Le déploiement, une phase à part entière dans le cycle de vie des entrepôts de données : application aux plateformes parallèles

Soumia Benkrid

▶ **To cite this version:**

Soumia Benkrid. Le déploiement, une phase à part entière dans le cycle de vie des entrepôts de données : application aux plateformes parallèles. Autre [cs.OH]. ISAE-ENSMA Ecole Nationale Supérieure de Mécanique et d'Aérotechique - Poitiers, 2014. Français. NNT : 2014ESMA0027 . tel-01127551

HAL Id: tel-01127551

https://theses.hal.science/tel-01127551

Submitted on 7 Mar 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

École nationale Supérieure d'Informatique
École Nationale Supérieure de Mécanique et d'Aérotechnique

THESE

pour l'obtention du Grade de
DOCTEUR DE L'ISAE-ENSMA & DE L'ESI

Ecole Doctorale
Secteur de recherche : informatique

Présentée par :

Soumia BENKRID

****

Le déploiement, une phase à part entière dans le cycle de vie des entrepôts de données : application aux plateformes parallèles

****

Directeurs de Thèse : ***Ladjel BELLATRECHE , Khaled-Walid HIDOUCI***

****

Soutenue le 24/06/2014
devant la Commission d'Examen

****

JURY

Président:	**Djamel Eddine ZEGOUR**	Professeur, ESI, Alger
Rapporteurs:	**Arnaud GIACOMETTI**	Professeur, Université François Rabelais de Tours
	Mahmoud BOUFAÏDA	Professeur, Université Mentouri, Constantine
Examinateurs:	**Yamine AÏT AMEUR**	Professeur, IRIT/ ENSEEIHT, Toulouse
	Karima BENATCHEBA	Professeur, ESI, Alger
	Pascal LIENHARDT	Professeur, Université de Poitiers
Directeurs :	**Ladjel BELLATRECHE**	Professeur, ISAE-ENSMA, Poitiers
	Khaled-Walid HIDOUCI	Professeur, ESI, Alger

Remerciements

C'est avec grand plaisir que je réserve cette page, en signe de gratitude et de reconnaissance à tous ceux qui m'ont aidée à la réalisation de ce travail.

Je remercie, tout d'abord, **Ladjel BELLATRECHE** pour sa rigueur et la pertinence de ses jugements qui ont été très constructives et m'ont permis de faire ce travail. Je lui suis également reconnaissante pour sa contribution à l'amélioration judicieuse de la qualité de ce document.

Je remercie **Walid HIDOUCI** pour son co-encadrement et ses encouragements.

Je remercie **Djamel Eddine ZEGOUR** pour m'avoir fait l'honneur d'être président du jury,

Je remercie cordialement **Arnaud GIACOMETTI** et **Mahmoud BOUFAÏDA** d'avoir accepté d'être rapporteurs de cette thèse. Je souhaite adresser également mes remerciements à **Yamine AÏT AMEUR**, **Karima BENATCHEBA** et **Pascal LIENHARDT** d'avoir fait l'insigne honneur d'accepter d'examiner mon travail;

Je remercie également Messieurs **Alfredo CUZZOCREA**, **Ahmad GHAZAL** et **Alain CROLOTTE**; respectivement Chercheur à *ICAR* et Maître de conférence à l'*Université de Calabria* (Italie), Architecte chez *Teradata* (Greater Los Angeles, Etats-Unis) et spécialiste de performance chez *Teradata* (Greater Los Angeles, Etats-Unis) avec qui j'ai collaboré pour valider une contribution non négligeable de mon travail.

J'adresse mes remercîments les plus vifs à **Mouloud KOUDIL**, Directeur de l'ESI, pour ses conseils et son soutien.

Je n'oublie pas non plus de remercier **Si-Larbi KHELIFATI** directeur de la post graduation et ses assistants **Djamel** et **Safia** pour leur professionnalisme et leur assiduité à me diriger.;

Un grand merci à **Ghislaine VOGUET**, **Zoé FAGET**, **Stéphane Jean**, **Allel HADJ ALI** et **Brice Chardin**, pour avoir consacré du temps à la lecture de ce manuscrit.

Ce travail de thèse a été réalisé dans le cadre d'une co-tutelle entre l'ESI et LIAS. Je remercie leur directeur **Emmanuel GROLLEAU** ainsi que tous les chercheurs et l'ensemble du personnel du laboratoire pour m'avoir fait profiter de leur expérience et de leur savoir.

Je voudrais exprimer à mes proches toute ma gratitude: ma chère grande mère, mes très chers parents, mes frères et mes sœurs, mes belles sœurs et mon beau frère. Sans leur soutien, leur confiance et leurs encouragements, je n'y serais jamais arrivée.

Un grand merci à tous mes amis pour leur dévouement et leur amitié sans faille qui m'ont tant soutenu moralement;

Ces remerciements ne seraient pas complets sans une pensée pour la personne qui a redonné un sens à l'amitié.

Merci, enfin, à toutes celles et ceux qui, de près ou de loin, ont contribué à l'aboutissement de ce projet.

A mes très chers parents

Table des matières

Table des figures **xx**

Liste des tableaux **xxi**

Liste des algorithmes **xxiii**

Glossaire **xxvi**

Introduction générale **1**

Partie I Etat de l'art **9**

Chapitre 1
Les entrepôts de Données Relationnels : États de l'Art **11**

1.1 Cycle de vie de conception d'un entrepôt de données 13

1.1.1 Analyse des besoins 14

1.1.2 Conception logique 15

1.1.2.1 Les modèles multidimensionnels 16

1.1.2.2 Les systèmes MOLAP 16

1.1.2.3 Les systèmes ROLAP 17
1.1.3 Phase ETL (Extract-Transform-Load) 18
1.1.4 Conception physique 19
1.1.5 Bilan : vers l'intégration de la phase de déploiement dans un cycle de vie 20

Chapitre 2
Cycle de déploiement des entrepôts de données parallèles 23

2.1 Les étapes de la phase de déploiement d'un EDP 24
2.1.1 Choix de l'architecture matérielle 24
2.1.1.1 Architectures conventionnelles 25
2.1.1.1.1 Architectures à mémoire partagée (shared-memory). 25
2.1.1.1.2 Architectures à disques partagés (shared disks). . 26
2.1.1.1.3 Architectures sans partage (shared-nothing) . . . 26
2.1.1.1.4 Architectures hybrides. 26
2.1.1.2 Architectures distribuées 27
2.1.1.2.1 Les architectures grappes de machines (cluster). . 27
2.1.1.2.2 Grille de calcul. 28
2.1.1.2.3 Cloud de calcul. 29
2.1.2 Fragmentation de données. 29
2.1.2.1 Les types de fragmentation. 30
2.1.2.1.1 La fragmentation verticale. 30
2.1.2.1.2 La fragmentation horizontale. 31
2.1.2.1.3 La fragmentation mixte. 32
2.1.2.2 Problème de la fragmentation horizontale 32
2.1.2.2.1 Travaux existants 33
2.1.2.2.1.1 Travaux existants dans le contexte centralisé 33
2.1.2.2.1.2 Travaux existants dans le contexte distribué 35
2.1.2.2.1.3 Travaux existants dans le contexte parallèle 35
2.1.2.3 Problème de la fragmentation verticale 43
2.1.2.3.1 Travaux existants 43

2.1.2.3.1.1 Travaux existants dans le contexte centralisé 43
2.1.2.3.1.2 Travaux existants dans le contexte distribué 44
2.1.2.3.1.3 Travaux existants dans le contexte parallèle 44
2.1.2.4 Problème de la fragmentation mixte 44
2.1.2.4.1 Travaux existants 45
2.1.2.4.1.1 Travaux de Cheng *et al* 45
2.1.2.5 Bilan et discussion 45
2.1.3 Allocation de données 46
2.1.3.1 Travaux existants 46
2.1.3.1.1 Travaux existants dans le contexte parallèle . . . 46
2.1.3.1.1.1 Placement circulaire (round-robin) 47
2.1.3.1.1.2 Placement par hachage (hash placement) . 47
2.1.3.1.1.3 Placement par intervalle (range Partitionning) 47
2.1.3.1.1.4 Travaux de Furtado et *al* 48
2.1.3.1.2 Travaux existants dans le contexte distribué . . . 49
2.1.3.1.2.1 Travaux de Huang et Chen 49
2.1.3.1.2.2 Corcoran *et al* 50
2.1.3.1.2.3 Travaux de Ahmad *et al* 50
2.1.3.1.2.4 Travaux de Rosa Karimi Adl *et al* 50
2.1.3.1.2.5 Travaux de Sarathy *et al* 51
2.1.3.1.2.6 Travaux de Hababeh *et al* 51
2.1.3.1.2.7 Travaux de Maik Thiele et al 51
2.1.3.2 Bilan et discussion 51
2.1.4 Réplication des fragments 52
2.1.4.1 Les modèles de réplication 53
2.1.4.2 Travaux existants. 55
2.1.4.2.1 Travaux existants dans le contexte parallèle . . . 55
2.1.4.2.1.1 Travaux de Borr: *Mirrored Declustering* . 55
2.1.4.2.1.2 Travaux de Hsioa et al : *Chained Declustering* 56

2.1.4.2.1.3 Travaux de Teradata: *Interleaved declustering* 56
2.1.4.2.2 Travaux existants dans le contexte distribué . . . 57
2.1.4.2.2.1 Travaux de Costa et *al* 57
2.1.4.2.2.2 Travaux de Zhu et *al* 58
2.1.4.2.2.3 Travaux de Chang *et al* 58
2.1.4.2.2.4 Travaux de Foresiero *et al* 58
2.1.4.3 Bilan et discussion 58
2.1.5 Equilibrage de charges 59
2.1.5.1 Concepts de base. 61
2.1.5.1.1 Formes du parallélisme. 61
2.1.5.1.2 Paradigme d'équilibrage de charge 62
2.1.5.1.3 Obstacles de l'équilibrage de charge 64
2.1.5.1.4 Algorithmes d'équilibrage de charge 66
2.1.5.2 Travaux existants 67
2.1.5.2.1 Travaux existants dans le contexte parallèle . . . 67
2.1.5.2.1.1 Travaux de Röhm et *al* 68
2.1.5.2.1.2 Travaux d'Akal et *al* 68
2.1.5.2.1.3 Travaux de Märtens et *al* 69
2.1.5.2.1.4 Travaux de Lima et *al* 69
2.1.5.2.1.5 Travaux de Phan et *al* 70
2.1.5.2.1.6 Travaux de Lima et *al* 70
2.1.5.2.2 Travaux existants dans le contexte distribué . . . 70
2.1.5.2.2.1 Travaux de Gorla *et al* 71
2.1.5.2.2.2 Travaux de Gounaris *et al* 71
2.1.5.3 Bilan et discussion 71
2.2 Les implémentations principales des EDPs 72
2.2.1 Teradata 73
2.2.2 IBM Netezza 73
2.2.3 Greenplum 74
2.2.4 Oracle Exadata 75

2.2.5 Microsoft SQL Server 76
2.3 Mesures de performances 77
2.4 Bilan et discussion 79

Partie II Nos propositions 83

Chapitre 3
Notre Modèle de Coût 85

3.1 Les paramètres liés à notre modèle de coût 86
3.1.1 Paramètres de l'entrepôt de données 86
3.1.1.1 Paramètres des requêtes 87
3.1.1.1.1 Estimation de la sélectivité des prédicats 88
3.1.1.1.2 Estimation de la sélectivité des prédicats complexes 88
3.1.2 Paramètres liés à l'architecture de déploiement 89
3.1.2.1 Paramètres de fragmentation 89
3.1.2.2 Paramètres d'allocation 90
3.1.3 Politique de notre traitement parallèle 92
3.1.3.1 Réécriture des requêtes 93
3.1.3.2 Identification des fragments faits 94
3.1.3.3 Identification du plan d'exécution optimal 94
3.1.3.3.1 Identification des jointures nécessaires 94
3.1.3.3.1.1 Scénario 1 : matching total. 94
3.1.3.3.1.2 Scénario 2 : matching partiel 95
3.1.3.3.2 Ordre d'exécution des jointures 97
3.1.3.3.3 Méthodes d'accès 97
3.1.3.4 Ordonnancement des sous-requêtes générées. 97

3.1.3.5 L'identification des nœuds valides 97
3.1.3.6 L'allocation des sous-requêtes 97
3.1.3.6.1 Formalisation 98
3.1.3.6.2 Algorithmes proposés 99
3.1.3.7 La collecte et la fusion des résultats 102
3.2 La définition de notre modèle de coût 103
3.2.1 Coût de traitement 103
3.2.2 Le coût de communication 104

Chapitre 4
Notre approche pas à pas de déploiement 107

4.1 Différentes architectures de déploiement d'un EDP 107
4.2 $\mathcal{F}\&\mathcal{A}$: une démarche conjointe de fragmentation et d'allocation 110
4.2.1 Formalisation du problème 111
4.2.2 Algorithme de conception 113
4.2.2.1 Phase de fragmentation de $\mathcal{F}\&\mathcal{A}$ 113
4.2.2.1.1 modèle d'un schéma de fragmentation 114
4.2.2.1.2 Algorithme Hill Climbing 115
4.2.2.1.2.1 Solution initiale 116
4.2.2.1.2.2 Opérateurs de mouvements 116
4.2.2.1.3 Algorithme génétique 119
4.2.2.1.3.1 Opérateur de sélection 120
4.2.2.1.3.2 Population initiale 121
4.2.2.1.3.3 Fonction d'évaluation 121
4.2.2.1.3.4 Opérateur de croisement 121
4.2.2.1.4 Opérateur de mutation 122
4.2.2.2 Phase d'allocation de $\mathcal{F}\&\mathcal{A}$ 123
4.2.2.2.1 Construction de la Matrice d'Usage des Fragments 123
4.2.2.2.2 Construction de la Matrice d'Affinités des Fragments 124
4.2.2.2.3 Le regroupement des fragments: $\mathcal{F}\&\mathcal{A}$-ALLOC . 124

4.2.2.2.4 Construction de la Matrice de Placement des Fragments. 125
4.2.2.2.5 Modèle de coût 128
4.3 $\mathcal{F}\&\mathcal{A}\&\mathcal{R}$: une approche globale pour la conception d'un EDP 130
4.3.1 Formulation du problème 131
4.3.1.1 Backgrounds 131
4.3.1.2 Formulation 132
4.3.2 Approche proposée : $\mathcal{F}\&\mathcal{A}\&\mathcal{R}$ 133
4.3.2.1 Procédure de fragmentation 133
4.3.2.2 Phase d'allocation des fragments 134
4.3.2.2.1 Formalisation 135
4.3.2.2.2 L'Algorithme Fuzzy declustering) 135
4.3.2.2.2.1 Description des données 136
4.3.2.2.2.2 Représentation de chaque fragment dans R^2 136
4.3.2.2.2.3 Regroupement des attributs 137
4.3.2.2.2.4 Conception du discriminateur 138
4.3.2.2.2.5 Construction de la Matrice de Placement des Fragments 138
4.3.2.2.3 Complexité de l'algorithme Fuzzy declustering . . 139
4.3.2.3 Modèle de coût de $\mathcal{F}\&\mathcal{A}\&\mathcal{R}$ 140

Chapitre 5
Évaluation théorique et réelle sur Teradata **143**

5.1 Le simulateur 144
5.2 Évaluation de performance de $\mathcal{F}\&\mathcal{A}$ 144
5.2.1 Paramètres d'expérimentation 144
5.2.2 Résultats expérimentaux obtenus 147
5.2.3 Validation sous Teradata 152
5.2.3.1 Données et charge de requêtes. 152
5.2.3.2 Architecture matérielle 154
5.2.3.3 Etapes de validation 155

5.2.3.3.1 Génération des schémas de fragmentation et d'allocation 155
5.2.3.3.2 Mise en œuvre des schémas obtenus sur Teradata 156
5.2.3.3.3 Résultats obtenus 156
5.2.3.4 Analyse des résultats obtenus 156
5.3 Évaluation de performance de $\mathcal{F}\&\mathcal{A}\&\mathcal{R}$ 161
5.3.1 Paramètres d'expérimentation 161
5.3.2 Résultats obtenus 162
5.4 Bilan et discussion 166

Partie III Conclusion et perspectives 169

Conclusion et perspectives 171

Partie IV Annexes 175

Annexe 1: Liste des requêtes SSB 177

Bibliographie **183**

Table des figures

1 *Les étapes de l'approche de conception itérative* 4
2 *Notre vision de conception* 4
3 *Principe de base de l'approche conjointe $\mathcal{F}\&\mathcal{A}$* 5
4 *Principe de base de l'approche conjointe $\mathcal{F}\&\mathcal{A}\&\mathcal{R}$* 5

1 *Architecture conceptuelle d'un entrepôt de données* 12
1.2 *Cycle de vie des entrepôts de données* 13
1.3 *Vue multidimensionnelle des données* 16
1.4 *Exemple d'un schéma en étoile.* 17
1.5 *Exemple d'un schéma en flocon en neige.* 18
1.6 *Structure générique du problème de la conception physique* 21
1.7 *La place de la phase de déploiement dans le cycle de vie* 21

1 *Cycle de vie de conception des des entrepôts de données parallèles* 24
2.2 *Architetcures matérielles usuelles* 25
2.3 *Exemple d'un cluster d'ordinateurs* 28
2.4 *Exemple de l'infrastructure grille de calcul* 28
2.5 *Exemple de l'infrastructure cloud de calcul* 29
2.6 *Evolution de la fragmentation de données* 30
2.7 *Exemple d'une fragmentation verticale* 31
2.8 *Exemple de la fragmentation horizontale* 32
2.9 *Exemple de la fragmentation horizontale* 33
2.10 *Approche de fragmentation basée sur le data mining* [50] 34
2.11 *Independent Relation* 36
2.12 *Principe de l'approche MDFH.* 38

2.13 *Architecture de l'Advisor de partitionnement proposé par Rao.* 40
2.14 *Architecture de l'Advisor de partitionnement proposé par Nehme.* 41
2.15 *Exemple de placement circulaire.* 47
2.16 *Exemple de placement par hachage.* 48
2.17 *Exemple de placement par intervalle.* 48
2.18 *Classification du placement de données* 52
2.19 *Mirrored Declustering* 56
2.20 *Chained Declustering* 57
2.21 *Interleaved declustering* 57
2.22 *Classification des travaux de réplication de données* 59
2.23 *Equilibrage de charge vs déséquilibrage de charge.* 60
2.24 *Parallélisme intra-opérateur de l'opérateur de sélection.* 62
2.25 *Parallélisme intra-opérateur de l'opérateur de jointure.* 62
2.26 *Parallélisme inter-opérateur.* 63
2.27 *Paradigme d'équilibrage de charge.* 63
2.28 *Interférences des tâches parallèles* 64
2.29 *Exemple du mauvais placement des tuples* 65
2.30 *Exemple de la mauvaise distribution de sélectivité* 65
2.31 *Exemple de la mauvaise des résultats intermédiaires* 66
2.32 *Classification des travaux d'équilibrage de charge* 72
2.33 *Teradata* 74
2.34 *IBM Netezza* 74
2.35 *Greenplum* 75
2.36 *Oracle Exadata* 76
2.37 *Microsoft SQL Server* 77
2.38 *Exemple des Speed-Up* 78
2.39 *Exemple des Scale-Up* 79
2.40 *Les étapes de l'approche de conception itérative* 80

3.1 *Exemple d'une grappe de machines* 89
3.2 *Distribution des données uniforme vs. distribution des données non uniforme* 91
3.3 *Paradigme de traitement parallèle* 92
3.4 *Architecture modulaire du nœud coordinateur* 93
3.5 *Plan d'exécution parallèle de la requêtes Q_2* 96

3.6 *Plan d'exécution parallèle de la requêtes Q_1* 96
3.7 *Ordonnanceur des requêtes* 98

4.1 *Methodologies de conception d'un EDP* 109
4.2 *Principe de l'approche $\mathcal{F}\&\mathcal{A}$* 110
4.3 *Schéma de fragmentation candidat $\mathcal{A}_{Product}$* 115
4.4 *Approche $\mathcal{F}\&\mathcal{A}$ basée sur l'algorithme Hill Climbing* 116
4.5 *Application des opérateurs Merge et Split* 117
4.6 *Approche $\mathcal{F}\&\mathcal{A}$ basée sur l'algorithme génétique* 119
4.7 *Exemple de l'opérateur de croissement* 122
4.8 *Exemple de l'opérateur de mutation* 122
4.9 *Les groupes de fragments générés* 125
4.10 *Organigramme de notre approche $\mathcal{F}\&\mathcal{A}\&\mathcal{R}$.* 133
4.11 *Allocation vs Classification* 135
4.12 *Représentation des fragments* 137
4.13 *Regroupement des Fragments associé à la matrice FAM (Table 4.4)* 138

1 *Cycle de validation de notre approche* 143
5.2 *Cas d'utilisations de notre simulateur* 145
5.3 *Schéma logique du banc d'essai APB-1 release II* 146
5.4 *Approche conjointe vs approche séquentielle* 147
5.5 *Speed-Up de $\mathcal{F}\&\mathcal{A}$ vs Speed-Up de l'approche itérative* 148
5.6 *Scale-Up de $\mathcal{F}\&\mathcal{A}$ vs Scale-Up de l'approche itérative* 148
5.7 *Effet de l'hétérogénéité de la grappe sur la performance du système* 149
5.8 *Effet de la puissance de calcul sur la performance du système* 150
5.9 *Effet de la capacité du stockage* 151
5.10 *Effet du Seuil de Fragmentation W sur la Performance de l'approche $\mathcal{F}\&\mathcal{A}$* . . 151
5.11 *Pourcentage de réduction du coût de traitement* 152
5.12 *Schéma logique du banc d'essai SSB* 153
5.13 *Architecture de Teradata* 154
5.14 *Attributs de fragmentation pour le deploiement sous Teradata* 156
5.15 *Répartition des Charges de la requête Q1.2* 158
5.16 *Répartition des Charges de la requête Q4.3* 159
5.17 *Répartition des Charges de la requête Q11* 159
5.18 *Répartition des Charges de la requête Q8* 160

5.19 *Répartition des Charges de la requête Q5* 161
5.20 *Comparaison entre les approches de conception d'un EDP* 162
5.21 *Effet du degré de réplication sur le speed-up de $\mathcal{F}\&\mathcal{A}\&\mathcal{R}$* 163
5.22 *Skew des valeurs d'un attribut vs Skew de Partitioning de Données* 164
5.23 *Dépendance entre degré de réplication et skew de traitement.* 164
5.24 *Effet du degré de skew des valeurs d'un attribut sur le seuil de fragmentation* . 165
5.25 *Effet du degré de skew des valeurs d'un attribut sur le temps d'exécution* 165
5.26 *Effet d'hétérogénéité sur la performance.* 166

Liste des tableaux

2.1 *Synthèse de comparaison entre les travaux de fragmentation* 46
2.2 *Synthèse de comparaison entre les travaux d'allocation.* 52
2.3 *Synthèse de comparaison entre les travaux de réplication* 59
2.4 *Synthèse de comparaison entre les travaux d'équilibrage de charge* 72

3.1 Les paramètres de l'entrepôt de données. 87
3.2 Les paramètres des requêtes 87
3.3 Les paramètres physiques 90
3.4 Les paramètres de la fragmentation 90
3.5 Les paramètres de placement 92

4.1 *Matrice d'Usage des Fragments* 123
4.2 *Matrice d'Affinité des Fragments* 124
4.3 *Matrice d'Usage des Fragments* 136
4.4 *Matrice d'Appartenance des Fragments* 138
4.5 *Matrice de Placement des Fragments* 139

5.1 *Cardinale des tables de* SSB 153
5.2 *Schéma de fragmentation candidat* 154
5.3 *Schéma de fragmentation de l'approche conjointe* 155
5.4 *Schéma de fragmentation de l'approche itérative* 156
5.5 *Temps d'exécution en Seconde* 157

Liste des algorithmes

1 *Allocation des Requêtes* ($\mathcal{SQ} - \mathcal{DPB}$) 101
2 *Algorithme de répartition de charges*(*Algo_Migration_Dynam*) 102
3 Algorithm F&A-HC 118
4 Algorithm GA 120
5 Algorithm $\mathcal{F}$&$\mathcal{A}$-ALLOC 127
6 *Fonction de "Low-Skew"* (Cr_i Chromosome) 134

Glossaire

DWA : Administrateur de l'Entrepôt de Données.
A_k : attribut de Fragmentation.
AG : algorithme génétique.
HC : Hill Climbing.
MOLAP : Multidimensional On-Line Analytical Processing.
OLAP : On-Line Analytical Processing.
RJE : Requêtes de Jointure en Etoile.
ROLAP : Rolational On-Line Analytical Processing.
SF : schéma de Fragmentation.
W : nombre de fragments que l'administrateur souhaite.
R : seuil de réplication.
F&A : sélection conjointe de la fragmentation et d'allocation.
F&A&R: sélection conjointe de la fragmentation et d'allocation et la réplication .
DBP : Dual Bin Packing.
CBD : grappe de Base de Données.
DP : partitionnement de donnes.
DA : allocation de données.
DR : réplication de données.
LB : équilibrage de charge.
QP : traitement parallèle.
ED : entrepôt de données.
EDP : entrepôt de données parallèle.
SN : Shared Nothing.
MC : Modèle de coût.
M_P : Matrice de Placement de Fragments.
M_U : Matrice d'Usage de Fragments.
M_A : Matrice d'Affinités des fragments.
MPSQ : Matrice d'Allocation des sous-requêtes.
Q : Charge de requêtes OLAP.
SSB : Star Schema Benchmark.

Introduction générale

"There is nothing more difficult to take in hand,
more perilous to conduct, or more uncertain in its success,
than to take the lead in the introduction of a new order of things".
-Niccolo Machiavelli(1469 -1527)

Contexte

Les entrepôts de données permettent, au travers de l'analyse de l'activité de l'entreprise, de produire des connaissances rigoureuses et pertinentes qui sont ensuite exploitées par les décideurs ou les scientifiques en vue d'améliorer les performances ou valider leurs théories. Un entrepôt de données est un dépôt de multiples sources de données hétérogènes, organisées sous un schéma unifié pour faciliter la gestion de la prise de décision [37]. Sa construction inclut l'identification des sources de données participant à l'entreposage, la transformation et le nettoyage des données qui peuvent être hétérogènes, l'intégration de données dans l'entrepôt (sous forme historisées), et sa connexion avec un serveur OLAP (Online Analytical Processing) pour analyser les données et les visualiser selon différents angles [81]. Les entrepôts de données sont aujourd'hui arrivés à maturité, ce qui se traduit par un nombre important d'entreprises et d'organismes qui les exploitent.

Les applications conçues autour de la technologie des entrepôts de données, à la différence de celles conçues autour des bases de données, stockent un historique des données afin de générer des connaissances toujours plus pertinentes et de meilleure qualité. Ceci en fait des applications consommatrices de masses importantes de données.

Dans la dernière décennie, cette masse de données est devenue extrêmement large. Des conférences spécialisées sur la gestion des cette masse de données ont été créés, comme la conférence *Extremely Large Databases Conference* lancée par l'*Université de Stanford, USA* (`https://conf-slac.stanford.edu/xldb-2013/`). Cette explosion des données est due au développement des applications autour des réseaux sociaux (Facebook, Linkedin, Twitter), la généralisation de l'utilisation des capteurs dans différents domaines (agriculture, nucléaire, médecine, etc.), les données scientifiques issues des applications astronomiques, environnements, agriculture, etc. En conséquence, la gestion et l'exploitation efficaces de cette masse sont devenues un *enjeu important*.

Pour répondre à ce besoin, un grand nombre de travaux proposent la sélection de structures d'optimisation (comme les vues matérialisées, les index, la compression, etc.) dans la phase physique de conception d'un entrepôt de données. Souvent ces structures sont sélectionnées sur des plateformes centralisées. Ces environnements classiques de déploiement des entrepôts de données ont montré leur limite pour répondre aux requêtes décisionnelles complexes connues sous le nom "Big Data Analytical Workload". Pour remédier à ces limites, des solutions parallèles ont été proposées. La principale motivation de l'utilisation d'une telle technologie de traitement parallèle dans les entrepôts de données volumineux ne dépend pas seulement de la nécessité de la haute performance, de l'évolution, de la fiabilité et de la disponibilité mais aussi du fait que les ordinateurs parallèles ne sont plus un monopole des *architectures de super-ordinateurs* [143]. Ils sont actuellement disponibles sous plusieurs formats tels que les Multi-Processeurs Symétriques (SMP), les Clusters, les Machines Massivement Parallèles (MMP), les architectures sans partage (Shared-Nothing) et les architectures à disques partagés (Shared Disks). L'architecture shared-nothing a été recommandée par DeWitt *et al.* [57] comme une architecture de référence pour la mise en œuvre des entrepôts de données à haute performance modélisés par un schéma en étoile.

Vue la diversité des plateformes, la proposition d'un processus de déploiement d'un entrepôt de données devient un enjeu important pour les entreprises. En analysant les travaux existants, nous avons identifié la présence d'un cycle de vie de déploiement, en dépit du fait *qu'il est le plus souvent ignoré par la communauté des entrepôts de données.*

Cette affirmation a été identifiée lors de notre collaboration avec l'entreprise *Teradata*[1], le leader mondial pour la gestion des entrepôts de données sur machine parallèle de type *shared nothing*. Cette phase de déploiement quelque soit la nature de la plateforme: centralisée ou parallèle est composée principalement de trois étapes principales à savoir: *(1)* la fragmentation (ou partitionnement[2]) de l'entrepôt de données, *(2)* l'allocation des fragments générés sur la plateforme de déploiement (tablespaces dans le cas centralisé et les nœuds dans le cas parallèle) et *(3)* la définition d'une stratégie de traitement des requêtes. Le problème de fragmentation des données consiste à diviser l'entrepôt de données en unités disjointes appelées *fragments* (ou *partitions*). Le partitionnement peut se faire horizontalement ou verticalement. Le partitionnement horizontal est essentiellement utilisé pour la conception des entrepôts de données parallèles. L'allocation des données consiste à placer les fragments générés par le processus de partitionnement sur la plateforme de déploiement.

L'allocation peut être soit redondante (avec réplication) ou non redondante (sans réplication) dans le cas de plateformes parallèles. Une fois les fragments placés, les requêtes sont exécutées sur les nœuds de la plateforme (dans le cas de déploiement parallèle). Le traitement parallèle des requêtes englobe : (1) la réécriture des requêtes globale selon le schéma de fragmentation, et (2) l'allocation des sous-requêtes générées sur les nœuds de la plateforme selon le schéma d'allocation telle que les nœuds soient uniformément chargés. L'équilibrage de charge entre les nœuds du cluster est un enjeu important pour atteindre la haute performance de l'entrepôt. En effet, un déséquilibre de charge peut être causé par l'un (ou la combinaison) des deux problèmes suivants : (i) la mauvaise répartition des données *(data skew)*, situation

1. Teradata a été adoptée par des grandes entreprises françaises comme *Carrefour* et *Banque Populaire*.
2. Dans ce manuscrit, nous utilisons les termes partitionnement et fragmentation de manière interchangeable.

où les données sont distribuées d'une manière non uniforme sur les différents nœuds de traitement. Ceci se produit généralement quand la fonction de partitionnement de données utilise un attribut dont la distribution des valeurs de données est biaisée (*Attribut Value Skew*); (ii) la mauvaise répartition de traitement (*Processing Skew*), situation où une grande partie de la charge de requêtes est exécutée sur peu de nœuds de traitement quand les autres nœuds sont relativement inactifs. Ceci est souvent dû à la répartition biaisée des données. Dans la littérature, l'équilibrage de charge est effectué via une redistribution des données des nœuds surchargés vers les nœuds sous chargés. Cette migration des données peut engendrer un coût de communication élevé et le nœud coordinateur peut devenir un goulot d'étranglement. En conséquent, la réplication des données est devenue une exigence pour éviter ce goulot et réduire le coût de communication. Effectivement, la réplication de données assure : (a) la disponibilité des données et la tolérance de panne en cas de défaillance, (b) la localité de traitement et (c) l'équilibrage de charge.

La plupart des travaux existants s'intéressent à une ou plusieurs phases de ce cycle et non pas à sa totalité. Dans cette thèse, nous nous intéressons au cycle de vie de déploiement d'un entrepôt de données *sur une plateforme parallèle*, où une approche globale de conception d'entrepôt de données parallèle est proposée. Pour valider nos propositions, nous considérons deux cas d'étude : (i) une machine parallèle de type *shared nothing*, une solution adaptée pour *des grandes entreprises* vu le coût d'acquisition et maintenance de cette machine. (ii) Un *cluster de bases de données* pour les petites et moyennes entreprises (PME) qui peuvent profiter de l'évolution technologique pour créer des clusters d'ordinateurs à *moindre coût*.

Problématique et contributions

En étudiant la littérature concernant la conception des bases de données parallèles en général et les entrepôts de données parallèles en particulier, nous avons remarqué que les étapes de la phase de déploiement sont traitées d'une manière indépendante ou isolée; ceci malgré le fait que chaque phase (à l'exception de la phase de choix de machine) prend en entrée les sorties de la phase précédente. Cette indépendance a fait naître quatre communautés de recherche principales: la première travaille sur la sélection des schémas de fragmentation, la deuxième traite le problème de placement des fragments, la troisième propose des solutions de réplication de fragments et la quatrième se concentre sur le problème d'équilibrage de charge. La conséquence de cette conception est que chaque étape ignore les processus d'autres phases et utilise son propre modèle de coût pour quantifier la qualité de sa solution. En conséquence, nous aurons une vision isolée.

Dans cette thèse nous proposons une vision de *composition* dans quelle nous identifions l'ensemble de paramètres pertinents de chaque phase ensuite nous identifions l'interaction entre l'ensemble des phases et finalement un modèle de coût global est proposé pour quantifier l'ensemble des solutions. En d'autres mots et pour illustrer notre démarche, supposons que chaque étape est associée à un acteur (ou outil): *Partitionnor* pour la fragmentation, *Allocator* pour l'allocation, *Replicator* pour la réplication, *Balancer* pour l'équilibrage de charge et *Query-Processer* pour le traitement de requêtes. Actuellement chaque acteur a sa propre métrique

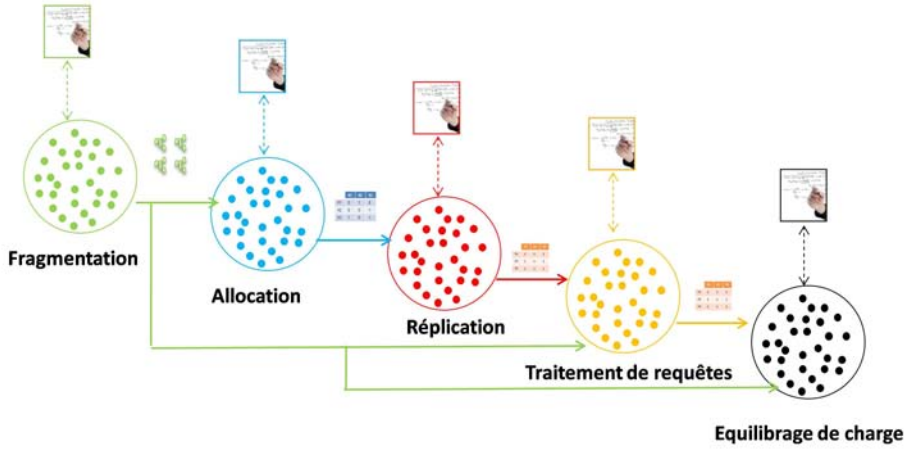


Figure 1 – *Les étapes de l'approche de conception itérative*

pour évaluer la qualité de sa solution. Cette dernière est souvent développée indépendamment des autres. Notre proposition augmente la collaboration dans le développement de métrique générale qui pourrait être utilisée par l'ensemble des phase de déploiement, ce qui favorise l'*omniscience des acteurs*.

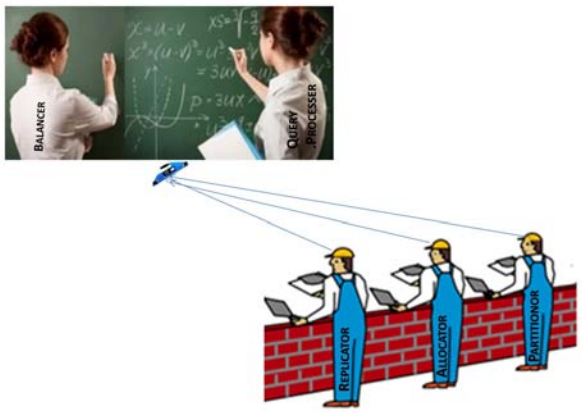


Figure 2 – *Notre vision de conception*

Unifier l'ensemble des phases est une tâche difficile vu la complexité de chaque étape (les problèmes liés à chaque étape sont connus comme NP-complet [1, 5, 147, 120, 64]). Vu cette difficulté, dans cette thèse nous proposons une approche incrémentale d'unification avec la présence d'un seul modèle de coût pour l'ensemble des étapes. Dans un premier temps, nous avons commencé par les deux premières étapes à savoir la fragmentation et l'allocation, vue leur forte interdépendance. Rappelons que certains travaux dans le contexte des entrepôts de données parallèles ont considéré ces deux problèmes d'une manière conjointe [134]. La première fusion est baptisée $\mathcal{F}\&\mathcal{A}$ pour signifier *Fragmentation and Allocation* (Figure 3).

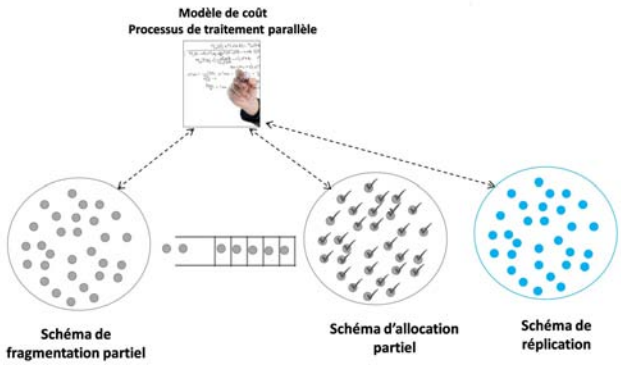


Figure 3 – *Principe de base de l'approche conjointe* $\mathcal{F}\&\mathcal{A}$

Dans un second temps, nous étendons notre approche $\mathcal{F}\&\mathcal{A}$ par l'ajout de la phase de réplication de données, ce qui donne $\mathcal{F}\&\mathcal{A}\&\mathcal{R}$, comme le montre la figure 4. Notons que la réplication est fortement dépendante de la fragmentation et de l'allocation. L'augmentation de notre vision initiale par la réplication nous permet de quantifier chaque solution potentielle de fragmentation sur l'allocation et la réplication. La solution ayant un coût minimal est retenue.

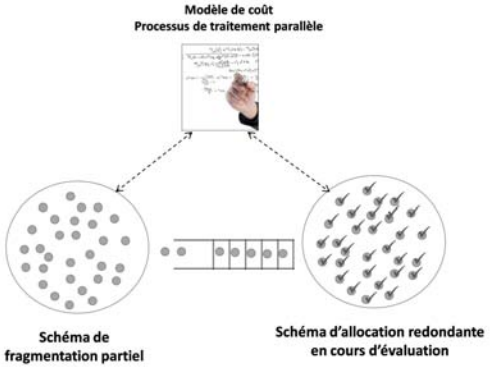


Figure 4 – *Principe de base de l'approche conjointe* $\mathcal{F}\&\mathcal{A}\&\mathcal{R}$

Pour valider nos propositions, nous avons considéré deux plateformes: une machine parallèle de type *shared nothing* recommandée par les communauté des entrepôts de données, et un cluster de bases de données hétérogènes. Nous voulons mettre l'accent sur l'hétérogénéité, car la majorité des travaux sur des clusters considèrent que l'ensemble des nœuds sont homogènes, c'est à dire que tous les nœuds possèdent les mêmes caractéristiques en termes de puissance de calcul, capacité de stockage, etc. Cette hypothèse n'est pas toujours valide dans la réalité où il est probable d'avoir un cluster de machines différentes.

Organisation de la thèse

Ce manuscrit est organisé en deux parties.

La première partie a pour objectif la présentation des concepts, méthodes et outils nécessaires pour répondre à notre problématique. Elle comporte deux chapitres. Le premier décrit l'ensemble des phases traditionnelles de cycle de vie de conception d'un entrepôt de données. Le second décrit les phases du cycle de vie de déploiement d'un entrepôt de données parallèle. Pour chaque phase, nous analysons les principaux travaux existant dans la littérature, puis nous présentons les grands éditeurs de bases de données parallèles ainsi que les mesures de performance des systèmes parallèles.

La seconde partie présente nos travaux et propositions en trois chapitres.

Nous proposons un nouveau modèle de coût qui unifie le processus de traitement parallèle des requêtes OLAP sur une grappe de bases de données, avec une nouvelle stratégie de placement de requêtes qui assure un bon degré d'équilibrage de charge.

Le chapitre 4 présente une nouvelle approche de déploiement d'un entrepôt de données parallèle. Dans un premier temps, nous avons proposé l'approche $\mathcal{F}\&\mathcal{A}$ où la fragmentation et l'allocation se font simultanément. Notre approche de conception est formalisée comme un problème d'optimisation à contraintes. Des algorithmes de fragmentation (hill climbing et génétique) et d'allocation (allocation à base d'affinités) sont utilisés pour le résoudre. Ensuite, nous avons étendu $\mathcal{F}\&\mathcal{A}$ en combinant la réplication avec la fragmentation et l'allocation. Nous proposons un nouvel algorithme d'allocation redondant de données basé sur la classification floue.

Le chapitre 5 présente tout d'abord les résultats des nombreuses expérimentation menées pour étudier et analyser la performance de nos approches. Ensuite nous présenterons la validation de notre approche sur le SGBD parallèle *Teradata* en utilisant les données du banc d'essais $\mathcal{SSB}$.

Le chapitre 6 conclut ce document et présente les perspectives et les axes de recherche qui pourront être poursuivis.

Publications

La liste suivante représente les articles publiés dans le cadre de cette thèse.

Ouvrages individuels ou collectifs

1. Ladjel BELLATRECHE, Kamel BOUKHALFA, Pascal RICHARD, **Soumia BENKRID**, Data Partitioning dor Designing and Simulating Efficient Huge Databases, Chapter: Scalable Computing and Communications: Theory and Practice, Publisher: Wiley, Editors: Samee U. Khan, Lizhe Wang, and Albert Y. Zomaya, pp.523-562 [94].

Revues nationales et internationales

1. Ladjel Bellatreche, Cuzzocrea Alfredo, **Soumia Benkrid**, Effectively and Efficiently Designing and Querying Parallel Relational Data Warehouses on Heterogeneous Database Clusters: The $\mathcal{F}\&\mathcal{A}$ Approach, in Journal of Database Management (JDM), 23(4): 17-51, 2012, (ScienceCitation Index Expanded (SciSearch®)) [17].
2. **Soumia Benkrid**, Ladjel Bellatreche, Une démarche conjointe de fragmentation et de placement dans le cadre des entrepôts de données parallèles, Technique et Science Informatiques (TSI), 30(8): 953-973, 2011 [22].

Conférences Internationales

1. **Soumia Benkrid**, Ladjel Bellatreche, Cuzzocrea Alfredo, Designing Parallel Relational Data Warehouses: a Global, Comprehensive Approach, ADBIS Special session on Big Data - New Trends and Applications (BIDATA), pp 141-150, Genoa, Italy. Spinger, 2013. [23]
2. Ladjel Bellatreche, **Soumia Benkrid**, Alain Crolotte, Alfredo Cuzzocrea, Ahmad Ghazal: The F&A Methodology and Its Experimental Validation on a Real-Life Parallel Processing Database System. International Conference on Complex, Intelligent and Software Intensive Systems (CISIS), pp. 114-121, IEEE, July, 2012, Palermo, Italy. [8]
3. Ladjel Bellatreche, **Soumia Benkrid**, Ahmad Ghazal, Alain Crolotte, Alfredo Cuzzocrea, Verification of Partitioning & Allocation Techniques on Teradata DBMS, The 11th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP'2011), pp. 158-169, LNCS, Springer, October, 2011, Melbourne, Australia [9].
4. Ladjel BELLATRECHE, Alfredo Cuzzocrea, **Soumia Benkrid**, $\mathcal{F}\&\mathcal{A}$: A Methodology for Effectively and Efficiently Designing Parallel Relational Data Warehouses on Heterogenous Database Clusters, in 12th International Conference on Data Warehousing and Knowledge Discovery (DAWAK'10), pp. 89-104, LNCS, Springer, September, 2010, Bilbao, Spain [15].
5. Ladjel BELLATRECHE, Alfredo Cuzzocrea, **Soumia Benkrid**, Query Optimization over Parallel Relational Data Warehouses in Distributed Environments by Simultaneous Fragmentation and Allocation, The 10th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP), pp. 124-135, LNCS Springer, Busan Korea, May, 2010 [16].
6. Ladjel Bellatreche, **Soumia Benkrid**: A Joint Design Approach of Partitioning and Allocation in Parallel Data Warehouses. 12th International Conference on Data Warehousing and Knowledge Discovery (DAWAK'09), LNCS, Springer, pp. 99-110, Linz, Austria. [7]
7. **Soumia Benkrid**, Ladjel Bellatreche, Habiba Drias: A Combined Selection of Fragmentation and Allocation Schemes in Parallel Data Warehouses. DEXA Workshops 2008, pp. 370-374, Turin, Italy. [24]

Conférences Nationales

1. **Soumia Benkrid**, Ladjel Bellatreche, Cuzzocrea Alfredo, Omniscience dans la Conception des Entrepôts de Données Parallèles sur un Cluster, 9èmes Journées Francophones sur les Entrepôts de Données et Analyse en Ligne (EDA'13), pp.43-52 edited by RNTI, 2013. [131]
2. **Soumia Benkrid**, Ladjel Bellatreche, Une démarche conjointe de fragmentation et de placement dans le cadre des entrepôts de données parallèles, 5èmes Journées francophones sur les Entrepôts de Données et l'Analyse en ligne (EDA'09), pp.91-106 edited by RNTI, 2009. [21]

Première partie

Etat de l'art

Chapitre 1

Les entrepôts de Données Relationnels : États de l'Art

"A data warehouse can't be bought, it must be built".
- Bill Inmon(1945-)

Plusieurs définitions ont été données pour le concept d'entrepôt de données. Nous retenons la définition de W.H. Inmon, considéré comme le père des ED qui le décrit comme "une collection de données orientées sujet, intégrées,non volatiles et historisées, organisées pour supporter un processus d'aide à la décision" [81].

En conséquence, les données possèdent les caractéristiques suivantes.

- *Intégrées.* Les données proviennent de différentes sources souvent structurées et codées de façon différente. L'intégration assure une représentation uniforme, cohérente et transparente. Cela résout les problèmes d'hétérogénéité des systèmes de stockage, des modèles de données et de sémantique de données.
- *Orientées sujets.* Les données s'organisent par sujet ou thème (clients, vendeurs, production, etc.), ce qui permet de rassembler toutes les informations utiles à la prise de décision.
- *Non volatiles.* Les données chargées sont utilisées en mode de consultation. Elles ne peuvent pas être modifiées par l'utilisateur.
- *Historisées.* L'entrepôt de données contient des données archivées afin de les utiliser pour les comparaisons, la prévision, etc.
- *Organisées.* Les informations issues des sources de données doivent être agrégées et réorganisées afin de faciliter le processus de prise de décision.

L'objectif principal d'un entrepôt de données est de fournir *rapidement* et de façon fiable les informations utiles à la prise de décision, cela nécessite la reconstitution des informations afin de les rendre plus facilement compréhensibles et utilisables. Afin d'atteindre cet objectif, l'architecture type d'un entrepôt de données, comme illustrée dans la figure 1, est structurée en quatre axes [90].

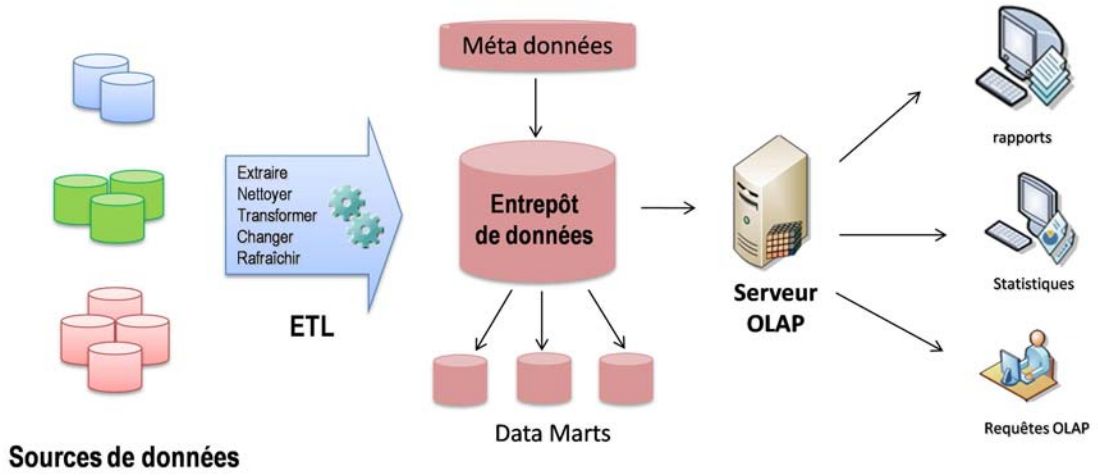


Figure 1 – *Architecture conceptuelle d'un entrepôt de données*

- *Les sources de données.* L'entrepôt de données stocke des données provenant de différentes sources d'informations hétérogènes et distribuées. Ces sources peuvent être des bases de données, des fichiers de données, des sources externes à l'entreprise, ... etc.
- *Le niveau d'extraction de données.* L'extraction est souvent effectuée à l'aide d'un outil d'ETL (Extract, Transform, Load). Elle consiste à aller chercher les données la où elles se situent, à les trier, et à les transformer éventuellement afin d'effectuer un prétraitement pour faciliter l'analyse. Dans cette phase, le nettoyage des données est fait: l'homogénéisation, la suppression des doubles, la détection de données non conformes. Ensuite, les données sont centralisées dans les bases de données de l'entrepôt de données.
- *Le niveau de fusion des données.* Ce niveau assure que les données en provenance des différentes bases concernées de l'entreprise sont intégrées et stockées dans la base de données de l'entrepôt en respectant son organisation par sujets. Ainsi, un entrepôt de données peut comporter plusieurs magasins de données. Ces derniers fes extraits de l'ED consacrés à un type d'utilisateurs et répondant à un besoin spécifique. Ils sont dédiés aux analyses décisionnelles de type OLAP.
- *Le niveau d'exploitation de données.* Ce niveau permet l'analyse et l'exploration des données entreposées. Il autorise la formulation de requêtes complexes afin de retrouver des faits à étudier, l'analyse en tendance des données (courbes d'évolution), l'aide à la prise de décision (extrapolation) et la découverte de connaissances (règles, contraintes, tendances).

Comme les bases de données traditionnelles, la conception des entrepôts de données passe par un cycle de vie [67, 129]. Dans la section suivante, nous détaillons l'ensemble de phases de ce cycle.

1.1 Cycle de vie de conception d'un entrepôt de données

Le cycle de vie de conception d'un entrepôt de données a été récemment revisité dans le cadre de la thèse en cotutelle entre le laboratoire LIAS de l'ISAE-ENSMA - Université de Poitiers et l'ESI, Alger de Selma Khouri [129], où l'ensemble de phases ont été détaillées. Il regroupe les phases suivantes: la planification, la conception et l'implémentation, la maintenance et la gestion de l'évolution et le test.

- **La planification**. Cette phase vise à préparer le terrain pour le développement de l'entrepôt de données. Elle consiste à:
  1. déterminer l'étendue du projet ainsi que les buts et objectifs de l'entrepôt à développer,
  2. évaluer la faisabilité technique et économique de l'entrepôt,
  3. identifier les futurs utilisateurs de l'entrepôt.
- **La conception et l'implémentation**. Cette phase consiste à développer le schéma de l'entrepôt et à mettre en place toutes les ressources nécessaires à son implémentation et à son déploiement.
- **La maintenance et la gestion de l'évolution**. Cette phase implique l'optimisation de ses performances périodiquement. L'évolution de l'entrepôt de données concerne la mise à jour de son schéma en fonction des différents changements survenant au niveau des sources ou des besoins des utilisateurs. Le résultat est une certaine forme de rétroaction, ce qui peut entraîner le retour à l'une des étapes précédentes dans la conception.
- **Les tests**. Tester un entrepôt de données est une tâche cruciale, il permet de tester l'ensemble des phases de conception de l'entrepôt de données en termes de qualité, performance, sécurité, etc. [59].

Le schéma illustré par la figure 1, représente la succession des tâches nécessaires à la mise en place des entrepôts de données efficaces. Chaque rectangle indique une phase.

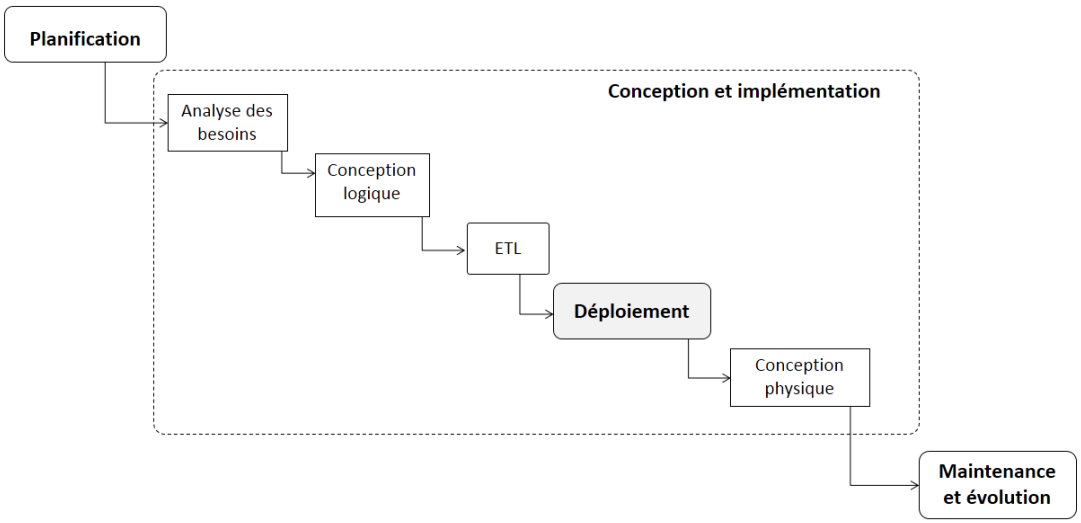


Figure 1.2 – *Cycle de vie des entrepôts de données*

Dans cette thèse nous nous concentrons sur la deuxième phase qui est la conception. Cette dernière comprend cinq étapes principales à savoir: l'analyse de besoins, la modélisation conceptuelle, la modélisation logique, la processus d'extraction-transformation-chargement (ETL) et une phase de modélisation physique [67]. Dans les sections suivantes nous détaillons ces différentes étapes.

1.1.1 Analyse des besoins

La phase d'analyse des besoins est une tâche cruciale sur laquelle repose le processus de la prise de décision. Les exigences sont déterminées pour produire une spécification formelle des données nécessaires au traitement de données, les relations naturelles et la plateforme logicielle pour le déploiement. Tout d'abord, il est nécessaire de distinguer les besoins fonctionnels des besoins non fonctionnels [105, 42]. D'une manière informelle, *les besoins fonctionnels* sont ceux qui caractérisent le système, comme par exemple les besoins en matière de performance, de type de matériel ou de type de conception. Ils peuvent concerner les contraintes d'implémentation (langage de programmation, type SGBD, système d'exploitation,... etc). Par contre, *les exigences non fonctionnelles* sont les exigences implicites auxquelles le système doit répondre. Citons la performance, la réutilisabilité, la fiabilité, ... etc. En particulier, les exigences fonctionnelles d'un entrepôt de données sont principalement liés à l'information que l'entrepôt est censé fournir, tandis que les besoins non fonctionnels n'affectent que les informations nécessaires à une utilisation correcte.

La collecte des exigences fonctionnelles est généralement classée en trois catégories.

– **La collecte axée sur les données** est une technique bottom-up qui commence par l'analyse des sources de données opérationnelles afin d'identifier toutes les données disponibles [68, 86]. Il est conseillé de faire recours à une telle stratégie lorsqu'une connaissance détaillée des sources de données est disponible à priori, que les schémas de sources présentent un bon degré de normalisation, et que la complexité de schémas des sources n'est pas trop élevé. Cette approche simplifie la conception de l'ETL où chaque donnée de l'entrepôt de données correspond à un ou plusieurs attributs des bases de données sources. Les exigences des utilisateurs jouent un rôle secondaire dans la détermination de l'information à analyser, le concepteur se charge de l'identification des faits, des dimensions et des mesures. En conséquence, la qualité du modèle multidimensionnel résultant sera très stable car il est basé sur le schéma des sources de données opérationnelles. Néanmoins, les schémas multidimensionnels obtenus ne peuvent pas répondre aux besoins des utilisateurs. Cela ne se produit pas seulement lorsque les utilisateurs professionnels demandent des renseignements qui ne sont pas effectivement présents dans les sources de données, mais aussi lorsque les indicateurs de performance souhaités ne sont pas directement disponibles et ils ne peuvent pas être obtenus par des calculs. L'approche axée sur données est simple et non coûteuse (en termes de temps et d'argent) car sa durée ne dépend que des compétences de concepteur et de la complexité des sources de données.

– **La collecte axée sur l'utilisateur** est une technique top-down qui intègre et harmonise les points de vue des utilisateurs pour obtenir un ensemble unique de schémas multidimensionnels [145]. L'accent est mis sur les techniques à utiliser pour faciliter la participation de l'utilisateur. Cette approche est très appréciée par les utilisateurs qui se sentent impliqués dans la conception; par contre, ils peuvent être déçus lors de l'élaboration de la cartographie des exigences sur les sources de données disponibles. Généralement, les managers ont une compréhension claire et partagée des objectifs, des processus et de l'organisation de l'entreprise. Ainsi, cette approche nécessite généralement un grand effort de la part du chef de projet, qui doit avoir une très bonne modération et le sens du leadership, afin d'intégrer les différents points de vue. En outre, le risque d'obsolescence des schèmes multidimensionnels résultants est élevé si les exigences exprimées par les utilisateurs sont basées sur les points de vue personnels et n'expriment pas la culture d'entreprise et les procédures de travail.

– **La collecte axées sur les objectifs** est une technique top-down qui focalise sur la stratégie de l'entreprise. Elle est extrapolée en interrogeant les cadres supérieurs de l'entreprise [27]. Différentes visions sont ensuite utilisées pour analyser et fusionner les exigences exprimées et obtenir une image cohérente des indicateurs de performance quantifiables. L'applicabilité de cette approche est strictement liée à la volonté de la haute direction à participer au processus de conception. L'objectif est de maximiser la pertinence des indicateurs identifiés et de réduire ainsi le risque d'obsolescence du schéma multidimensionnel obtenu.

Les schémas multidimensionnels d'un entrepôt de données obtenus en utilisant une seule stratégie sont généralement non complets et ils ne peuvent pas satisfaire les besoins de l'organisation et des utilisateurs. Certains auteurs ont combiné des stratégies axées sur données et d'autres axées sur les utilisateurs. D'autres travaux ont combiné les trois stratégies.

1.1.2 Conception logique

La conception logique d'un entrepôt de données vise à organiser et classer les informations des sources de données par sujet fonctionnel. Elle est préliminaire à la modélisation dimensionnelle où chaque sujet correspond à une table gérée au sein de l'entrepôt. Ce dernier permet d'isoler les données stratégiques et de conserver les métadonnées.

L'interrogation des entrepôts de données se fait par des requêtes OLAP. Ces requêtes correspondent à une structuration des données selon plusieurs axes d'analyse pouvant représenter des notions variées telles que le temps de la localisation géographique ou le code identifiant des produits. Les modèles de conception des systèmes transactionnels ne sont pas adaptés à ce type de requêtes complexes qui utilisent beaucoup de jointures, demandent beaucoup de temps de calcul et sont de nature ad hoc. Pour ce type d'environnement, une nouvelle approche de modélisation a été suggérée: *les modèles multidimensionnels.*

1.1.2.1 Les modèles multidimensionnels

La modélisation multidimensionnelle permet d'observer un sujet analysé comme un point dans un espace à plusieurs dimensions. Les données sont organisées d'une manière qui met en évidence le sujet en cours d'analyse et ses différentes perspectives d'analyse. Ainsi, ce modèle a donné naissance aux concepts de *fait* et de *dimension*.

1. Une *dimension* est une liste d'éléments organisés de façon hiérarchique. Par exemple, pour le client, nous pouvons avoir la hiérarchie: adresse, ville et pays. Ainsi, les tables de dimensions contiennent les niveaux hiérarchiques des dimensions ainsi que les formules à appliquer sur les données numériques pour passer d'un niveau à un autre. La figure 1.3 illustre un exemple d'hiérarchie.

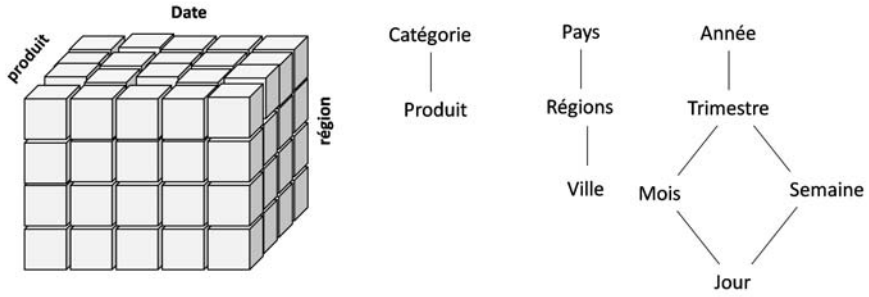


Figure 1.3 – *Vue multidimensionnelle des données*

2. Un *fait* représente le sujet analysé. Il est formé de mesures qui correspondent aux informations liées au thème analysé. Les mesures sont stockées dans des tables de faits qui contiennent les valeurs des mesures et les clés vers les tables de dimensions.

L'objectif majeur de cette modélisation est la vision multidimensionnelle des données, ce qui est assuré via le concept de cube de données. Ce dernier organise les données en une ou plusieurs dimensions qui déterminent une mesure d'intérêt.

Deux approches fondamentales sont utilisées pour construire des systèmes basés sur un modèle multidimensionnel.

1.1.2.2 Les systèmes MOLAP

Les systèmes de type MOLAP ("*Multidimensional On-Line Analytical Processing*") stockent les données dans un SGBD multidimensionnel sous la forme d'un tableau multidimensionnel où chaque dimension est associée à une dimension du cube. L'intérêt de cette approche est l'optimisation du temps d'accès, mais elle présente certaines limites telles que

- le besoin de redéfinir des opérations pour manipuler les structures multidimensionnelles,
- la difficulté de la mise à jour et de la gestion du modèle,
- la consommation de l'espace lorsque les données sont éparses, ce qui nécessite l'utilisation des techniques de compression. [6]

1.1.2.3 Les systèmes ROLAP

Les systèmes de type ROLAP ("*Rolational On-Line Analytical Processing*") utilisent un SGBD relationnel pour stocker les données de l'entrepôt. Le moteur OLAP est un élément supplémentaire qui fournit une vision multidimensionnelle de l'entrepôt, des calculs de données dérivées et des agrégations à différents niveaux. Il est aussi responsable de la génération des requêtes SQL mieux adaptées au schéma relationnel, qui bénéficient des structures d'optimisation existantes pour exécuter efficacement ces requêtes. Ces systèmes peuvent stocker de grands volumes de données, mais ils peuvent présenter un temps de réponse élevé. Leurs principaux avantages sont la facilité d'intégration dans les SGBDs relationnels existants et une bonne efficacité pour stocker les données multidimensionnelles.

Il existe deux schémas principaux pour modéliser les systèmes ROLAP [6].

Schéma en étoile est largement utilisé par les industriels. Il contient une table des faits normalisée et des tables de dimension qui sont généralement dé-normalisées afin de minimiser le nombre de jointures nécessaires pour évaluer une requête. La figure 1.4 illustre un schéma en étoile où la table des faits $VENTES$ stocke la quantité et le montant de vente d'un client pour un magasin donné. Les tables correspondant à $CLIENT$, à $TEMPS$, $PRODUIT$ et $MAGASIN$ comportent les informations pertinentes sur ces dimensions. Il est à noter que la fusion de plusieurs schémas en étoile, qui ont des tables de dimension, donne lieu à un *schéma en constellation*. Ainsi, le schéma contient plusieurs tables de faits.

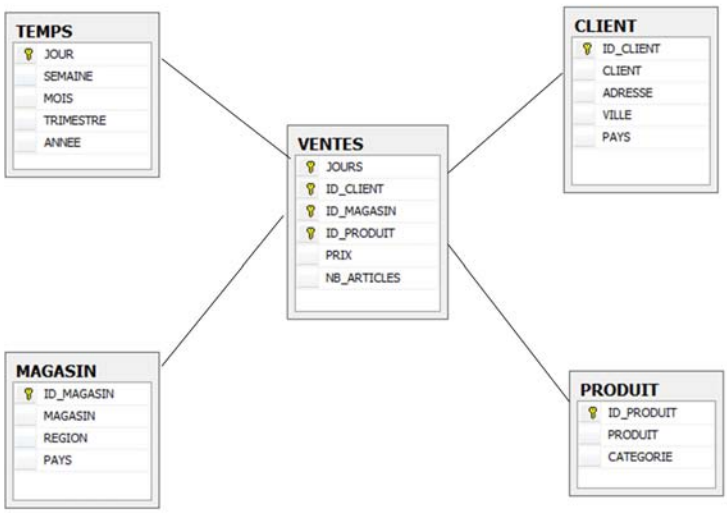


Figure 1.4 – *Exemple d'un schéma en étoile.*

Les requêtes typiques de ce schéma, nommées *requêtes de jointure en étoile (star-join queries)*, ont les caractéristiques suivantes :

- il y a des jointures multiples entre la table des faits et les tables de dimension,
- il n'y a pas de jointure entre les tables de dimensions.

Chaque table de dimensions impliquée dans une opération de jointure a plusieurs prédicats de sélection sur ses attributs descriptifs. La syntaxe générale de ces requêtes est la suivante:

```
SELECT <Liste de projection> <Liste d'agrégation>
FROM <Nom de la table des faits> <Liste de noms de tables de dimension>
WHERE <Liste de prédicats de sélection et de jointure>
GROUP BY <Liste des attributs de tables de dimension>
Order by <Liste des attributs de tables de dimension>
```

Schéma en flocon de neige (snowflake schema) est un raffinement du schéma en étoile. Certaines tables de dimensions sont normalisées selon leur hiérarchie en donnant lieu à de nouvelles tables. La figure 1.5 illustre le schéma en flocon de neige correspond au schéma en étoile de la figure 1.4.

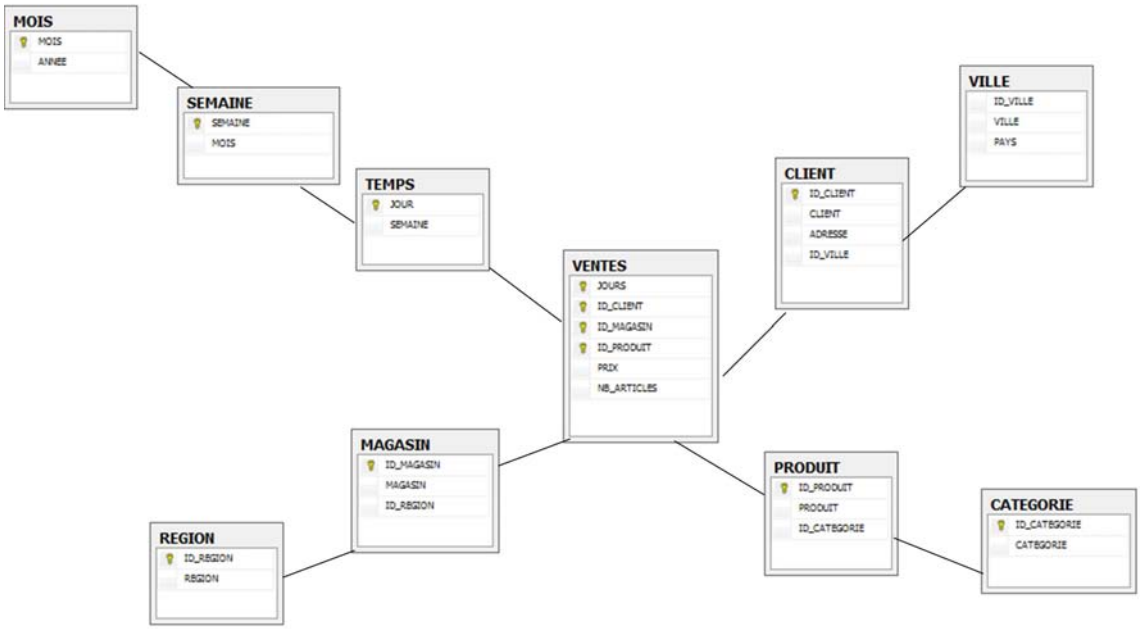


Figure 1.5 – *Exemple d'un schéma en flocon en neige.*

1.1.3 Phase ETL (Extract-Transform-Load)

Pour que des données sources soient exploitables, il est nécessaire de les agréger et de les nettoyer de tous les éléments non indispensables aux utilisateurs finaux. Cette opération d'extraction et d'homogénéisation des données est assurée par la technologie *ETL (Extraction, Transformation and Loading)*. L'ETL se charge de récupérer les données et de les centraliser dans l'entrepôt de données.

L'alimentation ETL se déroule selon les trois phases suivantes [90].

- **Extraction de données**. L'extraction est la première étape du processus d'apport de données à l'entrepôt de données. Extraire, cela veut dire lire et interpréter les données sources et les copier dans la zone de préparation en vue de manipulations ultérieures.
- **Transformation des données**. Il s'agit de l'action de transformer les données pour alimenter l'entrepôt de données. C'est là que le gros du processus ETL a lieu et est habituellement la partie qui prend le plus de temps. La source de données est rarement dans le format que nous voulons pour faciliter les opérations. Par conséquent, il est avantageux d'effectuer différents types de transformations de préparer la structure des données de telle sorte que les données peuvent être utilisées sans avoir besoin de ces manipulations structurelles complexes. Généralement, la partie transformation de ETL se concentre sur la consolidation des données, la correction des données, l'élimination de toute ambiguïté, l'élimination des données redondantes et le renseignement des valeurs manquantes.
- **Chargement des données**. Il s'agit de prendre la sortie de l'étape de transformation et de les placer dans l'emplacement approprié dans l'entrepôt de données. C'est une étape très délicate et exige une certaine connaissance des structures de système de gestion de la base de données (tables et index) afin d'optimiser au mieux le processus.

1.1.4 Conception physique

La conception physique est une étape cruciale du développement des bases/entrepôts de données. Elle est considérée comme une tâche importante de l'administration de l'entrepôt de données. Durant cette phase la description physique est traduite par la spécification des techniques de stockage et de recherche des données. Plus précisément, elle consiste à créer le meilleur modèle de stockage de données qui assure la performance adéquate et l'intégrité de la base de données. Dans les applications décisionnelles, la conception physique est devenue un enjeu important, comme l'indique Chaudhuri dans son papier 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.*". Cette amplification est due aux caractéristiques suivantes liées aux entrepôts de données : (1) le volume de données, (2) la complexité des requêtes et les (3) les exigences des décideurs sur le temps de réponse de requêtes. Pour offrir une meilleure utilisation d'un entrepôt de données, la conception physique est devenue un enjeu important [39].

Durant la phase de conception physique, l'administrateur doit effectuer quatre tâches principales : (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 le déploiement des solutions d'optimisation [93].

1. **Choix des structures d'optimisation** : il existe une large panoplie de structures d'optimisation pour la conception physique. L'identification des structures pertinentes exige un haut niveau d'expertise, car elle dépend de l'étude de la charge à optimiser et de la plateforme sur laquelle celle-ci s'exécute. Nous pouvons citer la fragmentation, les

vues matérialisées, les index, la compression, ... etc.

2. **Choix du mode de sélection** : l'optimisation peut être effectuée en utilisant une ou plusieurs structures d'optimisation. Dans le premier cas, il s'agit d'une sélection isolée. Dans le deuxième, il s'agit d'une sélection multiple. Dans ce cas, plusieurs scenarii sont possibles pour combiner ces techniques [32]. Le choix de l'ensemble des structures à employer et du mode de combinaison est déterminé en matière de performance du système.
3. **Développement des algorithmes de sélection** : les algorithmes proposés pour chaque problème d'optimisation subissent une certaine évolution, en passant d'algorithmes simples vers des algorithmes lourds. Les approches simples sont souvent faciles à mettre en œuvre, mais donnent une faible efficacité (comme les approches de la gestion du buffer [41]). Les algorithmes lourds donnent une meilleur efficacité mais avec un temps d'optimisation élevé (comme les approches de la fragmentation horizontale [35]). Les compromis s'avèrent très rares dans le problème de la conception physique.
4. **Validation et déploiement des solutions d'optimisation** : les recommandations obtenues d'un algorithme de résolution nécessitent une validation par le déploiement sur un environnement réel pour évaluer leur performance effective. Plusieurs outils commerciaux proposent ces services comme *Oracle SQL Acces Advisor* [51] et *DB2 Design Advisor* [153]. Cependant, ces outils présentent des limites liées aux structures d'optimisation et au mode de sélection fourni.

Vu la complexité de cette phase, un problème de la conception physique a été introduit [93]. Il est formalisé comme suit: Etant donné:

- un schéma d'un entrepôt de données,
- une charge de requêtes $\mathcal{Q} = \{Q_1, Q_2, \ldots, Q_L\}$ où chaque requête Q_l possède une fréquence d'accès f_l,
- un ensemble des structures d'optimisation $\mathcal{SO} = \{SO_1, SO_2, \ldots, SO_T\}$ supportées par le SGBD;
- un ensemble de contraintes liées à $\mathcal{SO}$: $\mathcal{C} = \{C_1, C_2, \ldots, C_T\}$ où chaque contrainte C_t est associée à une structure d'optimisation SO_t.

Le problème consiste à sélectionner une ou plusieurs structures d'optimisation pour réduire le coût d'exécution de la charge de requêtes $\mathcal{Q}$ et satisfaire les contraintes définies dans $\mathcal{C}$.

1.1.5 Bilan : vers l'intégration de la phase de déploiement dans un cycle de vie

En analysant la définition du problème de la conception physique, nous avons identifié l'absence de la phase de déploiement concernant la nature de plateforme sur laquelle l'entrepôt de données est stocké. Souvent, elle est implicite, *du fait que les concepteurs considèrent la plateforme de déploiement comme un fait.* Fréquemment, cette phase est intégrée dans les modèles de coût quantifiant la qualité de la solution. Nous proposons alors d'intégrer la phase de déploiement dans la conception physique. En conséquence, le problème de la conception physique devient alors comme suit:

Étant donné:

- un schéma d'un entrepôt de données relationnel *déployé sur une plateforme donnée*;
- une architecture matérielle de cette plateforme avec ses composantes.
- une charge de requêtes $\mathcal{Q} = \{Q_1, Q_2, \ldots, Q_L\}$ où chaque requête Q_l possède une fréquence d'accès f_l,
- un ensemble des structures d'optimisation $\mathcal{SO} = \{SO_1, SO_2, \ldots, SO_T\}$,
- un ensemble de contraintes liées à $\mathcal{SO}$: $\mathcal{C} = \{C_1, C_2, \ldots, C_T\}$.

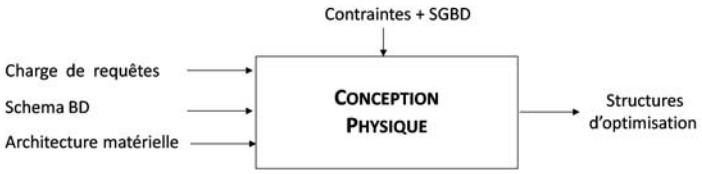


Figure 1.6 – *Structure générique du problème de la conception physique*

Le problème consiste à sélectionner des structures d'optimisation pour réduire le coût d'exécution de la charge de requêtes $\mathcal{Q}$ sur la plateforme et de satisfaire les contraintes définies dans $\mathcal{C}$.

Nous réclamons que la phase de déploiement soit bien explicitée dans le cycle de vie de conception d'entrepôt de données et elle se place juste avant la phase physique comme le montre la Figure **??**.

L'intégration mutuelle de la phase de déploiement dans la phase de conception physique des $\mathcal{ED}$ nous motive de détailler ses phases dans le chapitre suivant.

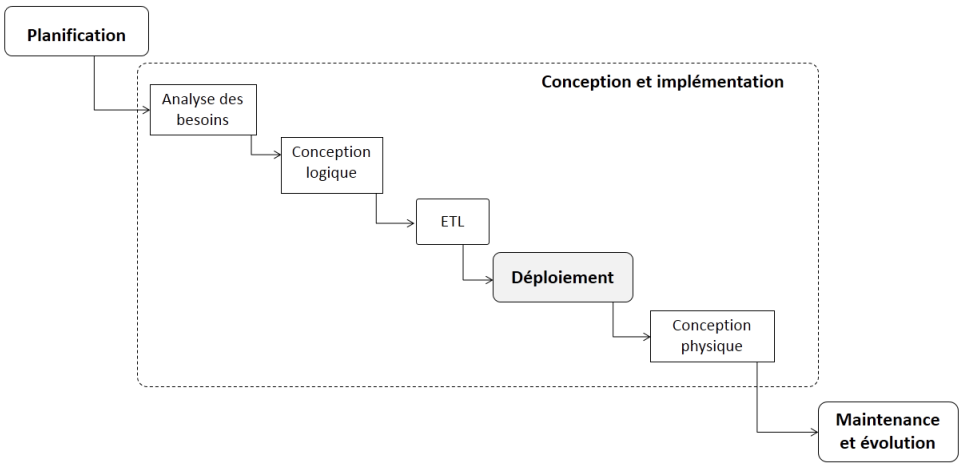


Figure 1.7 – *La place de la phase de déploiement dans le cycle de vie*

Conclusion

Dans ce chapitre nous avons décrit l'ensemble des phases traditionnelles de cycle de vie de conception d'$\mathcal{ED}$: l'analyse de besoins, la conception logique, ETL et la physique. Nous avons identifié le besoin de l'enrichir par la phase de déploiement. Cette phase devient de plus en plus importante vue la diversité des plateformes matérielles disponibles sur le marché (cloud, grilles de calcul, machine parallèle ..., etc.). La connaissance a priori de cette phase facilite aux concepteurs la définition des schémas d'ETL [19], la conception physique, ... etc. de l'$\mathcal{ED}$.

L'étude détaillée de cycle de vie de conception d'un entrepôt de données nous a permis d'associer à chaque phase de ce dernier son propre cycle. Prenons par exemple la phase de l'analyse de besoin, elle même a son propre cycle de vie qui est composé de quatre étapes à savoir: l'élicitation, la modélisation, la spécification et la validation [112]. Sellis and Simitsis [128] ont également identifié un cycle de vie de la phase ETL. Dans le chapitre suivant, nous détaillons les différentes étapes de la phase de déploiement.

Chapitre 2

Cycle de déploiement des entrepôts de données parallèles

"Nothing is particularly hard if you divide it into small jobs".
- Henry Ford (1863-1947)

La maturité et l'usage des bases de données ont amené à la définition d'un cycle de vie bien établi. Autour de ce dernier, un ensemble d'outils académiques et industriels ont été proposés pour concevoir des applications de bases de données (PowerAMC de l'entreprise Sybase, Rational Rose d'IBM, et DBMAIN [3]). Notons qu'à chaque génération de bases de données, ce cycle de vie a évolué. Prenons l'exemple de la phase d'ETL qui n'était pas intégrée au cycle de vie des bases de données classiques. Avec les développements et la sensibilisation des utilisateurs aux outils d'ETL (comme Talend open studio, Pentaho Data Integration, Oracle warehouse builder, ETL) la phase ETL a pris une place importante dans ce cycle.

Nous souhaitons adopter le même raisonnement pour la phase de déploiement parallèle des entrepôts de données. Depuis trois décennies, plusieurs travaux ont été menés dans le cadre des bases de données parallèles et distribuées, où un nombre important de systèmes ont été développés par les chercheurs (*The Gamma Database Machine Project* [56], *Bubba* [28], ...etc.) et les industriels (*Teradata* [137], *IBM Netezza* [46] ...etc.). Cette expérience peut être exploitée pour intégrer la phase de déploiement au cycle de vie de conception de base/entrepôt de données. Cette phase a un ensemble d'étapes qui sont : le choix de l'architecture matérielle parmi la panoplie de plateformes, le partitionnement des données de l'entrepôt et l'allocation des fragments aux nœuds. Cette allocation peut être suivie par une phase de réplication pour assurer une haute disponibilité du système. Finalement, une stratégie de traitement et d'équilibrage de charge doit être définie. (Figure 1).

Dans ce chapitre nous commençons par détailler chaque étape de la phase de déploiement ainsi que les principaux travaux la concernant. Cette description nous ramène à présenter une comparaison de ces travaux. Dans le deuxième temps, nous présentons les grands éditeurs de SGBD parallèles ainsi que les mesures de performance de ces systèmes. Finalement, nous

3. DBMAIN est initialement développé par les chercheurs du Laboratoire d'ingénierie des applications de Bases de Données des Facultés universitaires Notre-Dame de la Paix à Namur en 1991. Depuis janvier 2004,DBMAIN est développé et distribué par REVER S.A

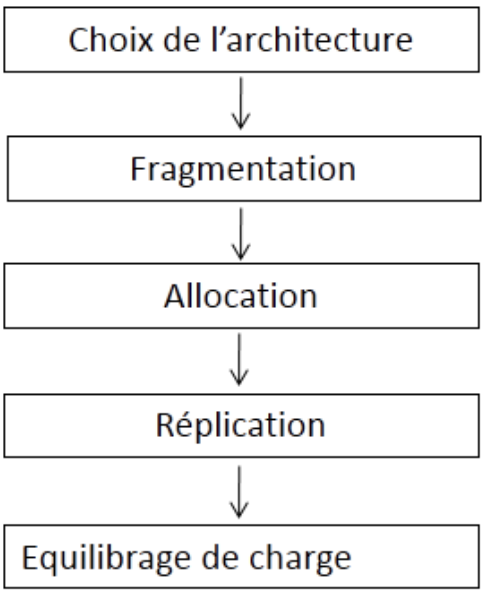


Figure 1 – *Cycle de vie de conception des des entrepôts de données parallèles*

proposons notre vision de composition des étapes de conception d'entrepôts de données sur une plateforme parallèle.

2.1 Les étapes de la phase de déploiement d'un EDP

Comme nous l'avons déjà mentionné, la phase de déploiement est composée de cinq étapes principales : (1) le choix de l 'architecture matérielle, (2) la fragmentation, (3) l'allocation, (4) la réplication et (5) l'équilibrage de charges. Ces étapes sont détaillées dans les sections suivantes.

2.1.1 Choix de l'architecture matérielle

Une plateforme de base de données est constituée d'un ou plusieurs serveurs, d'un système d'exploitation, d'un SGBD et d'un support de stockage des données.

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* [77]. 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). En conséquence, les architectures parallèles sont classifiées selon différentes catégories : shared-

memory, shared-disk, shared-nothing et shared-something. Dans ce qui suit, nous décrivons d'abord les trois architectures conventionnelles et leurs architectures hybrides qui tentent de combiner les avantages de chaque architecture puis les architectures alternatives telles qu'elles sont présentées dans la littérature [119, 135].

2.1.1.1 Architectures conventionnelles

Il y a trois architectures conventionnelles et des architectures hybrides qui tentent de combiner les avantages de chacune d'elles [58, 135]. La figure 2.2 illustre les trois architectures conventionnelles, avec leur architecture hybride.

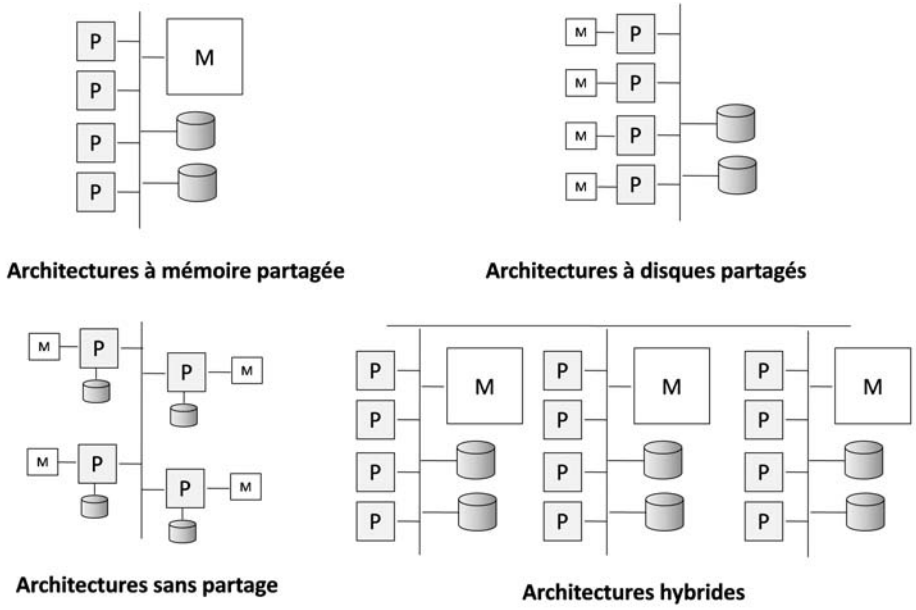


Figure 2.2 – Architetcures matérielles usuelles

2.1.1.1.1 Architectures à mémoire partagée (shared-memory). Les processeurs et les disques ont accès à une mémoire commune, typiquement par un bus ou un réseau d'interconnexion. L'intérêt de l'utilisation de cette architecture est l'efficience de la communication entre les processeurs ; les données sont accessibles par n'importe quels processeurs. Chaque processeur peut envoyer rapidement un message aux autres en écrivant en mémoire au lieu d'utiliser les canaux de communication. L'inconvénient majeur de cette architecture est le fait qu'elle n'est pas scalable (32 à 64 nœuds). Cela est dû au fait que le bus (le réseau d'interconnexion) devient un goulot d'étranglement. L'ajout de nouveaux processeurs implique l'accroissement du temps d'attente pour accéder à la mémoire principale partagée. Généralement, l'architecture à mémoire partagée a une large mémoire cache au niveau de chaque processeur de sorte que le référencement de la mémoire partagée est évité autant que possible. En outre, les caches doivent être cohérents, c'est-à-dire que si un processeur effectue une écriture dans un emplacement mémoire, les données de cet emplacement mémoire doivent être soit mises à jour ou

supprimées de n'importe quel autre processeur où les données sont mises en cache.

2.1.1.1.2 Architectures à disques partagés (shared disks). Chaque processeur détient sa propre mémoire centrale et le disque est partagé entre tous les processeurs. L'architecture à disques partagés est similaire à l'architecture à mémoire dans le sens qu'une mémoire secondaire est partagée. Elle souffre de congestion dans le réseau d'interconnexion quand plusieurs processeurs essayent d'accéder au disque en même temps. En effet, le traitement de l'ensemble des sous-requêtes nécessite la récupération des données du disque partagé pour les stocker dans sa mémoire locale. Ainsi, la différence principale entre l'architecture à mémoire partagée et celle à disque partagé est la hiérarchie de la mémoire qui est partagée.

Dans le contexte des nouvelles architectures logicielles, les architectures à disques partagés et à mémoire partagée sont considérés comme des *machines multiprocesseurs symétriques* (Symmetric Multi Processor, SMP).

Une machine SMP typique est constituée de plusieurs CPUs allant de 2 à 16 CPUs. Donc, un nombre élevé des processeurs n'est pas tolérable à cause des problèmes de passage à l'échelle. Chaque processeur maintient son propre cache et une mémoire principale partagée entre tous les processeurs. La taille de la mémoire principale et du cache diffère d'une machine à l'autre. Plusieurs disques peuvent être attachés à une machine SMP et tous les CPU auront un accès similaire. Le système d'exploitation alloue normalement les tâches selon un ordonnanceur. Un processeur est inactif, une tâche dans sa file d'attente est immédiatement attribuée. De cette façon, l'équilibrage est relativement facile à réaliser.

2.1.1.1.3 Architectures sans partage (shared-nothing) attribut à chaque nœud sa propre mémoire et son propre disque. Les processeurs communiquent entre eux via un réseau de communication à haut débit. Par ailleurs, les réseaux d'interconnexion des systèmes sans partage sont généralement conçus pour être évolutifs, de sorte que leur capacité de transmission augmente avec le nombre des nœuds qui sont ajoutés. En conséquence, l'architecture sans partage est plus scalable et peut aisément supporter un large nombre de processus. *Teradata, Grace* et le prototype de recherche Gamma sont de type *shared nothing*. Le problème de compétition pour accéder aux données partagées ne se pose pas pour cette architecture mais le problème d'équilibrage de charge est difficile à atteindre même pour les requêtes simples car les données sont stockées localement au niveau de chaque disque. Le problème de la mauvaise distribution est l'un des challenges dans le traitement parallèle des requêtes sur une machine de ce type.

Dans le contexte des nouvelles architectures logicielles, la machine sans partage est classifiée comme machine massivement parallèle(Massively Parallel Processing, MPP). Cette architecture est caractérisée par le haut débit de son réseau d'interconnexion.

2.1.1.1.4 Architectures hybrides. Il s'agit d'une combinaison des architectures sans partage et à mémoires partagées. Elle combine les avantages de chacune et compense leurs inconvénients respectifs. Elle réalise à la fois l'équilibre de charge des architectures à mémoires partagées et l'extensibilité des architectures sans partage. Cette architecture est nommée *Sha-*

red Something. Elle augmente la flexibilité de la configuration (nombre des nœuds et taille des nœuds) et diminue le coût de communication réseau car le nombre de nœuds est réduit. Le parallélisme intra-requête peut être isolé à un seul multiprocesseur shared-memory car il est beaucoup plus facile de paralléliser une requête dans une architecture à mémoire partagée que dans une architecture sans partage. En outre, le degré de parallélisme sur un seul nœud à mémoire partagée peut être suffisant pour la plupart des applications. Autrement dit, le parallélisme intra-requête est obtenu à travers l'exécution parallèle sur les nœuds.

Il en existe plusieurs variantes de cette architecture mais fondamentalement chaque nœud est une machine à mémoire partagée connectée au réseau d'interconnexion de l'architecture sans partage.

2.1.1.2 Architectures distribuées

Aujourd'hui, nous assistons à une émergence dans le développement des ordinateurs et des réseaux à haut débit. Cela a donné naissance à de nouvelles architectures matérielles distribuées qui peuvent être utilisées comme des plateformes logicielles pour le déploiement des systèmes parallèles

2.1.1.2.1 Les architectures grappes de machines (cluster). Un cluster est un ensemble de nœuds interconnectés pour partager des ressources pour un seul système. Les ressources partagées peuvent être un matériel comme un disque ou un logiciel comme un système de gestion de données. Les nœuds d'un cluster sont des composantes simples comme un micro-ordinateur (PC) ou des machines plus puissantes comme les SMP. L'utilisation des composantes off-the-shelf est essentielle pour obtenir le meilleur rapport prix/performance tout en exploitant le progrès continu des composantes matérielles. Dans sa forme la moins chère, l'interconnexion entres les nœuds peut être un simple réseau. Cependant, il existe actuellement un standard pour l'interconnexion des nœuds d'un cluster nommé *Myrinet and Infiniband* qui produit des réseaux à haut débit (Gigabits/sec) avec une faible latence pour le transfert des messages. Contrairement aux systèmes distribués, un cluster est regroupé géographiquement (sur le même site) et il est généralement homogène. L'architecture d'un cluster peut être soit sans partage soit à disque partagé.

Les clusters sans partage ont été largement utilisés car ils fournissent le meilleur rapport *qualité/prix* et peuvent atteindre des milliers de nœuds (evolutifs).

Les clusters à disques partagés existent sous deux formats.

- un NAS est une plateforme qui partage le disque sur un réseau en utilisant un protocole de distribution des fichiers systèmes comme *Network File System*. Le NAS est bien adapté pour les applications à faible débit telles que la sauvegarde et l'archivage de données à partir de disques durs des PC. Cependant, il est relativement lent et il ne convient pas pour la gestion des bases de données car il devient rapidement un goulot d'étranglement avec de nombreux nœuds,
- un SAN produit des fonctionnalités similaires mais avec des interfaces à niveau inférieur. Il utilise un protocole basé sur des blocs, ce qui facilite la gestion de la cohérence du cache. SAN fournit un haut débit de données et peut évoluer jusqu'à un grand nombre de nœuds.

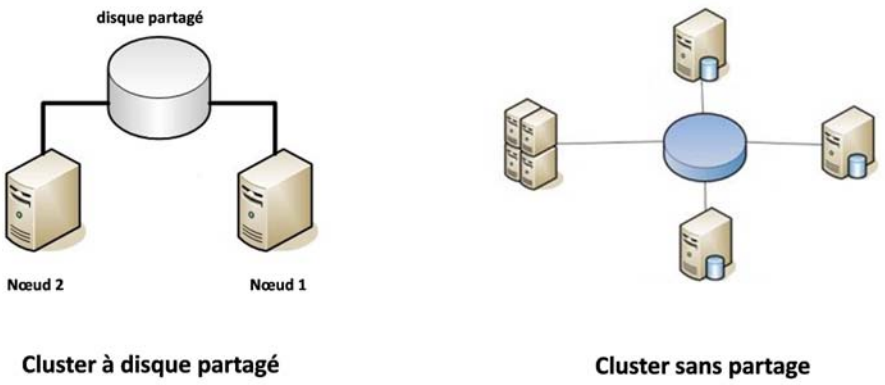


Figure 2.3 – *Exemple d'un cluster d'ordinateurs*

Sa seule limitation à l'égard de *shared-nothing* est son coût d'acquisition élevé.

Les architectures clusters ont beaucoup d'avantages. D'une part, elles combinent la flexibilité et la performance des architectures à mémoires partagées au niveau de chaque nœud. D'autre part, elles assurent l'extensibilité et la disponibilité de l'architecture à disques partagés ou sans partage. En outre, l'utilisation des nœuds off-the-shelf à mémoire partagée avec une interconnexion standard de clusters fournit une alternative rentable pour approprier une plateforme à haute performance comme $NUMA$ ou MPP.

2.1.1.2.2 Grille de calcul. Elle découle de la combinaison de ressources informatiques à partir de multiples domaines administratifs appliqués à une tâche commune, généralement à un problème scientifique, qui nécessite le traitement d'une grande quantité de données. Cette architecture permet le partage, la sélection et l'agrégation de ressources autonomes réparties géographiquement de manière dynamique durant l'exécution des requêtes en fonction de leur disponibilité, capacité de stockage, puissance de calcul, coût et qualité de service [75]. Ce sont rien d'autre que des clusters à grande échelle, que la figure 2.4 illustre un exemple d'infrastructure grille de calcul.

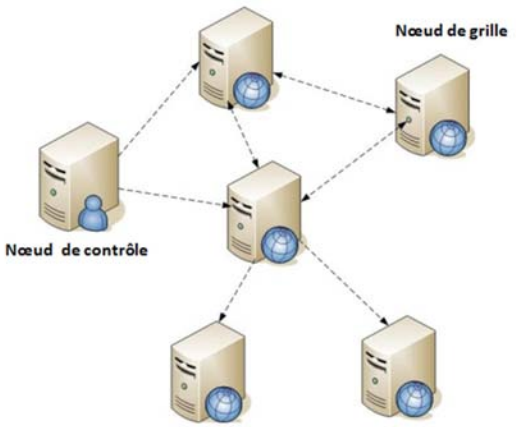


Figure 2.4 – *Exemple de l'infrastructure grille de calcul*

2.1.1.2.3 Cloud de calcul. Le *Cloud* est particulièrement avantageux pour les petites et moyennes entreprises qui souhaitent externaliser complètement leurs infrastructures de centres de données, ou les grandes entreprises qui souhaitent obtenir une haute capacité de calcul et de stockage sans engager de coût élevé de construction de centres de calcul importants en interne. Le *Cloud* est une extension de ce paradigme où les capacités des applications sont exposées comme des services sophistiqués qui peuvent être accessibles sur un réseau facturés selon la consommation.

Un nuage (*Cloud*) est un système parallèle composé d'un ensemble d'ordinateurs interconnectés, dynamiquement provisionnés et présentés comme une ou plusieurs ressources informatiques unifiées basées sur le service. Fondamentalement, un nuage peut être : un *nuage public* qui désigne une structure souple et ouverte dédié à la vente de services dont les informations peuvent être consultées à partir d'Internet. Un *Cloud privé* est un réseau propriétaire ou un centre de calcul qui fournit des services hébergés à un nombre limité de personnes. Les informations et les applications qui s'exécutent sur le Cloud privé peuvent être consultées à partir de l'intranet de l'entreprise en utilisant une connexion VPN entre le centre de calcul interne et l'infrastructure du sous-traitant. La figure 2.5 illustre un exemple de l'infrastructure Cloud de calcul.

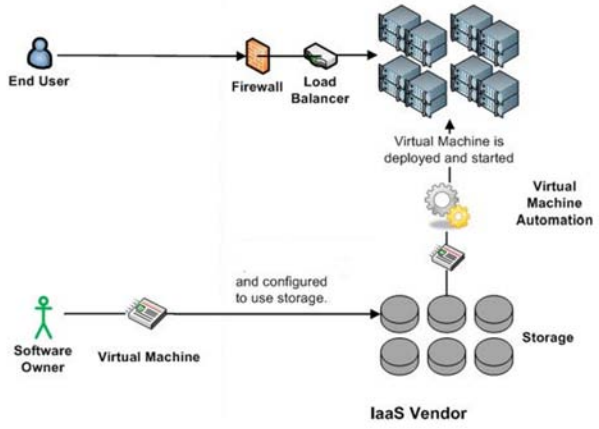


Figure 2.5 – *Exemple de l'infrastructure cloud de calcul*

2.1.2 Fragmentation de données.

La fragmentation est le processus de décomposer des objets d'accès (tables, vues matérialisées, index) en un ensemble de partitions disjointes. Elle a été introduite à la fin des années 70 et au début des années 80 [35] comme une *technique de conception logique* de bases de données traditionnelles, distribuées [117] et parallèles [58, 142]. Avec le développement des entrepôts de données, la fragmentation est devenue une des *structures d'optimisation la plus importante* de la phase de conception physique. Actuellement, les éditeurs de bases de données commerciaux et académiques proposent des modes de partitionnement de tables en utilisant des modes simples (*Hash, List et Range*) et des modes composés, toute combinaison deux à deux de modes simples

(ex. *Range-Range, Hash-List*, etc.). Le SGBD Oracle propose un outil (partitioning advisor) dans sa version 11G qui recommande aux administrateurs le bon partitionnement en fonction de la charge de requêtes. La figure 2.6 montre l'évolution de la fragmentation dans les travaux de recherche menés par la communauté des bases de données.

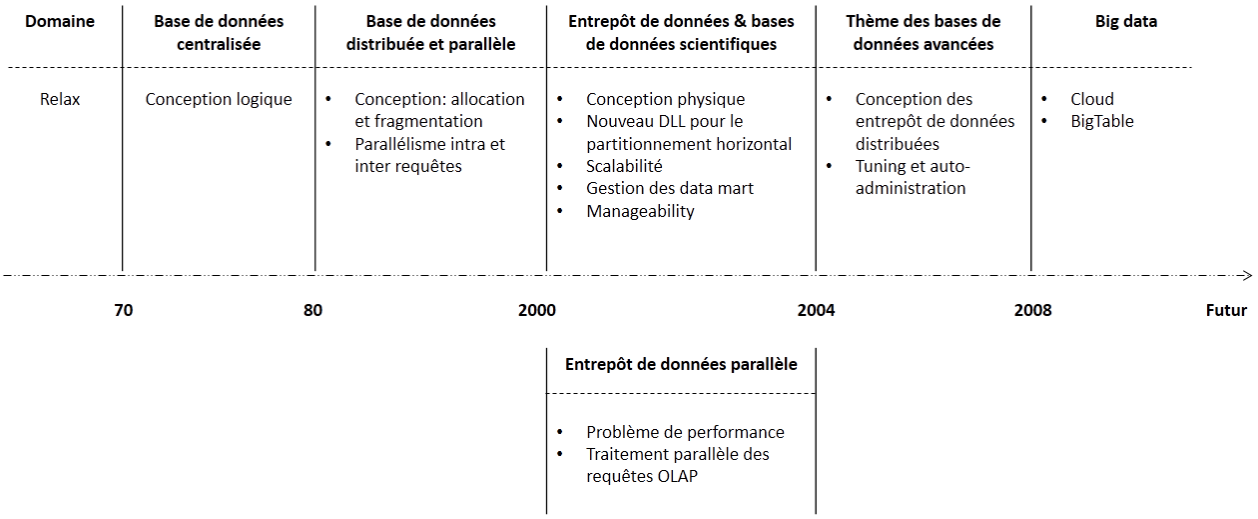


Figure 2.6 – *Evolution de la fragmentation de données*

Dans ce qui suit, nous présentons les types de fragmentation. Puis nous décrivons les principaux travaux existants dans la littérature.

2.1.2.1 Les types de fragmentation.

Dans la littérature, trois types de fragmentation sont définis : la fragmentation verticale, la fragmentation horizontale et la fragmentation mixte.

2.1.2.1.1 La fragmentation verticale. La fragmentation verticale consiste à diviser une relation R en sous relations appelées *fragments verticaux* résultant de l'application de l'opération de projection. Elle favorise naturellement le traitement des requêtes de projection portant sur les attributs utilisés dans le processus de la fragmentation, en limitant le nombre de fragments auxquels accéder. Mais elle requiert des jointures supplémentaires lorsqu'une requête accède à plusieurs fragments.

Exemple 1. *Soit la table Client (idClient Nom, Ville, Sexe) partitionnée comme suit*

$$Clients_1 : \prod_{idClient,Sexe} (Client)$$
$$Clients_2 : \prod_{idClient,Nom,Ville} (Client)$$

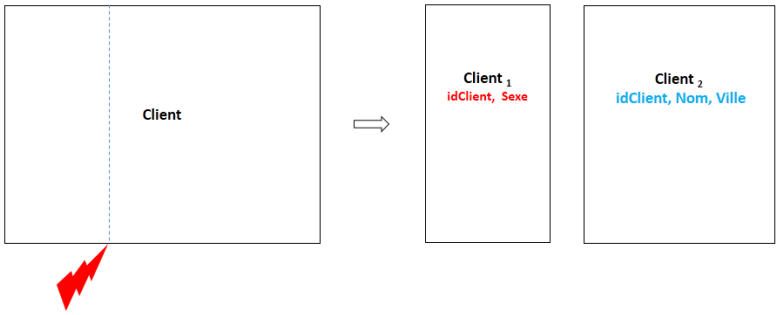


Figure 2.7 – Exemple d'une fragmentation verticale

2.1.2.1.2 La fragmentation horizontale. La fragmentation horizontale consiste à diviser un objet $\mathcal{O}$ (table, index, vues) de la base de données en sous-ensembles de lignes appelés fragments horizontaux résultant de l'application de l'opération de restriction. La reconstruction de l'objet $\mathcal{O}$ à partir de ces fragments horizontaux est obtenue par l'opération d'union de ces fragments.

Il existe deux versions de la fragmentation horizontale. *La fragmentation primaire* s'effectue grâce à des prédicats de sélection définis sur la relation. Par contre, *la fragmentation dérivée* se fait avec des prédicats de sélection définis sur une autre relation. Concrètement, la fragmentation dérivée d'une table S n'est possible que lorsqu'elle est liée avec une table T par sa clé étrangère. Une fois la table T fragmentée par fragmentation primaire, les fragments de S sont générés par une opération de semi-jointure entre S et chaque fragment de la table T . Les deux tables seront équi-partitionnées grâce au lien père-fils. A partir de ces deux définitions, nous constatons que la fragmentation primaire pourrait accélérer les opérations de sélection tandis que la fragmentation dérivée accélérerait les opérations de jointure.

Exemple 2. *Soit un schéma d'une base de données constitué d'une table Client et d'une table Ventes liée à la table Client par une relation de clé étrangère. La table Client est fragmentée en trois fragments, $Client_1$, $Client_2$, $Client_3$. Chaque fragment est défini par un prédicat de sélection sur l'attribut Ville de cette table:*

$$Clients_1 : \sigma_{Ville=Poitiers}\ (Client)$$
$$Clients_2 : \sigma_{Ville=Paris}\ (Client)$$
$$Clients_3 : \sigma_{Ville=Alger}\ (Client)$$

A partir du schéma de fragmentation de la table Client, la table Ventes est fragmentée en trois fragments en fonction des trois fragments de la table Client. Chaque fragment de la table Ventes est généré à l'aide d'une opération de semi-jointure ($\ltimes$) entre un fragment de la table Client et la table des faits comme suit :

$$Ventes_1 = Ventes \ltimes Clients_1$$
$$Ventes_2 = Ventes \ltimes Clients_2$$
$$Ventes_3 = Ventes \ltimes Clients_3$$

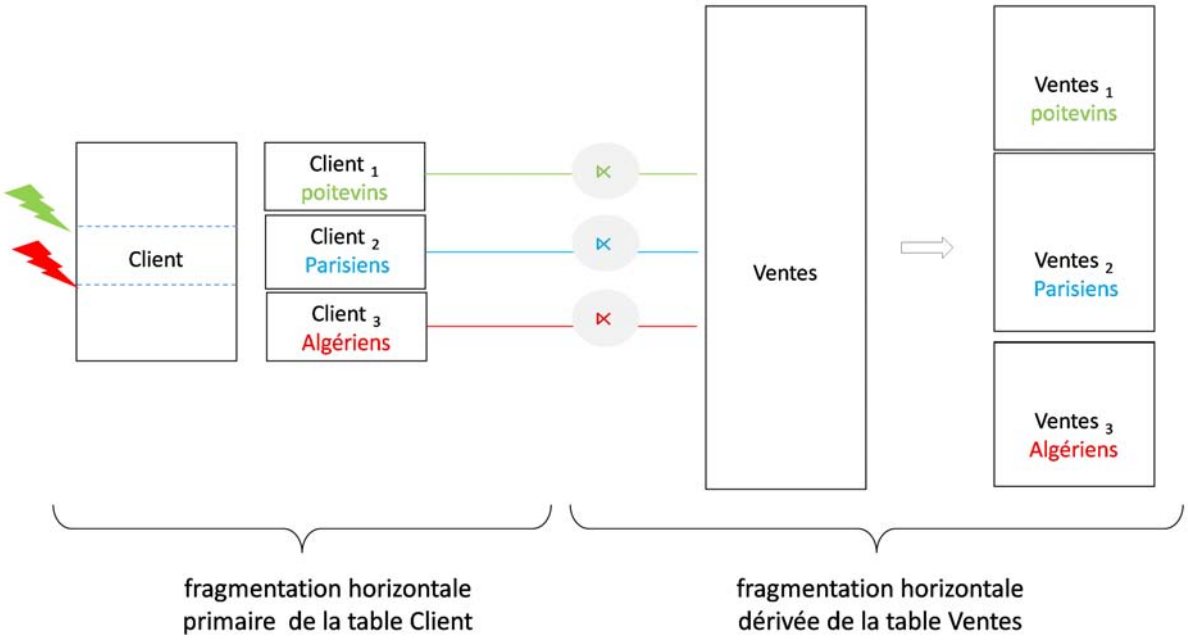


Figure 2.8 – *Exemple de la fragmentation horizontale*

Le schéma de fragmentation de notre base de données est illustrée dans la figure 2.8

2.1.2.1.3 La fragmentation mixte. La fragmentation mixte résulte de la combinaison des deux sortes de fragmentation citées ci-dessus; pour atteindre les avantages de chaque type de fragmentation. Elle consiste à partitionner une relation en sous ensemble d'ensemble, cette dernière étant définie par la fragmentation verticale et les sous ensembles par la fragmentation horizontale. Un exemple de la fragmentation mixte est illustré dans la figure 2.9

2.1.2.2 Problème de la fragmentation horizontale

Dans cette section, nous nous intéressons à la fragmentation horizontale, considérée comme une pré-condition de la conception de base/entrepôt de données parallèles [102]. Sélectionner un schéma de fragmentation d'une base/entrepôt de données est une tâche difficile [32]. Cette complexité est en relation avec le nombre de prédicats de sélection figurant dans une charge de requêtes.

Le problème de la fragmentation horizontale peut être formalisé comme suit: Étant donné un schéma d'une base/entrepôt de données, une charge de requêtes, le problème de la fragmentation consiste à partitionner les tables de cette base en fragments afin d'optimiser le coût d'exécution de la charge de requêtes.

La fonction objectif à optimiser dépend fortement de la plateforme de déploiement. Nous détaillons l'ensemble des travaux existants sur la résolution de ce problème dans le cas centralisé, distribué, et parallèle.

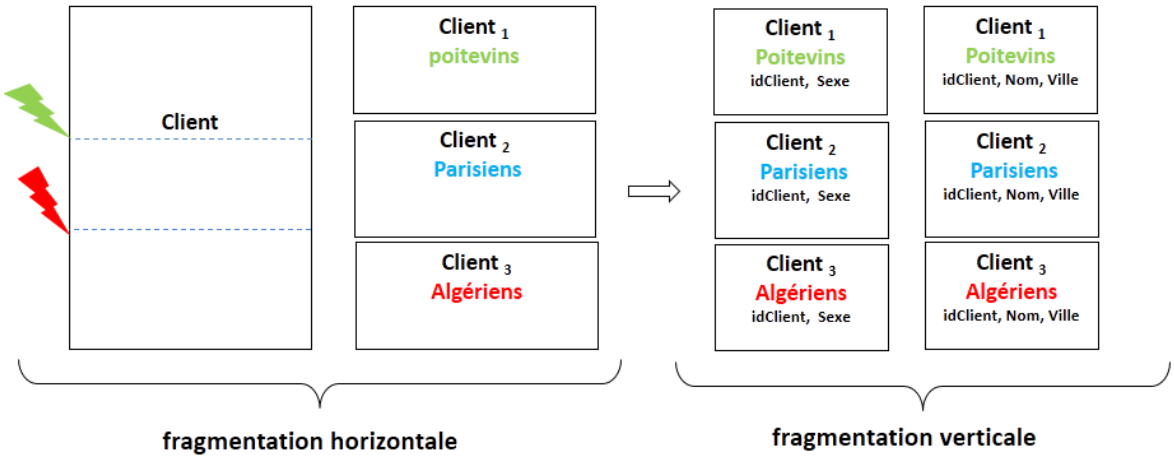


Figure 2.9 – *Exemple de la fragmentation horizontale*

2.1.2.2.1 Travaux existants .

La fragmentation horizontale a été largement adoptée par la communauté des bases de données. Plusieurs travaux ont été proposés. Nous citons quelques-uns ici.

2.1.2.2.1.1 Travaux existants dans le contexte centralisé

Travaux de Bellatreche .

Bellatreche [6] présente un nouvel algorithme de fragmentation horizontale dérivée sur un schéma en étoile et propose plusieurs approches basées sur un ensemble de requêtes. L'auteur ajuste les algorithmes proposés dans le contexte des bases de données réparties. Ces algorithmes se basent sur la complétude et la minimalité des prédicats ou sur les affinités des requêtes. Bellatreche remarque que ces méthodes génèrent un nombre important de fragments et rendent ainsi leur processus de maintenance très coûteux. Pour remédier à cet inconvénient, il propose des algorithmes de sélection d'un schéma de fragmentation optimal. Ces algorithmes visent à trouver un accord entre le coût de maintenance des fragments et le coût d'exécution des requêtes. Ils sont fondés sur des modèles de coût et procèdent en trois étapes : génération de plusieurs schémas de fragmentation, évaluation de ces schémas et sélection d'un schéma optimal.

Le premier algorithme proposé est exhaustif et consiste à construire tous les schémas de fragmentation possibles par fragmentation horizontale. Il énumère ensuite ces schémas et calcule pour chacun d'eux le coût d'exécution des requêtes de la charge. Il sélectionne finalement le schéma qui coïncide au coût minimum. Le deuxième algorithme est approximatif. Il construit un schéma initial par l'algorithme de fragmentation dirigé par les affinités, puis l'améliore par des opérations de fusion ou de décomposition des fragments.

Travaux de Mahboubi .

Les auteurs proposent une approche qui repose sur la technique de data mining pour la classification des prédicats de sélection extraits de la charge de requête Q. Elle a été proposée par Mahboubi [50] pour la fragmentation d'un entrepôt de données XML. L'approche se déroule en trois étapes.

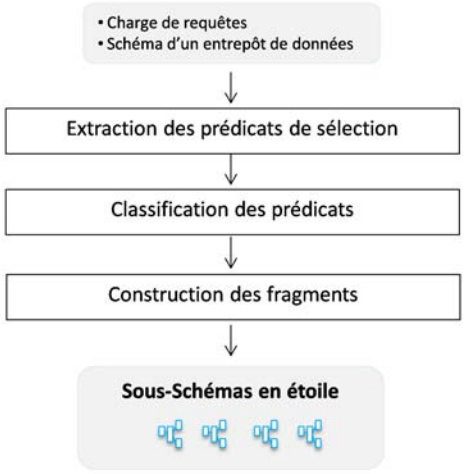


Figure 2.10 – *Approche de fragmentation basée sur le data mining* [50]

- **Extraction des prédicats de sélection**. Les prédicats de sélection de l'ensemble sont codés dans une matrice requêtes-prédicats QP. Cette dernière constitue le contexte de classification. La valeur de $QP[i][j]$ est égale à 1 si le prédicat p_j apparaît dans la requête q_i et à 0 sinon.
- **Classification des prédicats**. Les fragments horizontaux sont construits à partir de l'ensemble de prédicats. Les auteurs proposent de regrouper en classes les prédicats présentant des similarités au niveau syntaxique et ils adaptent l'algorithme de classification $k - Means$. $k - Means$ prend en entrée l'ensemble des prédicats P et le paramètre k qui représente au même temps le nombre de classes et de fragments. Il fournit en sortie l'ensemble de classes de prédicats C qui représente l'ensemble des fragments horizontaux.
- **Construction des fragments**. Chaque classe de C représente un fragment et est constituée d'un ensemble de prédicats. Pour chaque classe, les dimensions conformes aux prédicats sont identifiées à partir du document XML qui représente le schéma de l'entrepôt. Leurs noms sont indiqués par les éléments dimension et les prédicats qui leur correspondent par les éléments prédicats.

Travaux de Boukhalfa *et al* .

Les auteurs proposent [10] une fragmentation horizontale basée sur les algorithmes génétiques pour sélectionner les tables de dimension à fragmenter pour éviter l'explosion du nombre de fragments de la table des faits et de garantir une meilleure performance d'exécution des requêtes. En 2006, les auteurs [12] combinent l'algorithme génétique et le recuit simulé pour sélectionner le schéma de fragmentation horizontale d'un entrepôt de données. En 2008, Bou-

khalfa et ses collègues [13] proposent un algorithme de type Hill Climbing et ils le comparent ensuite avec l'algorithme de sélection de schéma de fragmentation via les algorithmes génétiques. Ils le comparent également à l'algorithme de sélection de schéma de fragmentation via le recuit simulé. Les résultats montrent l'efficacité (en termes de performance) de l'algorithme du recuit simulé. Boukhalfa *et al* ont également effectué une validation sous Oracle10g (ORACLE offre plusieurs modes de partitionnement) avec les données du banc d'essai APB-1 Council.

2.1.2.2.1.2 Travaux existants dans le contexte distribué

Travaux de Noaman et Barker .

Pour construire un entrepôt de données distribué, les auteurs exploitent une politique descendante pour la fragmentation horizontale [111] Elle part du schéma conceptuel global d'un entrepôt, qu'elle répartit pour construire les schémas conceptuels locaux. Cette répartition se fait en deux étapes essentielles : la fragmentation et l'allocation, suivies éventuellement d'une optimisation locale. Les auteurs offrent un algorithme qui dérive des fragments faits en se basant sur des requêtes définies sur les dimensions.

Travaux de Darabant et Campan .

Les auteurs proposent une méthode de fragmentation horizontale d'une base de données orientée objets distribuée [52]. Ils se basent sur une classification par la technique des k-means. Cette technique classe les instances d'objets dans des fragments en tenant compte des relations entre les classes (agrégation, associations et liens entre méthodes) et regroupe les objets similaires en se basant sur des conditions extraites des requêtes utilisateurs. Pour cela, les auteurs proposent des fonctions de similarité entre objets calculées selon différentes métriques et des méthodes qui permettent de choisir une distribution de classes initiale selon la sémantique des requêtes.

En 2005, Darabant propose d'utiliser cette technique pour partitionner une base de données orienté objets avec des attributs et des méthodes complexes [53]. L'auteur définit un attribut complexe comme un attribut de type ensemble ou intervalle, et une méthode complexe comme une méthode qui fait appel à une autre méthode d'une autre classe. L'auteur propose dans ces travaux une fragmentation horizontale dérivée. Dans une base de données orientée objet, cette fragmentation s'applique en deux étapes : (1) une fragmentation primaire qui groupe les instances de classes selon des conditions définies sur les attributs des classes ; et (2) une fragmentation dérivée qui regroupe les instances d'une classe selon les fragments des classes mères. L'algorithme de fragmentation proposé par l'auteur prend en compte les relations entre les classes (agrégation, associations et liens entre méthodes complexes) et vise à unifier les étapes de fragmentation horizontale dérivée (primaire et dérivée) en une seule.

2.1.2.2.1.3 Travaux existants dans le contexte parallèle

Travaux de Zilio et *al* .

Zilio s'est intéressé au problème de partitionnement de données dans les systèmes de gestion de base de données Shared Nothing [152]. Plus précisément, il traite le problème de sélection des attributs de partitionnement. Pour cela, il a proposé deux algorithmes de sélection des attributs de partitionnement. Le premier algorithme nommé *Independent Relation (IR)* utilise un graphe de requêtes pondérées pour générer l'importance de chaque attribut. Il comporte quatre étapes (voir figure 2.11).

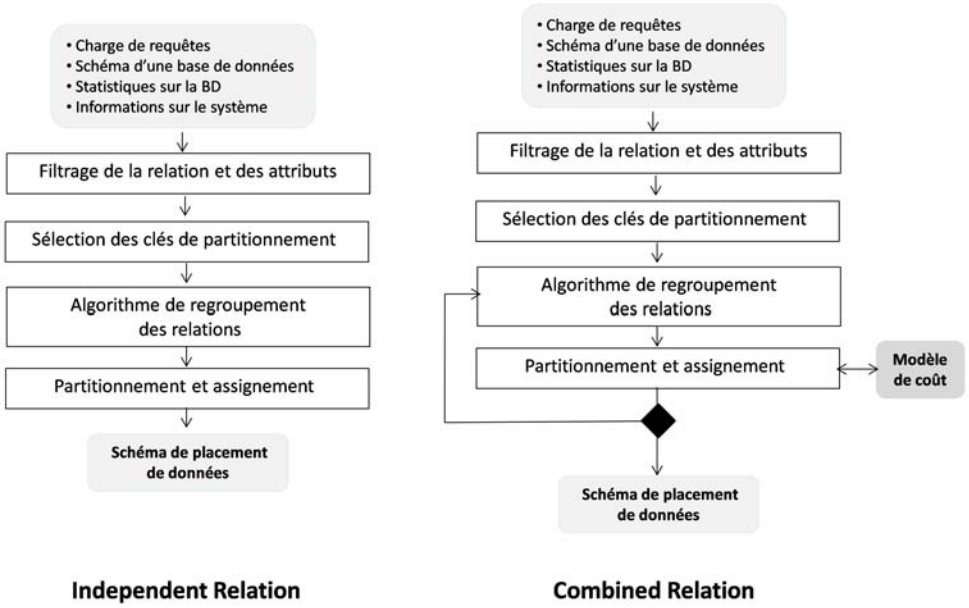


Figure 2.11 – *Independent Relation*

- *Filtrage de la relation et des attributs.* Dans le partitionnement par hachage ou par intervalle, le choix des colonnes ayant une cardinalité basse ou une mauvaise distribution de données, lorsqu'elles sont utilisées comme des attributs de partitionnement, induit généralement une distribution biaisée de la charge de travail. En effet, une bonne distribution est conditionnée par le choix d'un attribut dont le nombre de valeurs distinctes est inférieur à $10N$ (N est le nombre des nœuds) ou si certaines valeurs particulières présentent une fréquence d'apparition supérieure à $0.5/N$. De plus, les petites relations sont soit partitionnées et allouées à un seul nœud soit répliquées sur tous nœuds.
- *Sélection des clés de partitionnement.* Ce choix est fait selon les poids affectés aux attributs et les attributs ayant le poids le plus élevé méritent d'être choisis comme des clés de partitionnent d'une relation. En effet, un poids de pré-assignement est affecté à tous opérateurs importants dans le contexte de l'optimisation des requêtes parallèles et le poids d'un attribut est calculé par l'agrégation des poids des opérateurs qui l'utilisent dans la charge de requêtes. Le poids d'agrégation final d'un attribut reflète son importance dans la charge de travail. Ainsi, les attributs ayant un poids élevé seront choisis comme des clés de partitionnement, car ils sont susceptibles de bénéficier de l'ensemble des opérations qui contribuent à leur poids d'agrégation.
- *regroupement des relations.* Zilio effectue la fermeture transitive des clés de partitionnement sélectionnées selon les clauses de jointure figurant dans la charge de requêtes.

– *Partitionnement et assignement des relations*: L'idée de cette étape été emprunter des travaux de [44, 102]. Cette phase comporte trois étapes.

1. partitionner les relations en un ensemble de partitions dont le nombre est plus large que le nombre de nœud;
2. placer les grandes relations sur tous les nœuds du système;
3. placer les petites relations en utilisant la stratégie de remplissage (Bin Packing) de telle sorte que les données seront placées d'une manière équilibrée. Le partitionnement et l'assignement des petites relations sont effectués en même temps. L'algorithme recherche le meilleur emplacement et il vielle à ce que le poids de chaque nœud soit égal à la moyenne des poids. Les relations sont triées selon leur poids et allouées sur les nœuds de telle sorte que la contrainte de stockage est satisfaite.

L'inconvénient majeur de l'algorithme IR est qu'il n'utilise aucun modèle de coût pour estimer la qualité des clés de partitionnement utilisées. En effet, IR se base uniquement sur la pondération des opérateurs pour sélectionner les clés de partitionnement

Zilio a proposé un autre algorithme nommé *Combined Relation (Comb)* (Figure 2.11)qui génère la combinaison des clés pour toutes les tables. Comb utilise un optimiseur de requête pour choisir la meilleure combinaison des clés de partitionnement qui réduit le coût d'exécution d'une charge de requêtes complexes. Comb itère entre les phases partitionnement des attributs et regroupement des relations d'IR pour sélectionner le meilleur schéma de placement.

Travaux de Stöhr et *al* .

Stöhr et al [134] proposent une approche de construction et d'exploitation d'un entrepôt de données sur une machine parallèle ayant d disques. L'entrepôt considéré est modélisé par un schéma en étoile caractérisé par une table de faits volumineuse et un nombre de tables de dimension de petite taille. Ils proposent une approche de fragmentation qui décompose la table des faits en utilisant une méthode de partitionnement appelée Fragmentation Hiérarchique MultiDimensionnelle (MDHF). Elle consiste à fragmenter la table de faits en utilisant plusieurs attributs de tables de dimension. Chaque table de dimension est fragmentée en utilisant le mode intervalle (Range Partitionning) sur des attributs appartenant à des niveaux plus bas de la hiérarchie. Les tables dimensions et leur index (B*- arbre) sont stockées sur un seul disque de la machine parallèle à disque partagé. Pour accélérer les requêtes, des index de jointure en étoile bitmaps sont définis entre les tables de dimension et la table des faits sur des attributs appartenant à des niveaux plus haut de la hiérarchie. Le processus d'allocation ne concerne alors que les fragments de la table des faits et les fragments des index de jointure bitmaps définis. Notons que le nombre de fragments générés N est largement supérieur au nombre de disques d. Pour soutenir un degré de parallélisme élevé et un bon équilibrage de la charge, une allocation circulaire des fragments de la table des faits sur les disques est utilisée. Les index de jointure binaires définis sur les mêmes fragments sont placés consécutivement sur les nœuds afin de permettre un parallélisme intra-requêtes. Par exemple, si le fragment $frag_i$ de la table des faits est placé sur le nœud j, tous les k index de jointure qui lui sont associés sont placés sur les nœuds $j, j+1, \ldots, j+k-1 modulo d$

Les auteurs utilisent un parallélisme intra-requête où la requête est exécutée sur chaque

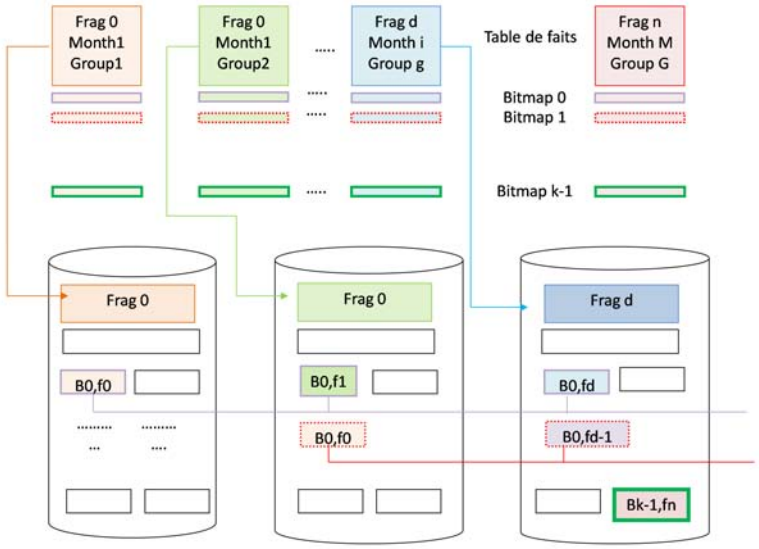


Figure 2.12 – *Principe de l'approche MDFH.*

fragment impliqué. Le traitement de la requête exploite les fragments de l'index de jointure en étoile bitmap pour diminuer le coût des entrées sorties. La fragmentation permet seulement l'identification du nombre maximal des sous-requêtes. Pour cela, les auteurs proposent une stratégie d'équilibrage de la charge pour atteindre un bon degré de parallélisme. Le traitement parallèle d'une requête en étoile Qi suit les étapes suivantes :

- déterminer les fragments faits à traiter en se basant sur les attributs de la requête Q_i et les attributs de fragmentation de la table des faits,
- déterminer pour chaque attribut de requête Q_i, tous les index de jointure bitmap associés. (l'accès aux index de jointure en étoile bitmap est nécessaire pour un q_i, si et seulement si "la dimension référencée dansQ_i n'est pas représentée dans le fragment F", ou la dimension référencée dans Q_i est représentée dans le fragment F, mais fait référence à un niveau plus élevé dans l' hiérarchie);
- assigner à chaque sous requête de Qi un fragment fait ainsi que les fragments des index de jointure bitmaps associés,
- pour chaque sous requête choisie pour exécuter la requête Q_i:
  1. sélectionner et traiter l'ensemble des pages pertinentes pour les fragments des index de jointure bitmap pour déterminer les identifiants des lignes RowID,
  2. sélectionner les pages faits contenant les RowID et exécuter l'agrégation,
  3. Réitérer les étapes a et b jusqu'à ce que toutes les pages du fragment soient traitées.

MDHF permet non seulement de limiter le nombre de fragments nécessaires pour le traitement des requêtes qui référencient l'attribut de fragmentation, mais également d'effectuer de nombreux autres types de requête en utilisant la structure hiérarchique des dimensions. *Stöhr et al* ont distingué quatre types de requêtes.

Q_1*: les requêtes définies sur les attributs de fragmentation.* Les requêtes qui font référence à

tous les attributs de fragmentation nécessitent généralement le chargement d'un seul fragment. Autrement dit, si Q_1 est en matching total avec les attributs de fragmentation, seulement un fragment sera chargé pour l'exécution de Q_1. En conséquent, les index de jointure bitmap ne seront pas utilisés car chaque tuple du fragment chargé est pertinent pour l'exécution de la requête Q_1. Par contre, si la requête se rapporte à un sous-ensemble des attributs de fragmentation, le nombre de fragments sera relativement réduit et les index de jointure bitmap définis sur les attributs qui n'appartiennent à aucune dimension fragmentée seront utilisés.

Q_2: les requêtes définies sur des attributs appartenant à un niveau bas dans la hiérarchie de l'attribut de fragmentation. Elles peuvent également bénéficier de schéma de fragmentation. En fait, chaque valeur d'un attribut appartenant à un bas niveau dans la hiérarchie correspond exactement à une valeur de l'attribut fragmentation. Si la requête référence toutes les dimensions fragmentées, elle nécessite le chargement d'un seul fragment pour son exécution, sinon plusieurs fragments seront chargés et les index de jointure bitmap seront utilisés, car les tuples de chaque fragment ne sont pas tous valides pour l'exécution de Q_2

Q_3: les requêtes définies sur des attributs appartenant à un niveau haut dans la hiérarchie de l'attribut de fragmentation. Elles peuvent également bénéficier du schéma de fragmentation. En effet, le nombre de fragments à charger sera plus grand que dans les cas précédent, car chaque valeur de l'attribut possède plusieurs valeurs associées de l'attribut fragmentation. Ainsi, le nombre de fragments augmente si, et seulement si, certaines dimensions de fragmentation sont impliquées.

Q_4: les requêtes définies sur des dimensions fragmentées. Elles vont également bénéficier du schéma de partitionnement comme dans le cas Q_2 et Q_3. Ainsi, toutes les requêtes référençant au moins un attribut d'une table de dimension fragmentée bénéficient de la fragmentation par la réduction du nombre des fragments à traiter et les index de jointure bitmap seront utilisés.

Travaux de Rao et *al* .

Rao et al s'intéressent à l'automatisation du processus de sélection du schéma de partitionnement des données (tables et vues) sur la machine shared nothing $IBMDB2$ [123]. L'objectif est de déterminer automatique le meilleur schéma de partitionnement de la base de données sur les nœuds de la machine parallèle pour optimiser une charge de requêtes données. Cette approche se base sur l'optimiseur de requêtes de $DB2$ pour évaluer la qualité du schéma de partitionnement.

Deux modes ont été intégrés à l'optimiseur : $RECOMMANDER$ et $EVALUATE$. En mode $RECOMMANDER$, l'optimiseur génère une liste des partitions pour chaque table qui est potentiellement bénéfique au traitement de la charge de requêtes. Des plans d'exécution sont générés selon le partitionnement recommandé. Ensuite, l'optimiseur évalue l'ensemble des plans alternatifs générés et il ajoute un plan qu'il juge optimal pour la requête à sa table CANDIDATE_PARTITION. En mode $EVALUATE$, l'optimiseur lit d'abord les plans stockés dans la table $CANDIDATE\_PARTITION$ et il les utilise pour remplacer la partition réelle pour la table correspondante. Après cela, l'optimiseur optimise la requête, en supposant que les tables sont partitionnées dans la nouvelle manière spécifiée. Quand une requête est optimisée, le plan de requête est généré sans être exécuté.

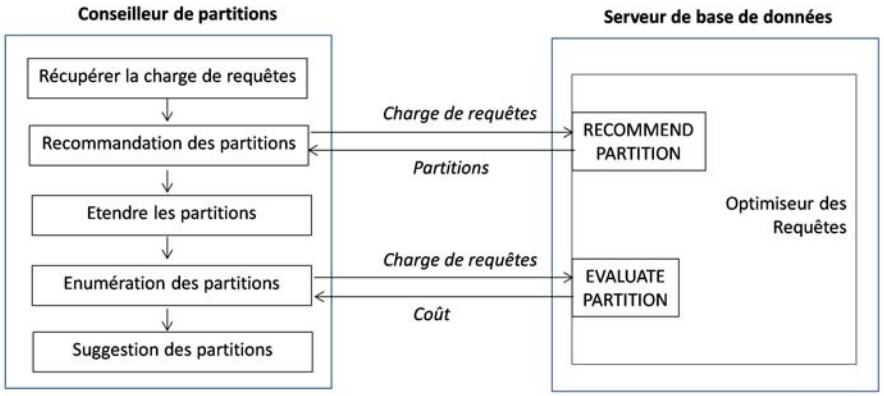


Figure 2.13 – *Architecture de l'Advisor de partitionnement proposé par Rao.*

Travaux de Taniar .

Taniar [136] présente une taxonomie des schémas d'indexation dans des systèmes de bases de données parallèles. Le partitionnement d'index n'a pas eu la même attention de la part de la communauté de partitionnement, que les tables de données, du fait que la plupart des structures d'index sont des arbres(contrairement aux tables qui ont une structure plate). En conséquent, le partitionnement d'index impose un certain degré de complexité par rapport au partitionnement des tables des données. Trois schémas d'indexation parallèles, le schéma d'Indexation Non Répliqué (NRI), le système d'Indexation Partiellement Répliqué (PRI) et le schéma d'Indexation entièrement Répliqué (FRI). Les deux schémas NRI et PRI ont trois variantes différentes, l'attribut index est également l'attribut de partitionnement de données, l'index local est construit à partir de ses données locales, et l'attribut de partitionnement d'index est différent de l'attribut de partitionnement des données (le partitionnement de données peut être différent de l'attribut indexé). Pour le schéma FRI, seules deux variantes sont connues, qui sont la première et la troisième variante de ce qui précède. Les auteurs discutent les stratégies de maintenance et analysent le besoin de stockage. Cette classification donne une possibilité complète de l'indexation dans les systèmes de bases de données parallèles, Oracle a adapté ces techniques d'indexation.

Travaux de Nehme et *al* .

Nehme et al [110] ont proposé une approche qui recommande la meilleure configuration de partitionnement pour une requête donnée. La configuration recommandée spécifie les relations qui doivent être répliquées et celles qui seront partitionnées selon une ou plusieurs colonnes, de sorte que le coût d'évaluation de la charge de requêtes est minimisé.

Pour assurer une recommandation plus précise dans un court laps de temps, l'approche est profondément intégrée à l'optimiseur de requêtes en parallèle. Elle utilise le mode d'optimisation $what - if$ [38]. Plus précisément, l'algorithme proposé nommé *Memo-Based Search Algorithm (MESA)* passe par quatre étapes.

– *Construction de la structure de données "Workload MEMO"*. Elle est l'union des MEMO

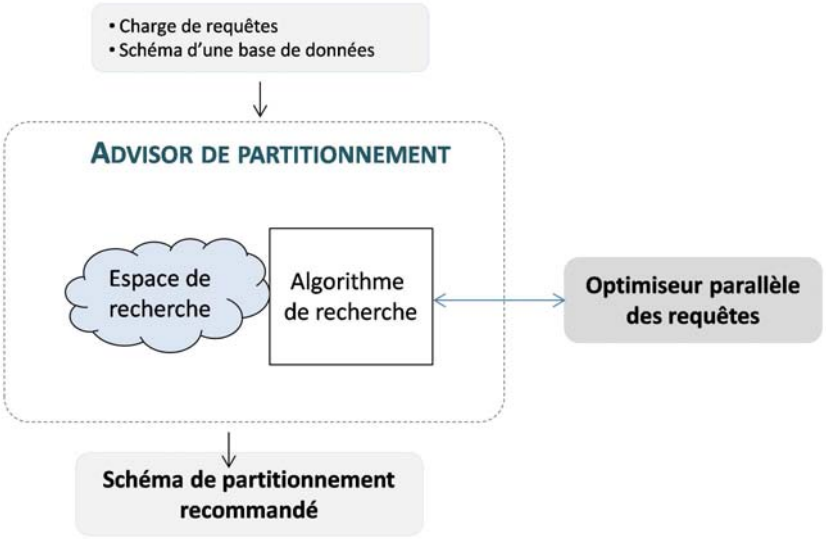


Figure 2.14 – *Architecture de l'Advisor de partitionnement proposé par Nehme.*

de chaque requête dans la charge de travail. La structure MEMO fournit une représentation compacte de l'espace de recherche de plans. Pour former cette structure, il faut générer d'abord une MEMO individuelle pour chaque requête dans la charge de travail; Ensuite, il faut relier les MEMO générées avec la racine en utilisant des arêtes attachant le nœud à la racine relier les MEMO avec les nœuds feuilles qui fusionne les table de base de données interrogées par les requêtes de la charge de travail

- *Sélection des colonnes candidates pour le partitionnement* : les colonnes référencées dans la clause de jointure ou dans la clause *group by* d'une requête sont recommandées par l'optimiseur comme des colonnes candidates au partitionnement.
- *Le *-partitionnement* signifie que toutes les options de partitionnement ou de réplication d'une table sont disponibles simultanément. Si une table est *-partitionnée, l'optimiseur choisit la colonne de partitionnement qui convient le mieux pour toutes les requêtes. Si la taille de table est inférieure à la limite de stockage, l'optimiseur peut aussi considérer la réplication. De cette façon, l'optimiseur considère simultanément toutes les alternatives possibles de *-partitionnement des tables partitionnées au cours d'une seule étape de post-traitement, et renvoie les plans d'exécution avec le plus petit coût global. Tous les plans qui en résultent ne sont pas valables lors de l'utilisation du *-partitionnement des tables. Plus précisément, une table donnée doit être physiquement partitionnée de façon unique.
- L'algorithme *Banch and Bound* est paramétré ainsi.
  Nœud. Chaque nœud représente une solution partielle ou complète. La solution est codée comme un tableau de d cellules dont chacune spécifie si la table est partitionnée (P) ou répliquée (R) ou *-partitionnement.
  Feuille. Une solution complète dans laquelle aucune table n'est autorisée à être *-partitionné.
  Fonction d'évaluation. Les solutions sont évaluées selon une fonction qui calcule le coût d'exécution correspondant à la charge de requêtes.

A chaque itération, l'algorithme sélectionne la feuille ayant le coût minimale et énumère toutes les solutions possibles. Pour accélérer la stratégie d'énumération, deux stratégies d'élagage ont été proposées. Dans le premier cas, aucun nœud descendant ne sera possible. Plus précisément, si l'espace total utilisé pour la réplication est supérieure à la contrainte de stockage. Dans le second cas, aucun descendant ne sera optimal. S'il n'y a pas une amélioration dans le coût de traitement de la charge de requête. Lorsqu'un nœud est choisi pour l'expansion, il faut choisir s'il sera *-partitionné, partitionné ou répliqué.

Travaux de Pavlo et *al* .

Ce travail a été fait en collaboration avec *Yahoo! Research*. *Pavlo et al* [120] ont proposé un l'outil d'aide à la sélection automatique d'une conception physique, ils l'ont nommé *Horticulture*. Il englobe le schéma de partitionnement, de réplication ainsi que le placement des procédures stockées. L'objectif est de minimiser le nombre de transactions distribuées dans le système, tout en réduisant les effets de l'asymétrie temporelle. Horticulture suit la procédure décrite ci-dessous pour sélectionner la meilleure configuration physique.

- *Conception initiale*. Horticulture utilise une heuristique pour générer une borne supérieure à la solution optimale. Elle comporte quatre étapes.
  - pour chaque table, sélectionner la colonne la plus fréquemment consultée dans la charge de travail comme attribut partitionnement horizontal;
  - identifier les tables à lecture seule qui peuvent être répliquées en respectant la contrainte de stockage;
  - pour les tables non répliquées, sélectionner les colonnes en lecture seule, accessibles le plus souvent des prédicats requêtes et qui ne sont pas des attributs de partitionnement pour les index secondaires et ne violent pas la contrainte de stockage;
  - Sélectionner les paramètres de routage pour les procédures stockées en fonction des paramètres référencés dans les prédicats des requêtes qui utilisent les colonnes de partitionnement de table sélectionnées dans l'étape 1.
- *Relaxation*. Cette étape permet la sélection aléatoire des tables dans la base de données et réinitialise leur attribut de partitionnement sélectionné en utilisant la relaxation. Cela permet d'éviter la sélection d'un minimum local. En effet, la génération d'une nouvelle configuration de conception doit d'abord spécifier le nombre et les tables à relaxer ainsi que les options de conception.
- *La recherche locale*. *Horticulture* exécute un algorithme de recherche en deux phases de manière itérative pour explorer les solutions. Ce processus est représenté par un arbre de recherche dans lequel chaque niveau coïncide avec l'un des éléments de base de données relaxés. Les niveaux de l'arbre de recherche sont divisés en deux sections dont chacune correspond à une phase de recherche. Dans la première phase, l'Horticulture explore les attributs candidats en utilisant l'algorithme Branch-and-Bound. Une fois que toutes les tables relaxées sont assignées, il effectue une recherche par force brute dans la deuxième phase pour sélectionner les paramètres d'allocation des procédures stockées. Chaque configuration est évaluée par un modèle de coût mathématique qui estime le coût d'exécution d'un échantillon d'une charge de requête. Le modèle de coût prend en considération le coût de communication ainsi que le facteur de skew.

2.1.2.3 Problème de la fragmentation verticale

Le problème de fragmentation verticale est fondamentalement plus complexe que le problème de fragmentation horizontale, cela en raison de la taille de son espace de recherche [118]. En effet, une relation de m attributs peut être fragmentée en m^m fragments.

Le problème de la fragmentation verticale consiste à déterminer comment partitionner une relation en fragments, dans le but de maximiser la performance du système. La sélection des fragments optimaux d'une relation minimise le nombre des entrées/sorties et favorise le parallélisme [60].

La fragmentation verticale dépend elle-aussi de la plateforme de déploiement. Dans cette section, nous présentons les principaux travaux de fragmentation verticale proposés dans le contexte centralisé, distribué et parallèle:

2.1.2.3.1 Travaux existants

2.1.2.3.1.1 Travaux existants dans le contexte centralisé

Travaux de Navathe et Ra .

Les auteurs ont proposé un algorithme de partitionnement binaire [107] et une approche de partitionnement graphique [109]. L'approche est basée sur le concept d'affinité qui désigne la fréquence des requêtes. La méthode exploite des matrices spécifiques (matrice d'usage et matrice des affinités) pour regrouper les prédicats de sélection selon les fréquences des requêtes qui les utilisent. Un groupe est identifié par un cycle (ensemble de prédicats) et désigne un fragment de dimension.

L'algorithme d'affinité part d'une table $T\{PK, A_1, A_2, \ldots, A_n\}$ et d'un ensemble de requêtes $\mathcal{Q} = \{Q_1, Q_2, \ldots, Q_L\}$ les plus fréquemment utilisées et leurs fréquences d'accès. L'ensemble des requêtes et de leurs fréquences d'accès est déterminé par l'administrateur.

La première approche suppose une relation ayant m attributs à fragmenter et un ensemble de n requêtes les plus fréquentes. Elle s'exécute en deux phases. Pendant la première phase, trois matrices sont construites.

- **Une matrice d'usage des attributs** dans laquelle la valeur de l'élément de la ligne i et la colonne j a pour valeur 1 si l'attribut j est accédé par la requête i (sinon cette valeur est nulle), est complétée par une colonne supplémentaire qui sauvegarde la fréquence d'accès pour chacune des requêtes.
- **Une matrice d'affinités d'attributs** contient m lignes et m colonnes correspondant aux attributs de la relation à fragmenter. La valeur de la cellule (i, j) affiche les valeurs d'affinités définies entre les attributs Cette affinité correspond à la somme des fréquences d'accès des requêtes accédant simultanément aux deux attributs.
- **Une matrice d'affinité ordonnée** est obtenue par l'application de l'algorithme de B.E.A (Bond Energy Algorithm) sur la matrice des affinités d'attributs. Par permutation

des lignes et des colonnes, l'algorithme B.E.A regroupe les attributs qui sont utilisés simultanément en fournissant une matrice sous la forme d'un semi-bloc diagonal.

La deuxième phase consiste à effectuer un partitionnement binaire récursif de la matrice d'affinités ordonné à la recherche des fragments verticaux qui optimisent une fonction objectif.

Pour la seconde approche, Navathe et *al* [109] ont présenté une méthode de regroupement des attributs basés sur les graphes. Cet algorithme construit à partir de la matrice des affinités d'attributs un graphe complet. Les nœuds de ce graphe représentent les attributs de la matrice d'affinités, et une arête, entre deux attributs, représenté la valeur d'affinité. Il cherche à générer des cycles dont les arêtes possèdent des grandes valeurs d'affinités.

Travaux de Gorla et Betty. Les auteurs exploitent les règles d'association pour la fragmentation verticale d'une base de données relationnelle [70]. Ils affirment que les performances du système peuvent être optimisées en réduisant le nombre d'accès aux données utiles pour une requête. Ils valident leur proposition avec deux bases de données réelles en faisant varier les seuils de support et de confiance.

2.1.2.3.1.2 Travaux existants dans le contexte distribué

Travaux de Bellatreche *et al* .

Les auteurs proposent une approche de fragmentation verticale pour partitionner une base de données objet distribuée [20]. Bellatreche et ses collègues proposent une approche top-down où l'entité de fragmentation est la classe. Ils présentent un algorithme de fragmentation verticale dans un modèle constitué par des attributs complexes et des méthodes complexes. Ce type de fragmentation facilite la décomposition de la requête, l'optimisation, et le traitement parallèle pour les systèmes de bases de données orientées objet distribués.

2.1.2.3.1.3 Travaux existants dans le contexte parallèle

Travaux de Datta *et al* .

Les auteurs ont proposé une nouvelle stratégie de traitement parallèle des requêtes de jointure [54]. Ce traitement parallèle est basé sur la fragmentation verticale. La fragmentation verticale a été utilisée comme une structure d'indexation. Chaque fragment représente une structure spéciale nommé Data Index (BDI). Pour accélérer les opérations de jointure, une extension du BDI nommée JDI est proposée. JDI est composé de BDI et une liste indiquant les enregistrements correspondants dans la table de dimension correspondante.

2.1.2.4 Problème de la fragmentation mixte

Le problème de la fragmentation mixte consiste à partitionner verticalement des fragments horizontaux ou à partitionner horizontalement des fragments verticaux. Peu de travaux se sont intéresé à ce problème.

Les algorithmes de fragmentation mixte ont été étudiés dans le contexte centralisé et distribué. Dans ce qui suit, nous présentons quelques-uns.

2.1.2.4.1 Travaux existants

Travaux existants dans le contexte centralisée

2.1.2.4.1.1 Travaux de Cheng *et al* .

Les auteurs étudient l'utilisation d'un algorithme de classification basée sur la recherche génétique pour le partitionnement de données [40]. Ils formalisent le problème comme un problème de voyageurs de commerce. Ils proposent un algorithme génétique pour sélectionner le schéma de fragmentation vertical ou horizontal. L'originalité de l'algorithme génétique proposé est l'utilisation de deux nouveaux opérateurs nommés respectivement: *Shortest Edge* et *Shortest Path* L'algorithme i a été initialement conçu pour le fragmentation verticale puis il a été prouvé qui'il facilement applicable au problème de la fragmentation horizontale

Travaux existants dans le contexte distribuée

Travaux de Navathe *et al* .

Les auteurs proposent une méthode de fragmentation mixte pour la conception initiale d'une base de données distribuée [108]. Ils appliquent la fragmentation horizontale et verticale simultanément en utilisant un seul algorithme. Le schéma de fragmentation mixte est représenté par une grille de cellules où chacune est définie par un prédicat de sélection et un prédicat de projection. Certaines cellules sont fusionnées pour réduire les accès disques.

2.1.2.5 Bilan et discussion

.

Nous avons vu l'importance de la fragmentation dans la conception des bases/entrepôts de données. Elle permet d'optimiser les requêtes et de faciliter la gestion des données selon le principe "diviser pour mieux gérer". Cette fragmentation a été déployée sur divers plateformes : centralisée, distribuée et parallèle.

Dans la littérature, la fragmentation a été largement étudiée. Ces approches différent dans le type de fragmentation appliqué (horizontale, verticale et mixte), l'élément fragmenté (base de données, entrepôt de données, index, ...etc.), les algorithmes utilisés pour la sélection du schéma de fragmentation (glouton, méta-heuristique, data mining, ...etc.) ainsi que la plateforme de déploiement (centralisée, distribuée et parallèle). Il est à noter que la majorité des approches utilisent un modèle de coût dédié pour évaluer la qualité du schéma de fragmentation retenu pour être déployé sur une plateforme précise. La fragmentation horizontale est favorisée, elle a été déployée sur les plateformes distribuées et parallèles.

Le tableau suivant résume les travaux présentés dans cette section. Pour chaque travail, il est mentionné l'unité fragmentée, l'algorithme de sélection et la plateforme de déploiement.

Type	**plateforme**	**Travaux**	**Unité fragmentée**	**Algorithme**
Horizontale	Centralisée	Bellatreche [6]	Entrepôt de données	Glouton+Graphe
		Boukhalfa *et al* [32]	Entrepôt de données	Méta-heuristiques
		Mahboubi *et al* [50]	Entrepôt de données XML	Data Mining
	Distribuée	Noaman et Barker [111]	Entrepôt de données	Glouton+Graphe
		Darabant et *al* [52]	base de données objets	Data Mining
	Parallèle	Zilio et *al* [152]	Base de données	Glouton+Graphe
		Stöhr et *al* [134]	Entrepôt de données	Glouton
		Rao et *al* [123]	Base de données	Glouton
		Taniar [136]	Index	Graphe
		Nehme et *al* [110]	Base de données	Génétique
		Pavlo et *al* [120]	Base de données	Branch-and-Bound+relaxation
Verticale	Centralisée	Navathe *et Ra* [107, 109]	Base de données	Affinité
		Gorla et Betty [70]	Base de données	Glouton+Graphe
	Distribuée	Bellatreche *et al* [20]	Base de données objet	affinité
	Parallèle	Datta et *et al* [54]	Entrepôt de données	Glouton
Mixte	Centralisée	Cheng et *al.* [40]	Base de données	Data Mining
	Distribuée	Navathe *et al*[108]	Base de données objet	Affinité

Tableau 2.1 – *Synthèse de comparaison entre les travaux de fragmentation*

2.1.3 Allocation de données

C'est une technique importante pour atteindre la haute performance des systèmes parallèles. D'une manière générale, l'allocation d'un ensemble d'objets $\{O_1, O_2, \ldots O_n\}$ est un problème important qui prouve la performance des applications exécutées dans un environnement distribué ou parallèle. Les objets à allouer peuvent être soit des données (tables, vues, fragments) soit des requêtes. Ces objets ont besoin d'être alloués d'une manière judicieuse sur les nœuds pour atteindre la haute performance d'un système. La génération et l'inspection de toutes les configurations possibles dans un espace à grande échelle est un problème NP-Complet d'ordre $2^{|O|}$ [5].

Pour réduire la complexité de ce problème, une panoplie de travaux de recherche a été proposée. La majorité des approches sont statiques (basés sur une charge de requêtes). Elles différent précisément dans le type d'algorithme utilisé pour sélectionner la meilleure configuration qui minimise le coût d'exécution d'une charge de requêtes. Dans ce qui suit, nous passons en revue les principaux travaux qui ont été proposées dans la littérature.

2.1.3.1 Travaux existants

Dans la littérature, les travaux de placement de données peuvent être scindés en deux grandes classes : (1) les stratégies déployés sur les plateformes parallèle comme le round-robin, hash placement et le range placement, et (2) les stratégies de placement axées sur les attributs déployées sur les environements distribués. Dans ce qui suit, nous décrivons les principaux travaux qui ont étudié cette question.

2.1.3.1.1 Travaux existants dans le contexte parallèle .

Le travaux proposés dans le contexte parallèle se basent sur les trois stratégies basiques de placement des données(round-robin, par hachage et par intervalle) ainsi que leurs variantes.

2.1.3.1.1.1 Placement circulaire (round-robin) .

Cette méthode round-robin place les tuples en fonction de leur numéro d'ordre. La première donnée sera placée sur le premier processeur. La seconde sur le deuxième et ainsi de suite jusqu'à placer la nouvelle donnée sur le dernier processeur. D'une manière générale, le *i ième* tuple de la relation est alloué sur le disque numéro $i \ mod \ n$ où n représente le nombre de disques. L'intérêt majeur de cette méthode est la distribution uniforme des tuples. Ainsi, elle garantit un bon parallélisme des entrées/sorties lors de la lecture des relations. Elle n'a cependant pas d'intérêt pour les relations intermédiaires (sauf si elles sont stockées sur disque).

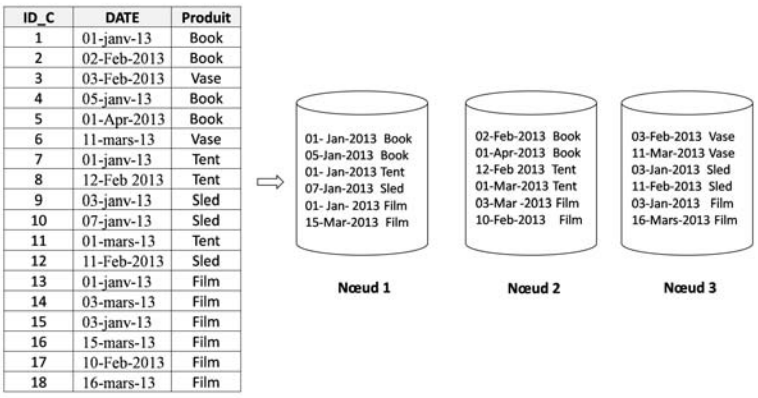


Figure 2.15 – *Exemple de placement circulaire.*

2.1.3.1.1.2 Placement par hachage (hash placement) .

La méthode de partitionnement par hachage, illustrée dans la figure 2.16, distribue l'ensemble des tuples en utilisant une fonction de hachage h sur un ensemble d'attributs. Cette fonction retourne le numéro du nœud dans lequel le tuple sera rangé. Cette méthode génère des fragments de taille différente. Elle convient idéalement aux applications qui donnent accès séquentiellement et associativement aux données. L'accès associatif aux tuples ayant une valeur d'attribut spécifique peut être dirigé vers un seul disque, évitant ainsi le surcoût dû à un lancement de requêtes sur plusieurs disques. Parmi les SGBDs commerciaux qui offrent cette méthode de partitionnement nous repérons Bubba [44], Gamma [56] et Teradata [137].

2.1.3.1.1.3 Placement par intervalle (range Partitionning) . Cette stratégie, illustrée dans la figure 2.17, 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. Les attributs formant la clé sont surnommés alors *attributs de partitionnement* ou c*lés de partitionnement*. La table partagée est dite *ordonnée*. Cette méthode comporte en deux phases :

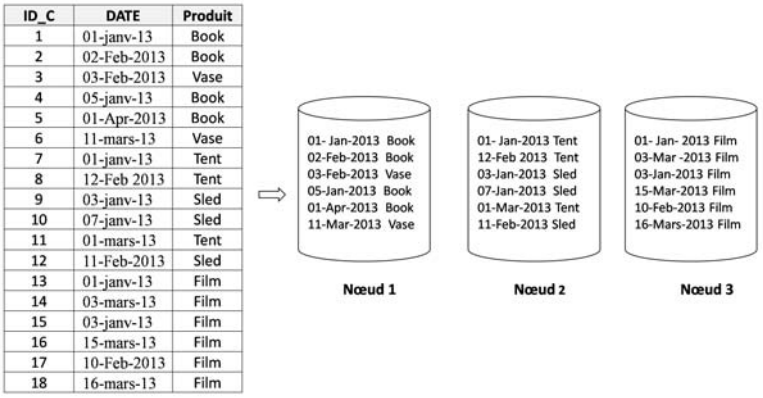


Figure 2.16 – *Exemple de placement par hachage.*

- la première phase permet de diviser l'espace des valeurs des attributs sélectionnés en intervalles, puis chaque intervalle est affecté à un nœud,
- la deuxième phase assigne chaque tuple t au nœud n si les valeurs d'attributs spécifiés du tuple t appartiennent à l'intervalle de n.

Cette technique convient aux requêtes dont le prédicat implique les attributs de partitionnement et donc les requêtes par intervalle. Cependant, le problème majeur est le risque d'avoir une allocation non uniforme des données (toutes les données placées dans une seule partition : problème data placement skew), et une exécution non uniforme dans laquelle toute l'exécution apparaît dans une seule partition, la mauvaise distribution des données ne garantissant pas un équilibre de charge entre les nœuds. Parmi les SGBDs commerciaux qui offrent cette méthode de partitionnement nous repérons Bubba [44], Gamma [56], Oracle [100] et Tandem [73].

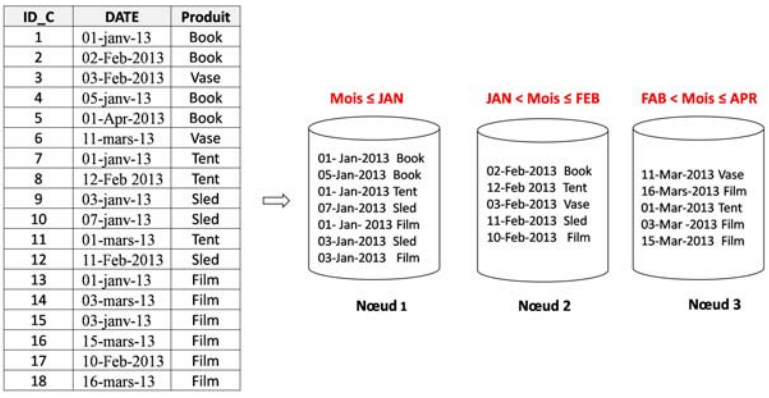


Figure 2.17 – *Exemple de placement par intervalle.*

2.1.3.1.1.4 Travaux de Furtado et *al* .

Furtado [64] propose une nouvelle stratégie basique de placement de données nommée *Node Partitioned Data Warehouses (NPDW)*. L'idée de base de la stratégie de placement de données proposée consiste à partitionner horizontalement la table des faits et les grosses tables de

dimension et à les allouer ensuite sur les nœuds en utilisant une distribution circulaire ou aléatoire. Les tables de petite taille sont répliquées sur tous les nœuds. Les tables de dimension les plus importantes sont fragmentées par hachage selon leurs clés primaires. La table des faits est à son tour partitionnée par hachage en utilisant les clés étrangères. Le choix des attributs de partitionnement est conditionné par la diminution du coût de traitement qui englobe l'ensemble des coûts suivants. [64]

- *Le coût de partitionnement* (Partitionning Cost) d'une relation consiste à trouver initialement la relation en mémoire secondaire ; diviser la relation en fragments par l'application d'une fonction de hachage sur l'attribut de jointure ;assigner les fragments aux nœuds. Le partitionnement implique le balayage total de la table. En conséquence, le coût de partitionnement est en croissance avec la taille de table.
- *Le coût de repartitionnement* (Repartitioning Cost) assure la réorganisation. Il applique d'abord une fonction de hachage sur un des différents attributs d'équijointure d'un fragment puis il redistribue les fragments résultants sur les autres nœuds pour traiter une jointure par hachage.
- *Le coût de communication des données* (Data Communication Cost) est en croissance avec la taille des données transférées entre les nœuds.
- *Le coût de traitement local* (Local Processing Cost) dépend typiquement du cas où la jointure est soutenue par une structure auxiliaire comme l'index et la taille des relations participantes à la jointure.
- *Le coût de fusion* (Merging Cost) est lié à l'application de la requête finale afin de rassembler les résultats partiels au niveau du nœud de fusion.

Il est à noter que cette fragmentation ne prend pas en considération les exigences de requêtes de jointure, comme les sélections sur les tables de dimension et les jointures entre la table des faits et les tables de dimension.

2.1.3.1.2 Travaux existants dans le contexte distribué Dans la littérature, plusieurs solutions ont été proposées dans le contexte distribué. Dans ce qui suit, nous citons quelques-uns.

2.1.3.1.2.1 Travaux de Huang et Chen .

Les auteurs ont proposé deux heuristiques [63], à base d'un modèle simple et complet qui reflète le comportement de la transaction dans une base de données distribuée, pour définir une allocation quasi optimale telle que le coût de la communication est minimisé. Le premier algorithme comprend trois étapes. Il construit tout d'abord la table d'allocation des fragments depuis la Matrice de Recherche des fragments RM et la matrice des fréquences $Freq$. Au niveau de la deuxième étape, les auteurs cherchent un compromis entre le coût de mise à jour de la requête et le coût du déplacement de la copie du fragment. Cette étape est répétée jusqu'à ce qu'il y ait plus de bénéfice, sauf si la copie en cours est la seule dans le réseau. La dernière étape cherche si un fragment n'est pas encore alloué, ce qui lance un petit traitement pour le choix d'un site parmi les candidats pour ce fragment. Le deuxième algorithme est constitué également de trois étapes. La première étape est identique à celle du premier algorithme. Par contre en deuxième étape, les entrées de la table d'allocation initiale sont analysées et pondérées. Le

poids de chaque entrée est défini comme étant le montant de données sauvé qui est accédé par les requêtes de mise à jour quand une copie du fragment est enlevée d'un site. Ce poids est calculé depuis la matrice des fréquences $Freq$, la matrice de mise à jour UM, la matrice de sélectivité SEL et la taille du fragment (F_j). A chaque fois, l'algorithme sélectionne le plus grand poids des entrées non explorées et il vérifie si le bénéfice est supérieur au coût. La copie du fragment est alors déplacée du site à moins que cette copie soit la seule dans le réseau. L'étape 2 est répétée jusqu'à ce que toutes les entrées soient parcourues. Les deux algorithmes ont la même complexité $O(nm2q)$ où n, m et q représentent le nombre de fragments, le nombre de sites et le nombre de requêtes respectivement. Les résultats de l'expérimentation montrent que dans les deux algorithmes l'heuristique est optimale, mais le premier algorithme est plus efficace que le deuxième, dans la plupart des cas.

2.1.3.1.2.2 Corcoran *et al* .

Les auteurs utilisent un algorithme génétique pour résoudre le problème d'allocation de données formalisé comme un problème combinatoire [45]. Ils définissent une fonction objective pour mesurer le coût de la transmission entre les sites résultant de chaque solution. Une pénalité est appliquée sur la solution qui ne satisfont pas la contrainte de stockage

2.1.3.1.2.3 Travaux de Ahmad *et al* .

Les auteurs proposent une approche basée sur les graphes de dépendance entre fragments de façon à réduire les coûts de transfert de données [1]. Ils suggèrent l'utilisation des algorithmes évolutionnaires comme l'algorithme génétique, l'évolution simulée (Simulated Evolution), le recuit moyen des champs (Mean Field Annealing) et la recherche aléatoire du voisinage (Random neighbourhood Search) pour la sélection de la meilleure configuration de l'allocation. L'objectif des algorithmes est de minimiser le coût d'exécution d'une charge de requêtes. L'algorithme SE s'avère le meilleur en qualité de solution et temps de calcul en générant plusieurs solutions optimales pour tous les pourcentages de coût; GA vient en deuxième position et MFA et RS génère relativement peu de solutions optimales ou proches de l'optimale. Par conséquent, les performances de RS et MFA se dégradent à de gros volume de données par contre SE et GA sont plus robustes. Dans le cas de la recherche exhaustive RS, est bien plus efficace en temps de traitement que SE et GA, et légèrement plus rapide que MFA. En termes de temps de réponse aux requêtes et temps d'exécution de l'algorithme, il n'y a aucun algorithme qui les satisfait en même temps. En général, RS a un temps de complexité réduit ce qui le rend rapide alors que SE est lent. L'algorithme RS est un bon choix du point de vue temps d'exécution et dégradation minimale de l'optimalité de la solution. D'une autre part, GA peut être un meilleur choix quand l'efficacité et la qualité de solution sont prises en compte en priorité.

2.1.3.1.2.4 Travaux de Rosa Karimi Adl *et al* .

Les auteurs [88] utilisent l'optimisation par colonie de fourmis pour résoudre le problème d'allocation de données. L'objectif est de concevoir un schéma d'allocation des données efficace qui minimise le temps total de la charge de requêtes avec le respect de la contrainte de stockage des sites. Les tests expérimentaux montrent que l'algorithme proposé est capable de produire

des solutions quasi-optimales dans un délai raisonnable.

2.1.3.1.2.5 Travaux de Sarathy *et al* .

Les auteurs [126] considèrent le problème de l'allocation des fragments comme un modèle non linéaire entier à contraintes qui est prouvé NP-difficile. Pour résoudre le problème, ils développent un algorithme basé sur la linéarisation et l'optimisation sous-gradient, pour résoudre ce modèle. *Menon* [103] a étendu le modèle entier de [100] en incluant des contraintes sur la capacité de stockage et le traitement Dans le même temps, il a simplifié les formulations de la programmation en nombres entiers pour étudier la version non redondante du problème d'allocation de fragment. Tant SARATHY et *Menon* de simplifier le problème en considérant un coût de transport constant entre une paire de sites. Par conséquent, le problème de la minimisation du coût total de la transmission des données est le problème de minimisation de l'ensemble des données transmises.

2.1.3.1.2.6 Travaux de Hababeh *et al* .

Les auteurs présentent une méthode qui intègre les sites en groupe pour atteindre la haute performance [76]. L'objectif est de minimiser le nombre d'E/S et le coût de communication entre les sites. Une méthode de classification est utilisée, elle consiste à grouper les sites en cluster pout réduire le coût de communication entre les sites. Ainsi, la disponibilité et la fiabilité est atteintes car plusieurs copies des fragments sont allouées.

2.1.3.1.2.7 Travaux de Maik Thiele et al .

Les auteurs s'intéressent au problème d'allocation de données pour un entrepôt de données en temps réel [139]. Ils considèrent une charge de requêtes mixtes (requêtes de mise à jour et des requêtes de sélection). Les auteurs considèrent une classe de services à deux objectifs ; l'un basé sur la métrique qualité de service (QoS) (comme le throughput, le temps de réponse moyen et stretch) et l'autre basé sur la métrique qualité de données (QoD). Ainsi, ils formalisent le problème comme étant un problème multi-objectif. Les auteurs assimilent le problème d'allocation à problème de sac à dos avec des contraintes d'inégalités supplémentaires et ils l'ont résolu via un algorithme de programmation linéaire.

2.1.3.2 Bilan et discussion

.

En résumé, les travaux existants s'intéressent à la formalisation du problème d'allocation sans réplication ou à la sélection du meilleur schéma d'allocation. La majorité des approches proposées s'intéresse au placement des objets. Ces approches diffèrent précisément dans le type d'algorithme utilisé (algorithme glouton, dirigé par des méta-heuristiques, dirigé par des techniques de Data Mining). Généralement, un modèle de coût mathématique est utilisé pour sélectionner la meilleure configuration qui minimise le coût d'exécution d'une charge de requêtes. Il est à noter que chaque étude vise à minimiser un critère de la fonction objective global du problème d'allocation de données.

En analysant plus les travaux existants, nous constatons que l'allocation a lieu uniquement sur les architectures distribuées sans partage et ses variantes. Plus précisément, la majorité des travaux concernant les bases de données parallèles font recours aux stratégies (et leurs variantes) de placement usuelles citées dans la secion 2.1.3.1.1 et ils n'utilisent aucun modèle de coût pour mesurer la pertinence du schéma déployé.

Le tableau ci-dessous présente une classification des principaux travaux en se basant sur les critères que nous venons de citer.

Plateforme	Travaux	Problème étudié	Unité allouée	Algorithme
machine parallèle	Placement circulaire	Placement	Tuples	Glouton
	Placement par intervalle	Placement	Tuples	Glouton
	Placement par hachage	Placement	Tuples	Hachage
	Furtado [64]	Placement	Fragment	Glouton
distribuée	Huang et Chen [63]	Placement	Fragment	Glouton
	Corcoran et *al* [45]	Placement	Fragment	Génétique
	Ahmad et al [1]	Placement	Fragment	Graphes
	Rosa Karimi Adl *et al* [88]	Placement	Fragment	Colonie de fourmis
	Sarathy *et al* [126]	Formalisation	Fragment	Programation Linéaire
	Menon *et al* [103]	Formalisation	Fragment	Programation Linéaire
	Hababeh et al *et al* [76]	Placement	Fragment	Classification
	Maik Thiele et al [139]	Formulation+Placement	Fragment	Multi-objectifs+sac à dos

Tableau 2.2 – *Synthèse de comparaison entre les travaux d'allocation.*

La figure 2.18 donne une comparaison entre les travaux que nous avons étudiés. Cette comparaison porte sur le problème étudié, les algorithmes de sélection utilisés par chaque travail ainsi que la nature dee la plateforme de déploiement.

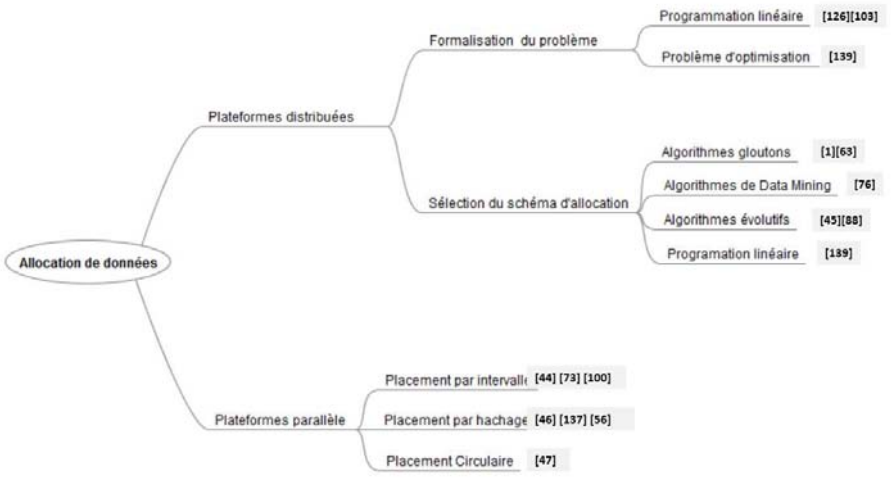


Figure 2.18 – *Classification du placement de données*

2.1.4 Réplication des fragments

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. La réplication consiste à placer plusieurs copies de l'objet sur différents nœuds. Elle fournit une grande disponibilité des données et garantit la tolérance aux pannes en dépit de la défaillance des nœuds. Elle améliore également les performances d'accès en augmentant la localité de référence.

La qualité du mécanisme de réplication est fortement affectée par le choix des paramètres et les stratégies suivants.

- *Sélection des répliques.* La sélection des données à répliquer dépend de la popularité et de l'importance des données, ceci peut être acquis en traçant l'historique des accès des utilisateurs.
- *Le nombre de répliques* est généralement fixé par l'administrateur selon la capacité de stockage et les accès à la bande passante.
- *Le placement des répliques.* Le problème consiste à trouver le meilleur emplacement en prenant en considération la capacité de stockage et la puissance de calcul des nœuds du système distribué. Comme le coût de communication est un facteur dominant, chaque copie doit être placée sur le nœud sur lequel y accède le plus souvent pour favoriser les accès locaux et éviter les accès réseaux.
- *la stratégie de rafraîchissement des répliques.* La mise-à-jour d'une copie doit être répercutée automatiquement sur toutes ses répliques. Ainsi, les stratégies de rafraîchissement de données définissent comment la propagation va être effectuée en précisant le contenu à propager, le modèle de communication utilisé, l'initiateur et le moment du déclenchement. Le rafraîchissement est fait grâce à une transaction appelée transaction de rafraîchissement dont le contenu peut être les données modifiées (writesets en anglais) ou le code de la transaction initiale.
- *Maintien de cohérence.* Les mises à jour peuvent être effectuées sur une seule réplique (appelée maître) avant d'être propagées vers les autres (esclaves). La cohérence peut être gérée de manière stricte (réplication synchrone) ou relâchée (réplication asynchrone) .

En effet, deux problèmes majeurs sont liés au problème de réplication de données : le problème de création des répliques et le problème de maintenance des répliques matérialisées. Définir une nouvelle approche pour maintenir la cohérence des données entre les répliques ne figure pas parmi les objectifs de cette thèse, nous définissons brièvement tous les éléments nécessaires à la mise en œuvre de la réplication des données. Nous présentons les différents modes de réplication existants et nous nous penchons ensuite sur les principaux travaux effectués pour résoudre le problème de placement des répliques.

2.1.4.1 Les modèles de réplication

Les techniques de réplication peuvent être classées selon deux caractéristiques : la stratégie de placement des répliques ou la stratégie de mise à jour des répliques [72]

Le placement des répliques sur le réseau affecte directement le mécanisme de contrôle de répliques.

- La configuration *maître-paresseux* (Lazy-master replication) est composée uniquement

d'une copie primaire et d'une ou plusieurs copies secondaires. Elle est nommée paresseuse. Dans un premier temps, seul le nœud contenant la copie primaire (nœud maître) se met à jour, puis dans un second temps, les copies secondaires sont rafraichies. Une telle configuration est recommandée pour les applications en lecture seule comme les entrepôts de données. Cependant, la disponibilité des données est limitée car dans le cas de panne du nœud maître la copie secondaire ne peut plus être mise à jour.

- *Réplication totale* consiste à stocker la réplique de chaque objet sur tous les nœuds. Cette stratégie de réplication fournit un bon équilibrage de charges et elle améliore également la disponibilité des données puisque n'importe quel nœud peut être remplacé par n'importe quel autre en cas de défaillance. Les transactions de mise à jour sont coûteuses car elles doivent s'exécuter sur tous les nœuds. En effet, lorsqu'une table R est mise à jour sur un nœud, toutes les autres copies ont besoin d'être rafraîchies.
- *Réplication partielle* permet à chaque nœud de contenir un sous-ensemble des répliques de telle sorte que les nœuds contiennent des sous-ensembles de répliques différents. Cette stratégie nécessite moins d'espace de stockage et réduit le temps de mise à jour. En effet, les transactions de mise à jour ne seront pas à être diffusées à tous les nœuds. Deux sortes de réplication partielle ont été distinguées.

  1. *Répliquer toutes les données sur quelques nœuds.* Le degré de réplication peut être dénoté par un paramètre $\nabla \in \{1, 2, ..., n\}$, ∇ signifie que chaque élément est représenté par ∇ répliques et n représente le nombre de nœud du système. $\nabla = 1$ signifie qu'il n' y a pas de réplication, $\nabla = n$ signifie la réplication totale des données et si $n \succ \nabla \succ 1$ alors chaque élément de données est répliqué ∇ fois. Plusieurs études ont adapté leur réplication partielle à cette définition [43, 148].
  2. *Quelques objets sont répliqués sur quelques nœuds.* Le degré de réplication peut être désigné par $\nabla \in [0, 1]$ qui représente le pourcentage d'objet de données entièrement répliqués sur tous les nœuds. Un élément de données est soit entièrement répliqué ou non répliqué. $\nabla = 0$ exprime qu'il n'y a pas de réplication, $\nabla = 1$ signifie la réplication totale et si $1 \succ \nabla \succ 0$ alors une portion de ∇ de chaque élément de données, est répliquée. Ce modèle de réplication partielle n'a été considérée que dans [3, 66].

Dans la **mise à jour**, l'un des défis majeurs est de maintenir la cohérence mutuelle des répliques, lorsque plusieurs d'entre elles sont mises à jour, simultanément, par des transactions. Gray et al [72] classifient les mécanismes de contrôle des répliques selon deux paramètres : le moment de rafraîchissement des répliques ("*quand ?*") et le degré de communication entre les serveurs ("*comment ?*"). Selon le premier critère, les stratégies de réplication peuvent être faites en mode synchrone ou asynchrone. Par contre, selon le deuxième critère de classification, la réplication peut être maître-esclave ou multi-maître.

- *Réplication synchrone (modèle multi-maître)* est aussi appelée "réplication en temps réel". Elle permet la récupération de données sans aucune perte. Au moment de la mise à jour, chaque mise à jour doit être reconnue et confirmée à la fois sur les sites primaires et secondaires; le rafraîchissement des copies peut ainsi s'effectuer en parallèle avec le nœud d'origine. De cette façon, le système garantit que toutes les copies sont toujours une image miroir exacte des données primaires.

L'avantage principal de la réplication synchrone est qu'elle assure que tous les données du système sont toujours à jour. Ainsi, les nœuds secondaires peuvent remplacer le nœud primaire en cas de défaillance. Ce type de réplication nécessite un matériel de stockage à haute performance, un logiciel de réplication de haute performance et un réseau de communication à haut débit. Elle est donc très coûteuse. Néanmoins, la réplication synchrone est appropriée pour les entreprises qui traitent des données critiques, ne peuvent se permettre aucun temps d'arrêt et nécessitent une protection à 100% pour récupérer leurs données.

– *Réplication asynchrone (Modèle maître-esclave)*, aussi appelée " stocker et propager " (Storeand Forward), elle consiste à mettre à jour la copie primaire et à stocker les opérations exécutées sur le nœud maître dans une queue locale pour les propager plus tard sur les nœuds qui stockent les copies secondaire à l'aide d'un processus de synchronisation. L'avantage majeur de la réplication asynchrone est qu'elle est applicable pour les réseaux larges et distants. Elle est plus flexible que la réplication synchrone à condition de définir les meilleurs intervalles de synchronisation. Par exemple, la synchronisation sera effectuée la nuit, à une heure de faible affluence. Cependant, la détermination d'un planning de réplication est une tâche complexe, puisqu'il s'agit de gérer les conflits émanant d'un éventuel accès en écriture sur une base esclave entre deux mises à jour.

2.1.4.2 Travaux existants.

Le besoin de la haute disponibilité des données signifie que la réplication est nécessaire. Des solutions matérielles comme les RAID ont été proposé. Dans ce qui suit, nous citons les principaux travaux qui s'intéressent à la réplication des fragments dans le contexte des bases de données parallèles. Les travaux proposés peuvent être classés en deux catégories. La première englobe les stratégies de réplication conventionnelles déployées sur les plateformes parallèles comme le Mirrored Declustering, le chained Declustering et le Interleaved Declustering; la deuxième se base sur des stratégies de placement pour allouer les répliques sur les environnements distribués.

2.1.4.2.1 Travaux existants dans le contexte parallèle Le travaux proposés dans le contexte parallèle se basent sur les trois stratégies: Mirrored Declustering , Chained Declustering, Interleaved declustering. Nous le détaillons dans ce qui suit.

2.1.4.2.1.1 Travaux de Borr: *Mirrored Declustering* .

Le principe de base utilisé dans le Mirrored Declustering (MD) [29] est de créer une deuxième copie des données et de la stocker sur un disque différent. Dans Mirrored Declustering, toutes les données d'un disque sont répliquées sur un autre ensemble des disques. Dans le cas où un disque tombe en panne, toute la charge du nœud est transférée vers le disque miroir qui prend le rôle du disque primaire. Comme illustré dans la figure 2.19, les données du disque i sont copiées sur un autre disque i'. Par exemple, si le nœud N_1 tombe en panne, sa charge de travail sera redirigée vers le nœud N_4.

En conséquence, MD offre un haut niveau de tolérance aux pannes, mais une mauvaise

répartition de la charge d'un nœud car le degré de déséquilibre des charges de travail entre les nœuds opérationnels augmente en cas de panne.

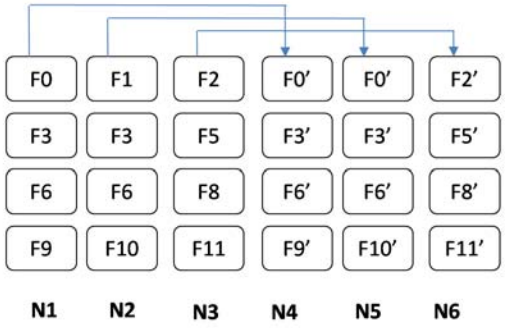


Figure 2.19 – *Mirrored Declustering*

2.1.4.2.1.2 Travaux de Hsioa et al : *Chained Declustering* .

Le Chained Declustering proposé par Hsioa et Dewitt [79] ne supporte que deux répliques des données. Pour n nœuds et une relation R partitionnée par intervalle en N fragments, chaque fragment est assigné à deux nœuds consécutifs de sorte qu'il a un fragment commun avec le nœud précédent et un autre commun avec le nœud suivant. Le premier nœud se chevauche avec le dernier nœud. La première copie d'un fragment est considérée comme la copie primaire et la seconde copie comme une réplique. En effet, les données sont non disponibles uniquement si deux nœuds adjacents sont en panne. Cependant, CD n'assure pas un bon équilibrage de charge parce que la charge du nœud défaillant est complètement assignée à ses deux nœuds adjacents. De plus, CD peut obtenir de meilleures performances en utilisant une stratégie d'équilibrage de charge dynamique. Le chaînage de données permet à chaque nœud de décharger une partie de sa charge à autre nœud qui contient des copies secondaires. Par le déchargement en cascade sur plusieurs nœuds, une charge uniforme peut être maintenue sur tous les autres nœuds

La figure 2.20 présente un exemple du CD. Les nœuds N_2 et N_6 sont chainés aux nœuds N_3 et N_1 respectivement, N_2 et N_6 peuvent décharger une partie de leur charge normale sur N_3 et N_1. De cette façon, le système assure une distribution uniforme de la charge de travail d'où l'équilibrage de charge du système.

2.1.4.2.1.3 Travaux de Teradata: *Interleaved declustering* .

C'est une stratégie proposée par le gestionnaire de base de données parallèle Teradata DBC/1012 [137]. La méthode ID consiste à diviser la copie secondaire de chaque relation en un ensemble de sous-partitions stockées dans des nœuds différents à l'exception du nœud qui contient la copie primaire. Chaque copie primaire du fragment est partitionnée en M partitions dénotées par F_ij où i, j représente la $j^{ième}$ partition de la copie primaire du fragment i. Quand un nœud tombe en panne, ID est capable d'assurer un meilleur équilibrage de charges que MD et CD, car la charge de travail du nœud défaillant est répartie entre les nœuds opérationnels. ID cherche à trouver un compromis entre l'équilibrage de charge et de la disponibilité des données

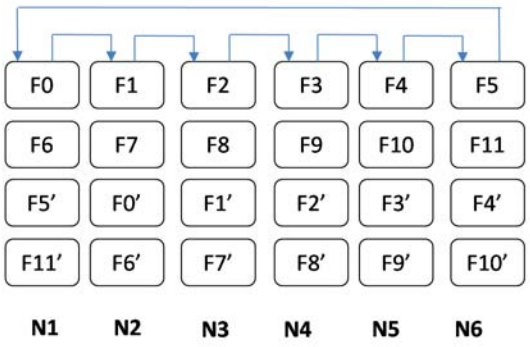


Figure 2.20 – *Chained Declustering*

du système. Pour un système distribué de 5 nœuds, la copie primaire du fragment F_1 alloué sur N_1 et partagé en 5 sous-partitions $F_{1.0}$,$F_{1.1}$, $F_{1.2}$, $F_{1.3}$ et $F_{1.4}$ allouées sur N_1, N_3, N_4, N_5 et N_6 respectivement. Quand le nœud N_1 est non disponible en raison de défaillance, sa charge est assignée aux autres nœuds. Toutefois, ID peut subir une dégradation des performances quand le nombre de nœuds est très grand et la taille des fragments petite. La figure 2.21 illustre un exemple

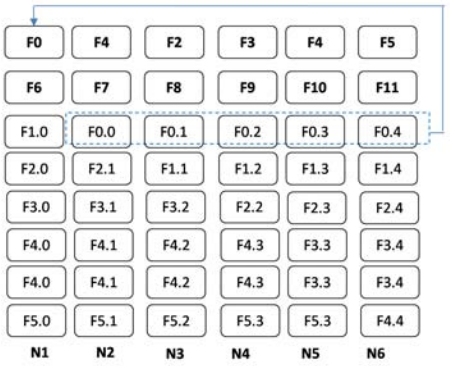


Figure 2.21 – *Interleaved declustering*

2.1.4.2.2 Travaux existants dans le contexte distribué

2.1.4.2.2.1 Travaux de Costa et *al* .

Les auteurs [48] s'intéressent à la conception d'un entrepôt de données parallèles dans un environnement de grille (*Grid-DWPA*). Pour cela, les auteurs définissent des paramètres de qualité de service (*QoS*) comme la priorité et le temps réservé pour le traitement des requêtes et ils définissent également des stratégies qui visent à exécuter une sous-requête sur un site. Dans le système *Grid-DWPA*, le placement de données est assuré par la stratégie $NPDW$ [64]. Cette stratégie ne fragmente que les relations volumineuses en se basant sur une charge de requêtes OLAP et réplique les petites relations sur chaque site de la grille. La réplication

des données est effectuée par une stratégie nommée *PRG (Partitioned Replica Groups)* qui consiste à dupliquer les fragments non volumineux sur les groupes de nœuds définis.

2.1.4.2.2.2 Travaux de Zhu et *al* .

Inspirés du *chained declustering*, ils proposent une nouvelle stratégie de réplication appelée *Shifted Declustering* [151]. *SD* consiste a augmenté la distance entre les deux répliques appartenant au même groupe de redondance jusqu'à ce que toutes les k unités appartenant au même groupe de redondance soient réparties sur tous les disques. La procédure de mise en place est identique à celle du chained declustering. Ainsi, SD élimine la limitation que les disques consécutifs transportent des données qui se chevauchent plus que les disques éloignés dans CD. Cela est dû à son système de mise en place qui fait appel à la distribution des répliques aux disques consécutifs. En revanche, les distances entre les disques contenant les répliques sont étendues, l'une par itération du nombre de groupes de redondance, afin de garantir que tous les disques restants partagent la charge de travail résultant d'un disque défectueux. Ainsi, *shifted declustering* assure la flexibilité dans le système pour le choix de n'importe quel disque disponible ainsi que le nombre de répliques.

2.1.4.2.2.3 Travaux de Chang *et al* .

Les auteurs proposent une approche de réplication de données dynamique sur une grille de données [36]. Le nœud de coordination est responsable de la gestion des répliques, il recueille les informations à partir des nœuds de traitement et il détermine les fichiers popularités selon fréquence d'accès. Le nombre optimal de répliques est calculé selon le seuil d'accès relatif de tous les fichiers au système.

2.1.4.2.2.4 Travaux de Foresiero *et al* .

Les auteurs proposent une heuristique inspirée du comportement des fourmis qui sont considérées comme des agents intelligents se déplaçant dans la grille et effectuant des répliques en se basant sur des fonctions de probabilité [61]. Cet algorithme repose sur l'utilisation combinée de nouveaux paradigmes distribués, à savoir, les techniques multi-agents et P2P.

2.1.4.3 Bilan et discussion

Les travaux cités dans cette section focalisent leur étude sur la localisation des répliques. La stratégie du placement des répliques sur les différentes unités de stockages peut avoir un grand impact sur la fiabilité du système et assurer une bonne tolérance aux pannes. Ces stratégies diffèrent principalement par leur unité de réplication, leur degré de réplication, leur algorithme de placement et leurs contraintes de la plateforme de déploiement.

Sur les plateformes parallèles, la majorité des approches proposées sont statiques. Les auteurs proposent des solutions génériques qui ne se basent ni sur une charge de requêtes, ni sur un modèle de coût. En conséquence, l'équilibrage de charge s'avère difficile à atteindre. Toutes les approches répliquent toutes les données sur quelques nœuds.

Sur les plateformes distribuées, un recours aux méta-heuristiques et aux techniques de data mining a été fait pour trouver le placement optimal des répliques ou identifier le nombre optimal des répliques (*) [4] Le tableau 2.3 synthétise les principaux travaux cités et les analysent selon les critères mentionnés ci-dessous.

Plateforme	Travaux	Unité répliquée	Algorithme	Redistribution
machine parallèle	Mirrored Declustering [29]	fragments	Glouton	Vers un seul nœud
	Chained Declustering [79]	fragments	Glouton	Pour quelques nœuds
	Interleaved declustering [137]	Partition du fragment	Glouton	Pour tous les nœuds
distribuée	Costa et Furtado [48]	fragments	Classification	Pour quelques nœuds
	Zhu [151]	Partition du fragment	Glouton	Pour quelques nœuds
	Chang *et al* [36]	fichier	Glouton	Pour quelques nœuds
	Forestiero *et al* [61]	fichier	Colonnie de fourmis	Pour quelques nœuds

Tableau 2.3 – Synthèse de comparaison entre les travaux de réplication

La figure 2.22 illustre une classification des travaux que nous avons étudiés. Cette classification porte sur le problème étudié, les algorithmes de sélection utilisés par chaque travail ainsi que la nature de la plateforme de déploiement.

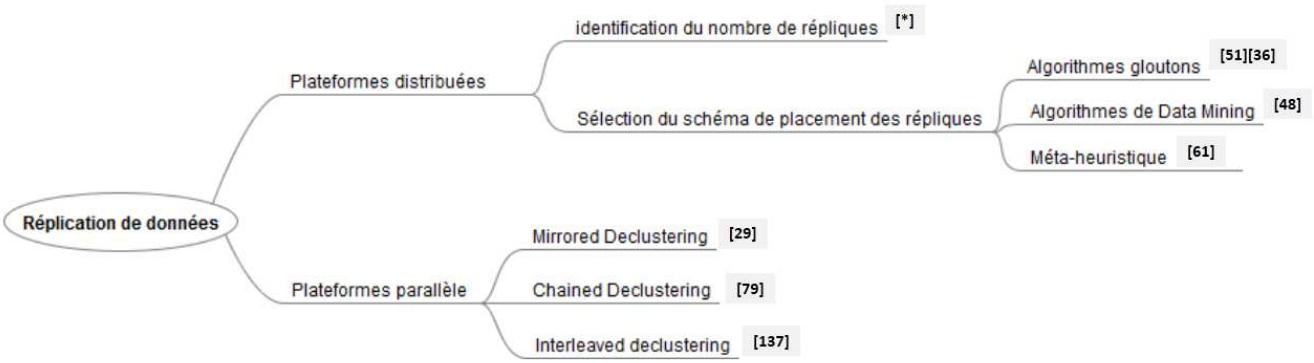


Figure 2.22 – Classification des travaux de réplication de données

2.1.5 Equilibrage de charges

Une fois que les données sont fragmentées, allouées et répliquées sur les nœuds 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 nœuds de traitement. Le principal enjeu est d'allouer la charge de requêtes d'une manière équilibrée. Dans cette section, nous décrivons brièvement les principaux travaux proposés pour atteindre la haute performance d'un entrepôt de données parallèle.

4. Y. Mansouri and R. Monsefi. Optimal number of replicas with qos assurance in data grid environment. In Asia International Conference on Modelling and Simulation, pages 1686173, 2008.

Le problème d'équilibrage de charge doit veiller que la distribution de la charge d'exécution d'une requête soit répartie le plus équitablement possible entre les différents nœuds de traitement. L'équilibrage de charge vise à minimiser le temps moyen de réponse de requête et à maximiser le degré de l'utilisation des ressources. En effet, le temps de traitement parallèle des requêtes correspond au temps de traitement occupé par le nœud de traitement qui traite la charge de requêtes la plus large. En conséquence, une fois qu'un nœud termine sa charge de travail, il rentre dans un état d'inactivité en attendant que les autres nœuds terminent leur charge de requêtes. Autrement dit, l'équilibrage de charge affecte à chaque nœud une charge proche de la charge moyenne du système. Le problème d'équilibrage de charge peut apparaitre avec le parallélisme intra et/ou inter opérateur. Généralement, le système d'équilibrage de charge réordonne les requêtes pour atteindre ses objectifs. Ce ré-ordonnancement de requêtes s'appelle*migration*.

Exemple 3. *Soient une machine parallèle constituée de quatre nœuds de traitement N_1, N_2, N_3, N_4 et une charge de travail W partagée en quatre sous-requêtes W_1, W_2, W_3 et W_4 allouées sur les nœuds N_1, N_2, N_3 et N_4 respectivement. Nous considérons les deux scénarii illustrés dans la figure 2.23. Dans la figure 2.23* - a*, le nœud N_3 termine le dernier donc les autres nœuds N_1, N_2 et N_3 passent par un moment d'inactivation en attendant que N_3 termine (l'état d'inactivation est représenté par des lignes pointillés). D'autre part, dans la figure 2.23* - b*, tous les nœuds de traitement terminent en même temps. En conséquence, nous disons qu'une stratégie d'équilibrage de charge est bonne si tous les nœuds de traitement reçoivent approximent la même charge de travail.*

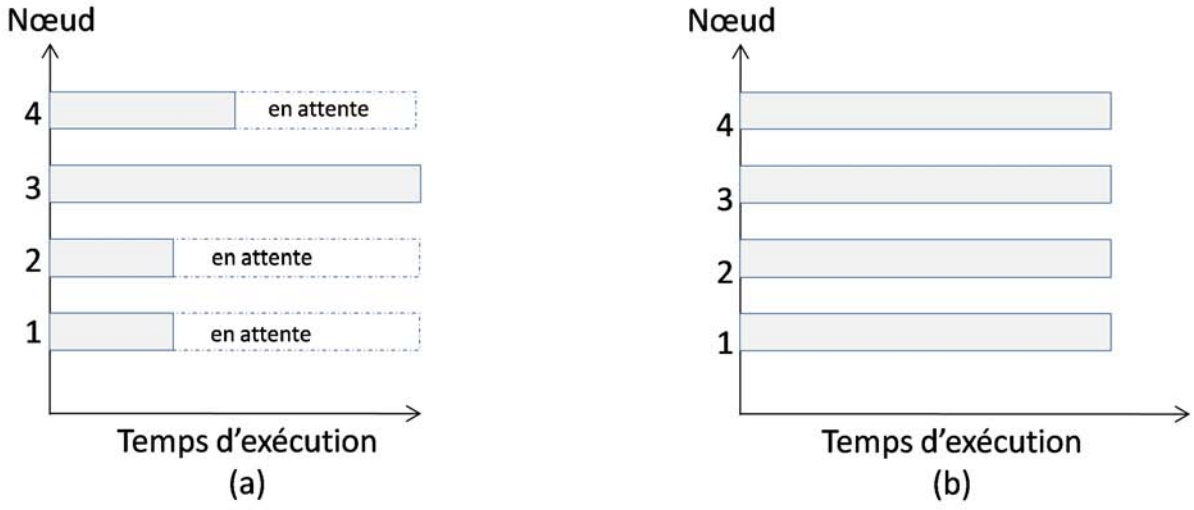


Figure 2.23 – *Equilibrage de charge vs déséquilibrage de charge.*

L'équilibrage de charge dépend également de l'architecture matérielle, il ne se pose pas pour les systèmes à disque partagé où chaque nœud a un accès direct à toutes les données. Ainsi, le système peut automatiquement atteindre l'équilibrage de charge en allouant la tâche suivante dans la file d'attente au premier processeur qui devient disponible. Cependant, l'équilibrage

de charge est un enjeu principal dans les architectures parallèles sans partage où chaque nœud n'a accès qu'à ses données locales.

Pour atteindre un gain de performance, un SGBD exploite la notion du parallélisme. Dans ce cadre, l'exécution parallèle d'une requête peut exploiter le parallélisme dans ses deux formes : intra-requêtes et inter-requêtes pour diminuer le coût d'exécution des requêtes. Le parallélisme permet l'exécution de plusieurs requêtes simultanément, souvent elles sont généralisées par l'exécution des transactions concurrentielles. Par contre, dans le parallélisme intra-opérateur, le même opérateur est exécuté par plusieurs processeurs, chacun travaillant sur un sous-ensemble de données. Dans ce qui suit, nous présentons les formes de parallélisme, puis le paradigme d'équilibrage de charge, et enfin, nous passons en revue les travaux d'état de l'art concernant l'équilibrage de charge.

2.1.5.1 Concepts de base.

Pour atteindre un gain de performance, un SGBD exploite la notion d'équilibrage de charge. Dans ce cadre, nous présentons la mise en œuvre du parallélisme intra et inter opérateur pour des modèles d'exécutions basés sur la fragmentation. Ensuite, nous décrivons le paradigme du traitement parallèle, les obstacles et les principale approches d'équilibrage de charge.

2.1.5.1.1 Formes du parallélisme. *Le parallélisme intra-requête* consiste à décomposer la requête en plusieurs sous-requêtes pour les exécuter simultanément. La décomposition de la requête consiste à diviser l'opérateur initial en sous-opérateurs nommés " instances de l'opérateur initial" et à réécrire les fonctions d'agrégation. Généralement, la décomposition de l'opérateur peut bénéficier de la répartition initiale des données; sinon, une redistribution préalable des données est requise avant l'exécution de l'opérateur. L'opérateur initial peut être une sélection ou une jointure. Dans ce qui suit, nous décrivons la procédure de décomposition de chacun.

L'opérateur de sélection sur une relation R est décomposé en n opérateurs de sélection identiques, chacun d'eux étant appliqué sur l'un des fragments de la relation initiale [30]. Comme illustré dans la figure 2.25. Il est à noter que cette décomposition est indépendante de l'algorithme de sélection (avec ou sans index). Généralement, si un index est utilisé, il sera fragmenté de la même manière que les données.

Par contre, pour l'opérateur de jointure, la décomposition est plus complexe vu la difficulté de ramener l'exécution de la jointure à l'exécution de n sous-jointures en utilisant un fragment de chaque relation en entrée. Il faut que chaque fragment de la deuxième relation contienne tous les tuples de cette relation susceptibles de se joindre au fragment correspondant de la première relation. Cela n'est possible qu'avec la fragmentation par hachage ou par intervalle dont l'attribut utilisé pour la fragmentation est l'attribut de jointure. Dans le cas contraire, une redistribution de l'une ou des deux relations (suivant l'attribut de jointure) est nécessaire [30].

Le parallélisme inter-opérateur consiste à exécuter en parallèle des opérateurs d'une même requête. Il existe deux formes de parallélisme inter-opérateur. [30]. Le parallélisme indépen-

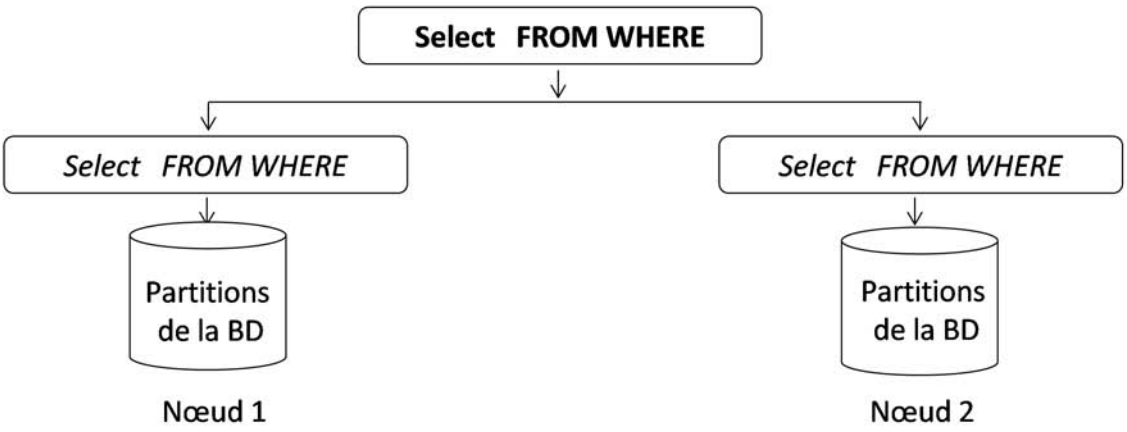


Figure 2.24 – *Parallélisme intra-opérateur de l'opérateur de sélection.*

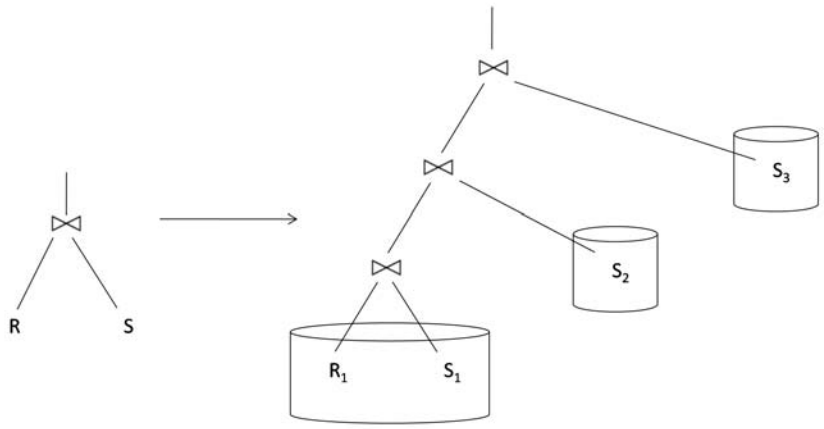


Figure 2.25 – *Parallélisme intra-opérateur de l'opérateur de jointure.*

dant permet d'exécuter en parallèle plusieurs opérations indépendantes d'une même requête. Le parallélisme en tuyau est basé sur la métaphore "*producteur-consommateur*". Il est utilisé dans le cas où le résultat d'une opération constituerait les données d'entrée pour l'opération suivante [30].

2.1.5.1.2 Paradigme d'équilibrage de charge Pour obtenir une bonne répartition de charge, il est nécessaire de déterminer le meilleur degré de parallélisme et de choisir judicieusement les nœuds de traitement pour son exécution. Le processus d'équilibrage de charge comporte quatre étapes [101].

1. *Partitionnement de la charge.* Le nœud coordinateur s'occupe de partitionnement de la charge de requête en un ensemble de sous-requêtes selon le schéma de partitionnement horizontale de la base de données. En effet, la fragmentation horizontale produit des sous-requêtes qui peuvent s'exécuter sur des nœuds de traitement indépendant. La taille

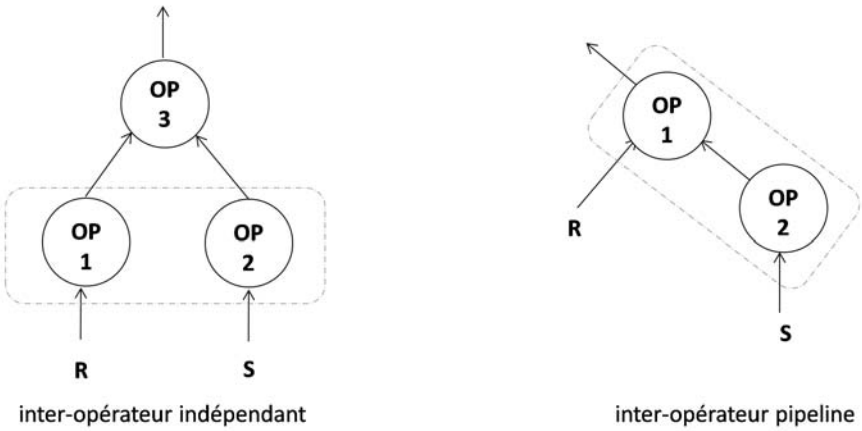


Figure 2.26 – *Parallélisme inter-opérateur.*

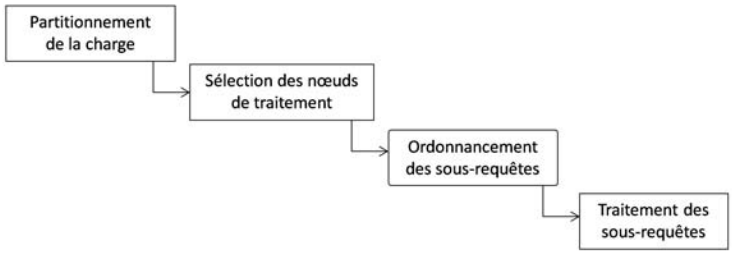


Figure 2.27 – *Paradigme d'équilibrage de charge.*

des fragments horizontaux impacte la stratégie d'équilibrage de charge, la taille petite des fragments assure plus de flexibilité et réduit les effets de skew. Ainsi, chaque sous-requête est caractérisée par sa taille qui correspond au temps d'accès de disques nécessaire pour le chargement des fragments.

2. *Sélection des nœuds de traitement.* Pour chaque sous-requête, l'ensemble des nœuds de traitement admissibles est déterminé selon le schéma de placement des fragments sur les nœuds de traitement ainsi que l'état du cache de chaque nœud. En conséquence, le nombre des nœuds admissibles est inférieur ou égal au nombre des nœuds de l'environnement parallèle.

3. *Ordonnancement des sous-requêtes.* Les sous-requêtes sont attribuées aux nœuds de traitement de telle sorte que tous les nœuds auront approximativement la même charge de travail. L'ordonnancement des sous-requêtes vise à minimiser le temps moyen de réponse des tâches et à maximiser le degré de l'utilisation des ressources. La détermination de la répartition de la charge finale est prédéterminée par la répartition des données, l'architecture du système, le contenu du cache, ou les dépendances entre données. L'ordonnancement des requêtes définit également l'ordre d'exécution des requêtes. Généralement, la migration de données est utilisée comme une solution naïve pour le problème d'équilibrage de charge. Il suffit de détecter les processeurs qui ont besoin de coopérer avec d'autres pour échanger des données ou des résultats intermédiaires. Ainsi, il faut trouver un compromis entre la communication entre les nœuds et l'équilibrage de charge.

4. *Traitement des sous-requêtes.* Quand un nœud reçoit sa charge de travail, il commence à vérifier l'existence des fragments sur son disque. S'il ne les trouve pas, il se les procure chez ses voisins. Une fois que tous les fragments sont disponibles, la charge de sous-requêtes affectée au nœud est exécutée avec l'objectif de réduire la consommation de la mémoire et à accélérer le traitement.

2.1.5.1.3 Obstacles de l'équilibrage de charge .

Un bon équilibrage de charge est crucial pour la performance d'un système parallèle où le temps de réponse de l'ensemble des opérateurs parallèles correspond à celui du plus large. L'équilibrage de charge peut être influencé par plusieurs problèmes : le contrôle de la concurrence, l'interférence et la communication, et la distribution biaisée. [135, 30]

L'exécution parallèle d'une requête nécessite *l'échange des messages de contrôle entre les différents processus et les processeurs concernés par l'exécution parallèle.* Ces messages assurent la synchronisation, le déclenchement des opérateurs de la requête et la signalisation de la fin d'exécution d'une requête. Plus le degré de parallélisme est élevé, plus le nombre de messages de contrôle est important, ce qui dégrade la performance désirée et impacte négativement l'équilibrage de charge.

Généralement, *l'exécution parallèle nécessite l'accès simultané à des ressources partagées.*

Pour les *Ressources matérielles*, une contention est créée au niveau du bus ou du disque lors des accès mémoire effectués par les processeurs ou les entrées/sorties sur le même disque effectuées par les processus. En effet, les accès mémoire et disque sont séquentiels au niveau du système. L'une des solutions proposée pour remédier à ce problème consiste à multiplier les ressources matérielles pour limiter les conflits d'accès.

Pour les *Ressources logicielles*, l'accès concurrentiel aux données produit des données incohérentes et des goulots d'étranglement. Ainsi, la sérialisation de ces accès est nécessaire pour assurer la cohérence des données. Malheureusement, cette solution pénalise le coût d'exécution des requêtes vu la mise en attente des requêtes. D'autre part, une solution est proposée, qui consiste à paralléliser la ressource partagée, c'est à dire à la décomposer en sous-ressources indépendantes. Deux sous-ressources distinctes peuvent alors être utilisées en parallèle, ce qui limite la probabilité d'interférence [30]. La figure 2.28 illustre un exemple.

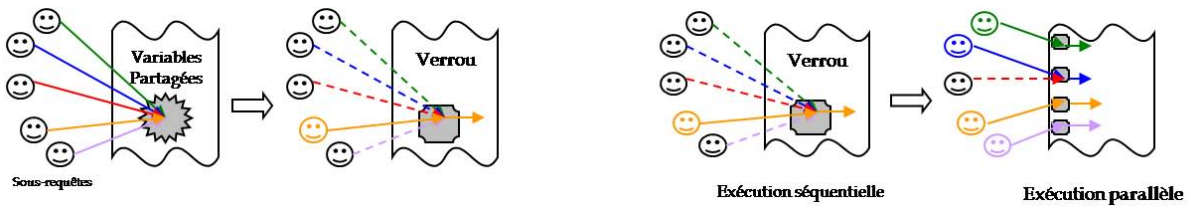


Figure 2.28 – Interférences des tâches parallèles

L'équilibrage de charge est impacté principalement par *la répartition biaisée des données et/ou de traitements.* Walton et al [144] ont classifié les effets d'une mauvaise répartition des données lors d'une exécution parallèle avec un modèle d'exécution fragmenté.

1. La mauvaise répartition des valeurs d'un attribut (Attribute Value Skew)

(AVS). Le skew des valeurs d'un attribut se produit lorsque les valeurs d'attribut ne sont pas réparties uniformément. Cela signifie que certaines valeurs d'attribut apparaissent avec des fréquences beaucoup plus élevées que les autres. L'AVS est une propriété de données et ne change pas entre les algorithmes. Toutefois, afin de bénéficier du parallélisme, les algorithmes utilisés doivent respecter les techniques de distribution pour équilibrer la charge des différents processeurs, même en présence de l'AVS.

2. La mauvaise répartition de données (Partition Skew) (PS). La mauvaise répartition se produit lorsque les données sont inégalement réparties sur les partitions. PS peut être divisé en quatre catégories.

   – *Le mauvais placement des tuples*, (tuples Placement Skew) (TPS), concerne la distribution initiale des données sur les disques des nœuds de traitement. Il apparaît généralement lorsque la fonction de partitionnement et/ou d'allocation se base sur des attributs ayant une mauvaise distribution des valeurs.

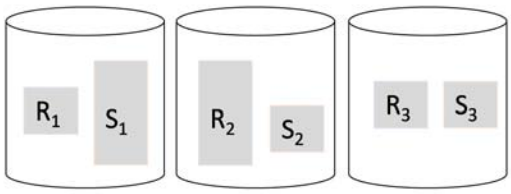


Figure 2.29 – *Exemple du mauvais placement des tuples*

   – *La mauvaise distribution de Sélectivité*, (Selectivity Skew) (SS), se produit lorsque les prédicats de sélection impliquent un nombre de tuples différent entre les processeurs. Autrement dit, elle est due à la variation de la sélectivité des prédicats de sélection entre les processeurs.

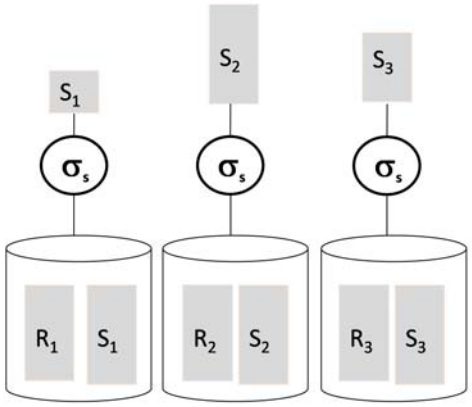


Figure 2.30 – *Exemple de la mauvaise distribution de sélectivité*

– *La mauvaise redistribution des fragments*, (Redustribution Skew) (RS), se produit lorsqu'il y a une différence entre la distribution des valeurs de la clé de jointure d'une relation et la distribution prévue par le mécanisme de redistribution.

– *La mauvaise distribution des résultats intermédiaires*, (Result Size Skew) (RSS), se produit lorsqu'il y a une grande différence entre la taille des résultats issues par chaque processeurs.

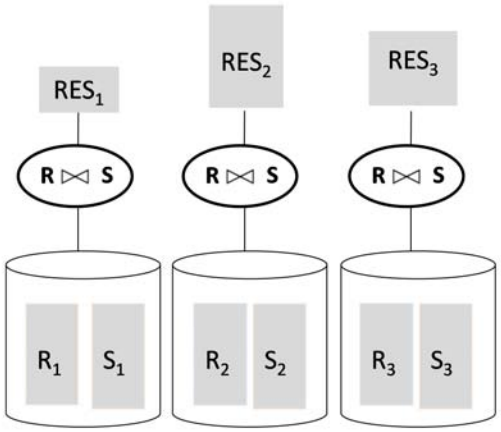


Figure 2.31 – *Exemple de la mauvaise des résultats intermédiaires*

3. La mauvaise distribution des capacités (Capacity Skew) (CS). Elle représente la charge de travail que chaque processeur est capable de la traiter. Cela peut être dû à l'hétérogénéité qui concerne l'architecture et le système d'exploitation comme la taille de la mémoire, version du système d'exploitation, puissance de calcul, capacité de stockage, . . . etc.
4. La mauvaise distribution de traitement (Processing Skew) (PS) est le problème majeur des bases de données parallèles. Il se produit lorsque le temps d'exécution des sous-requêtes sur les nœuds de traitement est hétérogène. Le PS influence négativement le temps d'exécution. Il est généralement dû à la mauvaise distribution de données.

2.1.5.1.4 Algorithmes d'équilibrage de charge .

Pour obtenir une bonne répartition de charge, il est nécessaire de choisir judicieusement la stratégie d'ordonnancement des requêtes. L'équilibrage des charges peut se faire de deux manières différentes. Dans la littérature, les principales solutions proposées s'appuient sur des techniques adaptatives ou spécifiques [30]. La première catégorie prend en compte la dynamique du système informatique, l'autre catégorie est statique mais elle est plus simple à mettre en œuvre.

Dans *Techniques adaptatives*, sont également appelées *équilibrage dynamique*, l'idée de base de ces techniques consiste à décider statiquement de la répartition initiale de la charge sur les nœuds de traitement en utilisant un modèle de coût qui estime la complexité de chaque instance d'opérateur. Puis la migration de données est utilisée pour adapter la mauvaise répartition de la charge entre les nœuds où un repartitionnement des grands fragments est exécuté. En

effet, il faut maintenir des statistiques comme les histogrammes sur les attributs des tables, plus particulièrement, les attributs de jointure, les attributs de fragmentation, les attributs participant aux prédicats de sélection. Une telle technique provoque des coûts supplémentaires (coût de communication, arrêt d'une requête, transfert de données, et relance de la requête). Il est important de comparer les gains de l'équilibrage dynamique avec les coûts d'administration d'un tel comportement. Ainsi, les coûts engendrés par une migration doivent être pris.

Plusieurs stratégies ont été proposées, KITSUREGAWA ET AL [91] ont proposé de détecter les partitions de grande taille et de les repartitionner sur les nœuds de traitement pour ajuster dynamiquement le degré du parallélisme. Ils ont utilisé des opérateurs de contrôle spécifique pour exécuter les plans d'exécution et détecter si les estimations statiques des résultats intermédiaires diffèrent des valeurs d'exécution. Si la différence entre la valeur estimée et la valeur réelle est significative, l'opérateur de contrôle appliqué sur la relation redistribue la relation afin d'empêcher la mauvaise redistribution de données. DBS3 [31] a lancé l'utilisation d'une technique adaptative basée sur le partitionnement des relations (comme dans shared-nothing) pour la mémoire partagée. En réduisant les interférences entre les processeurs, cette technique donne un excellent équilibrage de charge pour parallélisme intra-opérateur.

Dans *Les techniques spécialisées*, les algorithmes de jointures parallèles peuvent être spécialisés pour gérer le skew[58]. Ils reposent sur deux techniques principales: partitionnement par intervalle et échantillonnage. Le partitionnement par intervalle est utilisé au lieu du partitionnement par hachage pour minimiser la redistribution du skew lors de la création des partitions. Ainsi, les processeurs peuvent obtenir des plages de valeurs de même taille. Pour déterminer les valeurs qui délimitent les valeurs d'intervalle, l'échantillonnage est utilisé pour produire un histogramme des valeurs d'attribut de jointure, c'est-à-dire, le nombre de n-uplets pour chaque valeur d'attribut. L'échantillonnage est également utile pour déterminer quel algorithme utiliser et quelle relation utiliser pour la construction des partitions. Grâce à ces techniques, l'algorithme de jointure parallèle par hachage peut être adapté pour faire face au problème de la mauvaise distribution. Voici la procédure:

- échantillonner les relations construites pour déterminer les partitionnements par intervalles,
- redistribuer la relation construite pour les processeurs en utilisant les intervalles, chaque processeur construit sa table de hachage contenant les nouveaux tuples,
- redistribuer les relations *Probe* en utilisant les mêmes intervalles. Pour chaque tuple reçu, chaque processeur *Probe* sa table de hachage pour exécuter la jointure.

Cet algorithme peut encore être amélioré par les différentes stratégies d'allocation du processeur [58]. Une approche similaire consiste à modifier les algorithmes de jointure en insérant une étape d'ordonnancement qui se chargera de la redistribution de la charge lors de l'exécution [146]

2.1.5.2 Travaux existants

2.1.5.2.1 Travaux existants dans le contexte parallèle

2.1.5.2.1.1 Travaux de Röhm et *al* .

Ils proposent une approche de conception d'un entrepôt de données distribuées en utilisant le partitionnement physique [124]. Ils ont fragmenté horizontalement la table des faits et ont répliqué toutes les tables de dimension sur les nœuds du cluster de base de données. Les fragments de la table des faits sont alloués d'une manière circulaire sur tous les nœuds du cluster. Il est à noter que le nombre de fragments est égal au nombre des nœuds et que les attributs de partitionnement utilisés sont les clés primaires de la table des faits. Les auteurs proposent deux nouvelles stratégies nommées respectivement Balance-Query-Number et Affinity-Based Query Routing. La stratégie *Balance-Query-Number* est une variante de l'allocation circulaire; elle tente d'avoir le même nombre de requêtes actives sur chaque nœud de traitement. Lorsqu'une requête est soumisse au système, elle est acheminée vers le nœud ayant le plus petit nombre de requêtes actives. En effet, la liste des nœuds de traitement est triée selon le nombre de requêtes actives et le choix du nœud qui va contenir la requête en question est fait d'une manière circulaire Comme son nom l'indique, la stratégie *Affinity-Based Query Routing* consiste à assigner les requêtes qui accèdent aux mêmes données au même nœud pour bénéficier du contenu du cache et, ainsi, réduire le coût des entrées sorties. En effet, l'affinité existe si les requêtes accèdent aux mêmes données et la nature de l'accès est importante car elle influence le contenu du cache.

2.1.5.2.1.2 Travaux d'Akal et *al* .

Ils proposent une approche flexibe basée sur le *partitionnement virtuel* pour le traitement des requêtes [2], appelée *Simple Virtual Partitioning (SVP)*. SVP consiste à répliquer totalement la base de données sur les nœuds du cluster et à définir un *Clustred index* sur les clés primaires de la table des faits. Le partitionnement virtuel se fait sur les clés primaires des tables. L'attribut de partitionnement virtuel permet de générer des sous-requêtes dont chacune porte sur une partie différente de la table des faits. En effet, les sous-requêtes sont créées par l'ajout d'un prédicat sur l'attribut de partitionnement à la clause *where* de la requête d'origine. Les bornes des intervalles représentent les limites de séparation. L'objectif est d'attribuer la même quantité de données à traiter à tous les nœuds. Par exemple, l'exécution de la requête *Q : SELECT count(*) from R* sur un cluster de n nœuds en utilisation SVP nécessite le choix d'un attribut de partitionnement virtuel.

Soient A cet attribut et $I = [amin, amax[$ l'intervalle de ces valeurs. Ainsi, SVP produit exactement n sous-requêtes $Q_i 1 \leq i \leq n$ de la forme:

Q_i : SELECT count(*) FROM R WHERE $A \geq v_i$ and $A \leq v_{i+1}$,
où $v_i = amin + (i-1) \times S$ et $S = \frac{(amax-amin)}{n}$.

Il est important de noter que S est la taille de chaque partition qui doit être uniforme. Chaque intervalle est calculé et les sous requêtes Qi sont soumisses aux différents nœuds du cluster. Les auteurs supposent que l'attribut est connu et que les valeurs sont uniformément distribuées, le calcul des séparateurs des partitions devient trivial. Cependant, *Akal* a mis en cause ces hypothèses et a décrit les difficultés rencontrées pour déterminer les limites des partitions:

– l'hypothèse sur la distribution uniforme des valeurs n'est pas toujours vérifiée,
– la requête d'origine contient un prédicat sur l'attribut de partitionnement,
– si l'hypothèse sur la distribution uniforme des valeurs est vérifiée, il est difficile de déterminer les bornes des partitions pour des requêtes qui contiennent des semi-jointures.

Dans SVP, la taille des partitions est déterminée selon le nombre de nœuds du cluster. Dans le cas où le nombre des nœuds du cluster n'est pas grand et la taille de la base de données est volumineuse, la taille de partition deviendra très grande. En conséquence, le clusterd index ne sera pas utilisé et le problème d'équilibrage de charge restera toujours un challenge car chaque nœud traite uniquement une requête et l'exécution de la requête dépend du SGBD qui restera une boîte noire pour SVP.

2.1.5.2.1.3 Travaux de Märtens et *al* .

Les auteurs s'intéressent aux stratégies d'équilibrage de charge pour le traitement parallèle des schémas en étoile et leur index de jointure bitmap [101]. Ils ont constaté que les heuristiques simples pour l'ordonnancement des requêtes, notamment celles qui se basent sur le nombre et la taille des partitions, peuvent ne pas être très efficaces. En effet, le choix de la méthode appropriée dépend des propriétés de la charge de la requête et des caractéristiques du système, ce qui peut être difficile à déterminer. Dans certains cas, en particulier dans cas de présence du skew. Märtens et ses collégues proposent une solution alternative pour l'ordonnancement dynamique des requêtes sur une machine parallèle à disque partagé ou une machine parallèle sans partage. L'approche d'allocation des requêtes s'applique sur les bases de données de partitionnement horizontalement en un ensemble de fragments indépendants. En outre, le schéma de base de données est accompagné des index de jointure bitmap et des vues matérialisées. La solution naïve consiste à gérer les résultats intermédiaires stockés sur le disque. En effet, la charge CPU causée par les opérateurs dans la mémoire cache est implicitement prise en compte. Ils proposent une stratégie de planification intégrée qui simultanément considère les deux processeurs et les disques, ce qui concerne non seulement la charge de travail de chaque ressource, mais aussi la répartition de la charge au fil du temps.

2.1.5.2.1.4 Travaux de Lima et *al* .

Ils proposent *Fine-Grained Virtual Partitioning (FGVP)*, qui est une variante de SVP [96] . Contrairement à SVP, $FGVP$ utilise un large nombre des sous-requêtes au lieu d'une seule requête par nœud. L'objectif de la subdivision en sous-requêtes est d'éviter l'analyse complète de tables et de rendre le traitement des requêtes moins vulnérable aux $SGBD$. Les résultats expérimentaux préliminaires ont montré que $FGVP$ surpasse SVP pour certaines requêtes $OLAP$. Toutefois, c'est une méthode simpliste pour la détermination de taille de la partition, elle se base sur des statistiques de la base de données et des estimations sur le temps de traitement des requêtes. Dans la pratique, ces estimations sont difficiles à obtenir avec les SGBDs qui sont considérés comme des boîtes noires. L'un des problèmes fondamentaux posés par $FGVP$ est " *Comment déterminer la taille des partitions virtuelles ?*". Pour répondre à cette question, les auteurs proposent une variante de $FGVP$ nommée *Adaptive Virtual Partitioning* (AVP) [95] qui essaye de calculer la taille de chaque partition selon le catalogue d'information du SGBD (cardinalité, la valeur de distribution des attributs, l'existence des clustred index).

AVP commence par produire des sous-requêtes en ajoutant des prédicats de sélection sur l'attribut de partitionnement. Chaque nœud du cluster reçoit une seule requête ainsi que les intervalles à traiter (comme SVP). Cependant, au lieu d'exécuter une seule requête sur tout l'intervalle, le nœud subdivise l'intervalle et exécute de nombreuses sous-requêtes. Puis, il tente de soulever la taille de l'intervalle jusqu'à ce qu'il n'y ait pas de dégradation des performances. Les résultats expérimentaux préliminaires ont montré qu'AVP devance SVP pour certaines requêtes OLAP.

2.1.5.2.1.5 Travaux de Phan et *al* .

Ils ont proposé un Framework pour la coordination et l'optimisation de l'exécution des requêtes OLAP sur un cluster de serveurs de bases de données [121] . Le principal objectif est de trouver la répartition de charge optimal qui minimise le coût de construction des tables de requêtes à matérialisées (MQT) ainsi que le temps d'exécution de la sous-charge de travail de chaque nœud de traitement. Cependant cette solution est complexe car elle implique le temps de construire des tables de requêtes matérialisées et leur impact sur l'exécution de la requête. En effet, la construction des MQT doit prendre en considération l'espace de stockage réservé pour les MQT, la puissance de calcul des nœuds de traitement et la taille de l'espace de toutes les combinaisons de configurations possibles. Ce problème a été formalisé comme étant un problème combinatoire dont l'espace de recherche est exponentiel en nombre de requêtes, de MQT et de nœuds de traitement. Phan et *al* utilisent un algorithme génétique pour sélectionner la meilleure correspondance (requête - nœud) et (MQT - nœud).

2.1.5.2.1.6 Travaux de Lima et *al* .

C'est une approche qui combine les avantages du partitionnement virtuel et physique pour avoir une réplication partielle qui permet la redistribution dynamique des sous-requêtes entre les nœuds. Les auteurs n'utilisent aucune charge de requêtes pour déterminer le schéma de partitionnement; leur solution est destinée aux requêtes OLAP ad-hoc. Les auteurs ont utilisé la technique de partitionnement décrite dans [64], qui consiste à partitionner la table des faits et à répliquer toutes les tables de dimension sur les nœuds de l'environnement parallèle. Le processus de traitement d'une requête Q se fait en trois étapes :

- récupération des fragments valides (fragmentation physique) pour la requête Q
- exécution locale de ses sous-requêtes par chaque nœud(durant cette étape, les fragments ne sont pas traités uniquement par les sous requêtes, ils peuvent être réfactionnés virtuellement)
- si un nœud devient inactif, équilibrage de charge par la réallocation des sous-requêtes des nœuds occupés vers les nœuds inactifs qui contiennent des copies des fragments répliqués.

La technique de partitionnement virtuel rend possible la réallocation des sous-requêtes sans avoir recours au transfert des données. L'objectif principal est d'adapter automatiquement et dynamiquement la taille des partitions virtuelles à n'importe quelle sous-requête en évitant le parcours total (full scan) et en réduisant le temps nécessaire pour l'initialisation des requêtes.

2.1.5.2.2 Travaux existants dans le contexte distribué

2.1.5.2.2.1 Travaux de Gorla *et al* .

Les auteurs proposent une formalisation pour le problème d'allocation de la sous-requête [69]. Ils proposent également un modèle de coût analytique pour minimiser le temps de traitement total et minimiser le temps de réponse. La minimisation du coût total du système cherche généralement à réduire la consommation de ressources (CPU, ES, des canaux de communication), ce qui augmente le débit de transmission et le nombre de requêtes à exécuter. D'autre part, la réduction du temps de réponse peut être obtenue en ayant un grand nombre d'exécutions en parallèle sur les différents sites, ce qui nécessite une consommation élevée des ressources. En conséquence, le débit du système se réduit, le problème traité est *NP-complet*. Les auteurs ont utilisé un algorithme génétique pour en résoudre le problème. Les résultats trouvés indiquent que les plans d'exécution qui minimisent le temps de traitement ne sont pas forcement efficients pour atteindre un temps de réponse minimal.

2.1.5.2.2.2 Travaux de Gounaris *et al* .

Les auteurs s'intéressent aux stratégies de traitement adaptatives des requêtes pour équilibrer la charge et supprimer les goulets d'étranglement dans les plans de requêtes parallèles [71]. Ils proposent deux solutions, l'une pour l'équilibrage de charge dynamique et l'autre pour la gestion des goulots d'étranglement. Les deux solutions sont instanciées dans le cadre de la même architecture générique pour la construction des techniques adaptatives. Les résultats de l'évaluation empirique du prototype présenté montrent qu'elle peut conduire à des améliorations significatives de performances dans des scénarii représentatifs. En outre, la surcharge reste assez faible, ce qui est une propriété importante lorsque l'adaptabilité n'est pas nécessaire.

2.1.5.3 Bilan et discussion

L'équilibrage de charge est un phase cruciale au niveau de la conception d'un système distribué(Cluster, Cloud, Grille, architecture parallèle). Il couvre l'ensemble des techniques permettant une distribution équitable de la charge de travail entre les ressources disponibles pour optimiser le temps de réponse moyen d'une charge de requêtes.

Le problème d'équilibrage de charge est composé principalement de deux problèmes : *la réécriture des requêtes* et *l'allocation des requêtes* générées. Chaque requête est réécrite selon le schéma de fragmentation, et les sous-requêtes générées sont allouées sur les nœuds de traitement. Ainsi, le problème d'équilibrage est fortement lié à la fragmentation et à l'allocation de données(avec ou sans redondance).

Cette brève étude, nous a permis de distinguer deux classes de travaux, une qui s'intéresse à l'équilibrage de charge au moment de l'exécution des requêtes en faisant appel au transfert de données ou de sous-requêtes entre les nœuds de traitement, l'autre, qui fait recours à la réplication pour améliorer la disponibilité et atteindre le bon degré de réplication. Certains travaux utilisent d'autres structures d'optimisation comme les indexes pour assurer l'équilibrage de charge. La majorité des algorithmes utilisés sont gloutons et ils n'utilisent pas un modèle de coût dédié au traitement parallèle des requêtes pour trouver le meilleur schéma du système équilibré

Le tableau 2.4 résume les travaux présentés dans cette section. Pour chaque travail, il est mentionné le problème étudié, le type de réplication utilisée(la mention "-" signifie que la réplication n'a pas été utilisée), la nature de l'algorithme et la plateforme de déploiement.

Plateforme	**Travaux**	**Problème étudié**	**Fragmentation**	**Replication**		**Algorithme**
				Mode	*Unité*	
Parallèle (Cluster de PC, Grille de calcul, Machine parallèle)	Röhm et al [124]	Réécriture requêtes	Physique	Partielle	Fragments	Glouton
	Akal et al [2]	Réécriture requêtes	Virtuelle	Totale	BD	Glouton
	Lima et al [96]	Réécriture requêtes	Virtuelle	Totale	BD	Glouton
	Lima et al [97]	Réécriture requêtes	Physique	Partielle	Fragments	Glouton
	Gorla et al [69]	Allocation requêtes	Physique	-	-	Génétique
	Gounaris *et al* [71]	Allocation requêtes	Physique	-	-	Génétique
	Märtens et al [101]	Allocation requêtes	Physique	-	-	Glouton
	Phan et al [121]	Allocation requêtes	Physique	Partielle	MQT	Génétique

Tableau 2.4 – Synthèse de comparaison entre les travaux d'équilibrage de charge

La figure 2.22 illustre une classification des travaux que nous avons étudiés. Cette classification porte sur le problème étudié, le type de la réplication ainsi que la nature de la plateforme de déploiement.

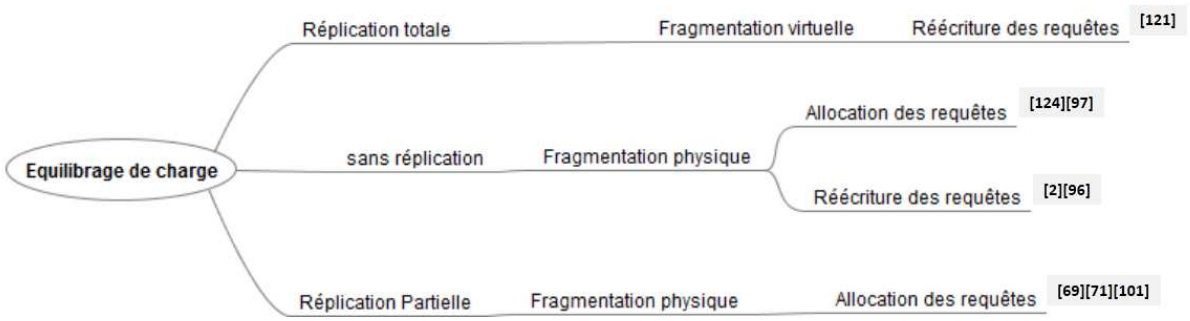


Figure 2.32 – Classification des travaux d'équilibrage de charge

2.2 Les implémentations principales des EDPs

Actuellement le monde des entrepôts de données est principalement couvert par les grands éditeurs de moteur de bases de données : *Teradata*, *Oracle*, *DB2* et *Sybase IQ*. Ces bases de données peuvent être classées en deux groupes : les moteurs dédiés au data warehouse comme

Teradata et Sybase IQ et les moteurs généralistes comme *Oracle* et *DB2*. Dans ce qui suit, nous présentons les principaux concepts de quelques prototypes d'architecture parallèles issus de la recherche et commercialisés. Nous décrivons en particulier leur architecture et leur processus de traitement parallèle des requêtes auxquels nous nous intéressons dans notre travail.

2.2.1 Teradata

Teradata [138] est un système de traitement massivement parallèle qui fonctionne sur une architecture sans *partage*. Son système de gestion de base de données est scalable pour toutes les dimensions à un système de base de données à base de requêtes (le volume de données, le nombre de requêtes et la complexité des requêtes). L'unité de base du parallélisme dans Teradata est un processeur virtuel nommé *Processeur d'Accès Modulaire* (Access Module Processor (AMP)). Chaque AMP exécute les fonctions des SGBD sur ses données. Ainsi, le verrouillage et le contenu du cache ne sont pas partagés, ce qui assure la scalabilité. La figure 2.33 illustre l'architecture Teradata. Un nœud est un système multi-core avec des disques et une mémoire. Il produit un pool de ressources (disques, mémoire, ..) pour les AMPs. BYNET est le réseau utilisé pour relier les différents AMPs dans un nœud avec les autres nœuds. Les données qui entrent dans une base de données Teradata sont traitées par un algorithme de hachage sophistiqué et automatiquement répartis entre tous les AMP dans le système. La stratégie de distribution de données est également utilisée comme une technique d'indexation, ceci réduit considérablement la quantité de travail du DBA normalement requise pour mettre en place un accès direct. Pour définir une base de données Teradata, l'administrateur choisit simplement une colonne ou un ensemble de colonnes comme un index primaire pour chaque table. La valeur contenue dans ces colonnes indexées est utilisée pour déterminer l'AMP, qui détient les données, ainsi que l'emplacement logique des données dans l'espace de l'AMP disque associé. Pour trouver une ligne, la valeur de l'index primaire est de nouveau passée dans l'algorithme de hachage pour générer deux valeurs: numéro d'AMP et ID. Ces valeurs sont utilisées pour déterminer immédiatement quel AMP est propriétaire de la ligne et où les données sont stockées.

2.2.2 IBM Netezza

IBM Netezza [46] est une architecture de type deux tiers. Elle combine les avantages de SMP et MPP et elle est nommée *plateformes de traitement massivement parallèle asymétrique* (*Asymmetric Massively Parallel Processing* ,AMPP). Elle est constituée d'un hôte avec une structure SMP nommé *Host* qu'il s'occupe de la génération des plans d'exécution des requêtes et de l'agrégation des résultats. Les nœuds *S-Blade*, avec une structure MPP, s'occupent l'exécution des requêtes. Chaque nœud S-Blade est connecté au disque par un processeur de données spécial nommé FPGA (Field Programmable Gate Array). S-Blade et Host sont connecté au réseau. *IBM Netezza* réduit ce goulot d'étranglement en utilisant la composante FPGA (Field-Programmable Gate Array) qui est considérée comme l'accélérateur de la base de données. La distribution réelle des données sur plusieurs disques est déterminée par la clé de répartition qui peut être choisie lors de la création de la table. Si aucune clé de répartition

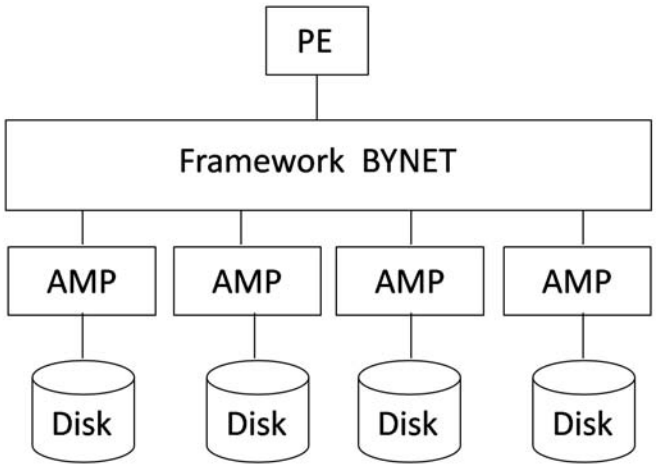


Figure 2.33 – *Teradata*

spécifiée n'est définie, les données sont distribuées aléatoirement. Le cas échéant, une fonction de hachage est appliquée sur la clé choisie. Le nombre maximum de colonnes qui peuvent participer à la clé de répartition est de quatre. Lorsque le système crée des enregistrements, il les attribue à une partition logique en fonction de leur valeur de clé de distribution.

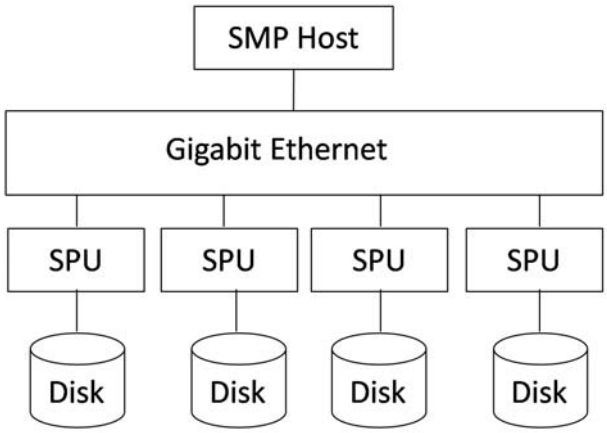


Figure 2.34 – *IBM Netezza*

2.2.3 Greenplum

Greenplum [47] est une architecture massivement parallèle (MPP) basée sur *PostgresSQL*. En effet, la base de données dans Greenplum est un tableau de bases de données PostgreSQL qui travaillent ensemble pour construire une seule base de données. Le nœud maître de Greenplum est l'instance de base de données où les clients se connectent et soumettent leurs requêtes SQL. Il contient uniquement un catalogue des tables de données. Le nœud maître coordonne le travail avec les autres instances de base de données (nœuds de traitement) dans le système pour assurer le traitement et le stockage des données. Les données et les index sont distribués sur les

segments disponibles et peuvent être traités exclusivement par le nœud maître. La distribution des données peut être déterminée par le hachage sur une clé qui peut être la clé primaire ou la composition de plusieurs colonnes de la table. Si les valeurs de hachage sont les mêmes, les données sont stockées sur le même segment. Une grande dispersion des données est assurée par la distribution d'une clé primaire. La clé primaire est également utilisée en l'absence de clé de distribution explicitement indiquée par l'administrateur. La seconde variante est une distribution aléatoire des données où les tuples sont distribués d'une manière circulaire sur les partitions.

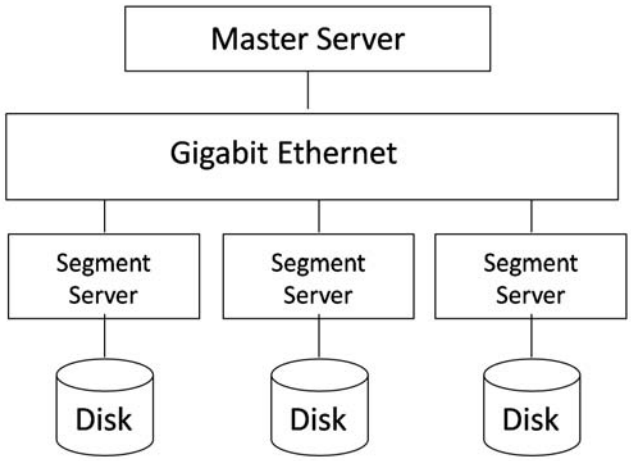


Figure 2.35 – *Greenplum*

2.2.4 Oracle Exadata

Oracle Exadata [114] est la seule solution de base de données offrant des performances extrêmes pour les applications d'entreposage de données aussi bien que pour le traitement transactionnel en ligne (OLTP), ce qui en fait la plateforme idéale pour la consolidation vers des Clouds privés. Elle constitue un ensemble complet associant serveurs, stockage, mise en réseau et logiciel, doté d'une évolutivité, d'une sécurité et d'une redondance exceptionnelle. Oracle Exadata permet d'améliorer les performances de l'ensemble des applications, d'accélérer la mise sur le marché des produits en éliminant les essais et les erreurs d'intégration des systèmes, et de prendre des décisions stratégiques plus judicieuses en temps réel. Les serveurs de stockage Oracle Exadata sont au cœur de chaque Exadata Database Machine d'Oracle. Ils associent des logiciels de stockage intelligents à du matériel standard pour offrir un stockage de base de données très performant du marché. Pour surmonter les limitations du stockage conventionnel, les serveurs de stockage Oracle Exadata font appel à une architecture massivement parallèle pour augmenter radicalement le débit de données entre le serveur de base de données et le stockage. Ils ont recours à la technologies innovantes, telles qu'Exadata Smart Scan, Exadata Smart Flash Cache et Hybrid Columnar Compressionpour l'entreposage de données, le traitement des transactions en ligne, les charges de travail mixtes, ... etc. Quand une requête est exécutée, les données stockées dans la grille des serveurs de stockage Exadata et

les serveurs de stockage agissent comme une sorte de préprocesseur pour accéder aux données à partir du disque de manière optimisée, en utilisant ce qu'Oracle appelle Smart Scan avant de transmettre les résultats de la base de données elle-même . Cela peut réduire considérablement la quantité de données que la base doit traiter et peuvent potentiellement bénéficier les utilisateurs dans un environnement d'entrepôt de données non-intrusif. Pour améliorer les performances dans les environnements OLTP, Oracle Exadata comprend également un stockage flash pour la mise en cache des données chaudes dans toutes les configurations.

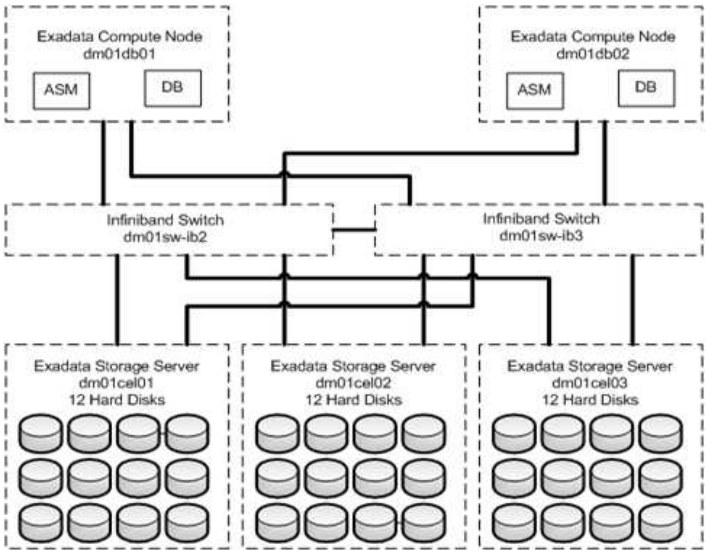


Figure 2.36 – *Oracle Exadata*

2.2.5 Microsoft SQL Server

*Microsoft SQL Server Parallel Data Warehouse (*PDW*)* [26] est une plateforme basée sur une architecture Massive Parallel Processing (MPP). Elle est composée d'un cœur applicatif nommé nœud de contrôle (*Control Node*) qui découpe les traitements pour les répartir et les exécuter en parallèle sur un certain nombre de serveurs appelés nœuds de traitement (Compute Node). Le Control Node communique avec les Compute Nodes à travers un réseau à haut débit. Chaque Compute Nodes possède sa propre mémoire, ses CPU, ses disques locaux, son environnement et sa plage de stockage dédiée. Les tables peuvent être soit répliquées sur chaque nœud de calcul de l'appareil, soit partitionnées par hachage sur une colonne spécifiée (s) dans les nœuds de traitement. Pour exécuter une requête, le nœud de contrôle transforme la requête de l'utilisateur en un plan d'exécution distribué (appelé plan de DSQL) constitué d'une séquence d'opérations (appel Operations DSQL). Chaque plan DSQL est composé de deux types d'opérations: les opérations SQL, qui sont des instructions SQL à exécuter sur les nœuds de traitement sous-jacentes des instances de SGBD, et les opérations DMS qui sont des opérations pour transférer des données entre les instances des différents nœuds de traitement.

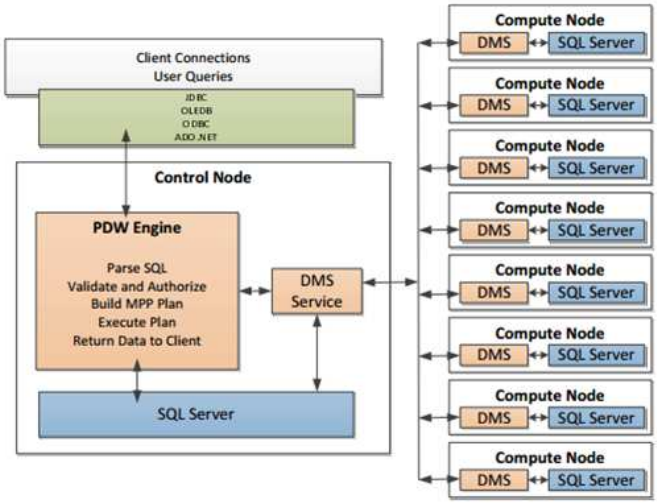


Figure 2.37 – *Microsoft SQL Server*

2.3 Mesures de performances

La scalabilité est une caractéristique des architectures multiprocesseur. Elle permet à une base de données d'utiliser des ressources additionnelles de manière optimale. On la définit comme étant la capacité d'une application à maintenir le même niveau de performance lorsque la charge augmente. Deux métriques sont généralement utilisées pour mesurer l'extensibilité d'un système parallèle : *speed-up* et *scale-up.*

Facteur de rapidité (*Speed-Up*)

Il mesure le gain de performance obtenu par l'augmentation du nombre des nœuds de traitement. Soit une requête de taille fixe exécutée de manière séquentielle en un temps T_s puis exécutée en parallèle sur p processeurs en un temps T_p, le speed-up obtenu par l'exécution parallèle est alors défini par

$$speed\ up(p) = \frac{Ts}{Tp} \tag{2.1}$$

Pour illustrer les différents scénarii possibles du facteur de rapidité, nous supposons que le temps d'exécution d'une requête Q sur un seul processeur nécessite 100 ms. La figure 2.38 présente le temps d'exécution de la requête Q sur une machine de 4 nœuds de traitement. Trois trois scénarii ont été identifiés.

– Si le coût d'exécution de la requêtes Q sur les quatre nœuds de traitement dire égal à 25 ms ($T_p = \frac{T_s}{4}$), cela signifie que le speed-up est égale à 4 et nous pouvons que *le speed up idéal* est atteint. La figure 2.38 *(a)* illustre cette situation.

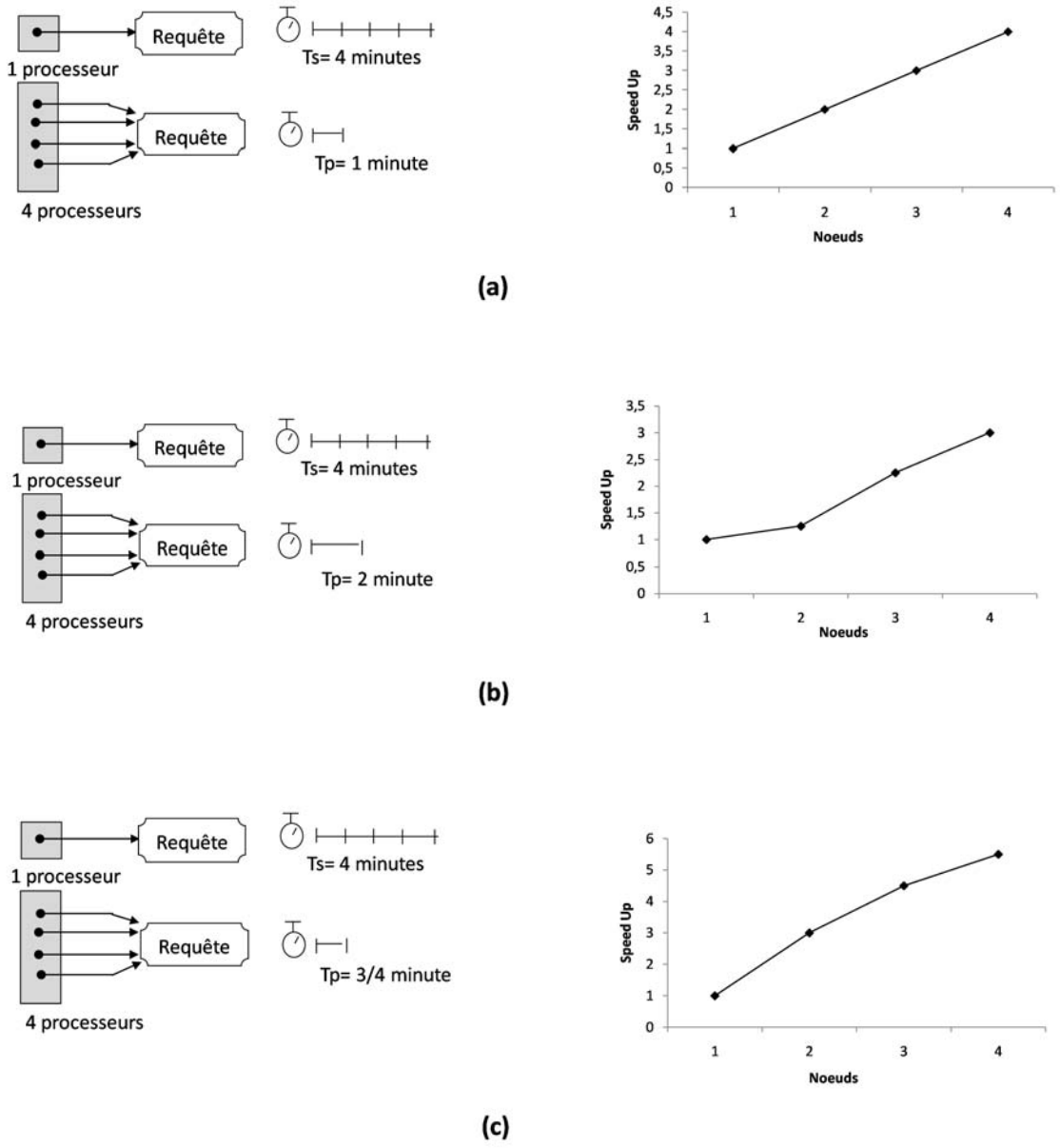


Figure 2.38 – *Exemple des Speed-Up*

– Si le temps découlé pour l'exécution de la requête Q sur les quatre nœuds de traitement prend plus de temps de 25ms($T_p \succ \frac{T_s}{4}$), la valeur du facteur de rapidité va baisser et nous disons qu'un f*acteur de rapidité sous-linéaire* est obtenu. La figure 2.38 *(b)* illustre cette situation.

– Dans un cas extrêmement rare, le temps de traitement de la requête Q sur les quatre nœuds de traitement peut être inférieur de 25 ms ($T_p \prec \frac{T_s}{4}$), ainsi, *le facteur de rapidité est sub-linéaire*, puisque le speed-up est supérieur au nombre des noeuds de traitement. La figure 2.38 *(c)* illustre cette situation.

Facteur de passage à l'échelle (*Scale-Up*)

Le passage à l'échelle est la capacité d'accompagner un accroissement de toute nature(volume, nombre de clients, charge des clients, vitesse de réseaux) de telle manière que le temps de réponse soit plus ou moins constant.

Le facteur de passage à l'échelle (*Scale-Up*) mesure la conservation du temps de réponse d'une requête pour une augmentation proportionnelle de la taille de la base de données et le nombre des nœuds de traitement de la machine parallèle. Le scale-up est idéal s'il reste toujours égal à 1 et il est dit linéaire.

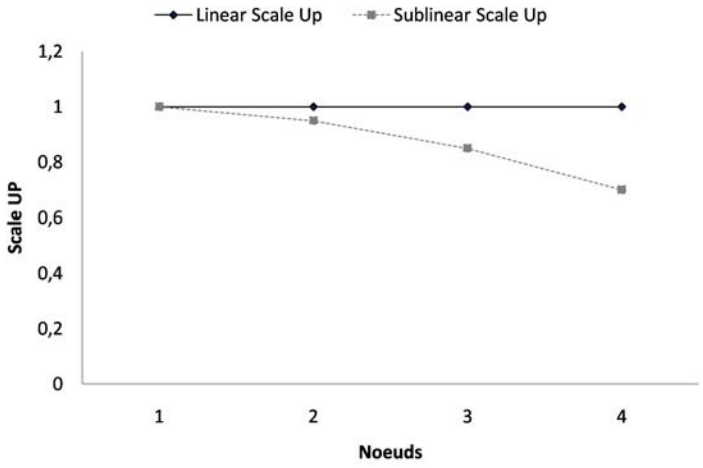


Figure 2.39 – *Exemple des Scale-Up*

Par exemple, soient deux problèmes P_1 et P_2 avec P_2 n fois plus gros que $P1$ et $T(Pi, p)$ le temps d'exécution du problème P_i sur un système avec p processeurs. Ainsi, Le scale-up est défini par la formule suivante:

$$scale\ up = \frac{T(P_1, p)}{T(P_2, p \times N)} \tag{2.2}$$

2.4 Bilan et discussion

Après une analyse fine de la littérature sur les travaux de le déploiement des bases de données parallèles en général et les entrepôts de données parallèles en particulier, nous remarquons que les problèmes liés au déploiement d'une base de données parallèle, à savoir, la fragmentation, l'allocation, la réplication et l'équilibrage de charge sont traités d'une manière indépendante (isolée). Le concepteur partitionne son entrepôt en utilisant une technique particulière de manière que le schéma de fragmentation généré optimise une charge de requêtes,

ensuite il alloue les fragments sur des sites (nœuds de la machine parallèle) en utilisant un algorithme particulier tel que le schéma d'allocation généré doit optimiser l'exécution de requêtes sur les différents nœuds. Puis, le concepteur duplique les fragments générés sur les nœuds de traitement en utilisant un algorithme dédié à la réplication de données afin d'assurer la haute disponibilité des données ainsi que la haute performance du système. Une fois les données partitionnées et allouées (d'une manière redondante), une stratégie d'équilibrage de charge est définie pour optimiser la charge de requêtes et atteindre la haute performance du système. Cette indépendance a fait naître quatre communautés, chacune travaillant sur l'un des principaux problèmes liés au problème de conception d'un entrepôt de données. Chaque communauté ne se soucie ni de la génération de ses entrées, ni d'une architectures de déploiement spécifique. En effet, les problèmes liés au déploiement des entrepôts de données ont été étudiés sur diverses architectures matérielles. Toutefois, les grands éditeurs de moteur de bases de données déploient les entrepôts de données parallèles en utilisant des approches de placement (circulaire, par intervalle et par hachage) et des stratégies de réplication conventuelles (chained declustering, mirrored Declustering, interleaved declustering). Malheureusement, ces approches n'utilisent aucun modèle de coût pour mesurer la performance des schémas obtenus avant son exploitation. Sachons que l'ensemble des rapports d'analyse sont élaborés durant la phase d'analyse des besoins. En outre, les autres approches proposées pour le déploiement d'un EDP utilisent au plus trois modèles de coût : un pour sélectionner l'ensemble de fragments optimisant l'ensemble de requêtes, un pour assurer une distribution efficace des fragments sur nœuds de la machine parallèle et un autre pour sélectionner le meilleur placement des répliques. Nous appelons cette démarche de déploiement par le déploiement itératif (ou séquentiel); ces étapes sont illustrées dans la figure ci-dessous.

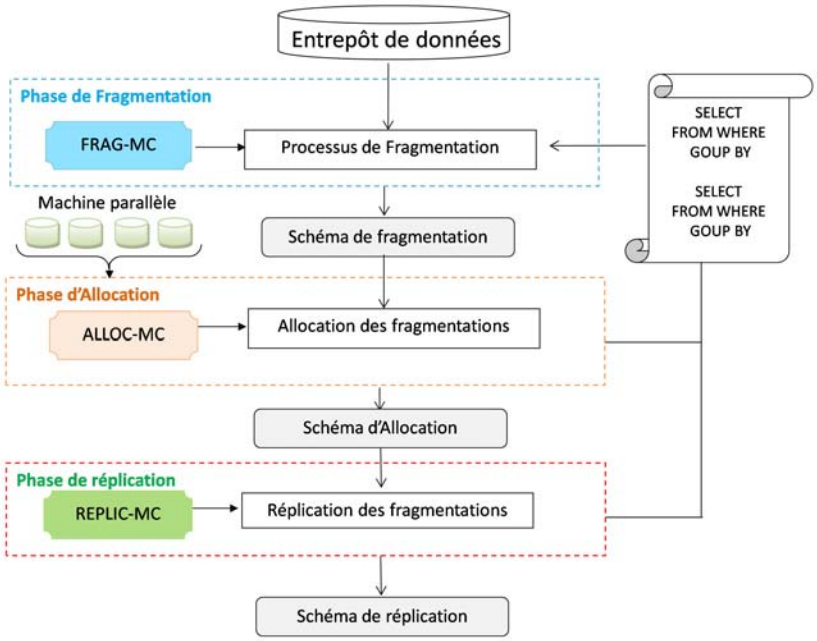


Figure 2.40 – *Les étapes de l'approche de conception itérative*

L'inconvénient majeur de ce cycle de déploiement est son ignorance de l'interdépendance entre la fragmentation, l'allocation, la réplication et l'équilibrage de charge. Comme nous avons

déjà mentionné, les problèmes liés à la conception parallèle sont mutuellement dépendants. De plus, chaque phase considère une métrique pour identifier la qualité de solution proposée. Cela génère une multitude de métriques hétérogènes qui peuvent pénaliser la solution finale. D'autre part, à l'exception des algorithmes conventionnels, l'approche de déploiement utilisée n'est pas générique. Chaque algorithme se déploie sur une architecture matérielle précise. En conséquence, les modèles de coût utilisés par ces algorithmes pour la sélection des schémas finaux ne prédisent pas correctement le coût; ils négligent l'effet de plusieurs paramètres. Les optimiseurs du SGBD sont sophistiqués et ils sont capables de prendre en considération des statistiques de base de données pour produire des estimations précises, mais le temps significatif consommé par les invocations de l'optimiseur pose des limitations sérieuses pour sélectionner la meilleure configuration de la conception physique, ce qui provoque de longues durées de fonctionnement, en particulier pour la conception physique de grandes bases de données ou d'entrepôts de données.

Ainsi, l'inexactitude dans la prédication de coût de la requête est nuisible pour la qualité des algorithmes, car ils augmentent les chances de sélectionner des solutions sous optimales. Par ailleurs, les approches proposées visent à atteindre la haute performance et elles ne considèrent la stratégie d'équilibrage de charge qu'au moment du traitement parallèle des requêtes. Donc, l'équilibrage de charge doit être pris en considération durant la phase de sélection du schéma de placement de données (qui englobe le schéma de fragmentation, d'allocation et de réplication). En conclusion, il apparaît donc qu'il existe un besoin fort d'une nouvelle approche générique de déploiement d'un entrepôt de données parallèle qui remédie à cet inconvénient.

Conclusion

Tout au long de ce chapitre, nous avons évoqué des éléments fondamentaux pour la compréhension des entrepôts de données parallèles auxquels nous allons faire référence dans les chapitres suivants. Nous pouvons conclure que principalement l"' déploiement d'un entrepôt de données parallèle consiste d'abord à partitionner son schéma ensuite à allouer les fragments générés sur les nœuds d'une machine parallèle. En conséquence, il est nécessaire de résoudre les problèmes suivants:

- quelle est l'architecture matérielle la plus adéquate pour un entrepôt de données ?
- comment l'entrepôt de données doit-il être fragmenté ?
- combien de copies de fragments doivent être répliquées ?
- comment les fragments vont-ils être alloués sur les nœuds de la machine parallèle ?
- quelle est l'information nécessaire pour le processus de fragmentation et d'allocation ?
- est-ce que la charge de travail est uniformément distribuée sur les nœuds ?
- quel est le processus le plus approprié au traitement parallèle d'une requête OLAP ?

Peu de travaux traitent d'une manière séquentielle (itérative) les problèmes liés au déploiement d'un entrepôt de données parallèle. En effet, les chercheurs utilisent plusieurs variantes des architectures sans partage pour mettre en place les EDP. Ainsi, plusieurs modèles de coût sont utilisés.

Plusieurs problèmes restent non résolus, dont principalement les algorithmes de placement où le critère utilisé change au point de dégrader l'équilibrage de la charge.

Nous devons donc faire recours à une nouvelle approche de déploiement générique sur les architecture distribuées sans partage. Notre approche prendra en considération l'interdépendance entre les sous-problèmes de la conception d'un entrepôt de données parallèle. Mais avant d'aborder cela, il nous faut définir un modèle unifié de coût représentant le paradigme de traitement parallèle et prenant en considération tous les paramètres des sous-problèmes de la conception d'un EDP: la fragmentation, l'allocation, la réplication et l'équilibrage de charge.

C'est pourquoi le chapitre suivant est consacré à la présentation de notre propre modèle de coût qui sera une métrique d'évaluation de la qualité de la conception d'un entrepôt de données parallèle.

Deuxième partie

Nos propositions

Chapitre 3

Notre Modèle de Coût

"Measurement is the first step that leads to control and eventually to improvement.
If you can't measure something, you can't understand it.
If you can't understand it, you can't control it.
If you can't control it, you can't improve it".
H. James Harrington (1611-1677)

La quantification de la qualité d'une solution de déploiement d'un entrepôt de données sur n'importe quelle plateforme passe souvent par la définition d'un modèle de coût prenant en compte les caractéristiques et les paramètres issus de la phase de déploiement. Les modèles de coût ont été largement utilisés dans le contexte des bases de données traditionnelles et avancées [32, 123, 110]. La première utilisation des modèles de coût concernait l'optimisation de l'exécution de requêtes. Historiquement, les optimiseurs de requêtes des SGBD utilisaient une approche dirigée par des règles (*Rule-based Approach*). Cette approche utilise un ensemble de règles bien définies comme exécuter aussitôt les opérations de sélections ensuite les opérations binaires comme la jointure. Avec l'apparition des gros schémas de bases de données impliquant un nombre important de tables (relations), cette approche a été substituée par des optimiseurs dits *dirigés par des modèles de coût* (*Cost-based Approach*). Pour illustrer le principe de cette optimisation, rappelons qu'une requête est composée d'un ensemble d'opérations algébrique, comme la sélection, la jointure, les agrégations, ... etc. L'exécution d'une requête se fait selon un plan d'exécution. Ce dernier est composé d'une séquence d'opérateurs algébriques ainsi que l'algorithme d'implémentation de chaque opérateur. Par exemple, pour une requête comportant deux jointures, un plan d'exécution peut montrer, qu'il faut *commencer* par la deuxième jointure, ensuite passer à la première, et que les deux jointures sont implémentées en utilisant un algorithme de *Hachage* [122]. Une requête peut avoir donc plusieurs plans d'exécution possibles. Le modèle de coût prend en paramètre un plan d'exécution et retourne son coût. 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.
La deuxième utilisation des modèles de coût est au niveau de problème de la conception physique. Comme nous l'avons déjà évoqué (cf. Chapitre 1), durant cette phase un ensemble de structures d'optimisation (comme les vues matérialisées, les index, le partitionnement, ... etc.) est sélectionné. Vue 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 algo-

rithmes génétiques, le recuit simulé [14] dirigées par des modèles de coût sont proposées. La troisième utilisation des modèles de coût concerne les étapes de la phase de déploiement de bases de données parallèles (voir le chapitre précédent).

Souvent tout modèle de coût est développé en fonction des composantes principales d'un SGBD [98, 115], 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 [5, 18]. Il dépend de la nature de la plateforme de déploiement.

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* (ex. la taille des tables ou fragments). L'obtention des paramètres calculés se fait à travers la définition des formules mathématiques. Ces dernières sont également nécessaires pour décrire le coûts des opérateurs locaux ainsi que le coût global. Pour calculer ces derniers, nous avons besoin de la politique de traitement de requêtes sur la plateforme parallèle. En sortie, nous aurons un coût final d'exécution d'une charge de requêtes sur une solution de déploiement caractérisée par un schéma de fragmentation, un schéma d'allocation, un schéma de réplication, et une politique de traitement de requêtes.

Dans la majorité des travaux existants, la définition des modèles de coût part d'un *sac* contenant l'ensemble de paramètres quelque soit leur origine. Pour rendre notre modèle de coût modulaire et flexible, nous proposons de catégoriser les paramètres, où ces derniers sont partitionnés en cinq groupes chacun est associé à une étape de la phase de déploiement.

Ce chapitre est consacré à la définition de l'ensemble de paramètres par phase, la présentation d'une politique d'exécution de requête sur une solution de déploiement à savoir une grappe de bases de données hétérogènes et l'élaboration des formules mathématiques pour le modèle de coût final estimant le coût d'une charge de requêtes exécutée sur un entrepôt de données fragmenté horizontallement et stocké sur une grappe de machine hétérogènes.

3.1 Les paramètres liés à notre modèle de coût

Dans cette section, nous présentons d'abord les paramètres pour les entrées du problème de déploiement à savoir l'entrepôt de données et les requêtes ensuite les paramètres liés à chaque étape et finalement la politique de traitement de requêtes. Une fois l'ensemble de paramètres définis, le modèle de coût global est alors défini.

3.1.1 Paramètres de l'entrepôt de données

Nous considérons les entrepôts de données modélisés par un schéma en étoile $\mathcal{DWS}$ et composés d'une seule 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 Bytes 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.

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

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

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

3.1.1.1 Paramètres des 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 d'équi-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 reqêtes Q_l

Tableau 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.

3.1.1.1.1 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 [84].

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

Nombre de travaux ont été développés pour estimer la sélectivité [84]. 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:

Soient A_i et A_j deux attributs d'une relation R, les formules de sélectivité sont les suivantes:

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

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

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

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

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

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

La taille d'une opération de jointure entre deux tables T_1 et T_2 est estimée en utilisant la formule suivante [141]:

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

Toutefois, certains travaux ne supposent aucune distribution particulière [106]. Dans ce cas, une approche à base d'histogrammes est proposée dans [83].

3.1.1.1.2 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 [74]. 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 $Sel(pred_1\ ET\ pred_2) = 0$.

3.1.2 Paramètres liés à l'architecture de déploiement

Notre entrepôt de données est déployé sur une grappe de base de données sans partage (Shared Nothing DataBase Cluster) notée $\mathcal{SN} - \mathcal{DBC}$ et composée de M nœuds de traitement *hétérogènes* $\mathcal{N} = \{N_1, N_2, \ldots, N_M\}$ dont chacun stocke une partie de l'entrepôt de données. Les nœuds de traitement sont reliés au nœud coordinateur par un réseau d'interconnexion et caractérisés par deux grandeurs. La première est *la puissance de calcul*, notée $P_i(1 \leq i \leq M)$, exprimée en nombre d'opérations que le nœud de traitement N_i peut les effectuer par seconde. Par contre, la seconde grandeur représente sa *capacité de stockage*, notée par S_i et est exprimée en *gigaoctets*. La figure 3.1 illustre un exemple d'une grappe de machines.

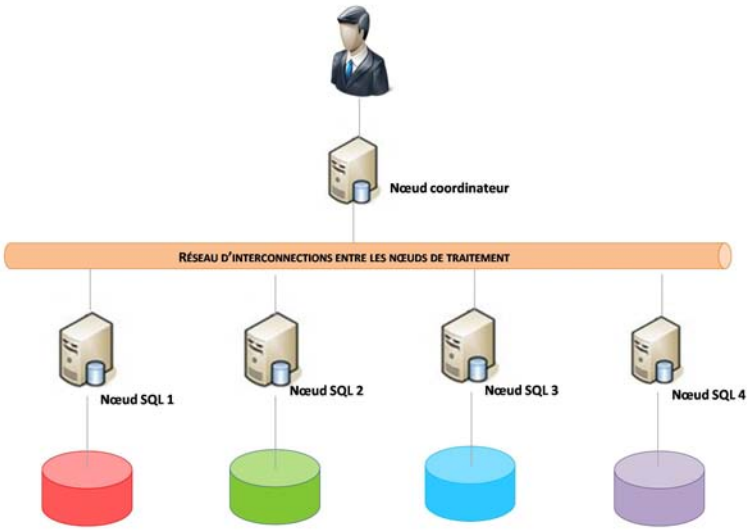


Figure 3.1 – *Exemple d'une grappe de machines*

Au niveau de chaque nœud N_i, les données sont chargées à partir du disque vers la mémoire principale *page* par *page*, Chaque *page* contient un lot d'enregistrements, sa taille est notée par PS_i et est exprimée en octets.

La communication entre les nœuds se fait grâce à une trame de données, par conséquent, le coût de communication $\mathcal{CC}$ peut être spécifié par une matrice carrée $(M \times M)$ dont les lignes et les colonnes représentent les nœuds de traitement. La valeur de chaque élément $CC(N_i, N_{i'})(1 \leq i, i' \leq M)$ est égale au coût de transfert d'une page de données entre le nœuds N_i et $N'_{i'}$ dans une unité de temps. Nous supposons que cette matrice est symétrique.

3.1.2.1 Paramètres de fragmentation

Dans notre travail, la table de faits est fragmentée horizontalement en utilisant les prédicats de sélection définis sur les tables de dimension(fragmentation dérivée). Soit $\mathcal{F} = \{F_1, F_2, \ldots, F_{N_F}\}$ l'ensemble des fragments faits générés. Le nombre de fragments (N_F) est défini comme suit : $N_F = \prod_{i=1}^{g} m_j$,où m_j et g sont le nombre de fragments de la table D_j et le nombre de tables qui ont participé au processus de fragmentation [32].

Pour éviter l'explosion de ce nombre(N_F), l'Administrateur de L'Entrepôt de Données est

Nom de paramètre	Description
$\mathcal{DBC}$	cluster de base de données à M nœuds $N = N_1, \ldots, N_m$
M	nombre de nœuds de $\mathcal{DBC}$
S_m	capacité de stockage d'un nœud N_m
P_m	puissance de calcul d'un nœud N_m
PS	la taille d'une page système d'un nœuds de traitement
M	nombre de lnœuds

Tableau 3.3 – Les paramètres physiques

doté de la possibilité de choisir le nombre des fragments maximal W ($N_F \leq W$), nommé *seuil de fragmentation*, qui facilite la maintenance de la base de données. [10].

Pour n'importe quel schéma de fragmentation, nous définissons une matrice d'usage de fragments ($\mathcal{M}_U$) décrivant l'usage des fragments par les requêtes. Les lignes et les colonnes de cette matrice sont associées aux k requêtes de départ et les N fragments obtenus par le schéma de fragmentation SF respectivement. La valeur de l'élément de la ligne i et la colonne j détient pour valeur 1 si le fragment j est accédé par la requête i, sinon cette valeur est nulle.

Nom de paramètre	Description
m_j	le nombre de fragments de la table D_j
W	le seuil de fragmentation
N_F	le nombre des fragments de faits
$\|\|F_i\|\|$	la cardinalité de F_i: nombre de n-uplet stockés dans F_i
$Size(F_i)$	la taille de F_i
$\mathcal{M}_U$	matrice d'usage des fragments

Tableau 3.4 – Les paramètres de la fragmentation

3.1.2.2 Paramètres d'allocation

Le placement des données est le processus affectant les fragments générés par la fragmentation sur les nœuds d'une grappe de machines. L'allocation peut être soit redondante (les fragments sont répliqués sur les nœuds) ou non redondante (chaque fragment réside dans un et un seul nœud).

Pour faciliter la localisation de données, nous modélisons le schéma de placement des fragments sur les nœuds par une matrice nommée Matrice de Placement de Fragments ($\mathcal{M}_\mathcal{P}$). Comme son nom l'indique, cette matrice représente la présence des fragments sur les nœuds. Ses lignes et ses colonnes sont associées aux N_F fragments obtenus par le schéma de fragmentation de l'entrepôt de données et les M nœuds associés de la grappe de machines respectivement.

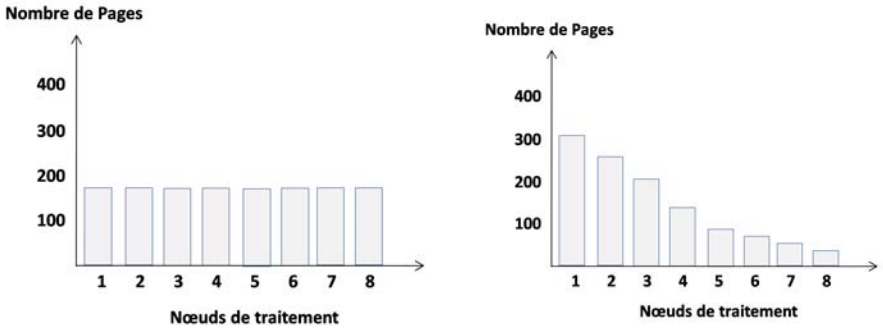
Chaque élément de la matrice $\mathcal{M}_\mathcal{P}$ s'écrit $\mathcal{M}_{\mathcal{P}ij}(1 \leq i \leq N_F; 1 \leq j \leq M)$. Il est binaire (0

ou 1) et défini par la variable de décision x_{ij}

$$x_{ij} = \begin{cases} 1 & si\ le\ fragment\ i\ est\ alloué\ sur\ le\ nœud\ j \\ 0 & sinon. \end{cases}$$

La taille de la portion de données stockée sur un nœud de traitement N_i est égale à la somme de la taille des fragments alloués sur N_i. Formellement

$$Taille(N_j) = \sum_{j=1}^{M} Size(F_i) \times \mathcal{M}_{\mathcal{P}ij}. \tag{3.1}$$



(a) (b)

Figure 3.2 – *Distribution des données uniforme vs. distribution des données non uniforme*

Comme illustré dans la figure 3.2, si les données sont distribuées uniformément sur les nœuds de traitement, la taille de chaque partition doit être égale à la taille de l'entrepôt de données divisée par le nombre des nœuds de la grappe.

$$\forall 1 \leq j \leq M : Taille(N_j) = \frac{\|ED\|}{M}. \tag{3.2}$$

Par contre, si la distribution de données est biaisée. la taille des données allouées au niveau de chaque nœuds de traitement est différente. Supposons que notre distribution biaisée suit le modèle de distribution *Zipf* [33] comme dans [135]. Pour un entrepôt de données de taille globale de $\|ED\|$ tuples, M nœuds de traitements et un facteur de *skew* α, la taille des données allouée au niveau de chaque nœud de traitement est

$$\forall 1 \leq j \leq M : Taille(N_j, \alpha) = \frac{\|ED\|}{j^{\alpha} \times \sum_{l=1}^{M} \frac{1}{l^{\alpha}}}. \tag{3.3}$$

Une fois l'allocation réalisée, les requêtes seront réécrites selon le schéma de fragmentation. L'ensemble des sous-requêtes générées sera alloué sur les nœuds de traitement. Pour cela, nous

définissons une matrice d'allocation des sous-requêtes. Cette matrice est nommée $\mathcal{MPSQ}$, chaque sous-requête est exécutée sur un et un seul nœud, ainsi, ses éléments sont binaires.

L'administrateur de l'entrepôt de données parallèle $\mathcal{DWA}$ tolère pour le placement de sous-requêtes sur les nœuds de traitement du $\mathcal{SN} - \mathcal{DBC}$ un déséquilibrage de charge avec un seuil de δ .

Nom de paramètre	Description
α	facteur de skew de placement de donénes
δ	degré de déséquilibrage de charge
$\mathcal{M}_P$	matrice de placemet des fragments
$\mathcal{MPSQ}$	matrice d'allocation des sous-requêtes
$Taille(N_j, \alpha)$	taille des données allouées sur N_j avec un skew de α

Tableau 3.5 – Les paramètres de placement

3.1.3 Politique de notre traitement parallèle

L'architecture *Shared Nothing* est composée principalement d'un nœud coordinateur relié aux nœuds de traitement par un réseau d'interconnexions à haut débit.

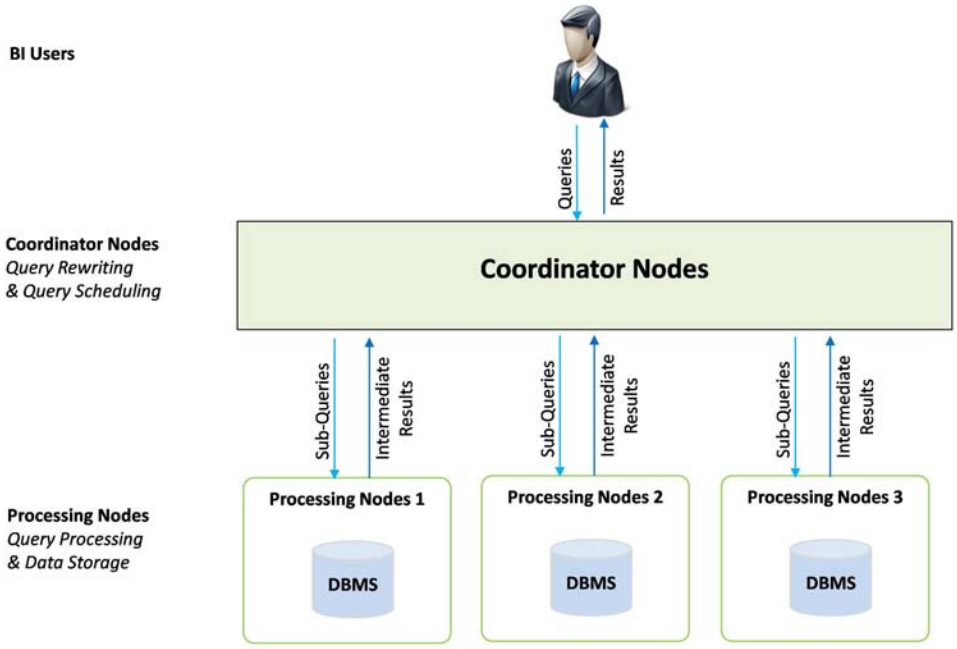


Figure 3.3 – *Paradigme de traitement parallèle*

Le nœud coordinateur (Coordinator Node, $\mathcal{CN}$), également appelé nœud de soumission, veille à ce que la charge de requêtes soit distribuée d'une manière équitable sur l'ensemble des nœuds. En effet, $\mathcal{CN}$ décompose la requête soumise Q_i en un ensemble de sous-requêtes et les

envoie aux nœuds de traitement. Si une situation de déséquilibrage de charge se présente, il redistribue la charge entre les nœuds de traitement.

Le nœud de soumission s'occupe également de la collecte des résultats partiels ainsi que de leur fusion pour avoir le résultat final de l'exécution de la requête Q_i. Son architecture modulaire, représentée dans la figure 3.4, est composée de trois modules : *réécriture de requête*, *ordonnancement de requêtes*, *collecte et fusion des résultats partiels*.

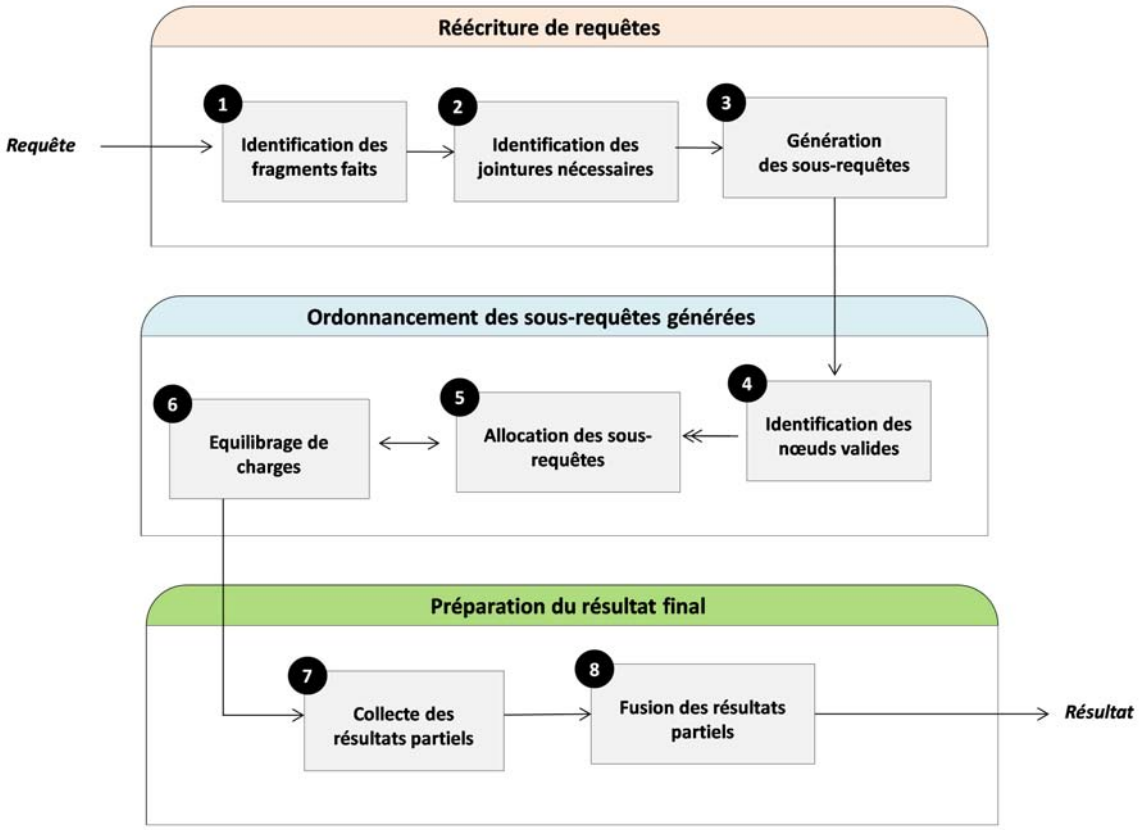


Figure 3.4 – *Architecture modulaire du nœud coordinateur*

Le nœud de traitement (Processing Node, $\mathcal{PN}_i(1 \leq i \leq M)$) est un hôte d'une instance Sql. Il est chargé du stockage des données et de l'exécution des sous-requêtes. Lorsqu'un nœud de traitement reçoit une charge de travail, il l'exécute d'abord sur ses données, ensuite, il transmet son résultat final au nœud de soumission (résultat partiel pour la requête en cours d'évaluation au niveau du système).

Dans ce qui suit, nous détaillons le processus de traitement d'une requête de jointure en étoile sur une grappe de machines sans partage.

3.1.3.1 Réécriture des requêtes

Ce module reçoit en entrée un schéma de fragmentation horizontale SF et une requête Q_i, il retourne un ensemble de sous-requêtes $\mathcal{SQ} = \{SQ_1, SQ_2, \ldots, SQ_G\}$. Autrement dit, chaque requête Q_i est réécrite en l'union des sous-requêtes de $\mathcal{SQ}$ définies sur les fragments de faits et

de dimension.

$$Q_i = \bigcup_{j=1}^{G} SQ_j. \tag{3.4}$$

La réécriture de Q_i consiste à déterminer initialement l'ensemble des fragments de fait appropriés à la requête en cours d'exécution, puis, à identifier les fragments de dimension qui doivent être joints aux fragments faits valides.

3.1.3.2 Identification des fragments faits

Avant de déterminer les équi-jointures entre les fragments des faits et de dimension, nous déterminons l'ensemble des fragments de faits concernés par la requête Q_i parmi les fragments de faits $\mathcal{F} = \{F_1, F_2, \ldots, F_{N_F}\}$ résultant du schéma de fragmentation SF. Un fragment F_j $(1 \leq j \leq N)$ est utilisé pour l'exécution d'une requête Q_i, *si et seulement si*, Q_i accède à au moins un tuple du fragment F_j [32].

A partir de l'ensemble des fragments des faits identifié, nous définissons la Matrice d'Usage des Fragments ($\mathcal{M}_\mathcal{U}$).

3.1.3.3 Identification du plan d'exécution optimal

Le plan d'exécution de chaque requête contient un nœud racine représentant le fragment fait et des nœuds feuilles représentant les fragments des tables de dimension. Pour optimiser l'exécution des sous-requêtes de jointure en étoile, une séquence optimale spécifiant l'ordre d'exécution des fragments de dimension doit être identifiée.

3.1.3.3.1 Identification des jointures nécessaires Certaines requêtes référencient des tables de dimension fragmentées, cela pré-calcule les jointures entre la table de faits et les tables de dimension. La requête est alors dite en *matching total* avec le schéma de fragmentation. D'autres référencient des tables de dimension qui ne sont pas fragmentées et leurs jointures avec la table des faits qui ne sont pas pré-calculées lors de la génération des fragments des faits. Cela, nécessite la jointure entre le fragment de fait et de dimension. La requête est dite alors en *matching partiel* avec le schéma de fragmentation. Deux exemples peuvent être considérés pour bien illustrer cette différence.

Soit un entrepôt de données fragmenté horizontalement. Les tables de dimension *Client*, *Produit* et *Temps* sont partitionnées selon les attributs *Ville*, *Catégorie* et *Temps* respectivement. Ainsi, la table des faits est fragmentée, en utilisant la fragmenentation dérivée, est partitionnée en huit fragments.

3.1.3.3.1.1 Scénario 1 : matching total. Supposons qu'une requête Q_2 qui calcule la somme des ventes effectuées par des clients *Algerois* et exprimée par

```
SELECT Sum(ventes)
FROM Ventes V, Client C
```

$Vente_1 : (Ville = Alg) \wedge (Cat = Be \vee Fi) \wedge (Mois = Ja \vee Fév \vee Ma)$
$Vente_2 : (Ville = Alg) \wedge (Cat = Be \vee Fi) \wedge (Mois = Av \vee Mai \vee Ju)$
$Vente_3 : (Ville = Alg) \wedge (Cat = Mu \vee Jo \vee Ja) \wedge (Mois = Ja \vee Fév \vee Ma)$
$Vente_4 : (Ville = Alg) \wedge (Cat = MuveeJo \vee Ja) \wedge (Mois = Av \vee Mai \vee Ju)$
$Vente_5 : (Ville = Pa \vee Po) \wedge (Cat = Be \vee Fi) \wedge (Mois = Ja \vee Fév \vee Ma)$
$Vente_6 : (Ville = Pa \vee Po) \wedge (Cat = Be \vee Fi) \wedge (Mois = Av \vee Mai \vee Ju)$
$Vente_7 : (Ville = Pa \vee Po) \wedge (Cat = Mu \vee Jo \vee Ja) \wedge (Mois = Ja \vee Fév \vee Ma)$
$Vente_8 : (Ville = Pa \vee Po) \wedge (Cat = Mu \vee Jo \vee Ja) \wedge (Mois = Av \vee Mai \vee Ju)$

```
WHERE V.CID=C.CID
AND C.Ville='Alger'.
```

Les fragments $Ventes_1$, $Ventes_2$, $Ventes_3$ et $Ventes_4$ sont valides pour Q_1 car ces quatre fragments regroupent toutes les ventes effectuées par les clients algérois. La jointure entre les quatre fragments et la table de dimension client est déjà pré-calculé, ainsi, aucune jointure n'est nécessaire. La réécriture de cette requête donne lui à quatre sous-requêtes SQ_1, SQ_2, SQ_3 et SQ_4 dont la syntaxe est la suivante.

```
SELECT Sum(ventes) FROM Ventes_1
UNION
SELECT Sum(ventes) FROM Ventes_2
UNION
SELECT Sum(ventes) FROM Ventes_3
UNION
SELECT Sum(ventes) FROM Ventes_4
```

La figure 3.5 illustre l'exécution parallèle de la requêtes qui résulte de l'union des quatre sous-requêtes identifiées.

3.1.3.3.1.2 Scénario 2 : matching partiel Considérons la requête Q_1 qui calcule la somme des ventes effectuées par des clients *Algerois* ayant acheté un produit au mois de *Mars*:

```
SELECT Sum(ventes)
FROM Ventes V, Client C, Temps T
WHERE V.CID=C.CID AND V.TID=T.TID
AND C.Ville='Alger' AND T.Mois='Mars'.
```

Les fragments de faits $Ventes_1$ et $Ventes_3$ sont valides pour l'exécution de Q_1, ils contiennent toutes les ventes effectuées par des clients algérois durant les mois de Janvier, Février et Mars or que Q_1 cherche les ventes effectuées par des clients algérois durant le mois de Mars seul. Ainsi, une jointure de $Ventes_1$ et $Ventes_3$ avec la table Client n'est pas nécessaire car $Ventes_1$ et $Ventes_3$ ne concernent que des clients algérois (cette jointure a été effectuée durant la construction de $Ventes_1$ et $Ventes_3$). Par conséquent seul le fragment $Temps_1$ sera joint pour sélectionner les tuples de $Ventes_1$ et $Ventes_3$ concernant le mois de Mars seulement. En résumé, nous obtenons deux sous-requêtes SQ_1 et SQ_2 dont leur syntaxe est la suivante:

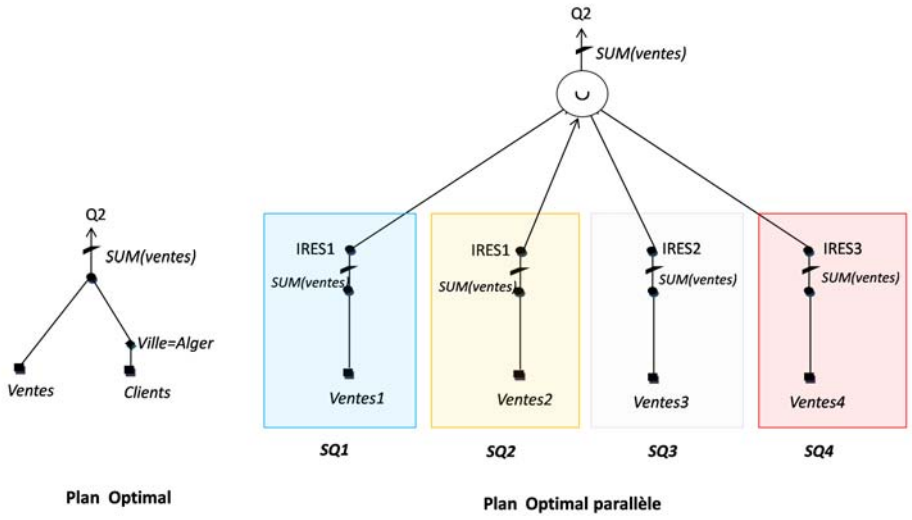


Figure 3.5 – *Plan d'exécution parallèle de la requêtes* Q_2

```
SELECT Sum(ventes)
FROM Ventes_1 V, Temps_1 T
WHERE V.CID=C.CID AND V.TID=T.TID
AND  T.Mois='Mars'
UNION
SELECT Sum(ventes)
FROM Ventes_3 V, Temps_1 T
WHERE V.CID=C.CID AND V.TID=T.TID
AND  T.Mois='Mars'.
```

La figure 3.6 décrit le plan d'exécution optimal parallèle obtenu après la réécriture de Q_1

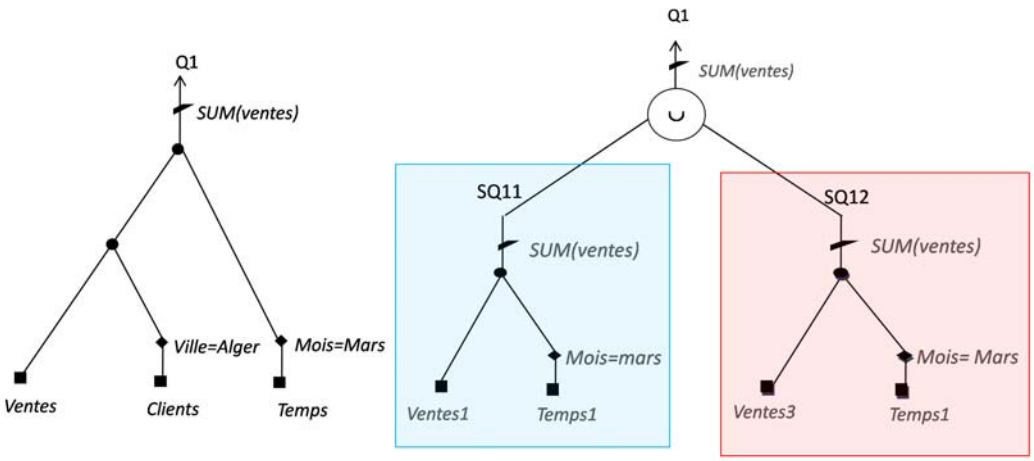


Figure 3.6 – *Plan d'exécution parallèle de la requêtes* Q_1

3.1.3.3.2 Ordre d'exécution des jointures .

Cette phase consiste à identifier l'ordre suivant lequel les tables seront jointes, incluant la table directrice (celle à partir de laquelle débute la jointure). Ce problème est appelé *problème de sélection d'ordre d'exécution des jointures des fragments* [132]. Il a été démontré NP-Complet [80].

Le défi est de déterminer l'ordre dans lequel les jointures entre les résultats partielles de la requête doivent être réalisées, tout en minimisant le temps d'exécution global de la requête. Les méthodes typiques pour ce problème impliquent l'exploration d'un espace de solution pour tenter de trouver des solutions à faible coût.

Plusieurs stratégies de sélection du meilleurs ordre d'exécution des jointures ont été proposées [85, 65, 32]. Dans la section 3.2.1, nous précisions la formule utilisée pour selectionner l'ordre adéquat d'exécution de nos jointures.

3.1.3.3.3 Méthodes d'accès .

En l'absence d'index de jointure d'autres méthodes d'accès sont utilisées. Parmi ces méthodes, nous trouvons la jointure imbriquée, la jointure par tri, la jointure par hachage,... etc. Nous utilisons la jointure par hachage. Ce choix s'est porté sur cette technique parce qu'elle est utilisée par plusieurs systèmes existants (e.g., Oracle, Informix). Les autres techniques peuvent également être utilisées dans cette thèse. La jointure par hachage se fait en deux phases [122]. Dans la première phase, les deux relations sont partitionnées suivant la même fonction de hachage appliquée aux attributs participant à la jointure. Dans la deuxième phase, les partitions en correspondance sont jointes.

3.1.3.4 Ordonnancement des sous-requêtes générées.

Le but de cette étape est de trouver l'emplacement judicieux des requêtes pour optimiser le coût d'exécution de la charge de requêtes et d'atteindre la haute performance. Dans cette optique, nous proposons un nouvel ordonnanceur de requêtes. Son architecture, illustrée dans la figure 3.7, est composée principalement de deux modules .

3.1.3.5 L'identification des nœuds valides

Elle dépend du schéma d'allocation. Un nœud $N_j (1 \leq j \leq M)$ est dit valide pour une requête Q_i, si et seulement s'il alloue au moins un fragment F_j valide pour la requête en question. Par exemple, *la requête* Q_1 *ne nécessite que le chargement des fragments* $Ventes_1$ *et* $Ventes_3$.

3.1.3.6 L'allocation des sous-requêtes

L'allocation des sous-requêtes est un problème important qui prouve la performance des applications exécutées dans un environnement parallèle. Le problème consiste à trouver le placement le plus judicieux des sous-requêtes sur les nœuds de la grappe de machines qui

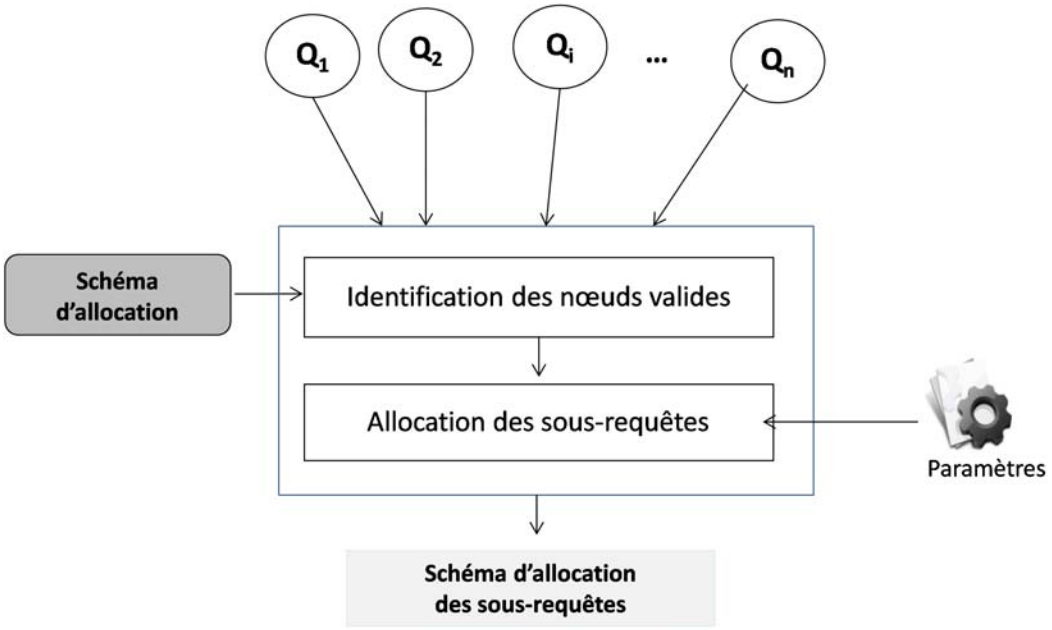


Figure 3.7 – *Ordonnanceur des requêtes*

assure un coût d'exécution de requêtes minimal et une distribution de charge équitable entre les nœuds de traitement. Notons que chaque sous-requête doit être placée sur un et un seul nœud de traitement.

3.1.3.6.1 Formalisation .

Nous formalisons le problème d'allocation des sous-requêtes comme un *problème d'optimisation à contraintes*. Soient

- un ensemble de fragments $\mathcal{F} = \{F_1, F_2, \ldots, F_N\}$, dont chaque fragment F_i, avec $1 \leq i \leq N_F$, est caractérisé par sa taille $Taille(F_i)$;
- un grappe de machine sans partage $\mathcal{SN} - \mathcal{DBC}$ à M nœuds $\mathcal{N} = \{N_1, N_2, \ldots, N_M\}$,
- un schéma d'allocation de l'ensemble des fragments $\mathcal{F}$ sur les nœuds de $\mathcal{SN} - \mathcal{DBC}$. Ce schéma est représenté par la matrice $\mathcal{M}_{\mathcal{P}}$;
- une charge de requêtes de jointure en étoile $\mathcal{Q} = \{Q_1, Q_2, \ldots, Q_K\}$ exécutée sur la grappe $\mathcal{SN} - \mathcal{DBC}$, chaque requête Q_k possèdant une fréquence d'accès f_l;
- *une contrainte d'équilibrage de charge* δ, représentant le degré de déséquilibrage de charge que l'administrateur de l'entrepôt de données parallèle $\mathcal{DWA}$ tolère pour le placement de sous-requêtes sur les nœuds de traitement du $\mathcal{SN} - \mathcal{DBC}$.

L'objectif est de déterminer l'état de la fonction *EstAllouer* pour minimiser le coût d'exécution des requêtes de la charge $\mathcal{Q}$ et satisfaire la contrainte δ. La fonction *EstAllouer* est définie par

$$EstAllouer(Q_i, N_j) = \begin{cases} 1 & si\ la\ Q_i\ est\ allouée\ sur\ le\ nœud\ N_j \\ 0 & sinon \end{cases}$$

Nous rappelons que le schéma de placement global des sous-requêtes par une matrice notée $\mathcal{MPSQ}$. Cette matrice représente la présence des sous-requête sur les nœuds. Les lignes et les colonnes de cette matrice sont associées aux L sous-requêtes obtenus par la réécriture d'une requêtes Q_i selon le schéma de fragmentation et les M nœuds. La valeur de chaque élément de cette matrice est définie selon la fonction $EstAllouer(SQ_i, N_j) avec 1 \leq i \leq L\ et\ 1 \leq j \leq M$.

3.1.3.6.2 Algorithmes proposés .

L'allocation de requêtes dépend de l'allocation des données qui peut être redondante (avec réplication) ou non redondante (sans réplication).

Dans le cas d'une allocation non redondante où chaque fragment est stocké sur un seul les nœuds de traitement, chaque sous-requête peut être placée sur un seul nœud de traitement. Ainsi, le coordinateur utilise son module d'équilibrage de charge pour améliorer la performance du système.

Dans le cas d'une allocation redondante, chaque fragment est stocké sur au moins deux nœuds. Ainsi, chaque sous-requête a la possibilité d'être allouée sur plus d'un nœud; il faut trouver l'emplacement adéquat des sous-requêtes pour minimiser le coût d'exécution de la charge de requête et maximiser l'utilisation de l'usage des données disponibles.

Dans notre travail, nous assimilons le problème d'allocation des sous-requêtes à un dual du problème de remplissage *(Dual Bin Packing Problem (DBPP))*. En effet, le problème Dual Bin Packing ($\mathcal{DBP}$) partitionne les n objets dont chacun est caractérisé par un poids $P_i (1 \leq i \leq n)$ en $\mathcal{M}$ sous-ensembles respectant la contrainte de capacité de stockage des bins $\mathcal{C} = \frac{1}{M} \sum_{i=1}^{N} P_i$.

L'objectif du $\mathcal{DBP}$ est de minimiser la somme des charges des bins, sachant que la taille d'un bin est définie par la somme des tailles des objets dans le bin. Le problème d'ordonnancement des n sous-requêtes sur les $\mathcal{M}$ nœuds de la grappe de machines consiste à minimiser le coût d'exécution totale des requêtes. Chaque sous-requête peut être considérée comme un item caractérisé par le temps d'exécution T_i de la requête Q_i et chaque nœud sera considéré comme un bin de capacité C:

$$C = \frac{1}{j^{\delta} \times \sum_{j=1}^{M} \frac{1}{j^{\delta}}} \times \sum_{i=1}^{N} T_i \tag{3.5}$$

$\mathcal{DBPP}$ a été montré comme un problème *NP-Complet* [87], il a pour but de minimiser la capacité de M bins identiques pour qu'on puisse emballer tous les éléments dans M bins sans violer la contrainte de capacité [4]. Dans la littérature, des procédures déterministes, pour construire les solutions faisables, ont été proposée pour ce problème, telles que [4]:

- *Dual Best-Fit Decreasing* (DBFD)
- *Dual Worst-Fit Decreasing* (DWFD)
- *Dual Best 3-Fit Decreasing* (DB3FD)
- *Dual Worst-Sum-Fit Decreasing*(DWSFD)
- *Longest Processing Time* (LPT)

Pour produire une solution quasi-optimale pour notre problème, nous avons opté pour la proposition d'un algorithme glouton (greedy algorithm) qui génère une solution réalisable. nous l'appelons $\mathcal{SQ} - \mathcal{DPB}$.

Dans le cas où la solution choisie n'est pas réalisable, il faut utiliser une méthode d'équilibrage de charge pour essayer de minimiser les violations de la capacité. Les solutions voisines d'une solution donnée peuvent être déterminées en échangeant deux éléments dans deux bins dont un bins est violé. Pour déterminer si une solution est meilleure qu'une autre, on peut se baser sur différents critères:

1. la solution avec plus de bins qui sont pleins est la solution meilleure,
2. la solution avec moins de bins qui sont violés est la solution meilleure. [4]

La procédure se termine lorsque la solution finale est réalisable pour le problème *Bin Packing*.

Notre algorithme agrandit la taille du bin pour permettre des solutions infaisables initialement. Nous commençons avec une solution initiale qui a des proportions similaires à celles de grandes et petites sous-requêtes dans les bins. Voici les principales étapes de $\mathcal{SQ} - \mathcal{DPB}$:

1. fixer le nombre de bins au nombre des nœuds valides,
2. calculer le coût d'exécution de chaque sous-requête,
3. trier la sous-requête en fonction du coût d'exécution dans un ordre décroissant,
4. calculer la charge moyenne des bins,
5. fixer la taille de chaque bin à une valeur supérieure à la charge moyenne des bins pour permettre des solutions infaisables,
6. construire une solution initiale avec une heuristique décroissante de type *Dual Best-Fit Decreasing* ,
7. utilisez la fonction objective qui minimise la somme des carrés des charges bin,
8. si la valeur obtenue est inférieure ou égale au seuil de déséquilibrage de charge. Aller à terminer,
9. sinon, utiliser l'algorithme d'échange de base pour échanger des éléments entre toutes les paires de bacs lors de la recherche locale pour éliminer les violations des capacités,
10. arrêter l'algorithme lorsque toutes les violations des capacités ont été enlevées. Si la solution est encore impossible après un nombre fixe d'itérations, augmenter la limite inférieure et construire une nouvelle solution initiale.

L'algorithme 1 montre les grandes lignes de la procédure. $\mathcal{SQ} - \mathcal{DPB}$ est un algorithme polynomial d'ordre $O(N_F)$.

Une fois les sous-requêtes sont placées sur le nœuds de traitement, le coordinateur utilise son module d'équilibrage de charge pour améliorer la performance du système qui vise à uniformiser la distribution de charge entre les nœuds de traitement en faisant recours à la migration de données entre les nœuds de traitement.

Algorithm 1 *Allocation des Requêtes* ($\mathcal{SQ} - \mathcal{DPB}$)

Entrées: M nœuds, Q_j requête
Sorties: MPSQ: Matrice de Placement des sous-requêtes
Soit `ListFrag` la liste des fragments valides pour la requêtesQ_j.
Soit `NumberFrag` le nombre des fragments dans `ListFrag`;
Soit `NumberValideNode` le nombre des nœuds valides pour les fragments de `ListFrag`;
Soit `ListSubQuery` la liste des sous- requêtes:
Estimer $SizeQ$ le nombre d'E/S nécessaires pour l'exécution de Q_j;
calculer la charge moyenne d'un nœud ;

$$LB = \frac{1}{\sum_{j=1}^{NumberValideNode} \frac{1}{j^{\delta}}} \times SizeQ \tag{3.6}$$

Trier la liste des fragments en ordre décroissant selon le taille des fragments;
Initailiser j à zéro
for $i = 1$ to $NumberFrag$ de la requête Q_j **do**
    Récupérer `ListNode` la liste des nœuds valide pour le $i^{ème}$ fragment de `ListFrag`.
    Calculer la charge de travail affecté à chaque nœud de la liste `ListNode` ;
    Assigner Q_j au nœud qui à la plus grande capacité pour traiter F_i;
end for

Dans ce qui suit, nous proposons notre stratégie d'équilibrage de charge [22]. Ce problème est un enjeu important pour atteindre une haute performance d'un entrepôt de données parallèle. Une redistribution des fragments des nœuds surchargés dénotés par $NSUR$ sur les nœuds sous chargés dénotés par $NSOUS$ est nécessaire.

Cette redistribution peut être formalisée comme suit : soit la matrice d'allocation des fragments $\mathcal{M}_{\mathcal{P}}$, M nœuds de traitement de la grappe $\mathcal{SN} - \mathcal{DBC}$ décrit par sa matrice de communication $\mathcal{CC}$. Le problème de répartition de la charge consiste à offrir un système équilibré assurant un coût de communication minimal. Pour le résoudre, quelques définitions s'imposent.

Définition 2. *Le niveau de chargement d'un nœud correspond au nombre de requêtes à exécuter sur ce nœud.*

Définition 3. *La moyenne de chargement correspond au nombre de fragments valides pour la requête sur le nombre des nœuds de la machine parallèle.*

Définition 4. *Un système est dit équilibré si tous les nœuds détiennent la même charge de travail.*

Définition 5. *Un nœud est dit surchargé si son niveau de chargement est supérieur à la moyenne de chargement.*

Définition 6. *Un nœud est dit normalement chargé si son niveau de chargement est égal à la moyenne de chargement.*

Définition 7. *Un nœud est dit sous chargé si son niveau de chargement est inférieur à la moyenne de chargement.*

Ces définitions nous permettent de classer les nœuds surchargés et sous chargés. Une fois classés, nous effectuons la migration de fragments de `NSUR` vers `NSOUS` à l'aide d'un algorithme nommé *Algo_Migration_Dynam*. Chaque nœud N_i de `NSUR` est associé à un poids `PSUR(N_i)` représentant le taille d'extra-requêtes qui le surcharge.

La $PSUR(N_i)$ est calculée comme suit:

$$PSUR(N_i) = niveau\ de\ chargement(N_i) - moyenne\ de\ chargement \tag{3.7}$$

D'une manière identique, à chaque nœud de `NSOUS` est associé un poids `PSOUS(N_j)` représentant le nombre de requêtes qu'il peut encore recevoir. `PSOUS(N_j)` est calculée comme suit :

$$PSOUS(N_i) = moyenne\ de\ chargement - niveau\ de\ chargement(N_j) \tag{3.8}$$

Maintenant nous avons tous les ingrédients pour proposer notre algorithme équilibrant les charges des nœuds.

Algorithm 2 *Algorithme de répartition de charges(Algo_Migration_Dynam)*

1: **Entrées**: `NSUR`, `NSOUS`, `MP`, `M`
2: **Sorties**: `MPSQ` : Matrice de Placement des Sous-Requêtes
3: **Fonctions**: *isSysLoaded()* : fonction booléenne, retourne vrai si le système est équilibré.
4: Rechercher le nœud $N_j \in$ `NSUR` ayant la plus grande priorité ;
5: Rechercher le nœud $N_k \in$ `NSOUS` qui minimise le coût de redistribution CT entre N_k et N_j
6: Transférer le(s) fragment(s) F alloué(s) sur N_j à N_k en mettant à jour la matrice `MPSQ`;
7: Calculer la priorité `p` de N_j ;
8: **if** (p=0) **then**
9: Supprimer N_j de `NSUR` et N_k de `NSOUS` ; aller à 13
10: **else**
11: Supprimer N_k de `NSOUS` et aller à 5
12: **end if**
13: **if** (isSysLoaded()) **then**
14: Aller à Fin
15: **else**
16: Aller à 1
17: **end if**
18: Fin

3.1.3.7 La collecte et la fusion des résultats

Une fois qu'un nœud de traitement achève la charge qui lui a été affectée, il transmet son résultat final au nœud de soumission. sachant que la communication entre les nœuds dépend

de la topologie du réseau d'interconnexion (en bus, en étoile, en anneau, réseau maillé). Après la collecte de tous les résultats partiels, le coordinateur s'occupe de leur fusion en appliquant exactement la requête initiale, pour regrouper les résultats partiels.

3.2 La définition de notre modèle de coût

Pour définir notre modèle de coût, nous présentons cinq hypothèses :
- les tables de dimensions sont répliquées sur les nœuds de la machine parallèle et résident dans leurs mémoires centrales.
- les conditions de sélection sont toujours descendues sur l'arbre syntaxique comme dans [92].
- la distribution biaisée suit le modèle de distribution *Zipf* [33] comme dans [135].
- la démarche de fragmentation horizontale est basée sur les prédicats comme dans [32].
- la communication entre les nœuds de la plateforme est symétrique.

Maintenant nous avons tous les ingrédients pour estimer le coût de traitement parallèle (CTP) d'une requête OLAP. Ce dernier comprend deux coûts : coût de traitement (CT) et coût de communication (CC). Les deux coûts sont exprimés en termes de pages.

$$CTP = CT + CC \tag{3.9}$$

3.2.1 Coût de traitement

Le coût de traitement dans un environnent parallèle est égal au temps d'exécution maximale d'un nœud. Le temps de traitement local au niveau de chaque nœud est calculé après l'élaboration du plan d'exécution optimal qui nécessite la définition d'ordre d'exécution des jointures.

Pour choisir le meilleur ordre d'exécution de jointure entre le fragment fait et les fragments de dimension, nous adoptons la stratégie *"Minimum share"* qui consiste à définir un seuil nommé $share(F, A_l, D)$. $Share(F, A_k, D)$ calcule le nombre d'instances du fragment de la table des faits F_i pour chaque instance utilisée du fragment D_{kj} de la table de dimension D_k à travers l'attribut A_l , clé étrangère dans le fragment F_i de la table des faits F. Le $share(F, D)$ est la moyenne des $share$ (F, A_l, D).

$$\|F_i \ltimes D_j\| = share(F_i, A_l) \times \|D_j\| . \tag{3.10}$$

Nous nous intéressons à la taille des résultats intermédiaire, pour éviter leur stockage sur le disque, ce qui peut augmenter le coût d'E/S.

Une fois que le plan d'exécution est identifié, les fragments sont chargés en mémoire pour le traitement. Chaque nœud soumet les résultats partiels des requêtes au nœud de soumission

qui s'occupe de la fusion des requêtes.

De plus, le temps de traitement d'une exécution parallèle est égal au temps de réponse du processeur le plus chargé. En conséquence, le coût de traitement est

$$\sum_{l=1}^{K} f_l \times \max_{1 \leq m \leq M} \left\{ \sum_{i=0}^{N_F} \frac{\mathcal{M}_{\mathcal{U}}[l][i] \times \mathcal{MPSQ}[i][m] \times Taille(F_i)}{P_m} \right\}, \tag{3.11}$$

3.2.2 Le coût de communication

Le coût de communication représente le coût nécessaire pour transmettre des données (résultats intermédiaires ou fragments) d'un nœud N_i vers un nœud N_j. Comme pour les disques, l'unité de mesure du coût de communication est la *page*. Ainsi, le coût de communication d'une donnée D est calculé en multipliant le nombre de pages à envoyer par le coût unitaire de communication. Notons que la taille de la table doit être divisée par la taille de la page afin de calculer le nombre de pages envoyées.

$$\sum_{l=1}^{K} f_l \times \sum_{g=1}^{G_l} \sum_{i=1}^{M} \sum_{j=1}^{M} CC[i][j] \times (MPSQ[g][i] \times (1 - M_{PF}[k][j])) \times \frac{Taille(F_k)}{PS} \tag{3.12}$$

où G_l représente le nombre de sous-requêtes de Q_l

En résumé, le coût d'exécution d'une charge de requête sur un cluster de machine est donné par la formule suivante

$$\sum_{l=1}^{K} f_l \times \Big(\max_{1 \leq m \leq M} \left\{ \sum_{i=0}^{N_F} \frac{\mathcal{M}_{\mathcal{U}}[l][i] \times \mathcal{M}_{MPSQ}[i][m] \times Taille(F_i)}{P_m} \right\} +$$

$$\sum_{g=1}^{G_l} \sum_{i=1}^{M} \sum_{j=1}^{M} CC[i][j] \times (MPSQ[g][i] \times (1 - M_{PF}[k][j])) \times \frac{Taille(F_k)}{PS} \Big) \tag{3.13}$$

Conclusion

Dans ce chapitre, nous avons d'abord montré l'intérêt et le rôle des modèles de coût dans l'optimisation, la sélection des structures d'optimisation et la phase de déploiement des bases/entrepôts de données. Nous avons également présenté une démarche modulaire pour définir l'ensemble des paramètres d'entrées de notre modèle de coût, où le sac des paramètres est partitionné en groupes dont chacun est associé à une étape particulière de la phase de déploiement. Cette démarche permet de rendre le modèle de coût *plus flexible et extensible (en ajoutant d'autres couches comme les paramètres liès auc entrées-sorties).* Pour les paramètres calculés, nous avons donné l'ensemble des formules permettant de les calculer en considérant un grappe de base de données *hétérogène.*

Une politique d'exécution d'une requête sur cette grappe est détaillée. Elle comprend la phase de réécriture de requêtes, l'identification des fragments des faits (sachant que les tables

de dimension sont stockées en mémoire centrale de chaque nœud), l'ordonnancement des sous requêtes issues d'une requête globale et l'identification des nœuds pertinents pour exécuter les sous requêtes. L'idée sous-jacente à cette phase est d'allouer les requêtes sur les nœuds de la grappes. Cette phase est formalisée comme un problème d'optimisation. Nous avons montré que ce problème soit dual du problème de remplissage (*Dual Bin Packing Problem*). Finalement, nous avons donné l'ensemble de formules de notre modèle de coût. L'intérêt de notre modèle est sa possibilité de quantifier n'importe quelle solution de déploiement.

Notre modèle de coût sera exploité par l'ensemble des algorithmes (fragmentation, allocation, réplication) qui seront développés dans les chapitres suivants.

Chapitre 4

Notre approche pas à pas de déploiement

"There is no one giant step that does it. It's a lot of little steps".
Peter A. Cohen(1946-)

Dans les chapitres précédents, nous avons montré la nature séquentielle des travaux existants sur le déploiement parallèle des entrepôts de données. Ces travaux ignorent l'interaction entre les différentes phases de déploiement. L'originalité de notre travail est qu'il propose de composer l'ensemble des phases. Nous sommes conscients que la composition de toutes les phases est une tâche difficile. Pour réduire cette complexité, nous proposons *une approche pas à pas* qui consiste d'abord à composer les deux premières phases à savoir la fragmentation et l'allocation, ensuite les deux avec la réplication. Pour chaque composition, un ensemble d'algorithmes est proposé. La deuxième particularité de notre travail est le fait qu'elle considère une grappe de bases de données composée de nœuds *hétérogènes*, contrairement aux approches existantes qui considèrent des nœuds homogènes.

Dans ce chapitre, nous détaillons notre approche de composition pas à pas. D'abord, nous décrivons les différentes architectures de déploiement d'un EDP. Ensuite notre démarche, appelée, $\mathcal{F}\&\mathcal{A}$ combinant la fragmentation et l'allocation. Une formalisation du problème joint de la fragmentation et d'allocation est d'abord présenté ensuite deux algorithmes de résolution sont décrits : un algorithme hill climbing et un algorithme génétique. Finalement, l'approche $\mathcal{F}\&\mathcal{A}\&\mathcal{R}$ combinant la fragmentation, l'allocation et la réplication est proposée. Elle suit la même démarche, formalisation et la proposition des algorithmes de sa résolution. Un algorithme basé sur la logique floue est présenté pour le processus d'allocation avec réplication.

4.1 Différentes architectures de déploiement d'un EDP

Le problème de déploiement d'un EDP modélisé par un schéma en étoile sur une grappe de bases de données consiste à fragmenter la table de faits F en N_F fragments de faits et à allouer les fragments générés ainsi que leurs copies de réplication sur les différents nœuds de la grappe sous un ensemble de contraintes à satisfaire. Ainsi, le problème de conception d'un EDP est principalement lié au trois problèmes : fragmentation, allocation et réplication. Chacun de ces problèmes est NP-Complet. La figure 4.1 représente les différentes architectures possibles des

méthodes de résolution du problème de déploiement d'un EDP.

Dans la figure 4.1 *(a)*, l'idée de base de cette architecture consiste à fragmenter d'abord l'EDP en utilisant n'importe quel algorithme de partitionnement, puis à allouer les fragments générés sur les nœuds de la grappe au moyen d'un algorithme d'allocation et enfin, à définir le schéma de réplication des fragments à l'aide de n'importe quel algorithme de réplication. Chacun des ces algorithmes (de partitionnement, d'allocation et de réplication) a son propre modèle de coût. Le principal avantage provenant de ces méthodes traditionnelles est le fait qu'elles sont applicables à un grand nombre d'environnements distribués. Contrairement à cela, leur principal inconvénient est le fait qu'elles négligent l'interdépendance entre le partitionnement, l'allocation et la réplication.

Dans la figure 4.1 *(b)*, l'idée consiste à diviser d'abord l'EDP en utilisant n'importe quel algorithme de partitionnement. Ensuite, les schémas d'allocation et de réplication des fragments générés sont déterminés en même temps. Le principal avantage de cette architecture est la prise en considération de l'interdépendance entre l'allocation et la réplication, qui sont étroitement liées. Par contre, le principal inconvénient est le fait que l'on néglige l'interdépendance entre le partitionnement des données et l'allocation de fragment..

Dans la figure 4.1 *(c)*, l'idée de base consiste à partitionner horizontalement l'entrepôt de données en un ensemble de fragments et à les allouer sur les nœuds de la grappe durant la même phase de conception. Ensuite, le schéma de réplication est déterminé en utilisant n'importe quel algorithme de réplication. L'avantage de cette architecture est le fait de déterminer conjointement les schémas de fragmentation et d'allocation. Cependant, l'inconvénient est la négligence de la dépendance étroite entre les processus d'allocation et de réplication.

Dans la figure 4.1 *(d)*, l'idée est la combinaison de la fragmentation, de l'allocation et de la réplication en un seul processus unifié. Une telle architecture est adaptée pour la conception d'EDP à partir du zéro. Elle a les avantages et les inconvénients de la conception conjointe.

Les architectures 4.1(c) et 4.1(d) sont quasi-équivalentes. la seule différence est au niveau du type de l'allocation de données. Ainsi, nous allons élaborer et mettre en œuvre les deux architectures pour les raisons suivantes : le processus de fragmentation est le cœur du processus de conception d'un EDP. Ainsi, la qualité de la conception de l'EDP dépend fortement de la qualité du processus de fragmentation. De plus, le partitionnement des données (DP), l'allocation de données (DA) et la réplication de données (RD) sont des problèmes importants pour la conception des EDPs. Ces problèmes sont étroitement liés. En général, il n'est pas possible de déterminer le schéma de fragmentation et l'allocation optimale en résolvant les deux problèmes de manière indépendante, car ils sont interdépendants. La phase d'allocation de données doit décider si des fragments seront répliqués ou pas. Autrement dit, au moment de la sélection du schéma de fragmentation, la décision du schéma d'allocation est faite. L'idée de base de notre approche consiste à déterminer la qualité du schéma de fragmentation généré selon le schéma d'allocation de données. L'allocation en elle-même peut être combinée à la réplication de données.

Les sections suivantes sont consacrées à présenter notre approche de déploiement d'un entrepôt de données parallèle, où la fragmentation et l'allocation se font simultanément.

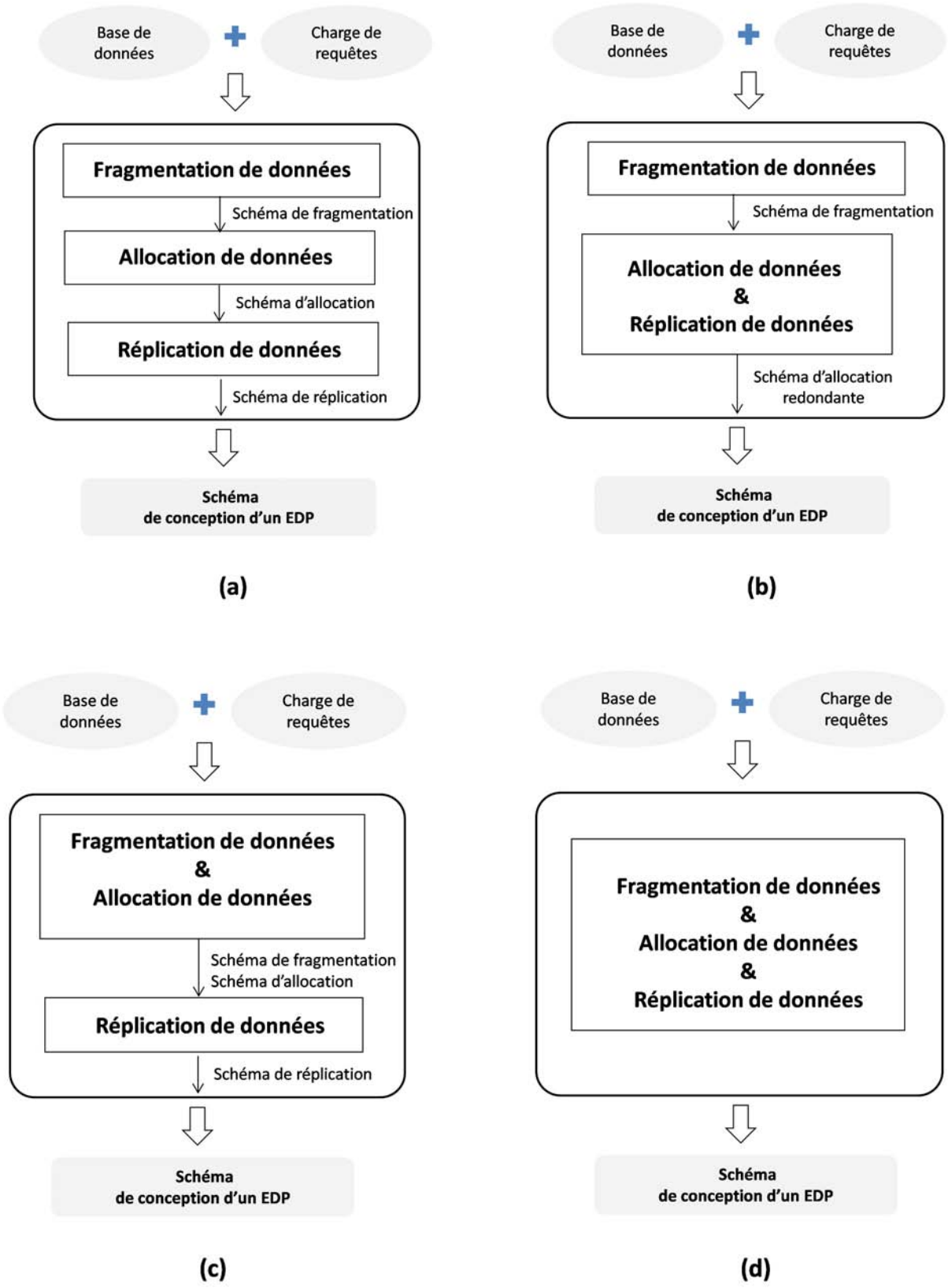


Figure 4.1 – *Methodologies de conception d'un EDP*

4.2 $\mathcal{F}\&\mathcal{A}$: une démarche conjointe de fragmentation et d'allocation

L'approche $\mathcal{F}\&\mathcal{A}$ consiste à partitionner l'entrepôt de données et à allouer les fragments générés sur les nœuds de la grappe de machines simultanément, Il nous faut donc une procédure de fragmentation et une autre d'allocation. Comme les deux problèmes sont connus *NP-complet* [125, 89, 14], les implémentations actuelles proposent très souvent la réalisation de *solutions sous-optimales* afin de réduire la complexité de calcul.

Afin de traiter le problème de conception d'EDP sur les grappes de bases de données, deux grandes classes de méthodes sont possibles: *la conception itérative* et des *la conception combinée*.

Les méthodes de conception itérative ont été proposées dans le cadre des bases de données distribuées et parallèles traditionnelles. L'idée sous-jacente consiste à fragmenter l'EDP en utilisant *un algorithme de partitionnement*, puis à allouer les fragments générés au moyen d' *un algorithme d'allocation*. Généralement, chaque algorithme de partitionnement et d'allocation a son propre modèle de coût. Le principal avantage de ces méthodes traditionnelles est qu'elles sont facilement applicables à un grand nombre d'environnements parallèles et distribués dans des environnements hétérogènes (par exemple, *les bases de données Peer-to-Peer*). Leur principal inconvénient est qu'elles négligent l'interdépendance entre le partitionnement des données et l'allocation de fragment. En effet, la fragmentation et l'allocation prennent en compte les exigences d'accès aux données et visent à obtenir, autant que possible, l'accès minimal aux références de données locales. Ainsi, il n'est pas possible de déterminer la fragmentation et l'allocation optimale qui résout les deux problèmes indépendamment, car ils sont interdépendants.

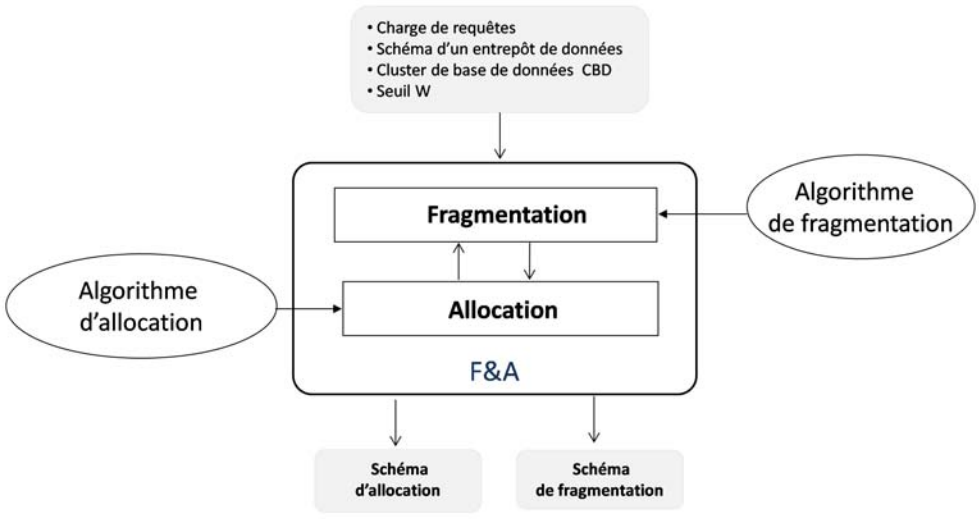


Figure 4.2 – Principe de l'approche $\mathcal{F}\&\mathcal{A}$

Pour dépasser les limitations découlant de ces méthodes, la méthode $\mathcal{F}\&\mathcal{A}$ *traite conjointement les problèmes de fragmentation et d'allocation.* Plus précisément, durant la phase de fragmentation, $\mathcal{F}\&\mathcal{A}$ exploite deux meta-heuristiques, l'algorithme Hill Climbing (HC) [55] et l'Algorithme Générique (GA) [78], que nous adaptons. Lors de la phase d'allocation, $\mathcal{F}$ $\&\mathcal{A}$ introduit une *formalisation basée sur les matrices* qui est capable de capturer les interactions entre les fragments, les requêtes d'entrée et les caractéristiques des nœuds de la grappe de machines (c'est-à-dire la puissance de traitement et la capacité de stockage), et d'effectuer l'allocation des tâches selon le schéma d'allocation résultant. Comme notre approche d'allocation est un algorithme basé sur les affinités, nous l'appelons $\mathcal{F}\&\mathcal{A}$-`ALLOC`. En outre, contrairement à l'approche itérative qui utilise *deux différents modèles de coûts* pour effectuer la fragmentation et l'allocation séparément, $\mathcal{F}\&\mathcal{A}$ fait usage d'un *seul modèle de coût*, (voir chapitre 3), qui surveille le *schéma de fragmentation généré*. Ce schéma "est utile" pour le *processus d'allocation réel*.

Dans ce qui suit, nous présentons une formalisation de notre approche de conception comme un problème d'optimisation à contraintes. Ensuite, nous présentons un algorithme de fragmentation et d'allocation pour le résoudre.

4.2.1 Formalisation du problème

Le problème de la conception d'un entrepôt de données parallèle sur une grappe de bases de données hétérogène peut être décrit comme *un problème d'optimisation à contraintes.*

Etant donné une charge de requêtes de jointure en étoile $\mathcal{Q}$, un schéma en étoile d'un entrepôt de données $\mathcal{DWS}$ et une grappe de base de données $\mathcal{DBC}$, la sélection de la conception la plus approprié pour $\mathcal{DWS}$ consiste à fragmenter la table de faits F de $\mathcal{DWS}$ en N_F fragments et de les allouer sur les différents nœuds de la grappe de machine $\mathcal{DBC}$ pour réduire le coût d'exécution de requêtes de $\mathcal{Q}$ sur $\mathcal{DBC}$ et satisfaire les contraintes de stockage et de maintenance.

Formellement, soient

- un entrepôt schéma de données $\mathcal{DWS}$ composé d'une table de faits $\mathcal{F}$ et d tables de dimensions $\mathcal{D} = \{D_1, D_2, \ldots, D_d\}$ (comme dans [64, 97], nous supposons que toutes les tables de dimension sont répliquées sur les nœuds de la grappe de bases de données et résident dans leurs mémoires centrales);
- une grappe de bases de données $\mathcal{DBC}$ à M nœuds $\mathcal{N} = \{N_1, N_2, \ldots, N_M\}$, chaque nœud N_m, avec $1 \leq m \leq M$, ayant une *capacité de stockage* S_m et une *puissance de traitement* P_M, exprimée en nombre d'opérations traitées dans une unité de temps;
- une charge de requêtes de jointure en étoile $\mathcal{Q} = \{Q_1, Q_2, \ldots, Q_K\}$ qui sera exécutée sur les M de $\mathcal{DBC}$, chaque requête Q_l avec $1 \leq l \leq L$, étant caractérisée par une *fréquence d'accès* f_l.

Nous fragmentons la table des faits en N_F fragments $\mathcal{FF} = \{F_1, F_2, \ldots, F_{N_F}\}$ et nous les

allouons simultanément afin de réduire le coût d'exécution de requêtes sur les M nœuds de la grappe $\mathcal{DBC}$:

$$Minimiser \sum_{k=1}^{K} f_k \times Cost(Q_k). \tag{4.1}$$

Nous avons trois contraintes.

– *Allocation non redondante. Une seule copie* de chaque fragment $F_i (1 \leq i \leq NF)$ est stockée sur tous les nœuds de la grappe $\mathcal{DBC}$. Pour illustrer cette contrainte, nous introduisons une *variable de décision binaire* x_{ij} telle que $x_{im} = 1$ si le fragment F_i est alloué sur le noeud N_m de $\mathcal{N}$ et $x_{im} = 0$ sinon. Ainsi, la contrainte d'allocation non redondante peut être formellement modélisée comme suit:

$$\forall 1 \leq i \leq N_F : \sum_{m=1}^{M} x_{im} = 1. \tag{4.2}$$

– *Capacité de stockage.* L'espace disponible au niveau de chaque nœud N_m de $\mathcal{N}$ est borné par sa capacité de stockage S_m. En conséquence, l'ensemble des fragments stockés dans N_m ne peut pas excéder S_m. Formellement, la contrainte de capacité de stockage s'exprime comme suit :

$$\forall 1 \leq i \leq N_F : \sum_{m=1}^{M} x_{im} \times Taille(F_i) \leq S_m. \tag{4.3}$$

Nous rappelons que la $Taille(F_i)$ désigne la taille du fragment F_i, et qu'elle est calculée comme suit:

$$Taille(F_i) = \left\lceil \frac{||F_i|| \times T_l}{P_S} \right\rceil, \tag{4.4}$$

où: $\|F_i\|$ désigne le cardinal de F_i(le nombre de lignes dans F_i), T_l désigne la longueur de chaque tuple de la table des faits F et P_S désigne la taille de la page d'un nœud N_m.

La contrainte de stockage peut, donc, s'écrire:

$$\forall 1 \leq i \leq N_F \sum_{m=1}^{M} x_{im} \times \frac{\|F_i\| \times T_l}{P_S} \leq S_m. \tag{4.5}$$

– *Seuil de fragmentation.* Pour éviter l'explosion du nombre de fragments, nous définissions un seuil de fragmentation, noté W qui présente le nombre de fragments autorisées par

le concepteur pour sa procédure d'allocation. En d'autres termes, W joue le rôle d'une borne supérieure sur le nombre de fragments autorisés dans l'EDP. La contrainte de maintenance W peut être formellement modélisée comme suit:

$$N_F = \prod_{j=1}^{d} M_j \leq W. \tag{4.6}$$

En résumé, notre principal problème de conception d'un EDP sur une grappe de base de données hétérogènes peut être formellement modélisé ainsi:

$$\begin{cases} Minimiser \sum_{k=1}^{L} f_k \times Cost(Q_k) \\ \\ Sous: \\ \\ \forall i \sum_{m=1}^{M} x_{im} \times \left[\frac{||F_i|| \times T_l}{PS}\right] \leq S_m \\ \\ \forall i \sum_{m=1}^{M} x_{im} = 1 \\ \\ \prod_{j=1}^{d} M_j \leq W \\ \\ \end{cases} \tag{4.7}$$

Dans la section qui suit, nous présentons une nouvelle méthode pour résoudre le problème décrit dans la formalisation 4.7.

4.2.2 Algorithme de conception

Notre approche contient principalement deux phases, nous les détaillons dans ce qui suit.

4.2.2.1 Phase de fragmentation de $\mathcal{F}\&\mathcal{A}$

La phase de partitionnement est considérée comme le noyau de l'approche $\mathcal{F}\&\mathcal{A}$. Elle consiste à partitionner horizontalement l'entrepôt de données sur la base des prédicats de la charge de requêtes. Les tables de dimension D_j ($1 \leq j \leq d$) sont partitionnées en exploitant les *prédicats de sélection* des requêtes de la charge $\mathcal{Q}$, le schéma de fragmentation obtenu est noté $\mathcal{FS}(D_j)$. Par contre, la table des faits $\mathcal{F}$ est partitionnée d'une manière dérivée.

La méthode de partitionnement de table de faits qui découle de cette approche est connue en littérature sous le terme de "*partitionnement référentiel*". Il a récemment été incorporé dans la couche des systèmes de base de données comme *Oracle11g* [130] et il est déployable sur les autres SGBD par l'utilisation des vues matérialisées. Il est à noter que la méthode de partitionnement est orthogonale à $\mathcal{F}\&\mathcal{A}$, ce qui signifie que toute approche de partitionnement, parmi celles disponibles dans la littérature, peut être intégrée dans $\mathcal{F}\&\mathcal{A}$ et utilisée comme méthode de référence. Cette particularité donne plus de mérite à l'approche $\mathcal{F}\&\mathcal{A}$ car elle peut

être considérée comme une approche générique facilement utilisable dans différents contextes applicatifs.

4.2.2.1.1 modèle d'un schéma de fragmentation Pour sélectionner le meilleur schéma de fragmentation de l'entrepôt de données, nous utilisons le concept de *schéma de fragmentation candidat*. Généré lors de l'exécution de l'algorithme de mise en œuvre de $\mathcal{F}\&\mathcal{A}$, il peut être choisi comme la solution finale représentée par l'ensemble des N_F fragments des faits qui seront alloués sur les nœuds de la grappe de bases de données.

La structure d'un schéma de fragmentation candidat doit être choisie avec soin car elle représente le schéma de fragmentation en mémoire. La taille de cette structure peut donc impacter négativement l'algorithme proposé.

Dans notre stratégie, la fragmentation se base sur la fragmentation des tables de dimension, donc, nous nous intéressons uniquement au schéma de fragmentation des tables de dimension pour réduire la taille de la structure représentant le schéma de fragmentation candidat.

Notre schéma de fragmentation candidat peut être présenté comme un *tableau multidimensionnel*, où chaque ligne représente le schéma de partitionnement d'un attribut d'une table de dimension qui participe au partitionnement. La valeur de chaque cellule d'un tableau donné représentant un attribut $\mathcal{A}_j$ appartient à l'intervalle $[1..n_i]$, où n_i est le nombre de sous domaines de l'attribut $\mathcal{A}_j$. Le schéma de fragmentation de chaque table est généré comme décrit dans la thèse de Kamel Boukhalfa [32].

1. **Enumération des prédicats de sélection**. Cette étape consiste à énumérer tous les prédicats de sélection simples utilisés par les L requêtes de départ. Chaque prédicat de sélection est défini par : $A\ \theta\ Valeur$ tel que A est un attribut d'une table et $\theta \in \{=, \prec, \succ, \leq, \geq\}$, et $Valeur \in Domaine(A)$
2. **Attribution des prédicats aux tables**. Les prédicats de sélection trouvés dans l'étape précédente sont définis a priori sur l'ensemble des tables. L'objectif de cette phase est d'attribuer à chaque table $D_i (1 \leq i \leq d)$ un ensemble de prédicats simples $SSPD_i$.
3. **Identification des tables à fragmenter**. Cette étape consiste à identifier les tables de dimensions à fragmenter. Pour ce faire, toute table D_i ayant un $SSPD_i$ vide, ne sera pas prise en compte dans le processus de fragmentation. Soit $D_{candidat}$ l'ensemble des tables ayant un $SSPD_i$ non vide. Soit g le cardinal de l'ensemble $D_{candidat}$.
4. **Vérification des règles de complétudes et de minimalité**. L'objectif de cette étape est de s'assurer que si une table est fragmentée en au moins deux fragments, elle sera accédée différemment par au moins deux applications [119]. Pour atteindre cet objectif, nous appliquons l'algorithme COM-MIN [116] à chaque table D_i appartenant à l'ensemble $D_{candidat}$. L'algorithme fournit en sortie une forme complète et minimale de ces ensembles.
5. **Fragmentation des tables**. Chaque table appartenant à l'ensemble $D_{candidat}$ est partitionnée. Tout attribut participant dans le processus de partitionnement est nommé *attribut de fragmentation*. Le partitionnement du domaine de chaque attribut peut être représenté par un *tableau multidimensionnel*, où chaque ligne représente le partitionnement du domaine de l'attribut de fragmentation. La valeur de chaque cellule d'un tableau

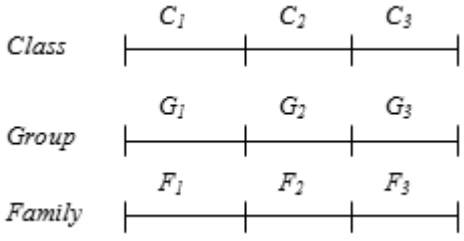
donné représentant un attribut A_i appartient à l'intervalle $[1 \dots n_i]$, où n_i représente le nombre de sous domaines de l'attribut A_i. En se basant sur cette représentation, le schéma de fragmentation de chaque table est généré comme suit. [11]

- Si toutes les cellules d'un attribut donné ont des valeurs différentes, alors tous les sous-domaines sont considérés pour fragmenter la table correspondante.
- Si toutes les cellules d'un attribut donné ont la même valeur cela signifie que cet attribut ne participe pas au processus de fragmentation.
- Si certaines cellules d'un attribut ont la même valeur, alors leurs sous-domaines correspondants sont fusionnés en un seul.

Ainsi, la table F est alors fragmentée en fonction de tous les schémas de fragmentation des tables de l'ensemble $D_{candidat}$. Chaque fragment horizontal F_i de la table F est défini de la manière suivante : $F_i = F \ltimes D_{1j} \ltimes D_{2k} \dots, D_{3k}$

Exemple 4. *Supposons que le partitionnement de l'entrepôt APB-1 version II est entraîné par les attributs de fragmentation Class, Groupe et Family.*

Et supposons que le domaine de chaque attribut est décomposé en trois sous-domaines distincts: $Dom(Class) = \{C_1, C_2, C_3\}$, $Dom(Group) = \{G_1, G_2, G_3\}$ *et* $Dom(Family) = \{F_1, F_2, F_3\}$, *comme le montre la figure 4.3 (a).*



(a)

Class	1	2	3
Group	1	1	2
Family	1	1	1

(b)

Figure 4.3 – *Schéma de fragmentation candidat* $\mathcal{A}_{Product}$

Notons que l'attribut $A_2 = Family$ n'est pas concerné par le processus de fragmentation, toutes ses cellules dans $\mathcal{A}_{Product}[2][h]$ ont la même valeur, avec $1 \leq h \leq 3$.

Une fois que le modèle formel du schéma de fragmentation est présenté, nous proposons deux algorithmes pour implémenter l'approche $\mathcal{F}\&\mathcal{A}$: algorithme hill climbing noté $\mathcal{HC}$ [55] et l'algorithme génétique noté $\mathcal{GA}$ [78], que nous adaptons à notre problème.

4.2.2.1.2 Algorithme Hill Climbing Le premier algorithme que nous proposons est un algorithme Hill Climbing, nommé $\mathcal{F}\&\mathcal{A} - \mathcal{HC}$. A partir d'une solution initiale, il se déplace itérativement dans l'espace de recherche vers des solutions voisines encore meilleures. HC comporte les étapes suivantes.

- *Trouver une solution initiale* $\mathcal{I}_0$ qui peut être obtenue par l'utilisation d'une *distribution aléatoire* pour les cellules du schéma de fragmentation candidat, pour chaque attribut de fragmentation A_K de la table de dimension D_j dans $\mathcal{D}$.

– *Améliorer itérativement la solution initiale* $\mathcal{I}_0$ en utilisant des mouvements locaux, tant que la réduction du temps d'exécution des requêtes de $\mathcal{Q}$ est possible et que les contraintes du problème sont satisfaites.

Soulignons que, comme le nombre de schémas de fragmentation candidats générés à partir de $\mathcal{DWS}$ est fini, l'algorithme HC *termine* son exécution en trouvant la solution finale $\mathcal{I}_F$. Cela garantit la convergence de la solution naïve (initiale) à un niveau théorique. Reste alors la définition formelle de ces opérateurs, que nous allons présenter.

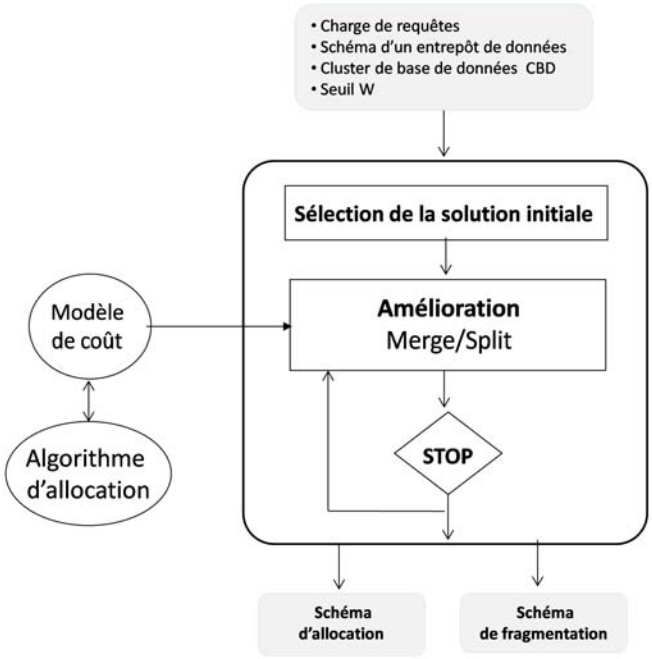


Figure 4.4 – *Approche $\mathcal{F}\&\mathcal{A}$ basée sur l'algorithme Hill Climbing*

4.2.2.1.2.1 Solution initiale .

Pour démarrer un algorithme Hill Climbing, il faut lui fournir une solution initiale à faire évoluer. La création de cette solution est entièrement aléatoire. Nous vérifions si elle vérifie la contrainte de maintenance (nombre de fragments générés inférieur ou égal au seuil de fragmentation).

Si elle vérifie la contrainte, nous gardons la solution, sinon, nous faisons appel à un algorithme glouton (greedy algorithm) qui fusionne les sous-domaines jusqu'à satisfaire la contrainte de maintenance.

4.2.2.1.2.2 Opérateurs de mouvements Notre solution naïve peut être améliorée par l'introduction de deux opérateurs spécialisés, *Merge* et *split*, ce qui nous permet de réduire encore le coût total du traitement des requêtes de la charge $\mathcal{Q}$.

Soit A_K, un attribut de fragmentation de la table de dimension D_j de $\mathcal{D}$, ayant $\mathcal{FS}(D_j)$ comme schéma de fragmentation.

- **L'opérateur Merge** prend en entrée deux partitions de A_K de $\mathcal{FS}(D_j)$ nommées $\mathcal{P}_D^p(A_k)$ et $\mathcal{P}_D^q(A_k)$, et retourne en sortie un nouveau schéma de fragmentation pour D_j, noté $\mathcal{FS}'(D_j)$. $\mathcal{P}_D^p(A_k)$ et $\mathcal{P}_D^q(A_k)$ ont été fusionnées en une partition dans le domaine de A_K, notée $\mathcal{P}_D^{p,q}(A_k)$. *Merge* réduit le nombre de fragments générés par le schéma de fragmentation candidat $\mathcal{FS}(D_j)$ de D_j. Il est utilisé lorsque le nombre de fragments générés par $\mathcal{FS}(D_j)$ ne satisfait pas la contrainte de maintenance $\mathcal{W}$. Formellement, l'opérateur *Merge* est défini ainsi:

$$Merge : \langle A_k, D_j, \mathcal{FS}(D_j), \mathcal{P}_D^p(A_k), \mathcal{P}_D^q(A_k)\rangle \rightarrow \langle A_k, D_j, \mathcal{FS}'(D_j), \mathcal{P}_D^{p,q}(A_k).\rangle \tag{4.8}$$

- **L'opérateur Split** est le dual de l'opérateur *Merge*. Il prend en entrée une partition de A_k du $\mathcal{FS}(D_j)$ et retourne en sortie un nouveau schéma de fragmentation pour D_j, noté $\mathcal{FS}'(D_j)$, où $\mathcal{P}_D(A_k)$ est divisée en deux partitions de A_K, notées $\mathcal{P}_D^p(A_k)$ et $\mathcal{P}_D^q(A_k)$. *Split* augmente le nombre de fragments générés par la fragmentation schéma $\mathcal{FS}(D_j)$ de D_j. Formellement, *Split* est défini ainsi:

$$Split : \langle A_k, D_j, \mathcal{FS}(D_j), \mathcal{P}_D(A_k)\rangle \rightarrow \langle A_k, D_j, \mathcal{FS}'(D_j), \mathcal{P}_D^p(A_k), \mathcal{P}_D^q(A_k).\rangle \tag{4.9}$$

Exemple 5. *Considérons le schema de fragmentation candidat $\mathcal{A}_{Product}$ (Voir 4). Après l'application de l'opérateur Merge sur l'attribut $A_0 = Class$. Ainsi, nous obtenons un nouveau schéma de fragmentation noté $\mathcal{A}'_{Product}$. De plus, la figure 4.5 montre aussi $\mathcal{A}'_{Product}$ après l'application de l'opérateur Split sur l'attribut $A_2 = Family$. Comme cet attribut n'est pas impliqué dans le processus de fragmentation, nous obtenons un schéma de fragmentation final faisant apparaitre le candidat final de ce schéma de fragmentation $\mathcal{A}''_{Product}$ comprenant 8 fragments au total.*

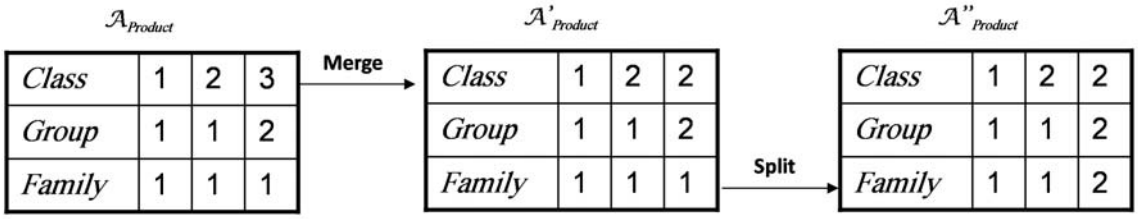


Figure 4.5 – *Application des opérateurs Merge et Split*

En fonction de ces opérateurs exécutés sur les schémas de fragmentation des tables de dimension, l'heuristique HC trouve toujours la solution finale $\mathcal{I}_F$, alors que le coût total du traitement des requêtes de la charge $\mathcal{Q}$ peut être réduite et que la contrainte de maintenance $\mathcal{W}$ peut être satisfaite.

L'algorithme 6 décrit la méthodologie $\mathcal{F}\&\mathcal{A}$ lorsque algorithme HC est utilisé comme un algorithme de fragmentation.

Algorithm 3 Algorithm F&A-HC

Input:
- database cluster $\mathcal{DBC}$;
- set of star queries $\mathcal{Q}$;
- maintenance constraint $\mathcal{W}$;

Output:
- The best fragmentation scheme $BestFragScheme$;

Begin
$FragScheme \leftarrow InitialSolution(\mathcal{DBC}, \mathcal{Q}, \mathcal{W})$
repeat
  $BestFragScheme \leftarrow FragScheme$
  if $(\neg IsFeasible(FragScheme))$ **then**
    $i \leftarrow 1$
    while $((\neg IsFeasible(FragScheme))$ AND $(i < L))$ **do**
      $Attrib \leftarrow USE[i]$
      while $(\neg IsFeasible(FragScheme))$ AND $(FragScheme.CanMerge(Attrib))$ **do**
        $FragScheme \leftarrow BestMerge(FragScheme, A)$
      end while
      if $(\neg FragScheme.CanMerge(Attrib))$ **then**
        $i \leftarrow i + 1$
      end if
    end while
  else
    $i \leftarrow N$
    while $((\neg IsFeasible(FragScheme))$ AND $(i > 0))$ **do**
      $Attrib \leftarrow USE[i]$
      while $(\neg IsFeasible(FragScheme))$ AND ($FragScheme.CanSplit(Attrib))$ **do**
        $P \leftarrow MaxSelectPartition(A)$
        $BestFragScheme \leftarrow Split(A, D, FragScheme, P)$
        if $((IsFeasible(BestFragScheme'))$ AND $(Cost(FragScheme) < Cost(BestFragScheme))$ **then**
          $FragScheme \leftarrow BestFragScheme$
        end if
      end while
      if $(\neg FragScheme.CanSplit(Attrib))$ **then**
        $i \leftarrow i - 1$
      end if
    end while
  end if
until $Cost(FragScheme) < Cost(BestFragScheme)$
return $BestFragScheme$
End

4.2.2.1.3 Algorithme génétique La principale motivation de l'utilisation des algorithmes génétiques pour résoudre le problème de conception d'un entrepôt de données parallèle sur une grappe de base de données se fonde sur le fait que les entrepôts de données peuvent facilement déterminer un grand nombre de schémas de fragmentation dans de tels environnements [133]. Traditionnellement, les algorithmes génétiques ont été utilisés pour sélectionner le schéma des outils d'optimisation, comme la sélection des vues matérialisées (*Zhang et al.* [150]), l'optimisation des requêtes de jointure (*Ioannidis et al .* [82]) et la sélection du schéma de fragmentation horizontale (*Boukhalfa et al.* [32]).

Ce sont des algorithmes d'optimisation s'appuyant sur des techniques dérivées de la génétique et des mécanismes d'évolution de la nature. Ils appartiennent à la famille des algorithmes évolutionnistes. Leur but est d'obtenir une solution approchée. Ils font évoluer une population d'individus grâce aux mécanismes de reproduction sexuée (croisement et mutation). A chaque génération, c'est un ensemble d'individus obtenus par l'application de l'opérateur de sélection qui sera mis en avant et non un individu particulier. En conséquence, un ensemble de solutions différentes est généralement généré.

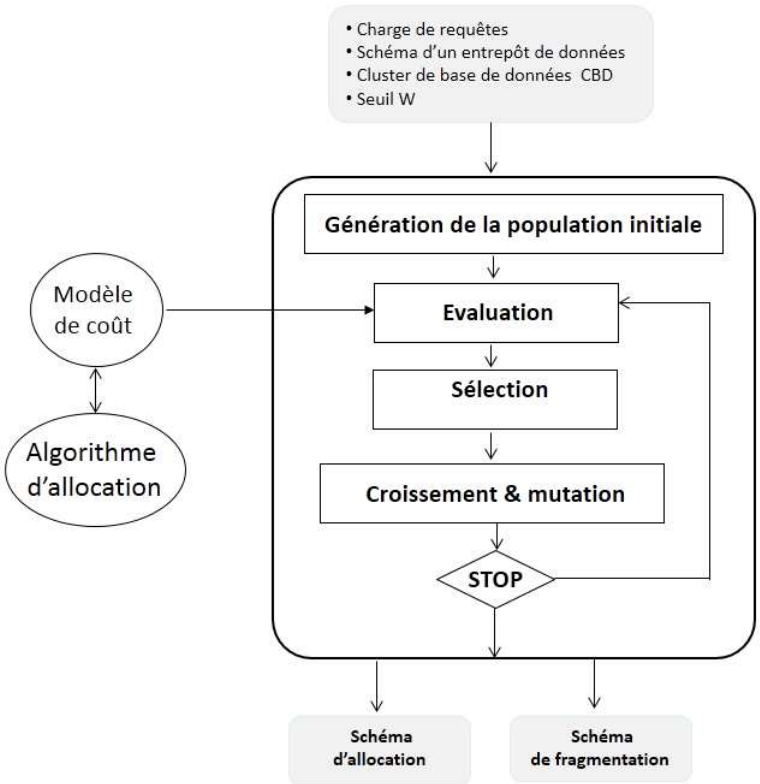


Figure 4.6 – *Approche $\mathcal{F}\&\mathcal{A}$ basée sur l'algorithme génétique*

Le concept fondamental lié à la structure des algorithmes génétiques est la notion de chromosome, qui est une représentation codée de solution possible. Pour démarrer un algorithme génétique, il faut lui fournir une population à faire évoluer en utilisant les trois opérateurs de recherche génétique (sélection, croisement et mutation) jusqu'à obtention d'une population présentant de meilleures caractéristiques. Comme illustré dans la figure 4.6, l'algorithme génétique

que nous avons adopté comme algorithme de base de l'approche $\mathcal{F}\&\mathcal{A}$ utilise le même modèle de coût que l'algorithme Hill Climbing pour évaluer la qualité su schéma de fragmentation. Les grandes étapes de cet algorithme sont décrites par l'algorithme 4.

Algorithm 4 Algorithm GA

Input:
- search space $\mathcal{S}$;;
- fitness function $\mathcal{J}$;
- maximum number of iterations $MaxIteration$;

Output:
- The best population $BestPopulation$;

Begin
$Population \leftarrow InitialPopulation(\mathcal{S}, \mathcal{J})$
$BestPopulation \leftarrow Population$
$i \leftarrow 1$
while ($\mathcal{J}(Population) < \mathcal{J}(BestPopulation)$ AND $i < MaxIteration$) **do**
  $Crossover(Population)$
  $Mutation(Population)$
  $Selection(Population)$
  $BestPopulation \leftarrow Population$
  $i \leftarrow i + 1$
end while
return $BestIndividu$
End

Maintenant la question cruciale pour notre recherche est la suivante :"*Comment adapter un algorithme génétique classique à notre problème*".

Nous proposons de faire usage du *mécanisme de croisement à multipoints* [140]. Les chromosomes sont croisés-dessus une fois pour chaque attribut de fragmentation impliqué par le processus de partitionnement cible, puis l'opérateur de mutation leur est appliqué. L'objectif est de produire un peu de progéniture, sur la base d'*outils probabilistes* appropriés. Si la liaison est obtenue sur un chromosome, les attributs ayant un nombre élevé de sous-domaines ont une probabilité plus grande d'être choisi pour le croisement que les attributs qui comportent un nombre faible de sous-domaines. Il convient de noter que cette approche nous donne une sémantique raisonnable de l'opération classique de croisement dans le cadre de paramètres de fragmentation d'un entrepôt de données. L'opération de recouvrement est appliquée de manière répétée jusqu'à ce qu'aucune nouvelle réduction du nombre de fragments appropriés ne soit obtenue. L'opérateur de mutation permet de créer de nouveaux chromosomes qui peuvent ne pas être présents dans n'importe quel membre des populations engendrées. Cela permet à l'algorithme génétique d'explorer toutes les solutions possibles de l'espace de recherche de cible.

Enfin, dans ce qui suit, nous allons détailler les opérateurs de croisement et de mutation, car ils jouent un rôle majeur dans l'algorithme GA en ce qui concerne l'influence sur sa performance.

4.2.2.1.3.1 Opérateur de sélection .

Dans les algorithmes génétiques classiques, l'opérateur de sélection permet de choisir les individus les plus prometteurs, ceux qui vont participer à l'amélioration de la population ini-

tiale. En effet, une "note" ou un indice de qualité est attribué à chaque individu en utilisant une méthode d'évaluation et l'ensemble sélectionné est amélioré en utilisant les opérateurs de croisement et de mutation. Dans notre approche $\mathcal{F}\&\mathcal{A}$ axée sur un algorithme générique, la *méthode de sélection par rang de roue (roulette)* [140] est exploitée pour identifier statistiquement les meilleurs individus d'une population et éliminer les mauvais. Cette technique de sélection choisit toujours les individus possédant les meilleurs scores calculés par modèle de coût de $\mathcal{F}\&\mathcal{A}$. Les chromosomes avec des valeurs de fitness élevés ont plus de chances d'être finalement retenus.

4.2.2.1.3.2 Population initiale .

La manière de créer chacun des individus de cette population est entièrement libre. Il suffit que tous les chromosomes créés soient de la forme d'une solution potentielle, et il n'est nullement besoin de songer à créer de bons individus. Ils doivent seulement fournir une réponse, même mauvaise, au problème posé.

Pour démarrer notre algorithme génétique, nommé $\mathcal{F}\&\mathcal{A}$- $\mathcal{GA}$, nous avons généré n individus de la même manière que l'algorithme Hill Climbing. Plus précisément, nous créons n individus dont la création est entièrement aléatoire. Puis, pour chaque solution générée, nous vérifions si elle vérifie la contrainte de maintenance (nombre de fragments générés inférieur ou égal au seuil de fragmentation).

- Si elle vérifie la contrainte, nous gardons le chromosome.
- Sinon, nous faisons appel à un algorithme glouton (greedy algorithm) qui fusionne les sous-domaines jusqu'à satisfaire la contrainte de maintenance.

4.2.2.1.3.3 Fonction d'évaluation .

L'évaluation des individus permet de définir quel individu est meilleur par rapport à un autre. Pour chaque solution (schéma de fragmentation) générée, notre fonction d'évaluation calcule le coût d'exécution de la charge de requêtes $\mathcal{Q}$ sur les M nœuds associés au grappe de base de données. Le coût de chaque solution est estimé selon le modèle de coût décrit dans la section 4.2.2.2.5.

4.2.2.1.3.4 Opérateur de croisement .

Il vise à échanger des gènes entre deux chromosomes donnés. L'opérateur de croisement combine deux parents pour reproduire de nouveaux enfants, avec l'idée que l'un de ces enfants peut rassembler toutes les bonnes fonctionnalités qui caractérisent ses parents. L'opérateur de croisement n'est généralement pas appliqué à tous les parents; il est plutôt appliqué avec une *probabilité de croisement* donné, notée P_c. Dans notre implémentation, nous supposons que tous les attributs de fragmentation du chromosome ont la même chance d'être sélectionnés par l'opérateur de croisement. Ce paramètre nous permet d'obtenir finalement une répartition équitable de bonnes caractéristiques de parents à travers les enfants. L'opération de croisement consiste à choisir les *points de passage* appropriés, puis générer les enfants à partir de ces points.

Soit D_j un tableau unidimensionnel dans $\mathcal{D}$ et soit $\mathcal{A}_{D_j}$ et $\mathcal{B}_{D_j}$ deux candidats du schéma de la fragmentation de D_j, respectivement. L'enfant $\mathcal{A'}_{D_j}$ hérite du premier gène du parent $\mathcal{A}_{D_j}$ jusqu'au premier point de passage atteint. Ensuite, tous les gènes du parent $\mathcal{B}_{D_j}$ sont copiés dans le chromosome de $\mathcal{A'}_{D_j}$ jusqu'au prochain point de passage atteint. Cette procédure génère finalement tout le chromosome de $\mathcal{A'}_{D_j}$. Le chromosome de $\mathcal{B'}_{D_j}$ est produit selon un procédé similaire.

Exemple 6. *Considérons deux schémas de fragmentation candidats. La figure 4.7 montre un exemple de croisement entre $\mathcal{A}_{Produit}$ et $\mathcal{B}_{Produit}$ qui génère deux schémas de fragmentation candidats enfants $\mathcal{A'}_{Produit}$ et $\mathcal{B'}_{Produit}$, respectivement.*

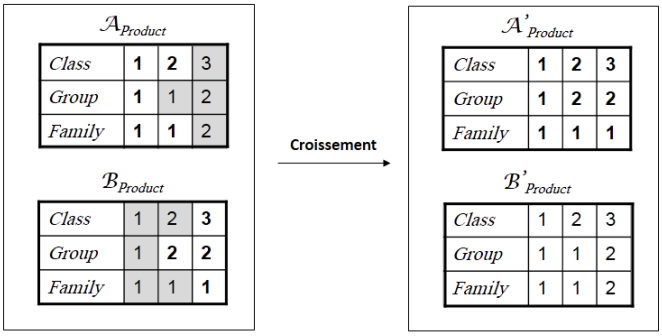


Figure 4.7 – Exemple de l'opérateur de croissement

4.2.2.1.4 Opérateur de mutation .

L'opérateur de mutation modifie un ou plusieurs gènes dans un chromosome, dans le but de parvenir à *variabilité stochastique* de l'algorithme GA et d'obtenir ainsi une convergence rapide vers une solution stable. Encore une fois, une approche probabiliste est exploitée, par la fixation de la *probabilité de mutation*, notée P_M, qui régit l'apparition de l'opération de mutation. Habituellement P_M est configuré pour être petit, ce qui garantit la réalisation d'une solution stable rapidement [117].

Exemple 7. *La figure 4.8 montre le résultat d'application de la mutation sur le schéma de fragmentation $\mathcal{A}_{Produit}$.*

Figure 4.8 – Exemple de l'opérateur de mutation

4.2.2.2 Phase d'allocation de $\mathcal{F}\&\mathcal{A}$

.

Comme souligné dans les sections précédentes, la phase d'allocation de données de $\mathcal{F}\&\mathcal{A}$ est effectuée simultanément à la phase de partitionnement des données. Fondamentalement, chaque schéma de fragmentation horizontale candidat généré soit par des algorithmes $\mathcal{HC}$ ou $\mathcal{GA}$ au cours de la phase de partitionnement des données est réparti entre les nœuds de la grappe de bases de données. L'objectif est de minimiser le coût de traitement des requêtes dans $\mathcal{Q}$ sur tous nœuds, tout en satisfaisant les contraintes de stockage et de traitement sur chaque nœud.

L'approche $\mathcal{F}\&\mathcal{A}$ introduit un formalisme innovant à base de matrices qui est capable de capturer l'affinité entre les fragments, l'interaction entre les requêtes d'entrée et les caractéristiques de la grappe de nœuds (c'est-à-dire, la puissance et la capacité de stockage de traitement). L'allocation des données sur les nœuds est faite avec un algorithme basé sur l'affinité, nommé $\mathcal{F}\&\mathcal{A}$- `ALLOC`.

Cet algorithme est une variante de l'algorithme Navathe et al [109] utilisé pour définir les fragments verticaux des tables relationnelles en utilisant une approche graphique. Il possède cinq étapes: l'énumération des attributs utilisés par la requête, la construction de la matrice d'usage des attributs, la construction de la matrice d'affinités des attributs, le regroupement des attributs et la construction des fragments verticaux.

Dans ce qui suit, nous décrivons d'abord les principales étapes de l'algorithme des affinités pour l'allocation des fragments (`Algo_Aff_Alloc_Frag`). Et, pour faciliter la compréhension de chaque étape, nous illustrons leur fonctionnement par un exemple.

4.2.2.2.1 Construction de la Matrice d'Usage des Fragments .

Cette matrice $\mathcal{M}_U$ modélise l'utilisation de fragments par des requêtes de $\mathcal{Q}$. À cette fin, $\mathcal{M}_U$ possède L lignes (nombre de requêtes) et N_F (nombre de fragments) colonnes. $\mathcal{M}_U[l][i] = 1$, avec $1 \leq l \leq L$ et $1 \leq i \leq N_F$, si le fragment F_i est impliqué par la requête Q_l; sinon $\mathcal{M}_U[l][i] = 0$ et une colonne *supplémentaire* est ajoutée à $\mathcal{M}_U$ pour représenter la fréquence d'accès f_l de chaque requête Q_L de $\mathcal{Q}$. [109]

Exemple 8. *Soit $Q = \{Q_0, Q_1, Q_2, Q_3\}$ et $F = \{F_1, F_2, F_3, F_4, F_5, F_6, F_7, F_8\}$ est l'ensemble des requêtes et des fragments générés, respectivement. La $\mathcal{M}_U$ possible est indiquée dans le tableau ci-dessous.*

	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	f_r
Q_0	1	0	1	0	1	0	1	0	20
Q_1	1	1	1	1	0	0	0	0	35
Q_2	0	0	1	0	1	1	1	1	30
Q_3	1	1	1	1	1	1	1	1	15

Tableau 4.1 – *Matrice d'Usage des Fragments*

4.2.2.2.2 Construction de la Matrice d'Affinités des Fragments .

La matrice d'affinité $\mathcal{M}_A$ modélise l'*affinité* entre deux fragments F_{I_p} et F_{i_q}. Cette matrice est générée de la même manière que la matrice d'affinité des attributs de la fragmentation verticale [109]. C'est une matrice carrée $(N_F \times N_F)$ symétrique dont les lignes et les colonnes représentent les fragments. Il est à noter que nous ne nous intéressons pas aux valeurs de la diagonale.

La valeur de chaque élément $MA(i, i')(1 \leq i, i' \leq N)$ est égale à la somme des fréquences d'accès (que nous notons par λ) de toutes les requêtes utilisant simultanément les deux fragments. Cela signifie que les fragments $F_i, F_{i'}$ sont utilisés en même temps avec une fréquence de λ .

Exemple 9. *A partir de la matrice d'usage des fragments de l'exemple précédant, la matrice d'affinité des prédicats est construite. Cette matrice contient* 8 *lignes et* 8 *colonnes autant que de fragments. La valeur* $aff(F_2, F_3)$ *est égale à* 50*, ce qui correspond à la somme des fréquences de* Q_1 *et* Q_3

	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8
F_1	-	50	70	50	65	15	35	15
F_2	50	-	50	50	15	15	15	15
F_3	70	50	-	50	65	45	65	45
F_4	50	50	50	-	15	15	1 5	15
F_5	65	15	65	15	-	45	65	45
F_6	15	15	45	15	45	-	45	45
F_7	35	15	65	15	65	45	-	45
F_8	15	15	45	15	45	45	45	-

Tableau 4.2 – *Matrice d'Affinité des Fragments*

4.2.2.2.3 Le regroupement des fragments: $\mathcal{F}\&\mathcal{A}$-ALLOC .

L'algorithme d'allocation $\mathcal{F}\&\mathcal{A}$, nommé $\mathcal{F}\&\mathcal{A}$- `ALLOC`, s'exécute sur le schéma de partitionnement horizontal candidat généré par l'algorithme de fragmentation ($\mathcal{HC}$ ou $\mathcal{GA}$) pendant la phase de partitionnement des données. $\mathcal{F}\&\mathcal{A}$- `ALLOC` utilise des structures matricielles pour capturer l'affinité entre les fragments, les requêtes d'entrée et les caractéristiques des nœuds de la grappe $\mathcal{DBC}$.

A cette étape, nous adaptons l'algorithme de regroupement graphique développé pour la fragmentation verticale par Navathe et al [109]. La structure de données utilisée est un *graphe complet et étiqueté*, appelé "graphe d'affinité des fragments". Les nœuds de ce graphe représentent les fragments, et une arête entre deux fragments représente la valeur d'affinité. L'objectif principal est la formation des groupes qui représentent l'unité d'allocation par la suite.

L'adaptation de cet algorithme concerne la formation du groupe où les fragments sont regroupés selon leur faible affinité, contrairement à Navathe où les attributs sont regroupés selon leur grande affinité. Il est à noter que ce type de regroupement augmente le parallélisme

entre les nœuds associés à la machine parallèle. Notre algorithme définit des groupes d'une taille limité.

L'algorithme part d'une matrice d'affinités entre l'ensemble des fragments $\mathcal{V}$ et construit le graphe complet et étiqueté appelé $\mathcal{GAF}(\mathcal{V}, \mathcal{E})$. Chaque arrête e $(F_i, F_{i'})$ est étiquetée par un poids représentant la valeur de l'affinité entre le fragment F_i et $F_{i'}$. Une fois le graphe d'affinité construit, l'algorithme commence par sélectionner un nœud F_i du graphe $\mathcal{GAF}(\mathcal{V}, \mathcal{E})$ d'une manière aléatoire. Ce nœud représentera le nœud de départ du groupe G_1. Ensuite, il essaye d'étendre le groupe G_1. Une fois le groupe formé, ses nœuds sont écartés, et la même procédure est réitérée sur les autres nœuds pour former d'autres groupes.

A la fin de la phase de regroupement, un ensemble de groupes $\mathcal{G} = \{G_1, G_2, \ldots, G_Z\}$ est obtenu et chaque groupe devient l'unité d'allocation.

Exemple 10. *Le graphe d'affinités est construit à l'aide de la matrice d'affinités des fragments. Il possède* 8 *nœuds, et chaque arrête est étiquetée par sa valeur d'affinité. Par exemple l'arrête* F_2*,* F_3 *est libellée par 50. Le processus de regroupement commence par le nœud* 1 *qui représente le nœud de départ du groupe* G_1*. En examinant la matrice d'affinité, nous ne trouvons qu'un seul prédicat* F_6 *ayant une petite affinité avec* F_1*. Le groupe* G_1 *est donc augmenté par l'ajout de* F_6*. L'algorithme cherche toujours à étendre* G_1*, mais nous ne pouvons pas inclure de nouveaux nœuds ayant une petite affinité. Le groupe* G_1 *est donc formé et nous l'éclatons en coupant ces éléments du reste des nœuds du graphe. L'algorithme réitère le processus de formation de groupes, en utilisant les nœuds restants. Finalement, quatre groupe sont obtenus et illustrés dans la figure.*

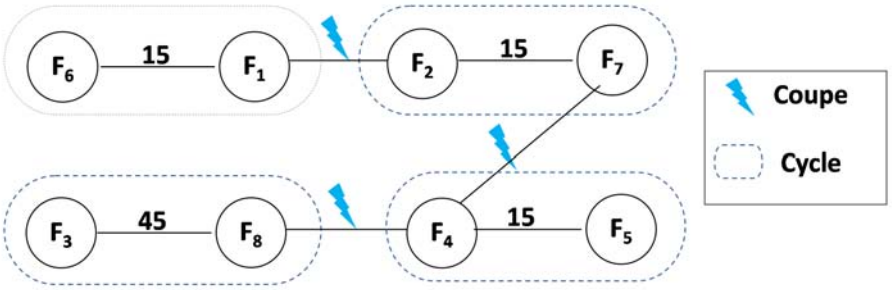


Figure 4.9 – *Les groupes de fragments générés*

4.2.2.2.4 Construction de la Matrice de Placement des Fragments. La matrice $\mathcal{M}_P$ représente la présence des fragments sur les nœuds. Les lignes et les colonnes de cette matrice sont associées aux N_F fragments obtenus par le schéma de fragmentation en cours d'évaluation et les M nœuds associés à la grappe de base de données $\mathcal{DBC}$ respectivement. Ses éléments sont binaires (0 ou 1) et définis comme suit.

$$M_P[i][m] = \begin{cases} 1 & si\ le\ fragment\ F_i\ est\ alloué\ sur\ le\ nœud\ N_m \\ 0 & sinon. \end{cases}$$

Un fragment F_i est pertinent pour l'exécution d'une requête Q_L de $\mathcal{Q}$ sur un noeud de traitement N_m de $\mathcal{N}$ si, *et seulement si*, la propriété suivanteest valide.

$$M_P[i][m] = 1 \wedge M_U[l][i], avec\ 1 \leq i \leq N_F,\ 1 \leq m \leq M\ et\ 1 \leq l \leq L. \tag{4.10}$$

Pour avoir les valeurs de la matrice $\mathcal{M}_P[i][m]$, il suffit d'exécuter l'algorithme $\mathcal{F}\&\mathcal{A}$- `ALLOC` qui passe par trois étapes.

1. Calcule la taille de chaque groupe G_Z.

$$Taille(G_Z) = \sum_i Taille(F_i), tel\ que\ Taille(F_i)\ désigne\ la\ taille\ du\ fragment\ F_i. \tag{4.11}$$

2. Trie les nœuds de la grappe de bases de données $\mathcal{DBC}$ par ordre descendant en fonction de leur capacité de stockage et de leur puissance de calcul.
3. Alloue "les plus grands" groupes de fragments sur les nœuds les plus puissants (le nœud ayant une capacité de stockage et une puissance de traitement élevées) dans $\mathcal{DBC}$ d'une manière circulaire.

L'algorithme $\mathcal{F}\&\mathcal{A}$- `ALLOC` *est l'une des contributions majeure de notre recherche*(variante du placement circualire). Dans ce qui suit, nous fournissons une description détaillée des procédures exploitées.

- $GetNumberOfNodes$ prend en entrée la grappe de base de données $\mathcal{DB}$ et retourne en sortie le nombre de nœuds, qui peuplent $\mathcal{DBC}$, noté M ;
- $InitializeMatrix$ prend en entrée deux entiers non négatifs u et v et retourne en sortie une matrice $M = (m_{ij})_{1 \leq i \leq u\ et\ 1 \leq j \leq v}$;
- $CreateGroups$ prend en entrée une matrice d'affinité $\mathcal{M}_A$ et retourne un ensemble de groupes;
- $InitializeSet$, appelée sans paramètres, initialise un ensemble vide de valeurs fondamentales;
- $GetSize$ prend en entrée un ensemble $\mathcal{S}$ et retourne en sortie la taille de $\mathcal{S}$;
- $GetObject$ prend en entrée un ensemble $\mathcal{S}$ et un entier non négatif z. Elle retourne le z^ième ensemble de l'ensemble $\mathcal{S}$;
- $ComputeFragGroupSize$ calcule la taille des z fragments du groupe G_z;
- Add ajoute un objet g à un groupe $\mathcal{G}$;
- $SortNode$ prend en entrée un ensemble d'objets $\mathcal{O}$ dont chaque objet O_i est pondéré par un poids (P_i, S_i) qui est utilisé comme critère pour trier les éléments de $\mathcal{O}$ d'une manière ascendante ou descendante $manner \in \{\texttt{ASC}, \texttt{DES}\}$;
- $RetrieveCandidateNodes$ prend en entrée un ensemble de nœuds $\mathcal{N}$ et un élément f de taille t. Elle retourne en sortie un sous-ensemble de $\mathcal{O}$ qui contient les noeuds capables de stocker f;
- $FindBestNode$ parmi une liste des noeuds $\mathcal{N}$. Elle retourne celui qui détient le poids le plus élevé pour contenir un élement un élément f de taille t;

Algorithm 5 Algorithm $\mathcal{F}\&\mathcal{A}$-ALLOC

1: **Input:**
   – database cluster $\mathcal{DBC}$;
   – set of star queries $\mathcal{Q}$;
   – number of fragments N_F ;
   – FUM $\mathcal{M}_U$;
   – FAM $\mathcal{M}_A$;
2: **Output:**
   – FPM $\mathcal{M}_P$;
3: **Begin**
4: $M \leftarrow GetNumberOfNodes(\mathcal{DBC})$
5: $\mathcal{M}_P \leftarrow InitializeMatrix(N_F, M)$
6: $FragGroups \leftarrow CreateGroups(\mathcal{M}_U, \mathcal{M}_A, \mathcal{W})$
7: $FragGroupSizes \leftarrow InitializeSet()$
8: $FragGroupNumber \leftarrow GetSize(FragGroups)$
9: $z \leftarrow 0$
10: **while** $(z < FragGroupNumber)$ **do**
11: $G_z \leftarrow GetObject(FragGroups, z)$
12: $Size(G_z) \leftarrow ComputeSize(G_z)$
13: $FragGroupSizes.Add(Size(G_z), z)$
14: $z \leftarrow z + 1$
15: **end while**
16: $SortNodes(\mathcal{DBC}, S_m, P_m, \texttt{DESC})$
17: $z \leftarrow 0$
18: **while** $(z < FragGroupNumber)$ **do**
19: $G_z \leftarrow GetObject(FragGroups, z)$
20: $Size(G_z) \leftarrow FragGroupSizes.Get(z)$
21: $CandidateNodes \leftarrow RetrieveCandidateNodes(\mathcal{DBC}, G_z, Size(G_z))$
22: $N_m^* \leftarrow FindBestNode(\mathcal{DBC}, \mathcal{Q}, CandidateNodes)$
23: $i \leftarrow 0$
24: **while** $(i < N_F)$ **do**
25: $\mathcal{M}_P[i][N_m^*] \leftarrow 1$
26: $i \leftarrow i + 1$
27: **end while**
28: $z \leftarrow z + 1$
29: **end while**
30: $ReplicateFragGroups(\mathcal{DBC}, FragGroups, \mathcal{M}_P)$
31: **return** $\mathcal{M}_P$
32: **End**

Il s'agit de trouver le schéma de placement des fragments qui *minimise* le coût total de traitement des requêtes de $\mathcal{Q}$ et *maximise l*a productivité de chaque nœud:

$$\sum_{l=1}^{L} f_l \times \max_{1 \leq m \leq M} \left\{ \sum_{i=1}^{N_F} \frac{\mathcal{M}_U[l][i] \times \mathcal{M}_P[i][m] \times Size(F_i)}{P_m} \right\}, \tag{4.12}$$

L désigne le nombre de requêtes exécutées sur $\mathcal{DBC}$, $max_{K_{min} \leq k \leq K_{max}} \left\{ Y(k) \right\}$ désigne l'opérateur *maximal*, qui récupère la valeur maximale d'un ensemble d'éléments donnés, et qui est appliqué à tous les $K_{max} - K_{min}$ des éléments de l'ensemble $\mathcal{Y}$. M désigne le nombre de noeuds de $\mathcal{DBC}$. N_F désigne le nombre de fragments appartenant à la solution, $\mathcal{M}_U$ la FUM, $\mathcal{M}_P$ la FPM, $Taille(F_i)$ la taille du fragment F_i, P_M la puissance de traitement du noeud N_m de $\mathcal{N}$.

4.2.2.2.5 Modèle de coût .

Le modèle de coût (voir *chapitre 3*) permet d'estimer le temps d'exécution nécessaire (exprimé en nombre de pages chargées) pour exécuter la charge de requêtes définie sur un schéma en étoile. Ici, nous ne prenons pas en compte le temps CPU et le coût de communication entre les nœuds de la machine parallèle, que nous supposons négligeable par rapport au temps nécessaire pour effectuer les entrées/sorties.

Étant un schéma en étoile défini par une table des faits F et d tables de dimension, chaque requête Q définie sur cet entrepôt comporte plusieurs prédicats de sélection (définis sur des attributs de dimension) et des jointures entre la table des faits et les tables de dimensions. Notre modèle de coût suppose que toutes les tables de dimension sont répliquées sur les nœuds de la grappe de base de données et résident dans leurs mémoires centrales. Une fois le schéma de fragmentation identifié, les fragments résultants doivent être alloués sur les nœuds de la machine parallèle d'une manière judicieuse qui assure un coût d'exécution de requêtes minimal.

Après l'allocation des fragments, l'exécution d'une requête de jointure en étoile Q sous une grappe de base de données suit le plan d'exécution suivant:

1. déterminer l'ensemble des fragments valides pour chaque requête de $\mathcal{Q}$,
2. déterminer l'ensemble des sous-requêtes pour chaque requête de $\mathcal{Q}$,
3. déterminer l'ensemble des nœuds valides pour l'ensemble des fragments choisis à l'étape 1,
4. déterminer le meilleur placement des sous-requêtes générées dans l'étape 2,
5. exécuter l'ensemble des sous-requêtes allouées au niveau de chaque nœud,
6. faire l'union des résultats obtenus sur chaque nœud de traitement,
7. calculer le coût d'exécution moyen noté *AvgCost* nécessaire pour l'exécution des requêtes de la charge $\mathcal{Q}$.

L'objectif est d'optimiser non seulement la valeur des solutions admissibles, mais également de pénaliser les solutions non réalisables. Les contraintes imposées par notre formalisation doivent être prises en compte. Pour des raisons de robustesse et de facilité de mise en œuvre, nous transformons le problème contraint en un problème sans contrainte. Cette transformation

s'effectue en ajoutant une pénalité à la fonction objective. En effet, le problème contraint peut être transformé en problème non-contraint à objectifs multiples.

L'ordre de grandeur de la pénalité à accorder à une violation de contrainte n'est pas simple à déterminer. Une trop petite pénalité ne guide pas suffisamment la recherche vers des solutions réalisables, alors que de trop grandes pénalités induisent des montagnes infranchissables dans l'espace de recherche, ce qui revient à interdire les solutions non réalisables. La technique la plus courante consiste à choisir une pénalité en mode division [149, 32]. Ainsi, pour un schéma de fragmentation $\mathcal{SF}$ la grandeur de pénalité *Pen(SF)* est calculée comme suit:

$$Pen(SF) = (W - NF) \times \prod_{j=1}^{M} (\frac{Taille(N_j)}{PS}). \tag{4.13}$$

En conséquence, notre fonction de coût s'écrit:

$$Cost(SH) = Pen \times \sum_{k=1}^{L} f_k \times Cost(Q_k). \tag{4.14}$$

Notons que le coût d'exécution $Cost(Q_k)$ est calculé selon les formules citées dans le précédant (Chapitre 3).

4.3 $\mathcal{F}\&\mathcal{A}\&\mathcal{R}$: une approche globale pour la conception d'un EDP

Le traitement parallèle des requêtes englobe la réécriture globale des requêtes selon le schéma de fragmentation, et l'allocation des sous-requêtes générées sur les noeuds de la grappe selon le schéma d'allocation pour que les nœuds soient uniformément chargés. Un autre problème sous-jacent aux étapes principales ci-dessus décrites doit être considéré durant le déploiement d'un EDP, celui de l'équilibrage de charge entre les nœuds de la grappe qui est primordial pour atteindre la haute performance de l'EDP. En effet, un déséquilibre de charge peut se produire pour deux raisons, ou pour la combinaison de ces deux raisons. Une mauvaise répartition des données (data skew) dans le cas où les données sont distribuées d'une manière non uniforme sur les différents nœuds de traitement, se produit généralement quand la fonction de partitionnement de données utilise un attribut dont la distribution des valeurs de données est biaisée (Attribut Value Skew); et une mauvaise répartition de traitement (Processing Skew), peut aussi se produire dans le cas qui fait référence à la situation où une grande partie de la charge de requêtes est exécutée sur peu de nœuds de traitement quand les autres nœuds sont relativement inactifs; cela est souvent dû à la répartition biaisée des données. Dans la littérature, l'équilibrage de charge est effectué via une redistribution des données des nœuds surchargés sur les nœuds sous chargés [99]. Cette migration des données peut engendrer un coût de communication élevé et le nœud coordinateur peut devenir un goulot d'étranglement. En conséquence, la *réplication des données* est devenue une exigence pour éviter le goulot d'étranglement et réduire le coût de communication. En effet, la réplication de données assure : la disponibilité des données et la tolérance de panne en cas de défaillance, la localité de traitement et l'équilibrage de charge.

Par conséquent, nous proposons une extension de l'approche $\mathcal{F}\&\mathcal{A}$ dont l'idée principale est d'exploiter l'interaction entre les différentes phases de conception et d'utiliser un modèle unifiant l'ensemble des phases. Cette unification permet de *cimenter les phases* et augmente l'omniscience des phases du déploiement.

L'idée de base derrière notre proposition est que lors de la fragmentation, chaque solution potentielle de fragmentation est testée pour l'allocation, la réplication et l'équilibrage de charge. La solution ayant un coût minimal est retenue pour la conception de l'EDP. Nous sommes conscients que cette approche peut être coûteuse. Pour réduire cette difficulté, le développement des algorithmes moins coûteux est envisagé.

Cette approche offre deux contributions principales.

- Les trois principales phases de déploiement d'un EDP(fragmentation, allocation et réplication) sont combinées en un processus unifié nommé *placement des données*. La qualité du schéma de placement des données sélectionné dépend fortement du schéma de fragmentation.
- Un nouvel algorithme d'allocation redondante (avec réplication) de fragments basé sur la classification floue en utilisant l'algorithme *Fuzzy k-means*.

4.3.1 Formulation du problème

Notre approche de conception est orientée par la fragmentation. Dans cette section, nous présentons brièvement les objets utilisés dans nos propositions ainsi qu'une formalisation du projet conjoint.

Dans ce qui suit, nous présentons d'abord quelques définitions nécessaires pour la compréhension de notre formalisation puis nous présentons notre formalisation.

4.3.1.1 Backgrounds

Quelques définitions s'imposent pour bien comprendre notre formalisation.

- *Le degré de Réplication* $\mathcal{R} = \{1, \ldots, M\}$ représente les $\mathcal{R}$ copies physiques d'un fragment F alloués sur les M noeuds de la machine parallèle. En particulier, $\mathcal{R} = 1$ représente qu'il n'y a pas de réplication (no replication), $\mathcal{R} = \mathcal{M}$ représente la réplication totale (full replication) des fragments, et $M \succ \mathcal{R} \succ 1$ représente la réplication de chaque fragment $\mathcal{R}$ fois (partial replication). Pour la réplication partielle, le degré de réplication peut être exprimé en poucentage comme suit :

$$\forall\ 1 \prec \mathcal{R} \prec M : \mathcal{CR}(\%) = \frac{(R-1) \times 100}{M}. \tag{4.15}$$

- *Le degré de Skew des Valeurs d'un Attribut.* Pour un attribut $\mathcal{A}$ dont le domaine de valeurs est partitionné en $\mathcal{S}$ sous-domaines ($\mathcal{S} \succ 1$), le degré de Skew des Valeurs d'un Attribut $\mathcal{A}$, noté par $\mathcal{SVA}(\mathcal{A})$, représente l'écart-type de la distribution des valeurs entre les sous-domaines. Soit $\mathcal{SEL}(SD_i)$ le facteur de sélectivité du sous-domaine SD_i, $\mathcal{SVA}(\mathcal{A})$ est défini comme suit:

$$\mathcal{SVA}(\mathcal{A}) = \sqrt{\frac{1}{S} \times \sum_{i=1}^{S} \left(SEL(SD_i) - \frac{1}{S} \right)^2}. \tag{4.16}$$

- *La taille d'un nœud* N_j, notée par $Size(N_j)$, représente la somme des tailles des fragments stockés sur le noeud N_j. Soient $Taille(F_i)$ le nombre des tuples d'un fragment F_i et $isStored(F_i, N_j)$ une fonction booléenne qui retourne 1 si F_i est stocké sur N_j, 0 sinon. Ainsi:

$$Size(N_j) = \sum_{i=1}^{N_F} Taille(F_i) \times isStored(F_i, N_j). \tag{4.17}$$

- *La charge de travail d'un nœud* N_j, notée par $Load(N_j, Q_k)$, représente le nombre de tuples traités pour évaluer Q_k. Formellement:

$$Load(N_j, Q_k) = \sum_{i=1}^{N_F} Taille(F_i) \times isValid(F_i, N_j, Q_k), \tag{4.18}$$

où $isValid(F_i, N_j, Q_k)$ est une fonction booléenne qui retourne 1 si F_i alloué sur N_j est utilisé pour l'exécution de Q_k; 0 sinon.

– *La taille moyenne d'un nœud.* Etant donné un entrepôt de données $\mathcal{DW}$ fragmenté en N_F fragments $\mathcal{F} = \{F_1, F_2, \ldots, F_{N_F}\}$ stockés sur $\mathcal{M}$ noeuds et θ [5] le facteur de skew de placement de données tel que $0 \leq \theta \leq 1$. Comme dans les travaux de Taniar [135], la taille moyenne d'un nœud, notée par MPS, est définie par

$$MPS = \frac{1}{\sum_{j=1}^{M} \frac{1}{j^{\theta}}} \times \sum_{i=1}^{N_F} Taille(F_i). \tag{4.19}$$

– *Le degré d'équilibrage de charge.* Etant donné un entrepôt de données $\mathcal{DW}$ fragmenté en N_F fragments $\mathcal{F} = \{F_1, F_2, \ldots, F_{N_F}\}$ stockés sur $\mathcal{M}$ nœuds. Un système est dit *équilibré*, si la distance entre toutes les charges et le centre de gravité [6] est nulle. Ainsi, le degré d'équilibrage de charge, noté par DLB, d'une requête Q_j est défini par

$$LBDS(Q_j) = \sqrt{\sum_{i=1}^{M} \frac{\left(Load(N_i, Q_j) - MeanLoad\right)^2}{\sigma^2}}, \tag{4.20}$$

où $MeanLoad = (\frac{1}{M} \sum_{i=1}^{M} Load(N_i, Q_j))$ et σ l'écart-type

4.3.1.2 Formulation

Considérons:

– un $\mathcal{EDR}$ modélisé par un schéma en étoile composé de d tables $\mathcal{D} = \{D_1, D_2, \ldots, D_d\}$ et d'une table de faits $\mathcal{F}$;
– un $\mathcal{DBC}$ à $\mathcal{M}$ nœuds de traitement $\mathcal{N} = \{N_1, N_2, \ldots, N_M\}$;
– une charge de requêtes de jointure en étoile $\mathcal{Q} = \{Q_1, Q_2, \ldots, Q_L\}$ exécutée sur la grappe $\mathcal{CBD}$, chaque requête Q_l possède une fréquence d'accès f_l;
– une *contrainte de maintenance* $\mathcal{W}$, représentant le nombre de fragments de $\mathcal{EDR}$ que le concepteur considère pertinent pour le processus d'allocation (ce nombre doit être largement supérieur au nombre de nœuds ($\mathcal{W} \succ\succ \mathcal{M}$));
– une *contrainte de réplication* $\mathcal{R}$, telle que $\mathcal{R} \leq \mathcal{M}$, représentant le nombre de copies de chaque fragment que le concepteur considère approprié pour le processus de traitement de requêtes;
– une *contrainte de mauvaise répartition des valeurs d'un attribut* θ, représentant le degré de la distribution non uniforme des valeurs sur les sous-domaines d'un attribut admis par le concepteur pour le sélection des attributs de fragmentation;
– une *contrainte de la mauvaise répartition de données* α, représentant le degré de la mauvaise répartition des données sur les nœuds de traitement que le DWA tolère pour le placement de données;

5. La mauvaise répartition suit la *distribution de Zipf*

6. le centre de gravité représente le point autour duquel la masse est répartie symétriquement. Dans notre ca, le centre de gravité est représenté pat la moyenne des charges

– une *contrainte d'équilibrage de charge* δ, représentant le degré de déséquilibrage de charge que le DWA admet pour le placement des sous-requêtes sur les nœuds de traitement.

Le problème de conception d'un entrepôt de données parallèle sur une grappe consiste à fragmenter la table des faits en N_F fragments et à allouer les fragments générés et leurs R copies simultanément afin de réduire le coût d'exécution de toutes les requêtes et satisfaire toutes les contraintes.

4.3.2 Approche proposée : $\mathcal{F}\&\mathcal{A}\&\mathcal{R}$

Elle consiste à partitionner l'entrepôt de données et à allouer en même temps les fragments d'une manière redondante. Pour la fragmentation, nous adaptons notre algorithme génétique [17] et pour l'allocation de données, nous présentons un nouvel algorithme d'allocation redondante. Les grandes étapes de cet algorithme sont décrites dans l'organigramme ci-dessous.

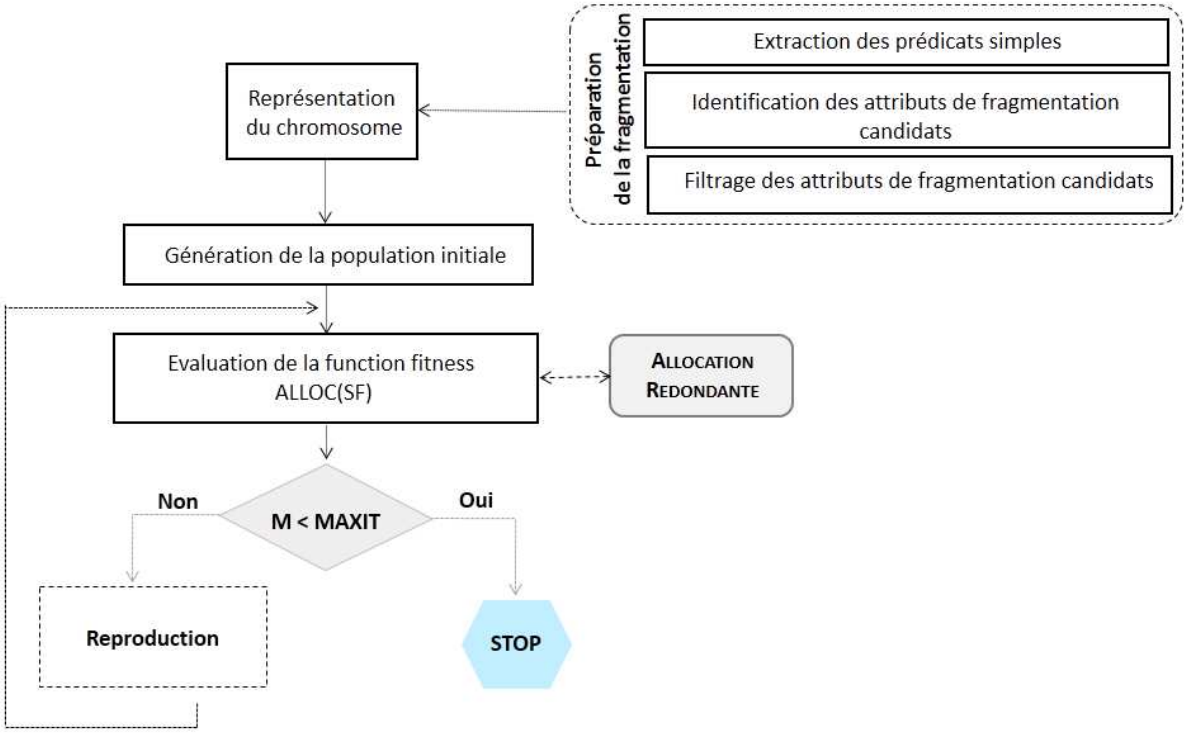


Figure 4.10 – *Organigramme de notre approche* $\mathcal{F}\&\mathcal{A}\&\mathcal{R}$.

4.3.2.1 Procédure de fragmentation

Pour sélectionner le schéma de partitionnement horizontal, l'approche $\mathcal{F}\&\mathcal{A}\&\mathcal{R}$ adopte notre algorithme génétique proposé dans la section 4.2.2.1. L'adaptation de cet algorithme

concerne le choix des attributs de fragmentation candidats et l'ajout d'un opérateur de reproduction.

Pour le choix des attributs de fragmentation, nous avons ajouter une étape qui nous permet d'eliminer les attributs ayant un degré de skew élevé.

Nous avons également utilisé un nouvel opérateur nommé "Low-Skew" . Cet opérateur est utilisé pour minimiser le degré de skew. Pour ce faire, l'opérateur calcule la mauvaise distribution de chaque sous-domaine et équilibre la répartition entre sous-domaines sans influencer le nombre de fragments. L'algorithme 6 décrit les grandes lignes de notre opérateur.

Algorithm 6 *Fonction de "Low-Skew"* (Cr_i Chromosome)

1: **begin**
2: For each fragmentation attribute in chromosome Cr_i calculate the Skewness rate
3: Get the attribute which has the maximum Skewness rate, let's A_{skew} this attribute
4: Get the distinct value of the attribute A_{skew}.
5: Get the indices of the corresponding value
6: Calculate the selectivity factor for each distinct value
7: Let's `vmax` the value corresponding to the sub-domain having the maximum selectivity factor
8: Let's `vmin` the value corresponding to the sub-domain having the minimum selectivity factor
9: Get one cell from the `vmin` to `vmax`
10: **end**

4.3.2.2 Phase d'allocation des fragments

Dans sa forme générique, le problème d'allocation des données consiste à déterminer le meilleur emplacement pour un ensemble de fragments sur les nœuds d'une grappe afin de minimiser le temps de la réponse d'une charge de requêtes. Ce problème peut être assimilé à un *problème de classification* qui consiste à placer un ensemble d'objets de données dans un certain nombre de classes en fonction d'une métrique.

L'allocation des fragments est fortement liée à la réplication des fragments. Pour cette raison, nous considérons une allocation redondante où chaque objet est alloué sur au moins un nœud. Ainsi, chaque objet appartient non pas à une classe, mais à toutes les classes avec un certain degré. Cette approche permet de prendre en compte l'*appartenance partielle à plusieurs classes*.

Nous proposons de formaliser le problème d'allocation comme un problème de *classification floue* qui est un cas particulier pour lequel chaque élément présente un degré d'appartenance. Ces degrés d'appartenance sont entre 0 et 1.

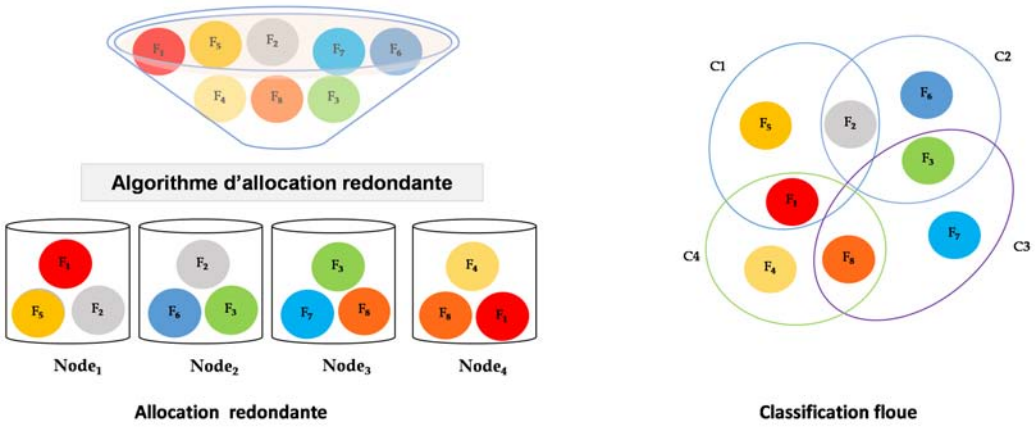


Figure 4.11 – *Allocation vs Classification*

4.3.2.2.1 Formalisation Le principe de classification floue est l'attribution d'éléments de données à plusieurs classes, avec divers degrés d'appartenance. Ainsi, le problème d'allocation des fragments est formalisé comme suit.

Considérons un ensemble de fragments $\mathcal{F} = \{F_1, F_2, \ldots, F_{N_F}\}$ *avec d dimensions dans l'espace euclidien* R^d, *i.e.* $F_j \in R^d$. *Le problème consiste à affecter chaque fragment* F_i *à R (R représente le degré de réplication) sous-ensembles flous en minimisant une* fonction objectif. *Le résultat de la classification floue peut être exprimé par une matrice d'appartenance U dont la valeur* $U[i][j] = u_{ij}$, *telle que* $i = 1..M$ *et* $j = 1..N_F$, *où* u_{ij} *appartient à [0,1] et satisfait la contrainte.*

$$\forall 1 \leq j \leq NF : \sum_{i=1}^{M} u_{ij} = 1. \tag{4.21}$$

La fonction objective f_O à optimiser est définie par

$$f_O = \sum_{k=1}^{NF} \sum_{i=1}^{M} u_{ij}^m \, ||X_k - V_i||^2 \,, \tag{4.22}$$

où $m \succ 1$ est un paramètre contrôlant le degré de flou (généralement m =2), X_k est le vecteur des points de données, V_i est le centre des clusters C_i et $||X_k - V_i||^2$ représente la distance euclidienne entre X_k et V_i.

4.3.2.2.2 L'Algorithme Fuzzy declustering) L'algorithme d'allocation proposé est une adaptation de l'algorithme proposé par Navathe et *al* [109] qui consiste à intégrer l'algorithme c-moyens flous au niveau de la phase de regroupement pour construire des classes redondantes.

L'algorithme des c-moyens flous (fuzzy c-means) est un algorithme de classification floue fondé sur l'optimisation d'un critère quadratique de classification où chaque classe est repré-

sentée par son centre de gravité[34]. L'algorithme nécessite de connaître le nombre de classes au préalable et génère les classes par un processus itératif en minimisant une fonction objectif. Ainsi, il permet d'obtenir une partition floue de l'image en donnant à chaque pixel un degré d'appartenance à une région donnée. Les valeurs des degrés d'appartenance sont regroupées dans une matrice $U = [u_{ik}]$ désigne le degré d'appartenance du pixel i à la classe k.

D'un point de vue global, l'utilisation d'ensembles flous peut se faire essentiellement lorsque les données sont incomplètement spécifiées ou fortement bruitées; ou encore que lorsque certains attributs sont difficilement mesurables avec précision ou difficilement quantifiables numériquement. A ce moment-là, il est naturel de recourir à des ensembles flous pour classifier les données.

Les étapes de notre procédure d'allocation sont les suivants.

4.3.2.2.2.1 Description des données .

Elle peut se faire grâce à la Matrice d'Usage des Fragments (M_U) qui fournit une information numérique concernant les valeurs prises par les attributs d'un objet. Nous rappelons que M_U représente l'utilisation des fragments par les requêtes. Les lignes et les colonnes de cette matrice sont associées aux n requêtes de départ et les N_F fragments obtenus par le schéma de fragmentation en cours d'évaluation, respectivement. La valeur $M_U[i][j]$ telle que ($1 \leq i \leq L$ and $1 \leq j \leq N_F$) est égale à 1, si la requête Q_i utilise le fragment F_j. Sinon, elle est à 0. Nous ajoutons à cette matrice une colonne représentant la fréquence d'accès de chaque requête.

Exemple 11. *Soit $F = \{F_1, F_2, F_3, F_4, F_5, F_6, F_7, F_8\}$ et $Q = \{Q_0, Q_1, Q_2, Q_3\}$ l'ensemble des fragments générés et des requêtes respectivement. La matrice d'usage des fragments de cet exemple est montrée dans la table 4.3.*

Queries	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	f
Q_0	1	1	1	0	1	0	1	0	20
Q_1	1	1	1	1	0	0	0	0	35
Q_2	0	0	1	0	1	1	1	1	30
Q_3	1	1	1	1	1	1	1	1	15

Tableau 4.3 – *Matrice d'Usage des Fragments*

4.3.2.2.2.2 Représentation de chaque fragment dans R^2 .

Chaque fragment F_i est représenté dans l'espace vectoriel à deux-dimension R^2 par les coordonnés (x, y). Les coordonnés du fragment F_i dans R^2 sont établies à partir du poids de classification. Le poids d'un fragment F_i est égal à la somme des fréquences d'accès de toutes les requêtes qui n'utilisent pas le fragment. Une fois le poids calculé, les coordonnées dans R^2 de chaque fragment F_i sont spécifiées comme suit : $(x, y) = (i, poids(F_i))$.

Exemple 12. *La représentation des fragments associés à la Table FUM 4.3 est illustrée dans la figure 4.12.*

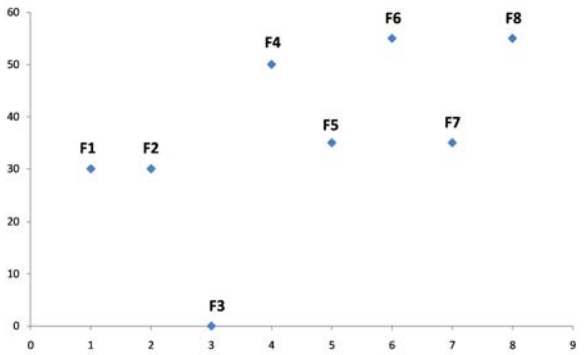


Figure 4.12 – *Représentation des fragments*

4.3.2.2.2.3 Regroupement des attributs .

Les algorithmes de regroupement ont pour but de partager un ensemble de données non étiquetées en c groupes, de telle sorte que les groupes obtenus contiennent des individus les plus semblables possible, tandis que les individus de groupes différents sont le plus dissemblables possible.

L'algorithme de classification floue le plus utilisé et connu est l'algorithme des Fuzzy c-moyennes où on suppose le nombre de classes connu. Le résultat délivré par cet algorithme est alors une matrice MAF ($N_F \times M$) des degrés d'appartenance, où N_F est le nombre de fragments et M le nombre de nœuds (nombre de classes à obtenir). L'algorithme des c-moyennes floues a été étudié essentiellement par Bezdek [25].

La valeur $MAF[i][j]$, telle que ($1 \leq i \leq N_F$ and $1 \leq j \leq M$), appartient à l'intervalle $[0, 1]$. Cette valeur est calculée comme suit:

$$MAF_{ij} = \left[\sum_{l=1}^{N_F} \left(\frac{d_{ij}}{d_{il}}\right)^{\frac{2}{m-1}}\right]^{-1},$$

où: d_{ij} représente la distance euclidienne entre X_k et V_i et m représente le coefficient flou de la partition.

Le principe de base est de former à partir des individus non étiquetés c groupes qui soient les plus homogènes et naturels possible. Homogène et naturel signifient que les groupes obtenus doivent contenir des individus les plus semblables possible, tandis que des individus de groupes différents doivent être les plus dissemblables possible.

Exemple 13. . *A partir de la représentation des fragments illustrée dans la figure 4.12, la MAF associée à cet exemple est montrée dans la table 4.4.*

	C_0	C_1	C_2	C_3
F_1	5,01E-03	2,79E-04	9,94E-01	4,35E-04
F_2	5,92E-03	2,72E-04	9,93E-01	4,29E-04
F_3	3,56E-09	1,00E+00	4,87E-09	1,53E-09
F_4	6,66E-02	6,07E-03	3,75E-02	8,90E-01
F_5	9,70E-01	7,97E-04	2,64E-02	2,84E-03
F_6	5,28E-03	6,94E-04	3,27E-03	9,91E-01
F_7	9,79E-01	7,83E-04	1,77E-02	2,82E-03
F_8	1,37E-02	1,81E-03	8,28E-03	9,76E-01

Tableau 4.4 – *Matrice d'Appartenance des Fragments*

4.3.2.2.2.4 Conception du discriminateur .

La discrimination a pour fonction de produire une partition de l'espace décrivant les données. Cette partition correspond aux classes. Afin de générer ces partitions, nous faisons usage du principe que les valeurs d'appartenance les plus élevées indiquent la confiance d'attribution des objets au cluster. Ainsi, nous avons trié les valeurs d'appartenance dans l'ordre décroissant et nous assignons chaque fragment F_k $(1 \leq k \leq NF)$ aux R (R étant le degré de réplication) premiers groupes, de sorte que la contrainte de placement de données est satisfaite. À la fin de cette étape, un ensemble de clusters $\mathcal{C} = \{C_1, C_2, \ldots, C_M\}$ est généré, de telle sorte que chacun d'eux représente un sous-ensemble de fragments.

Exemple 14. *Les groupes de fragments associés à la matrice MAF sont illustrés dans la figure 4.13.*

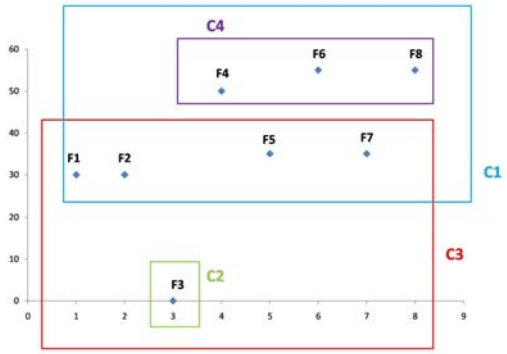


Figure 4.13 – *Regroupement des Fragments associé à la matrice FAM (Table 4.4)*

4.3.2.2.2.5 Construction de la Matrice de Placement des Fragments .

Une fois les partitions créées, cette matrice représente la présence des fragments sur les nœuds. Ses lignes et ses colonnes sont associées aux N_F fragments obtenus par le schéma de fragmentation en cours d'évaluation et les M nœuds associés à la grappe, respectivement. Les éléments de la matrice M_P sont binaires (0 ou 1). $M_P[i][m] = 1$, avec $1 \leq i \leq N_F$ et

$1 \leq m \leq M$, si le fragment F_i est alloué sur le nœud N_m, sinon $M_P[i][m] = 0$. Notre procédure d'allocation considère le cluster comme étant l'unité d'allocation. Les clusters sont placés d'une manière circulaire sur les nœuds.

Exemple 15. *Les clusters générés dans la figure 4.13 sont placés d'une manière circulaire sur les nœuds, la matrice M_P associé est illustrée dans la table 4.5.*

	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8
N_1	1	1	0	1	1	1	1	1
N_2	0	0	1	0	0	0	0	0
N_3	1	1	1	0	1	0	1	0
N_4	0	0	0	1	0	1	0	1

Tableau 4.5 – *Matrice de Placement des Fragments*

Comme un fragment appartient à un seul $\mathcal{R}$ cycles et que le cycle est alloué au niveau d'un seul nœud, nous pouvons dire que l'allocation est redondante vu que

$$\forall 1 \leq i \leq N_F : \sum_{m=1}^{M} FPM_{im} = R. \tag{4.23}$$

4.3.2.2.3 Complexité de l'algorithme Fuzzy declustering .

L'algorithme Fuzzy declustering est constitué de cinq étapes, nous décrivons la complexité de chacune d'elles :

- la phase description de données est basée sur la matrice d'usage des fragments. Sa complexité est équivalente à celle de construction de la matrice d'usage $O(L \times N_F)$ où L et N_F représentent le nombre des requêtes et des fragments, respectivement.
- la complexité de la phase de représentation de données est $O(N_F)$.
- l'algorithme de regroupement est de complexité de $O(N_F \times M)$ vu que l'objectif de cette phase est la construction de la matrice d'appartenance. Notons que M représente le nombre des nœuds .
- la complexité de la phase conception du discriminateur est $O(N_F \times M \times R)$ où R représente le degré de réplication.
- la complexité de la phase de construction de la matrice d'allocation des fragments est $O(k \times M)$ où k représente le nombre des fragments constituons chaque cycles.

La complexité totale de l'algorithme d'allocation est $O(L \times N_F + N_F + N_F \times M + N_F \times M \times R + k \times M)$. Cette complexité est équivalente à $O(N_F \times M \times R)$.

4.3.2.3 Modèle de coût de $\mathcal{F}\&\mathcal{A}\&\mathcal{R}$

Une fois le schéma de fragmentation généré et les fragments alloués, les requêtes globales seront évaluées sur la grappe de base de données $\mathcal{DBC}$. Le principal objectif est de minimiser le temps d'exécution total de l'ensemble des requêtes. Pour évaluer une requête de jointure en étoile, il faut identifier d'abord les fragments valides et leur localisation sur les nœuds. Comme notre allocation est redondante (chaque fragment est alloué sur R nœuds), nous utilisons un ordonnanceur pour trouver l'allocation la plus judicieuse pour chaque sous-requête. Il est à noter que chaque fragment valide donne lieu à une sous-requête. Le problème de traitement parallèle a pour but la minimisation du coût d'exécution total des requêtes de $\mathcal{Q}$ exécutées sur la grappe en satisfaisant la contrainte d'équilibrage de charge δ.

Le problème introduit ci-dessus est similaire à un dual du problème de remplissage *(Dual Bin Packing Problem (DBPP))*. Il est défini comme NP-complet [87]. Ainsi, pour produire une solution quasi-optimale pour ce problème, nous proposons un algorithme glouton (algorithme 1) qui doit trouver le meilleur placement des requêtes minimisant le coût d'exécution moyen de la charge de requêtes. Cet algorithme est détaillé dans le chapitre 3.

Les solutions qui violent l'une des contraintes décrites dans la formulation 4.3.1.2 sont pénalisées.

Conclusion

Nous nous sommes attachés dans ce chapitre à proposer une nouvelle approche de déploiement d'un entrepôt de données parallèle. Cette approche consiste à fragmenter l'entrepôt de données modélisé par un schéma en étoile et à allouer les fragments résultant aux divers nœuds d'une grappe de base de données. Plus précisément, nous avons proposé deux approches $\mathcal{F}\&\mathcal{A}$ et $\mathcal{F}\&\mathcal{A}\&\mathcal{R}$.

Dans un premier temps, nous avons proposé l'approche $\mathcal{F}\&\mathcal{A}$ [17]. Sa principale particularité est la prise en considération de l'interdépendance entre la fragmentation et l'allocation des fragments, elle prend également en considération l'hétérogénéité des nœuds en termes de puissance de calcul et de capacité de stockage. Notons également que l'une des principales caractéristiques distinctives de l'approche $\mathcal{F}\&\mathcal{A}$ est le fait qu'elle peut être facilement adapté par le concepteur en utilisant ses algorithmes préférés pour soutenir les phases de fragmentation et d'allocation. Pour la fragmentation, nous avons proposé deux algorithmes basés sur l'algorithme Hill Climbing et sur l'algorithme génétique. Pour le problème d'allocation, nous avons proposé une variété de l'approche du placement circulaire, $\mathcal{F}\&\mathcal{A}$- `ALLOC`, où au lieu d'allouer un fragment par nœud, nous avons un ensemble de fragments. L'ensemble des groupes de fragments est défini selon le même principe que l'algorithme d'affinité de Navathe [107].

Dans un second temps, nous avons présenté une méthode de déploiement nommée $\mathcal{F}\&\mathcal{A}\&\mathcal{R}$ [23], qui est une variante de $\mathcal{F}\&\mathcal{A}$. Son principal objectif est d'unifier les phases de déploiement d'un EDP. Pour Cela, une méthode de réplication originale, basée sur la logique floue est intégrée dans $\mathcal{F}\&\mathcal{A}\&\mathcal{R}$.

Le modèle de coût évaluant la qualité de notre solution intègre les différents concepts de phases de la conception.

Dans le chapitre suivant, nous présentons les résultats de l'implémentation de ces méthodes en utilisant les bancs d'essai APB-1 et SSB. Nous présentons également les résultats obtenus sur le SGBD parallèle Teradata.

Chapitre 5

Évaluation théorique et réelle sur Teradata

"The true method of knowledge is experiment".
William Blake(1757-1827)

L'objectif principal de ce chapitre est de valider l'ensemble des aspects conceptuels abordés au cours de ces travaux de recherche. Dans notre contexte, il peut être relativement difficile de conduire des expériences de longue durée dans un environnement réel. Nous avons donc choisi de faire usage des expérimentations en utilisant un simulateur pour tester nos algorithmes. L'utilisation d'un simulateur, nous permet de maîtriser l'ensemble des paramètres de la plate-forme simulée, ce qui est peut être impossible dans un environnement réel. Une fois que le bon schéma de déploiement trouvé, un mise en œuvre sur l'environnement réel aura lieu.

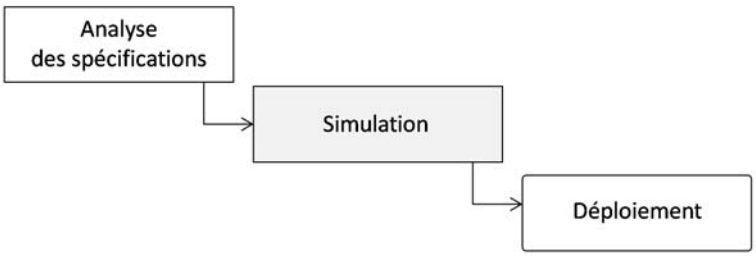


Figure 1 – *Cycle de validation de notre approche*

Pour valider notre approche et illustrer sa faisabilité, nous avons suit le processus décrit en la figure 1. Nous avons développé d'abord un simulateur pour faciliter l'étude expérimentale de notre approche. Puis, nous avons mené des expériences qui nous ont permis d'évaluer nos approches innovantes ($\mathcal{F}\&\mathcal{A}$ et $\mathcal{F}\&\mathcal{AR}$) présentées dans le chapitre précédent. Pour cela, nous présenterons pour chaque expérience toutes les informations concernant les instances utilisées par l'algorithme pour effectuer le déploiement de l'entrepôt de données parallèle, puis le résultat de notre expérimentation. Une validation réelle de l'approche $\mathcal{F}\&\mathcal{A}$ sur le SGBD parallèle *Teradata* est également décrite.

5.1 Le simulateur

Les logiciels de simulation sont des programmes d'applications qui s'exécutent en tant que plateforme graphique pour évaluer la fiabilité sans monopoliser la conception réelle des périphériques physiques ou des processus. Ils utilisent des modèles mathématiques précis qui simulent le fonctionnement du système et les paramètres physiques désirés.

Ces logiciels peuvent fournir des indications sur les processus, les architectures matérielles, les propriétés et les effets sur le système en peu de temps et à faible coût. La simulation peut ainsi améliorer la fiabilité et la disponibilité, elle aide aussi à réduire la durée du cycle de développement en permettant au concepteur de corriger les erreurs et d'identifier les paramètres et les exigences de conception optimisés avant qu'ils n'atteignent le stade de la mise en œuvre réelle. Elle permet la comparaison des mesures de performance pour différentes données de diagnostic. Son importance est de plus en plus soulignée dans le domaine des bases de données, où il est prouvé que son impact sur les coûts, la qualité et la fiabilité est triviale.

Le simulateur développé par nos soins permet d'assister l'administrateur afin de mettre en œuvre un entrepôt de données parallèle sur une grappe de bases de données. Il permet d'effectuer les tâches suivantes:

- charger le schéma en étoile d'un entrepôt de données,
- charger la charge de requêtes ainsi que leurs fréquences d'accès,
- charger les caractéristiques de l'architecture matérielle,
- introduire des paramètres de l'algorithme de fragmentation,
- introduire des paramètres de l'algorithme d'allocation,
- choisir de la stratégie de conception (itérative ou conjointe),
- générer la démarche de déploiement,
- générer des statistiques.

La figure 5.2 décrit les principaux cas d'utilisation de notre simulateur.

5.2 Évaluation de performance de $\mathcal{F}\&\mathcal{A}$

Dans cette section, nous avons mené une série d'expérimentations afin de valider et mettre en évidence le gain apporté par notre approche $\mathcal{F}\&\mathcal{A}$. Nous commençons par la présentation des paramètres d'expérimentation, puis nous présentons les résultats obtenus. Enfin, une validation sous Teradata est exposée.

5.2.1 Paramètres d'expérimentation

Afin d'évaluer attentivement l'efficacité et l'efficience de notre méthodologie proposée, $\mathcal{F}\&\mathcal{A}$, (pour la conception d'un entrepôt de données parallèle sur une grappe de bases de données), nous avons mené une campagne expérimentale intensive. Les algorithmes de l'approche $\mathcal{F}\&\mathcal{A}$ ont été développés dans un environnement Windows 7 en *Java* sur une machine 3 *Intel Pentium Core Duo* à 2,8 $GHzGB$ RAM.

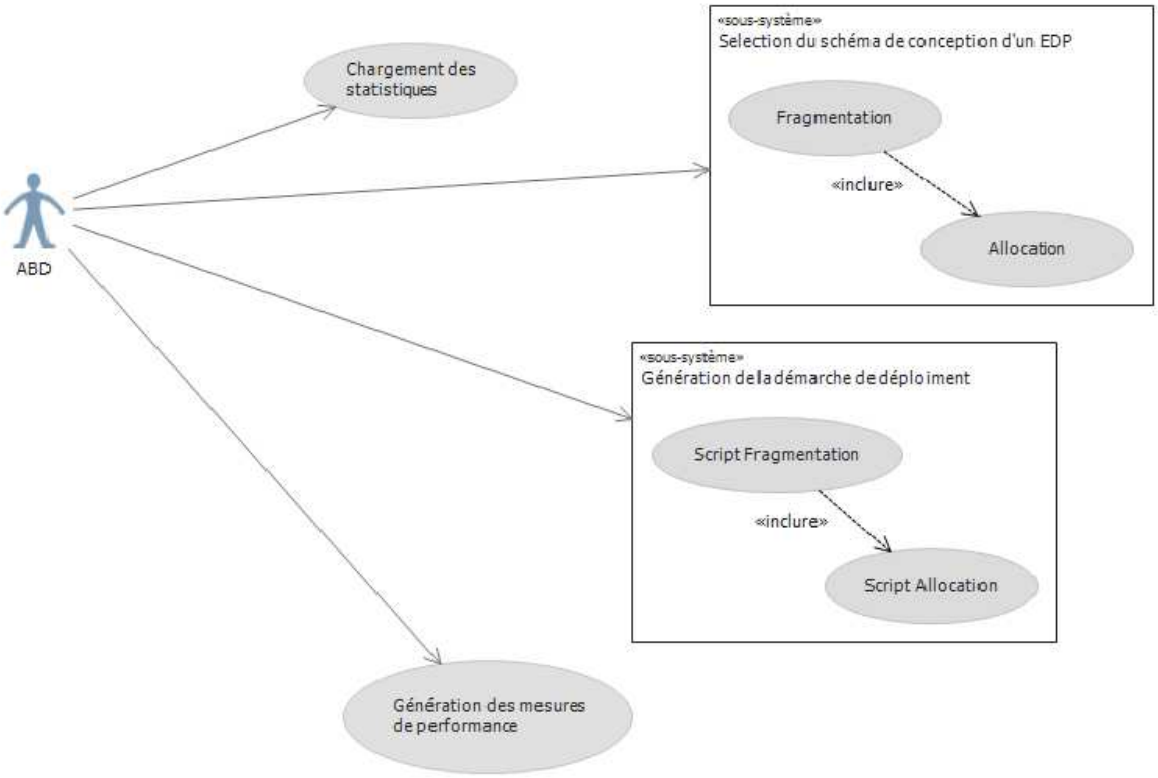


Figure 5.2 – *Cas d'utilisations de notre simulateur*

En ce qui concerne le réglage de notre cadre expérimental, nous avons considéré un environnement de grappe de bases de données simulé avec 128 nœuds. La capacité de stockage et la puissance de traitement de chaque nœud ont été générées selon une distribution aléatoire. Nous obtenons ainsi un environnement de grappe de bases de données totalement hétérogène.

En ce qui concerne la couche de données, nous avons considéré le banc d'essai *APB-1 version II* [49]. Dans le détail, il est caractérisé par une table de faits *Sales* ayant $24,786,000$ tuples, et quatre tables de dimensions, avec un nombre respectif de tuples: *product* (9000 tuples), *customer* (900 tuples), *time* (24 tuples), et *channel* (9 tuples).

En ce qui concerne la couche des requêtes, nous avons considéré une charge de travail de requête étoiles composée de 55 *requêtes mono-block* (les requêtes sans sous-requêtes imbriquées) caractérisées par 40 prédicats de sélection définis sur 9 attributs distincts: *Class*, *Groupe*, *Family*, *Line*, *Division*, *Year*, *Month*, *Retailer* et *All*. Les domaines de ces attributs sont répartis en sous-domaines: 4, 2, 5, 2, 4, 2, 12, 4, 5 respectivement. Dans notre évaluation expérimentale, nous ne considérons pas les requêtes de mise à jour. Notre processus d'allocation est non redondant, donc le coût de mise à jour ne change pas et reste similaire au coût des mises à jour dans un contexte centralisé. Contrairement à cela, si l'allocation redondante est exploitée [117], le coût de mise à jour joue un rôle essentiel. Ces aspects, tout en étant importants, sont en dehors du champ d'application de ce chapitre.

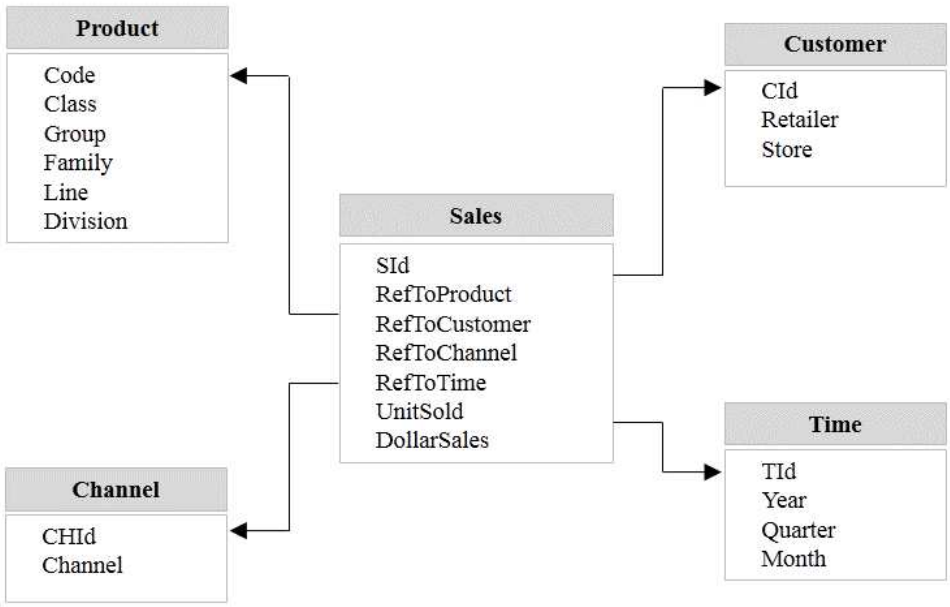


Figure 5.3 – *Schéma logique du banc d'essai APB-1 release II*

Il convient de noter que, généralement, les performances d'une méthodologie de conception d'un EDP sont évaluées selon trois mesures différentes [117]: *temps d'exécution*, *Speed-Up* et *Scale-Up*.

- Le temps d'exécution est défini comme la quantité de temps nécessaire pour évaluer les requêtes d'une charge de travail. Dans un environnement parallèle, le temps d'exécution moyen est égal au nombre total des Entrées/Sorties (E/S) nécessaires pour évaluer les requêtes divisé par la moyenne des *puissances de traitement* de nœuds. De manière plus détaillée, nous avons utilisé les secondes comme unités temporelles de référence pour évaluer la puissance de traitement.

- Le Speed-Up (rapidité) est la dimension la moins paramétrée qui nous fournit une mesure sur le *gain de performance* obtenu par la mise en parallèle d'une application par rapport à son homologue séquentiel. Le Speed-Up a été défini de plusieurs manières. Nous retenons celle qui le défnit comme la métrique qui calcule le *rapport entre les temps d'exécution* de l'application en séquentiel et celui du parallèle nécessaire pour la résolution du problème.

- La Scale-Up (évolutivité) mesure la conservation du temps de réponse d'une requête pour une augmentation proportionnelle de la taille de la base de données et des capacités de la configuration.

Comme dans ces études classiques, nous avons tenu compte de ces paramètres pour souligner l'efficacité et l'efficience de notre méthodologie proposée $\mathcal{F}\&\mathcal{A}$. Pour ce qui concerne les paramètres de l'algorithme génétique, nous avons fixé le nombre de génération à 20, la taille de la population initiale à 40 chromosomes par génération, le taux de croisement de 80% et le

taux de mutation de 20%.

5.2.2 Résultats expérimentaux obtenus

Nous avons effectué plusieurs sortes d'expériences, afin d'obtenir une riche et fiable évaluation expérimentale de $\mathcal{F}\&\mathcal{A}$. Tout d'abord, nous l'avons comparé à la méhode de conception itérative, où la fragmentation et l'allocation sont exécutées séquentiellement et sans aucune itération, déployée sur un environnement de grappe de bases de données hétérogènes.

Pour les deux méthodes de conception de l'EDP, nous fixons comme solution initiale *une distribution aléatoire* des valeurs des attributs de partitionnement candidat.

Comme $\mathcal{F}\&\mathcal{A}$, la fragmentation de l'approche itérative est basée sur les heuristiques HC [55] et GA [78] et l'allocation des fragments générés est faite avec l'algorithme $\mathcal{F}\&\mathcal{A}$-`ALLOC` . En ce qui concerne $\mathcal{F}\&\mathcal{A}$, nous avons fixé le seuil de fragmentation W à 500 et nous avons mesuré le temps d'exécution des requêtes par rapport à la variation du nombre des nœuds de la grappe M sur l'intervalle [2 : 128]. La figure 5.4 montre les résultats obtenus à partir de la première expérience, et nous confirme que l'approche conjointe surpasse l'approche itérative d'une manière significative. D'après les résultats obtenus, nous observons aussi que l'approche conjointe $\mathcal{F}\&\mathcal{A}$ basée sur GA comme algorithme de fragmentation surpasse $\mathcal{F}\&\mathcal{A}$ qui utilise l'algorithme HC comme algorithme de fragmentation.

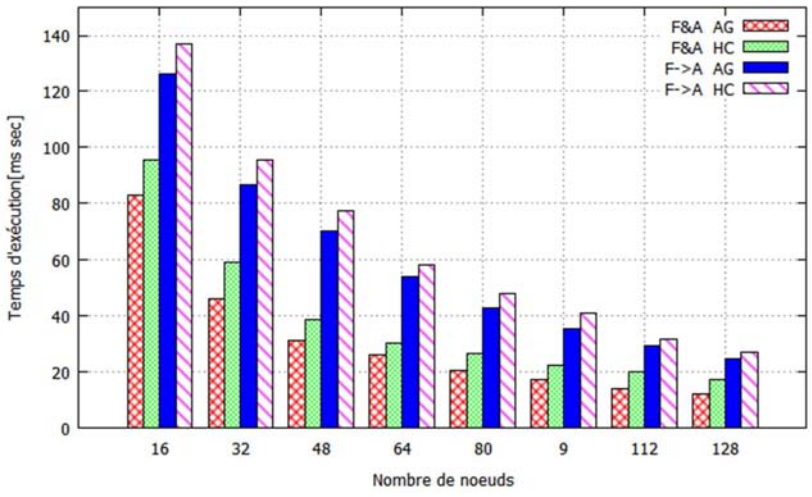


Figure 5.4 – *Approche conjointe vs approche séquentielle*

Dans la deuxième expérience, nous avons mis l'accent sur la rapidité et la scalabilité de l'approche $\mathcal{F}\&\mathcal{A}$. Nous avons examiné quatre scénarii d'application différents: $\mathcal{F}\&\mathcal{A}$ en fonction de GA; $\mathcal{F}\&\mathcal{A}$ en fonction de HC; une approche séquentielle classique basée sur GA; une approche séquentielle classique basée sur HC. Nous gardons les mêmes paramètres que la première expérience, $W = 500$ et $M \in [2 : 10]$. La figure 5.5 montre les résultats obtenus, en

particulier, ce qui concerne le facteur de rapidité. La figure 5.6 montre la mise à l'échelle de $\mathcal{F}\&\mathcal{A}$ et l'approche séquentielle classique.

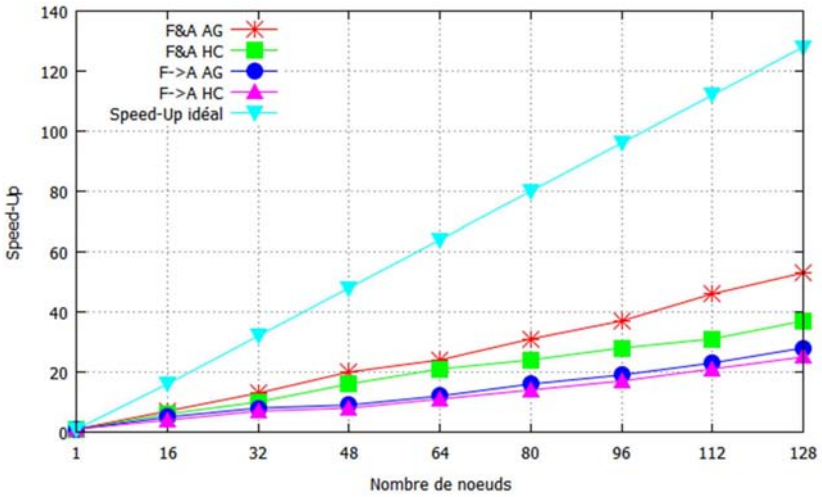


Figure 5.5 – *Speed-Up de $\mathcal{F}\&\mathcal{A}$ vs Speed-Up de l'approche itérative*

Pour ce qui concerne l'analyse de l'évolutivité, nous avons considéré un seuil de fragmentation de 200. Initialement le nombre de nœuds de la machine parallèle M est initialisé à 2 et la table de fait à 24786000 tuples. Ensuite, nous faisons varier la taille de la table de faits et le nombre de nœuds proportionnellement. Pour chaque valeur, nous calculons le facteur de passage à l'échelle résultant de l'exécution des 55 requêtes.

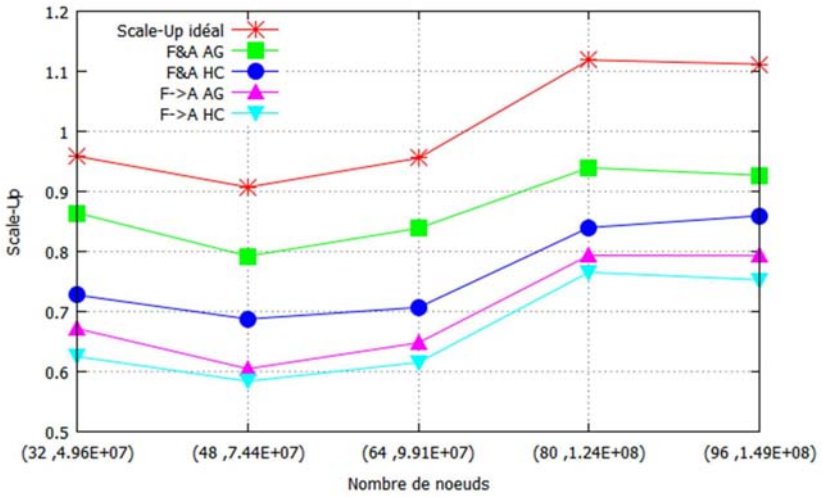


Figure 5.6 – *Scale-Up de $\mathcal{F}\&\mathcal{A}$ vs Scale-Up de l'approche itérative*

A partir de l'analyse des résultats obtenus, nous pouvons dire que la methode de conception $\mathcal{F}\&\mathcal{A}$ est évolutive, ce qui est une contribution remarquable de notre étude. Nous remarquons également que le speed up et le scale up obtenus possèdent une tendance *sub-linéaire*. Cela est dû à deux facteurs qui peuvent probablement empêcher l'obtention d'une vitesse linéaire : *la mauvaise répartition* de charges entre les nœuds de traitement de la machine parallèle et *le*

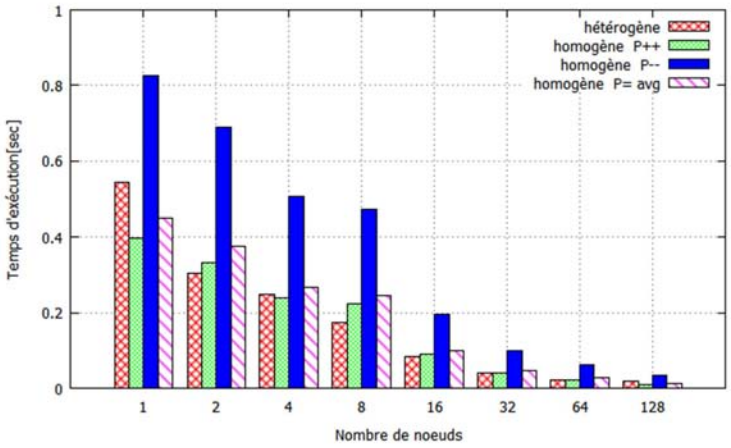
temps de collecte et de consolidation des résultats obtenus au niveau des nœuds de traitement par le nœud coordinateur. Le deuxième facteur étant une conséquence du premier facteur.

Dans la troisième expérience, nous avons mis l'accent uniquement sur $\mathcal{F}\&\mathcal{A}$. Nous avons évalué expérimentalement la configuration idéale d'une grappe hétérogène qui peut remplacer la configuration d'une grappe homogène. Pour cette fin, nous avons observé $\mathcal{F}\&\mathcal{A}$ dans quatre scénarii d'application différents qui peuvent survenir dans les environnements de clusters réels:

- un environnement grappe hétérogène dont sa puissance de calcul moyenne est notée $AVGHetP$,
- un environnement grappe homogène, appelé $P{+}{+}$, dont la capacité de calcul $AVGHomP$ est supérieure à la moyenne des puissances de calcul de l'environnement hétérogène $AVGHetP$ ($AVGHomP \succ\succ AVGHetP$),
- un environnement grappe homogène, appelé $P{-}{-}$, dont la capacité de calcul $AVGHomP$ est inférieure à la moyenne des puissances de calcul de l'environnement hétérogène $AVGHetP$ ($AVGHomP \prec\prec AVGHetP$),
- un environnement grappe homogène, appelé $P = AVG$, dont la capacité de calcul $AVGHomP$ est égale à la moyenne des puissances de calcul de l'environnement hétérogène $AVGHetP$ ($AVGHomP == AVGHetP$).

Pour tous les scénarii, nous avons supposé que la capacité de stockage satisfait la l'hypothèse

$$\sum_{m=0}^{M-1} S_m > Taille(DW), \tag{5.1}$$



(a)

Figure 5.7 – *Effet de l'hétérogénéité de la grappe sur la performance du système*

La figure 5.7 illustre les résultats obtenus et montre que l'approche $\mathcal{F}\&\mathcal{A}$ atteint le meilleur score de performance dans le cas du scénario $P++$, comme prévu. D'autre part, notons un phénomène intéressant et garanti: *la performance sur les environnements des grappes de bases de données hétérogènes surpasse la performance sur les deux scénarii restants, soit* $P--$ *et* $P = AVG$. Ainsi, nous pouvons conclure qu'une infrastructure hétérogène $AVGHetP$ peut remplacer la grappe de base de données homogène correspondant ayant une puissance de

calcul moyenne de $AVGHomP$ tel que $AVGHomP = AVGHetP$. C'est l'observation clé qui confirme clairement les avantages découlant de la méthode proposée $\mathcal{F}\&\mathcal{A}$.

Dans la quatrième expérience, nous avons examiné la performance de l'approche $\mathcal{F}\&\mathcal{A}$. Dans un premier temps, nous étudions l'importance de la prise en considération de la puissance de calcul dans le modèle de coût. Ainsi, la phase d'allocation de $\mathcal{F}\&\mathcal{A}$ a été effectuée en tenant compte de la puissance de calcul de nœuds dans le modèle du coût, alors que dans le second scénario la puissance de traitement de nœuds n'a pas été considérée. La figure 5.8 montre les résultats obtenus et confirme l'utilité de la prise en compte de la puissance de calcul.

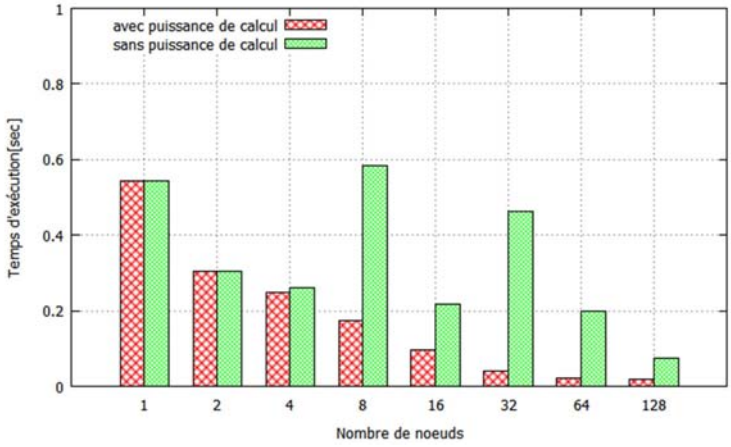


Figure 5.8 – *Effet de la puissance de calcul sur la performance du système*

Dans un second temps, nous étudions l'utilité de la prise en considération de la capacité de stockage, nous avons examiné deux scénarii liés à ce facteur critique, c'est à dire des environnements de la grappe de bases de données (hétérogènes) tels que les nœuds sont caractérisés par une grande capacité de stockage et des environnements de grappes de bases de données (hétérogènes) tels que les nœuds sont caractérisés par une faible capacité de stockage. Comme le montre la figure 5.9, $\mathcal{F}\&\mathcal{A}$ fonctionne mieux lorsqu'une grande capacité de stockage est prévue au niveau des nœuds.

Enfin, dans la cinquième expérience, nous étudions la performance de notre approche par rapport à la contrainte de maintenance W. Nous avons fait varier W dans l'intervalle $[100-500]$ en utilisant 40 prédicats et pour chaque valeur de W, nous exécutons l'algorithme sous la grappe de machines à 10 nœuds et nous calculons le temps d'exécution nécessaire pour le traitement de la charge de requêtes et le pourcentage de réduction du coût total de traitement de la charge de requête. La figure 5.10 et la figure 5.11 montrent les résultats obtenus pour les deux expériences. De ces résultats, il ressort clairement que l'augmentation du seuil améliore généralement la performance des requêtes car en relâchant W, plus d'attributs sont utilisés pour fragmenter l'entrepôt. Lorsque W est grand, les domaines sont décomposés en plus de partitions et donc chaque partition est moins volumineuse. Cela implique moins de données chargées pour exécuter les requêtes utilisant les attributs de fragmentation. Les résultats obte-

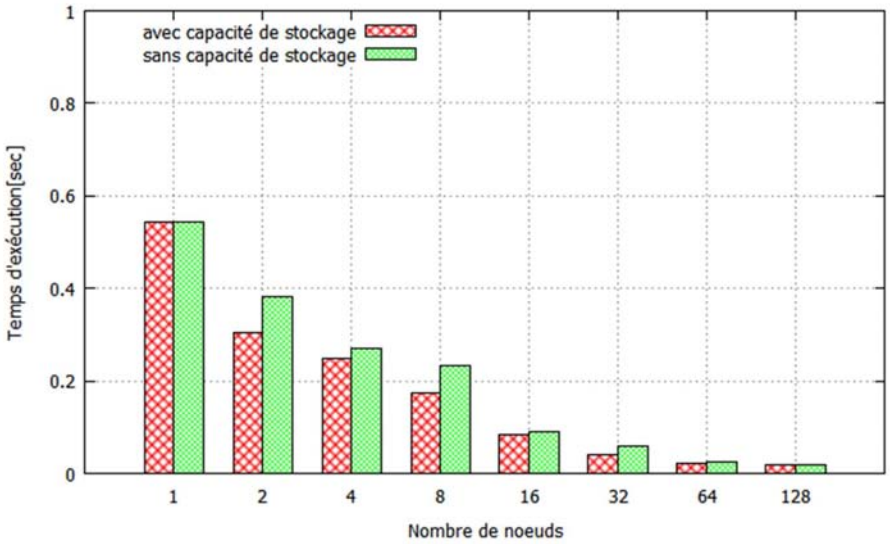


Figure 5.9 – *Effet de la capacité du stockage*

nus montrent que le nombre de fragments et d'E/S sont proportionnels au nombre d'attributs de fragmentation utilisés et au nombre de nœuds du cluster. Il convient de noter également que la performance de $\mathcal{F}\&\mathcal{A}$ se stabilise à partir de la valeur de $W = 400$. Ce résultat expérimental nous confirme l'importance de bien choisir le nombre final de fragments qui peuvent être générés.

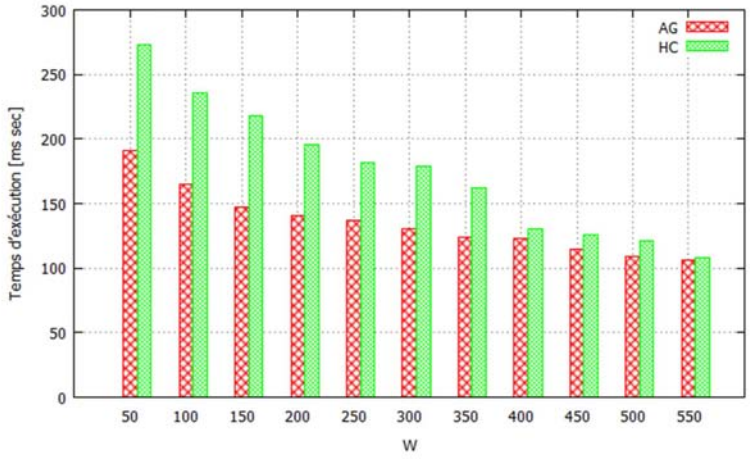


Figure 5.10 – *Effet du Seuil de Fragmentation W sur la Performance de l'approche* $\mathcal{F}\&\mathcal{A}$

Maintenant, nous avons pu estimer que nos algorithmes donnent de bons résultats, mais cela ne reste que des simulations donc il faut valider notre approche sur un environnement réel. Dans la section suivante, nous testons la performance du meilleur schéma de déploiement trouvé sur la plateforme Teradata.

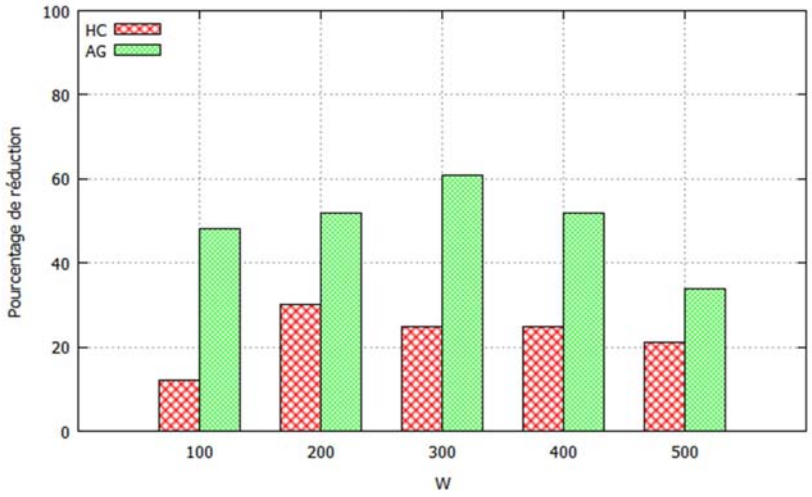


Figure 5.11 – *Pourcentage de réduction du coût de traitement*

5.2.3 Validation sous Teradata

Cette section présente les résultats d'une évaluation expérimentale de notre approche sur un système de bases de données parallèles réel.

Nous allons tout d'abord décrire le système, les ensembles de données, la charge de requêtes, et les mesures d'évaluation qui caractérisent nos expériences et ensuite nous concentrons sur l'analyse des résultats obtenus.

5.2.3.1 Données et charge de requêtes.

Afin de valider l'approche proposée, nous avons mené une série d'expérimentations basée sur le banc d'essai Star Schema Benchmark (SSB) qui est le schéma en étoile du banc d'essai TPC-H (http://www.tpc.org). SSB [113] comporte une table des faits LINEORDERS qui résulte de la fusion des tables LINEITEM and ORDERS de TPC-H et quatre tables de dimension PART, SUPPLIER, CUSTOMER et DATE. Son schéma logique est illustré dans la figure 5.12.

SSB ne précise pas d'autre indexes ou contrainte d'intégrité sur la base de données que l'identification des clés primaires sur les tables de dimension. Nous choisissons cette référence particulière car elle modélise un scénario DW réaliste et dispose d'un schéma en étoile que nous considérons dans notre travail. En particulier, nous avons généré plusieurs instances des données en utilisant le générateur de données fourni avec SSB. La taille de chaque instance est commandé par un facteur d'échelle (Scale Factor), noté par SF . Une valeur SF = X se traduit par un ensemble de données de taille XGB, avec 94% des données représentent des tuples de faits. Nous avons limité la valeur maximale de SF à 10 pour assurer l'exécution rapide de la charge de requêtes sur notre machine expérimentale unique. Nous avons généré 5,5 Go de données, une table de faits de 5,4 Go (59 986 052 tuples), et le total des tables de dimension est de 105 Mo. Le nombre de lignes sont donnés dans le tableau 5.1.

L'ensemble des requêtes proposées par SSB comporte 13 requêtes. Nous avons généré des

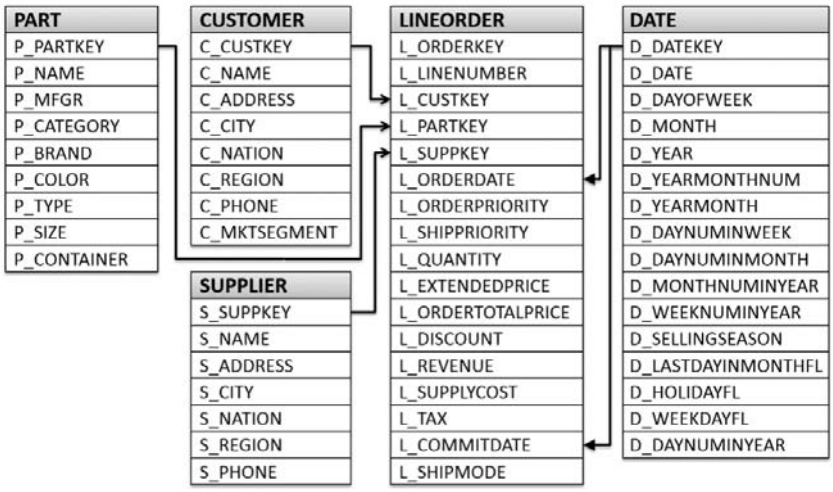


Figure 5.12 – *Schéma logique du banc d'essai SSB*

Table	Nombre de tuples
Lineorder	59986052
Part	800000
Customer	300000
Supplier	20000
Date	2556

Tableau 5.1 – *Cardinale des tables de* SSB

charges de requêtes de jointure en étoile à partir des requêtes proposées par SSB. Nous avons également exclu les requêtes Q1.1, Q1.2 et Q1.3 de la charge de travail car elles contiennent des prédicats de sélection qui concernent des attributs de table de faits. Cette fonctionnalité n'est pas prise en compte par notre approche. La modification n'affecte pas la généralité et la fiabilité de la charge des requêtes générées. De façon plus détaillée, en ce qui concerne les expériences présentées dans ce chapitre, nous avons augmenté la charge des requêtes à 2 fois. Les 22 requêtes obtenues proviennent des 13 requêtes originales en variant les valeurs des prédicats de sélection (la spécification de chaque requête de la charge est décrite dans l'annexe A. 1). Plus précisément, nous avons d'abord converti chaque requête de référence en une requête paramétrée en remplaçant chaque prédicat dans la requête par un paramètre, par exemple, s_region = "UNITED STATES" est converti en $s\_region = Reg$, où Reg est un paramètre. Pour obtenir une charge de requête, il suffit de remplacer les paramètres des prédicats par des valeurs. Nous avons utilisé 50 prédicats de sélection définis sur 11 différents attributs: c_region, c_nation, c_city, $p\_category$, p_brand, p_mfgr, $d\_yearmonth$, d_year, s_region, s_nation et s_city. Selon la charge de requêtes, les domaines de ces attributs sont divisés en sous-domaines: 5, 2, 3, 3, 5, 3, 2, 4, 5, 5 et 3, respectivement pour exécuter les algorithmes génétiques pour des approches conjointe et séquentielle. La table 5.2 dessine notre schéma de fragmentation candidat.

c_region	Africa	Middle est	America	Asia	Europe
c_nation	USA	ELSE			
c_city	UK11	UK15	ELSE		
p_category	$MFRG\#12$	$MFRG\#14$	ELSE		
p_brand	$MFRG\#2221$	$< MFRG\#2227$	$MFRG\#2228$	$MFRG\#2239$	ELSE
p_mfgr	$MFRG\#1$	$MFRG\#2$	ELSE		
d_yearmonth	Dec1997	ELSE			
d_year	1992-1993	1994-1995	1996-1997	1998	
s_region	Africa	Middle est	America	Asia	Europe
s_nation	USA	Egypt	Algeria	Canada	ELSE
s_city	UK11	UK15	ELSE		

Tableau 5.2 – *Schéma de fragmentation candidat*

5.2.3.2 Architecture matérielle

Teradata fournit des serveurs massivement parallèle(Massively Parallel Processing), conçus selon une architecture sans partage (shared nothing architecture) intégrant des serveurs Intel et une interconnexion redondante à haute vitesse le BYNET. L'unité de base du parallélisme dans Teradata est un processeur virtuel nommé Processeur d'Accès Modulaire (Access Module Processor (AMP)). Chaque AMP exécute les fonctions des SGBD sur ses données. Ainsi, le verrouillage et le contenu du cache ne sont pas partagés, ce qui assure la scalabilité. Son moteur de bases de données offre des caractéristiques de puissance et d'évolutivité particulières (prévisibilité, linéarité) quels que soient les paramètres d'évolution de la charge de travail qui lui est demandée, le volume et la variété des données, le nombre d'utilisateurs, la complexité des requêtes,... etc . La figure 5.13 illustre l'architecture détaillé de Teradata à deux nœuds.

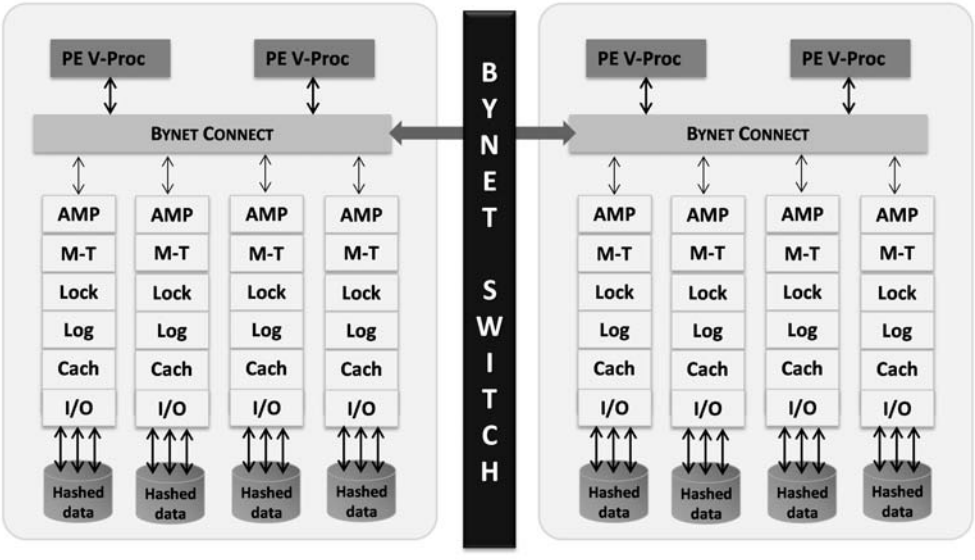


Figure 5.13 – *Architecture de Teradata*

Les systèmes Teradata fonctionnent sous plusieurs systèmes d'exploitation: NCR UNIX SVR4.2 MP-RAS, une variante de System *V UNIX d'AT&T Microsoft Windows 2000* et Windows Server 2003 SuSE Linux Enterprise Server pour les serveurs 64-bits d'Intel

Les entrepôts de données Teradata sont alimentés en batch ou en quasi temps réel grâce à des programmes spécifiques souvent fondés sur des ETL. Ils sont souvent accédés via ODBC ou JDBC par des applications fonctionnant sous Microsoft Windows ou UNIX.

Pour notre validation, nous avons déployé notre approche sur 12 nœuds Teradata.

5.2.3.3 Etapes de validation

Pour mettre en œuvre le schéma de fragmentation et d'allocation de l'approche séquentielle et conjointe sous Teradata nous avons procédé comme suit.

5.2.3.3.1 Génération des schémas de fragmentation et d'allocation Les algorithme séquentiel et conjoint sont appliqués sur la charge de requêtes générée. Nous avons utilisé le modèle de coût décrit dans le chapitre 3 pour sélectionner le schéma de fragmentation et d'allocation. Nous avons utilisé l'algorithme génétique comme un algorithme de base. Ce choix se justifie par le fait que les résultats de la partie de simulation ont prouvé que l'algorithme génétique est meilleur par rapport à l'algorithme Hill Climbing. Après le lancement de la simulation dans un environnement centralisé nous avons obtenu les deux schéma de fragmentation conjoint et séquentiel sont illustrés dans la table 5.3 et la table 5.4 respectivement.

$$PAJoint = \{c\_region; c\_city; d\_yearmonth; s\_region; n\_nation; s\_city\} \tag{5.2}$$

c_region	1	2	2	3	3
c_nation	1	1			
c_city	1	1	2		
p_category	1	1	1	1	1
p_brand	1	1	1	1	1
p_mfgr	1	1	1		
d_yearmonth	1	0			
d_year	1	1	1	1	
s_region	1	1	1	2	1
s_nation	1	1	1	1	2
s_city	1	1	2		

Tableau 5.3 – Schéma de fragmentation de l'approche conjointe

$$PASequential = \{c\_region; p\_catégory; d\_year; s\_region; n\_nation\} \tag{5.3}$$

En conséquence, nous remarquons qu'il y a des attributs communs utilisés par les deux schémas:

$$PAJoint \bigcap PASequential = \{c\_region; s\_nation; d\_year\}. \tag{5.4}$$

c_region	1	3	3	3	2
c_nation	1	1			
c_city	1	1	1		
p_category	1	1	1	1	1
p_brand	1	1	1	2	3
p_mfgr	1	1	1		
d_yearmonth	1	1			
d_year	1	2	1	1	
s_region	1	1	1	2	1
s_nation	1	2	1	2	1
s_city	1	1	1		

Tableau 5.4 – *Schéma de fragmentation de l'approche itérative*

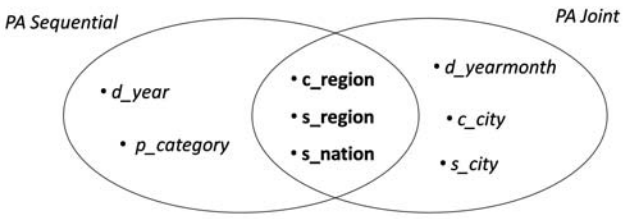


Figure 5.14 – *Attributs de fragmentation pour le deploiement sous Teradata*

5.2.3.3.2 Mise en œuvre des schémas obtenus sur Teradata Les résultats théoriques obtenus à partir de nos algorithmes sont mis en œuvre pour Teradata comme suit:

- les tables de dimensions sont distribuées sur les nœuds selon une fonction de hachage appliquée sur leur clé primaire.
- la table de faits est partitionnée selon le schéma de fragmentation obtenu par l'algorithme de simulation. Chaque fragment est représenté par une table séparée. Ces fragments ont été ensuite alloués sur les AMPs en utilisant une fonction de hachage qui reflète le schéma d'allocation.
- enfin, la table de faits LINEORDER est définie comme une vue avec UNION de ces fragments

5.2.3.3.3 Résultats obtenus Une fois que les schémas de fragmentation et d'allocation sont déployés, les requêtes sont exécutées l'une après l'autre sur Teradata. Le tableau 5.5 montre le temps d'exécution (en secondes) des requêtes pour les approches de conception.

5.2.3.4 Analyse des résultats obtenus

Les résultats obtenus montrent que l'approche conjointe dépasse largement l'approche itérative avec 38% d'avantage. Cependant, nous remarquons que chaque approche est meilleure par rapport à l'autre pour certaines requêtes; cela est dû au fait qu'un schéma de partitionnement n'est pas bénéfique pour toutes les requêtes de la charge de travail. En analysant les résultats obtenus de la machine Teradata, trois scénarii se présentent.

Queries	Joint	Sequential
Q01.1	0.12	0.59
Q01.2	0.11	0.11
Q01.3	0.12	0.11
Q04.1	0.60	0.58
Q04.2	0.54	0.53
Q04.3	1.08	1.13
Q05.0	0.36	0.15
Q06.0	0.53	0.13
Q07.0	0.46	0.58
Q08.0	0.14	2.15
Q09.0	0.08	0.14
Q10.0	0.07	0.16
Q11.0	0.32	0.62
Q12.0	0.34	0.72
Q13.0	0.18	0.63
Q14.0	0.18	0.24
Q15.0	0.18	0.57
Q16.0	0.18	0.51
Q17.0	0.19	0.24
Q18.0	0.29	0.64
Q19.0	0.56	0.66
Q20.0	0.49	0.33
Total	**7.12**	**11.52**

Tableau 5.5 – *Temps d'exécution en Seconde*

L'approche conjointe et l'approche itérative sont équivalentes. Elles sont équivalant en termes de temps de réponse pour certaines requêtes. Cela est dû au fait que les données impliquées par la requête sont distribuées de la même manière sur les nœuds de traitement. Autrement dit, le nœud le plus surchargé a la même quantité de données à traiter. Pour illustrer cette situation, nous considérons la requête Q01.2 exprimée en SQL:

```
select sum(lo_extendedprice*lo_discount) as revenue
from lineorder lo, DATE d}
where l.lo_orderdate=d.d_datekey
and d.d_year in ('1994')
group by year
```

Selon le schéma de fragmentation obtenu par l'approche séquentielle, cette requête nécessite le chargement de 36 fragments (03 *CUSTOMER*, 03 *SUPPLIER* et 04 *PART*). Par contre, l'approche conjointe implique le chargement de tous les 96 fragments obtenus. D'après la figure 5.15 , nous remarquons que le temps d'exécution de Q01.2 correspond au temps de traitement des données au niveau des nœuds N_9 et N_{10} respectivement pour l'approche séquentielle et conjointe. La quantité de données à traiter au niveau des deux nœuds est similaire, ce qui implique un temps de traitement similaire (Les nœuds de teradata sont homogènes).

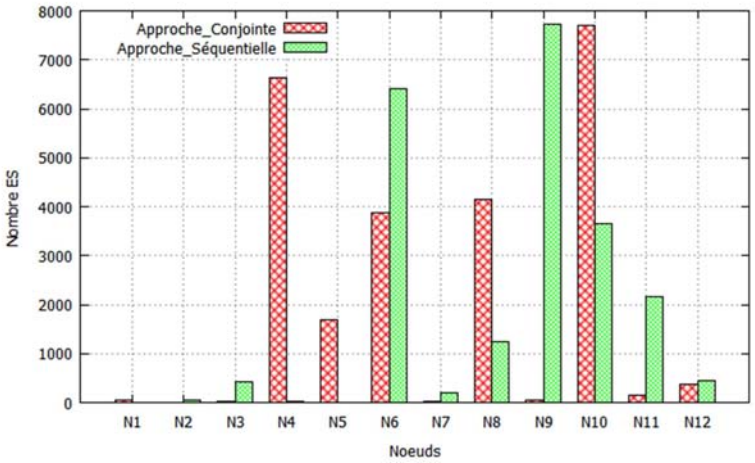


Figure 5.15 – *Répartition des Charges de la requête Q1.2*

L'approche conjointe est meilleure que l'approche itérative. Elle est dans 3 cas possibles.

1. Les deux approches utilisent les mêmes attributs de partitionnement, attribut de partitionnement de $PAJoint \bigcap PASequential$ (Voir figure 5.14), et leur schéma de partitionnement est similaire. Pour certaines requêtes de ce type, l'approche conjointe surmonte l'approche séquentielle car son schéma d'allocation est meilleur que celui de l'approche itérative. Par exemple, prenons la requête Q04.3 exprimée en SQL.

```
select sum(lo_revenue), d_year, p_brand
from lineorder l, DATE d, part p, supplier s
where l.lo_orderdate=d.d_datekey
and l.lo_partkey=p.p_partkey
and l.lo_suppkey=s.s_suppkey}
and p.p_category='MFGR#12'
and (s.s_region='ASIA' or s.s_region='MIDDLE EAST' or s.s_region='EUROPE')
group by d.d_year, p.p_brand
order by d.d_year, p.p_brand;
```

   Pour l'approche séquentielle, cette requête utilise 48 fragments de faits (03 CUSTOMER, 02 DATE, 02 SUPPLIER et 04 PART). D'autre part, l'approche conjointe utilise également 48 fragments de faits (06 CUSTOMER, 02 DATE, 04 SUPPLIER et 01 PART). Comme illustré dans la figure 5.16, nous constatons que le coût d'exécution impliqué par l'approche conjointe est inférieure celui de l'approche séquentielle; cela est dû au schéma d'allocation différent des approches.

2. Les deux approches utilisent les mêmes attributs de partitionnement mais leur schéma de partitionnement est différent. Le partitionnement différent d'un domaine implique des fragments de taille différente. L'optimiseur de requête charge des fragments de grande taille. Par exemple, nous considérons la requête Q11.0 exprimée en SQL.

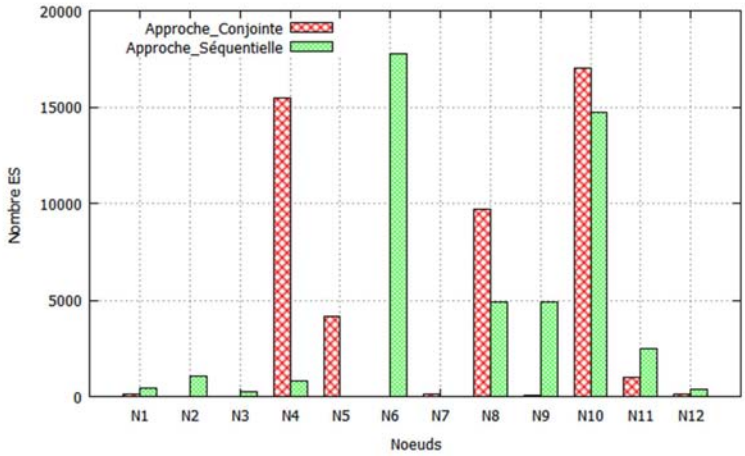


Figure 5.16 – *Répartition des Charges de la requête Q4.3*

```
select d_year, s_nation, sum(lo_revenue - lo_supplycost) as profit}
from DATE d, customer c, supplier s, part p, lineorder l
where l.lo_custkey=c.c custkey
and l.lo_suppkey=s.s_suppkey
and l.lo_partkey=p.p partkey
and l.lo orderdate=d.d datekey}
and c.c region='AMERICA'
and s.s region='AMERICA'
and (p.p mfgr='MFGR#1' or p.p mfgr='MFGR#2')}
group by d.d_year, c.c_ation
order by d.d_year, c.c_nation
```

La figure 5.17 montre la distribution des 16 fragments de faits impliqués pour l'exécution de la requête Q11.0 pour les deux approches. Nous remarquons que le makespan impliqué par l'approche conjointe est inférieur à celui de l'approche séquentielle, cela en raison de la taille du fragment.

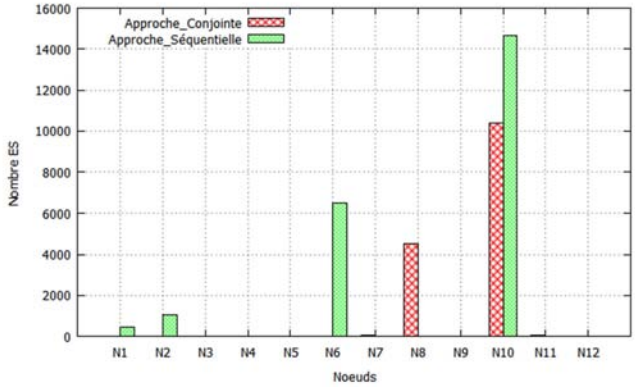


Figure 5.17 – *Répartition des Charges de la requête Q11*

3. Les requêtes référençent des attributs appartenant à l'ensemble. Par exemple, prenons Q08.0 requête exprimée en SQL.

```
select c_city, s_city, d year, sum(lo_revenue) as revenue
from customer c, lineorder l, supplier s, DATE d
where l.lo_custkey=c.c_custkey
and l.lo_suppkey=s.s_suppkey
and l.lo_orderdate= d.d_datekey
and (c.c_city='UNITED KI1' or c.c_city='UNITED KI5')
and (s.s_city='UNITED KI1' or s.s_city='UNITED KI5')
and d.d_yearmonth = 'Dec1997'
group by c.c_city, s.s_city, d.d_year
order by d.d_year asc, revenue textbf{desc};
```

le schéma de fragmentation obtenu par l'approche conjointe référence seulement 12 fragments pour traiter la requête Q08.0. En revanche, l'approche séquentielle référence tous les fragments de faits. La figure illustre la distribution des données à traiter

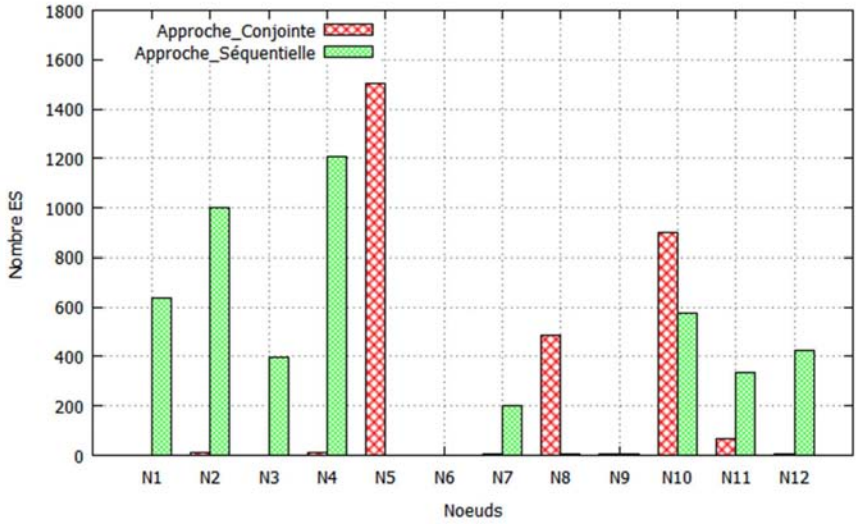


Figure 5.18 – *Répartition des Charges de la requête Q8*

L'approche itérative est meilleure que l'approche conjointe. Elle l'est dans les cas duals du cas 2. Soit la requête Q5 exprimée en SQL.

```
select sum(lo_revenue), d_year, p_brand
from lineorder l, .DATE d, part p, supplier s
where l.lo orderdate=d.d datekey
and l.lo_partkey=p.p_partkey
and l.lo_suppkey=s.s_suppkey
```

```
and p.p_brand in ('MFGR\#2221','MFGR\#2222', 'MFGR\#2223','MFGR\#2224',
                  'MFGR\#2225', 'MFGR\#2226','MFGR\#2227','MFGR\#2228')
and s.s_region='ASIA'
group by d.d_year, p.p_brand
order by d.d_year, p.p_brand
```

Comme illustré dans la figure 5.19 le temps d'exécution impliqué par l'approche séquentielle est inférieur car elle implique moins de fragments.

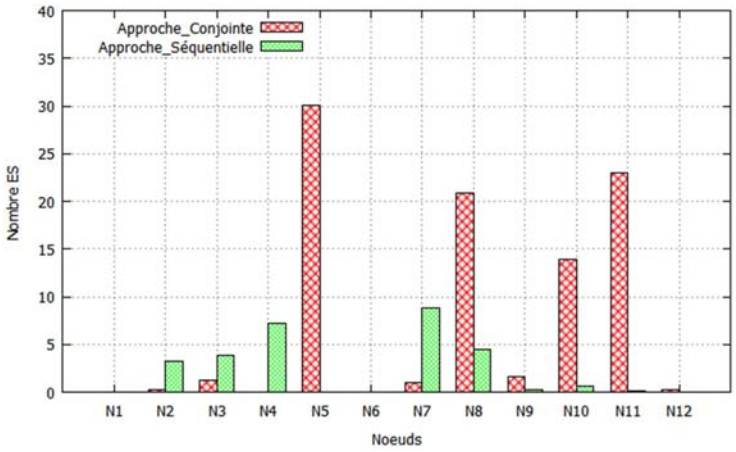


Figure 5.19 – *Répartition des Charges de la requête Q5*

5.3 Évaluation de performance de $\mathcal{F}\&\mathcal{A}\&\mathcal{R}$

Une série d'expérimentations nous a permis de valider et de mettre en évidence le gain apporté par notre approche $\mathcal{F}\&\mathcal{A}\&\mathcal{R}$. Nous commençons par la présentation des paramètres d'expérimentation puis nous présentons les résultats obtenus.

5.3.1 Paramètres d'expérimentation

Afin de valider l'approche proposée, nous avons mené une série d'expérimentations basée sur le banc d'essai Star Schema Benchmark (SSB) [113]. En particulier, nous avons généré plusieurs instances des données en utilisant le générateur de données fourni avec SSB où nous avons limité la valeur maximale de SF à 100, en vue d'assurer l'exécution rapide de la charge de travail de test sur une machine expérimentale unique.

En ce qui concerne la charge des requêtes, nous avons augmenté la charge des requêtes à 4 fois, les 36 requêtes obtenues proviennent des 10 requêtes originales en variant les valeurs des prédicats de sélection. Nous avons utilisé 20 prédicats de sélection définis sur 8 différents

attributs: $\{s\_region, d\_year, s\_nation, c\_city, c\_region, s\_city, p\_category, c\_nation\}$. Les domaines de ces attributs sont divisés en: 7, 5, 7, 6, 5, 6, 3 et 8 sous-domaines, respectivement.

Pour les paramètres de l'algorithme génétique, nous avons fixé le taux de croisement à (70%) et le taux de mutation à (30%) pour améliorer 40 chromosomes en 20 générations.

5.3.2 Résultats obtenus

Qautre expériences nous ont permis d'examiner l'efficacité et l'efficience de notre approche.

Première expérience: l'analyse de performances

Dans le premier test, nous avons comparé $\mathcal{F}\&\mathcal{A}\&\mathcal{R}$ avec trois autres approches de conception d'un EDP: le partitionnement, l'allocation et la réplication sont traités de manière isolée, la réplication de données est effectuée une fois que la fragmentation et d'allocation sont effectuées conjointement, l'allocation et la réplication sont traitées de manière conjointe et séparément du partitionnement. La figure 5.20 compare les performances relatives des quatre méthodes en fixant le seuil de fragmentation à 100 et le skew de valeur d'attribut à $0,5$.

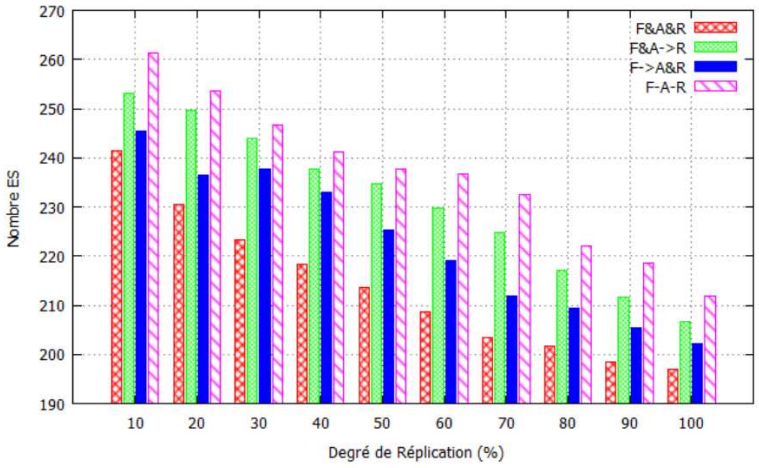


Figure 5.20 – *Comparaison entre les approches de conception d'un EDP*

Pour chaque approche de conception, nous faisons varier le degré de réplication de 1 à 10 et pour chaque valeur, nous avons calculé le nombre d'E/S nécessaires pour exécuter la charge des requêtes sur une grappe de 10 nœuds. Nous remarquons que l'approche $\mathcal{F}\&\mathcal{A}\&\mathcal{R}$ est plus adaptée à la conception d'un EDP que l'approche itérative et ses variantes. A partir de ces résultats, nous constatons également que l'augmentation du degré de réplication implique une augmentation de la performance du système en minimisant le coût d'exécution des requêtes.

Dans un second test, nous avons étudié l'impact de la réplication sur la scalabilité de notre approche. Pour cela, nous calculons le facteur de rapidité (speed up). Pour un seuil de fragmentation de 100, nous avons fait varier le nombre de nœuds de 1 à 32 et pour chaque valeur, nous

calculons le speed up pour les degrés de réplication suivants: $\mathcal{R}$: 8(25%), 16(50%), 24(75%) et 32(100%). Les résultats obtenus sont présentés dans la figure 5.21 et confirment que l'approche proposée est bien adaptée à la conception d'un EDP. De plus, l'augmentation du degré de réplication permet d'offrir une meilleure accélération. Dans le cas $\mathcal{R} = 100\%$, l'accélération est approximativement linéaire. Cela est dû au fait que la réplication donne des avantages supplémentaires découlant de l'équilibrage de charge qui n'élimine pas complètement les effets du skew. Cependant, la réplication requiert plus de mémoire pour le stockage et pour la maintenance des répliques (que nous négligeons dans cet article). Le degré de réplication doit donc être bien paramétré.

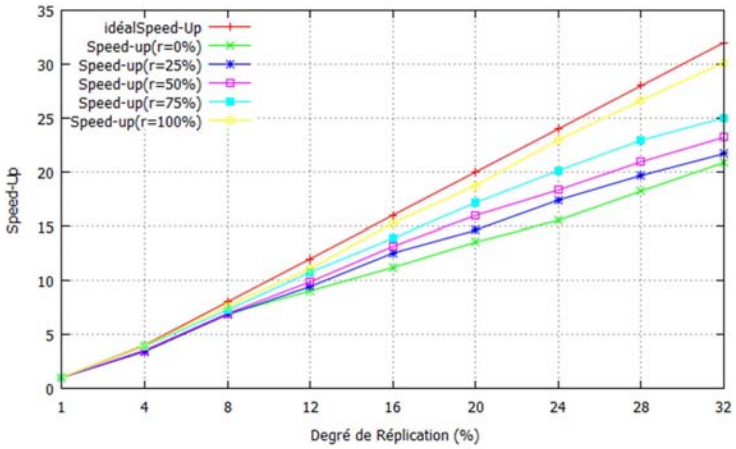


Figure 5.21 – *Effet du degré de réplication sur le speed-up de* $\mathcal{F}\&\mathcal{A}\&\mathcal{R}$

Deuxième expérience: la dépendance entre les paramètres de conception d'un EDPs

Dans le premier test, nous avons étudié la dépendance entre le degré de skew des valeurs d'attribut et le degré de skew de placement des fragments. Nous avons fixé le seuil de fragmentation à 100 et le nombre de nœuds à 10. Nous faisons varier le degré de skew des valeurs d'attribut de $0,2$ à 1 et pour chaque valeur, nous avons calculé le degré de skew de placement des fragments. La figure 5.22 montre les résultats obtenus et confirme que la mauvaise répartition de données augmente considérablement quand le degré de skew des valeurs d'attribut augmente.

Dans la deuxième évaluation, nous avons étudié l'impact du degré de réplication sur le traitement en parallèle. Nous faisons varier le degré de réplication de 1 à 10 et, pour chaque valeur, nous avons calculé le degré de d'équilibrage de charge. Comme le montre la figure 5.23, l'augmentation du degré de réplication réduit les effets négatifs du skew de données. D'autre part, l'augmentation du degré de réplication facilite l'atteinte de la haute performance de l'EDP.

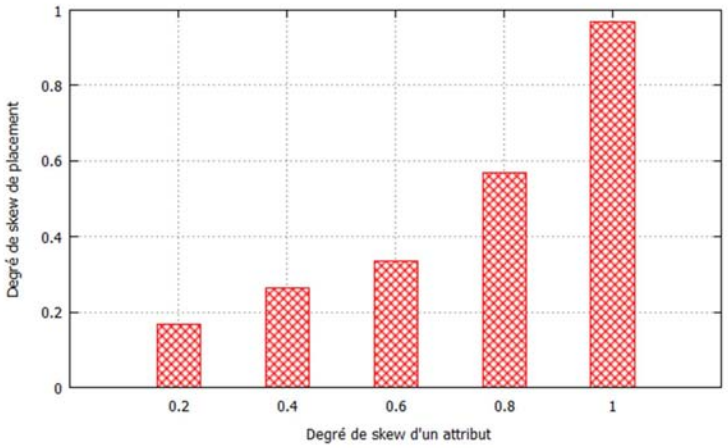


Figure 5.22 – *Skew des valeurs d'un attribut vs Skew de Partitioning de Données*

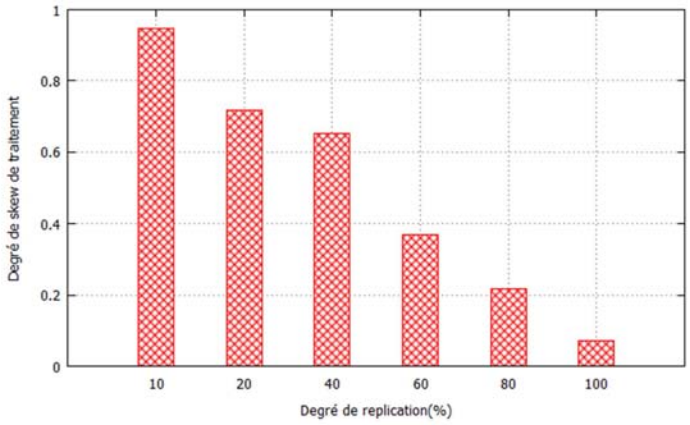


Figure 5.23 – *Dépendance entre degré de réplication et skew de traitement.*

Troisième expérience: l'effet de skew de données

Nous avons fixé le degré de skew de placement des fragments à $0,5$ et le seuil de fragmentation à 100. Nous faisons varier le degré de skew de placement des fragments de 0.2 à 1 et pour chaque valeur, nous calculons le coût d'exécution des requêtes sur une grappe de 10 nœuds pour un degré de réplication $\mathcal{R}$ égal à 2, 5, 8 et 10. La figure 5.24 montre que l'augmentation du degré de skew de placement des fragments dégrade la performance du système. Cela est du au fait que le degré d'équilibrage de charge augmente quand le degré de skew de placement des fragments augmente.

Dans un second test, nous nous sommes intéressés à l'effet du degré de skew des valeurs d'attribut sur le partitionnement. nous faisons varier le seuil de fragmentation $\mathcal{W}$ de 100 à 350 et le degré de skew des valeurs d'attribut de 0.2 à 1. Pour chaque valeur nous calculons le temps d'exécution de la charge de requêtes sur une grappe de 10 nœuds. Les résultats obtenus montrent que le degré de skew des valeurs d'attribut impacte la performance du système par la

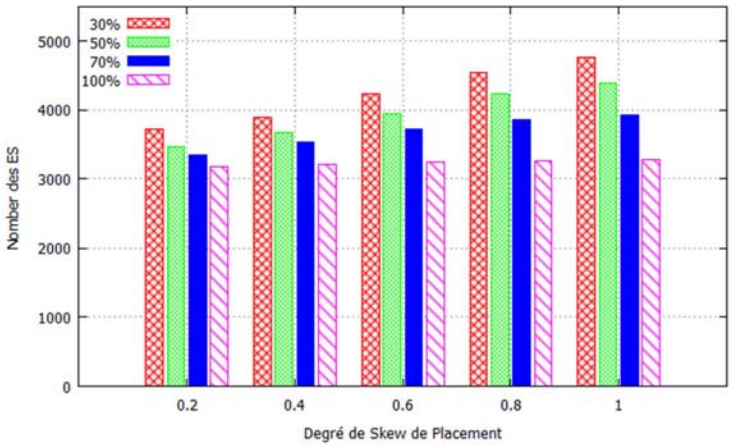


Figure 5.24 – *Effet du degré de skew des valeurs d'un attribut sur le seuil de fragmentation*

minimisation du nombre des fragments générés. Cela est dû au fait que notre approche élimine les attributs ayant un degré de skew des valeurs d'attribut élevé de la liste des attributs de fragmentation candidats. Ainsi le nombre de fragments se réduit et leur taille s'accroit. En conséquence, le degré de l'équilibrage de charge s'accroit (voir figure 5.25).

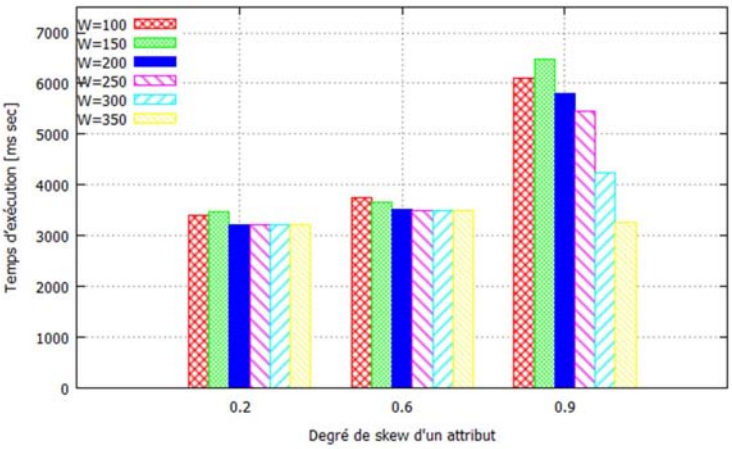


Figure 5.25 – *Effet du degré de skew des valeurs d'un attribut sur le temps d'exécution*

Ainsi, nous pouvons conclure que le degré de skew des valeurs d'attribut et de placement des fragments impacte le traitement parallèle. La performance du système augmente quand le degré de réplication augmente.

Quatrième expérience: l'effet d'hétérogénéité

Dans les expériences précédentes, nous avons supposé que la grappe est homogène (tous les nœuds ont la même capacité de calcul). Dans celle-ci, nous étudions l'effet de l'hétérogénéité. La puissance de calcul de chaque nœud a été attribuée en utilisant une fonction aléatoire pour tenir compte de l'hétérogénéité de la puissance de traitement de chaque nœud. Tous d'abord, nous avons normalisé les puissances de calcul des nœuds puis nous avons adapté notre approche comme suit:

1. *pour l'algorithme d'allocation*, nous utilisons notre algorithme $\mathcal{F}\&\mathcal{A}$_`ALLOC` pour assigner les classes des fragments aux nœuds. Il est à noter que l'algorithme $\mathcal{F}\&\mathcal{A}$-`ALLOC` prend en considération l'hétérogénéité de la puissance de calcul et du stockage des nœuds.
2. *pour la stratégie d'allocation des requêtes*, nous assignons chaque sous-requête au nœud le plus puissant qui peut la traiter

Nous avons gardé les mêmes paramètres que l'expérience précédente et nous avons étudié les performances de notre approche adaptée en mesurant le coût d'exécution moyen de la charge sur une grappe homogène et sur une grappe hétérogène. La puissance de calcul de chaque nœud de la grappe homogène est égale à la moyenne des puissances de calcul de la grappe hétérogène. Les résultats représentés dans la figure 5.26 montrent l'intérêt de la prise en considération des paramètres d'hétérogénéité de la grappe dans la stratégie de placement des données et des requêtes.

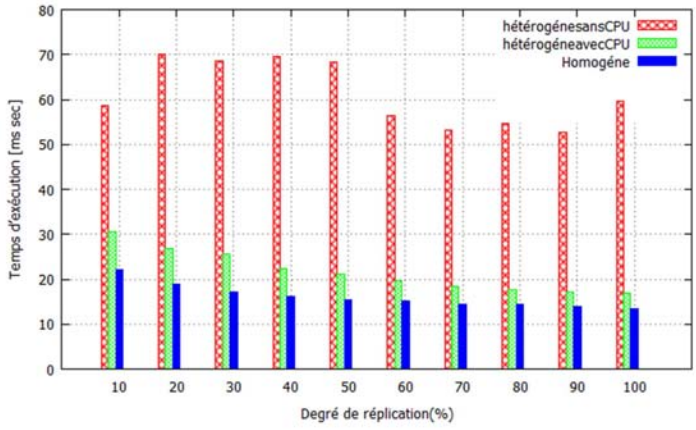


Figure 5.26 – *Effet d'hétérogénéité sur la performance.*

5.4 Bilan et discussion

Les résultats expérimentaux sont encourageants et montrent la faisabilité de nos approches. Aussi pouvons-nous conclure que nos approches $\mathcal{F}\&\mathcal{A}$ et $\mathcal{F}\&\mathcal{A}\&\mathcal{R}$ sont efficaces et efficientes pour le déploiement d'un EDP sur les clusters de bases de données homogénes ou hétérogènes. Ces résultats démontrent clairement les avantages de notre proposition.

Néanmoins, $\mathcal{F}\&\mathcal{A}$ souffre principalement d'un problème d'équilibrage de charges, ce qui est probablement dû à une mauvaise répartition des données et à une mauvaise répartition des charges entre les nœuds de traitement de la grappe de machines (le deuxième facteur étant une conséquence du premier). Cette mauvaise distribution s'explique par une distribution biaisée des valeurs des attributs de fragmentation ainsi que par une distribution biaisée de données lors de la phase de placement.

En revanche, les expérimentations de l'approche $\mathcal{F}\&\mathcal{A}\&\mathcal{R}$ donnent de bons résultats. Cela est dû au fait qu'elle : (1) prend en considération la distribution biaisée durant les phases d'allocation et de fragmentation et (2) prend en considération l'intégration de la réplication dans les phases de notre conception. Nous relevons aussi, lorsque les données sont répliquées, le transfert de données entre les nœuds diminue.

Bien que les approches proposées soient tout particulièrement destinées au déploiement d'un EDP sur les grappes de bases de données, il est intéressant d'examiner comment elles pourraient être appliquées à d'autres plateformes de données distribuées, comme les réseaux Peer-To-Peer (P2P) [104] et les infrastructures grille [62].

Pour ce qui concerne les réseaux P2P, malheureusement nos méthodes ne peuvent pas être appliquées, en raison du fait que, dans ces réseaux, les nœuds peuvent rejoindre et quitter le réseau librement, et que la même topologie de réseau peut changer rapidement. Ces caractéristiques font que les approches $\mathcal{F}\&\mathcal{A}$ et $\mathcal{F}\&\mathcal{A}\&\mathcal{R}$ sont inappropriées pour le déploiement d'un EDP au-dessus des réseaux P2P.

Contrairement aux réseaux P2P, dans un environnement de grappe de machines, les nœuds peuvent être ajoutés et supprimés de manière contrôlée, ce dernier étant une propriété essentielle de nos approches. Et pour ce qui concerne la mise en place des infrastructures de grille, nos approches peuvent être facilement adaptées pour couvrir ces environnements distribués spécialisés. Ainsi, dans une vision plus large, les grilles peuvent être conçues comme des grappes à haute performance, fortement hétérogènes. En conséquence, $\mathcal{F}\&\mathcal{A}$ et $\mathcal{F}\&\mathcal{A}\&\mathcal{R}$ auraient seulement besoin d'une fragmentation spécialisée et de modèles d'allocation capables de faire face aux particularités des réseaux, l'approche générale reste identique.

Conclusion

Tout au long du présent chapitre, nous avons présenté et discuté les mesures de performances que nous avons mené pour valider l'approche proposée. Les résultats expérimentaux sont encourageants et montrent la faisabilité de notre approche. Néanmoins, dans le but d'avoir de meilleures performances, des améliorations peuvent être apportées, et c'est ce que le chapitre suivant évoque.

Troisième partie

Conclusion et perspectives

Conclusion et perspectives

"To accomplish great things, we must not only act, but also dream; not only plan, but also believe".
Anatole France (1844-1924)

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.

Conclusion

Vu la diversité des plateformes de déploiement des bases/entrepôts de données le processus de conception de ces dernières devient une tâche cruciale afin d'assurer la performance de traitement. Malheureusement, les méthodologies de conception n'ont pas suivi cette évolution des plateformes. La plupart des méthodologies sont séquentielles et ignorent l'interaction entre les différentes phases. Dans la littérature, les phases de conception ont eu un énorme intérêt auprès des communautés de bases de données et de calcul à haute performance (*high performance computing (HPC)*). Mais l'absence d'une approche globale de déploiement rend les solutions existantes limitées. Notre collaboration avec l'entreprise *Teradata* nous a poussé à repenser la démarche de conception de bases/entrepôts de données parallèles en proposant un cycle de vie de déploiement qui peut être appliqué à toute plateforme de déploiement de type HPC. Ce cycle de vie comprend les phases suivantes : le choix de l'architecture matérielle, la fragmentation de données, l'allocation de données, la réplication de données et l'équilibrage de charge. Chacune de ces phases est un problème qui a été formalisé NP-Complet.

Dans cette thèse un ensemble de contributions a été proposé: (i) la proposition de cycle de vie de déploiement, (ii) des états d'art sur les phases de conception, (iii) la définition d'un modèle de coût pour quantifier les solutions de déploiement proposées sur des plateformes parallèles, (iv) la proposition d'une approche de composition pas à pas et finalement, (v) une validation sur une machine industrielle Teradata.

Etat de l'art

Notre travail a été guidé par une étude préalable des travaux existants traitant des problèmes de conception des entrepôts de données (base de données) parallèle. Dans un premier temps, nous avons étudié le cycle de vie de conception d'un entrepôt de données. Puis, nous avons décrit le cycle de vie de déploiement d'une base de données parallèle en exposant les principaux travaux existants dans chaque phase. Nous avons constaté que la plupart des travaux existants traitent les problèmes de fragmentation, d'allocation, de réplication et d'équilibrage de charges d'une manière isolée et n'exploitent par l'interdépendance entre ces deux problèmes. De plus, au niveau de chaque phase, une métrique d'évaluation est utilisée pour évaluer la pertinence du résultat obtenu.

Modèle de coût

Dans la littérature souvent peu d'attention est donnée aux modèles de coût. Dans cette thèse, nous avons donné plus d'importance à la définition de modèle de coût, car ils présentent le coeur de notre démarche. S'ils ne sont pas bien définis, la qualité de solution de déploiement est alors faussée. Nous avons d'abord proposé une démarche modulaire de collecte des paramètres de notre modèle de coût. Ses paramètres sont partitionnés en groupes, où chacun correspond à une phase de cycle de vie de déploiement. Cette vision donne plus de visibilité et de flexibilité à notre modèle de coût. Dans ce modèle, nous avons introduit une nouvelle stratégie d'ordonnancement des requêtes qui assure une exécution équilibrée. Pour cela, nous avons assimilé le problème de placement des sous-requêtes à un problème *dual Bin Packing* et nous avons proposé un algorithme glouton pour le résoudre. Finalement, notre modèle de coût quantifie chaque solution de déploiement.

Une démarche de composition pas à pas

Vu l'interaction entre les différentes phases de conception d'un entrepôt de données parallèle, nous avons proposé une démarche de composition pas à pas de l'ensemble des phases. Notre démarche est motivée par le fait que les problèmes liés aux phases sont difficiles. Nous avons d'abord commencé par combiner la fragmentation et l'allocation. L'idée principale est de partitionner l'entrepôt de données *sachant que ses fragments* soient alloués sur une machine parallèle ayant certaines caractéristiques [17].

Nous avons formalisé le problème comme étant un problème d'optimisation à contraintes. Deux méta-heuristiques (algorithme génétique et algorithme Hill climbing) ont été adaptées pour résoudre le problème de fragmentation et le modèle de coût proposé est utilisé comme la métrique d'évaluation. Pour le problème d'allocation, nous avons proposé une variante de l'approche du placement circulaire, où au lieu d'allouer un fragment par nœud, nous avons un ensemble de fragments.

Nous avons recensé le besoin d'étendre notre approche $\mathcal{F}\&\mathcal{A}$ par la prise en compte de la mauvaise distribution de données durant les phases de fragmentation et l'allocation. De plus, pour augmenter la disponibilité et assurer une exécution parallèle sans faire recours à

la migration de données, nous avons considéré la réplication de données. La réplication de données est fortement dépendante de la fragmentation et de l'allocation. Aussi, nous avons considéré la fragmentation, l'allocation et la réplication comme étant un seul processus unifié. Notre modèle de coût évalue la pertinence du schéma de placement obtenu. Nous avons adapté notre algorithme génétique par l'addition d'une fonction de pénalité pour éliminer les attributs ayant un degré de *skew* trop élevé pour être choisis comme des attributs de fragmentation. Pour l'allocation de données, nous avons formalisé le problème comme étant un problème de classification floue pour permettre la sélection d'un schéma d'allocation redondant en se basant sur la probabilité d'appartenance.

Évaluation Théorique et Réelle sur Teradata

Pour valider nos contributions, nous avons d'abord développé un simulateur. Le simulateur élaboré prend en charge un ensemble de cas d'utilisation nécessaire pour l'élaboration du schéma de déploiement. Les deux approches de conception $\mathcal{F}\&\mathcal{A}$ et $\mathcal{F}\&\mathcal{A}\&\mathcal{R}$ sont testées respectivement sous les bancs d'essai APB-1 release II et SSB . Notons que notre simulateur utilise notre modèle de coût mathématique pour assurer cette évaluation.

Une deuxième évaluation a été menée sur la machine *Teradata*. Nous avons préparé les plans de tests aux ingénieurs de cette entreprise et ces derniers ont procédé à l'évaluation de l'ensemble de requêtes sur le schéma de déploiement défini par nos algorithmes. Les résultats ont montré l'intérêt de la composition des phases de conception sur la démarche itérative.

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.

Prise en considération de l'interaction entre les requêtes

Dans nos travaux, nous avons considéré que la charge de requêtes s'exécute d'une manière séquentielle selon un ordre supposé existe. Dans le contexte des entrepôts de données, les requêtes partagent un nombre important d'opérations. Cela est du au fait que toute jointure passe par la table des faits. Cela est connu sous le nom de l'optimisation multi-requêtes qui a été identifié par Timos Sellis [127]. La prise en compte de cette interaction influe considérablement sur la définition de modèle de coût et l'ensemble des phase de déploiement.

Présence des structures d'optimisation

Une autre dimension non prise en compte par nos travaux est liée à l'absence des structures d'optimisation comme les vues matérialisées, les index sur chaque schéma de fragmentation. La

sélection des ces structures d'optimisation peut se faire après le processus de déploiement tout en adaptant notre modèle de coût en intégrant les paramètres liés aux structures d'optimisation.

Conception incrémentale

Un entrepôt de données est fait pour évoluer, cette évolution concerne les données, les requêtes, les supports de stockage, ... etc. Cet aspect dynamique impose l'adaptation des solutions proposées. Dans le laboratoire LIAS, des travaux sur la fragmentation incrémentale sont menés dans le cadre d'un entrepôt de données centralisé. Généraliser ces travaux au contexte parallèle serait une piste à explorer.

Considération de d'autres plateformes de déploiement et Emerging Hardware

Dans l'état de l'art nous avons insisté sur la diversité des plateformes de déploiement assurant le HPC. Dans notre validation, nous avons traité un seul type de déploiement. Il serait intéressant de considérer d'autres plateformes afin de tirer des leçons sur notre démarche de déploiement et la qualité de notre modèle de coût. Une deuxième piste liée à cette perspective concerne la définition d'un autre problème, qui consiste à choisir la meilleure plateforme de déploiement pour un schéma d'entrepôt de données et une charge de requête.

Rappelons que nous vivons à une émergence des hardwares (GPU, Accelerated processing unit, Field-Programmable Gate Array (FPGA)), mais notre modèle de coût suppose un hardware classique (les données stockées sur un disque dur). La prise en compte de ces hardwares exige une autre définition de modèle de coût.

Développement d'un outil de simulation de la phase de déploiement

L'identification d'un cycle de vie de déploiement nous motive de proposer un outil de déploiement qui pourrait jouer le rôle d'un simulateur de déploiement pour aider les concepteurs à choisir leur solution.

Preuve sur la composition

Les différentes phases de déploiement ayant été intégrées constitue une forme de composition de processus. Cette opération de composition est actuellement non explicitée ni formalisée. La formalisation des opérateurs de composition introduit dans ce travail permettrait d'une part de préciser les conditions d'utilisation de cette composition et d'autre part d'étudier ses propriétés.

Quatrième partie

Annexes

Annexe 1: Liste des requêtes SSB

Q01.1

```
select sum(lo_extendedprice*lo_discount) as revenue
from lineorder l, DATE d
where l.lo_orderdate=d.d_datekey
and d.d_year='1993' and l.lo_discount in (1,2,3) and l.lo_quantity=25;
```

Q01.2

```
select sum(lo_extendedprice*lo_discount) as revenue
from lineorder l, DATE d
where l.lo_orderdate=d.d_datekey
and d.d_year='1994' and l.lo_discount in (1,2,3) and l.lo_quantity=25;
```

Q01.3

```
select sum(lo_extendedprice*lo_discount) as revenue
from lineorder l, DATE d
where l.lo_orderdate=d.d_datekey
and d.d_year='1995' and l.lo_discount in (1,2,3) and l.lo_quantity=25;
```

Q04.1

```
select sum(lo_revenue), d_year, p_brand
from lineorder l, DATE d, part p, supplier s
where l.lo_orderdate=d.d_datekey
and l.lo_partkey=p.p_partkey and l.lo_suppkey=s.s_suppkey
and p.p_category='MFGR\#12'and s.s_region='AMERICA'
group by d.d_year, p.p_brand order by d.d_year, p.p_brand;
```

Q04.2

```
select sum(lo_revenue), d_year, p_brand
from lineorder l, DATE d, part p, supplier s
where l.lo_orderdate=d.d_datekey
and l.lo_partkey=p.p_partkey and l.lo_suppkey=s.s_suppkey
and p.p_category='MFGR\#12' and s.s_region='AFRICA'
group by d.d_year, p.p_brand order by d.d_year, p.p_brand;
```

Q04.3

```
select sum(lo_revenue), d_year, p_brand
from lineorder l, DATE d, part p, supplier s
where l.lo_orderdate=d.d_datekey and l.lo_partkey=p.p_partkey
and l.lo_suppkey=s.s_suppkey and p.p_category='MFGR\#12'
and (s.s_region='ASIA' or s.s_region='MIDDLE EAST' or s.s_region='EUROPE')
group by d.d_year, p.p_brand order by d.d_year, p.p_brand;
```

Q05.0

```
select c_city, s_city, d_year, sum(lo_revenue) as revenue
from customer c, lineorder l, supplier s, DATE d
where l.lo_custkey=c.c_custkey
and l.lo_suppkey=s.s_suppkey and l.lo_orderdate= d.d_datekey
and c.c_nation='UNITED STATES' and s.s_nation='UNITED STATES'
and d.d_year in(1992,1993,1994,1995,1996,1997)
group by c.c_city,s.s_city, d.d_year order by d.d_year asc, revenue desc;
```

Q06.0

```
select sum(lo_revenue), d_year, p_brandc f
rom lineorder l, DATE d, part p, supplier s
where l.lo_orderdate=d.d_datekey
and l.lo_partkey= p.p_partkey and l.lo_suppkey= s.s_suppkey
and p.p_brand='MFGR\#2239' and s.s_region='EUROPE'
group by d.d_year, p.p_brand order by d.d_year, p.p_brand;
```

Q07.0

```
select c_nation, s_nation, d_year, sum(lo_revenue) as revenue
from customer c, lineorder l, supplier s, DATE d
where l.lo_custkey=c.c_custkey
and l.lo_suppkey=s.s_suppkey
and l.lo_orderdate=d.d_datekey
and c.c_region='ASIA' and s.s_region='ASIA'
and d.d_year in(1992,1993,1994,1995,1996,1997)
group by c.c_nation, s.s_nation, d.d_year order by d.d_year asc, revenue desc;
```

Q08.0

```
select sum(lo_revenue), d_year, p_brand
from lineorder l, .DATE d, part p, supplier s
where l.lo_orderdate=d.d_datekey
and l.lo_partkey=p.p_partkey
and l.lo_suppkey=s.s_suppkey
and p.p_brand in ('MFGR\#2221','MFGR\#2222','MFGR\#2223','MFGR\#2224','MFGR\#2225','
and s.s_region='ASIA'
group by d.d_year, p.p_brand order by d.d_year, p.p_brand;
```

Q09.0

```
select c_city, s_city, d_year, sum(lo_revenue) as revenue
from customer c, lineorder l, supplier s, DATE d
where l.lo_custkey=c.c_custkey and l.lo_suppkey=s.s_suppkey
and l.lo_orderdate=d.d_datekey
and (c.c_city='UNITED KI1' or c.c_city='UNITED KI5')
and (s.s_city='UNITED KI1' or s.s_city='UNITED KI5')
and d.d_year in(1992,1993,1994,1995,1996,1997)
group by c.c_city, s.s_city, d.d_year order by d.d_year asc, revenue desc;
```

Q10.0

```
select c_city, s_city, d_year, sum(lo_revenue) as revenue
from customer c, lineorder l, supplier s, DATE d
where l.lo_custkey=c.c_custkey and l.lo_suppkey=s.s_suppkey
and l.lo_orderdate= d.d_datekey
and (c.c_city='UNITED KI1' or c.c_city='UNITED KI5')
and (s.s_city='UNITED KI1' or s.s_city='UNITED KI5')
and d.d_yearmonth = 'Dec1997'
group by c.c_city, s.s_city, d.d_year order by d.d_year asc, revenue desc;
```

Q11.0

```
select d_year, c_nation, sum(lo_revenue - lo_supplycost) as profit
from DATE d, customer c, supplier s, part p, lineorder l
where l.lo_custkey=c.c_custkey and l.lo_suppkey=s.s_suppkey
and l.lo_partkey=p.p_partkey and l.lo_orderdate=d.d_datekey
and c.c_region='AMERICA' and s.s_region='AMERICA'
and (p.p_mfgr='MFGR\#1' or p.p_mfgr='MFGR\#2')
group by d.d_year, c.c_nation order by d.d_year, c.c_nation;
```

Q12.0

```
select d_year, s_nation, p_category,sum(lo_revenue - lo_supplycost) as profit
from DATE d, customer c, supplier s, part p, lineorder l
where l.lo_custkey=c.c_custkey
and l.lo_suppkey=s.s_suppkey and l.lo_partkey=p.p_partkey
and l.lo_orderdate=d.d_datekey
and c.c_region='AMERICA' and s.s_region='AMERICA'
and (d.d_year=1997 or d.d_year=1998)and(p.p_mfgr='MFGR\#1' or p.p_mfgr= 'MFGR\#2')
group by d.d_year,s.s_nation,p.p_category order by d.d_year,s.s_nation, p.p_category;
```

Q13.0

```
select d_year, s_city, p_brand, sum(lo_revenue - lo_supplycost) as profit
from DATE d, customer c, supplier s, part p, lineorder l
where l.lo_custkey=c.c_custkey and l.lo_suppkey=s.s_suppkey
and l.lo_partkey=p.p_partkey and l.lo_orderdate=d.d_datekey
and s.s_nation='UNITED STATES' and (d.d_year=1997 or d.d_year=1998)
and p.p_category='MFGR\#14'
group by d.d_year, s.s_city, p.p_brand order by d.d_year, s.s_city, p.p_brand;
```

Q14.0

```
select d_year, s_city, p_brand1, sum(lo_revenue - lo_supplycost) as profit
from DATE d, customer c, supplier s, part p, lineorder l
where l.lo_custkey=c.c_custkey and l.lo_suppkey=s.s_suppkey
and l.lo_partkey=p.p_partkey and l.lo_orderdate=d.d_datekey
and s.s_nation='EGYPT' and (d.d_year=1997 or d.d_year=1998)
and p.p_category='MFGR\#14'
group by d.d_year, s.s_city, p.p_brand order by d.d_year, s.s_city, p.p_brand;
```

Q15.0

```
select d_year, s_city, p_brand, sum(lo_revenue - lo_supplycost) as profit
from DATE d, customer c, supplier s, part p, lineorder l
where l.lo_custkey=c.c_custkey and l.lo_suppkey=s.s_suppkey
and l.lo_partkey=p.p_partkey and l.lo_orderdate=d.d_datekey
and s.s_nation='ALGERIA' and (d.d_year=1997 or d.d_year=1998)
and p.p_category='MFGR\#14'
group by d.d_year, s.s_city, p.p_brand order by d.d_year, s.s_city, p.p_brand;
```

Q16.0

```
select d_year, s_city, p_brand, sum(lo_revenue - lo_supplycost) as profit
from DATE d, customer c, supplier s, part p, lineorder l
where l.lo_custkey=c.c_custkey and l.lo_suppkey=s.s_suppkey
and l.lo_partkey=p.p_partkey and l.lo_orderdate=d.d_datekey
and s.s_nation='ALGERIA' and (d.d_year=1996 or d.d_year=1997)
and p.p_category='MFGR\#14'
group by d.d_year, s.s_city, p.p_brand order by d.d_year, s.s_city, p.p_brand;
```

Q17.0

```
select d_year, s_city, p_brand, sum(lo_revenue - lo_supplycost) as profit
from DATE d, customer c, supplier s, part p, lineorder l
where l.lo_custkey=c.c_custkey and l.lo_suppkey=s.s_suppkey
and l.lo_partkey=p.p_partkey and l.lo_orderdate=d.d_datekey
and s.s_nation='CANADA' and (d.d_year=1997 or d.d_year=1998)
 and p.p_category='MFGR\#14'
group by d.d_year, s.s_city, p.p_brand order by d.d_year, s.s_city, p.p_brand;
```

Q18.0

```
select c_nation, s_nation, d_year, sum(lo_revenue) as revenue
from customer c, lineorder l, supplier s, DATE d
where l.lo_custkey=c.c_custkey and l.lo_suppkey=s.s_suppkey
 and l.lo_orderdate=d.d_datekey and c.c_region='AMERICA'
 and s.s_region='AMERICA' and d.d_year in(1992,1993,1994,1995,1996,1997)
group by c.c_nation, s.s_nation, d_year order by d.d_year asc, revenue;
```

Q19.0

```
select c_nation, s_nation, d_year, sum(lo_revenue) as revenue
from customer c, lineorder l, supplier s, DATE d
where l.lo_custkey=c.c_custkey and l.lo_suppkey=s.s_suppkey
and l.lo_orderdate=d.d_datekey and c.c_region='MIDDLE EAST'
and s.s_region='MIDDLE EAST' and d.d_year in(1992,1993,1994,1995,1996,1997)
group by c.c_nation, s.s_nation, d_year order by d.d_year asc, revenue desc;
```

Q20.0

```
select c_nation, s_nation, d_year, sum(lo_revenue) as revenue
from customer c, lineorder l, supplier s, DATE d
where l.lo_custkey=c.c_custkey and l.lo_suppkey=s.s_suppkey
and l.lo_orderdate=d.d_datekey and c.c_region='EUROPE'
and s.s_region='EUROPE' and d.d_year in(1992,1993,1994,1995,1996,1997)
group by c.c_nation, s.s_nation, d_year order by d.d_year asc, revenue desc;
```

Bibliographie

[1] I. Ahmad, K. Karlapalem, and R. A. Ghafoor. Evolutionary algorithms for allocating data in distributed database systems. In *in Distributed Database Systems, Distributed and Parallel Databases*, pages 5–32, 2002.

[2] F. Akal, K. Böhm, and H.-J. Schek. Olap query evaluation in a database cluster: A performance study on intra-query parallelism. In *Proceedings of the 6th East European Conference on Advances in Databases and Information Systems*, ADBIS '02, pages 218–231, London, UK, UK, 2002. Springer-Verlag.

[3] G. Alonso. Partial database replication and group communication primitives (extended abstract). In *in Proceedings of the 2 nd European Research Seminar on Advances in Distributed Systems*, pages 171–176, 1997.

[4] A. C. F. Alvim, C. C. Ribeiro, F. Glover, and D. J. Aloise. A hybrid improvement heuristic for the one-dimensional bin packing problem. *JOURNAL OF HEURISTICS*, 10:2004, 2004.

[5] P. M. G. Apers. Data allocation in distributed database systems. *ACM Transactions on database systems*, 13(3):263–304, 1988.

[6] L. Bellatreche. Utilisation des vues matérialisées, des index et de la fragmentation dans la conception logique et physique des entrepôts de données". Phd. thesis, Université de Clermont-Ferrand II, December 2000.

[7] L. Bellatreche and S. Benkrid. A joint design approach of partitioning and allocation in parallel data warehouses. In *11th International Conference on Data Warehousing and Knowledge Discovery (DAWAK)*, pages 99–110, 2009.

[8] L. Bellatreche, S. Benkrid, A. Crolotte, A. Cuzzocrea, and A. Ghazal. The f&a methodology and its experimental validation on a real-life parallel processing database system. In *CISIS*, pages 114–121, 2012.

[9] L. Bellatreche, S. Benkrid, A. Ghazal, A. Crolotte, and A. Cuzzocrea. Verification of partitioning and allocation techniques on teradata dbms. In *ICA3PP (1)*, pages 158–169, 2011.

[10] L. Bellatreche and K. Boukhalfa. An evolutionary approach to schema partitioning selection in a data warehouse environment. In *7th International Conference on Data Warehousing and Knowledge Discovery (DAWAK)*, pages 115–125, 2005.

[11] L. Bellatreche and K. Boukhalfa. La fragmentation dans les entrepï¿½ts de donnï¿½es : une approche basï¿½e sur les algorithmes gï¿½nï¿½tiques. In *Entrepï¿½ts de Donnï¿½es et Analyse en ligne (EDA'05): Revue des Nouvelles Technologies de l'Information*, pages 141–160, 2005.

[12] L. Bellatreche, K. Boukhalfa, and H. I. Abdalla. Saga: A combination of genetic and simulated annealing algorithms for physical data warehouse design. *in 23rd British National Conference on Databases (BNCOD'06)*, pages 212–219, July 2006.

[13] L. Bellatreche, K. Boukhalfa, and P. Richard. Data partitioning in data warehouses: Hardness study, heuristics and oracle validation. In *International Conference on Data Warehousing and Knowledge Discovery (DaWaK'2008)*, pages 87–96, 2008.

[14] L. Bellatreche, K. Boukhalfa, and P. Richard. Referential horizontal partitioning selection problem in data warehouses: Hardness study and selection algorithms. *International Journal of Data Warehousing and Mining*, 5(4):1–23, 2009.

[15] L. Bellatreche, A. Cuzzocrea, and S. Benkrid. *F&a*: A methodology for effectively and efficiently designing parallel relational data warehouses on heterogenous database clusters. In *DaWak*, pages 89–104, 2010.

[16] L. Bellatreche, A. Cuzzocrea, and S. Benkrid. Query optimization over parallel relational data warehouses in distributed environments by simultaneous fragmentation and allocation. In *Proceedings of the 10th international conference on Algorithms and Architectures for Parallel Processing - Volume Part I*, ICA3PP10, pages 124–135, Berlin, Heidelberg, 2010. Springer-Verlag.

[17] L. Bellatreche, A. Cuzzocrea, and S. Benkrid. Effectively and efficiently designing and querying parallel relational data warehouses on heterogeneous database clusters: The f&a approach. *J. Database Manag.*, 23(4):17–51, 2012.

[18] L. Bellatreche, K. Karlapalem, and Q. Li. Complex methods and class allocation in distributed object-oriented databases. *in the 5th International Conference on Object Oriented Information Systems (OOIS'98)*, pages 239–256, September 1998.

[19] L. Bellatreche, S. Khouri, and N. Berkani. Semantic data warehouse design: From etl to deployment à la carte. In *DASFAA (2)*, pages 64–83, 2013.

[20] L. Bellatreche, A. Simonet, and M. Simonet. An algorithm for vertical fragmentation in distributed object database systems with complex attributes and methods. *in International Workshop on Database and Expert Systems Applications (DEXA'96), Zurich*, pages 15–21, September 1996.

[21] S. Benkrid and L. Bellatreche. Une démarche conjointe de fragmentation et de placement dans le cadre des entrepôts de données parallèles. In *EDA*, pages 91–106, 2009.

[22] S. Benkrid and L. Bellatreche. Une démarche conjointe de fragmentation et de placement dans le cadre des entrepôts de données parallèles. *Technique et Science Informatiques*, 30(8):953–973, 2011.

[23] S. Benkrid, L. Bellatreche, and A. Cuzzocrea. Designing parallel relational data warehouses: A global, comprehensive approach. In *ADBIS (2)*, pages 141–150, 2013.

[24] S. Benkrid, L. Bellatreche, and H. Drias. A combined selection of fragmentation and allocation schemes in parallel data warehouses. In *DEXA Workshops*, pages 370–374, 2008.

[25] R. Bezdek, J. C. and W. Full. Fcm: The fuzzy c-means clustering algorithm. *Computers and Geo-sciences)*, 10(2-3):191–203, 1984.

[26] J. A. Blakeley, P. A. Dyke, C. A. Galindo-Legaria, N. James, C. Kleinerman, M. Peebles, R. Tkachuk, and V. Washington. Microsoft sql server parallel data warehouse: Architecture overview. In *BIRTE*, pages 53–64, 2011.

[27] M. Böhnlein and A. U. vom Ende. Business process oriented development of data warehouse structures. In *DATA WAREHOUSING*, pages 3–21, 2000.

[28] H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith, and P. Valduriez. Prototyping bubba, a highly parallel database system. *IEEE Trans. on Knowl. and Data Eng.*, 2(1):4–24, Mar. 1990.

[29] A. J. Borr. Transaction monitoring in encompass: reliable distributed transaction processing. In *Proceedings of the seventh international conference on Very Large Data Bases - Volume 7*, VLDB '81, pages 155–165. VLDB Endowment, 1981.

[30] L. Bouganim. Equilibrage de charges lors de l'execution de requêtes sur des architectures multiprocesseurs hybrides. Phd. thesis, Université de Versailles Saint Quentin en Yvelines, December 1996.

[31] L. Bouganim, B. Dageville, and P. Valduriez. Adaptive parallel query execution in dbs3. In *Proceedings of the 5th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '96, pages 481–484, London, UK, UK, 1996. Springer-Verlag.

[32] K. Boukhalfa. de la conception physique aux outils d'adlinistration et de tuning des entrepôts de données ". Phd. thesis, Université de Clermont-Ferrand II Ecole nationale supérieure de mécaniqye et d'aérotechnique, Juillet 2009.

[33] L. Breslau, P. Cue, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: Evidence and implications. In *In INFOCOM*, pages 126–134, 1999.

[34] R. L. Cannon, J. V. Dave, and J. C. Bezdek. Efficient implementation of the fuzzy c-means clustering algorithms. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 8(2):248–255, 1986.

[35] S. Ceri, M. Negri, and G. Pelagatti. Horizontal data partitioning in database design. *1982 ACM SIGMOD International Conference on Management of Data*, pages 128–136, 1982.

[36] R.-S. Chang, H.-P. Chang, and Y.-T. Wang. A dynamic weighted data replication strategy in data grids. In *Proceedings of the 2008 IEEE/ACS International Conference on Computer Systems and Applications*, AICCSA '08, pages 414–421, Washington, DC, USA, 2008. IEEE Computer Society.

[37] S. Chaudhuri and U. Dayal. An overview of data warehousing and olap technology. *Sigmod Record*, 26(1):65–74, March 1997.

[38] S. Chaudhuri and V. Narasayya. Autoadmin 'what-if' index analysis utility. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 367–378, June 1998.

[39] S. Chaudhuri and V. Narasayya. Self-tuning database systems: A decade of progress. *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 3–14, September 2007.

[40] C. H. Cheng, W.-K. Lee, and K.-F. Wong. A genetic algorithm-based clustering approach for database partitioning. *IEEE Transactions on Systems, Man, and Cybernetics, Part C*, 32(3):215–230, 2002.

[41] H.-T. Chou and D. J. DeWitt. An evaluation of buffer management strategies for relational database systems. In *Proceedings of the 11th International Conference on Very Large Data Bases - Volume 11*, VLDB '85, pages 127–141. VLDB Endowment, 1985.

[42] L. Chung, J. Cesar, and S. P. Leite. Non-functional requirements in software engineering, 1999.

[43] B. Ciciani, D. M. Dias, and P. S. Yu. Analysis of replication in distributed database systems. *IEEE Trans. on Knowl. and Data Eng.*, pages 247–261, 1990.

[44] G. P. Copeland, W. Alexander, E. Boughter, and T. Keller. Data placement in bubba. *ACM SIGMOD International Conference on Management of Data*, pages 99–108, 1988.

[45] A. L. Corcoran and J. Hale. A genetic algorithm for fragment allocation in a distributed database system. In *Proceedings of the 1994 ACM symposium on Applied computing*, SAC '94, pages 247–250, New York, NY, USA, 1994. ACM.

[46] N. Corporation. Netezza database user guide. 2009.

[47] N. Corporation. Greenplum database 4.1: Administrator guide. *EMC Corporation*, 2011.

[48] J. P. Costa and P. Furtado. Poster session: Towards a qos-aware dbms. In *ICDE Workshops*, pages 50–55, 2008.

[49] O. Council. Apb-1 olap benchmark, release ii. *http://www.olapcouncil.org/research/bmarkly.htm*, 1998.

[50] A. Cuzzocrea, J. Darmont, and H. Mahboubi. Fragmenting very large xml data warehouses via k-means clustering algorithm. *International Journal of Business Intelligence and Data Mining*, 4(3-4):301–328, 2009.

[51] B. Dageville, D. Das, K. Dias, K. Yagoub, M. Zait, and M. Ziauddin. Automatic sql tuning in oracle 10g. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, pages 1098–1109. VLDB Endowment, 2004.

[52] A. S. Darabant and A. Campan. Semi-supervised learning techniques: k-means clustering in OODB Fragmentation. In *Second IEEE International Conference on Computational Cybernetics (ICCC 04), Vienna, Austria*, pages 333–338. IEEE Computer Society, 2004.

[53] A. S. Darabant, A. Campan, and O. Cret. Using fuzzy clustering for advanced oodb horizontal fragmentation with fine-grained replication. In *Databases and Applications*, pages 116–121, 2005.

[54] A. Datta, B. Moon, and H. Thomas. A case for parallelism in data warehousing and olap. *in the 9th International Workshop on Database and Expert Systems Applications (DEXA98)*, pages 226–231, August 1998.

[55] L. Davis. Bit-climbing, representational bias, and test suite design. *4th International Conference on Genetic Algorithms*, pages 18–23, 1991.

[56] D. Dewitt, R. H. Gerber, G. Graefe, M. L. Heytens, K. B. Kumar, and M. Muralikrishna. Gamma - a high performance dataflow database machine. *VLDB*, 10:228–237, 1986.

[57] D. DeWitt, S. Madden, and M. Stonebraker. How to build a high-performance data warehouse. *http://db.lcs.mit.edu/madden/high_perf.pdf*.

[58] D. J. DeWitt, J. F. Naughton, D. A. Schneider, and S. Seshadri. Practical skew handling in parallel joins. In *Proceedings of the 18th International Conference on Very Large Data Bases*, VLDB '92, pages 27–40, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc.

[59] N. ElGamal, A. ElBastawissy, and G. Galal-Edeen. Data warehouse testing. In *Proceedings of the Joint EDBT/ICDT 2013 Workshops*, EDBT '13, pages 1–8, New York, NY, USA, 2013. ACM.

[60] C. I. Ezeife and K. Barker. Vertical fragmentation for advanced object models in a distributed object based system. In *In Proc. 8th Int. Conf. on Computing and Information. IEEE*. Publishers, 1995.

[61] A. Forestiero, C. Mastroianni, and G. Spezzano. Qos-based dissemination of content in grids. *Future Generation Comp. Syst.*, 24(3):235–244, 2008.

[62] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: enabling scalable virtual organizations. *International Journal of High Performance Computing Applications*, 15(3):200–222, 2001.

[63] Y. fu Huang and J. her Chen. Fragment allocation in distributed database design. *Journal of Information Science and Engineering*, 17:491–506, 2001.

[64] P. Furtado. Experimental evidence on partitioning in parallel data warehouses. In *7th ACM International Workshop on Data Warehousing and OLAP (DOLAP)*, pages 23–30, 2004.

[65] C. A. Galindo-Legaria, A. Pellenkoft, and M. L. Kersten. Fast, randomized join-order selection - why use transformations? In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 85–95, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[66] R. Gallersdörfer and M. Nicola. Improving performance in replicated databases through relaxed coherency. In *Proceedings of the 21th International Conference on Very Large Data Bases*, VLDB '95, pages 445–456, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[67] M. Golfarelli. Data warehouse life-cycle and design. In *Encyclopedia of Database Systems*, pages 658–664. 2009.

[68] M. Golfarelli, D. Maio, and S. Rizzi. The dimensional fact model: A conceptual model for data warehouses. *International Journal of Cooperative Information Systems*, 7:215–247, 1998.

[69] N. Gorla. Subquery allocations in distributed databases using genetic algorithms. *Journal of Computer Science & Technology*, 1, 2010.

[70] N. Gorla and B. P. W. Yan. Vertical fragmentation in databases using data-mining technique. In *Database Technologies: Concepts, Methodologies, Tools, and Applications*, pages 2543–2563. 2009.

[71] A. Gounaris, J. Smith, N. W. Paton, R. Sakellariou, A. A. A. Fernandes, and P. Watson. Adaptive workload allocation in query processing in autonomous heterogeneous environments. *Distributed and Parallel Databases*, 25(3):125–164, 2009.

[72] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. *SIGMOD Rec.*, 25(2):173–182, June 1996.

[73] T. P. Group. A benchmark of nonstop sql on the debit credit transaction. *SIGMOD Rec.*, 17(3):337–341, June 1988.

[74] J. R. Gruser. Modèle de coût pour l'optimisation de requête objet. Thèse de doctorat, Université de Paris VI, Décembre 1996.

[75] R. Gupta, O. Kumar, and A. Sharma. Article: Analytical study of various high performance computing paradigms. *International Journal of Applied Information Systems*, 1(9):16–21, April 2012. Published by Foundation of Computer Science, New York, USA.

[76] I. O. Hababeh, M. Ramachandran, and N. Bowring. A high-performance computing method for data allocation in distributed database systems. *J. Supercomput.*, 39(1):3–18, Jan. 2007.

[77] W. HASAN and et al. Open issues in parallel query optimization, 1996.

[78] J. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, Michigan, 1975.

[79] H. i Hsiao and D. J. Dewitt. Chained declustering: A new availability strategy for multiprocssor database machines. In *in Proceedings of 6th International Data Engineering Conference*, pages 456–465, 1990.

[80] T. Ibaraki and T. Kameda. On the optimal nesting order for computing n-relational joins. *ACM Trans. Database Syst.*, 9(3):482–502, Sept. 1984.

[81] W. H. Inmon. *Building the Data Warehouse*. John Wiley and Sons, Third edition, 2002.

[82] Y. Ioannidis and Y. Kang. Randomized algorithms algorithms for optimizing large join queries. In *1990 ACM SIGMOD International Conference on Management of Data*, pages 9–22, 1990.

[83] Y. E. Ioannidis. Universality of serial histograms. *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 256–267, August 1993.

[84] Y. E. Ioannidis and S. Christodoulakis. On the propagation of errors in the size of join results. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 268–277, June 1991.

[85] Y. E. Ioannidis and E. Wong. Query optimization by simulated annealing. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, SIGMOD '87, pages 9–22, New York, NY, USA, 1987. ACM.

[86] M. R. Jensen, T. Holmgren, and T. B. Pedersen. Discovering multidimensional structure in relational data. In *DaWaK*, pages 138–148, 2004.

[87] E. G. C. Jr., J. Y.-T. Leung, and D. W. Ting. Bin packing: Maximizing the number of pieces packed. 9:263–271, 1978.

[88] R. Karimi Adl and S. M. T. Rouhani Rankoohi. A new ant colony optimization based algorithm for data allocation problem in distributed databases. *Knowl. Inf. Syst.*, 20(3):349–373, Aug. 2009.

[89] K. Karlapalem and N. Pun. Query driven data allocation algorithms for distributed database systems. In *8th International Conference on Database and Expert Systems Applications*, pages 347–356, 1997.

[90] R. Kimball, L. Reeves, W. Thornthwaite, M. Ross, and W. Thornwaite. *The Data Warehouse Lifecycle Toolkit: Expert Methods for Designing, Developing and Deploying Data Warehouses with CD Rom.* John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1998.

[91] M. Kitsuregawa and Y. Ogawa. Bucket spreading parallel hash: A new, robust, parallel hash join method for data skew in the super database computer (sdc). In *Proceedings of the 16th International Conference on Very Large Data Bases*, VLDB '90, pages 210–221, San Francisco, CA, USA, 1990. Morgan Kaufmann Publishers Inc.

[92] W. Labio, D. Quass, and B. Adelberg. Physical database design for data warehouses. *Proceedings of the International Conference on Data Engineering (ICDE)*, 1997.

[93] B. Ladjel. Contributions à la conception et l'exploitation des systèmes d'intégration de données. Hdr. thesis, Université de Poitiers, Novembre 2009.

[94] B. Ladjel, B. Kamel, R. Pascal, and B. Soumia. pages 953–973. Wiley, 2012.

[95] A. A. B. Lima, M. Mattoso, and P. Valduriez. Adaptive Virtual Partitioning for OLAP Query Processing in a Database Cluster. In S. Lifschitz, editor, *SBBD 2004*, pages 92–105, Brasilia, Brésil, 2004.

[96] A. A. B. Lima, M. Mattoso, and P. Valduriez. Olap query processing in a database cluster. In *Euro-Par*, pages 355–362, 2004.

[97] A. B. Lima, C. Furtado, P. Valduriez, and M. Mattoso. Parallel olap query processing in database clusters with data replication. distributed and parallel databases. *Distributed and Parallel Database Journal*, 25(1-2):97–123, 2009.

[98] M. G. Lohman, D. Daniels, L. M. Haas, R. Kistler, and P. G. Selinger. Optimization of nested queries in a distributed relational database. *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 403–415, August 1984.

[99] T. Loukopoulos and I. Ahmad. Static and adaptive distributed data replication using genetic algorithms. *in Journal of Parallel and Distributed Computing*, 64(11):1270–1285, November 2004.

[100] T. Mahapatra and S. Mishra. *Oracle Parallel Processing.* O'Reilly, 2000.

[101] H. Märtens, E. Rahm, and T. Stöhr. Dynamic query scheduling in parallel data warehouses: Concurrency computation practice and experience. In *International Conference on Parallel Processing (Euro-Par02*, pages 11–12, 2003.

[102] M. Mehta and D. J. DeWitt. Data placement in shared-nothing parallel database systems. *VLDB Journal*, 6(1):53–72, 1997.

[103] S. Menon. Allocating fragments in distributed databases. *IEEE Transactions on Parallel and Distributed Systems*, 16(7):577–585, July 2005.

[104] D. Moore and J. Heleber. *Peer-to-peer: building secure, scalable, and manageable networks*. Osborne, 2002.

[105] J. Mylopoulos, L. Chung, and B. Nixon. Representing and using non-functional requirements: A process-oriented approach. *IEEE Transactions on Software Engineering*, 18:483–497, 1992.

[106] H. Naacke. Modï¿½les de coïï¿½t pour mïï¿½diateurs de bases de donnïï¿½es hïï¿½tïï¿½rogïï¿½nes. Thèse de doctorat, Universitï¿½ de Versailles Saint-Quentin-en-Yvelines, Séptembre 1999.

[107] S. Navathe, S. Ceri, G. Wiederhold, and J. Dou. Vertical partitioning algorithms for database design. *ACM Transaction on Database Systems*, 9(4):681–710, 1984.

[108] S. Navathe, K. Karlapalem, and M. Ra. A mixed partitioning methodology for distributed database design. *Journal of Computer and Software Engineering*, 3(4):395–426, 1995.

[109] S. Navathe and M. Ra. Vertical partitioning for database design : a graphical algorithm. *1989 ACM SIGMOD International Conference on Management of Data*, pages 440–450, 1989.

[110] R. V. Nehme and N. Bruno. Automated partitioning design in parallel database systems. In *SIGMOD Conference*, pages 1137–1148, 2011.

[111] A. Y. Noaman and K. Barker. A horizontal fragmentation algorithm for the fact relation in a distributed data warehouse. *in the 8th International Conference on Information and Knowledge Management (CIKM'99)*, pages 154–161, November 1999.

[112] B. Nuseibeh and S. Easterbrook. Requirements engineering: A roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 35–46, New York, NY, USA, 2000. ACM.

[113] P. O'Neil, E. B. O'Neil, and X. Chen. The star schema benchmark: http://www.cs.umb.edu/ poneil/starschemab.pdf. 2007.

[114] Oracle. Exadata technical overview.

[115] M. Özsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, 1991.

[116] M. T. Özsu and P. Valduriez. Distributed database systems : Where are we now? *IEEE COMPUTER*, 24(8):68–78, August 1991.

[117] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems : Second Edition*. Prentice Hall, 1999.

[118] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems, Third Edition*. Springer, 2011.

[119] T. M. Özsu and P. Valduriez. *Principles of Distributed Database Systems, third edition*. Springer, 2011.

[120] A. Pavlo, C. Curino, and S. Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel oltp systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 61–72, New York, NY, USA, 2012. ACM.

[121] T. Phan and W.-S. Li. Load distribution of analytical query workloads for database cluster architectures. In *EDBT*, pages 169–180, 2008.

[122] R. Ramakrishnan. *Database Management Systems*. WCB/McGraw Hill, 1998.

[123] J. Rao, C. Zhang, G. Lohman, and N. Megiddo. Automating physical database design in a parallel database. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 558–569, June 2002.

[124] U. Röhm, K. Böhm, and H.-J. Schek. Olap query routing and physical design in a database cluster. In *EDBT*, pages 254–268, 2000.

[125] D. Saccà and G. Wiederhold. Database partitioning in a cluster of processors. *ACM Transactions on Database Systems*, 10(1):29–56, 1985.

[126] R. Sarathy, B. Shetty, and A. Sen. A constrained nonlinear 0-1 program for data allocation. *European Journal of Operational Research*, 102(3):626–647, 1997.

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

[128] T. K. Sellis and A. Simitsis. Etl workflows: From formal specification to optimization. In *ADBIS*, pages 1–11, 2007.

[129] K. Selma. *Cycle de vie sémantique de conception de systèmes de stockage et de manipulation de données*. PhD thesis, oct 2013.

[130] O. D. Sheet. Oracle partitioning. *White Paper: http://www.oracle.com/technology/products/bi/db/11g/*, 2007.

[131] B. Soumia, B. Ladjel, and A. Cuzzocrea. Omniscience dans la conception des entrepôts de données parallèles sur un cluster. In RNTI, editor, *EDA*, pages 43–52, 2009.

[132] M. Steinbrunn, G. Moerkotte, and A. Kemper. Heuristic and randomized optimization for the join ordering problem. *The VLDB Journal*, 6(3):191–208, Aug. 1997.

[133] T. Stöhr, H. Märtens, and E. Rahm. Multi-dimensional database allocation for parallel data warehouses. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 273–284, 2000.

[134] T. Stöhr and E. Rahm. Warlock: A data allocation tool for parallel warehouses. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 721–722, 2001.

[135] D. Taniar, C. H. C. Leung, W. Rahayu, and S. Goel. *High Performance Parallel Database Processing and Grid Databases*. Wiley Publishing, 2008.

[136] D. Taniar and J. W. Rahayu. A taxonomy of indexing schemes for parallel database systems. *Distributed and Parallel Databases*, 12(1):73–106, 2002.

[137] Teradata. Dbc/1012 database computer system manual release 2.0. *Technical Document C10-0001-02*, 1985.

[138] Teradata. Database. database design. teradata. *Release 13.10 B035-1094-109A*, October 2010.

[139] M. Thiele, A. Bader, and W. Lehner. Multi-objective scheduling for real-time data warehouses. *Computer Science - R&D*, 24(3):137–151, 2009.

[140] A. Tucker, J. Crampton, and S. Swift. Rgfga: An efficient representation and crossover for grouping genetic algorithms. *Evolutionary Computation*, 13(4):477–499, 2005.

[141] J. D. Ullman. *Principles of Database and Knowledge-Base Systems*. vol. II. Computer Science Press, 1989.

[142] P. Valduriez. Parallel database systems: the case for shared-something. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 460–465, 1993.

[143] R. Vanderbei. *Linear Programming: Foundations and Extensions, Second Edition*. Springer, 2001.

[144] C. B. Walton, A. G. Dale, and R. M. Jenevein. A taxonomy and performance model of data skew effects in parallel joins. In *Proceedings of the 17th International Conference on Very Large Data Bases*, VLDB '91, pages 537–548, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc.

[145] R. Winter and B. Strauch. A method for demand-driven information requirements analysis in data warehousing projects. In *Proceedings of the 36th Annual Hawaii International Conference on System Sciences (HICSS'03) - Track 8 - Volume 8*, HICSS '03, pages 231.1–, Washington, DC, USA, 2003. IEEE Computer Society.

[146] J. L. Wolf, D. M. Dias, P. S. Yu, and J. Turek. New algorithms for parallelizing relational database joins in the presence of data skew. *IEEE Trans. on Knowl. and Data Eng.*, 6(6):990–997, Dec. 1994.

[147] O. Wolfson and A. Milo. The multicast policy and its relationship to replicated data placement. *ACM Trans. Database Syst.*, 16(1):181–205, Mar. 1991.

[148] S. yih Hwang, K. K. Lee, and Y. H. Chin. Data replication in a distributed system: A performance study. In *7 th International Conference on Database and Expert Systems Applications*, pages 708–717. Springer-Verlag, 1996.

[149] J. X. Yu, X. Yao, C.-H. Choi, and G. Gou. Materialized view selection as constrained evolutionary optimization. *Trans. Sys. Man Cyber Part C*, 33(4):458–467, Nov. 2003.

[150] C. Zhang and J. Yang. Genetic algorithm for materialized view selection in data warehouse environments. In *1st International Conference on Data Warehousing and Knowledge Discovery (DAWAK)*, pages 116–125, 1999.

[151] H. Zhu, P. Gu, and J. Wang. Shifted declustering: a placement-ideal layout scheme for multi-way replication storage architecture. In *Proceedings of the 22nd annual international conference on Supercomputing*, ICS '08, pages 134–144, New York, NY, USA, 2008. ACM.

[152] D. C. Zilio, A. Jhingran, and S. Padmanabhan. Partitioning key selection for a shared-nothing parallel database system. In *IBM Research Report RC*, 1994.

[153] D. C. Zilio, J. Rao, S. Lightstone, G. M. Lohman, A. Storm, C. Garcia-Arellano, and S. Fadden. Db2 design advisor: Integrated automatic physical database design. *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 1087–1097, August 2004.

Résumé

La conception d'un entrepôt de données parallèle consiste à choisir l'architecture matérielle, à fragmenter le schéma d'entrepôt de données, à allouer les fragments générés, à répliquer les fragments pour assurer une haute performance du système et à définir la stratégie de traitement et d'équilibrage de charges. L'inconvénient majeur de ce cycle de conception est son ignorance de l'interdépendance entre les sous-problèmes liés à la conception d'un EDP et l'utilisation des métriques hétérogènes pour atteindre le même objectif. Notre première proposition définie un modèle de coût analytique pour le traitement parallèle des requêtes OLAP dans un environnement cluster. Notre deuxième proposition prend en considération l'interdépendance existante entre la fragmentation et l'allocation. Dans ce contexte, nous avons proposé une nouvelle approche de conception d'un EDP sur un cluster de machine. Durant le processus de fragmentation, notre approche décide si le schéma de fragmentation généré est pertinent pour le processus d'allocation. Les résultats obtenus sont très encourageant et une validation est faite sur Teradata. Notre troisième proposition consiste à présenter une méthode de conception qui est une extension de notre travail. Dans cette phase, une méthode de réplication originale, basée sur la logique floue, est intégrée.

Mots-clés: Entrepôt de données; architecture parallèle; fragmentation; allocation, réplication, équilibrage de charge, modèle de coût analytique.

Abstract

Designing a parallel data warehouse consists of choosing the hardware architecture, fragmenting the data warehouse schema, allocating the generated fragments, replicating fragments to ensure high system performance and defining the treatment strategy and load balancing. The major drawback of this design cycle is its ignorance of the interdependence between subproblems related to the design of PDW and the use of heterogeneous metrics to achieve the same goal. Our first proposal defines an analytical cost model for parallel processing of OLAP queries in a cluster environment. Our second takes into account the interdependence existing between fragmentation and allocation. In this context, we proposed a new approach to design a PDW on a cluster machine. During the fragmentation process, our approach determines whether the fragmentation pattern generated is relevant to the allocation process or not. The results are very encouraging and validation is done on Teradata. For our third proposition, we presented a design method which is an extension of our work. In this phase, an original method of replication, based on fuzzy logic is integrated.

Keywords: Data Warehouse; parallel architecture; fragmentation; allocation, replication, Load balancing, analytical cost model.