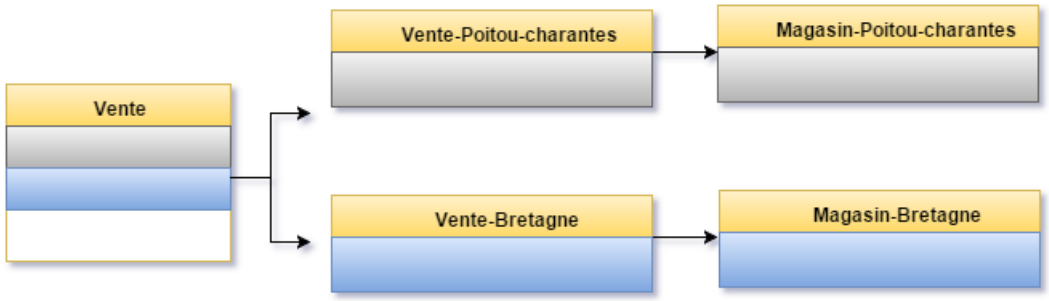


FIGURE 2.23 – Exemple de fragmentation horizontale dérivée de la table « Vente »

La fragmentation doit être sans perte d'information. Elle peut être onéreuse si les besoins des applications sont opposés. Pour effectuer une décomposition correcte, et assurer un résultat cohérent lors de la reconstruction de l'ensemble de données initiale, trois règles principales doivent être respectées:

1. **Complétude**: chaque donnée stockée sur une table T doit appartenir un fragment FT_i. Cette donnée peut exister sur plusieurs fragments dans le cas où une politique de réplication est adoptée.
2. **Reconstruction**: chaque table T fragmentée en un ensemble de fragments FT_i peut être reconstruite par une succession de jointures et d'opération d'unions relationnelles.

3. **Disjonction**: les algorithmes de fragmentation doivent fournir des fragments disjoint deux à deux. Une donnée affectée à un fragment ne doit pas l'être pour un autre fragment.

Le **groupement** et une technique connue depuis longtemps sous le nom de placement à proximité dans le modèle réseau [**?**] Elle permet de regrouper physiquement dans une même page (secteur du disque) ou des pages voisines, les tuples de deux tables (ou plus) ayant des caractéristiques communes (un ou plusieurs attributs en commun). Ces attributs partagés constituent les clés du groupement. Le groupement se fait selon les valeurs des attributs en commun. La technique de groupement peut être bénéfique dans les cas suivants:

— La jointure entre des table placées dans le même groupe
— La sélection des tuples d'une table qui ont la même valeur de clé du groupe
— Gain en espace de stockage car la valeur de la clé du groupe n'est stockée qu'une seule fois.

Toutefois, le groupement peut être pénalisant lors des accès individuels à table toute seule mais qui est groupé avec d'autres tables . Par contre, il est fortement conseillé dans le cas des jointures, car l'opération de jointure entre les tables est déjà faite au niveau de l'organisation physique (tables sur le même groupe). Le groupement de tables est totalement transparent aux utilisateurs qui ne spécifie aucune mention de groupe dans leurs commandes de manipulation de données.

Exemple 3

La commande suivante permet de grouper les tables client et commande selon la clé NumClient qui est clé primaire dans la table client et étrangère dans la table commande.
create cluster *commande_client (NumClient NUMBER(6))*
size *1000*
tablespace *vente*
storage *(INITIAL 500K*
next *400K*
minextents *3*
pctincrease *40);*

ensuite, la création des tables commande et client se fait d'une façon ordinaire en ajoutant la clause **cluster commande_client (NumClient)** dans le code de création des deux tables.

9 Conclusion

Dans ce chapitre, nous avons dans un premier temps décrit l'évolution des bases de données. Plusieurs concepts important ont été abordés. Nous avons présenté le modèle relationnel ainsi que les systèmes les plus importants qui l'utilisent. Ensuite, nous avons présenté les deux modes de représentation des données au sein d'un système de gestion de bases de données à savoir le

stockage orienté ligne et celui orienté colonne. Comme l'efficacité de l'accès aux données au niveau physique est influencée par le support de stockage, nous avons présenté les différents types de supports de stockage de données en particulier les disques dur et les mémoires flash. Nous avons beaucoup insisté sur la mémoire flash car d'un coté elle représente la technologie de stockage la plus récente et qui offre des performances meilleurs que les $\mathcal{HHD}$ et d'un autre coté de sa complexité qui nécessite d'être maîtrisée pour mieux l'intégrer dans le cadre des bases de données. Pour mieux comprendre la mémoire flash nous avons étudié la $\mathcal{FTL}$ qui représente le composant principal qui détermine les performances de ce type de support.

Chapitre 3

Description des Entrées des Modèles de Coût

Sommaire

1	**Introduction**	**65**
2	**Évolution des modèles de coût**	**66**
3	**Catégorisation des Paramètres de Modèles de Coût**	**69**
4	**Méta modèle pour le modèle de coût**	**82**
5	**Instanciation du modèle de coût générique**	**85**
6	**Études de Cas**	**86**
7	**Expérimentation**	**91**
8	**Conclusion**	**93**

1 Introduction

Comme nous l'avons déjà mentionné, la mesure de la qualité des solutions proposées par la phase physique nécessite la présence d'un modèle de coût mathématique. Ce dernier doit prendre en compte les caractéristiques et les paramètres issus de toutes les phases du cycle de vie de conception de l'entrepôt de données, à savoir l'analyse des besoins, les phases conceptuelle, logique, déploiement et physique. Les modèles de coût ont été largement utilisés dans toutes les générations de bases de données [**?**, **?**, **?**]. Leur première utilisation se réduisait à l'identification des plans optimaux de requêtes. Avec l'apparition des applications décisionnelles, la conception physique est devenue un enjeu important, comme l'indique *Dr. Surajit Chaudhuri* [**?**]. Durant cette conception, l'administrateur doit effectuer principalement une sélection d'une ou plusieurs structures d'optimisation (les index, les vues matérialisées, le partitionnement, etc.) *souvent, pour réduire le coût des requêtes*. Vu la taille importante de l'espace de recherche de chaque problème lié à la sélection d'une ou plusieurs structures d'optimisation, des méta heuristiques comme les algorithmes génétiques [**?**], le recuit simulé [**?**] dirigées par des modèles de coût ont été proposées. La troisième utilisation des modèles de coût concerne les étapes de la phase de déploiement de bases de données parallèles: la fragmentation, l'allocation, l'équilibrage de charges, etc.

Souvent, tout modèle de coût est développé en fonction des composantes principales d'un $\mathcal{SGBD}$ [**?**], d'où sa décomposition en trois parties : (1) le coût des entrées-sorties (IO) pour l'échange de données (lecture/écriture) entre la mémoire et le support de stockage comme les disques $\mathcal{HDD}$ ou $\mathcal{SSD}$, (2) le coût $\mathcal{CPU}$ et (3) le coût de communication sur le réseau $\mathcal{COM}$. Si les données sont réparties sur un réseau, le coût est souvent exprimé en fonction de la quantité totale des données transmises sur le réseau [**?**]. Il dépend alors, de la nature de la plate-forme de déploiement [**?**]. Notons que les différentes parties de modèle de coût dépendent fortement des supports de stockage et la politique de gestion du cache [**?**].

Un modèle de coût peut être vu comme une fonction ayant des entrées et une sortie. Les entrées représentent à la fois le schéma de l'entrepôt de données, la charge de requêtes et les paramètres de la phase de déploiement. Nous distinguons deux catégories de paramètres : *les paramètres calculés* (ex. les facteurs de sélectivité des prédicats) et les *paramètres non calculés* (p. ex. la taille des tables ou fragments). L'obtention des paramètres calculés se fait à travers la définition de formules mathématiques. Ces dernières sont également nécessaires pour décrire les coûts des opérateurs locaux ainsi que le coût global. Pour calculer ces derniers, nous avons besoin de la politique de traitement de requêtes (ordonnancement ou pas) et la gestion du cache du $\mathcal{SGBD}$. En sortie, nous aurons un coût final d'exécution d'une charge de requêtes exécutée sur un entrepôt de données.

Souvent, le point de départ du processus de définition de modèles est un ensemble de paramètres quelque soit leur origine (le modèle logique, physique, plate-forme de déploiement, etc.). Cela rend difficile la définition, la compréhension et la réutilisation des modèles de coût

existants. Pour rendre notre modèle de coût modulaire et flexible, nous proposons de catégoriser les paramètres, où ces derniers sont partitionnés en plusieurs entités, chacune est associée à des entrées du problème de la conception physique.

Ce chapitre est consacré à la méta modélisation de l'ensemble des paramètres de modèles de coût dans le but de les rendre génériques. Cette modélisation est similaire à celle utilisée dans la phase de construction d'entrepôt de données. Un autre aspect important concerne l'*ontologisation* de certains concepts utilisés par les modèles de coût (le cas des supports de stockage). Après le processus de la méta modélisation, nous proposons une généralisation des modèles de coût. Des exemples d'instanciation de notre modèle de coût générique sont donnés. Finalement, nous considérons un exemple d'usage de notre modèle de coût pour le problème de la gestion du cache et l'ordonnancement de requêtes.

2 Évolution des modèles de coût

En informatique, une discipline est souvent influencée par les progrès réalisés pour d'autres disciplines. Le développement de modèles de coût a toujours suivi l'évolution des bases de données (voir figure 3.1). Dans la première génération de $\mathcal{SGBD}$, les modèles de coût étaient simples. Cela est dû à la simplicité des requêtes définies dans le contexte de bases de données traditionnelles. Souvent, ces modèles estiment le nombre d'entrées/sorties lors de l'exécution d'une requête [**?**].. Pour illustrer cette situation, considérons l'exemple suivant.

Exemple 4

Soit la requête suivante qui parcours l'ensemble des tuples d'une table T d'un schéma d'une base de données.

```
SELECT * FROM T
```

Si nous nous intéressons au coût de cette requête en termes de nombre de pages qui transitent entre la mémoire centrale et le disque (le support de stockage de la table T), nous pouvons facilement l'estimer à l'aide de l'équation suivante:

$$|T| = \frac{||T|| * long(tuple)}{PS} \quad (1)$$

où $||T||$, $long(tuple)$ et PS représentent respectivement, le nombre de tuples de la table T (qui provient de la phase logique), longueur de chaque tuple (provenant le phase conceptuelle) et la taille de la page disque (provenant de la phase de déploiement).

Après l'émergence des bases de données volumineuses et la nécessité de sélectionner des structures d'optimisation, d'autres paramètres ont été pris en compte lors de l'élaboration des modèles de coût.

Exemple 5

Prenons le cas d'un index de jointure binaire souvent utilisé dans le processus d'optimisation de requêtes. Dans [?], l'espace de stockage pour un index de jointure binaire simple (concernant un seul attribut) IJB_j est calculé comme suit :

$$Storage(IJB_j) = (\frac{|A_j|}{8} + 8) \times |F| \tag{2}$$

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 en présence de IJB_j est:

$$Cost(Q, IJB_j) = log_m(|A_j|) - 1 + \frac{|A_j|}{m-1} + d\frac{||F||}{8PS} + ||F||(1 - e^{-\frac{Nr}{||F||}}) \tag{3}$$

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.

La troisième génération a introduit des paramètres liés à l'architecture de déploiement tels que les systèmes distribués, parallèles, les grappes de machines et le cloud [**?**, **?**, **?**]. Dans le contexte des bases de données déployées sur une plate-forme parallèle, certaines métriques ont été définies.

Exemple 6

Prenons l'exemple du problème de surcharge des noeuds d'une machine parallèle. Dans [?], un poids est associé à tout noeud N_i dénoté par $PSUR(N_i)$, représentant la taille d'extra-requêtes qui surcharge un noeud N_i.

Avec le développement de nouveau modèles de stockages tel que le modèle de stockage en colonne, une quatrième génération de modèles de coût a vu le jour et qui a pris en considération des paramètres liés à ce type de stockage. Parmi les recommandations du rapport de Beckman sur la recherche en bases de données publié en 2016 est le développement de modèles de coût pour les nouveaux systèmes de stockage [**?**].

Après l'émergence de matériel de stockage comme les mémoires flash, quelques efforts ont été menés pour développer des modèles de coût des opérations de base de requêtes comme la jointure. Dans l'exemple suivant, nous présentons un modèle de coût pour estimer l'opération de jointure entre deux tables R et S ; exécutée avec l'algorithme de boucles imbriquées (nested loop). Ce coût est donné par l'équation suivante [**?**]:

$$C_{Flash}(JBIM) = k \times \beta \times C_r(|R| \times |S| + |R|) \tag{4}$$

où k, β, C_r représentent respectivement le nombre de pages à écrire sur la mémoire flash pour écrire une page au sens SGBD, la latence de lecture (les écritures qui peuvent résulter d'une lecture) et le temps nécessaire de lire une page sur la mémoire flash. Ce modèle de coût est

raffiné en prenant en compte la taille du cache, dénotée par B comme suit:
Si (|R| ou |S| < B) alors:

$$C_{Flash}(JBIM) = k \times \beta \times C_r(|R| \times |S|) \quad (5)$$

Cette évolution des modèles de coût nous a motive à décrire d'une manière explicite l'ensemble de leurs paramètres ensuite les factoriser relativement à l'ensemble des couches de bases de données (requêtes, schéma de la bases de données, structures d'optimisation, architecture de déploiement) comme montré dans la figure 3.2. Cette généralisation nous permettra d'instancier tout modèle de coût.

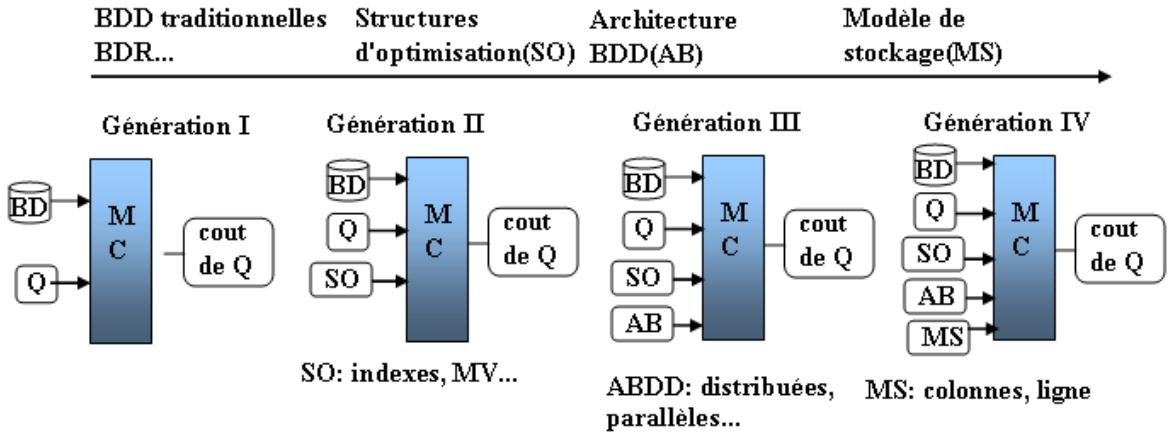


FIGURE 3.1 – Génération des modèles de coût

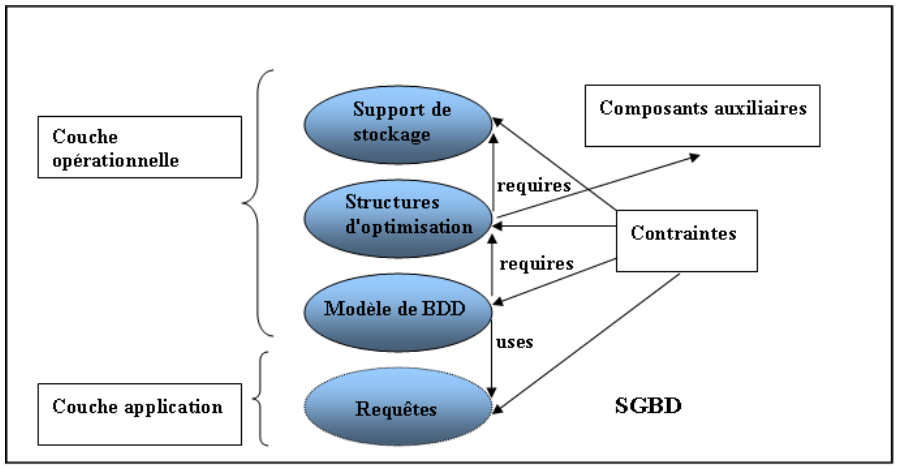


FIGURE 3.2 – Couches à prendre en charge par un modèle de coût

3 Catégorisation des Paramètres de Modèles de Coût

Dans cette section, nous motivons la catégorisation des paramètres utilisés dans tout modèle de coût mathématique. Avant de détailler ce processus, nous souhaitons partager avec le lecteur une expérience que nous avons eue récemment dans le développement de modèle de coût.

Dans le cadre de la thèse d'Amine ROUKH effectuée au laboratoire, nous voulons modifier le SGBD PostgreSQL qui est open source pour que son optimiseur prenne en compte la consommation énergétique lors de l'exécution de requêtes [?]. Nous avons alors récupéré le modèle de coût utilisé par l'optimiseur de requêtes de PostgreSQL (nommé **costsize.c**). Dans ce programme en langage C, les paramètres utilisés par le modèle de coût ne sont pas explicites, ils sont éparpillés dans chaque fonction de ce programme. Nous avons passé un temps considérable pour les comprendre et surtout identifier leurs entités. Voici un cliché de ce programme:

```
double seq_page_cost = DEFAULT_SEQ_PAGE_COST;
double random_page_cost = DEFAULT_RANDOM_PAGE_COST;
double cpu_tuple_cost = DEFAULT_CPU_TUPLE_COST;
double cpu_index_tuple_cost = DEFAULT_CPU_INDEX_TUPLE_COST;
double cpu_operator_cost = DEFAULT_CPU_OPERATOR_COST;

int effective_cache_size = DEFAULT_EFFECTIVE_CACHE_SIZE;

Cost disable_cost = 1.0e10;

bool enable_seqscan = true;
bool enable_indexscan = true;
bool enable_indexonlyscan = true;
bool enable_bitmapscan = true;
bool enable_tidscan = true;
bool enable_sort = true;
bool enable_hashagg = true;
bool enable_nestloop = true;
bool enable_material = true;
bool enable_mergejoin = true;
bool enable_hashjoin = true;
```

Devant cette situation, il est difficile de les comprendre et surtout réutiliser les modèles de coût existants. Des efforts de modélisation et méta-modélisation sont nécessaires. Dans les sections suivantes, nous décrivons l'ensemble des entités que nous considérons importantes dans la définition et l'élaboration des modèles de coût mathématiques.

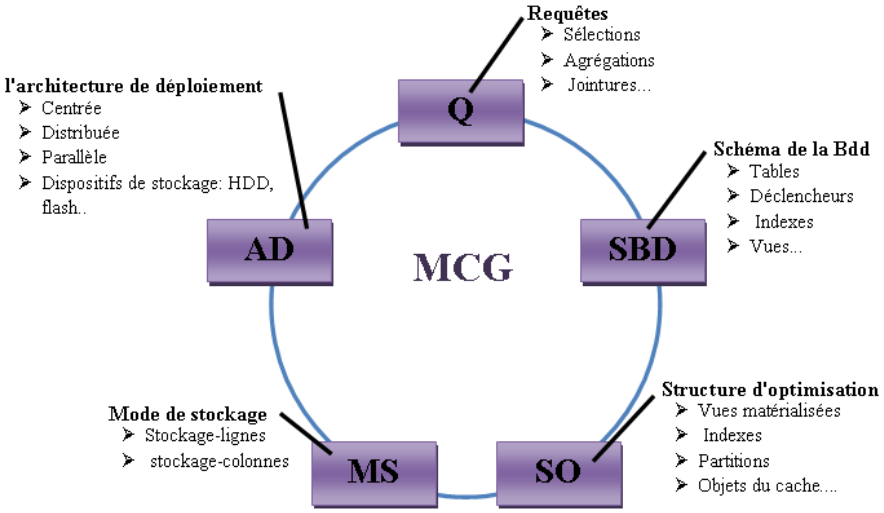


FIGURE 3.3 – Composantes du modèle de coût générique

3.1 Paramètres de l'entrepôt de données

Nous considérons les entrepôts de données modélisés par un schéma en étoile $\mathcal{DWS}$ composés d'une seul table de fait $\mathcal{F}$ et d tables de dimension $\mathcal{D} = \{D_1, D_2, \ldots, D_d\}$. Le domaine de chaque attribut $A_i (1 \leq i \leq n_{a_i})$ est décomposé en un ensemble de sous-domaines stables.

Deux paramètres de données sont importants pour chaque table T (F ou $D_i (1 \leq i \leq d)$). Le *nombre de n-uplet*, noté par $||T||$ et la *taille réelle* des tables mesurée par octets et notée par $|T|$.

La taille de l'entrepôt de données, en termes de tuples, est notée par $||ED||$; elle est égale à la somme des tailles des tables de dimensions et de la taille de la table de faits.

$$||ED|| = ||F|| + \sum_{j=1}^{d} \left\|D_j\right\|.$$

Nom de paramètre	Description
D_j	une table de dimension D_i
F	la table des faits F
d	nombre des tables de dimensions
$\left\|T_j\right\|$	la taille (la cardinalité) d'un tuple d'une table (fait ou dimension)
$\|T_j\|$	le nombre de pages stockant la table T_j
$\|\|ED\|\|$	la taille, en tuples, de l'entrepôt de données

TABLE 3.1 – Les paramètres de l'entrepôt de données.

A partir de ces paramètres, nous proposons un méta modèle représentant tout entrepôt de données (Figure 3.4).

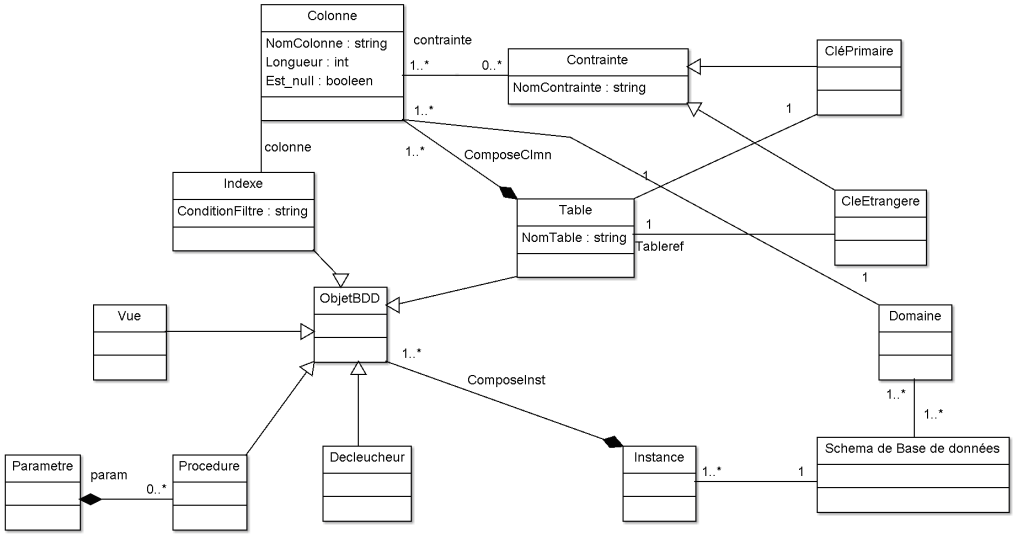


FIGURE 3.4 – Méta modèle de la base de données

3.1.1 Paramètres liés aux requêtes

Nous considérons une charge de requêtes de jointure en étoile composées d'un seul bloc, numérotées de Q_1 à Q_k. Ces requêtes exploitent la totalité du schéma de l'entrepôt par des opérations de sélection, de jointure et d'agrégation. Chaque requête Q_i a une fréquence d'accès f_i et contient un ensemble de prédicats de sélection $\mathcal{PSEL}$= $\{PSEL_1, PSEL_2, \ldots, PSEL_Z\}$, des prédicats de jointure $\mathcal{PJOIN}$= $\{PJOIN_1, PJOIN_2, \ldots, PJOIN_X\}$ et des fonctions d'agrégations (MIN, MAX, $COUNT$, SUM, AVG,...).

Nom de paramètre	Description
Q	Charge de requêtes
Q_i	une requêtes OLAP
k	nombre de requêtes dans Q
f_l	fréquence d'accès de la requêtes Q_l
$PSEL_i$	prédicat de sélection
$PJOIN_i$	prédicat de jointure
FS	facteur de sélectivité d'un prédicat
$Cost(Q_k)$	coût d'exécution de la requête Q_k

TABLE 3.2 – Les paramètres des requêtes

Chaque prédicat ($PSEL_i$ ou $PJOIN_j$) est caractérisé par son facteur de sélectivité $\mathcal{FS}$ que nous détaillons dans le paragraphe suivant.

3.1.1.1 Estimation de la sélectivité des prédicats

La sélectivité des prédicats (de sélection et de jointure) est un paramètre primordial pour estimer le coût des requêtes [**?**].

Définition 3

La sélectivité est un coefficient représentant le nombre d'objets sélectionnés rapporté à un nombre total de tuples d'une table. Si la sélectivité vaut 1, tous les objets sont sélectionnés. Si elle vaut 0, aucun objet n'est sélectionné.

De nombre de travaux ont été effectués pour estimer la sélectivité [**?**]. La plupart d'entre eux supposent une distribution uniforme des valeurs des attributs et une indépendance entre les attributs de chaque relation. Cette estimation se fait de la manière suivante :

Pour deux attributs A_i et A_j d'une relation R, la sélectivité est estimé par :

$$Sel(A_i = valeur) = \frac{1}{card(\pi_{A_i}(R))}$$

$$Sel(A_i > valeur) = \frac{max(A_i) - valeur}{max(A_i) - min(A_i)}$$

$$Sel(A_i < valeur) = \frac{valeur - max(A_i)}{max(A_i) - min(A_i)}$$

$$Sel(p(A_i) \wedge p(A_j)) = Sel(p(A_i)) * Sel(p(A_j))$$

$$Sel(p(A_i) \vee p(A_j)) = Sel(p(A_i)) + Sel(p(A_j)) - (Sel(p(A_i)) * Sel(p(A_j))$$

$$Sel(A_i \in \{valeurs\}) = Sel(A_i = valeur) * card(\{valeur\})$$

La taille d'une opération de jointure entre deux tables T_1 et T_2 est estimée en utilisant la formule suivante [**?**]:

$$||T_1 \bowtie_A T_2|| = \frac{||T_1|| \times ||T_2||}{max(card(\pi_A T_1), card(\pi_A T_2))}$$

Toutefois, certains travaux ne supposent aucune distribution particulière. Dans ce cas, une approche à base d'histogrammes est proposée dans [**?**].

3.1.1.2 Estimation de la sélectivité des prédicats complexes

Les prédicats complexes sont composés de prédicats conjonctifs (ET) et disjonctifs (OU). Grâce à la forme normale, la sélectivité des prédicats est calculée de proche en proche.

La formule générale pour la conjonction est la suivante:

$$Sel(pred_1\ AND\ pred_2) = Sel(pred_1) * Sel(pred_2|pred_1)$$

$pred_2|pred_1$ veut dire $pred_2$ sachant $pred_1$. Si les deux prédicats sont indépendants on aura $Sel(pred_2|pred_1) = Sel(pred_2)$. C'est toujours l'hypothèse envisagée [**?**]. En effet aucun système n'est réellement capable de tenir compte de la corrélation entre les prédicats.

Pour les disjonctions, la formule générale est la suivante:

$$Sel(pred_1\ OU\ pred_2) = Sel(pred_1) + Sel(pred_2) - Sel(pred_1 ET pred_2)$$

Si $pred_1$ et $pred_2$ sont mutuellement exclusifs, on a $Sel(pred_1\ ET\ pred_2) = 0$.

Le méta modèle de la Figure 3.5 correspond à la partie requêtes.

3.2 Paramètres liés aux Structures d'Optimisation

Dans le Chapitre 2, nous avons répertorié deux catégories de structures d'optimisation : les structures redondantes et non redondantes. Un autre point important correspondant aux structures d'optimisation est la nécessité d'avoir un algorithme pour les sélectionner pour une charge de requêtes et des besoins non fonctionnels. Étant donnée la complexité du problème de sélection de toute structure d'optimisation, l'appel à des méta heuristiques est souvent utilisé. Ces dernières peuvent être classées entre deux catégories [**?**]: les méthodes de trajectoire et les méthodes basées sur une population. Les méthodes de trajectoire manipulent une seule solution à la fois et tentent itérativement de l'améliorer. Elles construisent une trajectoire dans l'espace des solutions en tentant de se diriger vers des solutions optimales. Nous pouvons citer la recherche locale, le recuit simulé [**?**], la recherche tabou [**?**], etc. Les méthodes qui travaillent avec une population de solutions disposent d'une "base" de plusieurs solutions, souvent appelée population. Les algorithmes génétiques sont un bon exemple de ces méthodes. Ces algorithmes utilisent des paramètres pour leur bon fonctionnement. Si nous prenons le cas des algorithmes génétiques, nous identifions un nombre important de paramètres comme la taille de la population, nombre total de générations ou critère d'arrêt, probabilités d'application des opérateurs de croisement et de mutation, etc. La figure 3.6 présente un méta-modèle des paramètres des algorithmes de sélection des structures d'optimisation.

3.3 Paramètres liés à l'architecture de déploiement

Il existe de nombreuses architectures de déploiement de la base de données centralisée, distribuée, parallèle ou architecture cloud. Dans une architecture centralisée, le $\mathcal{SGBD}$ gèrent une seule base de données logique située sur un site et contrôlée par un seul système. Dans l'architecture distribuée, les données sont stockées sur de nombreux sites et chaque site possède une partie de la bases de donnée et aussi de son propre $\mathcal{SGBD}$formé de deux composantes: une pour fonctionner en mode locale et l'autre pour le mode distribué. Ce type de base de données supporte la technique de réplication des données pour améliorer les performances et la fiabilité. L'architecture parallèle a les mêmes caractéristiques que l'architecture distribuée sauf que la base de données est stockée sur le même site. L'utilisation de plusieurs $\mathcal{CPU}\int$ est la technique principale pour l'optimisation des requêtes (voir la figure 3.7).

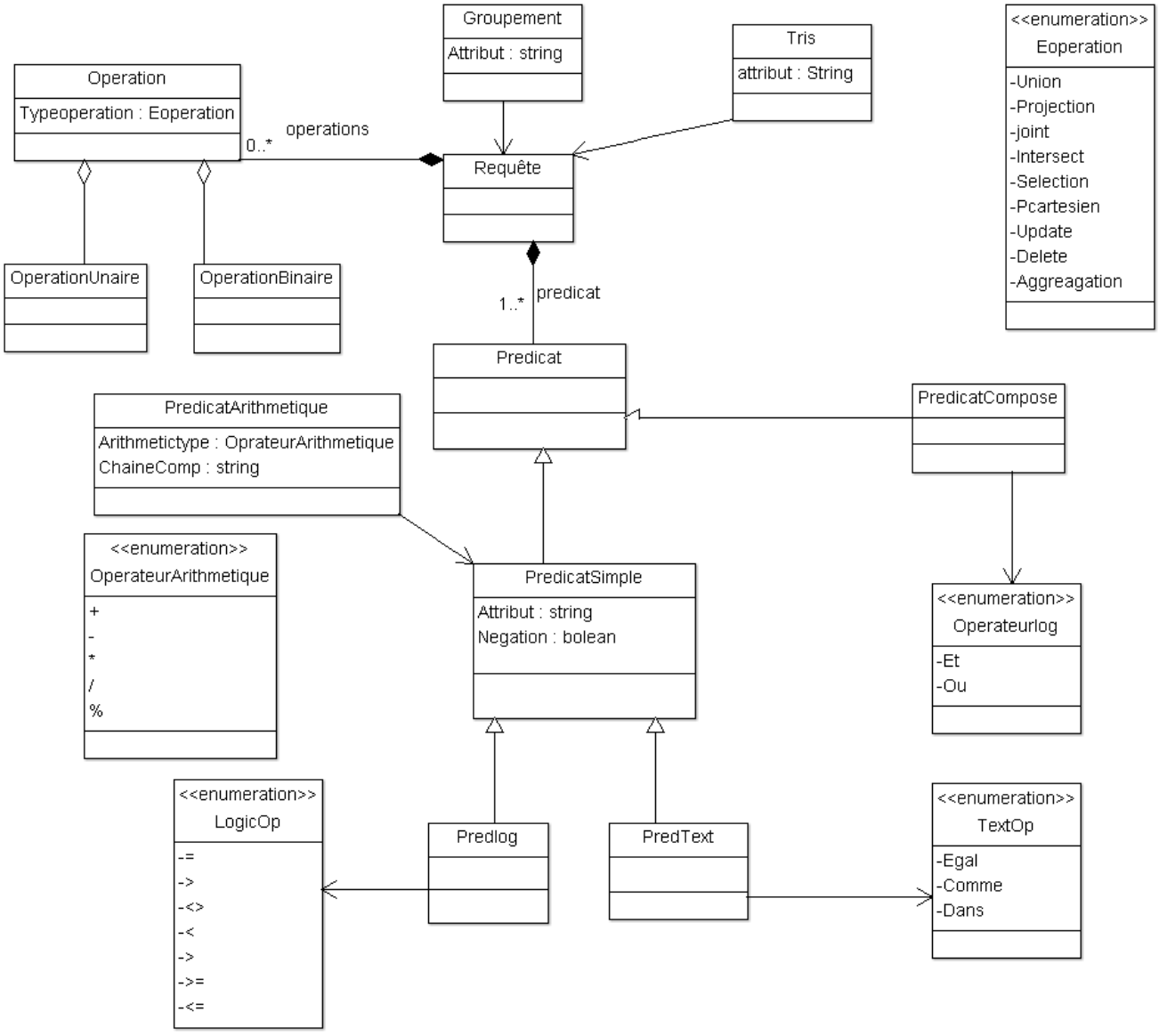


FIGURE 3.5 – Méta Modèle pour les requêtes

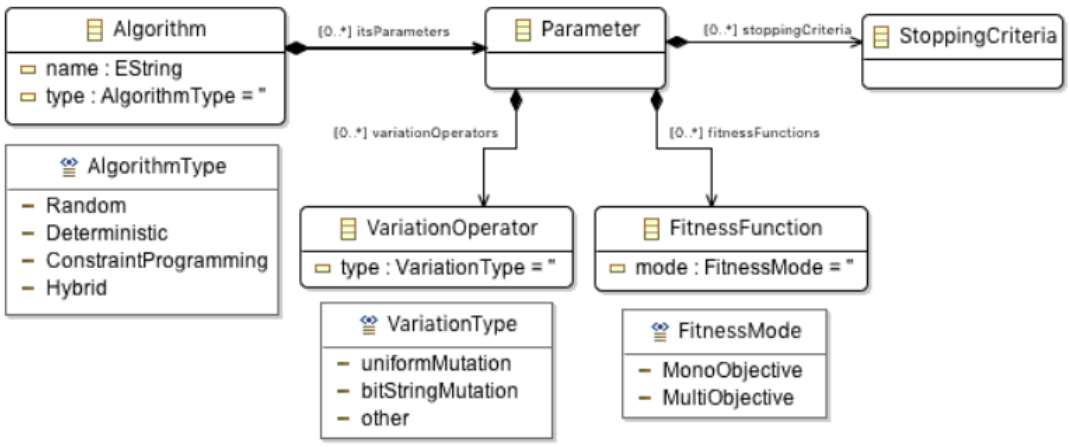


FIGURE 3.6 – Méta modèle pour les structures d'optimisation

3.4 Paramètres liés aux supports de stockage

Les supports de stockage les plus utilisés de nos jours sont les disques dur ($\mathcal{HDD}$) et les disques basés sur les mémoires flash ($\mathcal{SSD}$). Les disques $\mathcal{SSD}$ offrent de meilleures performances pour les accès aléatoires en les comparant avec les $\mathcal{HDD}$. Ainsi, la généricité de notre modèle de coût doit être capable d'exprimer les caractéristiques pour les différents périphériques de stockage. Parmi les inconvénients des $\mathcal{SSD}$, on trouve leur prix qui relativement élevé par rapport aux $\mathcal{HDD}$ [**?**], il est nécessaire de trouver le bon compromis pour utiliser l'une ou l'autre technologie de stockage.

3.5 Paramètres liés au modèle de stockage physique

Il existe deux méthodes de sauvegarde des tuples d'une table sur le support de stockage, le stockage en ligne (row store) et en colonne (column store) comme le montre la figure 3.8. Dans le premier mode, le système de gestion de base de données stocke les tuples d'une table comme une suite de lignes. L'avantage de cette méthode est la facilité d'insertion et de mise à jour des données. Dans le second mode, le système de gestion de base de données stocke la table comme une suite de valeurs (colonne par colonne). L'avantage de cette méthode est que l'une des colonnes peut être aussi un indexe. Ce type de ce stockage est mieux adapté pour les requêtes orienté lecture (parcours de tables) [**?**].

3.6 Une ontologie pour le support de stockage

Plusieurs langages d'ontologies ont été développés ces dernières années. On distingue les modèles d'ontologies supportant la définition des concepts primitifs uniquement (OCC) comme le modèle PLIB (normalisé sous ISO 132584 "parts Library") développé pour le domaine de

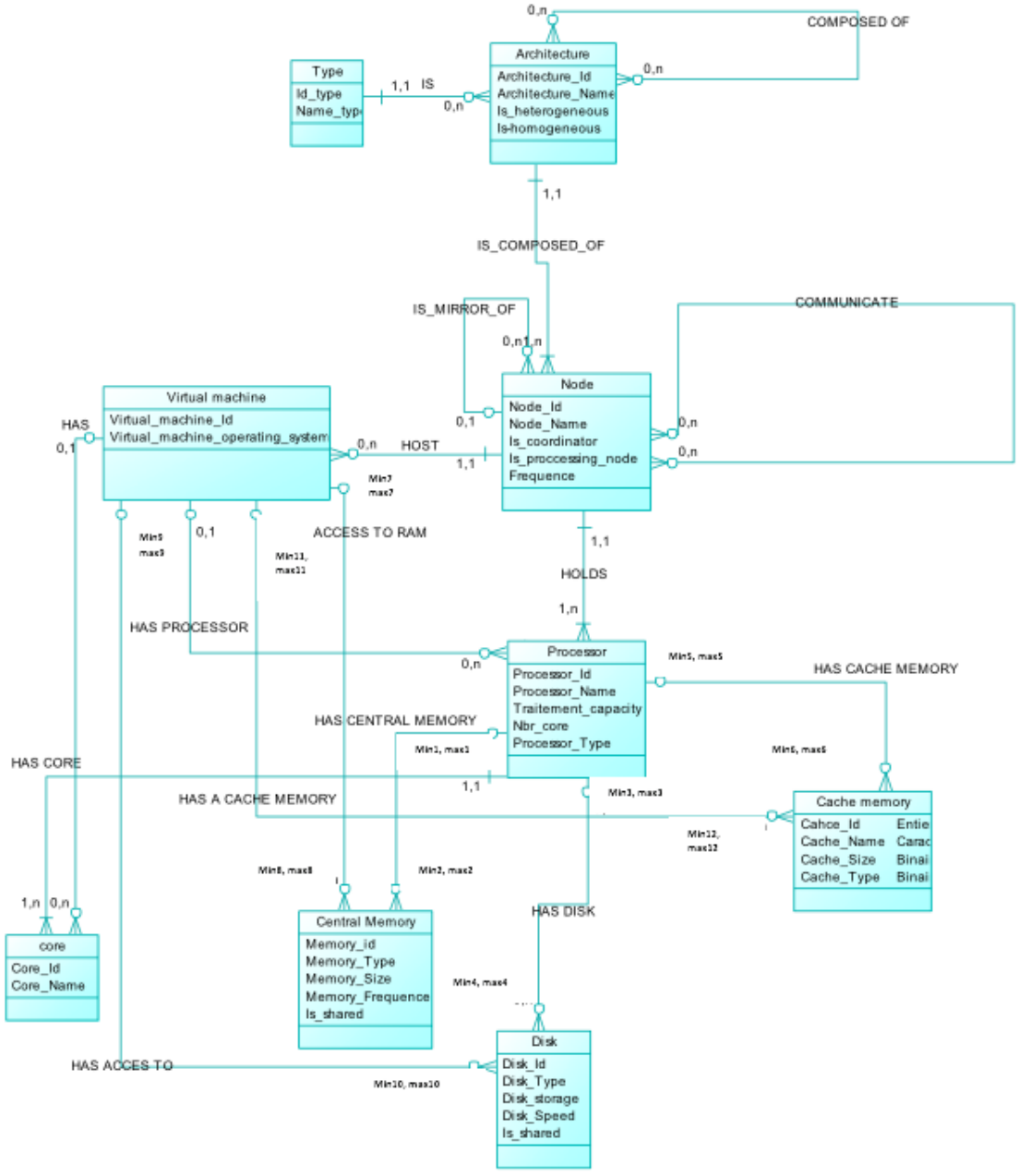


FIGURE 3.7 – Meta-modèle de l'architecture de déploiement

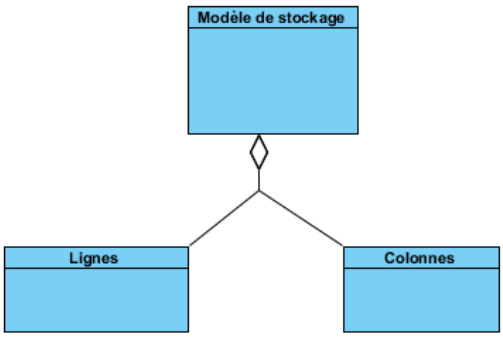


FIGURE 3.8 – Meta-modèle de stockage

l'ingénierie ; et les langages supportant la définition des concepts définis (OCNC) comme les langages du web sémantique RDF, RDF Schema, DAML+OIL et OWL.

— **PLIB**. PLIB est un langage qui a été conçu initialement afin de caractériser de manière précise les produits du domaine technique. Partant du constat qu'on ne peut définir de nouveaux termes qu'à partir de termes primitifs et que les termes primitifs d'un domaine technique sont eux-mêmes très nombreux et difficiles à appréhender ; l'objectif essentiel d'une ontologie PLIB est de définir de la façon la plus précise et la plus concise possible les catégories et les propriétés primitives qui caractérisent les objets d'un domaine du monde réel. Ce modèle permet ainsi de créer des ontologies conceptuelles canoniques en utilisant les constructeurs proposés. Une ontologie PLIB permet la définition de classes, de propriétés, de domaine de valeurs et d'instances, et permet également de définir la hiérarchisation des classes et des propriétés. L'ensemble de ces concepts sont identifiés de manière unique par un GUI (Globally Unique Identifier). Notre laboratoire a participé au processus de développement de PLIB [**?**].
— **Resource Description Framework (RDF)** : l'information sur le Web étant compréhensible seulement au niveau lexical-syntaxique, les outils logiciels indépendants ne sont pas en mesure de traiter l'information en l'absence des méta-informations. Ceci a amené le W3C (World Wide Web Consortium) à élaborer une couche supplémentaire au-dessus de XML appelée Ressource Description Framework (RDF) pour traiter ces méta-informations. Le W3C a développé RDF, un langage d'encodage de la connaissance sur les pages Web pour rendre cette connaissance compréhensible par les agents électroniques qui effectuent des recherches d'informations. RDF permet de décrire des ressources simplement et sans ambiguïté. Toute ressource est décrite par des phrases minimales composées d'un sujet, d'un verbe et d'un complément. On parle alors de déclaration RDF.
— **RDF Schema ou RDF(S)** : le langage RDF n'introduit que très peu de prédicats prédéfinis, il ne propose pas de constructeurs pour la conception d'ontologies. Il a donc été étendu par de nouveaux constructeurs pour permettre la définition d'ontologies, ce qui

a donné lieu au modèle RDF(S). RDF(S) est un des piliers du web sémantique car il permet de bâtir des concepts définis par rapport à d'autres concepts ayant la particularité d'être partagés à travers le web. RDF(S) définit la connaissance d'un domaine sous forme de classes, de sous classes, d'instances de classes et de relations entre les classes. Ce langage est cependant limité et ne permet que la description des vocabulaires simples (pas de cardinalités, deux classes ne peuvent pas avoir des instances communes, pas de liens précis entre les classes et entre les propriétés, etc.). Pour des vocabulaires plus complexes, les langages DAML+OIL ou OWL ont été proposés.

— **DAML+OIL (DARPA Agent Markup Language + Ontology Inference Layer)** : DAML+OIL a été crée suite à la fusion de deux travaux d'équipes de recherche qui sont : DARPA Agent Markup Language (DAML) du ministère de la défense américaine, et Ontology Inference Layer (OIL) développé par la communauté de recherche européenne. DAML est un langage de représentation qui fournit une sémantique formelle pour l'information. OIL a enrichi RDF(S) en offrant de nouvelles primitives permettant de définir les classes comme l'union de classes, l'intersection de classes et le complémentaire d'une classe. La fusion des deux langages a donné lieu à DAML+OIL, écrit en RDF, permettant d'enrichir le pouvoir d'expression de RDF(S). Le W3C a utilisé le DAML+OIL comme base pour la construction de son propre langage d'ontologies : le langage OWL.

— **OWL (Ontology Web Language)** : OWL est un langage de description d'ontologies conçu au départ pour la publication et le partage d'ontologies sur le Web sémantique. Il définit un vocabulaire riche pour la description d'ontologies complexes. Issu des logiques de description, OWL est basé sur une sémantique formelle définie par une syntaxe rigoureuse. Il dispose de fonctions facilitant la réutilisation d'autres ontologies et permet de définir des rapports complexes entre les ressources. Il intègre divers constructeurs sur les propriétés et les classes comme l'identité, l'équivalence, le complément d'une classe, les cardinalités, la symétrie, la transitivité, la disjonction, etc. OWL se présente en trois sous langages à expressivité croissante : OWL Lite, OWL DL et OWL Full. OWL DL est défini comme une extension d'OWL Lite, et OWL Full comme une extension d'OWL DL.

Plusieurs outils d'édition d'ontologie existent: on peut citer : Apollo [12]; OntoStudio [13] ; Protégé [14]; PLIBeditor.

3.7 Processus de construction d'ontologie

Comme tout produit informatique, la construction d'ontologie passe par un cycle bien identifié. Il existe deux grandes catégories de méthodes de création d'ontologie qui s'appliquent

12. http://apollo.open.ac.uk/
13. http://www.semafora-systems.com/en/products/ontostudio/)
14. (http://protege.stanford.edu/

dans des cas différents. Une partie des méthodes disponibles se sert de documents et d'informations qui existent déjà, elles visent alors à extraire des informations de l'existant. Les autres méthodes encadrent la démarche de création à partir de rien, p. ex. : avec des connaissances implicites du domaine. La première est aussi la plus ancienne, cette méthode globale est réalisée en plusieurs étapes.

La première étape consiste à analyser le domaine étudié. Cette analyse est commune à toutes les méthodes. Elle consiste à définir le périmètre de l'ontologie. En effet le but de l'ontologie étant d'être exhaustif sur un domaine, cette étape est cruciale pour dimensionner l'ontologie. Elle implique de faire des choix d'experts sur les éléments à inclure. Une fois le périmètre établi la définition de la granularité de l'ontologie est recommandée, c'est la deuxième étape du dimensionnement. Il s'agit là de décider de la précision des concepts utilisés, ce qui déterminera le moment à partir duquel un concept est considéré comme atomique (primitif). Cette fois il est nécessaire de faire intervenir les experts du domaine mais également les futurs utilisateurs pour s'assurer que les concepts seront exploitables par tous. La seconde étape est la conceptualisation du domaine. Elle est directement liée à l'usage qui sera fait de l'ontologie et elle constitue la continuité des choix précédemment effectués sur la granularité des concepts. La conceptualisation va également demander la création des liens entre les concepts. La dernière étape va davantage concerner l'implémentation ou la représentation de l'ontologie. L'ontologie obtenue doit couvrir l'ensemble des sources, systèmes d'information, utilisateurs, etc. Chacune des étapes citées requiert l'approbation de tous les utilisateurs de l'ontologie, c'est une démarche consensuelle globale. La méthode globale a ouvert la voie à des travaux spécifiques sur la création des ontologies. On remarque qu'elle se base sur deux hypothèses : Le domaine est bien défini et les utilisateurs peuvent aboutir à un consensus global.

Cette seconde méthode est issue de la méthode globale décrite précédemment. Des ontologies locales, spécifiques à des sources ou des usages, sont créées. Toutefois ces ontologies locales partagent un même vocabulaire. C'est-à-dire que si l'intersection des périmètres de deux ontologies locales est non-nul alors les concepts de l'intersection sont les mêmes dans les deux ontologies. Cette méthode est plus souple que la précédente. En effet il est possible de définir localement une ontologie pour appréhender une source de données particulière ou pour satisfaire les besoins spécifiques d'un utilisateur. Toutefois les domaines traités par les ontologies locales peuvent être hétérogènes et donc requérir des concepts spécifiques. On se retrouve alors avec une déclinaison d'un même concept pour décrire précisément le domaine. Ce qui oblige, ensuite, à créer des correspondances entre les concepts nécessaires à chaque ontologie, c'est-à-dire définir les correspondances entre les concepts des ontologies.

La construction d'une ontologie prend énormément de temps car elle doit être consensuelle. Pour assurer cette consensualité, des réunions impliquant l'ensemble de partenaires sont nécessaires afin de dégager les concepts et les propriétés pertinentes au domaine étudie. Nous avons eu l'occasion de participer à la construction d'une ontologie de domaine pour la mobilité éclectique dans le cadre d'une convention CIFRE avec l'entreprise EDF (Electricité de France) [**?**]. Nous avons réduit le temps de construction d'ontologie et surtout donner à EDF la possibilité

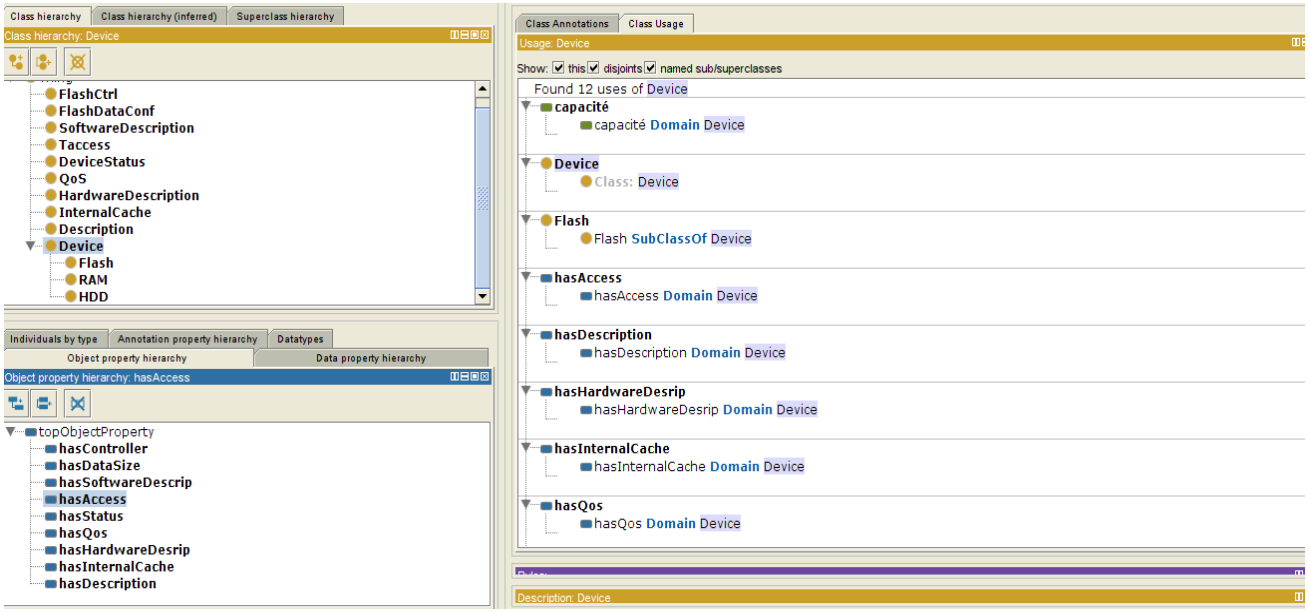


FIGURE 3.9 – Notre ontologie sous protégé

d'utiliser les ontologies, nous avons opté pour une solution par brique ontologique. Elle constitue une ontologie locale autour d'un seul concept, il s'agit de la classe principale de la brique. La classe principale constitue la raison d'être de la brique, l'unique objectif de la brique est de détailler ce concept. L'ontologie globale sera alors construite en composant l'ensemble de briques ontologiques.

Dans notre cas, nous avons opté pour la première approche de construction d'ontologie. La connaissance du domaine de stockage nous a motivé pour considérer cette approche. Pour ce faire, nous avons revu l'ensemble des paramètres associés aux disques durs et les mémoires flashs, et nous avons identifié l'ensemble de classes et les propriétés de notre ontologie.

La Figure 3.9 présente notre ontologie sous Protégé. Un fragment de cette ontologie décrite en OWL est ci-dessus décrit:

```
<?xml version="1.0"?>


<!DOCTYPE Ontology [
<!ENTITY xsd "http://www.w3.org/2001/XMLSchema#" >
<!ENTITY xml "http://www.w3.org/XML/1998/namespace" >
<!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#" >
<!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns#" >
]>

<Ontology xmlns="http://www.w3.org/2002/07/owl#"
```

```
xml:base="http://www.semanticweb.org/admin/ontologies/2016/9/untitled-ontology-18"
xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:xml="http://www.w3.org/XML/1998/namespace"
ontologyIRI="http://www.semanticweb.org/admin/ontologies/2016/9/untitled-ontology-
<Prefix name="rdf" IRI="http://www.w3.org/1999/02/22-rdf-syntax-ns#"/>
<Prefix name="rdfs" IRI="http://www.w3.org/2000/01/rdf-schema#"/>
<Prefix name="xsd" IRI="http://www.w3.org/2001/XMLSchema#"/>
<Prefix name="owl" IRI="http://www.w3.org/2002/07/owl#"/>
<Declaration>
<Class IRI="#Description"/>
</Declaration>
<Declaration>
<Class IRI="#Device"/>
</Declaration>
<Declaration>
<Class IRI="#DeviceStatus"/>
</Declaration>
<Declaration>
<Class IRI="#Flash"/>
</Declaration>
<Declaration>
<Class IRI="#FlashCtrl"/>
</Declaration>
<Declaration>
<Class IRI="#FlashDataConf"/>
</Declaration>
<Declaration>
<Class IRI="#HDD"/>
</Declaration>
<Declaration>
<Class IRI="#HardwareDescription"/>
```

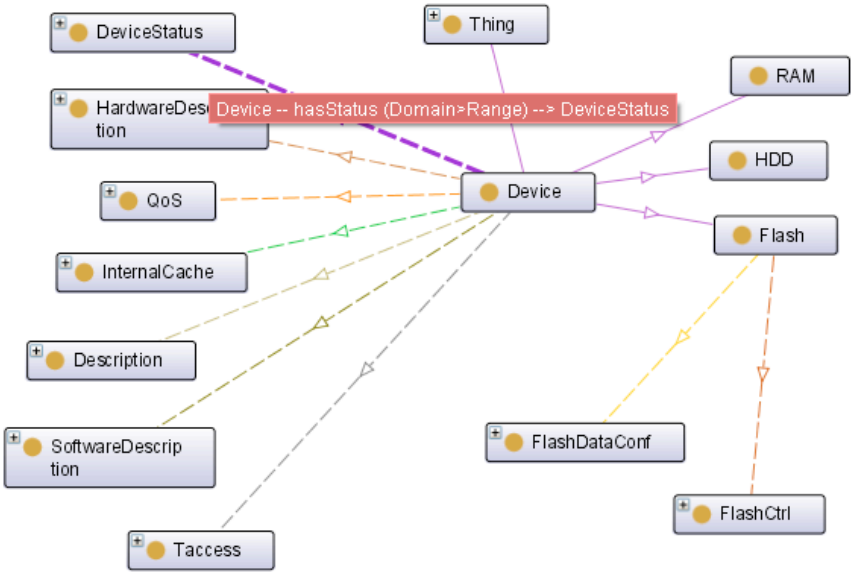


FIGURE 3.10 – Système de stockage hiérarchique

4 Méta modèle pour le modèle de coût

Comme nous l'avons déjà mentionné, un SGBD dans notre cas peut être représenté par 4-uplet suivant (Figure 3.11):

$$DBS = (\{ST_1, \ldots, ST_n\}, OS, SD, \{SD_1, ..., SD_n\}) \quad (6)$$

où: $\{ST_1, \ldots, ST_n\}$ représente un ensemble de structures de stockage, OS est un ensemble d'opération de l'algèbre relationnelle, SD est le support de stockage principal pour héberger le cache (database buffer) et $\{SD_1, \ldots, SD_n\}$ les supports de stockages secondaires. Le support de stockage principal est l'espace mémoire réservé pour le cache dans la mémoire centrale. Les supports de stockage secondaire sont des supports qui se trouvent au deuxième niveau hiérarchique du système de stockage. Ils sont généralement de taille importante mais leurs temps d'accès sont beaucoup moins par rapport au temps d'accès à la mémoire centrale. Notons qu'avec ce modèle, il est facile de modéliser d'autre architectures de bases de données comme les bases de données en mémoire centrale(in-memory databases) bases de données sur architecture raid et des bases de données sur des disques partagés.

Notons que toute *structure de stockage* ST fait l'abstraction du mode de stockage et de gestion des données à l'intérieur du SGBD. Elle peut être une table, une vue matérialisées, un fragment ou un index. Elle contient un ensemble de pages $P_1 \ldots P_n$. Le nombre de pages d'une

structure ST est noté $|ST|$. Sur cette structure, trois fonctions sont définies:

$$rs(ST) = \begin{cases} 1, & \text{si } ST \text{ est stockée en ligne (row-store)} \\ 0, & \text{sinon.} \end{cases} \tag{7}$$

$$cs(ST) = \begin{cases} 1, & \text{si } ST \text{ est stockée en colonnes (column-store)} \\ 0, & \text{sinon.} \end{cases} \tag{8}$$

$$comp(ST) = \begin{cases} 2, & \text{si } ST \text{ est fortement compressée} \\ 1, & \text{si } ST \text{ est faiblement compressée} \\ 0, & \text{sinon.} \end{cases} \tag{9}$$

La fonction K (SD, ST) retourne le pourcentage des pages qui se trouvent sur le cache qui sont alloués sur le support de stockage primaire (mémoire centrale).

L'ensemble des opérateurs OS: $\{O_1, \cdots O_m\}$ représente l'ensemble des opérations de l'algèbre relationnelle. Chaque opérateur $O_i \in OS$ peut être unaire ou binaire selon le nombre d'arguments. Un opérateur unaire a une seule ST_x en entrée et retourne une autre structure ST_y en sortie.

Un opérateur binaire a en entrée deux structures de stockage ST_x, ST_y et retourne une structure ST_z en sortie. Une *requête* peut être représentée par un arbre d'opérateurs O_i avec $O_i \in OS$.

Pour faire abstraction de notre modèle et des nombreux paramètres relatifs à chaque type de support de stockage, il est nécessaire d'avoir des fonctions de coût qui permettent d'estimer le temps de lecture des données à partir d'un support de stockage SD vers la mémoire centrale. Une première alternative consiste à trouver des fonctions analytiques, pour modéliser le temps d'accès, qui intègrent les couches du support de stockage et structure de stockage. Cette approche est contradictoire avec l'aspect générique de notre modèle de coût et nécessite un effort important pour la maintenance du modèle de coût. Nous avons choisi l'approche basée sur la technique de *machine learning* [**?**] [**?**] [**?**] pour estimer les différents paramètres.

Nous définissons les paramètres $T_{read}(ST,SD)$ et $T_{write}(ST,SD)$ représentant respectivement le temps de lecture et écriture de toutes les pages d'une structure de stockage ST pour un support de stockage SD.

$$T_{read}(ST,SD) = L_{read} + |ST| \cdot page_{size} \cdot B_{read} \tag{10}$$

$$T_{write}(ST,SD) = L_{write} + |ST| \cdot page_{size} \cdot B_{write} \tag{11}$$

Il est à noter que $|ST|$ représente le nombre de pages de la structure de stockage ST et qui est calculé par les fonctions de coût relatives à cet effet.

Pour exécuter une requête en utilisant une certaine structure de stockage, un ensemble d'opérations doit être effectué. Il varie selon le plan d'exécution choisi. Plus formellement, pour chaque couple (ST, Q_i) une séquence d'opérations $\{O_1, \ldots, O_m\}$ est générée. Pour deux plans d'exécution différents, l'ordre de l'ensemble des opérations $\{O_1, \ldots, O_m\}$ peut changer

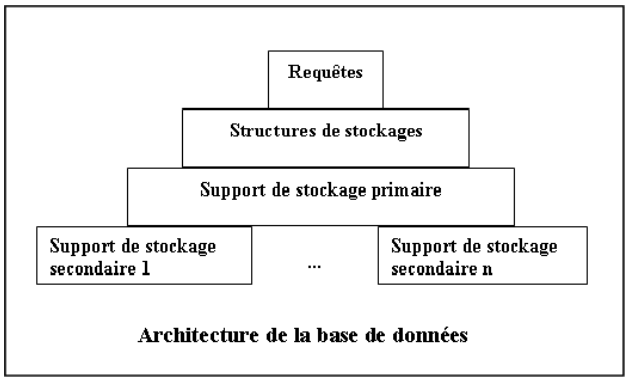


FIGURE 3.11 – Meta-modèle de stockage

aussi.

Les fonctions suivantes nous permettent d'estimer les coûts en termes d'opération d'entrée/sorties (E/S) effectuées pour accomplir l'exécution d'une requête (IO cost). Le coût en E/S d'une requête Q est la somme des coûts E/S pour toutes les opérations de Q $O_i \in Q$.

$$IOCost(Q, ST, SD) = \Sigma_{i=1}^{m} IOCost(O_i, ST_i, SD) \tag{12}$$

où ST_i est l'ensemble des structures de stockage résultantes des $(i-1)$ opérations précédentes. Les valeurs possibles de ST_i sont

$$ST_i = \begin{cases} \{\emptyset, \{ST_x\} \text{ si } O_i\text{est une opération unaire} \\ \{ST_x, ST_y\} \text{ si } O_i \text{ est binaire}\} \end{cases} \tag{13}$$

pour estimer le coût d'exécution d'une opération O_i pour une structure de stockage ST_i, deux cas sont distingués selon le mode de stockage qui peut être en ligne ou en colonne.

$$\begin{aligned} IOCost(O_i, ST_i, SD) = {} & rs(ST) \cdot IOCost_{row}(O_i, ST_i, SD) \\ & + cs(ST) \cdot IOCost_{col}(O_i, ST_i, SD) \end{aligned} \tag{14}$$

où $IOCost_{row}(O_i, ST_i, SD)$ et $IOCost_{col}(O_i, ST_i, SD)$ sont définis directement par la fonction de coût générale $IOCost(O_i, ST_i, SD)$. Le nombre de pages occupées par une structure de stockage $|ST_i|$ se calcule par la fonction de coût suivante:

$$|ST_i| = \frac{\#rows(ST_i) \cdot avgSize(c_i)}{pagsize} \tag{15}$$

pour un stockage en mode colonne, où c_i est la colonne impliquée dans la structure ST_i utilisée

par l'opération O_i, et $avgSize(c_i)$ est la taille moyenne des valeurs de la colonne c_i et

$$|ST_i| = \frac{\#rows(ST_i) \cdot avgSize(r_i)}{pagsize} \tag{16}$$

pour un stockage en mode lignes, où $avgSize(r_i)$ est la taille moyenne du tuple r_i.

Un des avantages de notre modèle de coût est qu'il permet de développer des algorithmes génériques pour traiter différents problèmes relatifs à la conception physique de bases de données. Un algorithme doit simplement spécifier la structure d'optimisation à utiliser (La Figure 3.12). Dans cette section nous présentons le procédé pour exploiter ces modèles pour définir des modèles de coût.

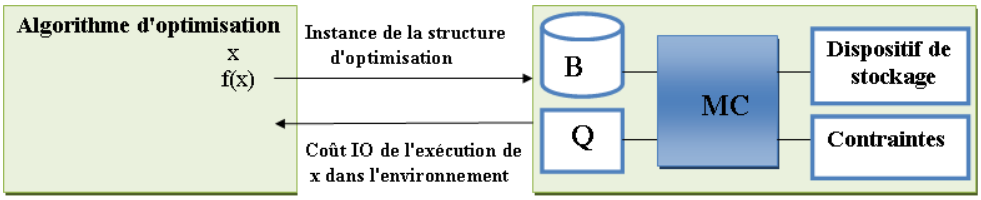


FIGURE 3.12 – Utilisation du modèle de coût générique par les algorithmes

5 Instanciation du modèle de coût générique

La modélisation générique permet de réduire le temps de développement des applications et de fournir beaucoup de flexibilité dans des situations différentes répondant aux changements des besoins et la diversité des environnements. Elle permet aussi de tester les modèles par des outils de simulations avant leurs implémentations dans des environnements réels. Pour montrer l'applicabilité de nos modèles, nous allons, dans cette section, montrer comment instancier notre modèle de coût générique pour deux SGBD *Open Source* avec des caractéristiques différentes à savoir Postgres [**?**] et MonetDB [**?**].

5.0.0.3 PostgreSQL Dans ce scénario nous supposons que nous disposons d'un disque dur comme support de stockage secondaire. Le SGBD Postgres supporte deux types d'index: l'index B^+ et l'index hash. Le SGBD Postgres est orienté lignes, la fonction *rs (Table)* est évaluée à vraie pour toutes les structures de stockage (table, index, vues, ..., etc.) et la fonction *rs (Table)* est toujours évaluée à faux. Le mode de stockage de Postgres supporte la compression de données ce qui fait que la fonction *comp (Table)* peut être évalué à vrai ou faux. Les index de type B^+ et les indexes hash ne supportent pas la compression de données. Alors:

7. `http://www.postgresql.org/docs/current/static/storage-toast.html`

$$\begin{aligned} Postgres =&(\{Table, HashIndex, B^{+}Index\},\\ &\{\sigma, \pi, \bowtie, \cup, \cap, groupby, sort, aggregate\},\\ &MainMemory, \{HardDiscDrive\}) \end{aligned}$$

avec

$$\begin{aligned} comp(HashIndex) = comp(B^{+}Index) = 0\\ rs(Table) = 1 \text{ and } cs(Table) = 0 \end{aligned}$$

5.0.0.4 MonetDB Dans ce cas, nous supposons que le stockage en mémoire secondaire se fait sur un disque basé sur les mémoires flash comme les disques SSD. Le SGBD MonetDB supporte le mode orienté colonne (column-store) via des tables d'association binaire ou chaque colonne est stockée sur une table BAT (Binary Association Table) [**?**]. Le SGBD supporte la technique de compression de données, la fonction $comp(BAT)$ peut retourner soit vrai ou faux. On peut alors écrire :

$$\begin{aligned} MonetDB =&(\{BAT\},\\ &\{\sigma, \pi, \bowtie, \cup, \cap, groupby, sort, aggregate\},\\ &MainMemory, \{SolidStateDrive\}\}) \end{aligned}$$

avec

$$rs(BAT) = 0 \text{ et} cs(BAT) = 1$$

6 Études de Cas

Pour montrer l'utilisation et l'efficacité de notre modèle de coût générique, nous considérons deux cas d'étude qui concerne le problème de gestion du cache ($\mathcal{GC}$) et l'ordonnancement de requêtes ($\mathcal{OR}$).

Nous avons choisi ces deux problèmes pour deux raisons principales: (i) ils sont concernés par les deux types de supports de stockage: les disques et le cache et (ii) lors de notre étude, Amira Kerkad faisait sa thèse autour de ces problème [**?**]. Nous voulions alors tester nos modèles. Avant de détailler ces études, nous commentons brièvement les deux problèmes.

6.1 Importance de la gestion du cache et l'ordonnancement des requêtes

la plupart des travaux sur la conception physique des bases de données traitent une seule technique d'optimisation à la fois et ignorent l'interaction entre les requêtes. Dans le cadre des entrepôts de données et en particulier le schéma en étoile, cette interaction est fortement présente. Toutes les requêtes soumises à un entrepôt de données ayant un schéma en étoile, utilisent une jointure entre une ou plusieurs tables de dimensions avec une table de fait. Ces requêtes appelées requêtes de jointure en étoile possèdent souvent des opérateurs en commun. Pour pouvoir réutiliser ces opérateurs en commun, il est nécessaire (1) de trouver le meilleur ordre des requêtes qui optimisent leur utilisation (2) de les matérialiser sur un support de stockage primaire ou secondaire. Cela met en évidence, une interaction entre les requêtes elles même et les requêtes avec les supports de stockage. Cette problématique a été étudiée dans la littérature dans [**?**] pour les vues matérialisées, [**?**] pour la gestion du cache. La combinaison du problème d'ordonnancement des requêtes et la gestion du cache a été étudié dans [**?**] et [**?**].

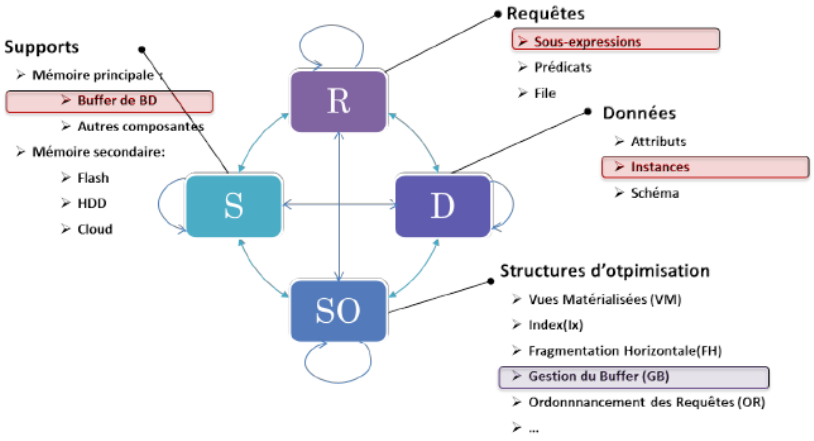


FIGURE 3.13 – Gestion du cache et ordonnancement des requêtes

Le $\mathcal{GCOR}$ est un problème d'optimisation qui a pour but de trouver à la fois une allocation du cache de la base de données et un ordre pour exécuter une charge de requêtes d'une manière optimale. Ces deux problèmes sont NP-complet vu que l'espace de recherche des solutions est très large [**?**]. Dans le cas ou la charge de requêtes est connue à l'avance, la $\mathcal{GCOR}$ peut être résolue par les approches suivantes:

1. Approche séquentielle: dans cette approche, on commence par l'ordonnancement des requêtes. Une fois l'ordre d'exécution des requêtes déterminé, il est passé à l'algorithme de gestion du cache. L'inconvénient de cette approche est que l'ordonnancement est fait sans prise en considération du cache, ce qui donne des solutions non optimales.
2. Approche problème joint: cette approche traite simultanément les deux problèmes de gestion du cache et l'ordonnancement des requêtes. Le problème de gestion du cache et d'ordonnancement des requêtes ($\mathcal{GCOR}$) peut être traité par les méthodes suivantes: (1)

Utilisation d'algorithmes heuristiques qui permettent de parcourir l'espace de recherche combiné des deux problèmes. Dans ce cas, l'espace de recherche des deux problèmes est très large. Ceci est dû à l'interaction des différentes solutions entre-elles. Cette approche peut être utilisée en mode déconnecté (heure de maintenance de la base de données, etc.) car en temps réel, elle est gourmande en temps de calcul. (2) Cette approche se base sur deux composantes, un algorithme d'ordonnancement ($\mathcal{AO}$) et une politique de gestion de cache ($\mathcal{PGC}$) qui permet de gérer l'allocation et le remplacement des données dans le cache. L'algorithme d'ordonnancement($\mathcal{AO}$) s'appuie sur la PGC pour diriger son processus d'ordonnancement (selon le contenu du cache).

3. La troisième approche est similaire dans son principe à l'approche précédente. Dans la réalité les objets du cache ne sont pas uniformes. Ils sont différents en terme de taille, fréquence, coût de chargement du disque et coût de calcul en mémoire centrale. La PGC prend en considération ces différences entre les objets du cache pour élire ou libérer un objet candidat. Cette approche est plus efficace que les autres car, elle traite le problème d'une façon plus objective en intégrant tous les paramètres nécessaires.

6.2 Formalisation du problème

Pour formaliser le problème de gestion du cache de bases de données et l'ordonnancement de requêtes, nous considérons les hypothèses suivantes:
(1) la connaissance préalable de la charge de requêtes (ordonnancement hors-ligne), **(2)** un environnement de bases de données centralisé $\mathcal{RDW}$ et **(3)** le cache est initialement considéré vide. La figure 3.13 donne une instanciation de nos méta modèles pour le problème de gestion du cache et l'ordonnancement des requêtes.

Pour représenter la charge de requêtes et faciliter leur manipulation ainsi que les objets du cache, nous utilisons la représentation proposée par Sellis et al.[**?**]. Cette représentation consiste à fusionner tous les plans d'exécution des requêtes dans un graphe appelé plan *MVPP* (Multi View processing plan).

Afin de formaliser le problème combiné de la $\mathcal{GCOR}$, nous procédons d'abord à une formalisation de chaque problème ($\mathcal{GC}$ et $\mathcal{OR}$) de manière isolée.

Le problème de $\mathcal{GC}$ est formalisé comme suit: **(1)** Entrées : (i) un entrepôt de données relationnel $\mathcal{EDR}$, (ii) une charge de requête $\mathcal{Q} = \{Q_1, Q_2, ..., Q_n\}$ représentée par un MVPP , (iii) un ensemble de noeuds intermédiaires $\mathcal{N} = \{no_1, no_2, ..., no_l\}$ du MVPP candidats pour le cache , **(2)** Les contraintes: un cache de taille $\mathcal{B}$ et **(3)** Sorties : une politique de gestion du cache $\mathcal{BM}$ qui permet d'allouer de l'espace mémoire sur le cache pour les noeuds candidats afin d'optimiser le coût globale d'exécution de la charge de requêtes $\mathcal{Q}$.

Le problème $\mathcal{OR}$ est formalisé comme suit: **(1)** Entrées : (i) un entrepôt de données relationnel $\mathcal{EDR}$, (ii) une charge de requêtes représentée par l'ensemble $\mathcal{Q} = \{Q_1, Q_2, ..., Q_n\}$, (iii) une politique de gestion du cache $\mathcal{PGC}$. **(2)** Sorties : requêtes ordonnées $\mathcal{QS} = \{S\,Q_1, S\,Q_2, \ldots, S\,Q_n\}$

tel que le coût d'exécution de la charge de requêtes $\mathcal{QO}$ soit minimal

En se basant sur ces formalisations, le problème $\mathcal{GCOR}$ peut être décrit comme suit

— Entrées: (i) un entrepôt de données relationnel $\mathcal{EDR}$, (ii) une charge de requêtes $\mathcal{Q} = \{Q_1, Q_2, ..., Q_n\}$ représentée par un MVPP, (iii) un ensemble $\mathcal{N} = \{no_1, no_2, ..., no_l\}$ de noeuds intermédiaires du plan MVPP et qui sont candidats pour le cache ;
— Contraintes: une taille du cache $\mathcal{B}$;
— Sorties : (i) un ensemble de requêtes planifié $\mathcal{SQ} = \{SQ_1, SQ_2, ..., SQ_n\}$ et (ii) une politique de gestion du cache $\mathcal{PGC}$, qui permet de minimiser le coût d'exécution globale des requêtes.

.

Comme mentionné précédemment, le problème de gestion du cache et l'ordonnancement des requêtes sont des problèmes difficiles. Dans le cadre des entrepôts de données relationnels, une charge de requêtes peut être optimisée par une solution qui résout à la fois le problème du cache et l'ordonnancement des requêtes en exploitant la forte interaction entre les requêtes de jointures en étoile qui contiennent plusieurs sous-expressions communes. Cependant la combinaison des deux problèmes augmente la complexité. Il a été démontré que l'ordonnancement des requêtes sans prise en charge du problème de gestion du cache est un problème *np*-complet [**?**].

Le problème de gestion du cache consiste à trouver la meilleure combinaison possible de sous-ensembles de noeuds d'une charge de requête qui devraient être mise dans le cache, ce qui constitue un grand espace de recherche (tous les sous-expressions possibles sur le schéma de la base de données). Ce problème est similaire au problème du sac à dos, où le sac dans notre cas représente le cache et les objets sont les noeuds à mettre dans le cache. Le problème du sac à dos est connue pour être *np*-complet [**?**, **?**, **?**].

6.3 Le modèle de coût utilisé

Afin d'instancier notre système ($\mathcal{DBS}$) de base de données dans le méta-modèle proposé, nous décrivons chaque composant de notre système pour la résolution du problème joint de gestion du cache et l'ordonnancement des requêtes. (1) Les structures de stockages ST utilisées (composantes bases de données) sont les tables relationnelles avec un stockage orienté colonne. Nous ignorons les indexes, les vues matérialisées, et nous supposons qu'il n'y ait pas de compression de données. (2) L'ensemble des opérations OS est constitué de toutes les opérations de l'algèbre relationnelle: projection, sélection, jointures, agrégation, tris, union. Ces opérations sont représentées dans la composante de requêtes dans notre modèle de coût générique ($\mathcal{MCG}$). (3) Le cache est alloué sur le support de stockage principal représenté par une zone dans la mémoire centrale. Pour le stockage en mémoire secondaire nous considérons que le support de base est le disque dur. Par conséquent, notre $\mathcal{DBS}$ est modélisé comme suit:

$$DBS = (\{Tables\}, \{\sigma, \pi, \bowtie, \cup, \cap, groupby, sort, aggregate\}, \\ MainMemory, \{HardDiscDrive\})$$

$$comp(Tables) = 0$$
$$rs(Tables) = 1$$
$$cs(Tables) = 0$$

6.4 Résolution du problème

La grande complexité des problèmes de bases de données et précisément le problème de gestion du cache et l'ordonnancement des requêtes, nous motive à développer des solutions efficaces. La plupart des algorithmes proposés sont de types: (1) simples: ces algorithmes sont basés des mesures de similarité entre les objets manipulés comme l'affinité entre les requêtes qui est une métrique qui permet de mesurer le nombre de résultats intermédiaires communs pour deux requêtes. Ces algorithmes sont simples mais n'assurent pas une bonne qualité des solutions finales (2) heuristiques ou méta-heuristiques inspirées de phénomènes naturels ou des comportements collectifs des animaux ou des insectes. Parmi ces algorithmes, on trouve: les algorithmes génétiques, le recuit simulé, le hill climbing, les colonies de fourmis, etc. Ces algorithmes permettent de chercher des solutions presque optimales mais ils sont gourmand en termes de temps de calcul. Dans notre cas, nous utilisons une troisième approche proposé dans [**?**] qui représente un compromis entre les deux approches précitées. Cette approche est appelée *Queen-Bee* et appartient à la classe de solution *diviser pour régner* pour réduire l'espace de recherche des solutions. Nous améliorons cette approche, en y intégrant un algorithme heuristique PDP (push down projection) qui permet de descendre les projections dans le MVPP à chaque fois que cela est possible.

6.4.1 L'algorithme queen-bee

L'idée principale de cet algorithme est de grouper les requêtes selon leur affinité et d'exécuter séquentiellement les requêtes dans chaque groupe appelé "hive". De chaque groupe, une requête appelée *queen-bee* est élue pour être la première à exécuter parmi les autres requêtes du même groupe. L'exécution de ces requêtes bénéficie des résultats intermédiaires mises en cache par la *queen-bee*. Une fois l'exécution de toutes les requêtes du même groupe est achevée, le cache est vidé pour l'exécution des requêtes des autres groupes restants.

6.4.2 Descente des projections

L'approche proposée peut être appliquée de la même façon pour des bases de données en colonnes ou en lignes. L'avantage de l'approche de stockage en colonnes réside dans le fait que les attributs non nécessaires pour les opérations des requêtes ne sont pas chargés. Pour avoir cet même avantage pour les bases de données stockées en lignes, nous avons intégré une heuristique $\mathcal{PDP}$ (Push-Down-Projections) qui permet de descendre les opérateurs de projections le plutôt possible dans les plans d'exécution de chaque requête pour réduire le volume de données traité par tous les algorithmes qui implémentent les opérateurs des requêtes et de ne mettre dans le cache que les données pertinentes pour l'optimisation.

7 Expérimentation

Nous présentons dans cette section notre étude expérimentale en utilisant notre modèle de coût mathématique théorique suivi d'une validation sur le $\mathcal{SGBD}$ Oracle11g. Nous avons utilisé le banc d'essais $\mathcal{SSB}$. Nous avons réalisé les tests avec plusieurs facteurs de passage à l'échelle (*scale factor*) allant de $SF = 1$ (un entrepôt de taille de 1 GO) jusqu'à $SF = 120$ (un entrepôt de taille de 120 Go). Nous avons réalisé les tests sur 30 requêtes de jointures en étoiles. Nous avons utilisé un serveur de $32Go$ de $\mathcal{RAM}$ et un processeur 2x2.40 Gh. Pour les résultats théoriques, un simulateur développé en Java nous a permis d'extraire automatiquement les différents paramètres pour notre modèle de coût.

7.1 Résultats des simulations

La figure 3.14 montre l'impact des deux algorithmes utilisés dans les tests, Queen-bee et $\mathcal{PDP}$ par rapport à une une exécution sans la technique proposée. Beaucoup de tests sont effectués sur un cache de taille 20 Go. La technique de mise en cache permet d'améliorer le temps global d'exécution de la charge de requête. L'algorithme $\mathcal{PDP}$ permet encore de réduire le temps d'exécution des requêtes en diminuant la taille des résultats intermédiaires juste après le parcourt des tables (opération de scan). Les deux techniques combinées nous ont permis de gagner entre 50 et 64% en temps de calcul des résultats de nos requêtes de test.

7.2 Validation des résultats

Pour confirmer les résultats théoriques et montrer la qualité de notre modèle de coût, nous avons effectué des test sur un environnement réel sous oracle 11g. Les expérimentations ont été effectuées sur le même ensemble de données en utilisant un facteur de passage à l'échelle

8. www.cs.umb.edu/poneil/StarSchemaB.PDF

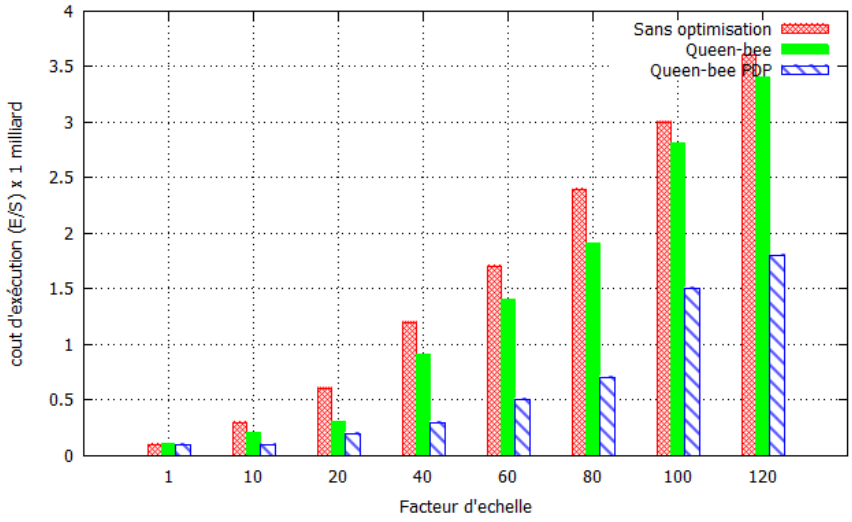


FIGURE 3.14 – Résultat de simulation théorique

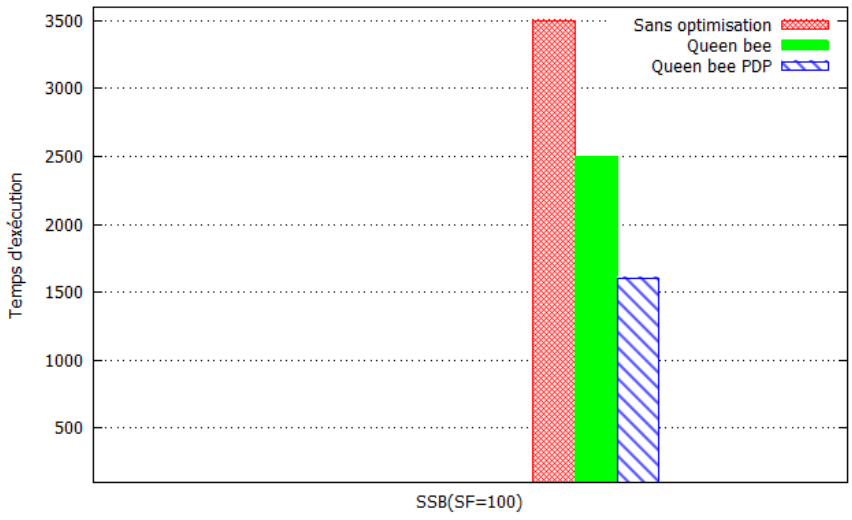


FIGURE 3.15 – Validation des résultats sous oracle

$SF = 100$. Pour avoir les mêmes conditions de test, nous avons forcé l'exécution des requêtes selon les mêmes plans recommandés par le simulateur (réécriture des requêtes et mise en cache manuelle via les commandes *oracle hints*. La figure 3.15 montre les résultats donnés par le $\mathcal{SGBD}$ oracle qui sont très similaires à ceux donnés par le modèle de coût.

8 Conclusion

Un modèle de coût est une composante centrale dans l'optimisation des requêtes. Il représente une métrique des besoins non fonctionnels comme le temps d'exécution de requêtes, la consommation énergétique, etc. Il est utilisé par un nombre important d'outils, comme les optimiseurs de requêtes, les outils de recommandations (advisors), etc. Un modèle de coût précis est celui qui prend en considération tous les paramètres effectifs nécessaires pour avoir une base de données optimale. Ces paramètres doivent être issus de toutes les couches du cycle de vie de la base de données (analyse des besoins, conceptuel, logique, physique et déploiement). Le modèle de coût est aussi la brique de base pour la conception physique.

Dans ce chapitre nous avons présenté un processus de description des entrées de tout modèle de coût, afin de les expliciter et surtout fournir aux développeurs des modèles de coût des paramètres à la carte correspondant à chaque couche d'une base de données/entrepôt de données. Nous avons considéré les dimensions suivantes: le schéma de la base de donnée, des requêtes, des supports de stockage et la plate de forme de déploiement. Un effort de méta modélisation a été effectué pour décrire l'ensemble de dimensions. Une ontologie de supports de stockage a été construite. Cette construction a été menée par une analyse des différentes propriétés décrivant les disques durs et les mémoires flash que nous pouvons trouver chez les constructeurs et dans les fiches de description de chaque support. Cette ontologie a été validée par les membres de l'équipe Ingénierie de Données et des Modèles, et plus précisément dans le cadre de la thèse d'Amira Kerkad [**?**]. Une particularité de notre modèle de coût est sa capacité d'intégrer l'interaction entre les structures d'optimisation.

Une instanciation de notre modèle de coût a été effectuée en considérant le problème combinant deux sous problèmes de la conception physique connus comme NP-complet qui sont : le problème d'ordonnancement des requêtes et la gestion du cache. Cette instanciation a considéré le cas d'un entrepôt de données centralisé sous le $\mathcal{SGBD}$ Oracle, stockage en lignes sur un support composé d'une mémoire centrale $\mathcal{RAM}$ et d'un disque dur $\mathcal{HDD}$). Nous tenons à signaler que la présence de notre modèle de coût a permis de réduire les efforts de définition ces modèles mathématiques et surtout fournir une vue globale et paramétrée de l'ensemble de propriétés qu'un modèle peut utiliser.

Nous envisageons de considérer d'autres instances d'entrepôts de données; déployées sur des $\mathcal{SGBD}$ et plate-formes différents afin de valider l'ensemble de contributions de ce chapitre.

Chapitre 4

Extension du cache du SGBD par la mémoire flash

Sommaire

1 Introduction 97
2 État de l'art 98
3 Vers un SGBD avec un système de stockage hybride 103
4 Exécution des requêtes dans le cache hybride 112
5 Étude expérimentale 116
6 Conclusion 120

1 Introduction

La performance des requêtes est une question clé dans la conception efficace d'un $\mathcal{SGBD}$. Comme montré dans les chapitres précédents, il existe une large panoplie de structures d'optimisation pour satisfaire ce besoin non fonctionnel. Toutefois, l'association de ces techniques aux nouvelles technologies comme les supports de stockage représente un *enjeu important* pour les développeurs du matériels et du logiciels. Dans nos synthèses approfondies dans les chapitres précédents sur le processus d'optimisation, nous avons conclu qu'il dépend fortement de trois principaux facteurs: (a) le type de la base de données (oltp, olap, objet, etc.), (b) les requêtes définies sur cette base de données et (c) des propriétés du support de stockage. Les requêtes décisionnelles définies sur un entrepôt de données relationnels ont une forte interaction, cela est dû au fait que toute requête passe systématiquement par la table des faits. En conséquence, il faut les traiter d'une manière groupée (multi-query optimization). Ce groupement se fait à l'aide de la fusion de l'ensemble de plans des requêtes. Dans les entrepôts de données traditionnels, ces noeuds sont candidats à la matérialisation. Une fois sélectionnées, les vues matérialisées sont stockées sur le disque. Dans cette thèse, nous proposons une approche différente et qui consiste à les stocker au niveau du cache afin de faire profiter d'autres requêtes de leur présence. Vu la taille du disque dur, il peut stocker un nombre important de vues matérialisées. Le cache a une capacité de stockage limitée par rapport aux disques. Pour relaxer cette contrainte, nous proposons dans cette thèse de les étendre avec les mémoire flash (Figure 4.1).

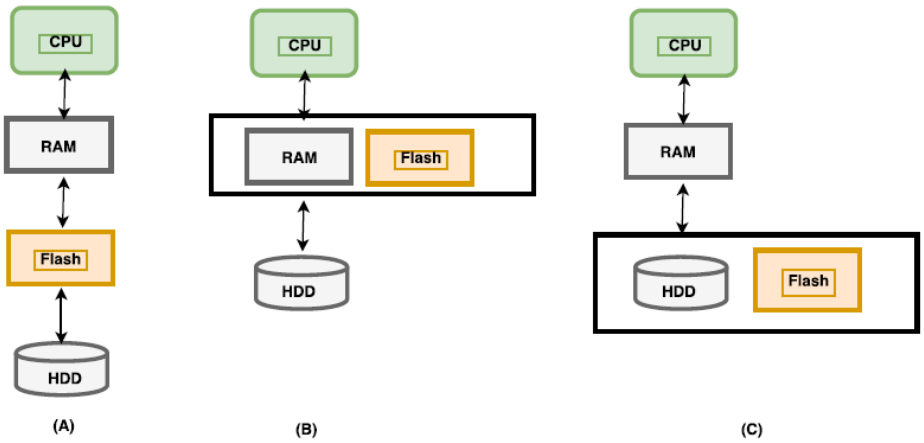


FIGURE 4.1 – Le flash pour augmenter la capacité de stockage des supports traditionnels

Dans ce chapitre nous montrons cette association qui regroupe les mémoires $\mathcal{RAM}$, $\mathcal{SSD}$ et $\mathcal{HDD}$. Pour réussir cette association, le développement d'un système de décision est nécessaire à l'aide de modèles de coût propres aux supports de stockage utilisés.

Nous débutons ce chapitre par un état de l'art sur l'intégration de la mémoire flash dans le $\mathcal{SGBD}$ et particulièrement la gestion du cache dans les contextes de disques durs et mémoires flashs. Après cette présentation, nous présentons notre approche associant les deux supports

pour stocker les résultats intermédiaires.

2 État de l'art

Traditionnellement, les $\mathcal{SGBD}$ sont conçus et optimisés pour les supports de stockage classiques, à savoir les disques durs. Le stockage orienté ligne, le groupement, les méthodes d'accès sont des exemples de techniques dont l'objectif est de réduire la latence du disque dur en générant plus d'accès séquentiels. Dans le Chapitre 2, nous avons vu que les mémoires flashs possèdent des propriétés différentes de celles des disques durs. Il serait intéressant de les combiner pour tirer le meilleur de chacun d'eux. Plusieurs travaux ont été proposés dans cette direction, que nous pouvons classer en quatre catégories: **(1)** le placement de données, **(2)** l'extension du cache, **(3)** les techniques d'indexation et **(4)** l'optimisation des jointures

En ce qui concerne le placement de données, les techniques proposées consistent à résoudre le problème de stockage de données sur la mémoire flash en prenant en considération ses caractéristiques de stockage et ses contraintes. Une première solution consiste à déléguer les tâches de la FTL à un système de fichier spécialisé ce qui à donné naissance à des systèmes comme JFFS et YAFFS. Les LSFS (log structured file systems) [**?**] sont des systèmes de fichiers qui traitent le support de stockage comme un journal circulaire et écrivent de manière séquentielle à la tête du journal. Pour les mémoires flash qui disposent d'une FTL, wang et al [**?**] montrent par une série de test sur deux systèmes de gestion de bases de données différents (commercial et Mysql open source) que les LSFS permettent d'atteindre un gain en performances d'un facteur de 3 à 6 par rapport à un système de fichier classique (EXT2: mise à jour directe des données). À cause de la contrainte de mise à jour indirecte de données (Voir la section 5.3.2 du chapitre 2), une page modifiée est réécrite sur un nouveau emplacement libre de la mémoire flash. L'approche proposée dans [**?**] permet de ne réécrire que la partie modifiée d'une page. Chaque page présente en mémoire centrale peut avoir un secteur log qui stocke ces modifications. Une région log est allouée sur chaque bloc pour stocker les parties modifiées des pages. Les mises à jour sont effectivement appliquées (écriture sur le log de la mémoire flash) lorsque le secteur log est rempli ou la page est évincée du buffer. Cette approche permet de minimiser les écritures mais ralentit la lecture car accéder à une page p, revient à lire la page et son log lp (voir la figure 4.2). Une amélioration de cette approche a été proposée dans [**?**]. Cette approche nommée PDL (Page-Differential logging) consiste à stocker un différentiel par rapport à la page au lieu de chaque modification. Elle permet de réduire le temps de lecture d'une page car cette lecture nécessite seulement l'accès à la page et son dernier différentiel. L'approche PAX (Partition across attributes) représente une architecture orientée colonne, proposée dans [**?**] indépendamment du type de support de stockage. Le principe de PAX est de garder les valeurs d'attribut de chaque enregistrement sur la même page (voir la figure 4.3). Les tuples qui se trouvent sur la même page sont ensuite verticalement fragmentés. Les valeurs relatives à chaque attribut sont stockées sur la même mini-page. En se basant sur l'architecture PAX, les auteurs dans [**?**] ont développé un

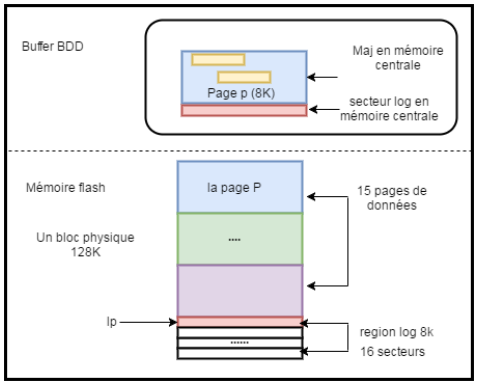


FIGURE 4.2 – L'approche IPL (in page logging) [**?**]

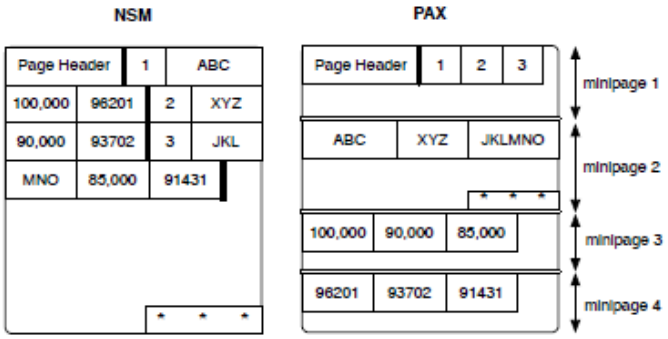


FIGURE 4.3 – PAX vs stockage en ligne (NMS: N-ary storage Model) [**?**]

algorithme nommé *flashscan* qui permet d'optimiser les opérations de sélection et de projection sur la mémoire flash en ne lisant effectivement que les mini-pages nécessaires pour satisfaire les requêtes.

Quelques initiatives ont été menées pour étendre le cache. Dans [**?**], les auteurs proposent une approche qui utilise un $\mathcal{SSD}$ comme cache entre la mémoire centrale et le $\mathcal{HDD}$. L'approche tente de mettre en cache les pages les plus fréquemment accédées. Le disque $\mathcal{HDD}$ est traité comme des blocs (extents) de 32 pages. Les extents du disque sont classés selon leurs températures (chaud et froid en analysant les traces). Initialement chaque extent a une température égale à 0. À chaque lecture d'une page du disque $\mathcal{HDD}$ (page absente du cache) la température de l'extent qui la contient est incrémentée par le gain accumulé si la page a été lue à partir du disque $\mathcal{SSD}$. Cette température prend aussi en considération les accès de type séquentiel et aléatoire. Lorsqu'une page recherchée ne se trouve pas sur le cache de la $\mathcal{RAM}$, elle est recherchée d'abord dans le cache $\mathcal{SSD}$ sinon dans le $\mathcal{HDD}$.

La politique d'admission dans le cache $\mathcal{SSD}$ est basée sur la température. L'objectif de la méthode est de cacher les pages ayant les températures les plus élevées. Si la température de la page lue du $\mathcal{HDD}$ est supérieur à la température la plus faible du cache, la page sera stockée sur le $\mathcal{SSD}$. Si une page est lue du disque et écrite dans le cache de la $\mathcal{RAM}$, elle peut être

copiée sur le $\mathcal{SSD}$ si sa température le permet. Lorsqu'une page sur le cache $\mathcal{RAM}$ est mise à jour, sa copie (si elle existe) dans le $\mathcal{SSD}$ est invalidé mais pas retirée. Une page retirée du cache $\mathcal{RAM}$ est normalement enregistrée sur le $\mathcal{HDD}$ et sur le $\mathcal{SSD}$ si elle est invalide.

L'objectif de l'approche proposée dans [**?**] est d'améliorer les performances du $\mathcal{SGBD}$ en utilisant un disque $\mathcal{SSD}$ comme une extension du cache RAM de base de données. Le principe de cette approche est de stocker les pages retirées du cache RAM sur le disque $\mathcal{SSD}$ en se basant sur leurs motifs d'accès (séquentiel ou aléatoire). L'approche se distingue par la façon dont elle gère ces pages. Trois stratégies sont proposées: **CW** (clean write), **DW**(dual write) et **LC** (lazy-cleaning). La stratégie **CW** n'écrit jamais des pages retirées du cache $\mathcal{RAM}$ vers le $\mathcal{SSD}$. Cette stratégie favorise les requêtes de lecture aléatoire qui est une une bonne caractéristique des $\mathcal{SSD}$. Elle ne nécessite pas non plus la synchronisation du contenu des pages entre $\mathcal{SSD}$ et $\mathcal{HDD}$. Quand une page est modifiée, elle est retirée du $\mathcal{RAM}$ cache. La stratégie **DW** l'écrit à la fois sur le $\mathcal{HDD}$ et le $\mathcal{SSD}$, ce qui permet au $\mathcal{SSD}$ de jouer le rôle d'un buffer de lecture pour le $\mathcal{HDD}$. Avec la stratégie **LC**, les pages modifiées sont écrites sur le cache $\mathcal{SSD}$ d'abord, et copiées ultérieurement sur le disque $\mathcal{HDD}$. Dans ce cas le $\mathcal{SSD}$ joue le rôle d'un cache d'écriture pour le $\mathcal{HDD}$.

Une approche proposée dans [**?**] consiste à utiliser un disque $\mathcal{SSD}$ comme un cache pour le $\mathcal{HDD}$. La politique de gestion du cache développée permet de rediriger les pages fréquemment lues vers le $\mathcal{SSD}$ et les pages fréquemment mises à jour vers le $\mathcal{HDD}$. Pour prendre en considération le changement du motif d'accès (séquentiel ou aléatoire), les auteurs ont développé trois algorithmes de migration des pages appelés : conservatif, agressif et hybride. L'objectif de chaque algorithme est de placer les pages sur le bon support. L'approche est basée sur un modèle analytique et simule plusieurs disques selon les paramètres fournis par leurs constructeurs. Dans [**?**] les auteurs décrivent trois techniques pour l'utilisation de la mémoire flash afin d'optimiser la gestion des données. Chaque technique représente un exemple de la façon dont la mémoire flash peut être utilisée dans les systèmes actuels pour répondre aux limites des disques durs. La première technique considère le traitement des transactions où les données sont résidentes en mémoire centrale, et montre comment utiliser la mémoire flash pour logger les tuples des transactions avant leur commit sur la base de données. La deuxième technique consiste à remplacer tout simplement le $\mathcal{HDD}$ par la mémoire flash. Dans ce scénario les auteurs montrent les améliorations de performance de la mémoire flash par rapport au disque $\mathcal{HDD}$. Ils montrent aussi la nécessité de répondre aux nouveaux défis introduits par les écritures aléatoires excessives. Ces dernières dégradent les performances de la base de données ce qui cause la défaillance prématurée de la mémoire flash. La troisième technique, montre comment la flash peut être utilisée pour améliorer les performances des entrepôts de données stockés principalement sur les disques durs. Cette technique consiste à utiliser la flash comme un cache pour la mise à jour des données de l'entrepôt.

D'autres travaux de recherche ont été proposés pour intégrer la flash comme un cache dont la granularité est supérieure à la taille d'une page pour différents objets d'une base de données (table, index, etc.). Une question fondamentale pour ce type de cache est de décider quelles

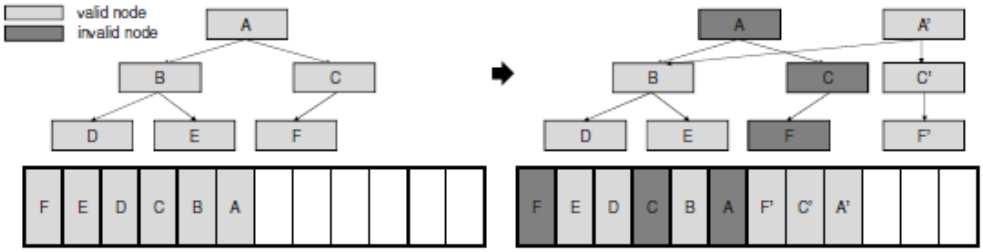


FIGURE 4.4 – B-tree: mise à jour d'un tuple [**?**]

sont les données à y mettre. Dans [**?**], les auteurs ont étudié l'applicabilité et l'impact d'utiliser les disques $\mathcal{SSD}$ pour le stockage de certains objets de bases de données qui sont caractérisés par des accès aléatoires en lecture et des accès séquentiels en écriture. Les auteurs montrent qu'un gain en performance jusqu'à un ordre de grandeur peut être atteint pour les transactions en affectant les journaux de transactions, les segments rollback et les *tablespaces* temporaires au disque $\mathcal{SSD}$. Dans [**?**] les auteurs ont développé un outil d'administration pour le $\mathcal{SGBD}$ BD2 [15] qui permet de migrer les objets de bases de données comme les tables, index et les vues matérialisées sur le disque $\mathcal{SSD}$. Cet outil scrute les entrées/sorties des requêtes pour déterminer le profil de chaque objet de la base de données. Le problème de migration est modélisé comme un problème d'optimisation. deux approches ont été proposées pour le résoudre?: un algorithme glouton et la programmation dynamique.

Plusieurs travaux d'indexation de données sur la mémoire flash existent. L'objectif est de définir de nouvelles structures d'indexation de type B-arbre mais compatibles avec le mode de fonctionnement de la mémoire flash. Dans un index de type B-arbre, seuls les clés et les pointeurs vers un le noeuds fils sont stockés dans les noeuds intérieurs. Les tuples sont enregistrés au niveau le plus inférieur de l'arbre. Sur la mémoire flash, la mise à jour d'un tuple (feuille de l'arbre) nécessite la réécriture de la page qui le contient sur un nouveau emplacement libre (contrainte : erase-before-write). Ce changement de l'adresse physique du tuple implique immédiatement la réécriture de toute les pages qui contiennent les adresses de ces prédécesseurs (parents) jusqu'à la racine, ce qui génère un nombre important d'écritures. Par exemple, dans la figure 4.4, la mise à jour de la page contenant le noeud F, nécessite la mise à jour des pages de C et A. La structure μ-tree proposée dans [**?**] est un index de type B-tree. L'idée principale de cet index est de stocker tous les noeuds à mettre à jour de la feuille jusqu'à la racine dans la même page. Lorsqu'un tuple est modifié, tous ses parents le sont aussi, mais cela ne nécessite l'écriture que d'une page, réduisant ainsi l'écriture. Par exemple, dans la figure 4.5, la mise à jour de la page contenant le noeud F, nécessite l'écriture d'une seule page contenant les nouvelles données F', C' et A'.

15. www.www.ibm.com/software/data/db2

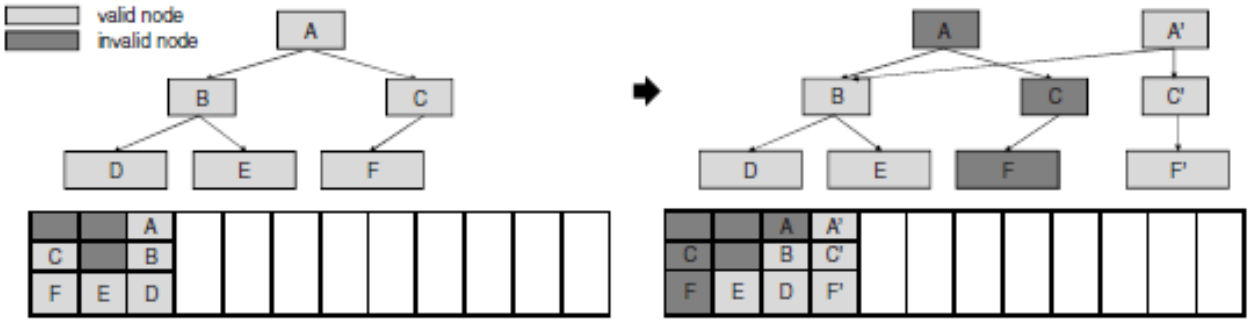


FIGURE 4.5 – μ-tree: mise à jour d'un tuple [**?**]

Agrawal et al [**?**] ont proposé un index appelé LA-tree (Lazy adaptative tree). Cet index possède un buffer de taille variable à chaque niveau de l'arborescence à partir de la racine pour éviter les mises à jour coûteuse. Cette approche utilise un algorithme appelé ADAPT qui décide à tout moment de faire le choix entre vider le buffer et de propager les mises à jour aux noeuds descendants ou de continuer la mise en cache des mises à jour des noeuds. Cette décision dépend de la charge en cours car la lecture des noeuds devient coûteuse lorsque la taille du buffer augmente.

Finalement, quelques travaux ont été proposés pour optimiser les opérations de jointures sur flash. Dans [**?**] les auteurs ont effectué une comparaison entre les deux algorithmes de jointure hash join et sort-merge join pour un système de gestion de bases de données commercial. Leurs résultats affirment que l'algorithme sort-merge est mieux adapté pour les mémoires flash de type $\mathcal{SSD}$ dans le cas du $\mathcal{SGBD}$ testé. Sur la base de l'architecture PAX [**?**], les auteurs dans [**?**] ont développé un algorithme de jointure qui minimise les entrées/sorties sur la mémoire flash. Les auteurs montrent que leur algorithme est plus efficace que l'algorithme de jointure par hachage hybride (hybrid hash join), particulièrement quand la cardinalité de la jointure est faible ou le nombre d'attributs résultats de la jointure est aussi faible.

2.1 Bilan

Notre contribution principale consiste à utiliser la mémoire flash comme un cache de deuxième niveau entre la mémoire centrale et le disque $\mathcal{HDD}$ pour les entrepôts de données. Contrairement aux approches ci-dessous citées qui cachent les données au niveau de la page, notre approche met les données dans le cache à une granularité supérieure qui est celle d'un résultat intermédiaire d'une requête. Cette technique est très convenable pour les entrepôts de données et les mémoires flash. Les requêtes décisionnelles partagent un nombre important d'opérations. Ce phénomène est connu sous le nom de l'optimisation multi-requêtes [**?**, **?**, **?**, **?**]. La plupart de ces travaux optimisent toutes les requêtes du même lot et génèrent un plan d'exécution unique pour toutes les requêtes. Les résultats intermédiaires bénéfiques pour l'exécution de toute la

charge de requêtes sont sélectionnés et matérialisés sur le support de stockage secondaire qui est le disque dur $\mathcal{HDD}$. Notons aussi que les approches classiques pour la $\mathcal{MQO}$ ignorent l'aspect dynamique des requêtes. Les mêmes vues matérialisées continuent à être stockées et utilisées même si la charge de requêtes change.

Dans la section suivante, nous détaillons notre approche d'hybridation de cache pour optimiser les requêtes décisionnelles dans les cas statique et dynamique.

3 Vers un SGBD avec un système de stockage hybride

Dans cette section, nous présentons une architecture d'un $\mathcal{SGBD}$ avec un cache hybride selon notre méta-modèle proposé (voir le chapitre 3). Vu la complexité de l'environnement de bases de données, nous considérons certaines hypothèses simplificatrices: (1) les structures de stockages concernent les tables de notre entrepôt de données avec un stockage orienté ligne, (2) absence de structures d'optimisation (index, vues matérialisées, compression, etc.), (3) l'ensemble des opérations OS est constitué de toutes les opérations de l'algèbre relationnelle : projection, sélection, jointures, agrégation, tris, union, et (4) la base de données est stockée sur un disque dur. Avec ces hypothèses, notre $\mathcal{SGBD}$ peut être modélisé comme suit:

$$\begin{aligned} DBS = (&\{Tables\}, \{\sigma, \pi, \bowtie, \cup, \cap, groupby, sort, aggregate\}, \\ &MainMemory, \{FlashMemory, HardDiscDrive\}) \end{aligned}$$

$$\begin{aligned} comp(Tables) &= 0 \\ rs(Tables) &= 1 \\ cs(Tables) &= 0 \end{aligned}$$

L'architecture de notre approche est décrite dans la Figure 4.6. Le fait d'avoir un cache hybride, il est important de lui associer une politique de gestion de cache. Pour un $\mathcal{SGBD}$ utilisant un cache classique, le moteur d'exécution de requêtes sollicite le gestionnaire de cache pour effectuer des lectures ou des écritures de pages. Le gestionnaire de cache à son tour interagit avec le gestionnaire de disque qui gère l'interface avec les supports de stockage physiques. Lorsqu'une page est demandée, le gestionnaire de cache vérifie d'abord le cache en $\mathcal{RAM}$ et renvoie la page demandée si elle est présente. Dans le cas contraire, s'il y a encore de l'espace libre dans le cache $\mathcal{RAM}$, le gestionnaire de cache demande au gestionnaire de disque de récupérer la page à partir du disque $\mathcal{HDD}$.

Si le cache $\mathcal{RAM}$ est plein, le gestionnaire de cache libère l'espace en choisissant une page

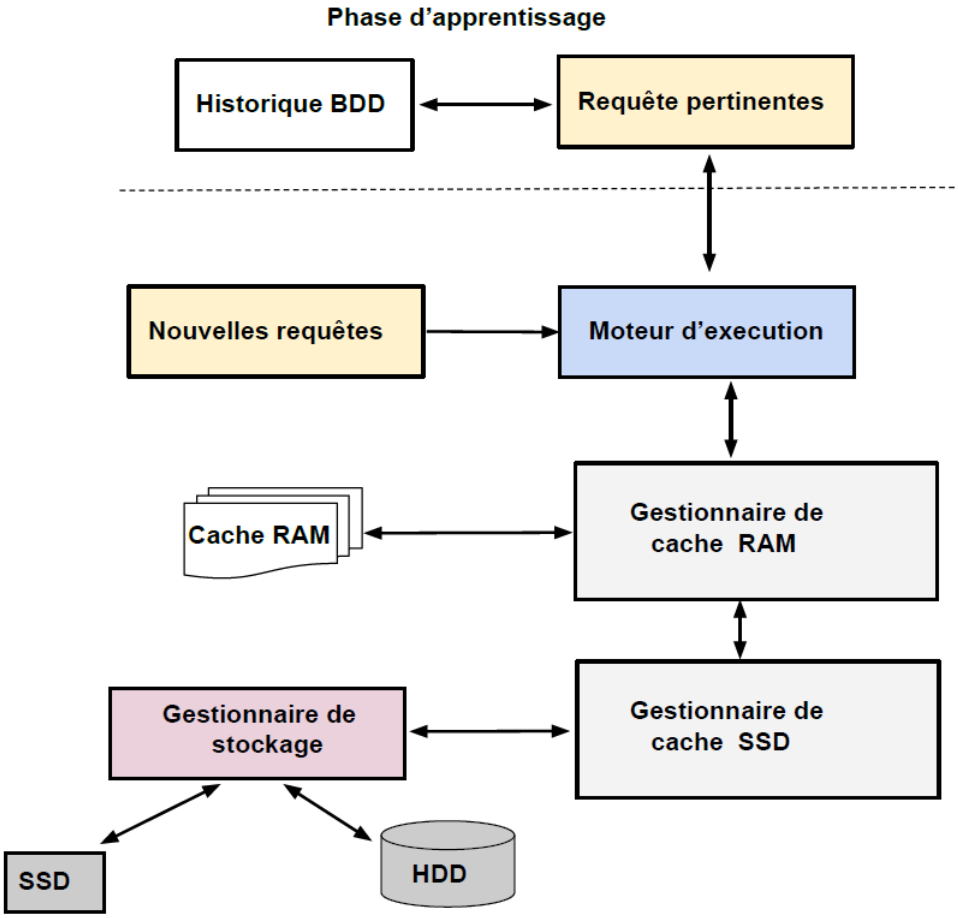


FIGURE 4.6 – Architecture du cache hybride

victime selon une certaine politique de remplacement. Enfin, si la page libérée du cache a subi une modification, elle sera réécrite sur le disque $\mathcal{HDD}$. Dans notre proposition, lorsque le moteur d'exécution de requête effectue des appels au gestionnaire de cache pour lui demander des résultats temporaires, celui-ci examine d'abord le cache RAM pour vérifier l'existence du résultat. Si le résultat est disponible, il sera renvoyé. Cependant, si le résultat est introuvable, le gestionnaire de cache $\mathcal{RAM}$ donne la main au gestionnaire du cache $\mathcal{SSD}$ qui vérifie d'abord si une copie du résultat est mis en cache dans le $\mathcal{SSD}$. Si oui, le résultat est lu à partir du $\mathcal{SSD}$ et ces informations d'utilisation sont mises à jour.

Sinon, le résultat est récupéré à partir du disque $\mathcal{HDD}$. S'il n'y pas d'espace libre dans le cache $\mathcal{RAM}$, le gestionnaire de cache $\mathcal{RAM}$ choisit un résultat victime pour libérer de l'espace. Le résultat victime libéré de la RAM est traité par le gestionnaire de cache $\mathcal{SSD}$ selon sa politique d'admission pour une éventuelle mise en cache dans le disque $\mathcal{SSD}$. Enfin, si l'espace de stockage SSD est plein, le gestionnaire du cache $\mathcal{SSD}$ récupère de l'espace en libérant un résultat victime.

3.1 Le module d'apprentissage

La principale fonction de ce module est de permettre de mieux optimiser le cache en exploitant les requêtes qui se trouvent dans le fichier log du $\mathcal{SGBD}$. L'extraction de ces requêtes se fait par des modules dédiés disponibles dans $\mathcal{SGBD}$. Par exemple, Oracle offre plusieurs vues appelées vues dynamiques de performances comme VSQL, VSQLAREA, V$SQLTEXT qui maintiennent l'historique de requêtes exécutées au sein du système. Ces vues offrent des statistiques sur les requêtes, leurs fréquences et bien d'autres mesures de performances. L'extraction de cette charge de requêtes représente une base de connaissance importante que nous pouvons utiliser pour identifier leur interaction et la fréquence de chaque requête.

Plus concrètement, une fois les requêtes historiques identifiées, nous construisons leur graphe unifié, appelé, $MVPP$. Soit $N = \{n_1, n_2, \ldots, n_w\}$ l'ensemble de noeuds du MVPP obtenu.

A partir de ces données, nous pouvons formaliser le problème de placement de l'ensemble de noeuds sur les deux supports de stockage de la manière suivante:

Étant donné:

— une charge de requêtes $Q = \{q_1, q_2, ..., q_n\}$ extraite des fichiers log, où chaque requête q_i possède une fréquence d'accès f_i, $(1 \leq i \leq n)$;

— l'ensemble de noeuds intermédiaire du MVPP: $N = \{n_1, n_2, \ldots, n_w\}$.

Le problème de placement de noeuds sur les deux supports consiste à trouver le meilleur schéma d'allocation des noeuds sur la $\mathcal{RAM}$ et la mémoire flash, qui réduit le coût total d'exécution de la charge de requêtes Q tout en respectant les contraintes d'espace disponible sur la $\mathcal{RAM}$ et la mémoire flash. Formellement

$$Maximiser_cout = \sum_{i=1}^{2} \sum_{j=1}^{n} (G_{ij} * x_{ij}) \tag{1}$$

sujet aux contraintes suivantes:

$$\begin{cases} \sum_{j=1}^{n} (Taille_j * x_{ij}) <= C_i, i = 1..2 \\ \sum_{i=1}^{2} (x_{ij}) <= 1, j = 1..n \end{cases}$$

où:

— G_{ij} représente le gain accumulé lorsque le noeud n_j est mis sur le support i. Le gain G_{ij} accumulé lorsque le résultat j est mis sur le support i est calculé par :

$$G_{ij} = \sum_{\forall q \in Q} (f_q * Cost_q(n_j))/nq_{rj}$$

avec f_q est la fréquence de la requête q, $Cost_q(n_j)$ et nq_{rj} désignent respectivement le coût d'exécution d'une requête q qui accède au noeud n_j et le nombre total des requêtes qui partagent n_j;

— $Taille_j$ représente la taille du noeud n_j;
— C_i est la capacité de stockage du support i;
— la variable décisionnelle x_{ij} qui est défini par :

$$x_{ij} = \begin{cases} 1 & \text{si le résultat } R_j \text{ est est stocké sur le support } S_i \\ 0 & \text{sinon} \end{cases}$$

3.1.1 Algorithmes de résolution de notre problème

Ce problème de placement de données sur les différents support est un problème de décision (voir figure 4.7). Pour sa résolution, nous avons utilisé deux heuristiques: le *hill climbing* et une approche génétique (voir les algorithmes 1 et 2). Dans l'approche génétique, les chromosomes sont codés avec des tableaux d'entiers. Chaque élément du tableau correspond à un résultat intermédiaire d'une requête. Si cet élément contient la valeur 1, le résultat est mis dans le cache sinon la valeur est 0. Les algorithmes ont été testé avec les paramètres montrés par le tableau 4.1. Comme le montre les figures 4.8 et 4.9, l'approche *hill climbing* a donné des résultats meilleurs mais avec un temps d'exécution plus long que l'approche génétique.

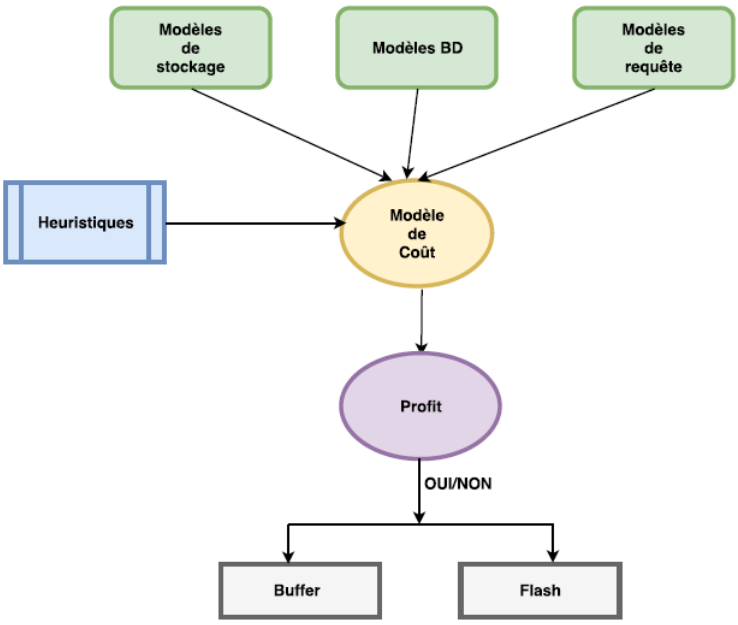


FIGURE 4.7 – Placement des données sur support: un problème de décision

3.2 Cache dynamique

Dans le premier cas, les noeuds pour le cache sont connu à l'avance. Mais dans un entrepôt de données où les requêtes sont ad-hoc nécessite le développement d'une approche dynamique

```
Input: L: liste des noeuds
Output: LC: Liste de noeuds à cacher
TmpGain = 0;
i=0;
Solution=L;
while i<nbre_iteration do
    LCtemp=nil;
    sol=Genere_permutation_aléatoire(solution);
    Taille = 0;
    Gain = 0;
    foreach résultat r dans Sol do
        if (Taille+ Taille[r] <= TailleCache) then
            Taille= Taille+ Taille[r];
            Gain= Gain+Gain[r];
            LCtemp=LCtemp+r;
        end
    end
    if (TmpGain < Gain) then
        tmpGain = Gain;
        LC=LCtemp;
    end
    i++;
end
retourner LC;
```

Algorithm 1: Algorithme du Hill Climbing

```
Input: P: une population de chromosome représentant des solutions, Gmax: nombre de
       génération
Output: S: un chromosome qui représente la meilleur solution
C1,C2,C3,Ctemp: chromosome; Gain=0;
Ctemp=nil; créer population initiale;
i=0;
while i<Gmax do
    C1=selectionner aléatoirement un chromosome;
    C2=selectionner aléatoirement un chromosome;
    C3=croisement(C1,C2); Mutation(C3); //selon les probabilités définies en paramètres
    if (TailleGlobale[C3] <= TailleCache) then
        remplacer(mauvaischromosome, C3)
    end
    i++;
end
foreach C: chromosome de P do
    if (Gain[C]>Gain then
        Gain= Gain[C];
        Ctemp=C;
    end
end
retourner C;
```

Algorithm 2: Algorithme génétique

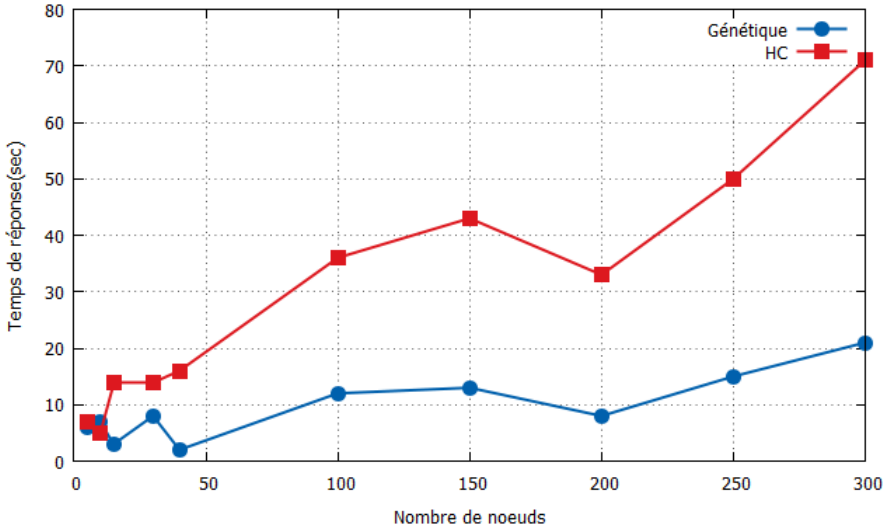


FIGURE 4.8 – Génétique Hill climbing

	Génétique	Hill Climbing
Nombre d'itération	-	1000
Nombre de génération	1000	-
Taille population	5	-
Probabilité de mutation	0.04	-
Nombre maximum de gènes à modifier	1	-

TABLE 4.1 – Paramètres des algorithmes

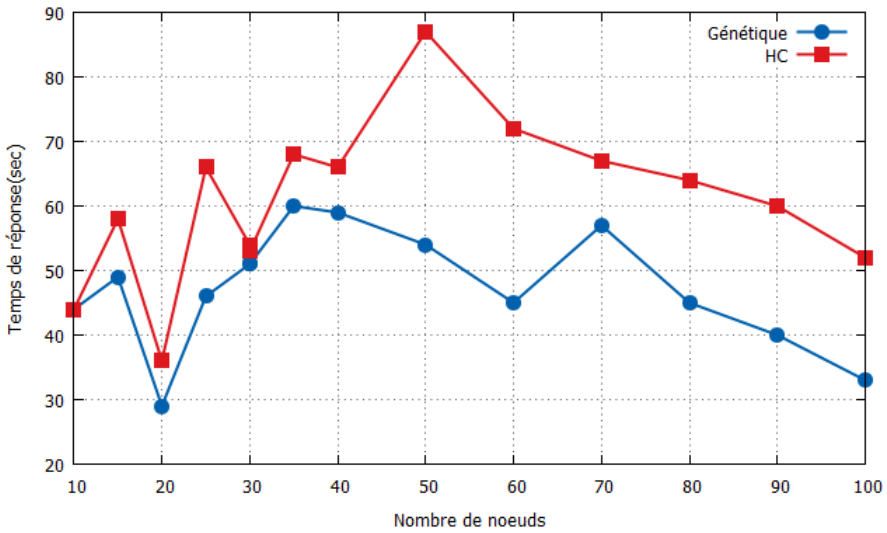


FIGURE 4.9 – Génétique Hill climbing

de gestion du cache.

Noeuds	n1	n2	n3	n4	n5	n6	n7	n8	n9	n10
q1	1	1	0	2	2	0	3	0	3	4
q2	1	1	0	0	0	2	3	0	0	0
q3	1	1	0	0	0	2	0	3	0	4
q4	1	1	0	0	0	2	0	0	0	0
q5	1	1	1	2	0	0	0	3	3	0

TABLE 4.2 – Représentation d'une matrice *MVPP*

Avant de développer notre approche, nous présentons des structures de données permettant de représenter le MVPP d'une charge de requêtes.

Construction de la Matrice d'Usage des noeuds

Cette matrice $\mathcal{M}_U$ modélise l'utilisation de noeuds par des requêtes de Q. À cette fin, $\mathcal{M}_U$ possède L lignes (nombre de requêtes) et N_F (nombre de noeuds) colonnes. $\mathcal{M}_U[l][i] = 1$, avec $1 \leq l \leq L$ et $1 \leq i \leq N_F$, si le neoud F_i est impliqué par la requête Q_l; sinon $\mathcal{M}_U[l][i] = 0$.

Exemple 7

La Table 4.2 montre une représentation sous forme de matrice du graphe unifié pour une charge de 5 requêtes. Le nombre total de noeuds de la charge de requêtes est de 10. Les noeuds 1, 2 et 3 représentent des tables de bases. Les noeuds 4, 5 et 6 représentent des sélections. Les noeuds 7, 8 et 9 représentent des jointures. Enfin, le noeud 10 représente une agrégation. Les valeurs 1, 2, 3, 4 de la matrice représentent respectivement: table de base, sélection, jointure et agrégation. Dans notre exemple, la requête q1 est composée des tables (n1 et n2), des sélections (n4 et n5), des jointures (n7 et n9) et de l'agrégation (n10).

Construction de la Matrice d'Affinités des requêtes

La matrice d'affinité $\mathcal{M}_A$ modélise l'*affinité* entre deux requêtes q_i et q_j. Cette matrice est générée de la même manière que la matrice d'affinité des attributs de la fragmentation verticale [**?**]. C'est une matrice carrée ($n \times n$) symétrique dont les lignes et les colonnes représentent les requêtes. Il est à noter que nous ne nous intéressons pas aux valeurs de la diagonale.

La valeur de chaque élément $aff(q_i, q_j)(1 \leq i, j \leq n)$ est égale au nombre de résultats intermédiaires communs aux deux requêtes.

3.3 Ordonnanceur heuristique des requêtes

Pour améliorer les performances du cache et augmenter la probabilité de trouver un résultat dans le cache (cache hit), il est nécessaire d'exécuter les requêtes ayant des noeuds en commun

en premier lieu. C'est l'objectif de la phase d'ordonnancement des requêtes. Cette phase permet de déterminer le meilleur ordre pour exécuter les requêtes en profitant de l'existence des noeuds partagées dans les caches (RAM ou Flash).

Les requêtes sont regroupées selon leur affinité. L'affinité entre deux requêtes donne le nombre de résultats intermédiaires partagés par eux. L'affinité de la charge de requêtes globale est représentée par un graphe dont les noeuds représentent les requêtes et les arrêtes représentent la mesure d'affinité entre les noeuds. Cette affinité peut être représentée par une matrice carrée dont les lignes et les colonnes représentent les noeuds. La Table 4.3 montre un exemple d'une matrice d'affinité. Elle représente la matrice d'adjacence d'un graphe d'affinité.

Requêtes	q1	q2	q3	q4	q5	q6	q7	q8	q9	q10
q1	3	0	0	0	4	0	3	0	0	0
q2	0	2	0	4	0	4	0	0	5	0
q3	0	0	2	0	0	0	0	4	0	5
q4	0	4	0	3	0	3	0	0	3	0
q5	4	0	0	0	3	0	3	0	0	0
q6	0	4	0	3	0	3	0	0	4	0
q7	3	0	0	0	3	0	2	0	0	0
q8	0	0	4	0	0	0	0	3	0	4
q9	0	5	0	3	0	4	0	0	2	0
q10	0	0	5	0	0	0	0	4	0	2

TABLE 4.3 – Matrice d'affinité des requêtes

Ce graphe d'affinité est ensuite partitionné pour créer des groupes de requêtes. Chaque partition contient les requêtes ayant une grande affinité entre elles. Pour ce faire, nous avons utilisé l'algorithme proposé dans [**?**] dans le cadre de la fragmentation verticale des tables relationnelles. Cet algorithme est polynomial car il nécessite w^2 opérations, où w représente le nombre de noeuds dans le graphe.

Exemple 8

Le résultat de partitionnement du graphe d'affinité correspondant à la matrice décrite dans la Table 4.3 est comme suit :

- *Partition 1 : 3 requêtes [q1, q5, q7]*
- *Partition 2 : 4 requêtes [q2, q4, q6, q9]*
- *Partition 3 : 3 requêtes [q3, q8, q10]*

Enfin, Pour générer l'ordonnancement final des requêtes, deux heuristiques différentes sont utilisées. La première consiste à trier les partitions (groupe de requêtes) dans l'ordre décroissant de leur cardinalité (en nombre de requêtes). Nous favorisons les grandes partitions pour mettre en cache le plus tôt possible, les résultats intermédiaires qui sont partagés avec un nombre maximal de requêtes.

Exemple 9

En prenant en considération les résultats de l'exemple précédent, nous pouvons ordonner les partitions :

1. *Partition 3 : [q2, q4, q6, q9]*
2. *Partition 1 : [q1, q5, q7]*
3. *Partition 2 : [q3, q8, q10]*

La deuxième heuristique nous permet de trier les requêtes à l'intérieur de la même partition. Les requêtes de chaque partition sont triées dans l'ordre croissant selon le nombre d'E/S nécessaires pour leurs exécution.

Exemple 10

Le résultat de tri à l'intérieur de chaque partition est:

1. *Partition 3 : [q6 → q4 → q2 → q9]*
2. *Partition 1 : [q5 → q1 → q7]*
3. *Partition 2 : [q10 → q3 → q8]*

L'ordre d'exécution final des requêtes est le suivant: [q6 → q4 → q2 → q9 → q5 → q1 → q7 → q10 → q3 → q8]

4 Exécution des requêtes dans le cache hybride

Nous présentons dans ce qui suit les différentes approches implémentées pour la gestion du cache dynamique.

4.1 Approche naïve

Cette approche est basée sur une politique de gestion de cache simple comme $\mathcal{LFU}$ (Least Frequently Used) ou $\mathcal{LRU}$ (Least Recently Used). Elle consiste à tester les noeuds composant la requête à exécuter vis-à-vis l'espace libre dans le cache. Dans cette approche, nous considérons que la mémoire flash est une extension de la $\mathcal{RAM}$. Dans le cadre des entrepôts de données, cette approche possède deux inconvénients majeurs:

1. elle n'est pas effective avec des caches de petites tailles (résultats intermédiaires de requêtes très grands) et
2. elle dégrade les performances dans le cas de caches de grandes tailles à cause de la matérialisation en ligne des résultats.

Le fonctionnement de l'approche est illustré par l'algorithme 3

```
Input: noeud résultat n
Output: Lecture/Écriture des données du résultat n
if opération = Lecture then
    if n réside dans le cache RAM then
        Retourner les données du résultat n du cache RAM;
        maj_frequence(n); //mettre à jour la fréquence de n
    else
        if n réside dans le cache flash then
            Retourner les données de n du cache flash;
            maj_frequency(n);
        end
    end
end
if opération = écriture then
    if cache RAM contient de l'espace then
        garder les données du résultat n dans le cache RAM;
    else
        if Flash contient de l'espace libre then
            mettre les données du résultat n dans le cache Flash;
        else
            TrouveVictim(n); // remplacer le noeud le moins fréquemment utilisé par n
        end
    end
end
```

Algorithm 3: L/E gestionnaire de cache LFU naif

4.2 Approche basées sur l'éligibilité au cache

Dans cette approche, les noeuds ne sont pas caché aveuglement comme l'approche naïve. Il sont classés selon leur taille, coût d'exécution, informations LRU, LFU...

Avant de présenter l'approche, nous donnons les fonctions de calcul du coût d'exécution d'une requête en prenant en considération la présence de certains noeuds dans les caches $\mathcal{RAM}$ et Flash ainsi que la fonction d'éligibilité d'un noeud pour être caché ou supprimé du cache. Ces fonctions sont générales, leur raffinement selon le type de la base de donnée ou le support de stockage utilisé ont été présentés dans le chapitre 3.

4.2.1 Coût d'exécution d'un noeud

Etant donné C un ensemble de noeuds qui se trouvent dans le cache $\mathcal{RAM}$ ou Flash et n un noeud à exécuter. le coût d'exécution de n est :

$$cout(n) = coutcalcul(n) + \sum_{\forall d \in C} Cout(d_i) \tag{2}$$

où le coût de calcul des descendant d_i de n est:

$$cout(d_i) = \begin{cases} 0 & : \text{si } d_i \text{ est stocké en RAM} \\ cout_f(d_i) & : \text{coût de lecture de } d_i \text{ de la flash} \\ cout_h(d_i) & : \text{coût de re-calcul de } d_i \text{ du HDD} \end{cases}$$

4.2.2 Éligibilité d'un noeud pour le cache

Les caches sont limités en terme de leur taille. Le choix des objets à mettre ou à éliminer du cache est le problème majeur du gestionnaire de cache. Dans le cadre des caches classiques qui opèrent au niveau de la page, la politique utilisée est très souvent simple comme $\mathcal{LFU}$ ou $\mathcal{LRU}$. Ces politiques maintiennent pour chaque page du cache son nombre d'utilisations et sa dernière occurrence, et évince respectivement la page la moins utilisée ou dont la dernière occurrence est la plus ancienne. Dans le cadre des entrepôts de données et l'optimisation multi-requêtes, ces politique ne sont pas efficaces pour les raisons suivantes:

1. La taille des noeuds qui est nettement supérieure à la taille d'une page
2. La taille différentes des pages
3. Le coût de calcul qui diffère d'un noeud à un autre

Pour mettre en cache ou évincer un noeud du cache, nous utilisons une métrique similaire à celle définie dans [**?**]. Notre métrique intègre quatre paramètres relatifs à la taille du noeud, son coût de calcul et son nombre de référencements par les requêtes et le temps de son dernier accès. Ces paramètre sont montrés dans le tableau 4.4

Paramètre	Signification
$taille(n)$	taille du noeud
$cout(n)$	cout de calcul du noeud
f_n	fréquence d'accès du noeud
$t_{dernier}$	temps du dernier accès au noeud

TABLE 4.4 – Paramètre d'admission et d'éviction du cache

L'objectif est de mettre en cache les noeuds les plus fréquents et qui nécessitent un grand coût de calcul et aussi de garder dans le cache les noeuds de taille plus grande pour diminuer le coût d'écriture sur la flash lors des évictions des noeuds. L'éligibilité d'un noeud pour le cache dépend donc, proportionnellement de son coût de calcul, son facteur $\mathcal{LFU}$ et de son facteur $\mathcal{LRU}$ et inversement de sa taille . La mesure d'éligibilité devient:

$$eligibilte(n) = f_n * t_{dernier} * \frac{cout(n)}{taille(n)} \tag{3}$$

4.2.3 Mise en cache des noeuds

Lorsqu'une nouvelle requête arrive, son plan d'exécution optimal est généré par l'optimiseur de requêtes. L'arbre représentant le plan est ensuite intégré au $\mathcal{MVPP}$. Pour exécuter la requête, l'exécuteur de requêtes sollicite les gestionnaires de caches pour déterminer les noeuds déjà matérialisés. L'exécution de la requête s'effectue en trois étapes (voir l'algorithme 4)

1. Déterminer les noeuds cachées qui peuvent être utilisées pour l'exécution du nouveau noeud
2. Déterminer s'il y a un nouveau noeud à matérialiser
3. Éxécuter la requête

4.3 Combinaison de l'approche d'éligibilité avec le mode statique

La phase d'apprentissage du cache expliqué dans la section 3.1 permet de mettre des résultats intermédiaires dans le cache en mode hors-ligne ce qui ne pourra avoir aucun effet négatif sur le temps d'exécution des requêtes en mode en-ligne. La mise en cache des résultats dans cette phase est similaire à la sélection des vues matérialisées. Tous les noeuds du MVPP peuvent être candidats à cette sélection. L'approche par éligibilité donne la priorité pour le cache au noeuds de petite taille et de coût de calcul élevé. Des noeuds de grande tailles comme les jointures n'ont pas une forte chance pour être matérialisées. Pour pallier cette lacune, nous combinons l'approche par éligibilité avec l'approche statique pour sélectionner les noeuds de grandes tailles comme les jointures pour les mettre en cache sur la flash. Les noeuds de grande tailles une fois mises dans le cache ont une forte chance d'y rester longtemps sans être évincés.

```
Input: q:requête
Output: ensemble N de noeuds présent dans les caches et le noeud n à cacher
L = ∅;
n_e: noeud;
int Tempeligible=0;
foreach node n do
    if eligible(n) > Tempeligible then
        n_e=n;
    end
    if (n est dans le cache RAM) ou (n est dans le cache Flash) then
        L = L ∪ n; // ajouter le noeud n à la liste
    end
end
L = L ∪ n_e;
Executer(q, L); // exécuter q en utilisant les noeuds matérialisées de L
```

Algorithm 4: Fonctionnement du Buffer Manager

Nous allons voir dans la section 5 l'apport très positif de cette combinaison sur la performance globale des requêtes.

5 Étude expérimentale

5.1 Star Schema Benchmark

Pour évaluer notre approche, nous avons utilisé le banc d'essai $\mathcal{SSB}$ (Star Schema Benchmark)[**?**]. Il représente une version dé-normalisée du banc d'essai TPC-H [16]. Il utilise un schéma en étoile comme modèle logique (Figure 4.10). Il est composé d'une table de faits *Lineorder* et quatre tables de dimensions *Part, Supplier, Date* et *Customer*. Il supporte plusieurs niveaux de passage à l'échelle (scale factor) et peut générer un volume de données qui peut atteindre 600 GO. Les cardinalités des tables du banc d'essai peuvent être calculées comme suit:

— Lineorder = $6.000.000 * SF$
— Date = 2.556
— Part = $200.000 * (1 + \log_2 SF)$
— Supplier = $2.000 * SF$
— Customer = $30.000 * SF$

Pour nos besoins d'expérimentation, nous avons définis le *Scale Factor* (SF) à 10, ce qui a généré un volume de données de 10 Go. La taille de chaque table en terme d'instances, avec $SF = 10$, est décrite dans la table 4.5.

16. www.tpc.org

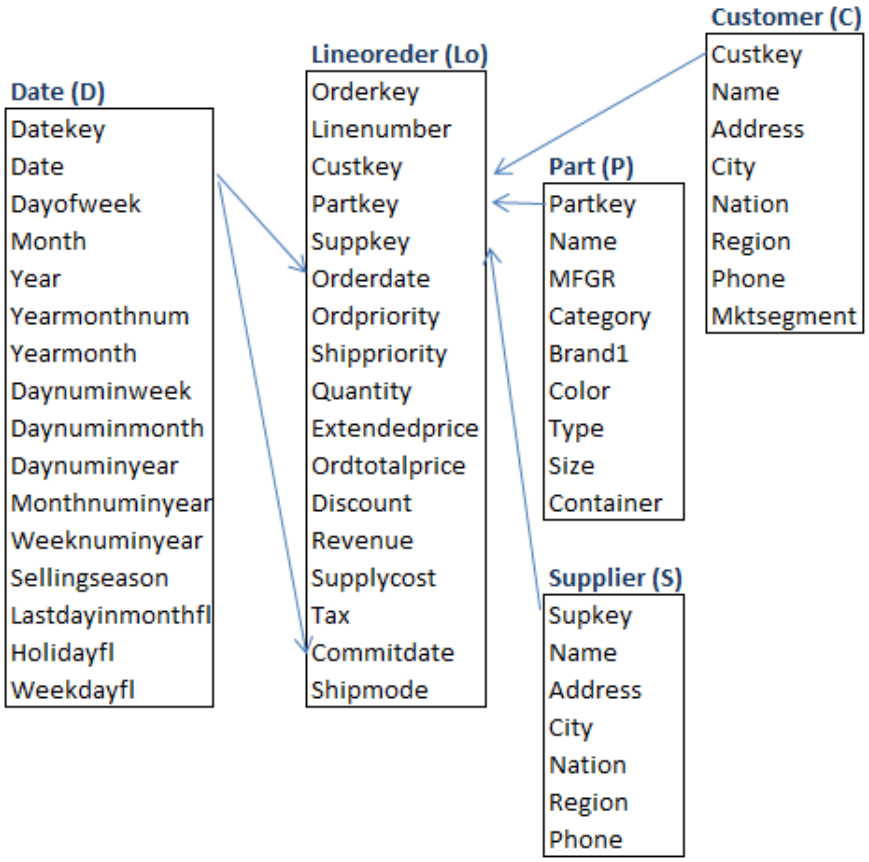


FIGURE 4.10 – Schéma du *SSB*

5.2 Charge de requêtes et environnement physique

Nous avons utilisé une charge de requêtes composée de trente (30) requêtes $\mathcal{SQL}$, numérotées de Q1 à Q30. Ces requêtes sont générées à partir des modèles de requêtes du banc d'essai SSB avec un facteur d'échelle égal à 10. Elles englobent des opérations de différents types: opération de sélection de type égalité et intervalles, des jointures, agrégations et tris. Les requêtes types sont décrits dans l'annexe A. Pour valider les résultats de notre modèle de coût, nous avons utilisé le système de gestion de bases de données *Orale 11g*. La configuration matérielle utilisée est la suivante :

— Micro-ordinateur : HP xw4600 Workstation
— Processeur : Quad processor Q6700 2,66 Ghz
— OS : Windows Vista Professional SP1, 64 bits
— RAM : 4 Go
— HDD : Seagate Barracuda 250 Go.
— SSD: Sumsung 64Go.

Tables	Nombre d'enregistrement	Taille d'un enregistrement(unités?)
Lineorder	60.000.000	355
Date	2.556	320
Part	800.000	144
Supplier	20.000	159
Customer	300.000	139

TABLE 4.5 – Taille des tables

5.3 Test des algorithmes

Dans cette section nous présentons les différents tests effectués dans le cas du banc d'essai SSB, les résultats et leurs interprétations. Pour montrer l'effet négatif de la matérialisation en ligne des résultats intermédiaires de taille importante, nous avons forcé la mise en cache de l'opération de scan de la table de fait lineoder pour la requête q1. La figure 4.11 montre que le temps d'exécution de la requête à augmenté de 2,60 ordres de grandeurs. La figure 4.12 montre

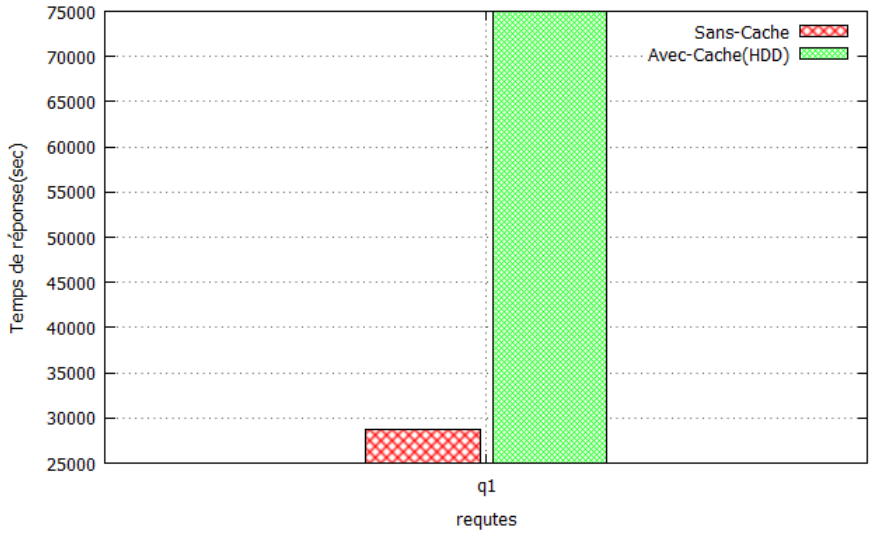


FIGURE 4.11 – Effet négatif du cache en ligne

le deuxième inconvénient de l'approche en ligne avec des caches de petite tailles. Dans ce cas nous avons testé l'approche avec l'algorithme $\mathcal{LFU}$ avec un cache de 4Go (1Go de RAM et 3 Go) de mémoire flash.

La figure 4.13 montre les résultats de test de l'approche par éligibilité pour des tailles de cache variées. Pour montrer l'efficacité du système de cache nous avons réalisé les tests sur 100 requêtes $\mathcal{SQL}$ générées sur la base du modèle de requêtes $\mathcal{SSB}$. L'expérience montre que l'intégration de la mémoire flash comme extension de la mémoire centrale avec contrôle des résultats à mettre en cache donne une réduction importante du temps d'exécution de la charge de requêtes.

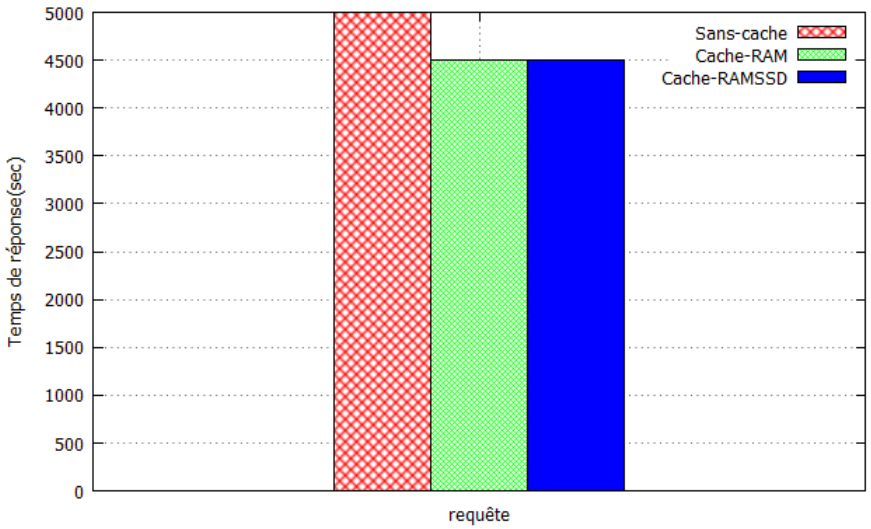


FIGURE 4.12 – Effet négatif du cache en ligne

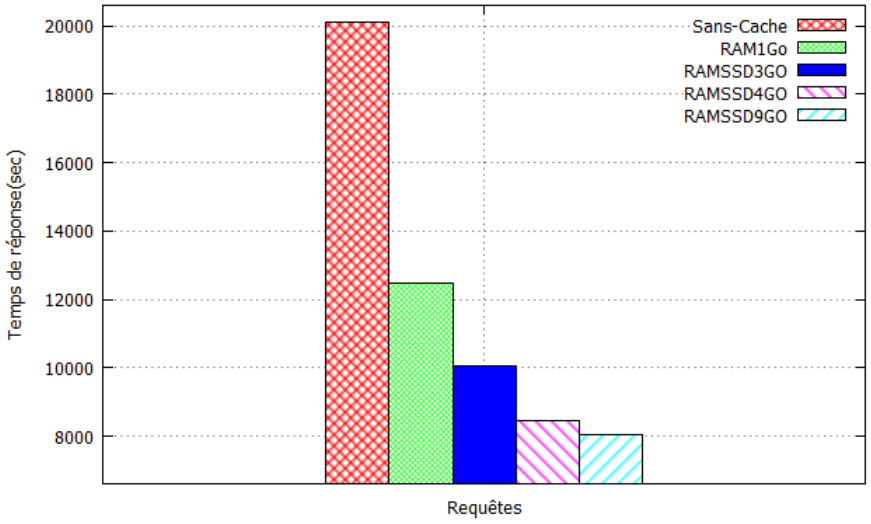


FIGURE 4.13 – Temps d'exécution de la charges de requêtes par l'approche par éligibilité

Les résultats intermédiaires de tailles importante sont très coûteux en terme de matérialisation. Nous avons utilisé une heuristique qui consiste à les stocker en mode statique pour ne pas ralentir l'exécution des requêtes. Les requêtes sélectionnées sont toutes de type jointure entre la table de fait et les tables de dimensions. Avec une contrainte d'espace de stockage de 15GO, nos algorithmes de sélection statique ont pu matérialiser la jointure entre la table de fait lineorder et la table de dimension date qui est de taille 11,0625 Go. La figure 4.14 montre l'amélioration du temps d'exécution de la charge de requête après cette matérialisation statique.

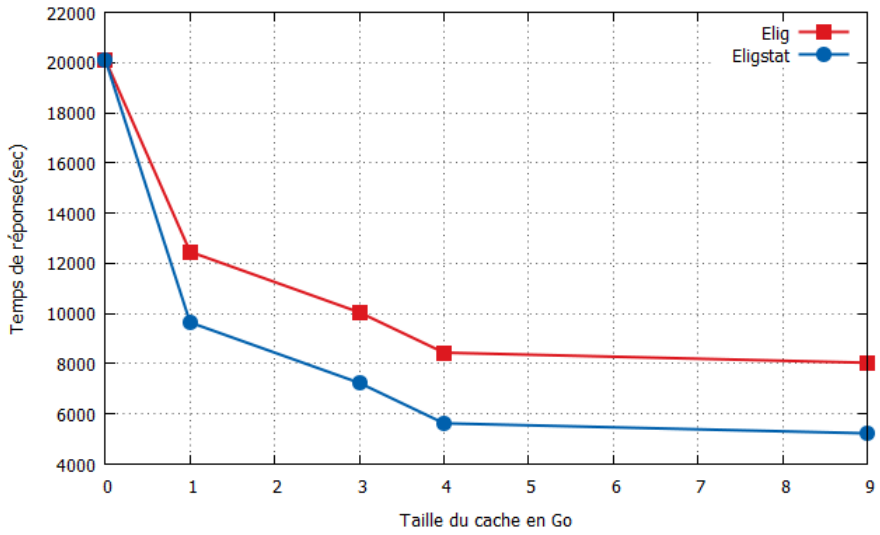


FIGURE 4.14 – Impact de la matérialisation statique

5.4 Validation sous Oracle 11g

Pour montrer l'apport de la matérialisation des résultats des requêtes sur la mémoire flash, nous avons extrait des séquences de la trace de mise en cache effectués sur la base du modèle de coût, ensuite nous avons crée manuellement un script SQL sous oracle 11g qui permet d'écrire et de lire les résultats sur le disque $\mathcal{SSD}$ et le disque $\mathcal{HDD}$ (voir la section 5.2). Les figures 4.15, 4.16 montrent les différences entre la matérialisation des résultats sur $\mathcal{HDD}$ et $\mathcal{SSD}$ et leurs lectures. Pour éviter l'écriture aléatoire qui est un mauvais scénario pour le disque $\mathcal{SSD}$, nous avons chargé les données séquentiellement avec des lots d'opérations d'insertion successives. Les tests montrent un gain d'environ 24% en lecture avec un temps d'accès en écriture presque similaire pour les disques $\mathcal{SSD}$ et $\mathcal{HDD}$.

6 Conclusion

Dans ce chapitre nous nous sommes focalisé sur l'optimisation des requêtes dans les entrepôts de données par les techniques de cache de données. Les requêtes multidimensionnelles traitent des volumes importants de données. Une mise en cache classique basée sur des pages isolées n'est pas efficace et crée un retardement d'exécution des requêtes à cause des traitements supplémentaires effectués par le processeur pour gérer chaque page indépendamment des autres. Les requêtes exécutées sur les entrepôts de données possèdent la propriété de partager un grand nombre de pages. Ces pages communes viennent du fait que les requêtes partagent souvent une ou plusieurs sous-expressions au niveau de la même requête ou pour des requêtes différentes. Dans ce contexte, il est profitable de concevoir des techniques de mise en cache de données qui exploitent ces possibilités de partage entre les requêtes. La mémoire centrale est un composant

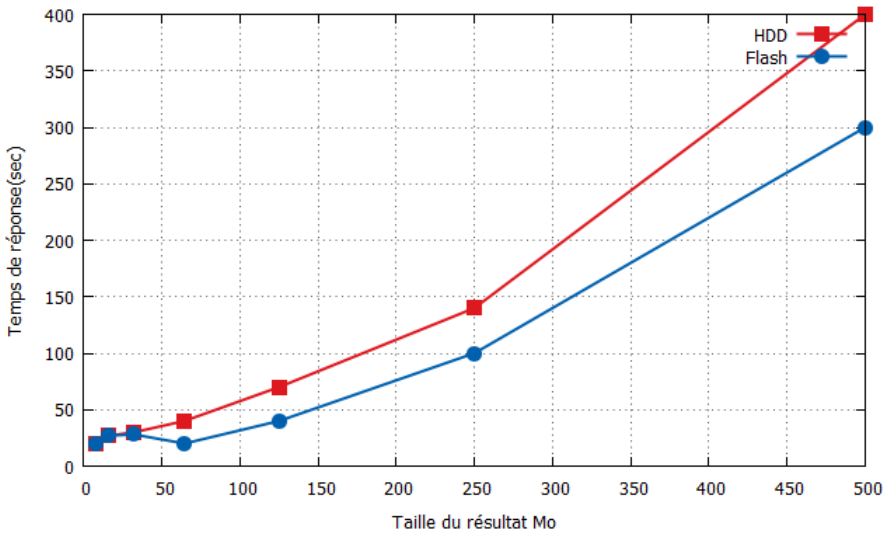


FIGURE 4.15 – Lecture des résultat de la flash et HDD

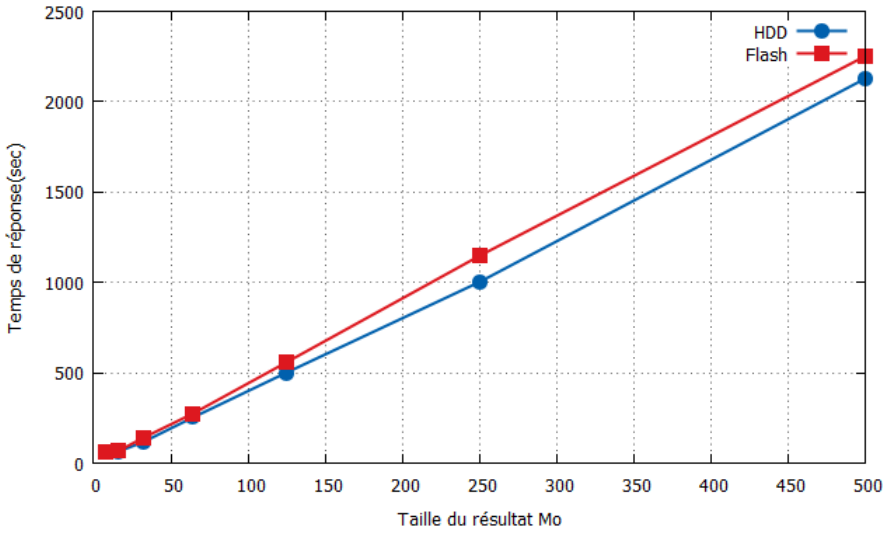


FIGURE 4.16 – Écriture des résultat de la flash et HDD

utilisé par toutes les applications d'un système informatique. C'est une ressource très importante qui nécessite d'être utilisée d'une manière efficace sans dépasser sa capacité. Pour arriver à ces fins, nous avons proposé un système de cache hybride qui utilise la mémoire flash comme extension du cache de la mémoire centrale. Cette hybridation permet d'augmenter la taille du cache en utilisant un support rapide comme la mémoire flash. Notre technique permet de donner une seconde chance aux résultats évincés du cache de la mémoire centrale pour être utilisé encore par les requêtes qui se présente au système. Notre approche permet de mettre en cache les résultats en commun selon leur mérite qui dépend essentiellement de leur taille et leur coût de calcul. La phase de mise en cache statique permet de sélectionner les plus grands résultats selon leur contribution pour les requêtes et les mettre directement sur la mémoire flash.

Chapitre 5

Conclusion Générale et perspectives

Sommaire

1 Conclusion 125
2 Perspectives 128

Dans ce chapitre, nous présentons un bilan général synthétisant nos principales contributions, ainsi qu'un ensemble de perspectives qui s'ouvrent à l'issue de notre étude.

1 Conclusion

La conception d'un entrepôt de données est une tâche complexe, du fait qu'elle nécessite un ensemble de phases complémentaires, à savoir l'analyse des besoins, la phase conceptuelle, la phase ETL, la phase logique, la phase de déploiement, la phase physique et la phase d'exploitation. Pour simplifier cette conception, des efforts de description des entrées de chaque phase ont été effectués. Ces efforts ont concerné principalement les phases de construction, où des méta modèles et des ontologies ont été proposés afin de simplifier le processus de correspondances entre les sources de données participant au processus de construction de l'entrepôt de données ainsi que les tâches d'ETL. Ces efforts ont été principalement effectués pour améliorer la qualité des données de l'entrepôt (*garbage in garbage out*). La phase physique n'a pas connu la même attention; bien qu'elle soit toujours associée à l'image de l'entrepôt de données, car les décideurs exigent un temps de réponse raisonnable lors de leur accès aux données. Les travaux de cette thèse ont essayé de satisfaire les deux critères dans l'environnement des entrepôts de données à savoir la qualité des données et la satisfaction du temps de réponse.

Pour satisfaire le deuxième critère la phase physique doit être revisitée afin de prendre en compte les progrès technologiques qui ont été menées durant les deux dernières décennies en termes de supports de stockage et des plateformes de déploiement. La qualité des solutions retournées par cette phase est souvent mesurée par l'utilisation des modèles de coût mathématiques prenant en compte toutes les dimensions de la technologie de base de données. Malheureusement, les paramètres de ces dimensions ne sont pas bien structurés et ils manquent de description.

Le fruit de cette thèse est le reflet des travaux menés dans notre équipe qui couvrent l'ensemble des phases du cycle de vie de conception des entrepôts de données. Cette expérience nous a aidé à reproduire les efforts de modélisation et d'ontologies développés de la phase de construction à la phase d'exploitation d'entrepôts de données.

Dans cette thèse un ensemble de contributions a été proposé: (i) l'identification de l'ensemble de dimensions nécessaires pour développer des modèles de coût mathématiques pour la phase physique, (ii) la méta modélisation de chaque dimension; (iii) la proposition d'une ontologie de domaine dédiée aux supports de stockage, (iv) l'extension de la mémoire cache par les mémoires flashs afin d'augmenter la capacité du cache et (v) la validation de nos contributions théoriquement et sur Oracle.

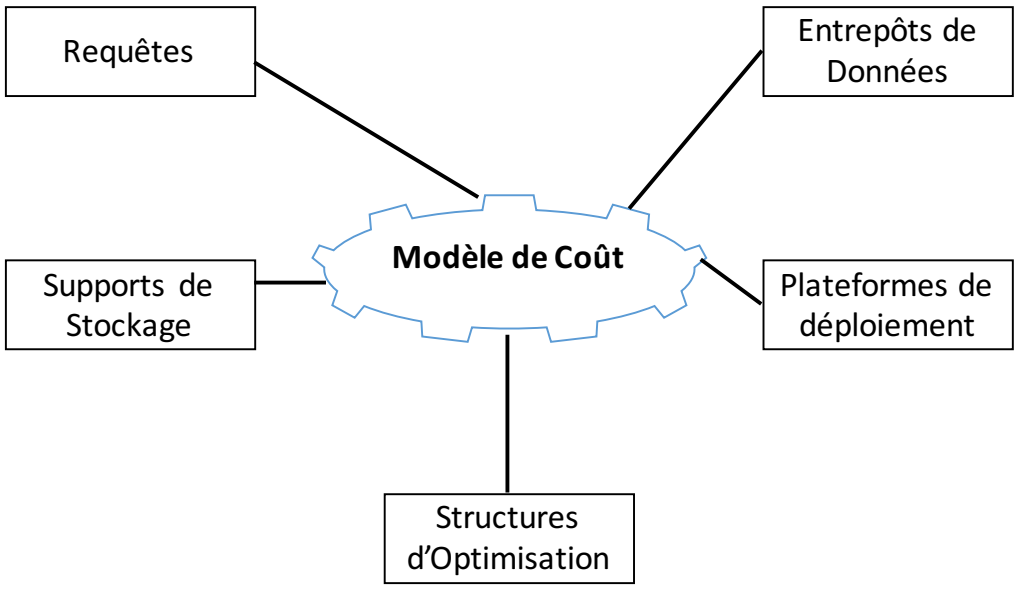


FIGURE 5.1 – Schéma en étoile d'un modèle de coût.

1.1 Identification des dimensions

Après une analyse approfondie des modèles de coût proposés à travers des générations de bases de données, nous avons pu identifier l'ensemble des dimensions que nous considérons pertinentes lors du développement de tout modèle de coût. Nous avons en conséquence identifié cinq principales dimensions: les supports de stockage, les requêtes, le schéma de la base de données, les structures d'optimisation et la plateforme de déploiement. Chaque dimension est décrite par un ensemble de propriétés dont certaines sont utilisées par les modèles de coût.

1.2 Méta modélisation des dimensions

Afin de rester sur la lignée Commun Warehouse Model (du standard de l'OMG), nous avons proposé des méta modèles pour chaque dimension afin de fournir aux développeurs de modèles de coût une vision à la fois globale et modulaire de l'ensemble de paramètres susceptibles d'être utilisés par ces modèles. Ces efforts nous ont permis d'avoir un schéma en étoile pour les modèles de coût (Figure 5.1). L'identification de l'ensemble de paramètres a été effectuée par l'étude des évolutions de chaque dimension qui coïncide avec les générations de bases de données et les progrès technologiques.

1.3 Une ontologie de supports de stockage

Lors de la catégorisation de l'ensemble de paramètres pertinents pour la construction d'un modèle de coût, nous avons facilement identifié le besoin de définir une ontologie de domaine

pour les supports de stockage. Cela est dû à la hiérarchisation qui existe entre ces supports; ce qui augmente la présence de relations de spécialisation/génération entre les supports de stockage. Vu la complexité de construction d'une ontologie, nous avons alors suivi une approche naïve qui consiste à examiner l'ensemble des paramètres des supports de stockage que nous pouvons facilement trouver dans les catalogues et les modes d'emploi de ces derniers. Ces paramètres ont été enrichis par d'autres que nous avons identifiés dans les articles scientifiques qui proposent des modèles de coût mathématiques. Cette ontologie a été décrite par l'éditeur Protégé (`protege.stanford.edu/`), et validée par les membres de l'équipe Ingénierie des Données et des Modèles du laboratoire LIAS.

La présence de l'ensemble des méta modèles correspondant à chaque dimension et l'ontologie de supports de stockage offre aux concepteurs et développeurs des paramètres à la carte. Ces derniers peuvent y piocher les dimensions et les paramètres qui les intéressent pour développer leur modèle de coût.

1.4 Hybridation du cache

La conséquence directe de nos efforts de développement d'ontologie pour les supports de stockage est l'identification de similarité entre eux. Cette similarité nous a poussé à combiner le cache mémoire avec les mémoires flashs pour augmenter sa capacité afin de stocker un nombre important de résultats intermédiaires de requêtes. Rappelons que dans le contexte des entrepôts de données, les requêtes OLAP présentent une forte interaction, qui augmente la présence des résultats intermédiaires (sous-expressions communes). Traditionnellement, ces derniers sont souvent matérialisés, et en conséquence stockés sur le disque. La taille du cache de la mémoire centrale étant limitée, nous l'avons étendu avec la mémoire flash. Ce choix est justifié par plusieurs raisons: (1) étant donné qu'il n'existe pas un algorithme de gestion de cache efficace pour toutes les situations, prévoir une solution matérielle peut être un enjeu important; (2) Les bonnes performances en lecture de la mémoire flash; (3) les capacités accrues de mémoire flash et son prix qui en baisse; et (4) le profil orienté lecture des requêtes décisionnelles (peu de mise à jour en ligne). Les paramètres variés relatifs à notre problématique comme les périphériques de stockage hétérogènes (RAM, Flash et disque dur) et le mode de stockage qui peut être de type ligne ou colonne nous a amené à instancier notre modèle de coût afin qu'il puisse prendre en compte les paramètres multiples des bases de données et les supports de stockage. Nous avons proposé par la suite une méthodologie pour l'optimisation des requêtes et la gestion du cache hybride sur la base de ce modèle de coût générique. Notre moteur d'exécution des requêtes, lit les pages nécessaires pour la réponse à une requête à partir du support de stockage principal (disque dur HDD), garde les résultats les plus importants pour l'exécution de toute la charge de requêtes dans le cache de la mémoire centrale. En cas de saturation du cache de la mémoire centrale, les résultats les moins importants sont sélectionnés pour être évincés vers le cache de la mémoire flash. L'importance des résultats pour le cache est calculée par une fonction qui favorise les résultats les plus coûteux en termes de temps de calcul, les plus petits

en taille, les plus fréquemment utilisés et enfin les plus récemment utilisés. Nous avons montré que, pour que le système de cache proposé soit efficace, il est nécessaire de prendre en considération deux modes de cache (1) statique et (2) dynamique. Dans le mode de cache statique, les résultats importants de grande taille, en particulier les jointures les plus fréquentes, sont mis directement sur la mémoire flash après une sélection préalable. Le mode de cache dynamique permet de mettre en cache les autres types de résultats et de gérer le cache d'une manière dynamique au fur et à mesure que les requêtes se présentent à l'exécuteur de requête. Nous avons développé un simulateur qui permet d'assister l'administrateur de la base de données pour résoudre des problèmes liés à la conception physique. Le simulateur se connecte à une base de données réelle via l'interface de programme (API) JDBC et récupère toutes les méta-données sur l'entrepôt. Le simulateur permet de faire l'ordonnancement des requêtes et leurs exécutions dans un environnement de cache hybride.

2 Perspectives

De nombreuses perspectives tant à caractère théorique que pratique peuvent être envisagées. Dans cette section nous présentons succinctement celles qui nous paraissent être les plus intéressantes.

2.1 Validation de l'ontologie

Certes nous avons construit une ontologie pour les supports de stockage; mais sa validation a été effectuée par les membres de notre équipe pour répondre à des besoins d'optimisation particuliers. Cette validation concernait l'usage de cette ontologie. Mais la complétude de son contenu n'a pas été établie. Pour ce faire, nous envisageons de faire appel à des experts de supports de stockage. Une fois validée, nous souhaitons la standardiser et surtout l'intégrer dans le méta modèle CMW.

2.2 Une ontologie par dimension

Dans cette thèse, nous avons proposé une ontologie pour une seule dimension représentant les supports de stockage. Cette réflexion peut être généralisée pour les autres dimensions: les bases de données, les requêtes, les plateformes de déploiement, et les structures d'optimisation. Les méta modèles de chaque dimension que nous avons définis représente une base de travail pour construire des telles ontologies [**?**].

2.3 Substitution de supports de stockage

Dans ce travail, nous avons proposé une ontologie de domaine sans exploiter les mécanismes de raisonnement offerts par cette ontologie. Ces derniers sont souvent implantés dans des outils nommés raisonneurs tels que RACER [**?**] et Pellet [**?**]. Une des utilisations des raisonneurs dans notre contexte pourrait être la substitution d'un support de stockage par un autre en cas de panne, en exploitant leur similarité.

2.4 Développement d'un outil pour les modèles de coût

Nous avons mentionné dans les premiers chapitres l'existence d'un nombre important d'outils (advisors) pour la conception physique. Ces outils ont été développés par les éditeurs commerciaux de SGBD comme *Oracle*, *IBM DB2* et *SQL Server* afin de simplifier les tâches des administrateurs. Il serait primordial d'avoir des outils pour la partie développement de modèles de coût, dans lesquels tous les méta modèles et les ontologies décrivant les dimensions sont affichés, ce qui permet à un développeurs de naviguer à travers les hiérarchies de ces modèles afin de sélectionner ses paramètres favoris.

2.5 Développement d'un suite logicielle pour la conception globale des entrepôts de données

En explorant le marché des produits dédiés à la conception des entrepôts de données, nous réalisons la présence d'un nombre important d'outils consacrés à l'ensemble de tâches de la phase de construction: (1) l'analyse des besoins comme celle proposé par IBM: Rational DOORS qui permet d'optimiser la gestion des exigences en matière de communication, de vérification et de collaboration à tous les niveaux de votre organisation et de votre chaîne logistique. Rational DOORS permet de capturer, d'effectuer le suivi, d'analyser et de gérer les changements d'informations et de démontrer la conformité aux réglementations et normes. (2) Des outils de conception proprement dite: on peut citer l'exemple de rationale Rose. (3) ETL qui a vu l'explosion des outils industriels et académiques comme IBM Information Server InfoSphere, SAS Data Integration Studio Oracle Warehouse Builder (OWB), Sap BusinessObjects Data Integration, etc.) et Talend Open Studio. (4) Advisors pour la phase physique (p. ex. Oracle Database Advisors). Les modèles de coût de ces advisors sont cachés (propriétaires). Les développeurs ont du mal à paramétrer ces modèles de coût. (5) des outils requêtes/reporting (SAP).

Il serait intéressant d'intégrer tous les outils dans la même *suite logicielle intégrée* pour la conception d'un entrepôt de données.

2.6 Etude de passage à l'échelle et retour d'expérience des concepteurs

Dans la validation de nos contributions, nous avons considéré des plateformes centralisées, avec des modèles de coût simples. Il serait important de tester d'autres instances d'entrepôts de données stockées sur des plateformes avancées comme les machines parallèles ou le Cloud, en considérant d'autres paramètres comme les structures d'optimisation.

Rappelons que dans notre étude, nous avons impliqué une doctorante (Amira Kerkad) qui avait une forte expérience dans le développement et l'utilisation de modèles de coût. Une autre perspective, que nous devons étudier de près, est l'implication de plusieurs types profils de concepteurs (novices et experts) afin de collecter leur expérience lors de l'utilisation de nos modèles en termes de simplicité d'utilisation et la qualité des modèles obtenus.

2.7 Vers un système persistant des modèles de coût

La Figure 5.1 montre un schéma en étoile des modèles de coût. Dans le but d'augmenter la réutilisation et la reproduction de ces derniers, il serait intéressant de persister ce schéma. L'entrepôt obtenu peut stocker les résultats des tests ainsi que les formules utilisées par l'ensemble de modèles de coût utilisé pour une tâche particulière de la phase physique. La motivation principale d'avoir un tel entrepôt de données est le constat réalisé par le Prof. *Jens Dittrich*, de l'Université de Saarbrücken, Allemagne. Ce dernier a pointé du doigt la difficulté des chercheurs de reproduire les résultats de certains travaux, ce qui remet en cause leur transparence et leur fiabilité. Ces travaux de thèse de Abdelkader Ouared et Lahcène Brahimi menés au laboratoire LIAS s'intéressent de près à ces problématiques. Ils ont ont donné lieu à plusieurs publications [**?**, **?**]

Annexe A

Annexe : Charge de requêtes

Dans cette annexe, nous présentons les modèles de requêtes du banc d'essai Star Schema Benchmark. Pour nos besoin d'expérimentation, nous avons généré plusieurs ensembles de requêtes de ces modèles pour évaluer nos algorithmes. L'ontologie des supports de stockage est également décrite en OWL.

```
--Q1.1 :
select sum(lo_extendedprice*lo_discount) as revenue
from lineorder, date
where lo_orderdate = d_datekey
and d_year = 1993
and lo_discount between1 and 3
and lo_quantity < 25;
--Q1.2 :
select sum(lo_extendedprice*lo_discount) as revenue
from lineorder, date
where lo_orderdate = d_datekey
and d_yearmonthnum = 199401
and lo_discount between4 and 6
and lo_quantity between 26 and 35;
--Q1.3 :
select sum(lo_extendedprice*lo_discount) as revenue
from lineorder, date
where lo_orderdate = d_datekey
and d_weeknuminyear = 6
and d_year = 1994
and lo_discount between 5 and 7
and lo_quantity between 26 and 35;
--Q2.1:
```

```
select sum(lo_revenue), d_year, p_brand1
from lineorder, date, part, supplier
where lo_orderdate = d_datekey
and lo_partkey = p_partkey
and lo_suppkey = s_suppkey
and p_category = 'MFGR#12'
and s_region = 'AMERICA'
group by d_year, p_brand1
order by d_year, p_brand1;
--Q2.2:
select sum(lo_revenue), d_year, p_brand1
from lineorder, date, part, supplier
where lo_orderdate = d_datekey
and lo_partkey = p_partkey
and lo_suppkey = s_suppkey
and p_brand1 = 'MFGR#2221'
and s_region = 'EUROPE'
group by d_year, p_brand1
order by d_year, p_brand1;
--Q3.1 :
select c_nation, s_nation, d_year, sum(lo_revenue) as revenue
from customer, lineorder, supplier, date
where lo_custkey = c_custkey
and lo_suppkey = s_suppkey
and lo_orderdate = d_datekey
and c_region = 'ASIA' and s_region = 'ASIA'
and d_year >= 1992 and d_year <= 1997
group by c_nation, s_nation, d_year
order by d_year asc, revenue desc;
--Q3.2 :
select c_city, s_city, d_year, sum(lo_revenue) as revenue
from customer, lineorder, supplier, date
where lo_custkey = c_custkey
and lo_suppkey = s_suppkey
and lo_orderdate = d_datekey
and c_nation = 'UNITED STATES'
and s_nation = 'UNITED STATES'
and d_year >= 1992 and d_year <= 1997
group by c_city, s_city, d_year
order by d_year asc, revenue desc;
```

```
--Q3.3 :
select c_city, s_city, d_year, sum(lo_revenue) as revenue
from customer, lineorder, supplier, date
where lo_custkey = c_custkey
and lo_suppkey = s_suppkey
and lo_orderdate = d_datekey
and (c_city='UNITED KI1'
or c_city='UNITED KI5')
and (s_city='UNITED KI1'
or s_city='UNITED KI5')
and d_year >= 1992 and d_year <= 1997
group by c_city, s_city, d_year
order by d_year asc, revenue desc;
--Q3.4 :
select c_city, s_city, d_year, sum(lo_revenue) as revenue
from customer, lineorder, supplier, date
where lo_custkey = c_custkey
and lo_suppkey = s_suppkey
and lo_orderdate = d_datekey
and (c_city='UNITED KI1' or c_city='UNITED KI5')
and (s_city='UNITED KI1' or s_city = 'UNITED KI5')
and d_yearmonth = 'Dec1997'
group by c_city, s_city, d_year
order by d_year asc, revenue desc;
--Q4.1 :
select d_year, c_nation, sum(lo_revenue - lo_supplycost) as profit
from date, customer, supplier, part, lineorder
where lo_custkey = c_custkey
and lo_suppkey = s_suppkey
and lo_partkey = p_partkey
and lo_orderdate = d_datekey
and c_region = 'AMERICA'
and s_region = 'AMERICA'
and (p_mfgr = 'MFGR#1' or p_mfgr = 'MFGR#2')
group by d_year, c_nation
order by d_year, c_nation
--Q4.2 :
select d_year, s_nation, p_category, sum(lo_revenue - lo_supplycost) as profit
from date, customer, supplier, part, lineorder
where lo_custkey = c_custkey
```

```
and lo_suppkey = s_suppkey
and lo_partkey = p_partkey
and lo_orderdate = d_datekey
and c_region = 'AMERICA'
and s_region = 'AMERICA'
and (d_year = 1997 or d_year = 1998)
and (p_mfgr = 'MFGR#1'
or p_mfgr = 'MFGR#2')
group by d_year, s_nation, p_category
order by d_year, s_nation, p_category
--Q4.3 :
select d_year, s_city, p_brand1, sum(lo_revenue - lo_supplycost) as profit
from date, customer, supplier, part, lineorder
where lo_custkey = c_custkey
and lo_suppkey = s_suppkey
and lo_partkey = p_partkey
and lo_orderdate = d_datekey
and c_region = 'AMERICA'
and s_nation = 'UNITED STATES'
and (d_year = 1997 or d_year = 1998)
and p_category = 'MFGR#14'
group by d_year, s_city, p_brand1
order by d_year, s_city, p_brand1
```

Annexe B

Annexe : Ontologie des supports de stockages

L'ontologie des supports de stockage en OWL:

```
<?xml version="1.0"?>


<!DOCTYPE Ontology [
<!ENTITY xsd "http://www.w3.org/2001/XMLSchema#" >
<!ENTITY xml "http://www.w3.org/XML/1998/namespace" >
<!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#" >
<!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns#" >
]>


<Ontology xmlns="http://www.w3.org/2002/07/owl#"
xml:base="http://www.semanticweb.org/admin/ontologies/
2016/9/untitled-ontology-18"
xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:xml="http://www.w3.org/XML/1998/namespace"
ontologyIRI="http://www.semanticweb.org/admin/ontologies/2016/9/
untitled-ontology-18">
<Prefix name="rdf" IRI="http://www.w3.org/1999/02/22-rdf-syntax-ns#"/>
<Prefix name="rdfs" IRI="http://www.w3.org/2000/01/rdf-schema#"/>
<Prefix name="xsd" IRI="http://www.w3.org/2001/XMLSchema#"/>
<Prefix name="owl" IRI="http://www.w3.org/2002/07/owl#"/>
<Declaration>
<Class IRI="#Description"/>
```

```
</Declaration>
<Declaration>
<Class IRI="#Device"/>
</Declaration>
<Declaration>
<Class IRI="#DeviceStatus"/>
</Declaration>
<Declaration>
<Class IRI="#Flash"/>
</Declaration>
<Declaration>
<Class IRI="#FlashCtrl"/>
</Declaration>
<Declaration>
<Class IRI="#FlashDataConf"/>
</Declaration>
<Declaration>
<Class IRI="#HDD"/>
</Declaration>
<Declaration>
<Class IRI="#HardwareDescription"/>
</Declaration>
<Declaration>
<Class IRI="#InternalCache"/>
</Declaration>
<Declaration>
<Class IRI="#QoS"/>
</Declaration>
<Declaration>
<Class IRI="#RAM"/>
</Declaration>
<Declaration>
<Class IRI="#SoftwareDescription"/>
</Declaration>
<Declaration>
<Class IRI="#Taccess"/>
</Declaration>
<Declaration>
<ObjectProperty IRI="#hasAccess"/>
</Declaration>
```

```
<Declaration>
<ObjectProperty IRI="#hasController"/>
</Declaration>
<Declaration>
<ObjectProperty IRI="#hasDataSize"/>
</Declaration>
<Declaration>
<ObjectProperty IRI="#hasDescription"/>
</Declaration>
<Declaration>
<ObjectProperty IRI="#hasHardwareDesrip"/>
</Declaration>
<Declaration>
<ObjectProperty IRI="#hasInternalCache"/>
</Declaration>
<Declaration>
<ObjectProperty IRI="#hasQos"/>
</Declaration>
<Declaration>
<ObjectProperty IRI="#hasSoftwareDescrip"/>
</Declaration>
<Declaration>
<ObjectProperty IRI="#hasStatus"/>
</Declaration>
<Declaration>
<DataProperty IRI="#Bandwith"/>
</Declaration>
<Declaration>
<DataProperty IRI="#Energie"/>
</Declaration>
<Declaration>
<DataProperty IRI="#Firme"/>
</Declaration>
<Declaration>
<DataProperty IRI="#FltType"/>
</Declaration>
<Declaration>
<DataProperty IRI="#GarbageColType"/>
</Declaration>
<Declaration>
```

```
<DataProperty IRI="#Modèle"/>
</Declaration>
<Declaration>
<DataProperty IRI="#Nom"/>
</Declaration>
<Declaration>
<DataProperty IRI="#NomSoft"/>
</Declaration>
<Declaration>
<DataProperty IRI="#Online"/>
</Declaration>
<Declaration>
<DataProperty IRI="#Online_depuis"/>
</Declaration>
<Declaration>
<DataProperty IRI="#RadomRead"/>
</Declaration>
<Declaration>
<DataProperty IRI="#RandomWrite"/>
</Declaration>
<Declaration>
<DataProperty IRI="#SeqRead"/>
</Declaration>
<Declaration>
<DataProperty IRI="#SeqWrite"/>
</Declaration>
<Declaration>
<DataProperty IRI="#Taille"/>
</Declaration>
<Declaration>
<DataProperty IRI="#TailleBlock"/>
</Declaration>
<Declaration>
<DataProperty IRI="#Taillepage"/>
</Declaration>
<Declaration>
<DataProperty IRI="#Throughput"/>
</Declaration>
<Declaration>
<DataProperty IRI="#TypeConnexion"/>
```

```
</Declaration>
<Declaration>
<DataProperty IRI="#Version"/>
</Declaration>
<Declaration>
<DataProperty IRI="#VersionSoft"/>
</Declaration>
<Declaration>
<DataProperty IRI="#capacité"/>
</Declaration>
<SubClassOf>
<Class IRI="#Flash"/>
<Class IRI="#Device"/>
</SubClassOf>
<SubClassOf>
<Class IRI="#HDD"/>
<Class IRI="#Device"/>
</SubClassOf>
<SubClassOf>
<Class IRI="#RAM"/>
<Class IRI="#Device"/>
</SubClassOf>
<ObjectPropertyDomain>
<ObjectProperty IRI="#hasAccess"/>
<Class IRI="#Device"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
<ObjectProperty IRI="#hasController"/>
<Class IRI="#Flash"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
<ObjectProperty IRI="#hasDataSize"/>
<Class IRI="#Flash"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
<ObjectProperty IRI="#hasDescription"/>
<Class IRI="#Device"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
<ObjectProperty IRI="#hasHardwareDesrip"/>
```

```
<Class IRI="#Device"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
<ObjectProperty IRI="#hasInternalCache"/>
<Class IRI="#Device"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
<ObjectProperty IRI="#hasQos"/>
<Class IRI="#Device"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
<ObjectProperty IRI="#hasSoftwareDescrip"/>
<Class IRI="#Device"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
<ObjectProperty IRI="#hasStatus"/>
<Class IRI="#Device"/>
</ObjectPropertyDomain>
<ObjectPropertyRange>
<ObjectProperty IRI="#hasAccess"/>
<Class IRI="#Taccess"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
<ObjectProperty IRI="#hasController"/>
<Class IRI="#FlashCtrl"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
<ObjectProperty IRI="#hasDataSize"/>
<Class IRI="#FlashDataConf"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
<ObjectProperty IRI="#hasDescription"/>
<Class IRI="#Description"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
<ObjectProperty IRI="#hasHardwareDesrip"/>
<Class IRI="#HardwareDescription"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
<ObjectProperty IRI="#hasInternalCache"/>
```

```
<Class IRI="#InternalCache"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
<ObjectProperty IRI="#hasQos"/>
<Class IRI="#QoS"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
<ObjectProperty IRI="#hasSoftwareDescrip"/>
<Class IRI="#SoftwareDescription"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
<ObjectProperty IRI="#hasStatus"/>
<Class IRI="#DeviceStatus"/>
</ObjectPropertyRange>
<SubDataPropertyOf>
<DataProperty IRI="#Bandwith"/>
<DataProperty abbreviatedIRI="owl:topDataProperty"/>
</SubDataPropertyOf>
<DataPropertyDomain>
<DataProperty IRI="#Bandwith"/>
<Class IRI="#QoS"/>
</DataPropertyDomain>
<DataPropertyDomain>
<DataProperty IRI="#Energie"/>
<Class IRI="#HardwareDescription"/>
</DataPropertyDomain>
<DataPropertyDomain>
<DataProperty IRI="#Firme"/>
<Class IRI="#SoftwareDescription"/>
</DataPropertyDomain>
<DataPropertyDomain>
<DataProperty IRI="#FltType"/>
<Class IRI="#FlashCtrl"/>
</DataPropertyDomain>
<DataPropertyDomain>
<DataProperty IRI="#GarbageColType"/>
<Class IRI="#FlashCtrl"/>
</DataPropertyDomain>
<DataPropertyDomain>
<DataProperty IRI="#Modèle"/>
```

```
<Class IRI="#Description"/>
</DataPropertyDomain>
<DataPropertyDomain>
<DataProperty IRI="#Nom"/>
<Class IRI="#Description"/>
</DataPropertyDomain>
<DataPropertyDomain>
<DataProperty IRI="#NomSoft"/>
<Class IRI="#SoftwareDescription"/>
</DataPropertyDomain>
<DataPropertyDomain>
<DataProperty IRI="#Online"/>
<Class IRI="#DeviceStatus"/>
</DataPropertyDomain>
<DataPropertyDomain>
<DataProperty IRI="#Online_depuis"/>
<Class IRI="#DeviceStatus"/>
</DataPropertyDomain>
<DataPropertyDomain>
<DataProperty IRI="#RadomRead"/>
<Class IRI="#Taccess"/>
</DataPropertyDomain>
<DataPropertyDomain>
<DataProperty IRI="#RandomWrite"/>
<Class IRI="#Taccess"/>
</DataPropertyDomain>
<DataPropertyDomain>
<DataProperty IRI="#SeqRead"/>
<Class IRI="#Taccess"/>
</DataPropertyDomain>
<DataPropertyDomain>
<DataProperty IRI="#SeqWrite"/>
<Class IRI="#Taccess"/>
</DataPropertyDomain>
<DataPropertyDomain>
<DataProperty IRI="#Taille"/>
<Class IRI="#InternalCache"/>
</DataPropertyDomain>
<DataPropertyDomain>
<DataProperty IRI="#TailleBlock"/>
```

```
<Class IRI="#FlashDataConf"/>
</DataPropertyDomain>
<DataPropertyDomain>
<DataProperty IRI="#Taillepage"/>
<Class IRI="#FlashDataConf"/>
</DataPropertyDomain>
<DataPropertyDomain>
<DataProperty IRI="#Throughput"/>
<Class IRI="#QoS"/>
</DataPropertyDomain>
<DataPropertyDomain>
<DataProperty IRI="#TypeConnexion"/>
<Class IRI="#HardwareDescription"/>
</DataPropertyDomain>
<DataPropertyDomain>
<DataProperty IRI="#Version"/>
<Class IRI="#Description"/>
</DataPropertyDomain>
<DataPropertyDomain>
<DataProperty IRI="#VersionSoft"/>
<Class IRI="#SoftwareDescription"/>
</DataPropertyDomain>
<DataPropertyDomain>
<DataProperty IRI="#capacité"/>
<Class IRI="#Device"/>
</DataPropertyDomain>
<DataPropertyRange>
<DataProperty IRI="#Bandwith"/>
<Datatype abbreviatedIRI="owl:real"/>
</DataPropertyRange>
<DataPropertyRange>
<DataProperty IRI="#Energie"/>
<Datatype abbreviatedIRI="owl:real"/>
</DataPropertyRange>
<DataPropertyRange>
<DataProperty IRI="#Firme"/>
<Datatype abbreviatedIRI="xsd:string"/>
</DataPropertyRange>
<DataPropertyRange>
<DataProperty IRI="#FltType"/>
```

```
<Datatype abbreviatedIRI="xsd:string"/>
</DataPropertyRange>
<DataPropertyRange>
<DataProperty IRI="#GarbageColType"/>
<Datatype abbreviatedIRI="xsd:string"/>
</DataPropertyRange>
<DataPropertyRange>
<DataProperty IRI="#Modèle"/>
<Datatype abbreviatedIRI="xsd:string"/>
</DataPropertyRange>
<DataPropertyRange>
<DataProperty IRI="#Nom"/>
<Datatype abbreviatedIRI="xsd:string"/>
</DataPropertyRange>
<DataPropertyRange>
<DataProperty IRI="#NomSoft"/>
<Datatype abbreviatedIRI="xsd:string"/>
</DataPropertyRange>
<DataPropertyRange>
<DataProperty IRI="#Online"/>
<Datatype abbreviatedIRI="xsd:string"/>
</DataPropertyRange>
<DataPropertyRange>
<DataProperty IRI="#Online_depuis"/>
<Datatype abbreviatedIRI="xsd:string"/>
</DataPropertyRange>
<DataPropertyRange>
<DataProperty IRI="#RadomRead"/>
<Datatype abbreviatedIRI="xsd:string"/>
</DataPropertyRange>
<DataPropertyRange>
<DataProperty IRI="#RandomWrite"/>
<Datatype abbreviatedIRI="xsd:string"/>
</DataPropertyRange>
<DataPropertyRange>
<DataProperty IRI="#SeqRead"/>
<Datatype abbreviatedIRI="xsd:string"/>
</DataPropertyRange>
<DataPropertyRange>
<DataProperty IRI="#SeqWrite"/>
```

```
<Datatype abbreviatedIRI="xsd:string"/>
</DataPropertyRange>
<DataPropertyRange>
<DataProperty IRI="#Taille"/>
<Datatype abbreviatedIRI="owl:real"/>
</DataPropertyRange>
<DataPropertyRange>
<DataProperty IRI="#TailleBlock"/>
<Datatype abbreviatedIRI="owl:real"/>
</DataPropertyRange>
<DataPropertyRange>
<DataProperty IRI="#Taillepage"/>
<Datatype abbreviatedIRI="owl:real"/>
</DataPropertyRange>
<DataPropertyRange>
<DataProperty IRI="#Throughput"/>
<Datatype abbreviatedIRI="owl:real"/>
</DataPropertyRange>
<DataPropertyRange>
<DataProperty IRI="#TypeConnexion"/>
<Datatype abbreviatedIRI="xsd:string"/>
</DataPropertyRange>
<DataPropertyRange>
<DataProperty IRI="#Version"/>
<Datatype abbreviatedIRI="xsd:int"/>
</DataPropertyRange>
<DataPropertyRange>
<DataProperty IRI="#VersionSoft"/>
<Datatype abbreviatedIRI="xsd:int"/>
</DataPropertyRange>
<DataPropertyRange>
<DataProperty IRI="#capacité"/>
<Datatype abbreviatedIRI="xsd:int"/>
</DataPropertyRange>
</Ontology>



<!-- Generated by the OWL API (version 3.4.2) http://owlapi.sourceforge.net -->
```

Table des figures

1.1 Les efforts d'établissement des correspondances entre les schémas locaux et global 4
1.2 Cycle de vie de conception d'un entrepôt de données 5
1.3 Un exemple de l'ontologie IEC 6
1.4 Correspondances entre les ontologies locales et globale 7
1.5 Différentes architectures pour la modélisation conceptuelle d'un entrepôt de données 8
1.6 L'évolution du niveau de la description 10
1.7 L'évolution du niveau de la description 11
1.8 Les éléments de la phase en Aval 14
1.9 Les paramètres d'un modèle de coût pour une seule instance de base de données 15
1.10 Les dimensions d'un modèle de coût 16
1.11 Les paramètres d'un modèle de coût pour plusieurs instances de bases de données 16
1.12 Organisation de la thèse 20

2.1 Evolution du prix de stockage 24
2.2 Matériel émargeant 24
2.3 Architecture d'un entrepôt de données 28
2.4 Représentation MOLAP par un tableau Multidimensionnel 28
2.5 Exemple d'un entrepôt de données modélisé par un schéma en étoile 29
2.6 Exemple d'un entrepôt de données modélisé par un schéma en flocon de neige . 29
2.7 *Architetcures matérielles usuelles* 35

2.8 stockage par lignes 36
2.9 stockage par colonnes 37
2.10 Exemple stockage par colonnes 38
2.11 Structure physique des disques durs 40
2.12 Les types du *SSD* 41
2.13 Architecture d'un disque SSD 42
2.14 Exemple de traduction d'adresse par page 45
2.15 Exemple de traduction d'adresse par bloc 46
2.16 traduction hybride d'adresse 47
2.17 Exemples de plan d'exécution de requêtes 49
2.18 Plan unifié de requêtes 51
2.19 Les techniques d'optimisation 53
2.20 Entrepôt de données pour les ventes 54
2.21 Exemple de fragmentation verticale de la table produit 58
2.22 Exemple de fragmentation horizontale primaire de la table Magasin 59
2.23 Exemple de fragmentation horizontale dérivée de la table « Vente » 59

3.1 Génération des modèles de coût 68
3.2 Couches à prendre en charge par un modèle de coût 68
3.3 Composantes du modèle de coût générique 70
3.4 Méta modèle de la base de données 71
3.5 Méta Modèle pour les requêtes 74
3.6 Méta modèle pour les structures d'optimisation 75
3.7 Meta-modèle de l'architecture de déploiement 76
3.8 Meta-modèle de stockage 77
3.9 Notre ontologie sous protégé 80
3.10 Système de stockage hiérarchique 82
3.11 Meta-modèle de stockage 84
3.12 Utilisation du modèle de coût générique par les algorithmes 85
3.13 Gestion du cache et ordonnancement des requêtes 87
3.14 Résultat de simulation théorique 92

3.15 Validation des résultats sous oracle 92

4.1 Le flash pour augmenter la capacité de stockage des supports traditionnels . . . 97
4.2 L'approche IPL (in page logging) [**?**] 99
4.3 PAX vs stockage en ligne (NMS: N-ary storage Model) [**?**] 99
4.4 B-tree: mise à jour d'un tuple [**?**] 101
4.5 μ-tree: mise à jour d'un tuple [**?**] 102
4.6 Architecture du cache hybride 104
4.7 Placement des données sur support: un problème de décision 106
4.8 Génétique Hill climbing 109
4.9 Génétique Hill climbing 109
4.10 Schéma du *SSB* 117
4.11 Effet négatif du cache en ligne 118
4.12 Effet négatif du cache en ligne 119
4.13 Temps d'exécution de la charges de requêtes par l'approche par éligibilité . . . 119
4.14 Impact de la matérialisation statique 120
4.15 Lecture des résultat de la flash et HDD 121
4.16 Écriture des résultat de la flash et HDD 121

5.1 Schéma en étoile d'un modèle de coût. 126

Liste des tableaux

2.1 Les formalismes et structures d'optimisation utilisés dans la conception des bases de données traditionnelles 23

2.2 Les formalismes et structures d'optimisation utilisés dans la conception des bases de données traditionnelles 25

2.3 Comparaison entre les bases de données classiques et les entrepôts de données . 28

2.4 Stockage orienté ligne 36

2.5 Comparaison des caractéristiques des SSD et HDD 43

2.6 Evolution de revenus des services *Cloud* 2010-2015 48

2.7 Index bitmap construit sur la base de region pour la table magasin 55

2.8 Index bitmap de jointure construit sur la base de catégorie du Produit sur la table Vente 56

3.1 Les paramètres de l'entrepôt de données. 70

3.2 Les paramètres des requêtes 71

4.1 Paramètres des algorithmes 109

4.2 Représentation d'une matrice *MVPP* 110

4.3 Matrice d'affinité des requêtes 111

4.4 Paramètre d'admission et d'éviction du cache 115

4.5 Taille des tables 118

Glossaire

API: Application Program Interface
ATA: Advanced Technology Attachment
BDD: Base De Données
CODASYL: Conference on Data Systems Languages
DRAM: Dynamic Random Access Memory
EEPROM: Electrically-Erasable Programmable Read-Only Memory
FTL: Flash Translation Layer
ED: Entrepôt de Données
EXT2: Second Extended File System
HDD: Hard Disk Drive
JFFS:Journaling Flash File System
LPN: Logical Page Number
MCG: Modèle de Coût Générique
MLC: Mult-Level Cell
OLAP: Online Analytical Processing
OLTP: Online Transaction Processing
PPN: Physical Page Number
SATA: Serial Advanced Technology Attachment
SCSI: Small Computer System Interface
SD: Storage device
SGBD: Système de gestion de bases de données
SLC: Single Level Cell
SQL: Structured Query Language
SRAM: Static Random Access Memory
SSD: Solide State Drive
RAM: Random Access Memory
RPM: Revolutions Per Minute
YAFFS: Yet Another Flash File System

Résumé.

Les entrepôts de données sont devenus matures pour stocker et exploiter les masses de données issues des capteurs, réseaux sociaux, etc. pour des fins d'analyse. L'accentuation des demandes d'analyse par les entreprises est motivée par la présence de trois éléments principaux: (i) l'évolution technologique qui couvre les plate-formes de déploiement (p. ex. les machines parallèles, les grilles de calcul, Cloud) dotées de supports de stockage avancés (p. ex., les mémoires flash) et des processeurs de calcul puissants (p. ex., les processeurs graphiques (GPU)). (ii) Les nouveaux paradigmes de programmation comme Map-Reduce et Spark et (iii) L'ingénierie Dirigée par les Modèles (IDM) qui a mis à disposition des concepteurs et développeurs des outils, concepts et langages pour créer et transformer des modèles hétérogènes. Avant d'utiliser ces progrès technologiques, l'entrepôt de données doit être construit et préparé à sa bonne exploitation. La construction et l'exploitation d'un entrepôt de données est une tâche complexe. Elle se décompose en deux principales phases : une dite *en amont* comportant la construction de l'entrepôt et une autre dite *en aval* dédiée à son exploitation. La phase de construction a vu l'utilisation massive des efforts de description et de méta modélisation afin de faciliter la définition des correspondances entre les schémas locaux des sources de données et le schéma de l'entrepôt de données et de réduire l'hétérogénéité entre les sources. Des standards de méta modélisation comme *Common Warehouse Models* (CWM) ont vu le jour. Malheureusement, la phase d'exploitation, en général, et sa tâche physique, en particulier n'ont pas eu la même utilisation des solutions de description et de méta modélisation, bien qu'elle est considérée comme un tunnel de toutes les phases de cycle de vie de conception d'un entrepôt de données. Durant cette phase; des modèles de coût mathématiques sont utilisés pour quantifier la qualité des solutions proposées. Le développement de ces derniers nécessite des efforts de collection et d'analyse des paramètres pertinents. Pour bien simuler le fonctionnement d'un entrepôt de données, toutes les dimensions d'un SGBD doivent être intégrées: le schéma de la base, les requêtes, les supports de stockage, les plateformes de déploiement et les structures d'optimisation. Dans cette thèse, nous proposons de décrire en détail ces dimensions avec des mécanismes de méta-modélisation. Vu la similarité et la hiérarchisation qui existent entre les supports de stockage, nous avons développé une ontologie de domaine dédiée aux supports de stockage. Elle permet d'expliciter leurs propriétés. Les similarités entre ces supports nous a motivé à hybrider le cache mémoire avec les mémoires flashs pour augmenter sa capacité afin de stocker un nombre important de résultats intermédiaires de requêtes décisionnelles.Rappelons que l'optimisation de requêtes dépend fortement de la taille de buffer de SGBD cible. Il contient des copies d'une partie des données existant sur une mémoire secondaire. Ces copies sont stockées sous forme de pages et résultent d'un traitement de données par le SGBD. Elles peuvent être constituées des relations, des plans de requêtes ou des données pré-calculées (sous forme de relations, résultats de jointure, etc.). L'intérêt de mettre des données dans le buffer est de pouvoir les réutiliser par d'autres requêtes en réduisant le nombre d'accès en mémoire secondaire où se logent les données de façon permanente. La réutilisation de ces données permet d'augmenter la performance du système grâce aux caractéristiques physiques de la mémoire centrale qui rendent le coût de lecture et d'écri-

ture négligeables par rapport au temps d'accès aux données sur le disque. Nos contributions sont validées à l'aide des expérimentations en utilisant nos modèles de coût théoriques et le SGBD Oracle.

Vers une Description et une Modélisation des Entrées des Modèles de Coût Mathématiques: Applications aux Entrepôts de Données

Présenté par:

Cheikh SALMI

Directeurs de thèse:

Ladjel BELLATRECHE et **Jalil BOUKHOBZA**

Résumé Les entrepôts de données (ED) sont devenus une technologie mature. L'accentuation des demandes d'analyse est motivée par l'évolution technologique, Les nouveaux paradigmes de programmation et L'ingénierie Dirigée par les Modèles (IDM). Avant d'utiliser ces progrès technologiques, l'entrepôt de données doit être construit et préparé pour sa bonne exploitation. La phase de construction a vu l'utilisation massive des efforts de description et de méta modélisation afin de faciliter la définition des correspondances entre les schémas locaux des sources de données et le schéma de l'ED et de réduire l'hétérogénéité entre les sources. La phase d'exploitation et sa tâche physique, en particulier n'ont pas eu la même utilisation des solutions de description et de méta modélisation, bien qu'elle est considérée comme un tunnel de toutes les phases de cycle de vie de conception d'un ED. Durant cette phase; des modèles de coût mathématiques sont utilisés pour quantifier la qualité des solutions proposées. Le développement de ces derniers nécessite des efforts de collection et d'analyse des paramètres pertinents. Pour bien simuler le fonctionnement d'un ED, toutes les dimensions d'un SGBD doivent être intégrées. Dans cette thèse, nous proposons de décrire en détail ces dimensions avec des mécanismes de méta-modélisation. Vu la similarité et la hiérarchisation qui existent entre les supports de stockage, nous avons développé une ontologie de domaine dédiée aux supports de stockage. Elle permet d'expliciter leurs propriétés. Les similarités entre ces supports nous a motivé à hybrider le cache mémoire avec les mémoires flashs pour augmenter sa capacité afin de stocker un nombre important de résultats intermédiaires partagés par plusieurs requêtes décisionnelles. La réutilisation de ces résultats permet d'augmenter la performance du SGBD. Nos contributions sont validées à l'aide des expérimentations en utilisant nos modèles de coût théoriques et le SGBD Oracle.

mots clés Mémoires flash, Entrepôts de données, ontologies, phase physique, méta modélisation.

Abstract Data warehouses (DW) have become a mature technology. The emphasis of the analysis requests is driven by technological change, the new programming paradigms and Model Driven Engineering(MDI). Before using these technological advances, the DW must be built and prepared for its proper operation. The construction phase has seen massive description efforts and meta modeling to facilitate the definition of correspondence between local data sources schemas and DW schema and to reduce heterogeneity between sources. despite its importance in all stages of the design life cycle of an DW, the operational phase and in particular its physical task, did not have the same interest in term of description and meta modeling. During this phase, mathematical cost models are used to quantify the quality of the solutions proposed. The development of these models requires collection efforts and analysis of relevant parameters. To simulate the operation of a DW, all the dimensions of a DBMS must be integrated. In this thesis, we propose to describe in detail these dimensions with meta-modeling mechanisms. Given the similarity and hierarchy between storage media, we have developed an ontology dedicated to storage media, which makes explicit their properties. The similarities between these supports motivated us to develop a hybrid cache based on flash memory. This increases the cache ability to store a large number of intermediate results shared by multiple decision-support queries. The reuse of these results will increase the overall performance of the DBMS. Our contributions are validated with experiments using our theoretical cost models and the Oracle DBMS.

KeyWords: Flash memory, Data warehouse, star schema, decisional query, performance optimization, intermediate query result.