République Algérienne Démocratique & Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Université Ferhat ABBAS – SETIF-1
(UFAS-1). (ALGERIE)

THESE

Présentée à la Faculté des Sciences
Département d'Informatique
Pour l'Obtention du Diplôme de

Doctorat En Sciences

Thème

Cubes ROLAP sous forme de SDDS multidimensionnelles

Par :

Amel MECHRI

Soutenu le : **devant la commission d'examen :**

Pr. Fouzi SEMCHEDINNE	Président	Professeur à l'Université Ferhat ABBAS de Sétif 1
Pr. Djamel Eddine ZEGOUR	Directeur	Professeur à l'Ecole Supérieure d'Informatique ESI, Alger
Pr. Walid-Khaled HIDOUCI	Co-Directeur	Professeur à l'Ecole Supérieure d'Informatique ESI, Alger
Dr. Fatima BOUMAHDI	Examinatrice	MCA à l'Université Saâd DAHLEB de Blida 1
Dr. Amina MADANI	Examinatrice	MCA à l'Université Saâd DAHLEB de Blida 1
Dr. Samir FENANIR	Examinateur	MCA à l'Université Ferhat ABBAS de Sétif 1
Dr. Fateh SEGHIR	Examinateur	MCA à l'Université Ferhat ABBAS de Sétif 1

Résumé

Les Structures de Données Distribuées et Scalables (SDDS) constitue un modèle de gestion de données spécifiquement conçu pour les environnements distribués, dans lesquels les données doivent être partitionnées et réparties dynamiquement entre plusieurs serveurs tout en garantissant la scalabilité du système.

Un fichier SDDS se compose d'enregistrements distribués de manière dynamique sur un ensemble de serveurs, organisés selon différentes catégories de SDDS, chacune reposant sur des algorithmes spécifiques, telles que le hachage linéaire (LH*), le partitionnement par intervalle (RP*) ou encore le digital hashing (CTH*).

Une caractéristique fondamentale des SDDS est l'élimination de tout composant centralisé de gestion des adresses, simplifiant ainsi la communication entre les clients et les serveurs. En supprimant la nécessité d'un point de coordination central, qui peut constituer un goulot d'étranglement, cette architecture réduit considérablement le nombre de messages échangés et diminue le temps d'accès aux données, améliorant ainsi l'efficacité globale des systèmes distribués.

En parallèle, l'entrepôt de données (Data Warehouse, DW) joue un rôle fondamental dans les systèmes d'aide à la décision en permettant l'analyse et l'exploration de grandes quantités de données multidimensionnelles. Ces données sont représentées sous forme de cube et sont exploitées via le traitement analytique en ligne relationnel (ROLAP), qui repose sur des bases de données relationnelles pour stocker et interroger les données.

Bien que des recherches approfondies aient été réalisées dans les domaines des entrepôts de données et des SDDS, aucune étude antérieure n'a combiné ces deux domaines. Notre travail propose une approche innovante pour implémenter un cube ROLAP dans une SDDS en utilisant l'algorithme de hachage linéaire (LH*), permettant ainsi d'optimiser les cubes ROLAP et de concevoir un système de gestion et d'analyse des données dans des systèmes distribués et scalables performant.

Mots clés : SDDS, Hachage linéaire dynamique, Algorithme de hachage linéaire LH*, Entrepôt de données, Cube ROLAP, Données multidimensionnelles, Opérations OLAP.

Abstract

Scalable Distributed Data Structures (SDDS) is a data model specifically designed for distributed environments, where data must be dynamically partitioned and distributed across multiple servers while ensuring system scalability.

An SDDS file consists of records that are dynamically distributed over servers. These records are organized based on various types of SDDS, each relying on specific algorithms, such as Linear Hashing (LH*), Range Partitioning (RP*), or Digital Hashing (CTH*).

A key feature of SDDS is the elimination of any centralized address component, thereby simplifying communication between clients and servers. By removing the need for a central coordination point, which can act as a bottleneck, this architecture significantly reduces the number of exchanged messages and shortens data access times, ultimately improving the overall efficiency of distributed systems.

Meanwhile, the Data Warehouse (DW) plays a critical role in decision support systems by enabling the analysis and exploration of large volumes of multidimensional data. These data are represented in the form of a cube and utilized through Relational Online Analytical Processing (ROLAP), which leverages relational databases to store and query the data effectively.

Although extensive research has been conducted in the fields of Data Warehouses and SDDS, no prior studies have combined these two domains. Our work introduces an innovative approach to implementing a ROLAP cube within an SDDS using the Linear Hashing algorithm (LH*). This approach optimizes ROLAP cubes and enables the design of an efficient data management and analysis system for scalable and distributed environments.

Keys words : SDDS, Dynamic linear hashing, Linear hashing algorithm LH*, Data warehouse, ROLAP cube, Multidimensional data, OLAP operations.

ملخص

تشكل الهياكل الموزعة والقابلة للتوسع (SDDS) نموذجًا لإدارة البيانات مصممًا خصيصًا للبيئات الموزعة، حيث يجب تقسيم البيانات وتوزيعها ديناميكيًا بين عدة خوادم مع ضمان قابلية النظام للتوسع.

يتكون ملف SDDS من سجلات يتم توزيعها ديناميكيًا على مجموعة من الخوادم، ويتم تنظيمها وفقًا لأنواع مختلفة من SDDS ، يعتمد كل نوع منها على خوارزميات محددة، مثل التجزئة الخطية (LH*) ، أو التقسيم حسب النطاق (RP*) ، أو التجزئة الرقمية (CTH*).

إحدى الخصائص الأساسية لـ SDDS هي التخلص من أي مكون مركزي لإدارة العناوين، مما يبسط التواصل بين العملاء والخوادم. من خلال إزالة الحاجة إلى نقطة تنسيق مركزية، والتي يمكن أن تشكل عنق زجاجة، تقلل هذه البنية بشكل كبير من عدد الرسائل المتبادلة وتقلل من زمن الوصول إلى البيانات، مما يحسن الكفاءة العامة للأنظمة الموزعة.

يلعب مستودع البيانات (DW) دورًا أساسيًا في أنظمة دعم القرار، حيث يتيح تحليل واستكشاف كميات كبيرة من البيانات متعددة الأبعاد. يتم تمثيل هذه البيانات على شكل مكعب وتُستغل عبر المعالجة التحليلية العلائقية عبر الإنترنت (ROLAP) ، التي تعتمد على قواعد بيانات علائقية لتخزين البيانات واستجوابها.

على الرغم من إجراء أبحاث معمقة في مجالي مستودعات البيانات وSDDS، لم تجمع أي دراسة سابقة بين هذين المجالين. يقترح عملنا نهجًا مبتكرًا لتطبيق مكعب ROLAP داخل SDDS باستخدام خوارزمية التجزئة الخطية .(LH*) يتيح هذا النهج تحسين مكعبات ROLAP وتصميم نظام فعال لإدارة وتحليل البيانات في بيئات موزعة وقابلة للتوسع.

الكلمات المفتاحية:

الهياكل البيانية الموزعة والقابلة للتوسع، التجزئة الخطية الديناميكية، خوارزمية التجزئة الخطية *LH، مستودع البيانات، مكعب المعالجة التحليلية العلائقية عبر الإنترنت، البيانات متعددة الأبعاد، عمليات المعالجة التحليلية عبر الإنترنت.

Table des matières

ChapitreIII : Les Structures de Données Distribuées et Scalables (SDDS)

Partie II : Conception et implémentation d'un entrepôt de données classique et en SDDS DW_SDDS

ChapitreIV : Conception d'un entrepôt de données classique et en SDDS DW_SDDS

ChapitreV : Implémentation d'un entrepôt de données classique et en SDDS DW_SDDS

Liste des figures

ChapitreII : Les entrepôts de données (Data warehouse)

ChapitreIII : Les Structures de Données Distribuées et Scalables (SDDS)

ChapitreIV : Conception d'un entrepôt de données classique et en SDDS DW_SDDS

ChapitreV : Conception d'un entrepôt de données classique et en SDDS DW_SDDS

ChapitreVI : Résultats et discussion

Liste des tableaux

Liste des algorithmes

Acronymes

BD : Bases de Données

CTH : Compact Trie Hashing

DW : Data Warehouse

ED : Entrepôt de Données

HOLAP : Hybrid On-Line Analytical Processing

IAM : Image Adjustment Message

JDBC : Java DataBase Connectivity

JVM : Java Virtual Machine

LH : Linear Hashing

MOLAP : Multidimensional On-Line Analytical Processing

MySQL : My Structured Query Language

OLAP : On-Line Analytical Processing

OLTP : Online Transaction Processing

ROLAP : Relational On-Line Analytical Processing

RP : Range Partitionning

SDDS : Scalables Distributed Data Structures

SGBDR : Système de Gestion de Bases de Données Relationnelles

SQL : Structured Query Language

TCP : Transmission Control Protocol

TH : Trie Hashing

UDP : User Datagram Protocol

Chapitre I

Introduction générale

I.1. Contexte d'étude

Le concept de Structure de Données Distribuée Scalable (SDDS) a été introduit pour la première fois en 1993. Il constitue un modèle de gestion de données spécifiquement conçu pour les environnements distribués, dans lesquels les données doivent être partitionnées et réparties dynamiquement entre plusieurs serveurs tout en garantissant la scalabilité du système.

Un fichier SDDS se compose d'enregistrements distribués de manière dynamique sur un ensemble de serveurs, organisés selon différentes catégories de SDDS, chacune reposant sur des algorithmes spécifiques, telles que le hachage linéaire (LH*), le partitionnement par intervalle (RP*) ou le digital hashing (CTH*).

Une caractéristique fondamentale des SDDS est l'élimination de tout composant centralisé de gestion des adresses, simplifiant ainsi la communication entre les clients et les serveurs. En supprimant la nécessité d'un point de coordination central, qui peut constituer un goulot d'étranglement. Cette architecture réduit considérablement le nombre de messages échangés et diminue le temps d'accès aux données, améliorant ainsi l'efficacité globale des systèmes distribués.

En parallèle, l'entrepôt de données (Data Warehouse, DW) joue un rôle fondamental dans les systèmes d'aide à la décision en permettant l'analyse et l'exploration de grandes quantités de données multidimensionnelles. Ces données sont représentées sous forme de cube et sont exploitées via le traitement analytique en ligne relationnel (ROLAP), qui repose sur des bases de données relationnelles pour stocker et interroger les données.

I.2. Problématique

Bien que les SDDS et les entrepôts de données soient chacun largement étudiés, rare sont celles ayant exploré la combinaison de ces deux approches. En effet, intégrer ces deux concepts pourrait offrir des gains considérables en termes de performances, de gestion et d'analyse des données dans les systèmes distribués et scalables.

L'objectif de notre travail, est donc d'essayer d'intégrer les SDDS avec les cubes ROLAP pour les optimiser. Une nouvelle approche consistant à combiner les forces des SDDS avec celles du ROLAP permettrait de concevoir un système analytique distribué performant.

Toutes fois, cette démarche représente un certains nombre de défis à relever :

1. Adapter les SDDS pour gérer les données multidimensionnelles des cubes ROLAP.
2. Implémenter une architecture garantissant des communications fiables et efficaces entre les différents composants du système distribué.
3. Assurer l'exécution des requêtes SQL de manière transparente pour l'utilisateur dans un environnement distribué.
4. Comparer les performances d'un cube ROLAP classique distribué avec celles d'une version exploitant les SDDS.

I.3. Objectifs

La présente étude propose une méthode innovante pour implémenter un cube ROLAP au sein d'une SDDS. Plus précisément, cette implémentation s'appuie sur l'algorithme de hachage linéaire (LH*), reconnu pour ses performances et son adaptabilité dans les systèmes distribués.

Les objectifs spécifiques de ce travail sont les suivants :

1. Construire une première version d'un cube ROLAP classique et distribué basée sur une base de données relationnelle (MySQL).
2. Développer une deuxième version distribuée d'un cube ROLAP en intégrant des SDDS. Cette version est basée sur l'algorithme de hachage linéaire LH* pour la gestion des données multidimensionnelles.
3. Mettre en place des mécanismes d'exécution de requêtes SQL, en garantissant des opérations telles que les agrégations, filtres et jointures soient performants dans un système distribué.
4. Comparer les deux approches en termes de scalabilité, temps de réponse et nombre de messages échangés.
5. Améliorer la scalabilité et les performances d'accès et de manipulation des données distribuées.

I.4. Contributions

Par l'intégration d'un cube ROLAP dans une SDDS, des avantages significatifs ont étaient démontrés, notamment une réduction des messages échangés entre les deivers composants, un temps d'accès aux données réparties amélioré, et une gestion des données plus efficace, par rapport aux approches classiques basées sur un site de communication central.

Le présent travail a abouti aux contributions majeures suivantes :

1. **Une nouvelle méthode d'intégration** : La mise en œuvre d'un cube ROLAP au sein d'une SDDS en utilisant le modèle de hachage linéaire LH*, permettant une gestion distribuée et dynamique des données multidimensionnelles.
2. **Une amélioration des performances** : Par rapport aux architectures traditionnelles, l'approche proposée démontre que le nombre de messages échangés entre les serveurs et les clients a été diminué, ce qui par conséquence, à permis de réduire le temps d'accès aux données, améliorant ainsi la gestion globale des ressources.
3. Une méthodologie claire et bien structurée pour implémenter les cubes ROLAP en SDDS.
4. Concevoir un prototype fonctionnel reposant sur Java comme langage de programmation, MySQL comme système de gestion de base de données, et des sockets pour assurer la communication entre les nœuds du système distribué.
5. Une évaluation approfondie des performances de l'approche distribuée classique et en SDDS.

En résumé, cette étude propose une solution innovante pour surmonter les limites des approches traditionnelles en matière de gestion et d'analyse de données dans des systèmes distribués. Elle établit de nouvelles bases pour la recherche en intégrant les structures de données distribuées et scalables avec les cubes ROLAP, offrant ainsi des perspectives prometteuses pour le développement de systèmes analytiques distribués performants

I.5. Organisation de la thèse

La thèse se compose de sept chapitres, structurés de la manière suivante :

- L'introduction débute par une présentation du contexte d'étude, abordant les enjeux des SDDS et des entrepôts de données. Ensuite par la problématique, en met en évidence les défis actuels liés aux cubes ROLAP et à leur intégration dans une SDDS. Les objectifs à atteindre sont clairement définis, notamment l'optimisation des performances et la mise en place d'une solution distribuée efficace pour implémenter un cube ROLAP au sein d'une SDDS. Les contributions principales du travail incluent la proposition de nouvelles méthodologies d'intégration hybride et le développement d'un prototype fonctionnel. L'organisation de la thèse est ensuite détaillée, expliquant l'enchainement

des chapitres. Enfin, les publications académiques issues de ce travail sont présentées pour valoriser les résultats obtenus.

- Après l'introduction, le deuxième chapitre présente une étude approfondie des entrepôts de données, en abordant leur construction, structure, modélisation et implémentation. Il explore également l'analyse en ligne (OLAP), tout en décrivant les types de serveurs et les essentielles opérations OLAP applicables pour l'interrogation d'un cube de données. Enfin, le chapitre se conclut par une présentation de la distribution d'un entrepôt, en détaillant les principes de fragmentation, de répartition et de l'architecture client-serveur.
- Le troisième chapitre est consacré à l'étude des structures de données distribuées (SDDS). Il en expose leurs principes fondamentaux, leurs caractéristiques et leur classification. De plus, pour chaque classe, on a choisi de décrire un des algorithmes les plus répondus, accompagné de ses principales variantes et illustré par des exemples bien détaillés pour faciliter la compréhension du principe de fonctionnement.
- Le quatrième chapitre décrit en détail la conception des deux entrepôts de données : un entrepôt classique et un entrepôt reposant sur les SDDS (DW_SDDS). Ce chapitre couvre la conception des entrepôts, leur architecture, ainsi que le processus d'exécution des requêtes d'ajout et de recherche pour gérer et exploiter les données.
- Le cinquième chapitre détaille la mise en place des entrepôts de données classiques et ceux basés sur des SDDS en précisant les choix techniques. Il commence par expliquer le choix du Java comme langage de programmation, MySQL comme moteur de gestion des bases de données et finalement les sockets comme moyen de communication. Par la suite, on décrit le processus d'exécution des requêtes, depuis l'initiation d'une demande de connexion par le client jusqu'à la réception des résultats par celui-ci. Les différentes connexions établies et les procédures exécutées par tous les composants du système sont également expliquées.
- L'avant dernier chapitre propose une comparaison des deux entrepôts présentés précédemment. Cette comparaison s'effectue en termes de nombre de messages échangés et de temps d'exécution requis pour l'accès aux données, dans le but d'identifier la solution la plus performante.
- Enfin, une conclusion vient synthétiser le travail réalisé, tout en mettant en lumière les perspectives et pistes de recherche future.

I.6. Publications académiques

Les travaux réalisés dans le cadre de cette thèse ont donné lieu aux publications académiques présentées ci-dessous :

- **Journal :** Amel MECHRI, Bilal BOUAITA, Djamel Eddine ZEGOUR, Walid-Khaled HIDOUCI, "*Design and Implementation of a ROLAP Cube in Scalable Distributed Data Structure*", Engineering, Technology & Applied Science Research Vol. 15, No. 1, 2025, 20279-20284 20279. DOI: https://doi.org/10.48084/etasr.9648.
- **Webinaire :** Amel MECHRI, Djamel Eddine ZEGOUR, Walid-Khaled HIDOUCI, "*An overview of Scalable Distributed Data Structures*", Séminaire sur les Sciences Exactes, Tlemcen le 09 Novembre 2024.

PARTIE I

ETUDES BIBLIOGRAPHIQUES

Chapitre II

Les entrepôts de données (Data warehouse)

II.1. Introduction

Auparavant, dans les systèmes d'information des entreprises, les données étaient stockées dans des bases de données (BD) bien structurées. Un système OLTP (Online Transaction Processing) proposait un ensemble d'outils pour manipuler ces bases de données et effectuer des opérations de consultation, de suppression ou de mise à jour, permettant ainsi d'extraire les informations nécessaires pour répondre aux besoins des utilisateurs.

Avec le développement des technologies de l'information et des réseaux informatiques, d'autres types de données ont émergé. L'information, autrefois uniquement structurée, est désormais également de nature semi-structurée ou non structurée, comme les fichiers XML, les pages Web, les fichiers plats ou les feuilles Excel, ….

De plus, il est devenu impératif, au sein d'une même entreprise de relier les différentes sources d'information quelques soient leur nature pour échanger leur données. Cependant, un défi majeur s'est posé : comment gérer cette volumétrie croissante de données hétérogènes en termes de structure, de format, de représentation et de langage d'interrogation ?

Par ailleurs, les entreprises ont développé de nouveaux besoins. Elles cherchent à mettre en place des techniques innovantes pour suivre l'évolution des données, prédire le comportement des clients et prendre des décisions stratégiques afin de mieux les satisfaire et d'accroître leur chiffre d'affaires.

C'est dans ce contexte, que sont nés les entrepôts de données et les techniques OLAP (Online Analytical Processing).

Un entrepôt de données constitue le cœur de l'informatique décisionnelle. Il intègre les données pertinentes et nécessaires à une activité spécifique de l'entreprise. Ces données sont extraites de diverses sources d'information hétérogènes, puis nettoyées, épurées, homogénéisées et mises à la disposition des analystes et décideurs.

OLAP, quant à lui, fournit un ensemble d'outils pour l'analyse et la manipulation des données contenues dans l'entrepôt de données, ainsi que pour l'exploration des relations qui existent entre ces données.

II.2. Qu'est ce qu'un entrepôt de données

Les entrepôts de données [31, 35] ou les data warehouse (DW) sont apparus dans les années 1990 [61]. Plusieurs définition on été données au concept entrepôt de données, parmi lesquelles :

- ☑ Un entrepôt de données peut être vu comme un lieu de stockage des données issues des différentes sources d'information d'une entreprise pour être utilisées par la suite par les utilisateurs avec des outils OLAP [26].
- ☑ Ou un entrepôt est un espace de stockage de données transactionnelles issues de sources de données variées [26]. Ces données hétérogènes extraites des différentes sources d'information sont réorganisées, homogénéisées et représentées dans un seul format en vu de leur utilisation par les gestionnaires [52] pour leur aider dans le processus de la prise de décision.
- ☑ Ou encore selon Bill Inmon, le fondateur de ces entrepôts : "un entrepôt de données est une collection de données orientées sujet (ou thématiques), intégrées, non volatiles et historisées (évolutives dans le temps) qui sert de support pour le processus d'aide à la décision" [23, 48, 60].

Les données d'un entrepôt de données se distinguent par les caractéristiques essentielles suivantes [25, 51]:

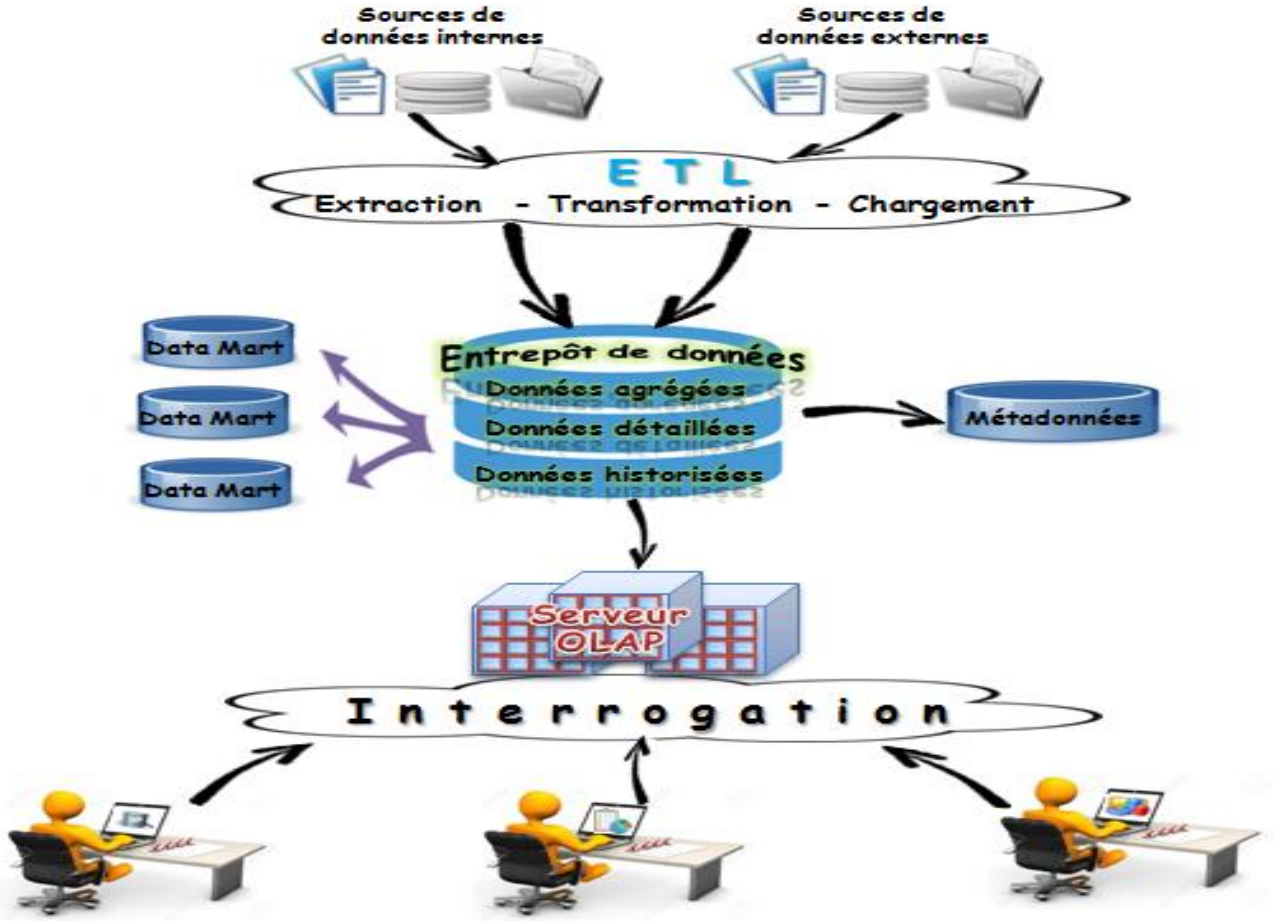
- ➲ ***Orientées sujet :*** Un entrepôt de données est constitué d'un ensemble de données relatives à un sujet d'entreprise contrairement aux données des systèmes de production qui sont organisées par des processus fonctionnels.
- ➲ ***Intégrées :*** Un entrepôt de données rassemble des données provenues d'un ensemble de sources d'information multiples et variées, qui peuvent être internes ou externes à l'entreprise. Ces données sont représentées dans différents formats. Avant leur intégration dans l'entrepôt de données, les données doivent être homogénéisées et unifiées (c'est-à-dire, une donnée doit avoir une description et un codage unique) pour avoir un état cohérent. Cette homogénéisation et mise en correspondance entre les données hétérogènes est une étape très importante et très complexe qui influe sur la qualité de l'information.
- ➲ ***Non volatiles :*** Les données de l'entrepôt de données sont stockées d'une manière permanente. Avec le temps, de nouvelles données peuvent être ajoutées mais les données déjà intégrées sont rarement modifiées ou supprimées [52]. La préservation des différentes valeurs d'une donnée permet de garder une traçabilité de longue durée pour aider à la prise de décision.
- ➲ ***Historisées :*** Les données sont non volatiles et peuvent être archivées. L'archivage consiste à garder plusieurs versions pour la même donnée depuis son entreposage, ce qui

permettra de prévoir l'évolution des données dans le temps et aidera à la prise de décision.

II.3. Construction et exploitation d'un entrepôt de données

La mise en place d'un système à base d'entrepôt de données passe par trois phases essentielles :

- ✓ Phase1 : Sélection et collection des données à partir des sources d'information.
- ✓ Phase2 : Intégration des données au sein de l'entrepôt de données.
- ✓ Phase3 : Manipulation et interrogation des données de l'entrepôt de données.



FigureII.1 : Construction et exploitation d'un ED

a) Phase1 : Sélection et collection des données à partir des sources d'information

Avant tout, il faut choisir les sources de données nécessaires à la mise en place de l'entrepôt de données. Les sources d'information sont nombreuses, variées, hétérogènes, distribuées et autonomes, peuvent être aussi bien internes à l'entreprise qu'externes.

b) Phase2 : Intégration des données au sein de l'entrepôt de données [47]

Avant leur chargement dans l'entrepôt de données, les données sont transformées et unifiées, grâce au processus d'extraction, transformation et chargement ETL (Extracting, Transforming and Loading) sans affecter les sources d'origine ou perturber le système source.

ETL (ou data pumping)

C'est un ensemble d'outils utilisés pour l'alimentation de l'entrepôt de données. Le processus ETL est plutôt un système complexe qui se déroule en plusieurs étapes : extraction, transfert, chargement et rafraîchissement.

☑ ***Extraction :*** Durant cette étape, il s'agit de définir, identifier, sélectionner et collecter les données nécessaires et relatives au sujet à analyser à partir des sources d'information.

☑ ***Transformation :*** Ce processus consiste à : trier, nettoyer [24], filtrer et homogénéiser les données hétérogènes qui sont issues des sources variées afin de les représenter dans un état fiable et cohérent.

La transformation consiste donc à représenter les données hétérogènes dans un format unique en résolvant les conflits d'intégration [26] ou les divergences qui existent entre ces données, tel que [12] :

- Utilisation des noms différents pour représenter la même donnée (*Exemple :* NomProduit et DesignationProduit).
- Utilisation des noms identiques pour représenter des données différentes (*Exemple :* Nom pour représenter le nom d'un produit et le nom d'un client).
- Utilisation de différentes unités de mesures pour quantifier la même donnée comme l'unité monétaire, le format de date, l'unité de mesure impériale ou système métrique ….
- Problème de valeurs manquantes [59] ou dupliquées, ……
- La représentation d'une information (*Exemple :* F et M, Masculin et Féminin ou 1 et 2 pour représenter le sexe d'une personne).
- ……

☑ ***Chargement et rafraîchissement :*** Après la mise en format et l'unification des différentes données, elles vont être stockées ou chargées dans l'entrepôt de données. Le rafraîchissement consiste à entrevoir périodiquement les données de l'entrepôt de données pour en insérer d'autres données ou de nouvelles valeurs de données déjà stockées dans l'entrepôt de données et qui ont été modifiées dans leur source d'origine.

Les Data marts

Les data marts ou magasins de données sont des sous ensembles de données [48] extraites à partir de l'entrepôt de données pour être manipulées par certains utilisateurs [26]. Elles sont spécialisées et liées à un sujet particulier de l'entreprise.

Par exemple si l'entrepôt de données concerne l'activité "Vente", un data mart contient les informations concernant un magasin

La mise en œuvre d'un data mart est une tâche moins compliquée et nécessite peu de temps. Par rapport aux entrepôts, les data marts ont une structure plus légère, un temps de réponse plus court [12] ce qui a pour avantage d'accroître la performance du système. Cependant, la multiplication des data marts entraîne une complexification de la gestion des données.

c) Manipulation et interrogation de l'entrepôt de données

Après le chargement de l'entrepôt de données par les données nécessaires, un utilisateur peut accéder et manipuler son contenu en utilisant des outils d'analyse tel que : outils de fouille de données (ou data mining), tableurs, outils d'analyse OLAP, outils de reporting …..

II.4. Structure d'un entrepôt de données

Les données d'un entrepôt de données peuvent être classées selon deux axes [23, 47, 59] : synthétique et historique.

a. Axe synthétique : Cet axe représente la hiérarchie des données [26] : détaillées, fortement ou faiblement agrégées.

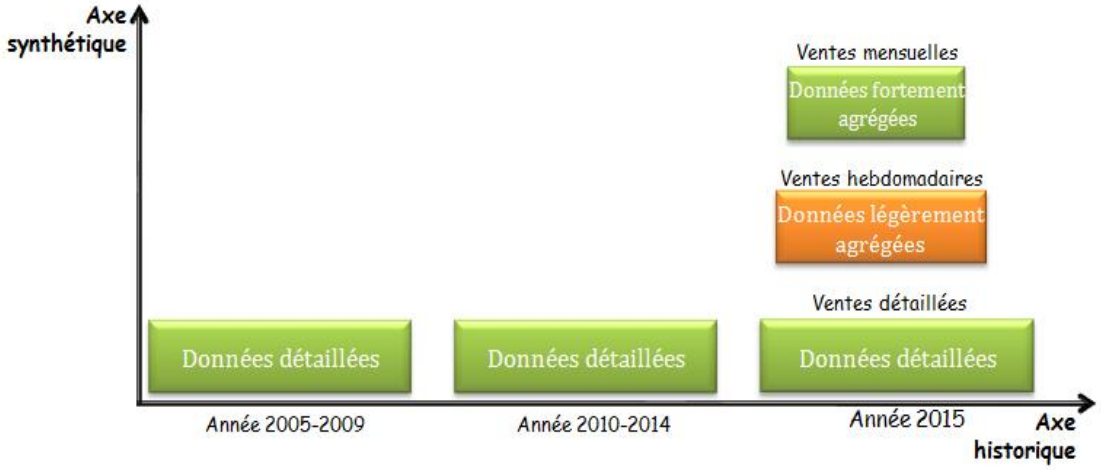
- ***Données détaillées :*** Ce sont les données les plus récentes et les plus fréquemment consultées. L'ajout d'une nouvelle donnée se fait généralement à ce niveau. Elles sont volumineuses et stockées sur le disque pour avoir un accès rapide.
- ***Les données agrégées (ou résumées) :*** Elles sont la synthèse de données détaillées ce qui permet de réduire le volume des données et l'espace disque nécessaire pour leur stockage. Elles peuvent être légèrement ou fortement agrégées. Les données fortement agrégées synthétisent les données agrégées.

b. Axe historique : Il représente l'archivage de données.

- ***Données historisées :*** Ce sont les données détaillées conservées au fil du temps et qui concernent des événements passés. Elles sont rarement sollicitées et donc elles sont stockées sur des mémoires d'archives.

➲ ***Les métadonnées :*** Une métadonnée est une donnée qui décrit une autre donnée. Dans un entrepôt de données, les métadonnées regroupent un ensemble d'information essentielle et nécessaire pour la gestion et l'exploitation efficace de son contenu, tel que : la sémantique d'une donnée (signification), son origine (sa source, le programme l'ayant créée et modifiée, …), les règles de transformation et d'agrégation, le stockage (format ...), date de chargement, les programmes susceptibles de manipuler la donnée, ….

Les métadonnées sont contenues dans un référentiel partagé par toute personne utilisant l'entrepôt de données (concepteur, analystes, administrateurs, …).



FigureII.2 : Types de données dans un entrepôt

II.5. Modélisation multidimensionnelle [61]

La représentation des données d'un entrepôt de données et des traitements OLAP a impliqué la conception d'un nouveau modèle de données multidimensionnel avec la définition de nouveaux concepts: Fait, dimension, mesure, hiérarchie et cube.

La modélisation multidimensionnelle considère le sujet à analyser comme un point dans un espace à plusieurs dimensions [59]. Les données sont organisées de manière à mettre en évidence le sujet (qui est le fait) et les différentes perspectives d'analyse (dimensions).

a. Fait :

Le fait modélise le sujet d'analyse [23], il est formé d'indicateurs d'analyse appelés "mesures" [47] ou "indicateurs" qui représentent les différentes valeurs de l'activité analysée.

Les mesures sont généralement des valeurs numériques calculées à l'aide d'une fonction d'agrégation (sum, average, max, min, count…). Elles peuvent être additives (peuvent être calculées selon toutes les dimensions), semi-additives (peuvent être calculées selon certaines dimensions) ou non additives [47, 61].

Le fait est représenté par une table de fait.

b. Dimension :

Une dimension modélise une perspective ou un axe d'analyse d'un fait [23]. Une dimension représente un point de vue depuis lequel, les mesures peuvent être observées [62]. Elle se compose de paramètres (ou attributs) qui sont des descriptions textuelles. Les paramètres sont organisés d'une manière hiérarchique qui modélise les différents niveaux de granularité (ou de détail) des axes d'analyse.

Chaque dimension est représentée par une table.

c. Table de fait

C'est la table principale du modèle multidimensionnel, elle représente le sujet étudié selon divers axes d'analyse (dimensions) [60]. Elle contient des clés vers les tables de dimension [48].

d. Tables de dimension

Se sont les tables complémentaires à la conception de la table de fait. Elles contiennent des attributs sous forme de descriptions textuelles. Chaque table de dimension représente un axe d'analyse selon lequel vont être étudié les données observables (faits) [60].

La table de fait est plus volumineuse (elle contient de grands nombres de lignes) par rapport aux tables de dimension, alors que ces dernières contiennent plus de champs (plus de colonnes).

Exemple :

Table de fait : Vente

Vente
Référence
IdDate
IdPointVente
QteVendue
Montant

Table de dimension :

Date
IdDate
Jour
Mois
Trimestre
Année

FigureII.3 : Exemple d'une table de fait et de dimension

e. Le cube de données

Les tables ne conviennent pas à la représentation des données multidimensionnelles d'un entrepôt de données ; on privilégie plutôt l'utilisation de cubes. Un cube permet d'explorer et d'analyser les données sous plusieurs dimensions. Lorsque ces dimensions dépassent le nombre de trois, on parle alors d'hypercube.

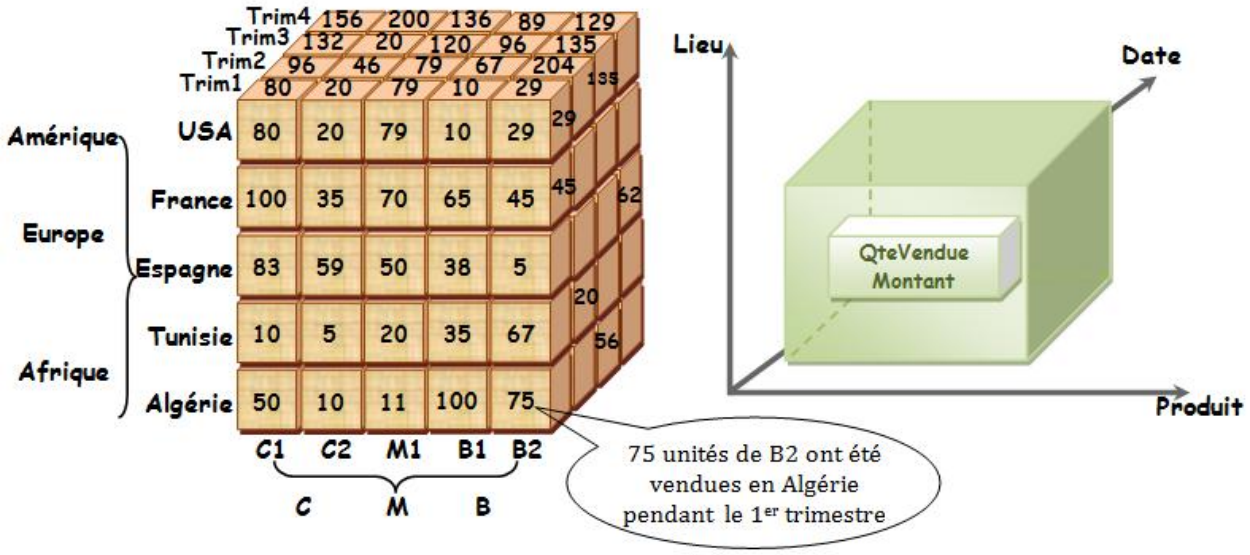
Un cube est constitué de cellules, chacune pouvant renfermer une ou plusieurs mesures. Chaque cellule est localisée par rapport aux axes de cube qui correspondent aux différentes dimensions. Une dimension est composée de membres représentant les différentes valeurs.

Exemple :

On veut modéliser les ventes au sein d'une filiale de construction d'automobile.

Considérons le fait "***Vente***", il se compose des mesures : *QteVendue*, *Montant*. Ce fait sera exploré selon trois dimensions : ***Lieu***, ***Date*** (on représente les quatre trimestres) et ***Produit*** (C pour le type Citadine, M pour Monospace et B pour Berline).

Dans le cube suivant, seule la mesure « *QteVendue* » est représentée.



FigureII.4 : Représentation d'une mesure dans un cube

La dimension ***Lieu*** peut être représentée selon deux hiérarchies : par pays, ou par continent. On peut également ajouter une autre hiérarchie concernant la représentation des ventes par ville.

De même pour la dimension ***Produit*** ou ***Date***, on peut ajouter d'autres niveaux hiérarchiques.

"Un utilisateur choisit les dimensions selon lesquelles il veut analyser les faits et choisit pour chaque dimension le niveau hiérarchique sur lequel il veut travailler".

Dans la modélisation multidimensionnelle, on ajoute souvent la dimension "T***emps***" pour historiser l'évolution de données, d'où l'accroissement du volume de l'entrepôt de données.

II.6. OLAP (On-Line Analytical Processing)

Le concept OLAP, connu dans l'informatique décisionnelle, désigne un ensemble de moyens et techniques utilisés pour l'exploration, l'analyse des données multidimensionnelles et les relations entre ces données pour aider l'usager dans son processus de prise de décision.

II.6.1. Serveurs OLAP

Il existe trois types de serveurs [51] pour l'implémentation et l'exploitation d'un entrepôt de données :

a) Le serveur ROLAP (Relational OLAP)

b) Le serveur MOLAP (Multidimensional OLAP)

c) Le serveur HOLAP (Hybrid OLAP)

II.6.1.1. Le serveur ROLAP

Ce type de serveur utilise une BD relationnelle pour le stockage et la gestion des données. Les requêtes multidimensionnelles vont être traduites en requêtes relationnelles [24] et le modèle logique de données multidimensionnel sera traduit en un modèle de stockage relationnel.

Un fait est représenté par la table de fait, une dimension par une table de dimension. La table de fait est représentée par des colonnes qui sont les mesures de l'activité, et les clés étrangères référençant les tables de dimension servant de jointure entre ces deux types de tables.

La technologie ROLAP présente deux avantages :

- Le nombre de jointures nécessaires à l'exécution d'une requête est réduit [62].
- Utilisation d'un modèle (relationnel) qui a déjà prouvé son efficacité et qui est relativement simple pour représenter des données multidimensionnelles et complexes.

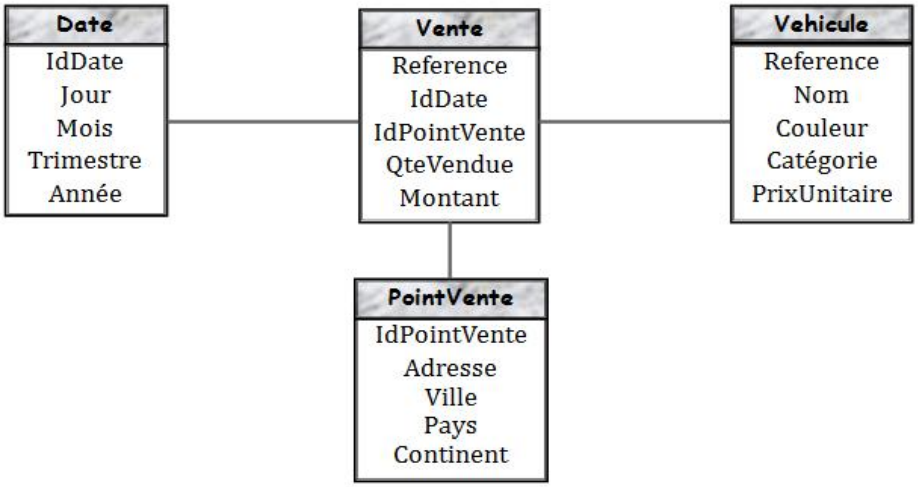
En l'occurrence, Le langage de requête n'est pas assez puissant et flexible pour traiter des données multidimensionnelles, il requiert des extensions pour supporter les requêtes multidimensionnelles.

Trois modèles sont utilisés pour la représentation logique des données : la représentation en étoile, en flocon et en constellation [12, 51].

i. Modèle en étoile :

Le modèle en étoile se compose d'un ensemble de tables de dimension reliées à la table de fait centrale [53]. Les tables de dimension ne sont pas reliées entre eux. La table de fait regroupe les différentes mesures et les clés étrangères référençant les tables de dimension. La clé primaire de la table de fait est formée par la concaténation des clés étrangères.

Exemple :



FigureII.5 : Modèle en étoile

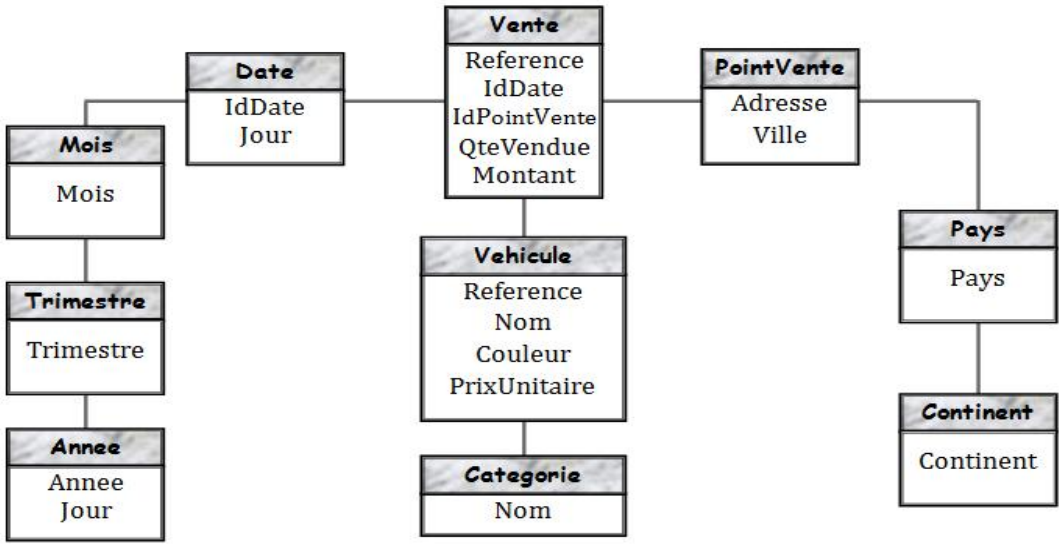
- Dans ce modèle, la structure est simple, directe et facilement compréhensible.
- Le temps de réponse à une requête est réduit grâce au nombre restreint de jointures.
- En outre, ce modèle ne permet pas de représenter les hiérarchies des dimensions. Toutes les données quelques soient leurs niveaux hiérarchiques sont regroupées dans la même table.
- il y a une redondance de données due aux relations dénormalisées.

ii. Modèle en flocon :

Ce modèle repose sur le même principe du modèle précédent, il se compose d'une table de fait et des tables de dimensions qui sont éclatées (ou décomposées) en sous hiérarchie et les relations sont normalisées [62].

Une table de dimension représente un seul niveau de la hiérarchie, la table de niveau le plus bas (de plus fine granularité) est reliée à la table de fait.

Exemple :



FigureII.6 : Modèle en flocon

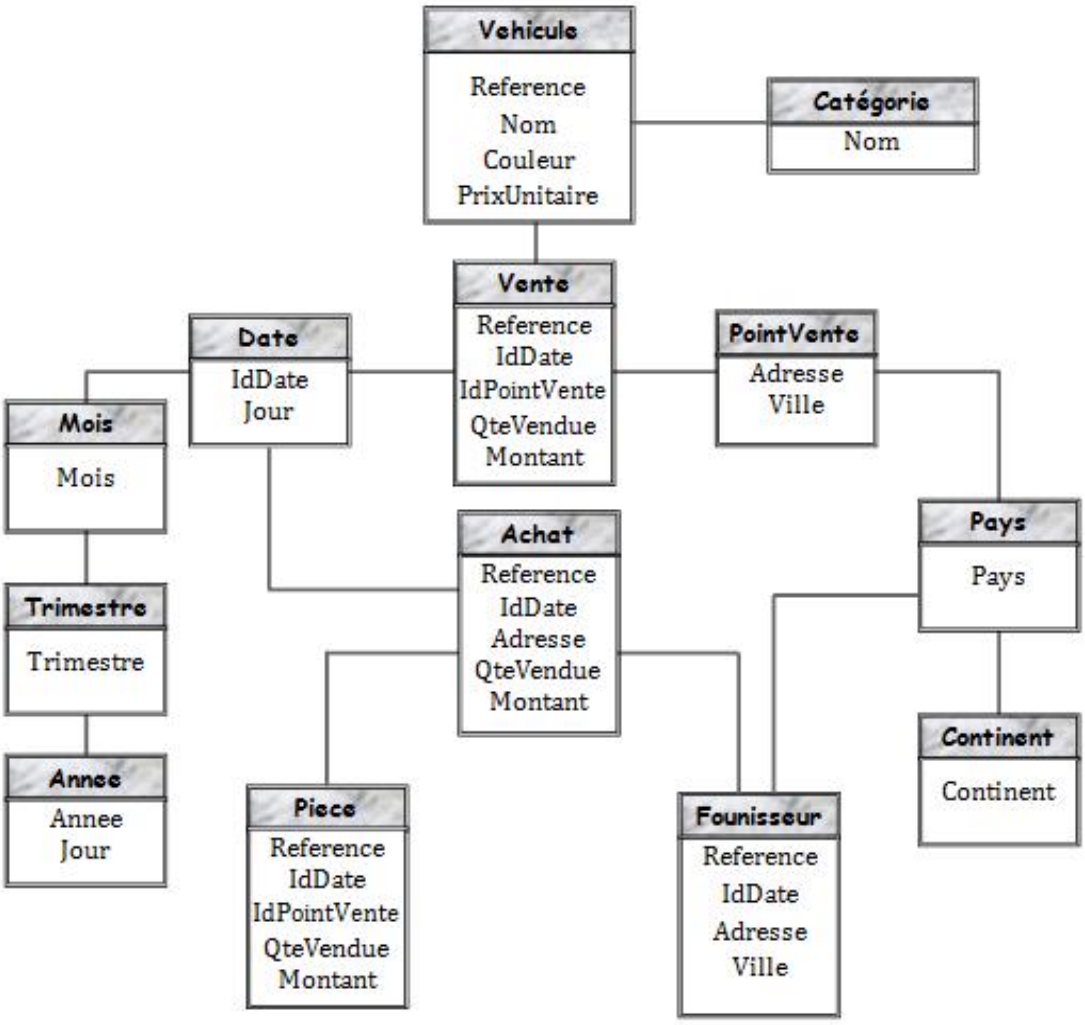
- La description des hiérarchies d'une dimension offre une meilleure analyse du fait.
- La normalisation des relations [60] et la non redondance de données réduit leur taille et l'espace mémoire nécessaire à leur stockage.
- Néanmoins, ce modèle offre une structure plus complexe [60], peu lisible et nécessite un nombre plus élevé de jointures [62] pour l'exécution d'une requête.

iii. Modèle en constellation :

Dans une constellation, on dispose de plusieurs modèles dimensionnels qui partagent les mêmes dimensions, c'st-à-dire, les tables des faits ont des tables de dimensions en commun.

Le modèle en constellation fusionne plusieurs modèles en étoile qui utilisent des dimensions communes [48, 59].

Exemple :



FigureII.7 : Modèle en constellation

II.6.1.2. Le serveur MOLAP

Dans un serveur MOLAP, les données sont stockées dans une BD multidimensionnelle. Les données sont représentées sous forme d'un tableau de N dimensions, où chaque colonne est associée à une dimension de l'hypercube de données. Les requêtes sont très puissantes et flexibles du fait qu'il y a un accès direct aux données agrégées et déjà calculées et contenues dans les cellules du cube.

Le problème dans ce modèle est qu'il n'existe pas un modèle physique standard.

II.6.1.3. Le serveur HOLAP (Hybrid OLAP)

C'est une approche hybride qui supporte le stockage de données en relationnel et en multidimensionnel. En effet, les données agrégées et qui sont fréquemment utilisées sont stockées dans des cubes pré-calculés alors qu'une grande quantité de données, les moins utilisées, sont stockées dans des tables multidimensionnelles.

Actuellement, la plupart des systèmes utilisent une approche hybride, l'approche ROLAP pour la manipulation de l'entrepôt de données, et l'approche multidimensionnelle pour la gestion des data marts.

II.6.2. Opérations OLAP [29, 33, 61]

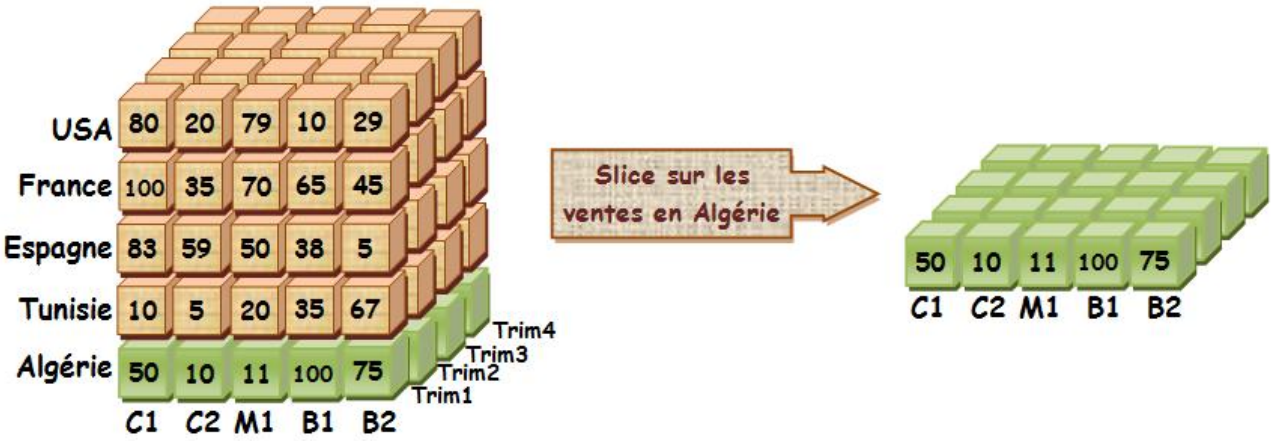
Pour exploiter et manipuler les données d'un cube, nous devons le visualiser selon différentes facettes et formes (représentation bidimensionnelle ou tridimensionnelle avec changement de hiérarchies, de structure, …). Pour ce faire, un ensemble d'opérateurs, répartis en trois catégories, est à disposition :

II.6.2.1. Opérateurs de sélection [47]

Ces opérateurs sont utilisés pour restreindre l'ensemble de données à analyser et ne sélectionner que celles répondant à des critères spécifiques. On distingue l'opérateur de sélection "Slice" et l'opérateur de projection "Dice".

- **Slice (couper) :** L'opération **Slicing** consiste à extraire une tranche spécifique du cube, c'est-à-dire sélectionner un sous-ensemble des données en fixant une dimension à une valeur spécifique.

Exemple :



FigureII.8 : Application de l'opérateur « Slice » sur un cube

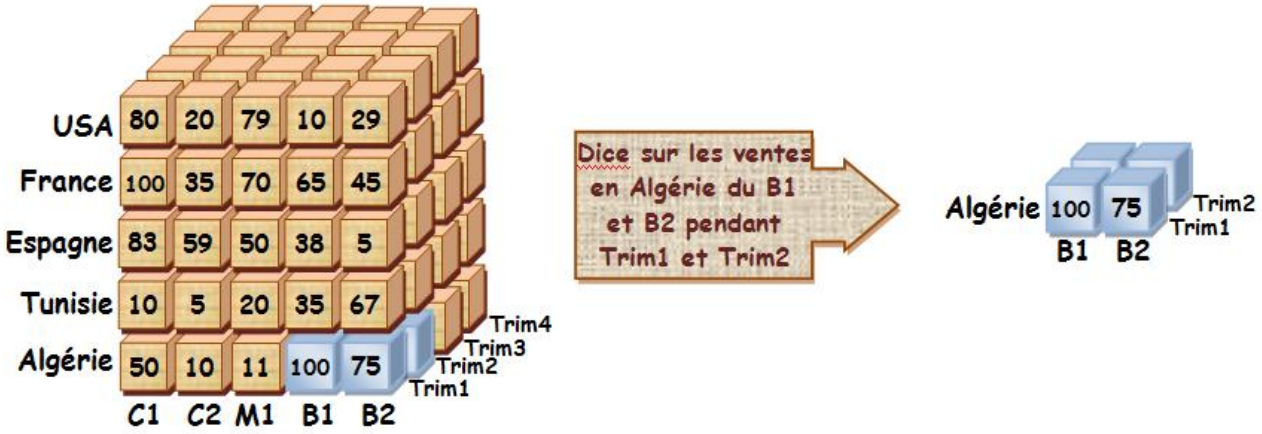
Ici, on "coupe" le cube en sélectionnant les ventes de véhicules pour l'Algérie. La requête correspondante est la suivante :

```
SELECT V.Nom, Vt.QteVendue, PV.Pays
FROM Vente Vt
JOIN Vehicules V ON Vt.Reference = V.Reference
JOIN PointVente PV ON Vt.IdPointVente = PV.IdPointVente
WHERE PV.Pays = 'Algérie';
```

FigureII.9 : Requête démontrant l'opérateur « Slice »

- **Dice (découper) :** Cet opérateur a le même rôle que l'opérateur slice, mais la condition peut être appliquée sur des attributs de N axes d'analyse (deux ou plus), ce qui créera un sous-cube.

Exemple : Afficher les ventes en Algérie pour les véhicules "B1" et "B2".



FigureII.10 : Application de l'opérateur « Dice » sur un cube

Ici, on découpe le cube en sélectionnant les données de ventes pour des véhicules spécifiques dans un pays spécifique.

```
SELECT V.Nom, Vt.QteVendue, PV.Pays
FROM Vente Vt
JOIN Vehicules V ON Vt.Reference = V.Reference
JOIN PointVente PV ON Vt.IdPointVente = PV.IdPointVente
WHERE V.Nom IN ('B1', 'B2') AND PV.Pays = 'Algérie';
```

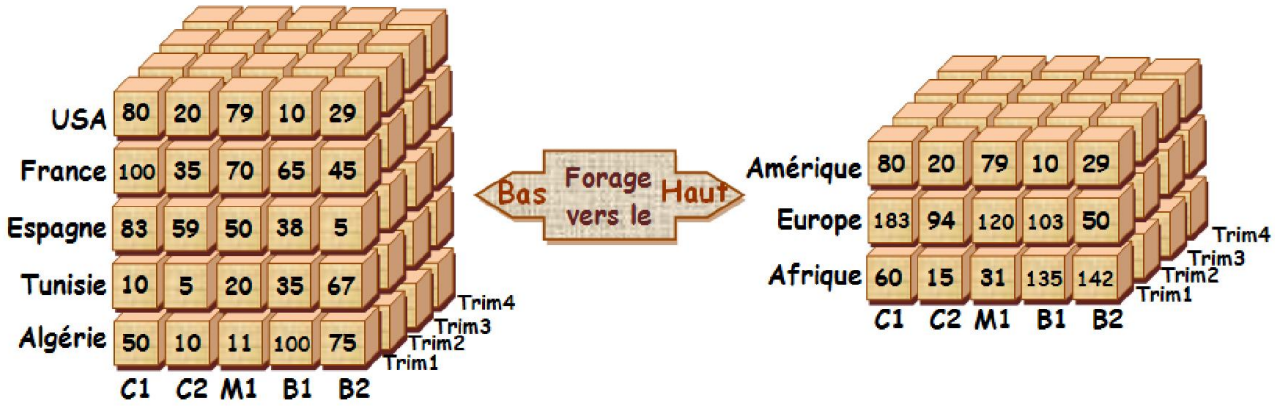
FigureII.11 : Requête démontrant l'opérateur « Dice »

II.6.2.2. Opérateurs de forage

Cette famille d'opérateurs agit sur la granularité [60, 62], ils permettent la navigation entre les différents niveaux de la hiérarchie des axes [27].

- **Roll up ou Drill up (Aggrégation vers le haut) :** On utilise cet opérateur de forage vers le haut pour permettre d'analyser les données en passant d'un niveau de hiérarchie à un autre moins détaillé ce qui donnera une vision plus globale. Des fonctions d'agrégation sont utilisées pour calculer les données du niveau global supérieur.
- **Drill down, Roll down ou Scale down (Aggrégation vers le bas):** Contrairement au forage vers le haut, avec l'opérateur Roll up, le forage vers le bas avec l'opérateur Drill down permet de descendre dans la hiérarchie. Il est possible donc de passer d'un niveau de granularité à un autre plus détaillé.

Exemple :



FigureII.12 : Application des opérateurs de forage sur un cube

La requête suivante permet d'afficher les ventes enregistrées par continent, offrant ainsi un niveau de détail supérieur à celui des ventes par pays:

```
SELECT  PV.Continent, V.Nom, SUM(Vt.QteVendue)
FROM  Vente  Vt
JOIN PointVente PV ON PV.IdPointVente = Vt.IdPointVente
JOIN Vehicules V ON Vt.Reference = V.Reference
GROUP BY PV.Continent, V.Nom WITH ROLLUP;
```

FigureII.13 : Requête démontrant l'opérateur « Roll up »

Pour passer à un niveau de détail inférieur et afficher les ventes enregistrées par pays plutôt que par continent, on applique la requête suivante:

```
SELECT PV.Pays, V.Nom, SUM(Vt.QteVendue)
FROM Ventes Vt
JOIN PointVente PV ON PV.IdPointVente = Vt.IdPointVente
JOIN Vehicules V ON Vt.Reference = V.Reference
```

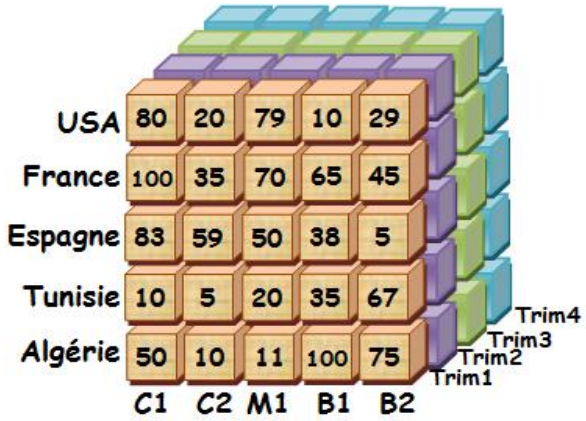
FigureII.14 : Requête démontrant l'opérateur « Roll down »

II.6.2.3. Opérateurs sur la structure [27]

Ces opérateurs peuvent être appliqués sur la structure d'un cube afin d'en offrir une représentation et une visualisation sous un angle différent.

- **Split (séparation) :** Cet opérateur de division offre la possibilité de réduire le nombre des axes (dimensions) et de passer d'une représentation tridimensionnelle d'un cube à une représentation tabulaire (deux dimensions).

Exemple : Un split sur la dimension "Date" va donner quatre tableaux représentant les ventes de chaque trimestre. On va donc représenter chaque tranche du cube par un tableau.



Trim1	C1	C2	M1	B1	B2
USA	80	20	79	10	29
France	100	35	70	65	45
Espagne	83	59	50	38	5
Tunisie	10	5	20	35	67
Algérie	50	10	11	100	75

FigureII.15 : Application de l'opérateur « Split » sur un cube

La requête exprimant cette opération de séparation peut être formulée ainsi :

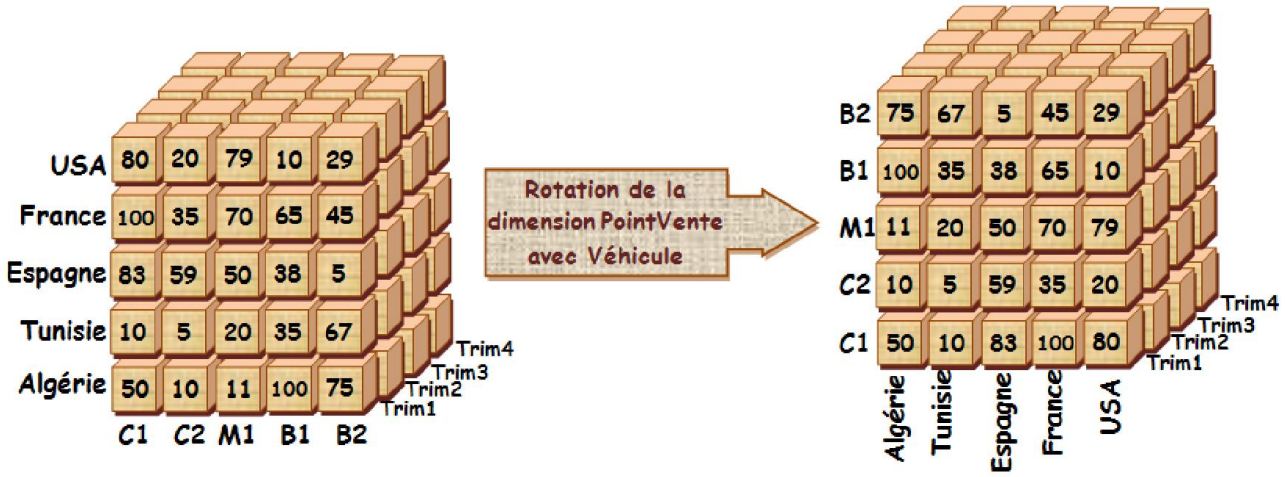
```
SELECT D.Trimestre, PV.Pays, V.Nom
FROM Ventes Vt
JOIN PointVente PV ON PV.IdPointVente = Vt.IdPointVente
JOIN Vehicules V ON Vt.Reference = V.Reference
JOIN Date D ON Vt.IdDate = D.IdDate
GROUP BY D.Trimestre, PV.Pays, V.Nom
ORDER BY D.Trimestre;
```

FigureII.16 : Requête démontrant l'opérateur « Split »

- **Pivot ou Rotate :** Ces opérateurs sont utilisés pour faire pivoter ou tourner le cube pour voir les données sous différents angles. Il existe trois types de rotation :

- **Rotation de dimension :** En échangeant un axe d'analyse par un autre, cette rotation permet de visualiser le cube selon une autre face sans modification des mesures.
- **Rotation de hiérarchie :** Changer les attributs de l'axe d'analyse par d'autres attributs appartenant à une autre hiérarchie du même axe.
- **Rotation de fait :** Pour changer le sujet en cours avec un autre sujet tout en gardant les mêmes dimensions et hiérarchies du cube. Il faut toutefois que les dimensions (du sujet initial et de rotation) soient compatibles.

Exemple : Rotation entre les dimensions Pays et Véhicule.



FigureII.17 : Application de l'opérateur de rotation de dimension sur un cube

En appliquant la requête SQL suivante, on passe d'une vue Vehicule / Pays /Trimestre à une vue Pays / Vehicule / Trimestre :

```
SELECT V.Nom, PV.Pays, D.Trimestre, Vt.QteVendue
FROM Ventes Vt
JOIN PointVente PV ON PV.IdPointVente = Vt.IdPointVente
JOIN Vehicules V ON Vt.Reference = V.Reference
JOIN Date D ON Vt.IdDate = D.IdDate
GROUP BY D.Trimestre, V.Nom, PV.Pays
```

FigureII.18 : Requête démontrant l'opérateur « Rotate »

Cela permet de "faire pivoter" les axes en affichant les pays en première position.

- **Nest (emboîtement) :** Cet opérateur d'emboitement permet d'inclure (d'imbriquer) les attributs d'un axe (qui n'est pas affiché) en utilisant une représentation bidimensionnelle. En d'autres termes, **Nest** consiste à imbriquer des requêtes SQL.
- L'opérateur "**Unnest**" reconstitue une dimension séparée à partir des membres imbriqués.

Exemple : En s'appuyant sur le même cube, le Nest des dimensions "PointVente" et "Date " produira la représentation suivante :

Vente		C1	C2	B1	M1	M2
Trim1	USA	80	20	79	10	29
	France	100	35	70	65	45
	Espagne	83	59	50	38	5
	Tunisie	10	5	20	35	67
	Algérie	50	10	11	100	75
Trim2	USA	.....	.....	.....	.....	.....
.....						

TableauII.1 : Application de l'opérateur « Nest » sur un cube

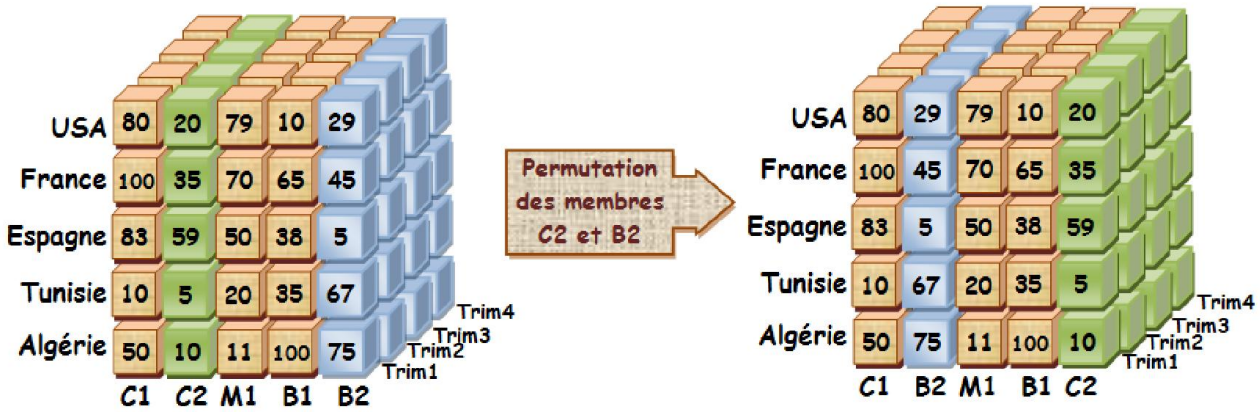
Ici, on imbrique une sous-requête qui sélectionne les ventes totales par Pays, puis on sélectionne le Trimestre correspondant.

```
SELECT D.Trimestre, PV.Pays,
  COALESCE(SUM(CASE WHEN Nom = 'C1' THEN Quantite ELSE 0 END), 0) AS C1,
  COALESCE(SUM(CASE WHEN Nom = 'C2' THEN Quantite ELSE 0 END), 0) AS C2,
  COALESCE(SUM(CASE WHEN Nom = 'B1' THEN Quantite ELSE 0 END), 0) AS B1,
  COALESCE(SUM(CASE WHEN Nom = 'M1' THEN Quantite ELSE 0 END), 0) AS M1,
  COALESCE(SUM(CASE WHEN Nom = 'M2' THEN Quantite ELSE 0 END), 0) AS M2
FROM
   (SELECT D.trimestre, PV.Pays, Vt.Quantite, V.Nom
    FROM Vente Vt
    JOIN Date D ON Vt.IdDate = D.IdDate
    JOIN PointVente PV ON Vt.IdPointVente = PV.IdPointVente
    JOIN Vehicule V ON Vt.Reference = V.Reference )
AS SousRequete
GROUP BY D.Trimestre, PV.Pays
```

FigureII.19 : Requête démontrant l'opérateur « Nest »

COALESCE() : Cette fonction est utilisé pour remplacer toute valeur NULL de la QteVendue par un **0**.

- **Switch (Changement de dimension) :** L'opérateur de permutation **Switch** permet l'inversement des membres d'une dimension, en quelque sorte il s'agit de la permutation de deux tranches du cube.



FigureII.20 : Application de l'opérateur « Switch » sur un cube

La requête suivante permet d'échanger les membres de dimension Véhicule, permettant une permutation entre les véhicules de type C2 avec celles de type B2.

```
SELECT D.Trimestre, PV.Pays,
CASE
   WHEN V.Nom = 'C2' THEN 'B2' -- Pour changer B2 et C2
   WHEN V.Nom = 'B2' THEN 'C2'
   ELSE V.Nom -- Pour les autres véhicules, on garde leur original nom
END
AS Nom_Vehicule,
QteVendue
FROM Vente Vt
JOIN Date D ON Vt.IdDate = D.IdDate
JOIN PointVente PV ON Vt. IdPointVente = PV.IdPointVente
JOIN Vehicule V ON Vt.Reference = V.Reference
WHERE V.Nom IN ('C2', 'B2')
```

FigureII.21 : Requête démontrant l'opérateur « Switch »

- **Push (Propagation des calculs) :** L'enfoncement avec cet opérateur permet de convertir les attributs d'un axe en mesures (faire passer les membres comme contenus de cellules). Contrairement à "**Pull**" (opérateur de retrait) qui fait passer les mesures en paramètres.

Exemple : L'application de l'opérateur **Push** sur le cube de données donnera la représentation suivante :

Vente	C1	C2	B1	M1	M2
Trim1	USA : 80,	USA : 20,	USA : 79,	USA : 10,	USA : 29,
	France : 100,	France : 35,	France : 70,	France : 65,	France : 45,
	Espagne 83,	Espagne : 59,	Espagne : 50,	Espagne : 38,	Espagne : 5,
	Tunisie 10,	Tunisie : 5,	Tunisie : 20,	Tunisie : 35,	Tunisie : 67,
	Algérie 50,	Algérie : 10,	Algérie : 11,	Algérie : 100,	Algérie : 75,
Trim2	.....	.....	.....	.....	.....

TableauII.2 : Application de l'opérateur « Push » sur un cube

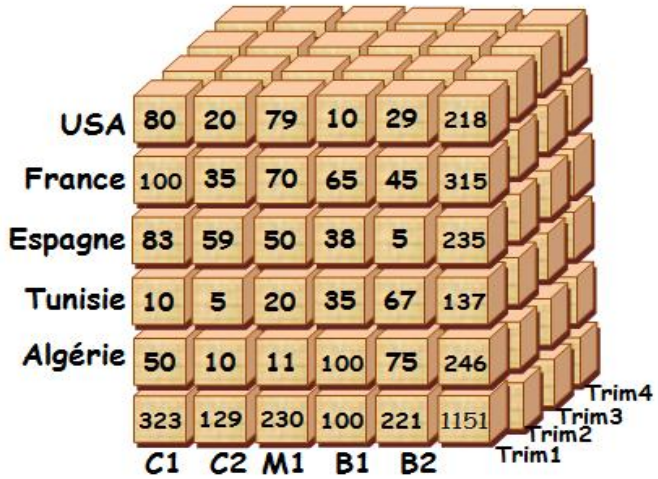
La requête en dessous permet de convertir les attributs de la dimension PointVente en des mesures concaténées avec les quantités vendues tel que indiqué dans le tableau :

```
SELECT D.trimestre, V.Nom
GROUP_CONCAT(CONCAT(PV.Pays, ':',SUM(Vt.Quantite)) SEPARATOR ',')
FROM Vente Vt
JOIN Date D ON Vt.IdDate = D.IdDate
JOIN PointVente PV ON Vt.IdPointVente = PV.IdPointVente
JOIN Vehicule V ON Vt.Reference = V.Reference
GROUP BY D.trimestre, V.Nom
```

FigureII.22 : Requête démontrant l'opérateur « Push »

- **GROUP_CONCAT()** : Cette fonction permet de concaténer plusieurs lignes en une seule.
- **CONCAT(PV.Pays, ': ', SUM(F.Quantite))** : Pour concaténer le pays avec la somme des quantités, séparés par un deux-points, pour chaque groupe (trimestre et véhicule).
- **SEPARATOR ','** : Ce séparateur permet de séparer les pays et quantités par une virgule.

- **Cube (Cube multidimensionnel)** [47, 61] **:** Permet de calculer des sous-totaux et un total final dans le cube.



FigureII.23 : Application de l'opérateur « Cube » sur un cube

L'opération **Cube**, comme indiquée dans la requête suivante permet d'agréger les données sur toutes les dimensions en une seule requête, créant ainsi des sous-totaux pour chaque combinaison de dimensions :

```
SELECT
   COALESCE(D.Trimestre, ' '),
   COALESCE(PV.Pays, ' '),
   COALESCE(V.Nom, ' '), SUM(Vt.QteVendue)
FROM Vente Vt
JOIN Date D ON Vt.IdDate = D.IdDate
JOIN PointVente PV ON Vt. IdPointVente = PV. IdPointVente
JOIN Vehicule V ON Vt.Reference = V.Reference
GROUP BY CUBE(D.Trimestre, PV.Pays, V.Nom)
HAVING D.Trimestre IS NULL OR PV.Pays IS NULL OR V.Nom IS NULL
```

FigureII.24 : Requête démontrant l'opérateur « Cube »

Tous les exemples cités en dessus, représentent les opérations OLAP les plus couramment utilisées pour explorer, manipuler et analyser les cubes ROLAP permettant de naviguer à travers différentes perspectives des données, offrant ainsi une flexibilité optimale pour l'analyse des ventes sous divers angles et niveaux de détail.

Il existe également d'autres opérateurs moins utilisés comme : AddM, DetM, Aggregate, Unfold et Fold.

- **Fold** : La factualisation consiste à transformer les paramètres d'une dimension en mesure.
- **Unfold** : La paramétrisation permet de transformer une mesure en paramètre dans une nouvelle dimension.
- **AddM** : Pour ajouter une mesure.
- **DetM** : Pour calculer le déterminant de la matrice.
- **Aggregate** : Pour regrouper résumer les données à différents niveaux de granularité.

II.7. Distribution d'un entrepôt de données

Dans un monde de plus en plus numérisé, les systèmes traditionnels de gestion des entrepôts de données centralisés se sont retrouvés confrontés à de nouveaux challenges face à l'explosion du volume des données. C'est alors que les entrepôts de données distribués se sont imposés comme une solution essentielle, permettant non seulement de stocker et gérer ces données à grande échelle, mais aussi de maintenir des performances optimales lors des processus de requêtage et d'analyse, afin de prendre des décisions stratégiques.

Il convient alors de définir un entrepôt de données distribué comme une architecture dans laquelle les données sont divisées et stockées sur plusieurs serveurs ou nœuds, assurant ainsi une scalabilité et une disponibilité élevée.

Cette approche de distribution des données s'articule autour de deux étapes principales : la fragmentation et la répartition. La fragmentation consiste à diviser les données en fragments, tandis que la répartition assure leur distribution sur différents serveurs. Ce qui permettra d'assurer une gestion optimale des ressources et un accès plus rapide aux données.

II.7.1. Fragmentation d'un entrepôt de données

La fragmentation consiste à décomposer un ED en plusieurs fragments de telle façon que la combinaison de ces fragments produit l'intégralité des données sources, sans perte [58] ou ajout d'information.

Le but de cette opération de fragmentation est de réduire le temps d'exécution des requêtes. En effet, il s'agit d'éviter des requêtes complexes en les décomposant en sous requêtes où chacune est exécutée par un serveur.

La fragmentation d'un entrepôt peut être verticale, horizontale primaire ou dérivée, ou bien hybride [19, 20, 49, 50].

II.7.1.1. Fragmentation verticale

Dans la fragmentation verticale, l'entrepôt va être décomposé en des fragments verticaux où chaque fragment regroupe un sous ensemble d'attributs de la relation plus la clé primaire. La clé primaire est nécessaire pour pouvoir reconstituer la relation initiale par jointure.

La répartition des attributs de la relation se fait grâce à une opération de projection [19, 58].

Exemple : Soit la table de dimension "PointVente" qui contient les enregistrements suivants :

IdPointVente	Ville	Pays	Continent
Cl1	Sétif	Algérie	Afrique
Cl2	Madrid	Espagne	Europe
Cl3	Alger	Algérie	Afrique
Cl4	Tokyo	Japon	Asie
Cl5	Barcelone	Espagne	Europe

IdPointVente	Ville
Cl1	Sétif
Cl2	Madrid
Cl3	Alger
Cl4	Tokyo
Cl5	Barcelone

Sous-table1

La table globale est fragmentée verticalement par projection en deux sous-tables

IdPointVente	Pays	Continent
Cl1	Algérie	Afrique
Cl2	Espagne	Europe
Cl3	Algérie	Afrique
Cl4	Japon	Asie
Cl5	Espagne	Europe

Sous-table2

Sous-table1 = Π [IdPointVente, Ville] PointVente ;
Sous-table2 = Π [IdPointVente, Pays, Continent] PointVente

FigureII.25 : Fragmentation verticale

Le principal inconvénient de la fragmentation verticale réside dans la nécessité d'effectuer des jointures supplémentaires lorsqu'une requête implique plusieurs fragments.

II.7.1.2. Fragmentation horizontale

Il existe deux types de fragmentation horizontale : primaire et dérivée [49, 50].

II.7.1.2.1. Fragmentation horizontale primaire

Ce type consiste à fragmenter un entrepôt en plusieurs lignes appelés fragments horizontaux en appliquant une opération de restriction grâce à des prédicats de sélection définis sur la relation.

Une opération d'union permet de réaliser l'opération inverse, c'est-à-dire, reconstituer la relation initiale partir de ces fragments horizontaux.

Exemple : La fragmentation horizontale de la table de dimension "PointVente", effectuée sur la base d'une restriction par continent, donnera les sous-tables suivantes :

IdPointVente	Ville	Pays	Continent
Cl1	Sétif	Algérie	Afrique
Cl3	Alger	Algérie	Afrique
Sous-Table1 = σ[Continent = "Afrique"]			

IdPointVente	Ville	Pays	Continent
Cl2	Madrid	Espagne	Europe
Cl5	Barcelone	Espagne	Europe
Sous-Table2 = σ[Continent = "Europe"]			

IdPointVente	Ville	Pays	Continent
Cl4	Tokyo	Japon	Asie
Sous-Table3 = σ[Continent = "Asie"]			

FigureII.26 : Fragmentation horizontale primaire

II.7.1.2.2. Fragmentation horizontale dérivée

Fragmentation horizontale dérivée consiste à fragmenter une table T1 qui a une relation avec une autre table T2 par une clé étrangère et que cette dernière est déjà fragmentée en fragmentation horizontale primaire.

Les fragments dérivés sont obtenus par une semi jointure entre la table T1 et chaque fragment de la table T2.

Exemple : Considérons la table de fait "Vente" suivante qui est reliée à la table de dimension "PointVente" par la clé étrangère *IdPointVente* :

IdPointVente	Reference	IdDate	QteVendue	Montant
Cl1	Ref1	07-03-2019	7	7700000
Cl2	Ref2	01-10-2020	3	3000000
Cl3	Ref2	25-07-2020	10	10000000
Cl1	Ref1	13-03-2021	2	2200000
Cl5	Ref3	01-01-2020	5	7500000

La table de fait globale "Vente" sera fragmentée en fonction de la table de dimension "PointVente" qui a déjà été fragmentée horizontalement en fonction de l'attribut *Continent* :

IdPointVente	Reference	IdDate	QteVendue	Montant
Cl1	Ref1	07-03-2019	7	7700000
Cl3	Ref2	25-07-2020	10	10000000
Cl1	Ref1	13-03-2021	2	2200000
Sous-table_Vente1 = Vente ⋉ Sous-table1				

IdPointVente	Reference	IdDate	QteVendue	Montant
Cl2	Ref2	01-10-2020	3	3000000
Cl5	Ref3	01-01-2020	5	7500000
Sous-table_Vente2 = Vente ⋉ Sous-table2				

FigureII.27 : Fragmentation horizontale dérivée

Donc, dans la fragmentation horizontale primaire, la fragmentation se fait sur la table même, alors qu'en fragmentation horizontale dérivée la fragmentation se fait en fonction d'une autre table.

La fragmentation primaire permettrait accélérer les opérations de sélection, tandis que la fragmentation dérivée améliorerait l'accélération des opérations de jointure.

II.7.1.3. Fragmentation hybride

Ce type de fragmentation consiste à mélanger les deux types précédents, c'est-à-dire fragmenter l'entrepôt verticalement puis fragmenter chaque fragment horizontalement ou vice versa [9, 49, 50].

II.7.2. Répartition d'un entrepôt de données

Une fois les données fragmentées, elles sont ensuite réparties ou distribués sur différents nœuds. La répartition est le processus qui consiste à distribuer des fragments d'un entrepôt sur plusieurs serveurs ou nœuds garantissant ainsi une haute disponibilité des données.

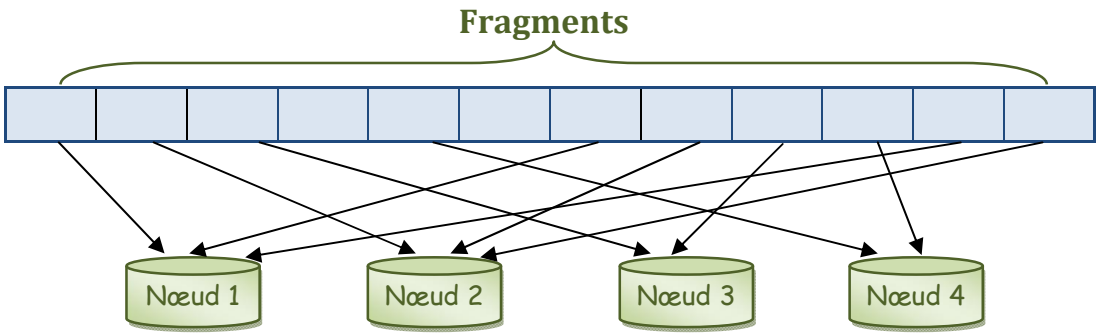
Les objectifs principaux de la répartition des fragments d'un entrepôt de données sont :

- Optimiser les performances globales en réduisant la charge d'un seul serveur.
- Assurer une distribution équitable des fragments en les répartissant de manière homogène sur l'ensemble des serveurs.
- Traiter indépendamment des fragments sur différents nœuds tout en maintenant la cohérence et l'intégrité des données.
- Permettre une répartition dynamique des fragments, qui doivent être redirigés en fonction de la charge et de l'augmentation des données, assurant ainsi un rééquilibrage pour maintenir de meilleures performances.

Une fois les données décomposées en fragments, ces derniers vont être répartis sur ldes nœuds en utilisant différentes stratégies de répartition [19, 53, 67] :

II.7.2.1. Répartition circulaire ou Round Robin

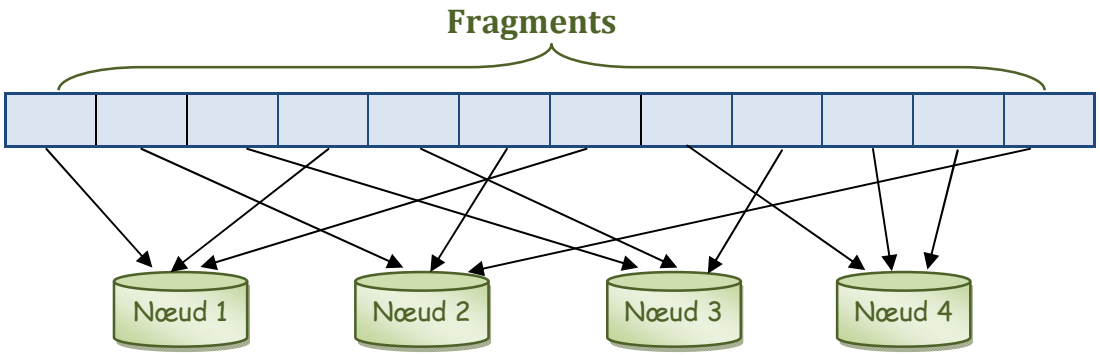
Cette stratégie simple et efficace répartit les nœuds de manière séquentielle et équitable sur les serveurs disponibles, où chaque fragment **i** est placé sur le nœud **i mod n**, avec n représentant le nombre total des nœuds.



FigureII.28 : Répartition des fragments en Round Robin

II.7.2.2. Répartition par hachage

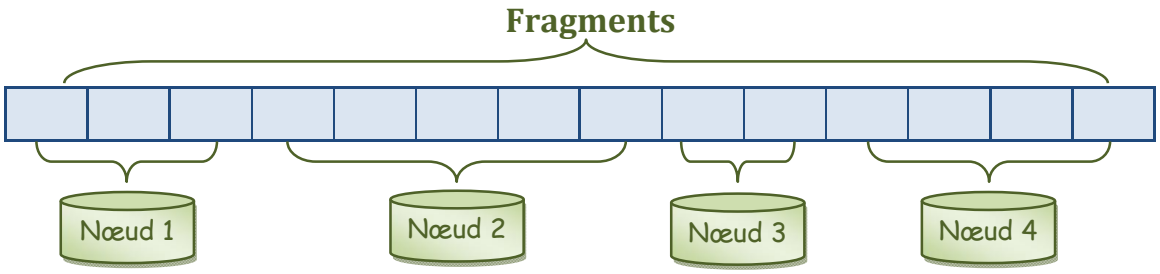
La répartition des fragments entre les différents serveurs dépend d'une fonction de hachage appliquée à un attribut. Cela assure une distribution uniforme des fragments.



FigureII.29 : Répartition des fragments par hachage

II.7.2.3. Répartition par intervalle

Les fragments sont attribués à des serveurs spécifiques en fonction de la plage de données. Cette méthode peut entraîner un déséquilibrage de la charge entre les différents nœuds du système.



FigureII.30 : Répartition des fragments par intervalle

II.8. Architecture Client/Serveur [59]

Dans l'architecture Client/serveur, un groupe de machines, appelées serveurs, fournit des services à un autre groupe, les clients, qui demandent ces services.

L'entrepôt de données peut être conçu selon ce modèle client-serveur, particulièrement adapté aux systèmes décisionnels en raison de sa capacité à gérer efficacement de grands volumes de données tout en maintenant des performances optimales.

Dans cette architecture, les données sont réparties sur plusieurs serveurs, chacun hébergeant une portion de l'entrepôt. Les clients peuvent interroger les serveurs de manière transparente, sans avoir à se soucier de l'emplacement physique des données.

II.9. Conclusion

En conclusion, un entrepôt de données est une solution essentielle pour l'organisation et l'analyse des données au sein des entreprises. Il permet de rassembler des données hétérogènes et variées provenant de sources multiples de manière cohérente et fiable.

La conception et la mise en œuvre d'un entrepôt de données nécessitent une étude approfondie basée sur les besoins spécifiques de l'organisation, tels que le volume de données à traiter et la complexité des requêtes.

Un entrepôt de données s'articule généralement autour de deux concepts clés : les faits, qui représentent les données quantitatives à analyser, et les dimensions, qui fournissent les axes d'analyse, organisés sous forme de cube OLAP, une structure multidimensionnelle permettant d'explorer les données selon différentes perspectives pour faciliter l'analyse décisionnelle

Savoir choisir un modèle de données (étoile, flocon, constellation) et un serveur (ROLAP, MOLAP ou HOLAP) approprié est fondamental. Une bonne sélection garantira que l'entrepôt sera performant et optimisé pour l'analyse décisionnelle.

Par la suite, les entrepôts de données distribués ont été conçus pour répondre aux exigences des systèmes décisionnels modernes. En combinant fragmentation, répartition des données et architecture client-serveur, ils garantissent une gestion distribuée et optimale des données. Cette approche assure une haute disponibilité, une évolutivité des systèmes et un accès transparent aux données.

Chapitre III

Les Structures de Données Distribuées et Scalables (SDDS)

III.1. Introduction

Un fichier est un ensemble d'informations sauvegardées sur un support de stockage dont la taille est fixe. La taille d'un fichier peut augmenter suite à des opérations de mise à jour (ajout, modification ou suppression). Cependant, si cette augmentation dépasse la capacité de stockage disponible, les modifications ne pourront pas être enregistrées et toute nouvelle information sera donc perdue.

Les SDDS représentent une nouvelle classe de données qui permettent la distribution dynamique d'un fichier sur plusieurs emplacements ou serveurs. Lorsque la taille d'un fichier augmente, de nouveaux serveurs seront ajoutés, permettant ainsi à la taille du fichier de croître indéfiniment sans se heurter à une limite de capacité de stockage.

Un autre avantage majeur des SDDS réside dans la distribution d'un seul fichier sur différents sites et la gestion décentralisée de grands volumes de données. Cette approche permet un accès et un traitement parallèles, tout en maintenant des performances d'accès élevées.

III.2. Structures de Données Scalables et Distribuées

Les SDDS sont une classe de données apparue en 1993[19], et conçue pour les systèmes multi-ordinateurs [13, 31, 43].

Un fichier SDDS est constitué d'un ensemble d'enregistrements stockés sur des sites dits serveurs [54], qui peuvent être accéder par d'autres sites clients. Initialement, un fichier est stocké sur un seul serveur, suite à des opérations d'insertion de nouveaux enregistrements, il peut s'étendre sur un nombre théoriquement illimité de serveurs [31].

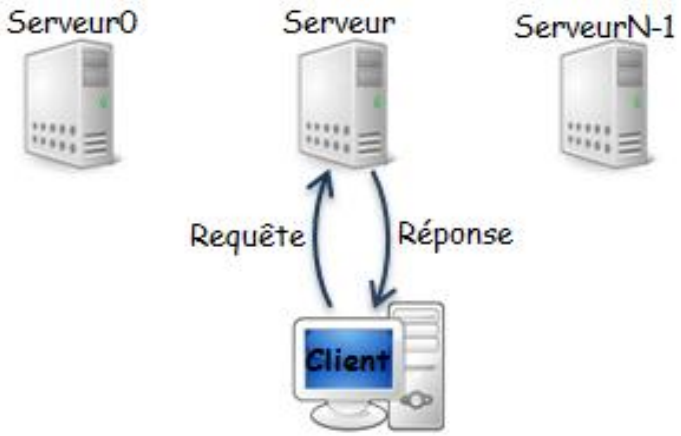
Chaque enregistrement est identifié par une clé unique, qui sert pour le calcul de son adresse.

Un client peut accéder à un serveur grâce à son propre schéma d'adressage qui forme son image [45]. Cette image rassemble un ensemble de paramètres permettant de calculer les adresses des enregistrements.

La distribution d'un seul fichier SDDS sur plusieurs serveurs favorise un traitement parallèle augmentant ainsi la disponibilité du fichier sans compromettre les performances d'accès pour les données qui peuvent s'accroître continuellement. Cela illustre parfaitement la notion de scalabilité [16, 17, 22], qui désigne la capacité d'un système à se

développer considérablement en taille tout en maintenant son efficacité et ses performances.

Aussi, l'accès aux SDDS qui sont stockées sur les mémoires vives des serveurs [31], s'effectue dans un temps réduit par rapport aux données qui sont stockés sur disques locales.



FigureIII.1 : Architecture globale d'un système de distribution de données en SDDS

Les SDDS sont utilisées dans divers domaines [42], notamment :

- **Bases de données distribuées :** Google Bigtable et Apache Cassandra.
- **Entrepôts de données :** comme Amazon Redshift et Google BigQuery.
- **Gestion de documents et archives électroniques :** Comme exemples SharePoint et Alfresco.
- **eGov (Gouvernement électronique) :** Tels que Gov.uk et FranceConnect.
- **Données en streaming (stream data repositories) :** comme le cas du système Apache Kafka.
- **Systèmes de recommandation :** Comme le font Netflix et Amazon.
- **Big Data et analytique :** Exemples Hadoop et Spark.
- **Gestion de logs et événements :** A l'instar de l'ELK Stack.

III.2.1. Propriétés des SDDS

- Contrairement aux fichiers classiques qui sont stockés sur un seul site, un fichier SDDS peut être réparti sur plusieurs sites serveurs. La répartition se fait d'une manière dynamique [16] et transparente [66] à l'utilisateur, ainsi, un fichier peut s'accroitre suite à des opérations d'insertion et d'éclatement [46] vers un nombre théoriquement infini de serveurs, comme il peut se rétrécir par des opérations de suppression.
- La répartition d'un fichier sur plusieurs sites, permet d'éviter d'avoir un seul site central [14, 37, 56], prévenant ainsi les risques de goulot d'étranglement [19] et l'altération des performances d'accès.
- Un fichier SDDS est réparti sur plusieurs sites serveurs qui peuvent être accédés par des sites clients. Chaque client a son propre image [37] sur la répartition du fichier SDDS.

  L'accroissement progressif du fichier implique le déplacement des enregistrements de ce dernier vers d'autres serveurs et par conséquence le changement de leur adresse. Ce changement de répartition n'est pas envoyé aux clients d'une manière synchrone, ainsi, un client peut avoir une image erronée sur la répartition du fichier qui induit à commettre des erreurs d'adressage [56].
- Un serveur peut détecter une erreur d'adressage, il réoriente alors la requête reçue vers le serveur adéquat qui se chargera d'envoyer un message d'ajustement d'image IAM (Image Adjustment Message) au client [4] pour qu'il corrige son image et ne plus refaire la même erreur.

III.2.2. Classification des SDDS

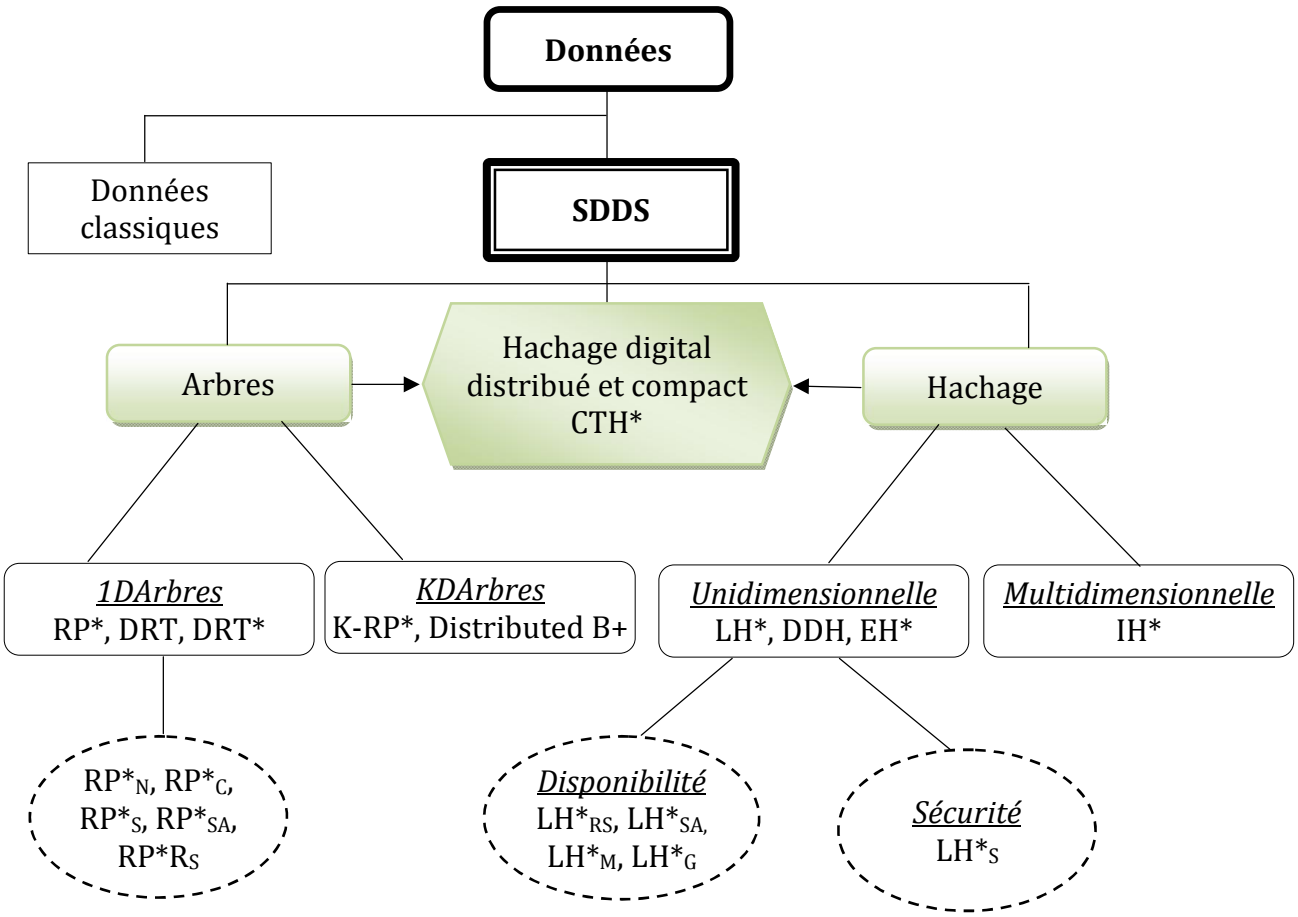
Selon la méthode de répartition des enregistrements d'un fichier, les SDDS se classent principalement en deux catégories [9, 66] :

- ***La distribution par hachage :*** Cette approche repose sur l'utilisation d'une fonction de hachage appliquée aux clés des enregistrements. La fonction génère une adresse de stockage, permettant d'affecter un nouvel enregistrement ou de retrouver rapidement l'adresse d'un enregistrement existant. Ce mécanisme est particulièrement apprécié pour sa simplicité et son efficacité dans la gestion des données réparties de manière uniforme, réduisant ainsi les risques de déséquilibre de charge entre les sites.
- ***La distribution par intervalle :*** Cette méthode repose sur une structure arborescente pour représenter les adresses, visant à préserver l'ordre des enregistrements afin de

faciliter les requêtes basées sur des intervalles. Cette organisation est particulièrement adaptée aux systèmes nécessitant des accès séquentiels ou des analyses par intervalles, mais peut entraîner des déséquilibres de charge si certaines plages de valeurs sont plus denses que d'autres.

- Une troisième catégorie, appelée ***hachage digital*** a été définie. Elle combine les avantages des deux approches précédentes. Elle utilise un mécanisme de hachage tout en préservant une forme d'organisation structurée des données. L'objectif est de bénéficier de la simplicité et de l'efficacité du hachage tout en maintenant l'ordre des enregistrements.

Pour chaque catégorie de SDDS, on trouve une multitude d'algorithmes, les plus connues sont représentés par le schéma suivant :



FigureIII.2 : Classification des SDDS

III.3. Distribution par hachage

Avant de présenter la première SDDS, il convient de donner un aperçu du hachage linéaire, concept fondateur de ces structures de données.

III.3.1. Le hachage linéaire dynamique

Le hachage linéaire dynamique a été défini en 1978 par Litwin et Larson. Par la suite, pour l'adapter aux environnements répartis, plusieurs algorithme ont vu le jour, notamment : LH*, DDH*, EH*, IH*… [9, 67].

Dans ce qui suit, nous allons explorer le LH puis le LH* (Distributed Linear Haching ou le hachage linéaire distribué), l'algorithme le plus répandu et le plus utilisé [14].

III.3.1.1. LH : Linear Haching

Un fichier LH est composé d'un ensemble d'enregistrements, chacun identifié par une clé (C) [56]. Les enregistrements sont regroupés en des cases numérotées 0, 1, 2, …. N-1, une case peut contenir jusqu'à b enregistrements.

Un fichier débute avec une seule case (case0), et peut s'étendre vers d'autres cases après des opérations d'insertion, ou se rétrécir avec fusion des cases.

Le calcul de l'adresse d'un enregistrement se fait à l'aide d'une fonction de hachage h_i (C) appliquée à sa clé pour trouver la case qui l'héberge, tel que : h_i (C) = C mod N * 2^i, où :

- N : le nombre de cases, initialisé à 1.
- i : le niveau du fichier [43], i = 0, 1, 2, 3, ….. Le i s'incrémente de 1 à chaque fois que la taille du fichier double.

Si un nouvel enregistrement va être ajouté dans une case contenant déjà b enregistrements [17], on dit qu'il y a **« une collision »** et que la case est en débordement [].

Un éclatement va être effectué [54], une nouvelle case sera ajoutée et presque la moitié des enregistrements [31, 56] de la case éclatée seront transférés vers la nouvelle qui peut être accédée en utilisant la fonction de hachage h_{i+1}. La case où s'est produite la collision n'est pas forcément celle qui va être éclatée [19], mais la case désignée par un pointeur n [14].

L'algorithme déterminant la nouvelle adresse des enregistrements transférés, correspondant à la case créée après l'éclatement, est le suivant:

```
a ← hi (C)
Si  a < n  Alors
    a ← hi+1 (C)
FinSi
```

AlgorithmeIII.1 : Calcul des adresses des enregistrements

- Si a < n alors C appartient à une case déjà éclatée et le calcul d'adresse se fait par h_{i+1}.
- Si a ≥ n alors C appartient à une case non éclatée et le calcul d'adresse se fait par h_i.

Après chaque éclatement, le compteur n est incrémenté de 1 pour indiquer l'adresse de la prochaine case à éclater. Les éclatements des cases dans LH se font d'une manière cyclique. n prend consécutivement les valeurs : 0 ; 0, 1 ; 0, 1, 2 ; 0, 1, 2, ... N - 1 ; 0 ... 2^i *N-1 ; ….. [14]

```
Créer une nouvelle case
Recalculer les adresses de toutes les clés de la case n
n ← n+1
Si  n >=2i * N  Alors
    n ← 0
    i ← i+1
FinSi
```

AlgorithmeIII.2 : Mise à jour de i et n après l'éclatement d'une case

Si n >= 2^i*N alors, on a dépassé le nombre total de cases, n est ainsi remis à 0 (pointe vers la première case) et le niveau de fichier i est incrémenté. La nouvelle fonction d'adressage sera h_{i+1} au lieu de h_i.

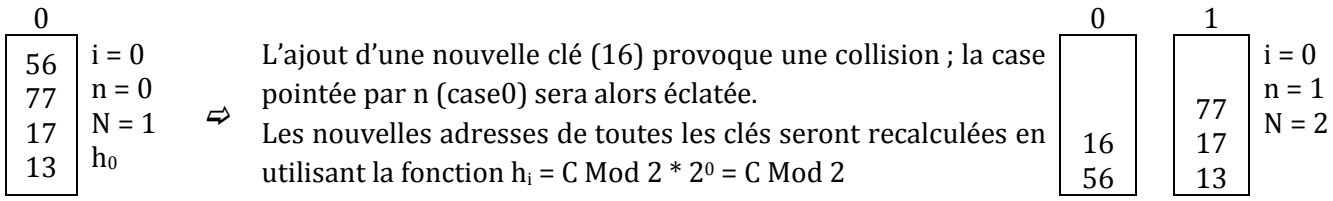
Exemple :

Initialement, il existe une seule case "case0" avec i = 0, n = 0, N = 1. On suppose que la capacité de stockage des cases égale à 4 (b = 4).

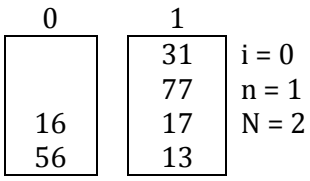
Chaque enregistrement est représenté par une clé. Les premières clés 13, 17, 77, 56 seront toutes dirigées vers case0.

La fonction de hachage appliqué pour calculer l'adresse de toutes ces clés :

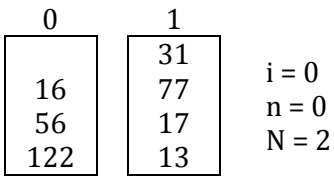
$$\mathbf{h_i(C) = C\ Mod\ 1 * 2^0 = C\ Mod\ 1.}$$



L'ajout des clés (31) se fait sans collision dans la case pointée par n (case1).



L'ajout de la clé (122) provoque une collision dans la case1. Les nouvelles adresses des clés seront calculées en utilisant la fonction $h_i = C \text{ Mod } 2 * 2^0 = C \text{ Mod } 2$



FigureIII.3 : Répartition des enregistrements selon LH

➲ **Fusion de cases**

Suite à une suppression, le nombre des enregistrements contenus dans une case(i) diminue [39].

Si le nouveau nombre des enregistrements dans la case(i) plus le nombre des enregistrements dans une autre case(j) est inférieur à b, alors on peut fusionner les deux cases (i et j) en une seule et libérer l'une d'entre elles. Les enregistrements vont donc être regroupés dans une seule case.

L'algorithme de libration d'une case se présente comme suit:

```
Libérer la case n
n ← n -1
Si  n < 0  Alors
    n ← 2^i * N-1
    i ← i-1
FinSi
```

AlgorithmeIII.3 : Mise à jour de i et n après la libération d'une case

III.3.2. Distribution du LH : LH* [67]

Le LH* a été élaboré par Litwin, Neimat et Schneider en 1993 [54], c'est une version distribuée et extensible de la méthode du hachage linéaire LH [2, 14, 54].

Le LH* consiste à distribuer un fichier sur plusieurs serveurs chacun identifié par une adresse logique 1, 2, 3….. Un serveur héberge et gère une case du fichier qui peut contenir b enregistrements. Les cases sont stockées dans les mémoires vives des serveurs.

Un fichier commence avec une seule case (ou paquet), et peut s'étendre d'une manière dynamique vers d'autres serveurs par des opérations d'insertion, ou se rétrécir par des opérations de fusion de cases [14].

Le principe du LH* est le même que celui de LH dynamique. Chaque enregistrement est identifié par une clé servant pour le calcul de son adresse.

Une case maintient 2 paramètres :

- i : le niveau du fichier,
- n : adresse de la prochaine case à éclater.

Le nombre total des cases $N = 2^i + n$ [14].

Un fichier LH* est accédé par des sites autonomes dits clients. Chaque client a une image du fichier LH* [31], contenant la valeur présumée du niveau de la case (i') et l'adresse de la prochaine case à éclater (n') (i' et n' initialisés à 0) qui sont utilisés pour le calcul de l'adresse logique a' d'un enregistrement de clé (C).

Dans un environnement réparti, plusieurs clients peuvent accéder à un serveur et changer les valeurs de i et n. Les nouveaux changements ne sont pas envoyés aux clients d'une manière synchrone, ainsi, les valeurs réelles du niveau du fichier et la case à éclater au niveau du serveur peuvent être différentes de celle de l'image du client. Dans ce cas, une requête envoyée peut s'adresser au mauvais serveur.

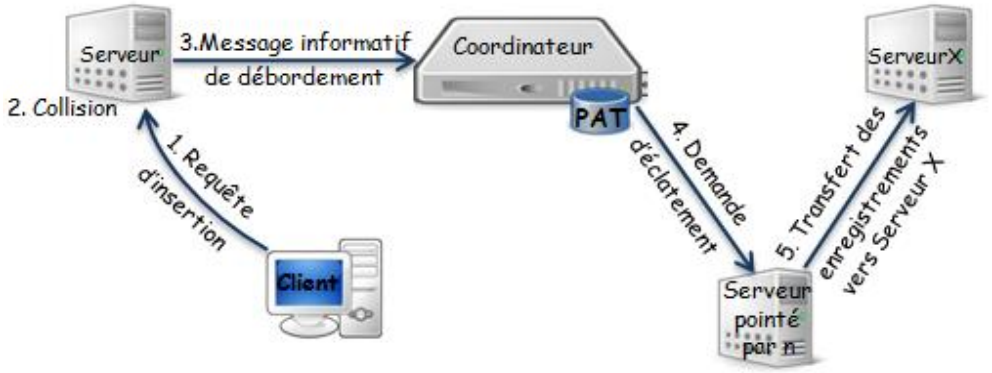
À la réception d'une requête, un serveur LH* vérifie si la requête lui est destinée ou non, et donc il la traite ou la renvoie vers un autre serveur avec l'envoi d'un message correctif IAM (qui comporte le niveau et l'adresse de la case correcte) au client, pour que ce dernier actualise son image et ne plus refaire la même erreur d'adressage.

➲ Eclatement d'une case

Dans LH*, un site spécial a été ajouté : le coordinateur, implanté au-dessus de la première case LH*. Il est le seul à avoir une image correcte du fichier [37]. Il a pour rôle d'assurer la cohérence entre les paramètres [57] des clients et des serveurs et de coordonner les éclatements des cases.

Quand une case déborde, le serveur informe le coordinateur en lui envoyant un message informatif, ce dernier envoie un message de demande d'éclatement au serveur pointé par n (contenant la case (i)).

Un nouvel serveur (une nouvelle case (j)) sera ajouté. La moitié des enregistrements de la case i vont être transférés vers la case j, leur nouvelle adresse est calculée en appliquant la fonction de hachage h_{i+1}, tel que : $h_i = C \bmod N * 2^i$



FigureIII.4 : Collision et éclatement d'un serveur

➲ **Envoi d'une requête**

Un client envoie sa requête (recherche, ajout ou mise à jour) à un site serveur d'adresse a', calculée à l'aide de la fonction de hachage h_{i+1}, en utilisant ses propres paramètres i' et n' conformément à l'algorithme suivant :

```
a' ← h_i' (C)
Si  a' < n'  Alors
    a' ← h_i'+1 (C)
FinSi
```

AlgorithmeIII.4 : Calcul d'une adresse côté client avec i' et n'

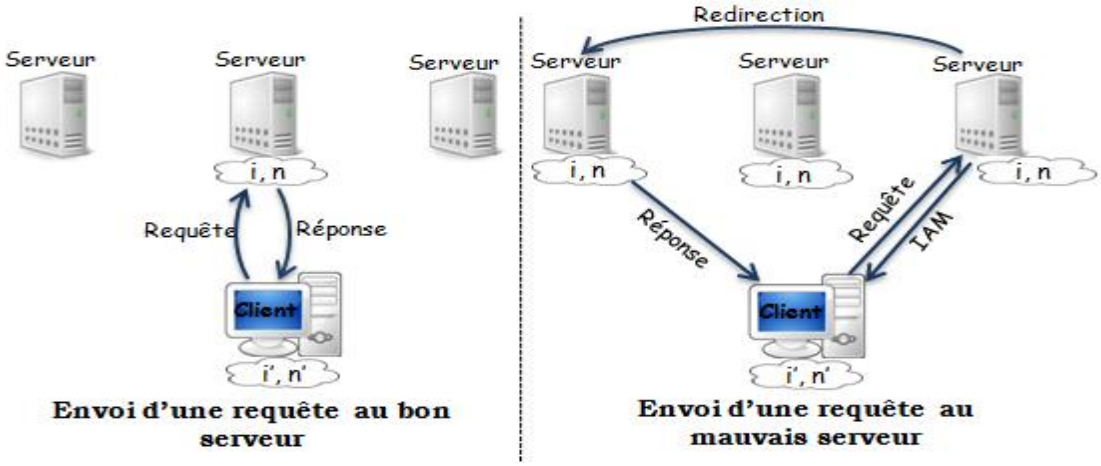
➲ **Réception d'une requête par un serveur**

A la réception d'une requête, le serveur destinataire calcule l'adresse a selon ses paramètres i et n pour vérifier si la requête lui est destinée ou non [40]. Si l'adresse a est la même a' (il s'est avéré que c'est le bon serveur), il la traite et envoie la réponse au client, sinon, il la renvoie vers le serveur d'adresse a'' [2].

```
a' ← hj(c)
Si  a' = a   Alors
     Accepte c
Sinon
     a'' ← hj - 1(c)
FinSi
Si  (a < a'') Et (a'' < a')   Alors
     a' ← a''
     Envoies c à la case a'
FinSi
```

AlgorithmeIII.5 : Test et renvoi d'une requête par un serveur

Dans le cas où a' <> a (erreur d'adressage), le serveur recevant la requête suppose que le serveur contenant l'enregistrement recherché n'a pas encore subi un éclatement alors il utilise la fonction h_{j-1} pour calculer a''.



FigureIII.5 : Envoi et réception d'une requête

Après l'acheminement de la requête au serveur approprié, le serveur ayant redirigé la requête envoie un message correctif avec ses paramètres réels au client pour qu'il ajuste son image. Le client met alors à jour son image (i' et n').

➲ Réajustement de l'image du client

En cas d'erreur d'adressage, un client reçoit un message d'ajustement d'image (IAM) [2] avec les valeurs de la première case ayant reçu le message comportant son niveau actuel j [43], pour que le client met à jour ses propres paramètres i' et n' selon l'algorithme suivant :

```
i' ← j-1
n' ← a +1 //a : numéro du serveur ayant transmis l'IAM
Si  n' >= 2^i'  Alors
    n' ← 0
    i' ← i'+1
FinSi
```

AlgorithmeIII.6 : Réajustement de l'image d'un client

➲ **Fusion de cases**

Le nombre d'enregistrements d'une case hébergée sur un serveur (i) peut diminuer à la suite d'une opération de suppression [39, 40]. Si la somme du nouveau nombre d'enregistrements dans la case de ce serveur et ceux d'une autre case sur un serveur (j) est inférieure à une valeur seuil b, il devient alors il est possible de fusionner les cases des serveurs (i) et (j) en une seule. Cette fusion permet la libération de l'un des serveurs, et les enregistrements seront ainsi regroupés dans une seule case.

III.3.3. Variantes du LH* [9, 19]

Afin d'augmenter le degré de disponibilité, plusieurs variantes de l'algorithme LH de base ont été proposées :

➲ **LH*$_{LH}$ [10, 11, 37] :**

Proposée par Karlesson, Litwin et Risch, cette technique utilise 2 niveaux d'indexations, le premier indexe réseau sert pour trouver la case recherchée en utilisant le LH*, le deuxième indexe interne à la case permet d'accéder à un enregistrement selon LH.

➲ **LH*$_M$ (with Mirroring) :**

Afin d'assurer la tolérance aux pannes et la haute disponibilité. Le LH$_M$ contient un fichier organisé en LH* et son miroir en LH* également, les deux fichiers sont fonctionnels et interrogeables. Les données sont sauvées sur un fichier et répliquées sur son miroir. Pour ce faire, il est nécessaire de mettre en place des techniques de gestion de répliques et de propagation des mises à jour afin d'éviter toute incohérence entre les copies des mêmes données.

Le principal inconvénient de cette technique réside dans le coût de stockage élevé, résultant du nombre de fichiers dupliqués.

➲ **LH*$_S$ (with stripping) :**

Chaque enregistrement est divisé en k segments en plus d'un segment k+1 dit de parité calculé à partir des segments 1, 2, …. k.

Un fichier segment contient tous les segments S_i (i = 1, 2, 3, ….. k+1) de tous les enregistrements avec le segment de parité.

Cette SDDS LH*$_S$ nécessite moins d'espace de stockage que la SDDS LH*$_M$, mais les opérations de mise à jour nécessitent davantage de temps et génèrent un plus grand nombre de messages.

➲ **LH*$_G$ (with record grouping) :**

Pour augmenter le degré de disponibilité d'un fichier LH*, on utilise des pages de parité par groupe d'enregistrements.

➲ **LH*$_{SA}$ (Scalable Availability):**

Plusieurs fichiers de parité sont utilisés, ce qui permet de récupérer plusieurs pages simultanément. Le nombre de fichiers de parité augmente progressivement à mesure que le fichier s'étend, offrant ainsi une résistance accrue aux pannes. Le coût de stockage dépend de la taille du fichier.

➲ **LH*$_{RS}$ (Scalable Availability Using Reed Solomon Codes)** [39, 40, 64] **:**

Un fichier LH*RS est subdivisé en groupes de parité, où un groupe est formé de m cases de données et de k cases de parité codés selon les codes de Reed Solomon. Chaque enregistrement appartient à un seul groupe.

III.4. Distribution par intervalle

À la différence des SDDS reposant sur le hachage, où une clé est insérée dans une case disponible sans considération de l'ordre des clés, les SDDS fondées sur la distribution par intervalle adoptent une approche différente : chaque clé est insérée dans la case correspondant à l'intervalle de valeurs auquel elle appartient. Parmi ces SDDS, on trouve RP* et ses variantes : RP*n, RP*c, RP*s, DRT, DRT*, K-RP*, K-DRT, ……[9]

III.4.1. RP* : Range Partitioning [8, 9, 41, 67]

Le RP* est une structure de données basée sur les principes des arbres B+ développée par Litwin, Neimat et Schneider en 1994.

Un fichier RP* se compose de plusieurs enregistrements répartis sur des cases. Un enregistrement se compose d'un champ clé ordonné, utilisé pour son identification, et d'un champ non clé, destiné à contenir la donnée. Les enregistrements sont stockés sur des serveurs ordonnés logiquement selon les valeurs des clés. Chaque serveur dispose d'une case (ou bucket) avec une capacité de b enregistrements.

Un paquet possède un en-tête avec 2 valeurs : une clé minimale λ et une clé maximale Λ. L'intervalle]λ , Λ] (initialisé respectivement à -∞ et +∞) est appelé portée ou rang de la case. Toute insertion d'un nouvel enregistrement s'effectue dans cette case.

Un enregistrement de clé C appartient à une case si sa clé est incluse dans le rang de la case, ie : $\lambda < C \leq \Lambda$. S'il y a un ajout d'un nouvel enregistrement et que le nombre des enregistrements dépasse la capacité b de la case, elle est alors débordée et sera éclatée. Le processus d'éclatement se déroule comme suit :

- Définir la clé du milieu CM de la case i (celle en débordement et de rang]λ , Λ]).
- Créer une nouvelle case j en fin du fichier.
- Définir la portée da la case j qui est égal à [CM , Λ].
- Déplacer la moitié des enregistrements de la case i vers la case j (ceux dont C > CM).
- Modifier la portée da la case i = [λ , CM].

Exemple :

Examinons l'exemple suivant. Nous allons introduire progressivement tous les enregistrements représentés par leur clé : 3, 520, 10, 17, 79, 56, 16, 19, 60, 37, 6, 170, 280, 1, 318, 7, 11, 725. On suppose que b = 4.

- Initialement un fichier RP* est réparti sur une seule case (0) avec :
  $\lambda = -\infty$ et $\Lambda = +\infty$.
- Les 4 premiers enregistrements vont vers la case0 selon l'ordre des clés : 3, 10, 17, 520.

0
520
17
10
3
+∞
-∞

- L'ajout d'une nouvelle clé (79) provoque une collision dans la case0, sa capacité est dépassée.
- Toutes les clés (avec la clé qui a provoquée la collision) vont être ordonnées : 3, 10, 17, 79, 520.
- La clé du milieu = 17.
- Les clés dont la valeur est inférieure ou égale à 17 restent dans la case0, les autres (supérieure à 17) vont être déplacées vers une nouvelle case1.
- La portée de la case0 =]-∞ , 17].
- La portée de la case1 =]17, +∞[.

0	1
17	
10	520
3	79
17	+∞
-∞	17

0	1
17	520
16	79
10	56
3	19
17	+∞
-∞	17

- L'ajout des clés (56, 16, 19) se fait sans collision.

0	1	2
17		
16	60	
10	56	520
3	19	79
17	60	+∞
-∞	17	60

- L'ajout d'une nouvelle clé (60) provoque une collision dans la case1.
- Toutes les clés vont être ordonnées : 19, 56, 60, 79, 520.
- La clé du milieu = 60.
- Les clés dont la valeur est inférieure ou égale à 60 restent dans la case1, les autres vont être déplacées vers une nouvelle case2.
- La portée de la case1 =]17, 60].
- La portée de la case2 =]60, +∞[.

L'ajout de la clé (37) ⇨ se fait sans collision

0	1	2
17	60	
16	56	
10	37	520
3	19	79
17	60	+∞
-∞	17	60

L'ajout de la clé (6) provoque ⇨ une collision dans la case0

0	1	2	3
	60		
10	56		
6	37	520	17
3	19	79	16
10	60	+∞	17
-∞	17	60	10

Après l'insertion de tous les enregistrements, ⇨ on obtient la répartition suivante :

0	1	2	3	4	5
	60				
6	56	280	11	725	
3	37	170	17	520	10
1	19	79	16	318	7
6	60	280	17	+∞	10
-∞	17	60	10	280	6

FigureIII.6 : Répartition des enregistrements selon RP*

III.4.2. Variantes du RP* [6, 45, 66]

➲ **RP*$_N$: No index**

Dans RP*$_N$, la communication entre les clients et les serveurs se fait en multicast, c'est-à-dire qu'un client envoie sa requête avec le nom du fichier vers tous les serveurs. Chaque serveur vérifie si la clé de la requête appartient à sa portée, ou s'il y a une intersection entre les deux intervalles dans le cas d'une requête par intervalle ; si c'est le cas, il traite la requête et envoie la réponse en unicast au client. Les serveurs non concernés ignorent la requête.

Dans les requêtes par intervalle, la réponse se fait par l'union des portées des cases qui ont répondu, ou bien le client se contente des réponses obtenues à l'expiration d'un time-out.

La réception d'une réponse avec seulement les clés max et min d'une case indique que la requête est infructueuse.

➲ **RP*$_C$: Client index**

Le nombre de messages échangés diminue dans RP*$_C$ du fait que les requêtes et les réponses sont envoyées en unicast, seulement les redirections en multicast.

Un client dispose d'une image du fichier sous forme d'une table dynamique T[0, 1, 2, ….]. Chaque entrée de la table contient le couple (A, C), où A est l'adresse d'une case, C est sa clé max. Si A = * alors l'adresse est inconnue et tout message va être envoyé en multicast.

Pour envoyer sa requête, le client parcours sa table à la recherche d'une entrée dont la clé est immédiatement supérieure à la clé de l'enregistrement recherché. Si A = * alors la requête sera envoyée en multicast sinon au destinataire de l'adresse A.

A la réception d'une requête, le serveur vérifie la clé avec sa portée, s'il y a intersection, alors il la traite et envoie la réponse plus l'adresse et la portée de la case au client, si c'est un message redirigé, il ajoute en plus l'adresse et la portée de la case intermédiaire.

Si la requête ne lui est pas destinée (clé n'appartient pas à sa portée), et si la requête est en multicast alors pas de suite, sinon (requête en unicast) il la redirige en multicast aux autres serveurs avec l'adresse et la portée de la case recherchée.

Ajustement de l'image client

Suite à une erreur d'adressage, le client reçoit en plus de la réponse un champ contenant le triplet (λ, A, Λ) pour corriger son image, tel que [λ, Λ] représente la portée du serveur qui a traité la requête et A son adresse.

➲ **RP*$_S$: Server index**

RP*$_S$ ajoute à RP*c un index reparti sur des serveurs d'index. Cet index représente une image de la structure globale de la répartition du fichier. Tous les messages même de redirection sont envoyés en unicast.

Il existe d'autres SDDS basées sur les arbres comme : RP*$_{RS}$ (une variante à haute disponibilité), RP*$_{HA}$ (pour la réplication de chaque intervalle de l'espace des clés sur deux serveurs), k-RP* (une variante multi-attributs), DRT (Distributed Random Tree), BDST (Balanced and Distributed Search Trees) et LDT (Logarithmic Distributed Search Tree).

III.5. Le hachage digital

Le hachage digital est une technique proposée par Pr. Litwin en 1981. Elle constitue une approche efficace pour la gestion de fichiers monoclés, ordonnés et dynamiques. Aussi, elle combine les avantages de deux techniques précédemment abordés : la simplicité et l'efficacité des algorithmes de hachage, ainsi que la préservation de l'ordre des enregistrements, caractéristique des méthodes basées sur les arbres.

Pour indexer et ordonner les enregistrements d'un fichier, le hachage digital utilise un arbre binaire lexicographique ou arbre digital maintenu en mémoire centrale et considéré comme une fonction de hachage.

Avant d'aborder le CTH*, une SDDS appliquée aux environnements distribués, il est nécessaire de comprendre d'abord le TH, qui en constitue la base. Nous aborderons ensuite le CTH, qui représente une amélioration du TH.

III.5.1. TH : Trie Haching [2, 4, 66]

Un fichier TH se compose d'un ensemble d'enregistrements organisés en des cases stockées en mémoire secondaire. Les cases sont numérotées 0, 1, 2, … N. Chaque case peut contenir jusqu'à b enregistrements au maximum.

Un enregistrement est identifié par une clé primaire de la forme $d_0d_1d_2 \dots d_i \dots d_{k-1}d_k$, où :

- d_i : C'est un digit d'un alphabet donné.
- k+1 : La longueur de la clé.
- i : C'est un numéro qui représente le niveau du digit.

➲ Eclatement d'une case et construction de l'arbre digital

Le débordement d'une case provoqué par la tentation d'ajouter un nouvel enregistrement alors qu'elle contient déjà b enregistrements implique son éclatement. Une nouvelle case va alors être ajoutée et la moitié des enregistrements de la case débordée vont être transférés vers la nouvelle case tout en respectant l'ordre des clés. Cela entraîne également l'extension de l'arbre digital par l'ajout de nouveaux nœuds.

A l'aide de l'exemple suivant nous allons illustrer le principe du hachage digital ainsi que la construction de l'arbre digital.

Notons :

- Le caractère barre « | » et espace « _ » représentent respectivement la plus grande et la plus petite valeur d'un digit.
- (d, i) : Un nœud interne de l'arbre digital de digit d et niveau i.
- Les nœuds externes (ou feuilles) contiennent les adresses des cases.

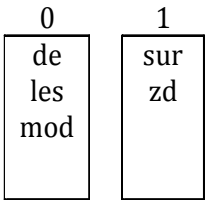
Exemple :

Nous allons distribuer la liste des enregistrements suivants représentés par leur clé : de, mod, zd, sur, les, th, tel, uv, av, par, ps, le, rnb, st, xy, zero, xp, cv, cth.

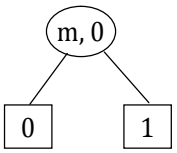
- Initialement l'arbre est vide : "|0"
- Les 4 premiers clés : de, mod, zd, sur seront insérées dans l'ordre dans la case0.

Le graphe associé :

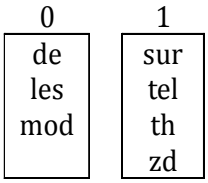
- L'ajout de l'enregistrement "les" provoque une collision dans la case0.
- Toutes les clés seront ordonnées : de, les, mod, sur, zd.
- La clé du milieu CM = "mod".
- Les clés inférieures ou égales à "mod" restent dans la case0.
- Les clés supérieures à "mod" seront transférées vers une nouvelle case1.
- L'arbre est mis à jour : le premier digit de CM "mod" est le digit de l'arbre, suivi du numéro de la case débordée, puis celui de la nouvelle case.
- L'arbre devient : "m0|1".
- La clé Max de la case 0 = "m", case 1 = "|"



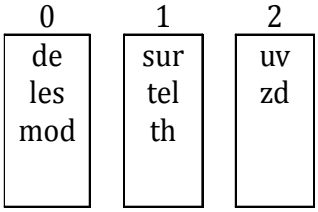
Le graphe associé :



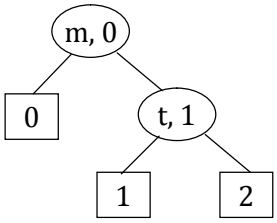
- L'ajout des enregistrements "th, tel" se fait sans collision.
- L'arbre reste : "m0|1".



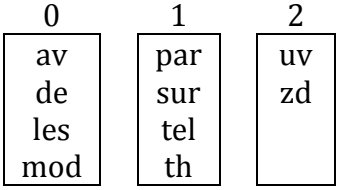
- L'ajout de l'enregistrement "uv" provoque une collision dans la case1.
- Toutes les clés seront ordonnées : sur, tel, th, uv, zd.
- La clé du milieu CM = "th".
- Les clés inférieures ou égales à "th" restent dans la case1.
- Les clés supérieures à "th" seront transférées vers une nouvelle case2.
- La nouvelle valeur de la clé Max de la case1 = "t".
- L'arbre devient : "m0 t1|2".



Le nouveau graphe :



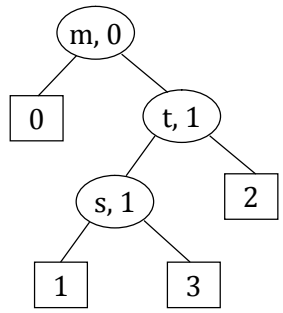
- L'ajout des enregistrements "av, par" se fait sans collision.
- L'arbre reste : "m0 t1|2".



- L'ajout de l'enregistrement "ps" provoque une collision dans la case1.
- Toutes les clés seront ordonnées : par, ps, sur, tel, th.
- La clé du milieu CM = "sur".
- L'arbre devient : "m0 s1 t3|2".

0	1	2	3
av de les mod	par ps sur	uv zd	tel th

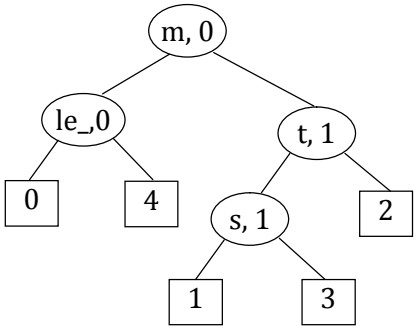
Le nouveau graphe :



- L'ajout de l'enregistrement "le" provoque une collision dans la case0.
- Toutes les clés seront ordonnées : av, de, le, les, mod.
- La clé du milieu CM = "le".
- Le nouveau digit de l'arbre "le_" pour différencier "le" de "les"
- L'arbre devient : "le_0 m4 s1 t3|2".

0	1	2	3	4
av de le	par ps sur	uv zd	tel th	les mod

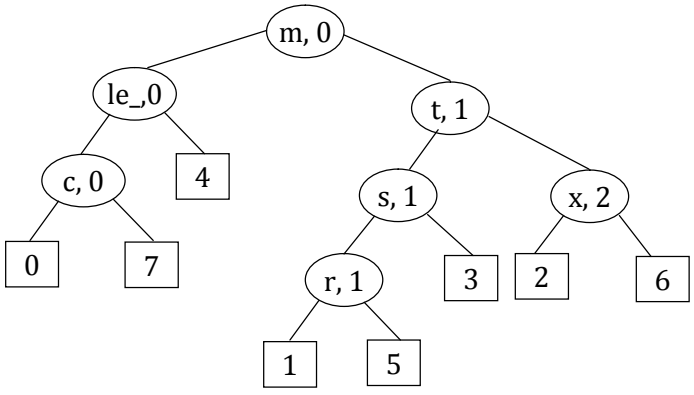
Le graphe associé :



Après la répartition des toutes les clés, on obtient l'arbre : c0 le_7 m4 r1 s5 t3 x2 |6

0	1	2	3	4	5	6	7
av cth cv	par ps rnb	uv xp xy	tel th	les mod	st sur	zd zero	de le

Le graphe associé est le suivant :



FigureIII.7 : Répartition des enregistrements selon TH

Chaque nœud de l'arbre possède deux pointeurs : un vers le fils gauche et un autre vers le fils droit. La représentation de l'exemple en dessus est de la forme suivante :

Pointeur droit	-1	4	7	-2	3	5	6
digit, N°Case	m, 0	le_,0	c, 0	t, 1	s, 1	r, 1	x, 2
Pointeur gauche	-1	-1	0	-1	-1	1	2

FigureIII.8 : Illustration des pointeurs associés aux nœuds

III.5.2. CTH : Le hachage digital compact [5, 66]

Dans le hachage digital, la taille de l'arbre dépend de la taille du fichier, plus le fichier est volumineux, plus l'arbre croit et nécessite plus d'espace mémoire. Pour palier ce problème, une nouvelle méthode a été définie : le CTH.

L'objectif principal du CTH est de rendre l'arbre plus compact et par conséquence réduire l'espace mémoire nécessaire pour sa représentation.

Dans le CTH, l'arbre digital est constitué d'un ensemble de nœuds internes et externes. Un nœud interne est un digit, alors qu'un nœud externe est un pointeur vers une case du fichier. Les pointeurs vers les nœuds internes sont supprimés.

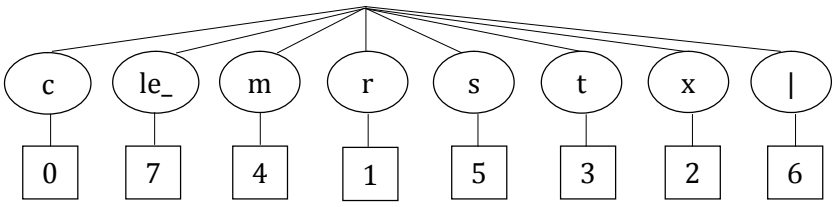
Le parcours de l'arbre se fait séquentiellement (en préordre), branche par branche et de haut en bas.

➲ Construction de l'arbre digital

Reprenant l'arbre digital de l'exemple précédent. Le parcours de l'arbre obtenu se fait en préordre, en allant de haut en bas et gauche vers la droite en omettant les nœuds internes.

L'arbre résultant est : c0 le_7 m4 r1 s5 t3 x2 |6

En éliminant les pointeurs vers les nœuds internes, on obtient le graphe suivant :



FigureIII.9 : Arbre de distribution selon CTH

Les nœuds internes sont les digits, les feuilles sont les adresses des cases. Le digit représente la valeur maximale de la clé, ainsi, la valeur de la clé max de la case0 = c (si une clé a une valeur inférieur ou égale à c, elle va être insérée dans la case0), pour la case2 c'est x, …..

➲ **Recherche**

En effectuant un parcours séquentiel de l'arbre, on essaye d'identifier la case contenant la clé recherchée, ie : la case dont l'intervalle de clés englobe la clé à trouver.

Une fois la case trouvée, on vérifie si elle contient la clé recherchée ou non.

➲ **Recherche par intervalle**

Il s'agit de trouver tous les enregistrements dont la clé appartient à l'intervalle de la requête $]C_{Min}, C_{max}]$. L'arbre est parcouru séquentiellement à la recherche d'une case contenant C_{Min} puis poursuivre la recherche dans l'ordre jusqu'à la rencontre d'une case dont la clé maximale est supérieur à C_{Max}.

➲ **Insertion**

Avant d'insérer un nouvel enregistrement, le processus de recherche est lancé.

- Si la clé de cet enregistrement se trouve dans l'une des cases alors rien à faire.
- Si la clé n'existe dans aucune case, alors elle va être insérée dans la case où l'intervalle de clés englobe la valeur de cette clé. Si la capacité b de la case est dépassée alors exécuter le processus d'éclatement de la case.

➲ **Suppression et fusion**

Pour supprimer un enregistrement de clé C, il faut tout d'abord localiser la case qui le contient. Une fois la clé supprimée, on prévoit la fusion de 2 cases. La fusion est possible entre 2 cases sœurs et dont le nombre des enregistrements des 2 cases est inférieur ou égal à la capacité b.

La fusion implique donc la suppression d'une case et la diminution de la taille de l'arbre par la suppression d'un nœud interne et externe.

Aussi, la dernière case du fichier sera mutée à la place de la case supprimée.

III.5.3. CTH* : Hachage Digital Compact Distribué [66]

Le CTH* est une adaptation du CTH aux environnements distribués. Il a été proposé par le professeur Zegour en 2004. Il repose sur le même principe que le CTH sauf que les cases sont réparties sur des serveurs.

CTH* est une SDDS qui repose sur une architecture client-serveur dont le nombre de cases est théoriquement infini.

Chaque serveur possède un arbre digital partiel qui représente les éclatements provoqués sur ce serveur, un intervalle de clés ordonnées et une case contenant les données.

Initialement, on a un seul serveur (0) avec l'arbre |0 et une case vide, toute insertion est dirigée vers ce serveur.

Suite aux éclatements, le fichier s'étend vers d'autres serveurs, les données seront reparties comme dans le CTH, l'arbre digital au niveau du serveur éclaté est étendu pour garder la trace de tous les éclatements sur ce serveur, alors que le nouveau serveur est initialisé avec un arbre vide "CMax[0] n", où CMax est sa clé maximale, n est son numéro logique qui le différencie des autres serveurs.

Un client a son propre arbre digital partiel initialisé à |0 qui ne correspond pas toujours à l'arbre d'un serveur destinataire d'une requête. L'arbre du client sera mise à jour suite à la réception des messages d'ajustement d'image IAM envoyés par des serveurs lors des erreurs d'adressage.

Un coordinateur est un élément principal du CTH*. C'est un site spécialisé dans la gestion des éclatements. Il comporte à son niveau 2 tables : une table d'allocation dynamique PAT (Physical Allocation Table) et une table d'allocation de fichier FAT (File Allocation Table).

Quand un nouveau serveur s'ajoute aux serveurs déjà existants, il envoie ses paramètres (adresse, espace disponible, …) au coordinateur pour l'ajouter au PAT. Cette table dynamique est utile pour la gestion des débordements et l'allocation des serveurs.

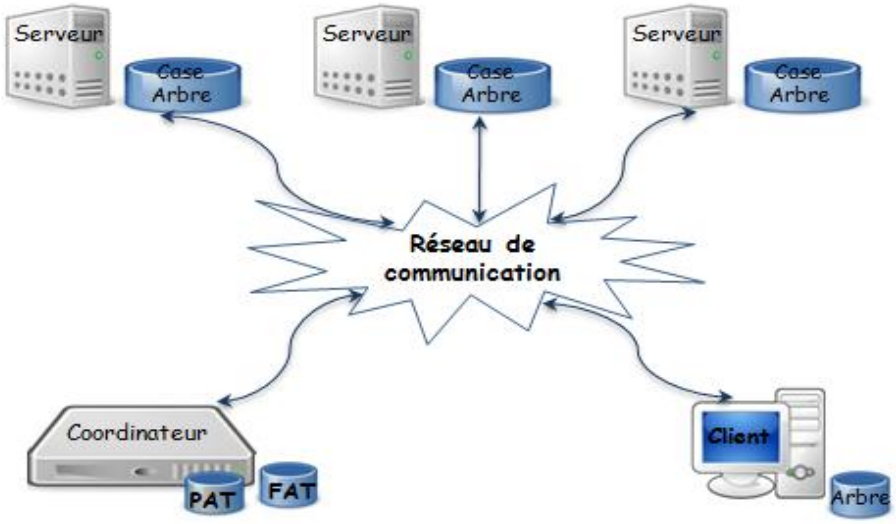
FAT est une table qui sert pour la gestion des opérations sur les fichiers, elle répertorie pour chaque fichier, son nom ainsi que ses propriétés. Chaque fichier a un nom unique pour l'identifier et qui a été donné par le coordinateur.

Dans le cas d'un débordement d'un case, le serveur en question (contenant la case débordée) peut envoyer un message en multicast aux autres serveurs pour solliciter un serveur disponible. Chaque serveur a son propre table PAT (contenant la liste des serveurs). Après l'éclatement de la case débordée, le serveur met à jour sa PAT. Le problème qui se pose dans ce cas est l'assurance de la cohérence entre l'état réel des serveurs et leur PAT.

Une autre solution de gestion de débordement, consiste à contacter le coordinateur qui est le seul à maintenir une table contenant l'état de tous les serveurs (PAT). Si le

coordinateur trouve dans la PAT un serveur disponible, il lui affecte un numéro logique, récupère ses paramètres et les envoie au serveur en débordement qui exécute le processus d'éclatement CTH.

S'il ne trouve pas de serveur disponible, il envoie au serveur en débordement un message lui indiquant qu'aucun serveur n'est disponible. La clé ne sera pas donc insérée et le client est prévenu de l'échec de l'opération.



FigureIII.10: Architecture d'un système basé sur CTH*

➲ **Eclatement d'une case et expansion de l'arbre digital**

Si un serveur S déborde, il informe le coordinateur qui se chargera de trouver un serveur disponible S'. La case de S est éclatée selon l'algorithme d'éclatement CTH, les clés seront divisées entre la case débordée et la nouvelle case S'.

L'arbre de S' sera initialisée avec les digits de la clé Max (CMax = C1, C2, …. Ck) de S suivi de S' et de k-1 nœuds Nil.

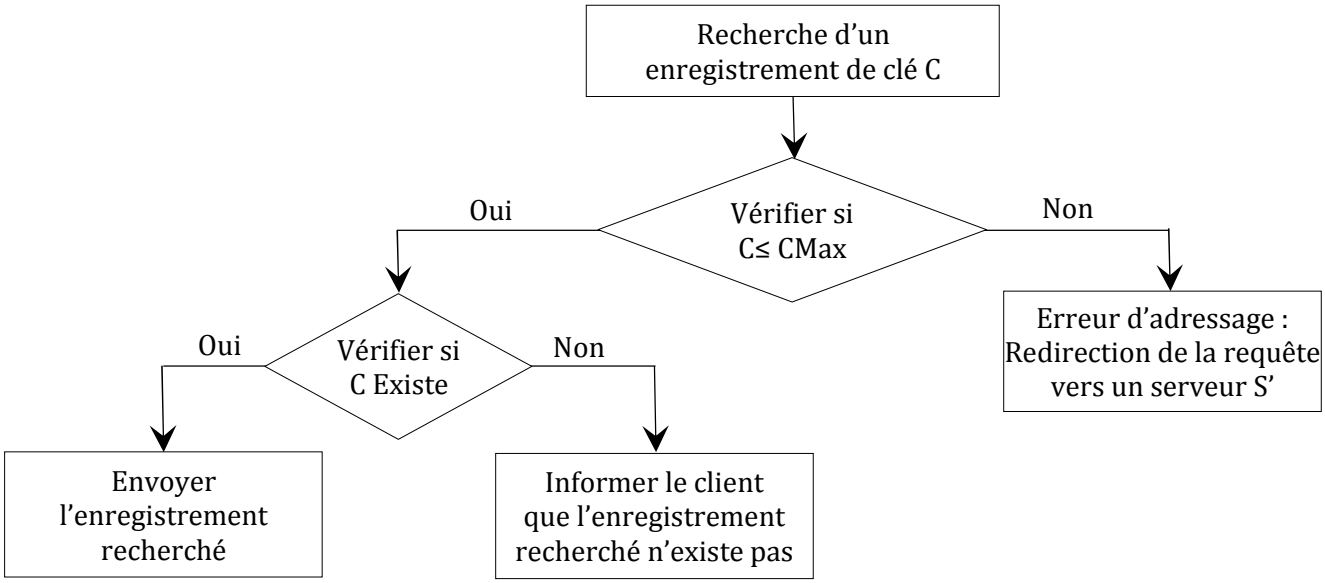
S'il n y a pas de serveur disponible, alors échec de l'opération et l'éclatement ne peut pas être effectué.

➲ **Recherche**

Pour trouver le serveur supposé contenir un enregistrement de clé C, le client consulte son arbre digital selon l'algorithme de recherche CTH. Il adresse alors sa requête au serveur trouvé.

En recevant la requête, le serveur vérifie si C est inférieure à sa clé Max. Si c'est le cas, alors il traite la requête et envoie l'enregistrement recherché au client s'il le trouve, sinon, il lui informe que l'enregistrement recherché n'existe pas.

Si la clé est supérieure à la clé Max du serveur, alors il s'agit d'une erreur d'adressage. Le serveur effectue une recherche dans son arbre digital selon l'algorithme CTH pour trouver le serveur qui peut contenir C et lui redirige la requête en lui indiquant l'adresse du client. En même temps, il envoie un IAM au client pour corriger son image.



FigureIII.11 : Processus de recherche d'un enregistrement

➲ **Recherche par intervalle**

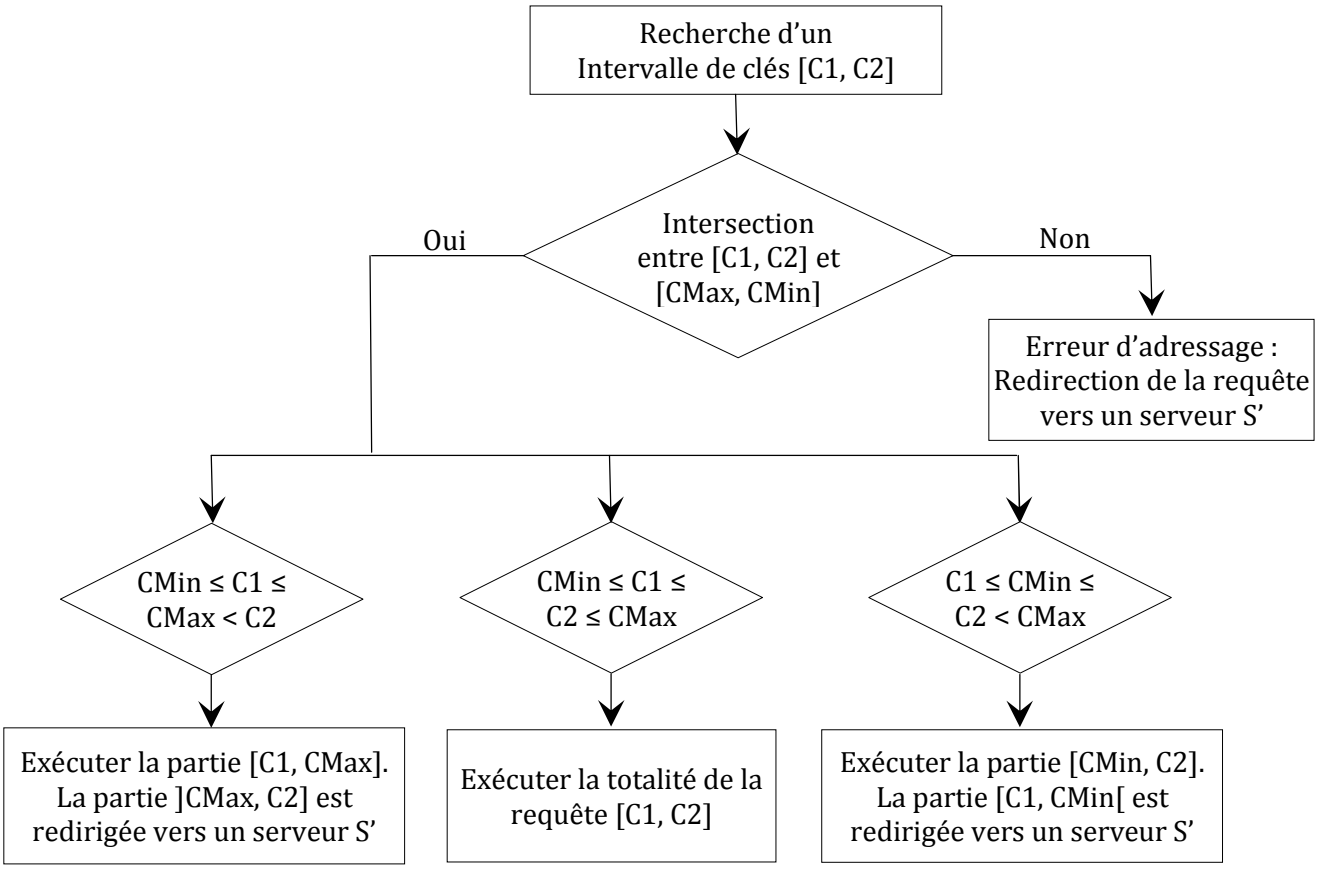
Dans ce type de requête, il s'agit de trouver un ensemble de clés appartenant à un intervalle [C1, C2]. Le client recherche dans son arbre digital tous les serveurs susceptibles contenir une des clés, ie : au moins une clé de l'intervalle [C1, C2] est inférieure la clé Max du serveur.

Un serveur a un intervalle de clé [CMin, CMax]. A la réception de la requête plusieurs cas sont observés :

- S'il n y a aucune intersection entre l'intervalle de la requête et celui su serveur, alors c'est une erreur d'adressage. La requête va être redirigée vers un autre serveur avec l'envoi d'un IAM au client.
- Si l'intervalle de la requête est inclus dans l'intervalle du serveur, alors la requête est traitée et la réponse sera envoyée au client.
- Si une partie de l'intervalle de la requête est incluse dans l'intervalle du serveur, alors le serveur traite la partie en intersection avec son intervalle. Il y a deux cas possible :
  - Si C1 est incluse dans l'intervalle [CMin, CMax] et C2 est supérieure à CMax (Cmin ≤ C1 ≤ CMax < C2), alors traiter la partie [C1, CMax]. Le reste de l'intervalle de la

requête]CMax, C2] va être redirigé vers un autre serveur avec l'envoi d'un IAM au client.

- Si C2 est incluse dans l'intervalle [CMin, CMax] et C1 est inférieure à CMin (C1 ≤ Cmin ≤ C2 < CMax), alors traiter la partie [CMin, C2]. Le reste de l'intervalle de la requête [C1, CMin[va être redirigée vers un autre serveur avec l'envoi d'un IAM au client.



FigureIII.12 : Processus de recherche d'un intervalle de clés

➲ Insertion

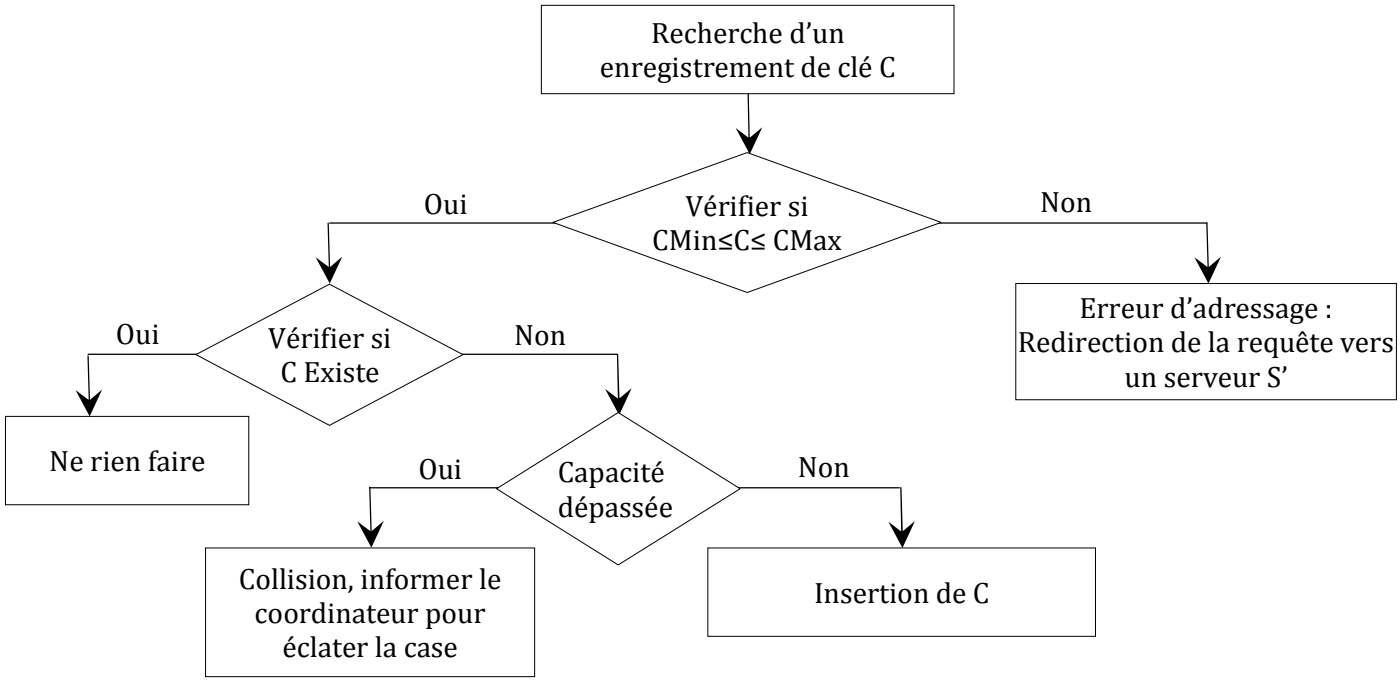
Le client applique l'algorithme de recherche CTH sur son arbre digital pour trouver le serveur qui peut contenir un enregistrement de clé C (C doit être inférieure à la clé Max du serveur).

A la réception de la requête, le serveur vérifie si C est incluse dans son intervalle de clés, ie : CMin ≤ C ≤ CMax, si tout est en ordre, alors :

- Si la clé existe déjà alors ne rien faire.
- Si la clé n'existe pas et qu'il y a assez d'espace (le nombre d'enregistrement déjà contenu dans le serveur est inférieur à b), alors il insère C et met à jour son arbre.
- Si la clé n'existe pas et la capacité est dépassée (il ya une collision), alors le

processus d'éclatement CTH* est exécuté.

Si C n'appartient pas à l'intervalle de clés du serveur, alors il applique l'algorithme de recherche CTH sur son arbre pour trouver le serveur adéquat (dont C est inférieure à sa clé Max) et lui redirige la requête avec l'envoi d'un IAM au client.



FigureIII.13 : Processus d'insertion d'un enregistrement

➲ **Suppression et fusion**

Comme on a vu précédemment (processus d'éclatement), le client envoie sa requête au serveur après avoir récupérer son adresse.

Le serveur recherche la clé :

- S'il ne la trouve pas alors c'est une erreur d'adressage, il recherche le serveur adéquat et lui renvoie la requête avec un IAM au client.
- Si la clé existe alors il la supprime et met à jour son arbre.

La suppression d'un enregistrement peut provoquer une fusion (comme dans le CTH), si le nombre d'enregistrement des deux serveurs est inférieur à b.

➲ **Ajustement de l'image du client**

Soit CMax la clé Max actuelle du serveur S, C'Max la clé Max du serveur S selon l'image du client (C'Max est l'ancienne clé Max de S et qui est devenue la clé Max du nouveau serveur S'' ajouté après l'éclatement). Une partie de l'arbre de S (premiers digit) est déjà incluse dans l'arbre du client, Préfixe représente cette partie commune.

- *Côté serveur :*

Le serveur S recherche dans son arbre un serveur S' qui a comme clé Max C'Max ainsi que le serveur qui le précède S'' (S peut être lui-même S'').

Pour que le client ajuste son image avec les vraies valeurs, le serveur S envoie au client le serveur S', la partie de l'arbre CPrefixe, ….. Ck, S'' (l'ancienne arbre dépourvu *de ses N premiers digits* ou Prefixe).

- *Côté client :*

On a : CMax = C'Max, CPrefixe, … Ck

Le client recherche dans son arbre la première branche dont sa clé Max CM est supérieure à CMax. Il y a une partie commune entre CMax et CM représentée par Prefixe'.

Le client remplace une séquence de son arbre digital CPrefixe', … S par la séquence de l'arbre du serveur CPrefixe+1, … Ck S S'.

III.6. Conclusion

Au cours de ce chapitre, nous avons vu une nouvelle classe de données : SDDS. C'est une structure de données qui permet de répartir un fichier sur différentes sites serveurs.

Selon la distribution des données, trois classes de SDDS ont été présentées : des SDDS dont la distribution de données est basée sur le hachage linéaire, celles basées sur la distribution par intervalle et une dernière SDDS qui est le hachage digital.

Bien que les SDDS par hachage ont montré leur efficacité, leur facilité de manipulation, elle pose en outre le problème de la non préservation de l'ordre des données surtout quand il s'agit d'exécuter des requêtes par intervalle.

Les SDDS par intervalle (dont la structure utilisée pour la représentation de données est l'arbre) règle le problème posé par les SDDS par hachage au profit d'une complexité de manipulation et de temps d'accès accrus.

Les SDDS basées sur le hachage digital tirent profit des 2 classes précédentes, elles utilisent un arbre digital pour représenter les données tout en préservant leur ordre.

PARTIE II

CONCEPTION ET IMPLEMENTATION D'UN ENTREPOT DE DONNEES CLASSIQUE ET EN SDDS DW_SDDS

Chapitre IV

Conception d'un entrepôt de données classique et en SDDS DW_SDDS

IV.1. Introduction

Après avoir exploré en profondeur les entrepôts de données et les SDDS, ce chapitre se consacre à la conception d'un entrepôt de données classique ainsi qu'un entrepôt de données reposant sur une SDDS.

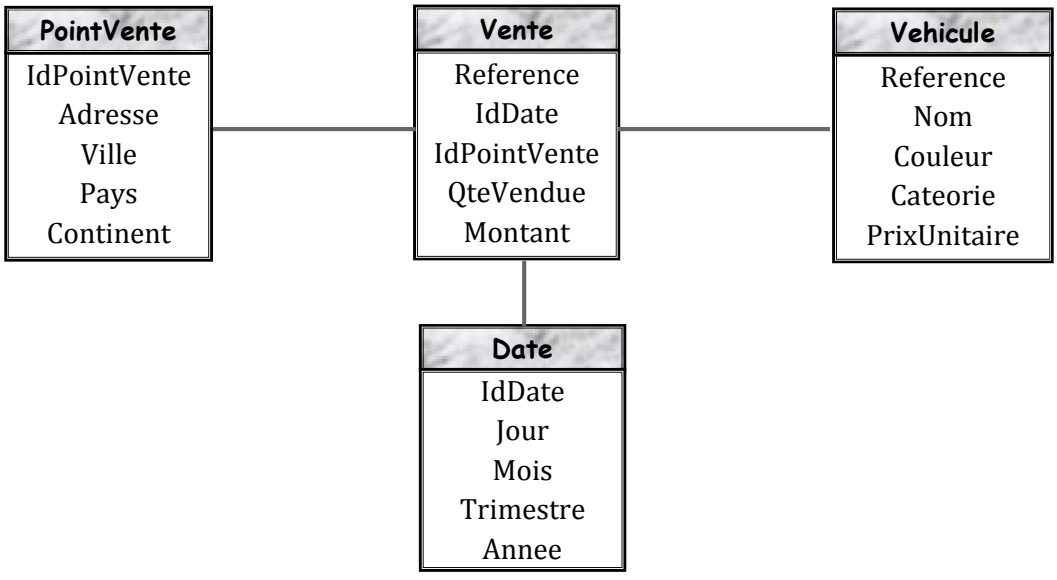
Nous commencerons par présenter le modèle de données adopté, à savoir le schéma en étoile. Par la suite, nous expliquerons en détail la conception des deux types d'entrepôts, en mettant en lumière leur architecture, leurs composants ainsi que les principes de fonctionnement spécifiques à chaque approche. Nous décrirons également le traitement des différentes requêtes pouvant être appliquées sur un entrepôt de données, en nous focalisant sur les celles de recherche et d'ajout, tout en détaillant leur gestion au sein de chaque système.

IV.2. Conception de l'entrepôt de données

Pour la conception de l'entrepôt de données, nous optons le modèle en étoile, caractérisé par un ensemble de tables de dimension reliées à une table de fait centrale. Ce modèle se distingue par sa structure claire, directe et facile à interpréter. Aussi, son principal avantage réside dans la réduction du temps de réponse aux requêtes, grâce au nombre limité de jointures nécessaires.

La table de fait est représentée par des colonnes qui sont les mesures de l'activité, et les clés étrangères référençant les tables de dimension. Les clés étrangères servent pour la jointure entre la table de fait et les tables de dimensions.

Le modèle de données que nous proposons est illustré selon le schéma en étoile suivant:



FigureIV.1 : Schéma en étoile

Le domaine d'application choisi est l'industrie automobile, en raison de la grande quantité de données générées et la nécessité de les répartir sur plusieurs serveurs pour une meilleure gestion.

La table de fait « *Vente* » représente les ventes enregistrées dans une compagnie de fabrication d'automobile. Elle est reliée aux tables de dimension « *Date*, *PointVente* et *Véhicule* » par les clés étrangères. Les mesures *QteVendue* et *Montant* représentent le nombre de véhicules vendus et le montant d'une opération de vente.

Les tables de dimension *Date*, *PointVente* et *Véhicule* représentent respectivement :

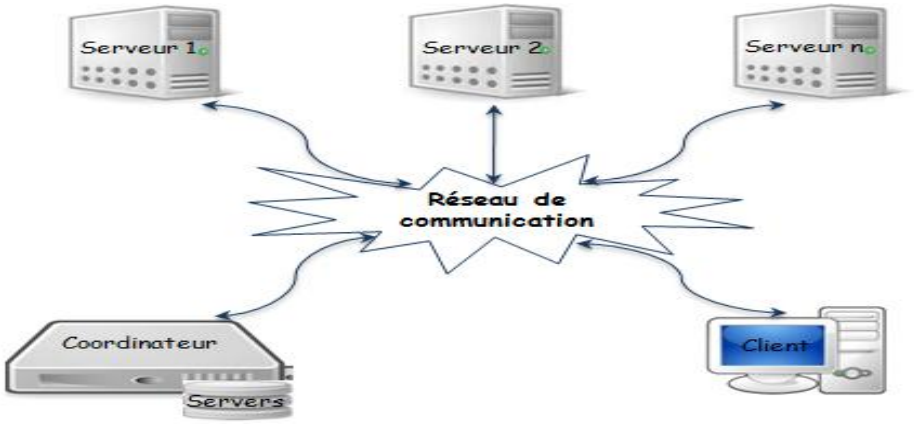
- La date d'une vente (incluant le jour, le mois, l'année et le trimestre).
- Les attributs d'un client (omprenant son code, son nom, son adresse, ainsi que sa ville, son pays et son continent).
- Les propriétés d'un véhicule (sa référence, son nom, sa couleur, sa catégorie et son prix d'unité).

IV.3. Architecture d'un entrepôt de données classique

IV.3.1. Les composants d'un entrepôt de données classique

L'entrepôt va être implémenté dans une architecture client/ serveur avec la présence d'un site central : le coordinateur.

Le client soumet une requête, et le serveur y répond en conséquence.



FigureIV.2 : Architecture globale d'un entrepôt de données classique

➲ **Le client**

C'est un site qui peut formuler des requêtes d'ajout ou de recherche et par conséquence, il va recevoir des réponses à ses requêtes (sous forme de message de confirmation d'ajout ou d'un ensemble d'enregistrements pour une requête de recherche).

Le client ne communique pas directement avec un serveur mais passe par l'intermédiaire du coordinateur. Ainsi, la localisation des serveurs est transparente pour le client.

➲ **Le serveur**

C'est un site où sont stockés les différents enregistrements de l'entrepôt. Initialement, un seul serveur est disponible. À mesure que de nouveaux enregistrements sont ajoutés et que la capacité de stockage du serveur atteint ses limites, des serveurs supplémentaires seront intégrés pour héberger ces nouveaux enregistrements.

Chaque serveur est identifié par :

- Un numéro logique. C'est un identifiant qui le différencie des autres serveurs.
- Une adresse pour le localiser.
- Le numéro du port de communication, utilisé pour établir la connexion entre les clients et le serveur.
- Sa capacité totale de stockage, qui définit la quantité maximale de données qu'il peut héberger.
- Espace de stockage disponible, indiquant la capacité restante pour ajouter de nouveaux enregistrements.
- Un indicateur pour spécifier si c'est le serveur actif.
- Et aussi un autre indicateur désignant le serveur destiné à héberger les nouveaux enregistrements lorsque le serveur actif atteint sa capacité maximale.

➲ **Le coordinateur**

C'est un site intermédiaire qui a pour rôle d'acheminer une requête d'un client vers le serveur ou sont stockées les données, et inversement, envoie les réponses reçues des différents serveurs vers le client.

Le coordinateur dispose à son niveau des tables de dimension : *Date*, *PointVente* et *Véhicule*. En plus d'une table *Servers* contenant la liste des serveurs hébergeant les données avec leurs paramètres essentiels (numéro, adresse, numéro du port de communication, espace de stockage disponible et total).

Cette table est essentielle pour l'allocation des serveurs et la gestion des répartitions des données.

Quand un nouveau serveur est intégré en raison de la saturation des serveurs existants, ce nouveau serveur va être ajouté à la liste des serveurs participant en envoyant ses paramètres au coordinateur qui les enregistre dans la table *Servers*.

Il existe deux types de requête possibles:

- Requête portant sur la table de fait et / ou les tables de dimension.
- Requête portant sur les tables de dimension.

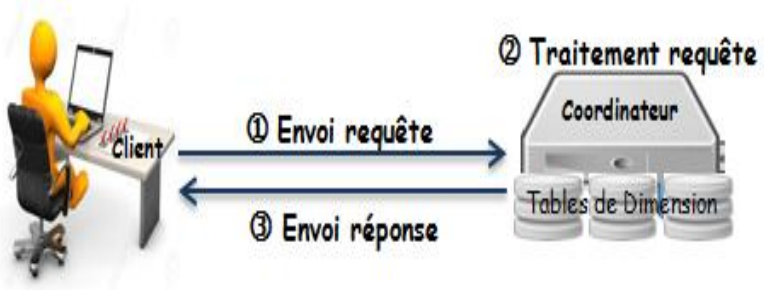
Pour chacune des requêtes, il peut s'agir d'une requête de recherche (de consultation) ou bien d'une requête d'ajout (d'insertion).

Dans notre étude, l'accent est mis sur l'accès à la table de fait, plutôt qu'aux tables de dimension, qui est fréquemment sollicitée, notamment pour des opérations d'ajout et de consultation contrairement aux tables de dimension, rarement modifiées. Ces dernières seront dupliquées sur chaque serveur, nécessitant ainsi un mécanisme de gestion de la cohérence entre les différentes copies.

La table de fait sera répartie sur plusieurs serveurs à l'aide de l'algorithme LH*. Initialement, un seul serveur, S0, est utilisé pour stocker un nombre limité d'enregistrements de la table de fait, organisés en case. Lorsque la capacité maximale de la case est atteinte et que des débordements surviennent, un éclatement de la case est déclenché, entraînant l'ajout d'un nouveau serveur, S1. Conformément à l'algorithme LH*, la moitié des enregistrements est alors déplacée vers ce serveur S1.

IV.3.2. Requête d'ajout ou de recherche sur une table de dimension

Lorsque l'utilisateur envoie sa requête, le coordinateur prend en charge son exécution si elle concerne uniquement une table de dimension. Dans ce cas, il renverra une réponse sous forme de message de confirmation pour une requête d'ajout, ou un ensemble d'enregistrements pour une requête de recherche.



FigureIV.3 : Exécution d'une requête sur les tables de dimension

IV.3.3. Requête d'ajout sur la table de fait

Pour ajouter de nouveaux enregistrements, le client envoie sa requête d'insertion au coordinateur. Ce dernier consulte la table *Servers* pour extraire les informations concernant le serveur actif (adresse, numéro de port, espace disponible), c'est-à-dire : celui qui va recevoir les nouveaux enregistrements. Dans certains cas, il extrait également les informations concernant le serveur suivant, c'est-à-dire : celui qui deviendra actif après la saturation du serveur actif.

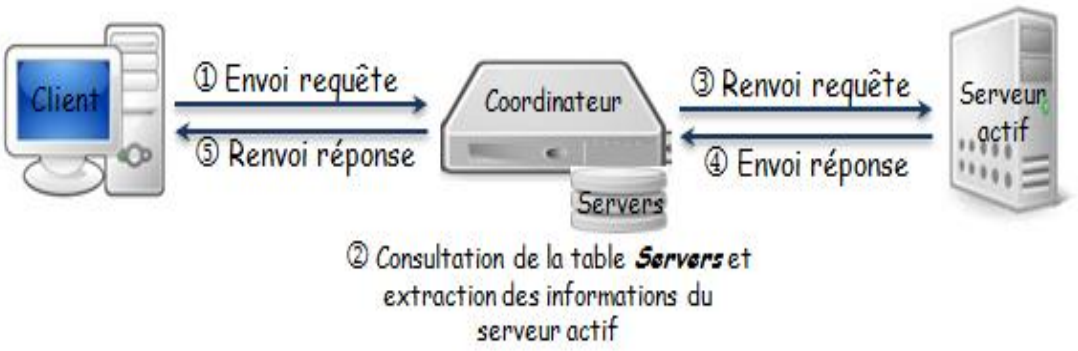
Selon l'espace de stockage disponible du serveur actif, il y peut y avoir deux cas possibles :

i) L'espace disponible est supérieur au nombre des enregistrements

Le coordinateur transmet la requête d'insertion au serveur actif. Les nouveaux enregistrements vont être ajoutés au niveau de ce serveur, qui envoie un message de confirmation d'ajout au coordinateur, lequel transmet ensuite cette confirmation au client.

Par la suite, le coordinateur met à jour l'espace disponible du serveur actif.

Espace disponible (nouveau) =
Espace disponible (ancien) – Nombre des enregistrements insérés



FigureIV.4 : Exécution d'une requête d'ajout (i)

ii) L'espace disponible est inférieur au nombre des enregistrements

L'espace disponible du serveur actif n'est pas assez suffisant pour ajouter tous les nouveaux enregistrements.

Le coordinateur décompose la requête, envoie un ensemble d'enregistrements au serveur actif (le nombre des enregistrements envoyés égale l'espace disponible du serveur actif). Le reste des enregistrements va être envoyé vers le serveur suivant.

Aussi, l'état actif est attribué au deuxième serveur, l'état suivant à un autre serveur.

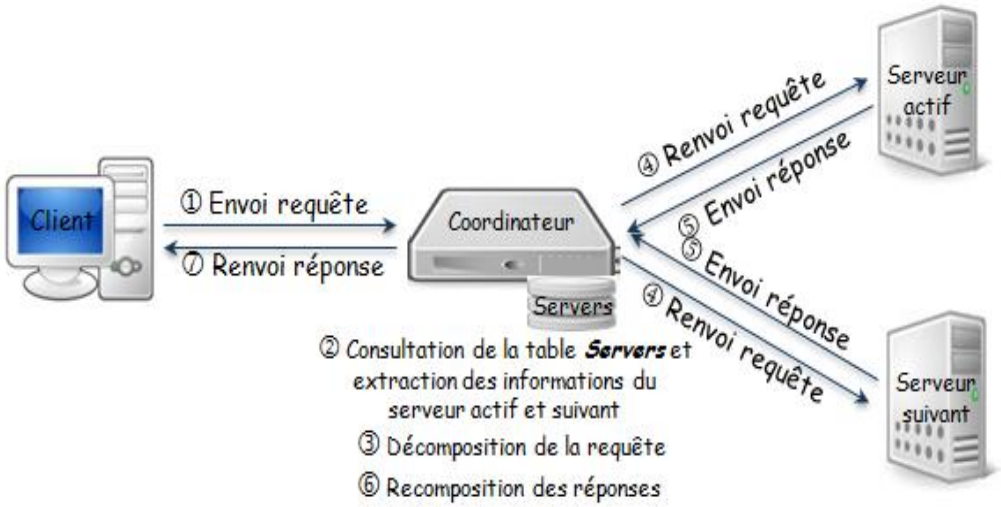
L'espace disponible des deux serveurs est mis à jour.

L'espace disponible de l'ancien serveur actif est mis à 0 :

Espace disponible (nouveau) = ***Espace disponible (ancien) – Nombre des enregistrements ajoutés = 0***

Pour le nouveau serveur actif :

Espace disponible (nouveau) = ***Espace disponible (ancien) – Nombre des enregistrements ajoutés***

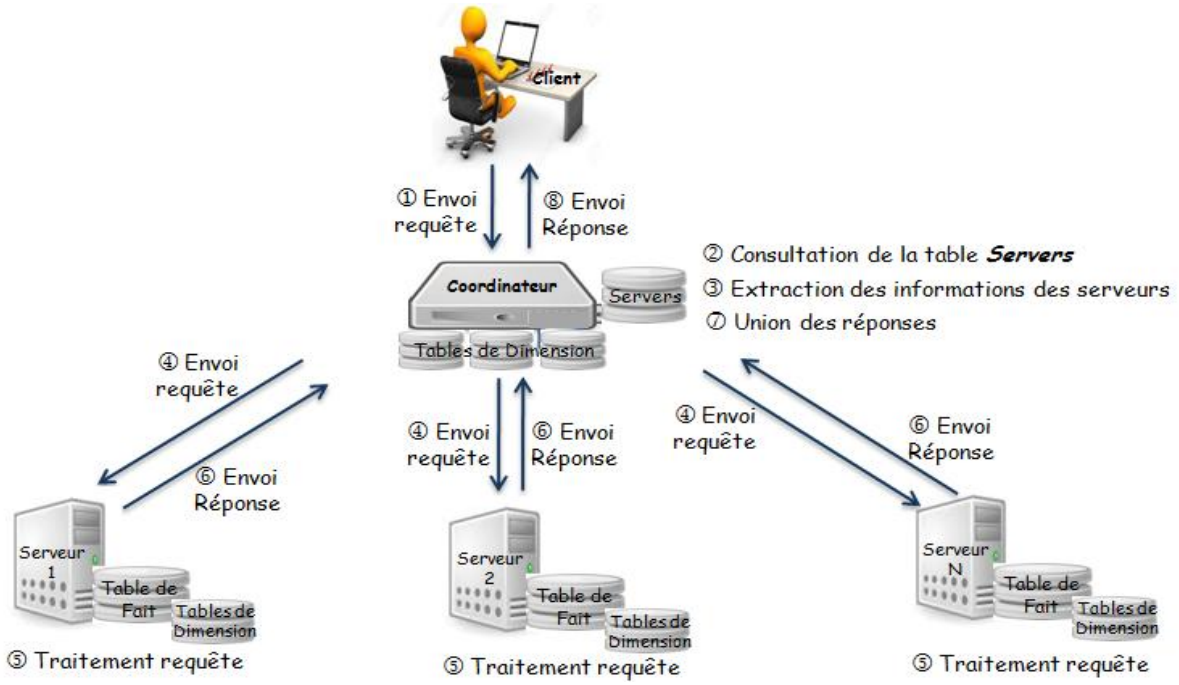


FigureIV.5 : Exécution d'une requête d'ajout (ii)

IV.3.4. Requête de recherche sur la table de fait

Le client envoie sa requête de recherche au coordinateur. Le coordinateur extrait les informations des serveurs de la table *Servers* pour leur renvoyer la requête en multicast.

A la réception des réponses, le coordinateur les recompose par une opération d'union et les envoie au client.



FigureIV.6 : Exécution d'une requête de recherche

IV.4. Architecture d'un entrepôt de données en SDDS (DW_SDDS)

IV.4.1. Les composants d'un DW_SDDS

Dans un système SDDS, l'architecture reste similaire à celle d'un entrepôt classique, comprenant client, serveur(s) et coordinateur. Toutefois, la différence majeure existante concerne le rôle du coordinateur.

➲ **Le client**

Le site client envoie des requêtes d'ajout ou de recherche à un serveur, qui va lui répondre par un message de confirmation d'ajout pour une requête d'insertion ou d'un ensemble d'enregistrements pour une requête de recherche.

Le client communique directement avec un serveur. Il dispose à son niveau des informations nécessaires pour établir la connexion avec le serveur actif (adresse, numéro du port de communication). Ces informations constituent ce qu'on appelle l'image du client, lui permettant d'interagir de manière efficace avec le serveur.

➲ **Le serveur**

Dans un système DW_SDDS, il y a, au départ, un seul site serveur capable de contenir un nombre limité d'enregistrements de la table de fait organisés sous forme d'une case. Chaque case peut stocker jusqu'à b enregistrements.

L'avantage d'un système DW_SDDS réside dans la capacité à distribuer dynamiquement un fichier sur plusieurs serveurs à mesure qu'il augmente en taille. Ainsi, de nouveaux serveurs sont ajoutés pour héberger les enregistrements supplémentaires. Ces derniers sont alors redistribués ou répartis sur les nouveaux serveurs en appliquant un algorithme LH*, garantissant une gestion optimale de la répartition des données.

Chaque serveur est identifié par :

- Un numéro : un identifiant unique qui permet de distinguer ce serveur des autres.
- Une adresse pour le localiser.
- Le numéro du port de communication, utilisé pour établir la connexion entre les clients et le serveur.
- Sa capacité totale de stockage : détermine la quantité maximale de données qu'il peut accueillir.
- Espace de stockage disponible : indique la capacité restante pour ajouter de nouveaux enregistrements.
- Un indicateur spécifie si le serveur est actuellement actif.
- Et aussi un autre indicateur pour désigner le serveur qui prendra en charge l'hébergement des nouveaux enregistrements lorsque le serveur actif atteint sa capacité maximale.

➲ **Le coordinateur**

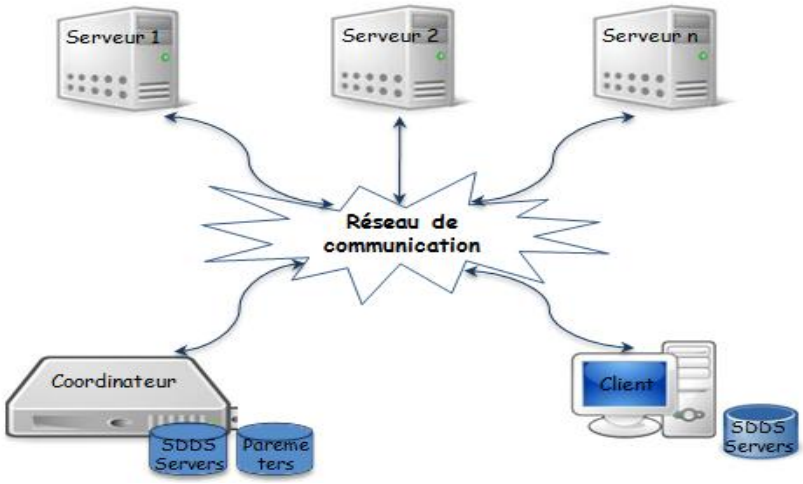
Dans un système DW_SDDS, le coordinateur n'est sollicité qu'on cas de débordement [56], c'est-à-dire: quand le nombre des nouveaux enregistrements à ajouter au niveau du serveur actif est supérieur à son espace disponible. Le coordinateur n'interfère jamais dans le processus d'évaluation des requêtes [46].

Le coordinateur dispose à son niveau d'une table *SDDSServers* contenant la liste des serveurs hébergeant les données avec leurs paramètres essentiels (numéro, adresse, numéro du port de communication, espace total et espace disponible).

Le coordinateur a également à son niveau une autre table *Parameters* regroupant des informations (i, n et N) servant pour le calcul des adresses des enregistrements lors des débordements afin de les répartir vers le serveur correspondant.

- N : représente le nombre de serveurs, initialisé à 1.
- i : représente le niveau du fichier, i = 0, 1, 2, 3, ….. Le i s'incrémente de 1 à chaque fois que la taille du fichier double.

Le coordinateur est le seul à disposer d'une image correcte du fichier. Son rôle consiste à assurer la cohérence entre les paramètres des clients et des serveurs, ainsi qu'à coordonner les éclatements des cases.



FigureIV.7 : Architecture globale d'un entrepôt DW_SDDS

IV.4.2. Requête d'ajout sur la table de fait

Pour l'ajout d'un ou de plusieurs enregistrements, le client, disposant de sa propre image du système, extrait les paramètres du serveur actif. La communication s'établit directement entre le client et le serveur sans l'intervention du coordinateur.

Si aucune erreur d'adressage n'est détectée, c'est-à-dire: le serveur communiqué est effectivement le serveur actif, deux situations peuvent se présenter :

i) L'espace de stockage disponible du serveur actif est supérieur au nombre des enregistrements.

ii) L'espace de stockage disponible du serveur actif est inférieur au nombre des enregistrements.

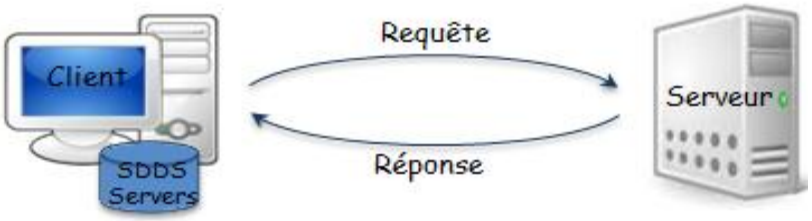
Notez qu'une erreur d'adressage survient lorsque les informations du serveur actif détenues par le client ne sont pas à jour, ainsi les valeurs réelles du serveur actif diffèrent de celles connues par le client, entraînant l'envoi de la requête vers un serveur incorrect.

Ces informations se changent lors des débordements où l'état de serveur actif est transféré d'un serveur à un autre. Il revient alors au coordinateur d'envoyer un message

relatif au client contenant les nouveaux paramètres du serveur actif afin que le client actualise sa propre image.

i) L'espace de stockage disponible du serveur actif est supérieur au nombre des enregistrements.

Le client transmet directement une requête d'ajout de nouveaux enregistrements au serveur actif. Celui-ci procède à l'insertion des enregistrements et retourne ensuite un message de confirmation d'ajout au client.



FigureIV.8 : Exécution d'une requête d'ajout (i)

ii) L'espace de stockage disponible du serveur actif est inférieur au nombre des enregistrements.

Le client envoie une requête d'ajout au serveur actif.

A la réception de cette requête, le serveur vérifie si l'espace disponible est suffisant pour insérer les nouveaux enregistrements.

Si le nombre des enregistrements dépasse l'espace disponible, un débordement se produit : le serveur actif contacte alors le coordinateur en lui adressant un message informatif. Le coordinateur, à son tour, consulte sa table *SDDSServers* pour extraire l'adresse du serveur suivant et lui envoie une demande d'éclatement.

Chaque enregistrement possède une clé utilisée pour calculer son adresse. Lorsqu'un débordement survient, les adresses de tous les enregistrements (nouveaux et déjà présents sur le serveur suivant) sont recalculées à l'aide de la fonction de hachage h_i. Ensuite, la moitié des enregistrements est transférée vers un nouveau serveur afin de rééquilibrer la charge.

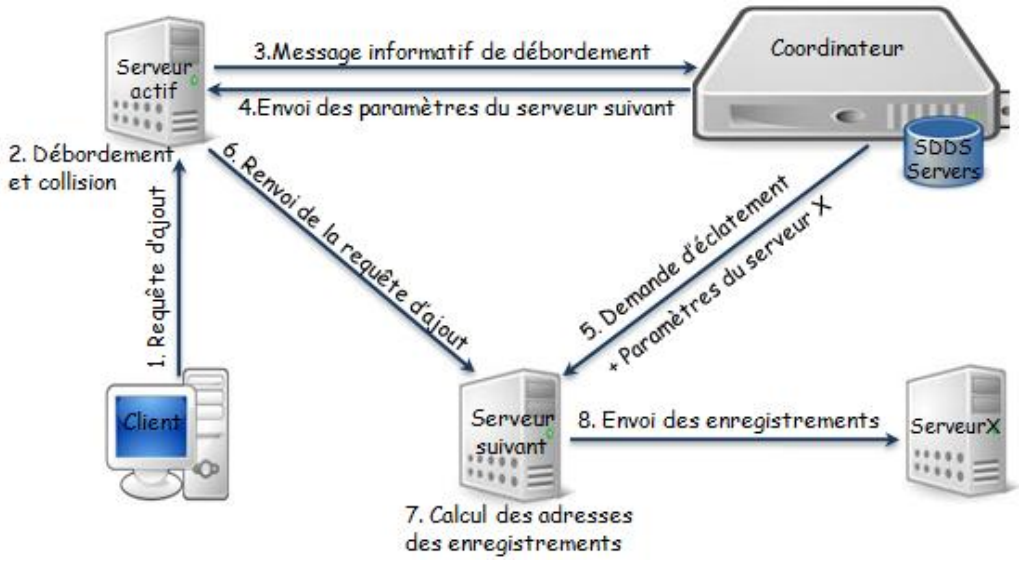
$$h_i = C \bmod N * 2^i$$

C représente la clé de l'enregistrement. La nouvelle adresse de l'enregistrement est calculée comme suit :

```
a ←hi (C)
Si  a < n  Alors  //envoie vers le nouveau serveur
    a ←hi+1 (C)
FinSi
```

AlgorithmeIV.1 : Calcul de l'adresse d'un enregistrement

n représente le numéro du prochain serveur à éclater, c'est-à-dire : le serveur suivant devant accueillir une partie des enregistrements lors d'un débordement.



FigureIV.9 : Débordement et éclatement d'un serveur

Une fois tous les enregistrements ajoutés, que ce soit sur le serveur suivant ou le nouveau serveur, un message de confirmation d'ajout sera envoyé au client.

Le coordinateur procède également à la mise jour des paramètres : le serveur actif, le serveur suivant pointé par n, ainsi que le niveau du fichier i.

Le nouveau serveur deviendra le serveur actif. Le serveur suivant est déterminé en appliquant l'algorithme suivant :

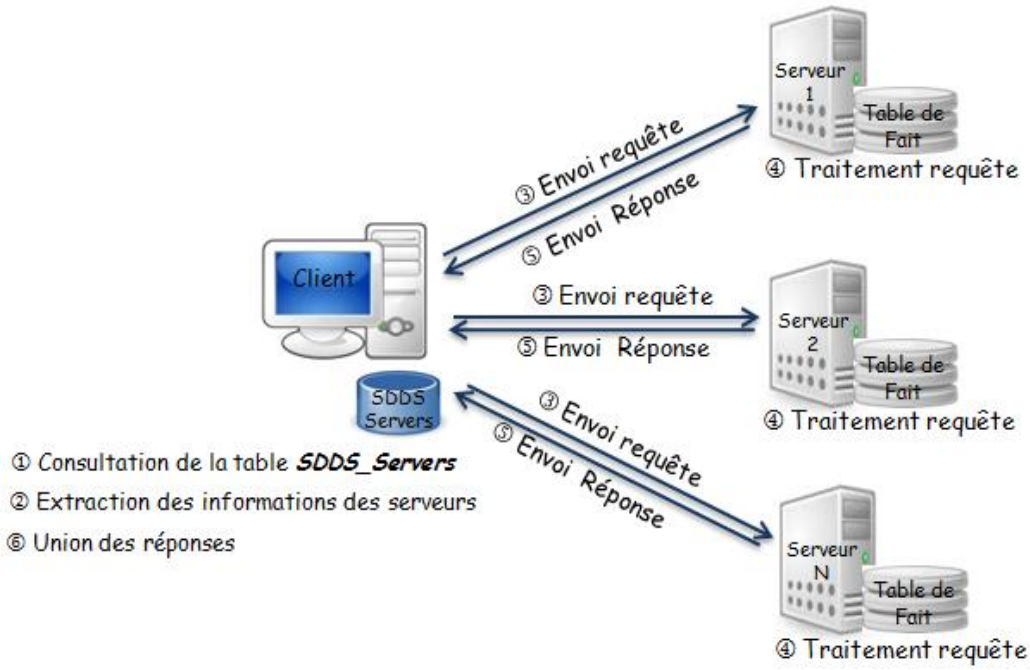
```
n ← n +1
Si  n >=2i  Alors
    n ←0
    i ← i+1
FinSi
```

AlgorithmeIV.2 : Mise à jour de i et n après l'éclatement d'un serveur

IV.4.3. Requête de recherche sur la table de fait

Si la rechercher se porte sur un ensemble d'enregistrements, le client envoie sa requête à l'ensemble des serveurs par multicast après avoir extrait leurs paramètres depuis sa table locale.

Les serveurs répondent ensuite au client, qui recompose les résultats reçus par une opération d'union avant de les afficher.



FigureIV.10 : Exécution d'une requête de recherche d'un ensemble d'enregistrement

Si le client recherche un seul enregistrement, il calcule son adresse à l'aide de la fonction de hachage afin de déterminer le serveur hébergeant cet enregistrement. La requête est alors adressée directement au serveur désigné.

En cas d'erreur d'adressage, le serveur contacté redirige la requête vers le coordinateur, qui se charge de l'acheminer vers le serveur approprié.

IV.5. Conclusion

En conclusion, ce chapitre a présenté en détail la conception des deux systèmes d'entrepôt de données : un entrepôt classique et un DW_SDDS.

Nous avons abordé les principes de conception de chaque système, en mettant en évidence le choix du modèle et le processus de gestion données. Aussi, nous avons expliqué le déroulement des communications entre les différents composants à savoir le client, le coordinateur et les serveurs.

Bien que cette section ne présente pas encore les résultats expérimentaux, elle jette les bases nécessaires pour l'implémentation et l'évaluation comparative ultérieure. Les prochaines étapes consisteront donc, à la mise en place réelle des entrepôts puis l'analyse des performances de ces deux architectures en termes de messages échangés et de temps d'exécution des requêtes, permettant ainsi de tirer des conclusions sur leur efficacité et performance.

Chapitre V

Implémentation d'un entrepôt de données classique et en SDDS DW_SDDS

V.1. Introduction

Alors que dans le chapitre précédent, la conception d'un entrepôt de données classique et d'un DW_SDDS basé sur les SDDS a été détaillé, le présent chapitre se concentre sur l'aspect implémentation des deux entrepôts, en explorant chaque étape, depuis la demande d'établissement d'une connexion initié par le client jusqu'à la réception des résultats de des requêtes formulées.

Ce chapitre vise à fournir une compréhension approfondie des aspects techniques associés à la mise en œuvre de ces deux types d'entrepôts de données. L'implémentation débute par démontrer le choix de Java comme langage de programmation, et de MySQL comme moteur de base de données pour la gestion des cubes ROLAP. Les raisons justifiant ces choix seront présentées dans les sections suivantes.

Une explication sera également apportée concernant le choix des sockets comme moyen de communication entre les différents composants du système, en mettant l'accent sur le processus d'établissement de la connexion initiée par le client. Cette étape est cruciale pour assurer une interaction fluide et efficace entre un processus client et un processus serveur, garantissant ainsi une transmission fiable des requêtes et des réponses.

Par la suite, nous décrirons en détail le processus d'exécution des requêtes. Ce processus englobe plusieurs étapes clés, depuis l'initiation d'une connexion par le client jusqu'à la réception des résultats.

Aussi, nous expliquerons les différentes connexions établies et les procédures exécutées par les divers composants du système, en mettant en évidence la coordination nécessaire entre les éléments du système pour assurer une communication fiable et efficace, ainsi que pour garantir une exécution transparente des requêtes pour l'utilisateur.

V.2. Le langage de programmation Java [21, 28, 30, 38, 65]

Java est un langage de programmation puissant et largement utilisé dans divers domaines, tels que :

- Les applications mobiles (surtout pour les systèmes androïd),
- Les jeux vidéo,
- Le développement des applications d'entreprise,
- La recherche scientifique,
- Les applications Web,

- La communication réseau,
- Les applications interagissant avec des bases de données.

Outre cette polyvalence, Java a été choisi comme langage de développement pour d'autres raisons, notamment :

- **Portabilité et indépendance vis-à-vis des plates-formes :** Le code Java est compilé en bytecode, un format intermédiaire qui peut être exécuté sur n'importe quelle plate-forme disposant d'une Java Virtual Machine (JVM). Cette particularité facilite le déploiement sur divers systèmes sans nécessiter d'adaptations.
- **Simplicité :** Java est connu pour sa syntaxe claire et sa facilité d'apprentissage, permettant même aux débutants de s'initier facilement au développement.
- **Programmation réseau avancée** : Java offre des bibliothèques intégrées pour la gestion des sockets, telles que *java.net.Socket* et *java.net.ServerSocket* pour la gestion des communications réseau. Ce qui est particulièrement utile dans notre travail pour établir la communication dans le système ROLAP distribué.
- **API et bibliothèques** : Java dispose d'une vaste collection d'API, de bibliothèques et frameworks open-source qui offrent un large éventail de fonctionnalités. Cela simplifie le développement des applications ainsi que l'exécution des tâches courantes, comme JDBC (Java DataBase Connectivity), qui facilite l'accès à une base de données.
- **Une communauté large et active** : Avec une communauté de développeurs mondiale, Java offre une abondance de ressources, de tutoriels et de forums pour aider à améliorer les compétences des développeurs, résoudre les problèmes et partager des connaissances.
- **Multithreading :** Java prend en charge le multithreading, permettant à plusieurs threads de s'exécuter simultanément au sein d'un même programme. Cette fonctionnalité est essentielle dans les environnements distribués pour effectuer plusieurs tâches en parallèle.
- **Efficacité du code source** : Grâce à ses bibliothèques de classes bien structurées et qui ne sont liées qu'à l'exécution, le code écrit en Java est concis. Cela permet de réduire la taille du pseudo-code tout en garantissant une exécution efficace.

V.3. Le moteur de bases de données MySQL [1, 65]

Bien qu'il existe une variété de systèmes de gestion de bases de données relationnelles (SGBDR), tels que PostgreSQL ou Microsoft SQL Server, chacun avec ses propres avantages, MySQL, l'un des plus populaires, est souvent préféré pour sa simplicité d'utilisation, sa fiabilité, son vaste soutien communautaire et ses performances dans des environnements web à grande échelle.

Notre choix a été porté sur MySQL comme moteur de base de données pour la gestion des cubes ROLAP grâce à ses nombreux atouts, parmi lesquels :

- **Gratuité et open-source :** MySQL est libre d'utilisation et modifiable par tout le monde. Cela permet aux développeurs de l'utiliser aussi bien pour des projets personnels que commerciaux sans avoir à dépenser de l'argent pour des licences coûteuses.
- **Fiabilité et maturité** : MySQL est l'un des systèmes de gestion de bases de données relationnelles les plus robustes et éprouvés.
- **Facilité d'utilisation** : Avec une interface intuitive et des commandes SQL simples, MySQL est accessible même pour les débutants, tout en restant performant pour les utilisateurs avancés.
- MySQL offre une **interface utilisateur intuitive** et une variété de commandes SQL simples pour gérer les bases de données. Cela le rend accessible aux débutants, tout en restant performant pour les utilisateurs avancés.
- **Grande performance :** MySQL est conçu pour gérer des bases de données volumineuses et est capable de gérer des millions d'enregistrements, même sous des charges élevées et des connexions simultanées.
- **Grande communauté :** MySQL bénéficie d'un vaste réseau de développeurs et d'utilisateurs qui contribuent à son développement et à sa maintenance. Offrant ainsi une multitude de ressources et d'extensions pour aider les utilisateurs à résoudre les problèmes et à améliorer les fonctionnalités.
- **Support des requêtes complexes** : MySQL prend en charge les agrégations, les jointures et les index, des fonctionnalités indispensables pour manipuler efficacement les cubes ROLAP.
- **Compatibilité avec Java** : Grâce à JDBC, MySQL permet une interaction simple et efficace entre Java et MySQL.

V.4. Les sockets [7, 22, 36, 38, 63]

La communication entre les différents nœuds d'un système distribué peut se faire selon plusieurs mécanismes, tels que la messagerie asynchrone, les RPC (Remote Procedure Calls), ou les systèmes de partage de fichiers distribués.

Dans notre cas, nous avons opté pour les sockets. Ce choix a été motivé par la flexibilité et la simplicité qu'offrent les sockets pour établir des connexions directes et bidirectionnelles entre les nœuds d'un système distribué, qu'ils soient localisés sur un même site ou sur des sites différents permettant ainsi l'échange de données.

L'importance des sockets repose sur plusieurs autres aspects, tels que :

- **Communication bidirectionnelle**

Les sockets permettent une communication bidirectionnelle entre deux machines ou processus via un réseau. Cette communication est essentielle pour e**nvoyer des requêtes** et recevoir des réponses des différents composants du système : client, serveur et coordinateur.

- **Établissement de connexions fiables** *:* Les sockets basés sur le protocole TCP permettent d'établir des connexions fiables entre deux nœuds. Cela garantit que les messages et les données seront reçus dans leur intégralité, sans altération ni corruption et dans le même ordre dans lequel elles ont été envoyées.

  Cette caractéristique de fiabilité est cruciale dans notre projet pour éviter les incohérences dans les échanges de données entre les nœuds du système ROLAP distribué.

- **Transmission de données structurées :** Les sockets peuvent être utilisés pour échanger des types de données variés allant du texte brut, comme les requêtes SQL aux données sérialisées.

- **Support de la scalabilité :** Les sockets facilitent une communication distribuée efficace, ce qui favorise la scalabilité. Ils permettent entre autre de :

  - ***Ajouter de nouveaux nœuds*** *:* Les sockets permettent l'enregistrement de tout nouveau nœud rejoignant le système auprès des autres nœuds, tout en assurant la synchronisation des données.
  - ***Répartir les charges dynamiquement****:* Les sockets facilitent l'échange de données lors des mises à jour ou des ajustements dynamiques dans un système distribué.

- **Simplicité et compatibilité** *:* Les sockets sont compatibles avec presque tous les langages de programmation et systèmes d'exploitation, ce qui les rend extrêmement polyvalents. Par exemple : En Java, les classes **Socket** (pour le processus client) et **ServerSocket** (pour le processus serveur), fournissent une API simple permettant d'établir des connexions entre l'initiateur de la connexion et celui qui l'accepte.

  Cette compatibilité facilite également leur intégration avec des technologies comme MySQL ou des frameworks de gestion des requêtes.

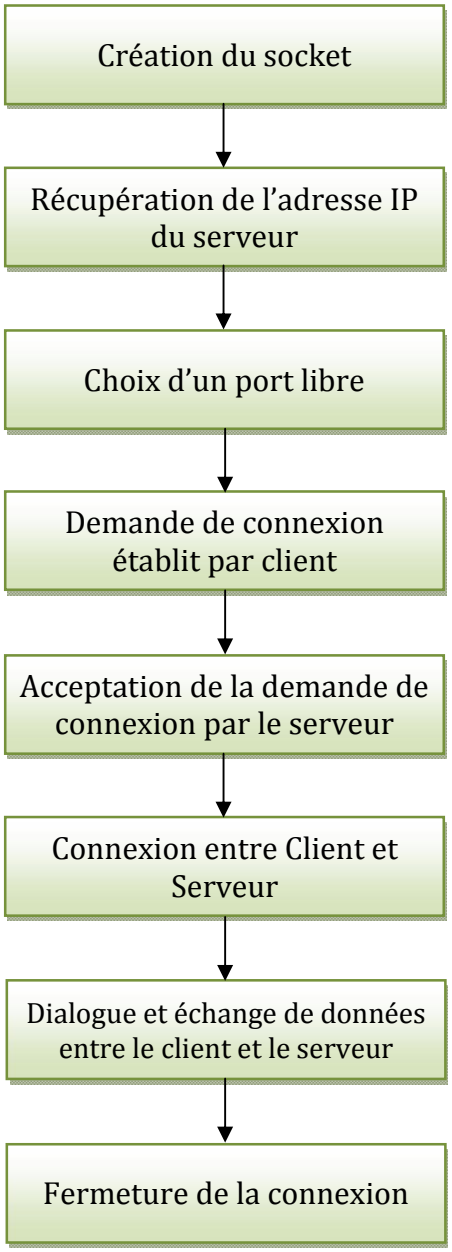
V.4.1. Utilisation des sockets dans un contexte distribué

Dans un environnement distribué, un client et un serveur communiquent en ouvrant chacun un socket. Chaque socket est associé à une adresse machine (IP) et à un numéro de port, formant ainsi une interface unique pour la communication et l'échange de données entre les deux parties.

FigureV.1 : Format d'un socket

L'utilisation des sockets se résume en trois étapes essentielles :

1. **Établissement d'une connexion** :
   - Le processus ***Serveur*** se met à l'écoute sur un port spécifié pour détecter toutes les connexions entrantes.
   - C'est le processus ***Client*** qui initie la connexion en envoyant une demande au serveur.
2. **Échange de messages :**
   - Après l'établissement d'une connexion, l'envoi et la réception de messages se fait sous forme de flux de données entrant et sortant.
   - En Java, la classe *Socket* (client) et *ServerSocket* (serveur) offrent plusieurs méthodes permettant la communication entre le processus **Client** et le processus **Serveur**.
3. **Clôture de la connexion :**
   - Une fois la communication terminée, les processus **Client** et **Serveur** ferment les sockets pour libérer les ressources réseau.



FigureV.2 : Etapes d'une communication via socket

V.4.2. Mode connecté (TCP) versus Mode non connecté (UDP)

La communication via un socket peut être en établie selon deux modes : connecté ou non connecté.

En mode connecté, le protocole TCP est utilisé pour établir une connexion fiable entre le client et le serveur, garantissant que les messages arrivent dans le bon ordre et sans corruption. En revanche, en mode non connecté, où le protocole UDP est utilisé, le client et le serveur échangent des messages sans établir de connexion préalable. Ce mode est certes plus rapide, mais ne garantit pas la livraison des messages ni leur ordre.

Dans notre étude, le mode connecté a été choisi pour permettre les échanges de messages et de données pour plusieurs raisons :

- **Fiabilité assurée** : Grâce aux connexions fiables et orientées flux, les données vont être reçues dans le même ordre que celui dans lequel elles ont été envoyées.
- **Absence de pertes ou duplications :** Chaque message est transmis sans risque de perte ou de duplication.
- **Fragmentation :** Avant leur envoi, les grandes quantités de données sont automatiquement divisées en fragments plus petits, puis réassemblées à leur réception.
- **Contrôle adapté :** Les sockets offrent un contrôle direct sur la communication, permettant de l'ajuster en fonction des exigences spécifiques du système.
- **Compatibilité étendue :** Les sockets sont compatibles avec de nombreux protocoles et langages de programmation, ce qui les rend polyvalents dans différents environnements.

Dans le cadre de cette thèse, les sockets sont utilisés pour permettre la communication entre les nœuds d'un système d'entrepôt de données distribué. Ils servent à échanger des, ainsi qu'à exécuter des requêtes SQL de manière distribuée. Les sockets, grâce à leur fiabilité et à leur compatibilité, jouent un rôle fondamental dans la mise en œuvre de ce système.

Ainsi, les sockets sont utilisés pour faciliter la communication entre les nœuds d'un système d'entrepôt de données distribué (client, coordinateur et serveurs). Ils permettent l'échange de données ainsi que l'exécution de requêtes SQL de manière distribuée. Grâce à leur fiabilité et à leur compatibilité avec divers protocoles et le langage Java, les sockets jouent un rôle essentiel dans la mise en œuvre de ce système.

V.5. Architecture Client/Serveur

L'entrepôt de données sera mise en place selon une architecture client/serveur, un modèle bien adapté aux systèmes décisionnels en raison de sa capacité à gérer efficacement de grands volumes de données tout en maintenant des performances optimales. Dans cette architecture, les données sont réparties sur plusieurs serveurs, chacun hébergeant une partie de l'entrepôt. Les clients peuvent accéder aux données en interrogeant les serveurs de manière transparente, sans se soucier de leur emplacement physique.

Dans la présente étude, on s'intéresse plus spécifiquement à la répartition de la table de fait, élément central de l'entrepôt de données. La table de fait est généralement volumineuse, car elle regroupe des données transactionnelles détaillées, et sujette à des modifications fréquentes, avec des ajouts réguliers de nouvelles données, nécessitant une gestion efficace des opérations d'insertion et de recherche.

Dans le cadre de cette étude, nous avons opté pour la fragmentation horizontale et le partitionnement par hachage selon l'algorithme LH*, afin de diviser la table de faits en plusieurs partitions, chacune renfermant un sous-ensemble des lignes de la table et étant hébergée sur un serveur distinct. Cette stratégie favorise également la scalabilité, facilitant l'ajout de nouveaux serveurs pour héberger des fragments supplémentaires à mesure que le volume des données augmente.

V.6. Echange de requêtes entre un client et un serveur

Pour échanger les données entre les différents composants (client – coordinateur - serveurs) de l'entrepôt distribué, une connexion doit être établie entre ces composants via des sockets.

C'est au client d'initier la communication avec un ou des serveurs. Différentes types de requête peuvent être adressé du client à un serveur :

- **Requête de recherche :** Consultation d'un ou plusieurs enregistrements dans la table de faits.
- **Requête d'ajout :** Insertion de nouveaux enregistrements dans la table de faits.
- **Requête de recherche sur une table de dimensions :** Consultation d'informations spécifiques dans une table de dimensions.
- **Requête d'ajout sur une table de dimensions :** Insertion de données dans une table de dimensions.

Le déroulement des requêtes de recherche soit pour la table de fait ou les tables de dimension se fait de la même manière depuis la demande de connexion initié par le client jusqu'à la réception des réponses des différents serveurs, sans oublier le rôle que joue le coordinateur comme un cite central qui gère la communication entre le client et les serveurs dans le cas d'un entrepôt classique.

Comme a été indiqué dans le chapitre précédent, dans ce travail on s'intéresse plutôt à la table de fait qu'aux tables de dimension. C'est-à-dire, on va présenter le déroulement des scénarios d'exécution pour les requêtes de recherche et d'ajout orientées vers la table de fait.

V.6.1. Requête de recherche

Dans ce type de requête, il s'agit de rechercher des enregistrements dans la table de fait selon quelques critères.

Prenons comme exemple la requête SQL suivante, qui recherche toutes les ventes effectuées par le client "CL10" le 24-03-2024 :

```
SELECT V.Nom, Vt.QteVendue, Vt.Montant
FROM Vente Vt
JOIN Vehicule V ON Vt.Reference = V.Reference
JOIN PointVente PV ON Vt.IdPointVente = PV.IdPointVente
JOIN Date D ON Vt.IdDate = D.IdDate
WHERE PV.CodePtV = "Cl10" AND D.IdDate = '24-03-2024';
GROUP BY V.Nom, D.IdDate
```

FigureV.3 : Exemple d'une requête de recherche orientée vers la table de fait

- **SELECT :** Permet de sélectionner les noms des véhicules (V.Nom), les quantités vendues (Vt.QteVendue) et les montants des ventes (Vt.Montant).
- **JOIN :** Sert de relier la table des faits aux tables de dimension en utilisant les clés étrangères présentes dans la table des faits, qui correspondent aux clés primaires des tables de dimension.
- **WHERE :** Spécifie les critères de recherche afin de ne retenir que les ventes effectuées par le point de vente ayant le code "Cl10", et qui ont eu lieu à la date du 24-03-2024.
- **GROUP BY :** Regroupe les résultats par le nom du véhicule et la date de vente, en l'occurrence le 24 mars 2024, pour le point de vente "Cl10".

V.6.2. Requête d'ajout

Dans une requête d'insertion, l'objectif est d'ajouter un ou plusieurs enregistrements à la table de faits. Pour insérer un nouvel enregistrement, une requête SQL typique pourrait être formulée ainsi :

```
Insert Into Vente (CodePtV, Reference, IdDate, QteVendue, Montant)
Values ("CL1", "Ref23", "2024-03-07", 2, 2600000)
```

FigureV.4 : Exemple d'une requête d'ajout orientée vers la table de fait

Cette requête permet d'insérer un nouvel enregistrement (via **INSERT INTO**) dans la table des faits, avec les données suivantes (représentées via **VALUES**) : le code du client (CL1) ayant acheté deux véhicules (QteVendue), dont la référence est (Ref23), la date de la transaction (07/03/2024), ainsi que le montant total de l'achat (calculé comme le produit du prix unitaire du véhicule par la quantité vendue)."

V.7. Implémentation d'un entrepôt de données classique

Dans un entrepôt de données classique, toute communication entre un client et un ou plusieurs serveurs transitent par le coordinateur. Ce site central, a pour rôle de distribuer les requêtes aux serveurs concernés, puis de transmettre les réponses au client.

Lorsqu'une requête est traitée par plusieurs serveurs, les réponses de ces derniers sont d'abord combinées par le coordinateur avant d'être envoyées au client.

V.7.1. Requête de recherche sur la table de fait

Qu'il s'agisse de rechercher un seul enregistrement ou plusieurs, ce type de requête est toujours diffusé en multicast à l'ensemble des serveurs du système.

Une fois sa requête formulée, le client établit une connexion via un socket pour communiquer avec le coordinateur. Ce dernier joue le rôle d'intermédiaire entre le client et les serveurs.

Le traitement de la requête se déroule en plusieurs étapes :

a. Connexion entre un client et le coordinateur

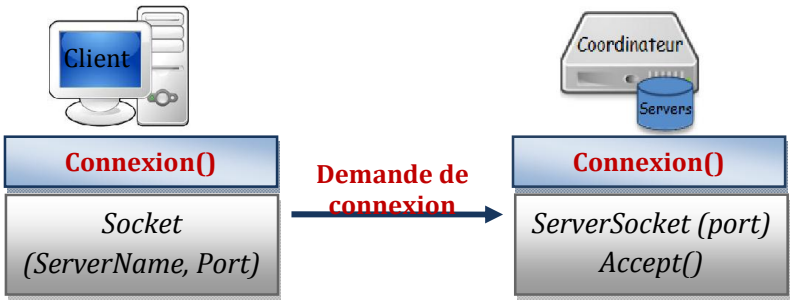
Côté client :

La première étape consiste à établir une connexion avec le coordinateur en utilisant la procédure *Connexion()*. Le client initie cette connexion en créant un objet *Socket*, qu'il transmet au coordinateur via un port de communication spécifique, dont la valeur est indiquée dans la table *Servers*.

La procédure *Connexion()* prend deux paramètres : l'adresse du serveur et le numéro du port de communication.

Côté coordinateur :

Le coordinateur se met en écoute sur le port spécifique. Lorsqu'il reçoit une demande de connexion du client, il accepte la communication grâce à la méthode *accept()* de la classe *ServerSocket*.



FigureV.5 : Connexion entre le client et le coordinateur

b. Echange de données entre le client et le coordinateur

Une fois la connexion établie entre le client et le coordinateur, les deux parties peuvent échanger des données via les flux *InPutStream* et *OutPutStream*.

Côté Client :

Le client transmet au coordinateur un message de type *String* structuré en deux parties:

- **Part[0]** : Cette première partie du message indique le type de la requête. Elle détermine la méthode à exécuter et, par conséquent, définit implicitement la nature de la requête (recherche ou insertion) tout en précisant la table concernée
- **Part[1] :** Cette seconde partie contient le contenu détaillé de la requête.

FigureV.6 : Structure d'une requête de recherche

Côté coordinateur :

Lorsqu'un message est reçu du client, le coordinateur l'analyse pour identifier le type de requête à exécuter, spécifié dans Part[0]. Dans ce cas précis, Part[0] indique qu'il s'agit d'une requête de recherche.

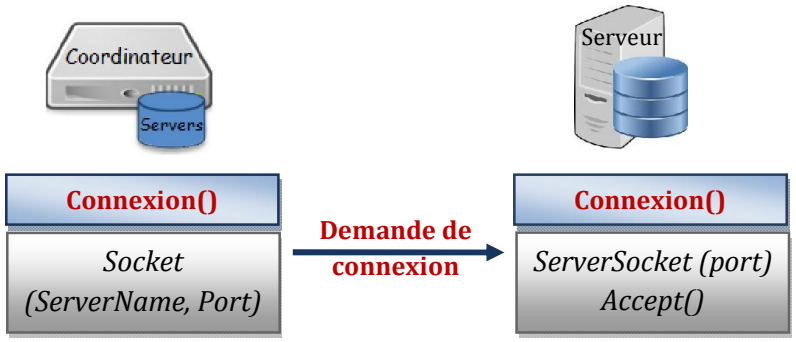
Le coordinateur doit ensuite transmettre cette requête aux différents serveurs du système.

c. Connexion entre le coordinateur et les serveurs

Côté coordinateur :

Le coordinateur diffuse la requête de recherche en multicast à tous les serveurs hébergeant les enregistrements de la table de faits. Pour cela, il doit ouvrir autant de sockets sur autant de ports qu'il y a de serveurs.

Le coordinateur commence par consulter la table *Servers* afin de récupérer les paramètres de communication de l'ensemble des serveurs concernés. Ensuite, il envoie une demande de connexion avec ces serveurs en exécutant la procédure *Connexion()*. Cette demande est adressée aux numéros de port extraits de la table *Servers*.



FigureV.7 : Connexion entre le coordinateur et un serveur

Côté serveur :

Lorsqu'un serveur reçoit une demande de connexion sur son port de communication, il doit la valider en utilisant la méthode *accept().*

d. Echange de données entre le coordinateur et les serveurs

Côté coordinateur :

Une fois la connexion établie entre le coordinateur et les serveurs, l'échange de messages et de données peut s'effectuer via les flux *InputStream* et *OutputStream*. *InputStream* permet de lire un flux de données entrant, tandis qu'*OutputStream* permet d'envoyer un flux de données sortant vers une destination.

Le message transmis aux serveurs est identique à celui reçu du client. Il se compose de deux parties : la première indique le type de requête, tandis que la seconde contient les détails de la requête à exécuter.

Côté serveur :

A la réception de la requête de recherche provenant du coordinateur, chaque serveur procédera à l'exécution des étapes suivantes :

1) Décomposer le message reçu du coordinateur pour extraire le nom de la méthode à exécuter (Part[0]).

2) Appeler la méthode correspondante et lui transmettre la requête (Part[1]) afin de la traiter.

3) Se connecter à sa table locale pour effectuer la recherche demandée.

4) Envoyer le résultat au coordinateur sous forme d'un tableau d'enregistrements de type *ArrayList < String >*.

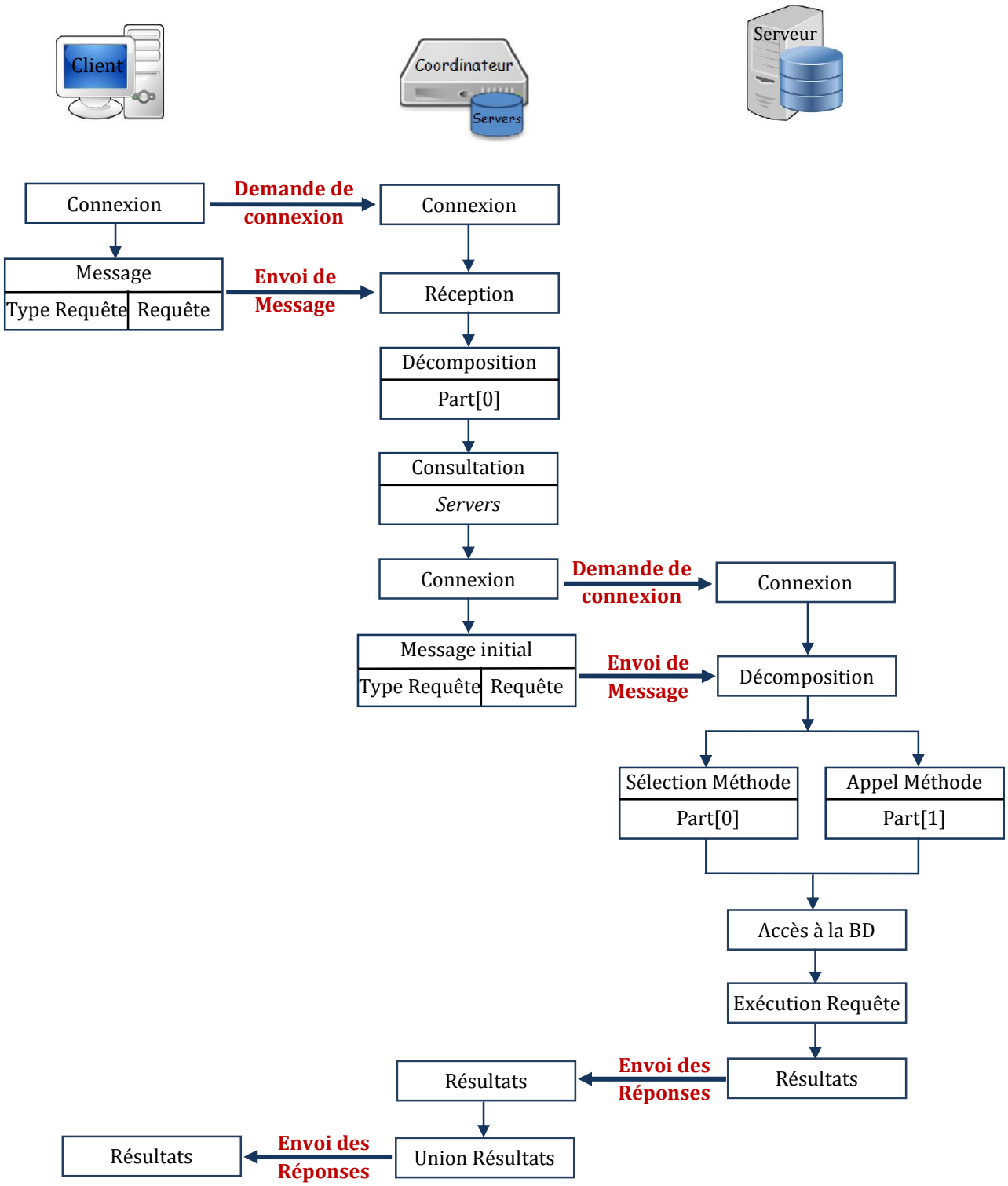
e. Echange de données entre le coordinateur et le client

Côté coordinateur :

Une fois les résultats reçus des différents serveurs, le coordinateur les regroupe dans un tableau unique qu'il transmet ensuite au client.

Côté Client :

Les résultats de la requête sont reçus sous la forme d'un objet *ArrayList<String>*, qui est ensuite converti et affiché sous un format lisible, tel qu'un tableau bidimensionnel.



FigureV.8 : Scénario d'exécution d'une requête de recherche

V.7.2. Requête d'ajout sur la table de fait.

Dans cette requête d'ajout, visant à insérer d'un ou de nouveaux enregistrements dans la table de faits, la communication s'établit par la création de sockets à la fois entre le client et le coordinateur, ainsi qu'entre le coordinateur, les serveurs actif et suivant.

Les étapes d'exécution de cette requête sont les suivantes :

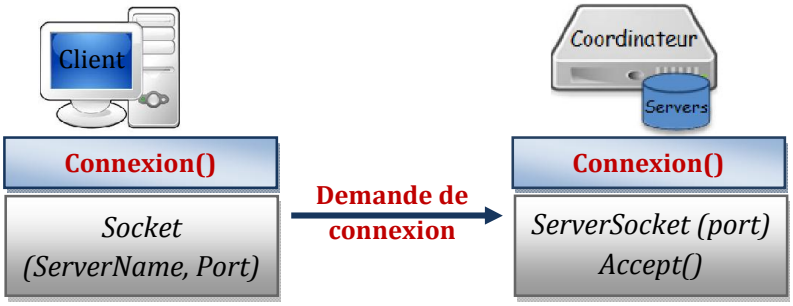
a. Connexion entre un client et le coordinateur

Côté client :

A l'instar du déroulement de l'exécution d'une requête de recherche, la première étape consiste à établir une connexion entre le client et le coordinateur à l'aide de la procédure *Connexion()*. Le client initie cette connexion en envoyant un nouveau objet *Socket* au coordinateur sur le port spécifié pour se communiquer.

Côté coordinateur :

Le coordinateur reste en écoute sur un port spécifique. Lorsqu'il reçoit une demande de connexion de la part du client, il établit la communication en utilisant la méthode *accept()* de la classe *ServerSocket*.



FigureV.9 : Connexion entre le client et le coordinateur

b. Echange de données entre le client et le coordinateur

Une fois la connexion établie entre le client et le coordinateur, les deux parties peuvent échanger des données via les flux *InPutStream* et *OutPutStream*.

Côté Client :

Le client envoie un message au coordinateur sous forme de chaîne de caractères (*String*), structurée en deux parties : Part[0] et Part[1], de la même manière que dans la requête de recherche :

- **Part[0] :** Pour spécifier la méthode à exécuter, définissant ainsi la nature de la requête (recherche ou insertion) et la table concernée.
- **Part[1] :** Ce champ contient la requête elle-même, formulée selon les besoins spécifiques du client.

FigureV.10 : Structure d'une requête d'ajout

Côté coordinateur :

A la réception du message du client, le coordinateur :

1) Décompose le message afin d'extraire le type de la requête (Part[0]).

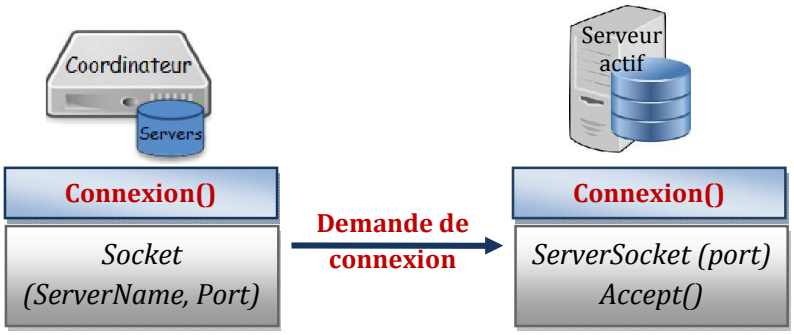
2) Étant donné qu'il s'agit d'une requête d'ajout, le coordinateur consulte la table *Servers* pour identifier les paramètres du serveur actif, c'est-à-dire le serveur chargé de recevoir la requête et d'y insérer les nouveaux enregistrements.

Deux cas de figure sont possibles :

i) Si l'espace disponible du serveur actif est suffisant pour insérer tous les enregistrements, la communication dans ce cas se fait entre le coordinateur et le serveur actif.

a. Connexion entre le coordinateur et le serveur actif

Le coordinateur envoie une demande de connexion au serveur actif.



FigureV.11 : Connexion entre le coordinateur et le serveur actif

Côté serveur actif :

En recevant la demande de connexion sur le port de communication, le serveur actif doit l'accepter via la méthode *accept()*.

b. Echange de données entre le coordinateur et le serveur actif

Côté coordinateur :

Une fois la connexion établie entre le coordinateur et le serveur actif, l'échange de messages et de données peut se faire via les flux entrants (***InputStream***) et sortants (***OutputStream***).

Le message transmis par le coordinateur au serveur est identique à celui reçu primitivement du client, ie : il se compose de deux parties Part[0] et Part[1].

Côté serveur actif :

A la réception du message provenant du coordinateur, le serveur actif effectuera les actions suivantes :

1) Analyser le message pour extraire le nom de la méthode à exécuter (contenu dans Part[0]).
2) Invoker la méthode correspondante en lui transmettant la requête (Part[1]) pour être exécutée.
3) Accéder à sa table locale pour effectuer l'opération d'ajout.
4) Envoyer un message de confirmation au coordinateur, indiquant que les enregistrements ont été ajoutés avec succès.

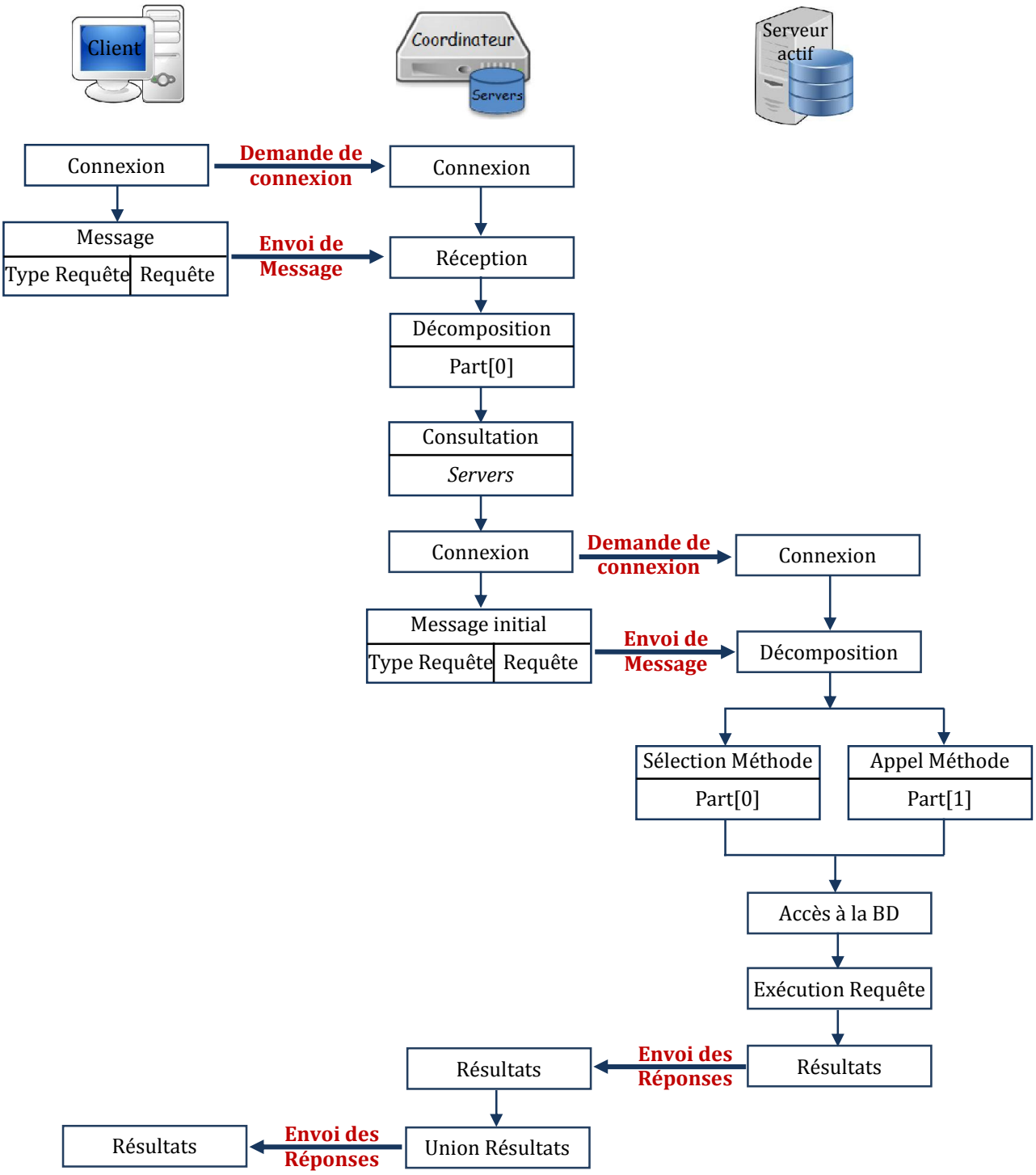
c. Echange de données entre le coordinateur et le client

Côté coordinateur :

Une fois le message de confirmation d'ajout reçu du serveur actif, le coordinateur le transmet au client.

Ensuite, le coordinateur met à jour les paramètres du serveur actif, notamment l'espace de stockage disponible, selon la formule suivante :

Nouvel espace disponible
=
Ancien espace disponible – Nombre d'enregistrements insérés



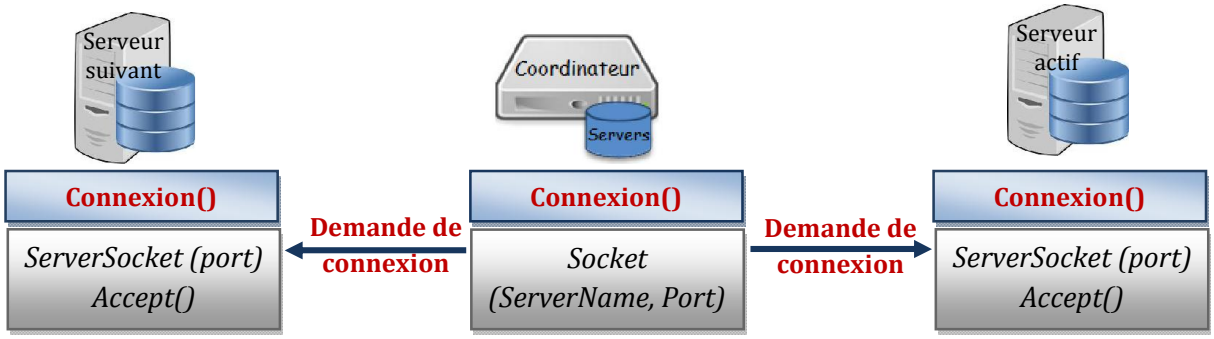
FigureV.12 : Scénario d'exécution d'une requête d'ajout au niveau du serveur actif

ii) Si le serveur actif dispose d'espace, mais insuffisant pour stocker la totalité des nouveaux enregistrements, alors le coordinateur récupère les paramètres des deux serveurs de la table *Servers* : le serveur actif et le serveur suivant (celui désigné comme prochain serveur actif en cas de saturation du serveur actuel). Par la suite, il établit une connexion simultanée avec ces deux serveurs.

a. Connexion entre le coordinateur et les deux serveurs actif et suivant

Côté coordinateur :

Le coordinateur envoie deux demandes d'établissement de connexion aux deux serveurs, en utilisant la procédure *Connexion()*. Il veille, bien entendu, à se connecter à des ports distincts pour chacun des serveurs.



FigureV.13 : Connexion entre le coordinateur et les serveurs actif et suivant

Côté serveurs :

Lorsqu'une demande de connexion est reçue sur son port de communication, chaque serveur doit l'accepter à l'aide de la méthode *accept()*. Cela permet d'établir la connexion et de permettre l'échange ultérieur de messages et de données avec le coordinateur qui a initié cette connexion.

b. Echange de données entre le coordinateur et les deux serveurs actif et suivant

Côté coordinateur :

Après l'acceptation de la demande de connexion par les deux serveurs, le coordinateur dispose désormais de deux canaux de communication : l'un avec le serveur actif et l'autre avec le serveur suivant.

En fonction de l'espace de stockage disponible du serveur actif, récupéré depuis la table *Servers*, le coordinateur segmente la requête. Il transmet au serveur actif uniquement le nombre d'enregistrements correspondant à sa capacité, tandis que les enregistrements restants sont acheminés au serveur suivant.

Côté serveurs :

À la réception du message encapsulant les enregistrements à insérer, les deux serveurs vont :

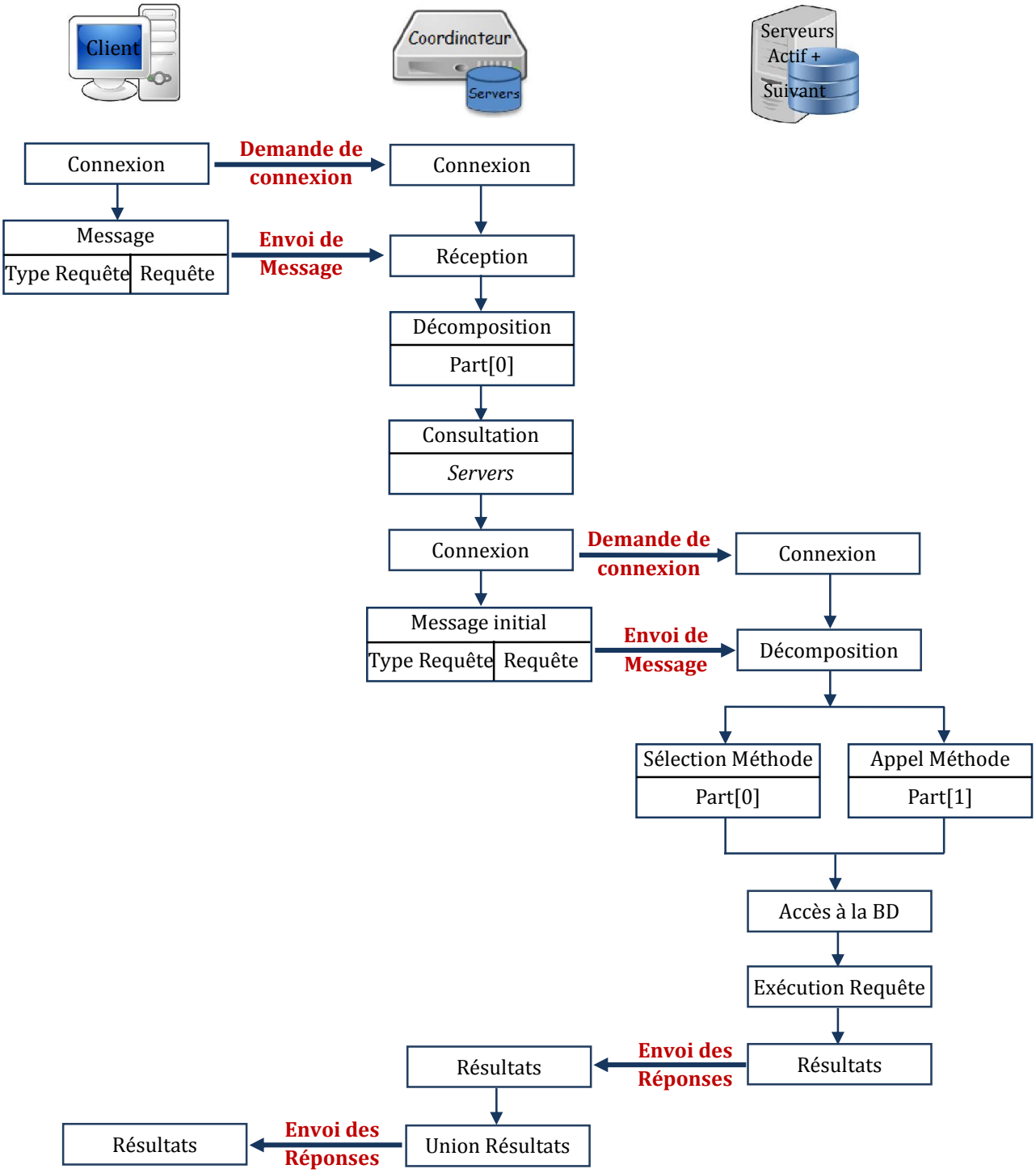
1) Décomposer le message afin d'extraire le nom de la méthode à exécuter (Part[0]).
2) Transmettre la requête (Part[1]) à la méthode appropriée pour qu'elle soit exécutée.
3) Se connecter à leur table locale pour insérer les enregistrements reçus.
4) Envoyer un message de validation d'ajout au coordinateur.

c. Echange de données entre le coordinateur et le client

Côté coordinateur :

Après la réception des messages de validation d'ajout des deux serveurs, le coordinateur les retransmet au client.

En dernière étape d'une requête d'ajout, le coordinateur met à jour les paramètres des deux serveurs dans la table *Servers*. Le serveur suivant devient le nouveau serveur « actif », et un autre serveur est désigné comme le nouveau serveur « suivant ». L'espace de stockage disponible du précédent serveur actif est mis à 0, tandis que celui du nouveau serveur actif est recalculé en soustrayant le nombre d'enregistrements insérés de son espace disponible initial.



FigureV.14 : Scénario d'exécution d'une requête d'ajout

au niveau du serveur actif et suivant

V.8. Implémentation d'un entrepôt de données en SDDS : DW_SDDS

Contrairement aux entrepôts de données classiques, où toutes les communications transitent systématiquement par le coordinateur, un entrepôt de données basé sur un SDDS (DW_SDDS) n'implique le coordinateur qu'en cas de débordement.

Le client peut communiquer directement avec un serveur en se basant sur son image, qui contient tous les paramètres relatifs à la répartition des enregistrements, ainsi que les adresses des serveurs les hébergeant.

Pour échanger des données, une connexion directe doit d'abord être établie entre le client et le serveur, cette connexion étant initiée par le client.

V.8.1. Requête de recherche sur la table de faits d'un seul enregistrement

Dans ce type requête, le client cherche à localiser un seul enregistrement dans la table de faits. Pour cela, il commence par consulter son image afin d'obtenir l'adresse du serveur hébergeant l'enregistrement recherché, ainsi que le numéro de port de communication.

Les étapes d'exécution de cette requête se résument comme suit :

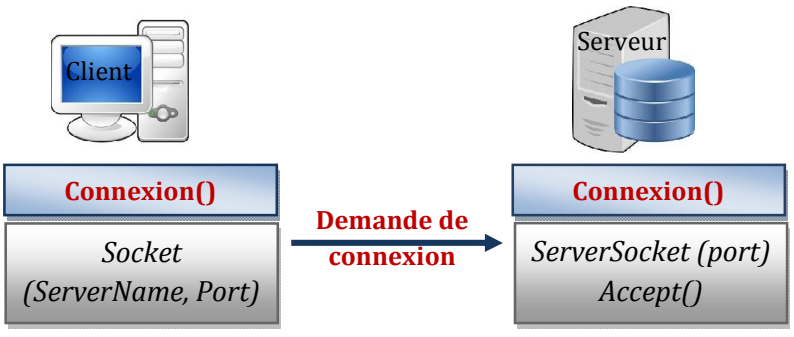
a. Connexion entre un client et le serveur

Côté client :

Le client initie une connexion avec le serveur hébergeant l'enregistrement recherché en exécutant la procédure *Connexion().* Un objet *Socket* est alors créé et transmis au serveur via le port de communication spécifié, extrait par le client de son image locale.

Côté serveur :

Le serveur se met en écoute sur le port spécifique. Lorsqu'il reçoit une demande de connexion du client, il accepte la communication grâce à la méthode *accept()* de la classe *ServerSocket*.



FigureV.15 : Connexion entre un client et un serveur

b. Echange de données entre le client et le serveur

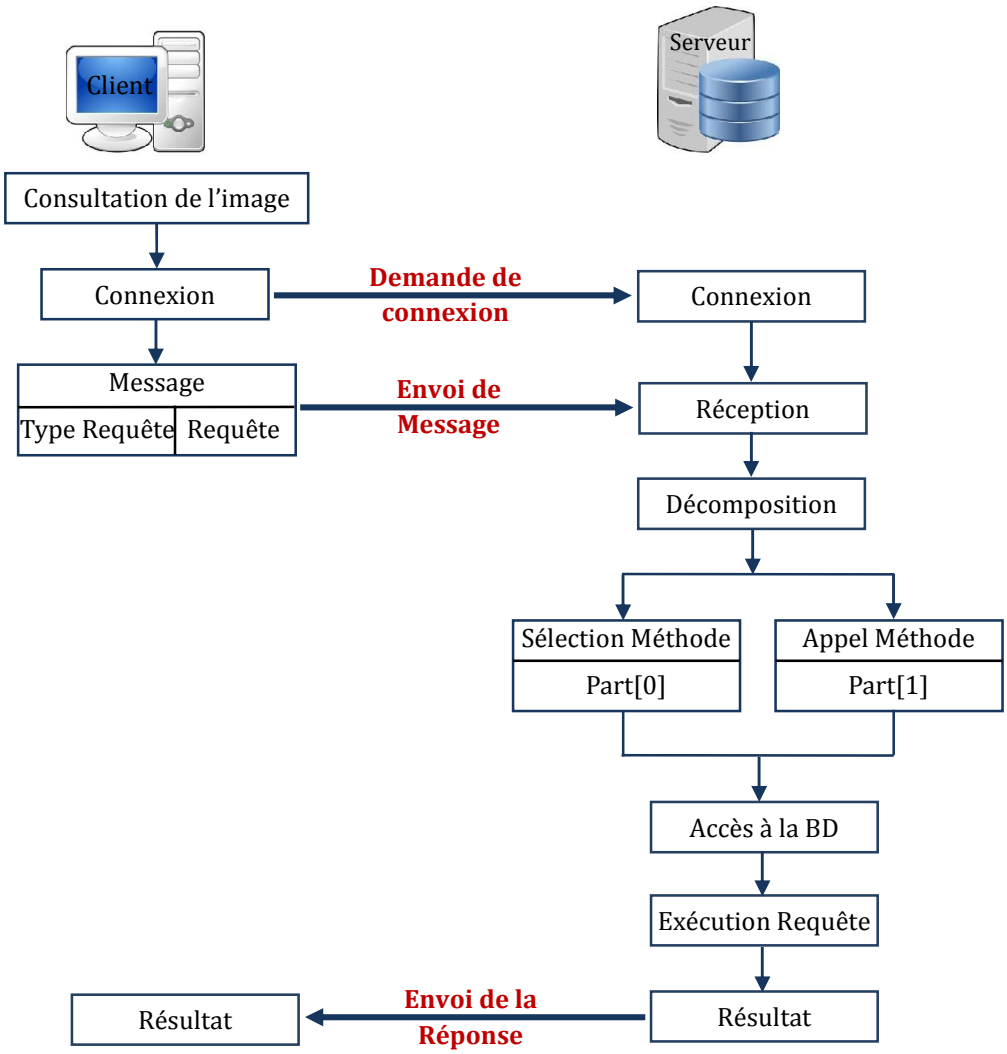
Côté client :

Dès lors, le client et le serveur peuvent communiquer et échanger des messages : le client envoie au serveur la requête à exécuter en flux ***OutputStream***, et reçoit les résultats de son exécution en *In**putStream***.

Côté serveur :

Le serveur recevant la requête de recherche, suivra les étapes suivantes :

1) Décomposer le message reçu du client pour extraire le nom de la méthode à exécuter (Part[0]).
2) Invoker la méthode correspondante et lui transmettre la requête (Part[1]) afin de la traiter.
3) Se connecter à sa table locale pour effectuer la recherche demandée.
4) Le serveur localise l'enregistrement dans sa base de données locale et prépare la réponse.
5) Le serveur renvoie le résultat (l'enregistrement trouvé ou un message indiquant qu'il n'existe pas) au client.



FigureV.16 : Scénario d'exécution d'une requête de recherche d'un seul enregistrement

V.8.2. Requête de recherche sur la table de fait de plusieurs enregistrements

Pour effectuer une recherche de plusieurs enregistrements dans la table de faits, le client envoie une requête en multicast à tous les serveurs hébergeant ces enregistrements. Le client commence d'abord par consulter son image afin de récupérer les paramètres nécessaires pour chaque serveur, notamment leurs adresses et numéros de port.

Ensuite, le traitement la requête se déroulera selon les étapes de suivantes :

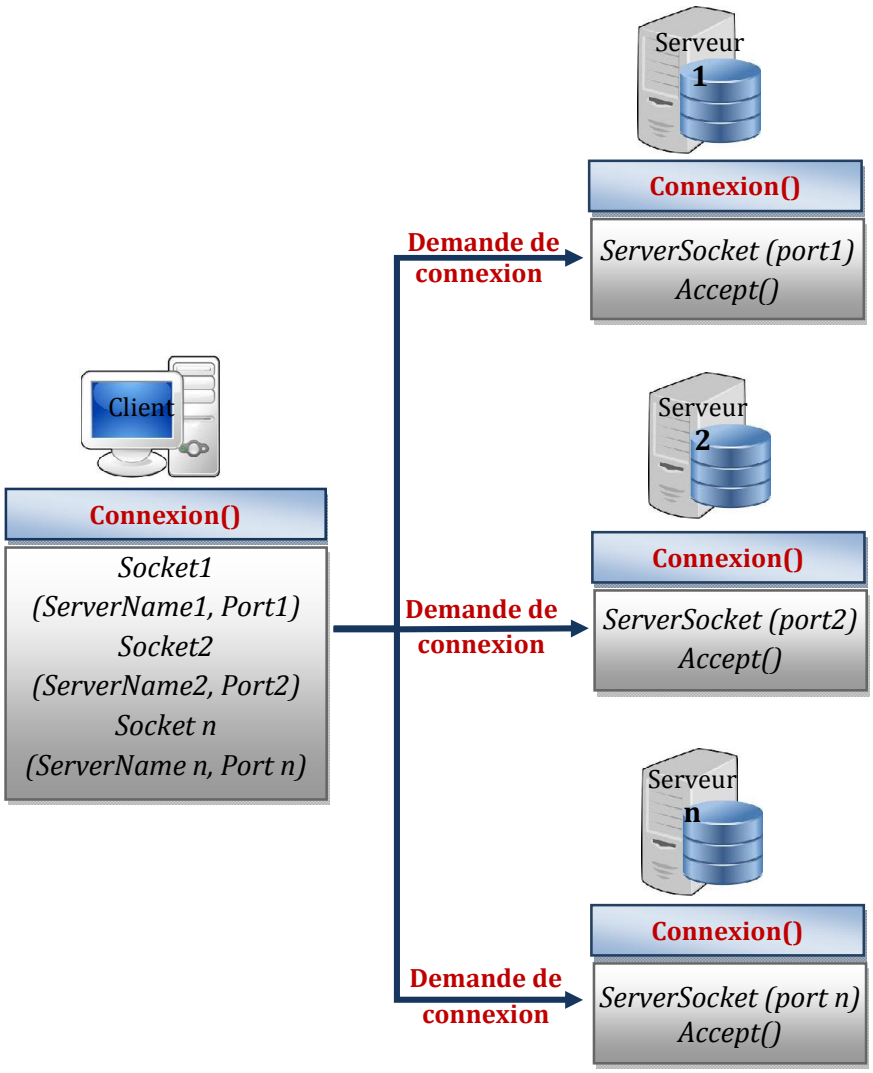
a. Connexion entre un client et les serveurs

Côté client :

Étant donné qu'il s'agit d'un envoi en multicast, le client initie une demande de connexion à l'ensemble des serveurs du système via la procédure *Connexion().* Pour ce faire, il doit ouvrir un socket distinct pour chaque port associé à chacun des serveurs.

Côté serveur :

Chaque serveur se met en écoute sur le port spécifique. Lorsqu'il reçoit une demande de connexion du client, il accepte la communication.



FigureV.17 : Connexion entre un client et n serveurs

b. Echange de données entre le client et les serveurs

Dès que la connexion entre le client et les serveurs est établie, les échanges de données s'opèrent par le biais des flux entrants *InputStream* et sortants *OutputStream*.

Côté Client :

Le client procède à la transmission au serveur un message de type *String* contenant le type de requête et la requête détaillée (enregistrements à rechercher).

Côté serveur :

Le serveur, après avoir reçu la requête de recherche, exécutera les étapes suivantes :

1) Extrait le nom de la méthode à exécuter (Part[0]) du message reçu.

2) Invoker la méthode correspondante et lui passer la requête (Part[1]) pour traitement.

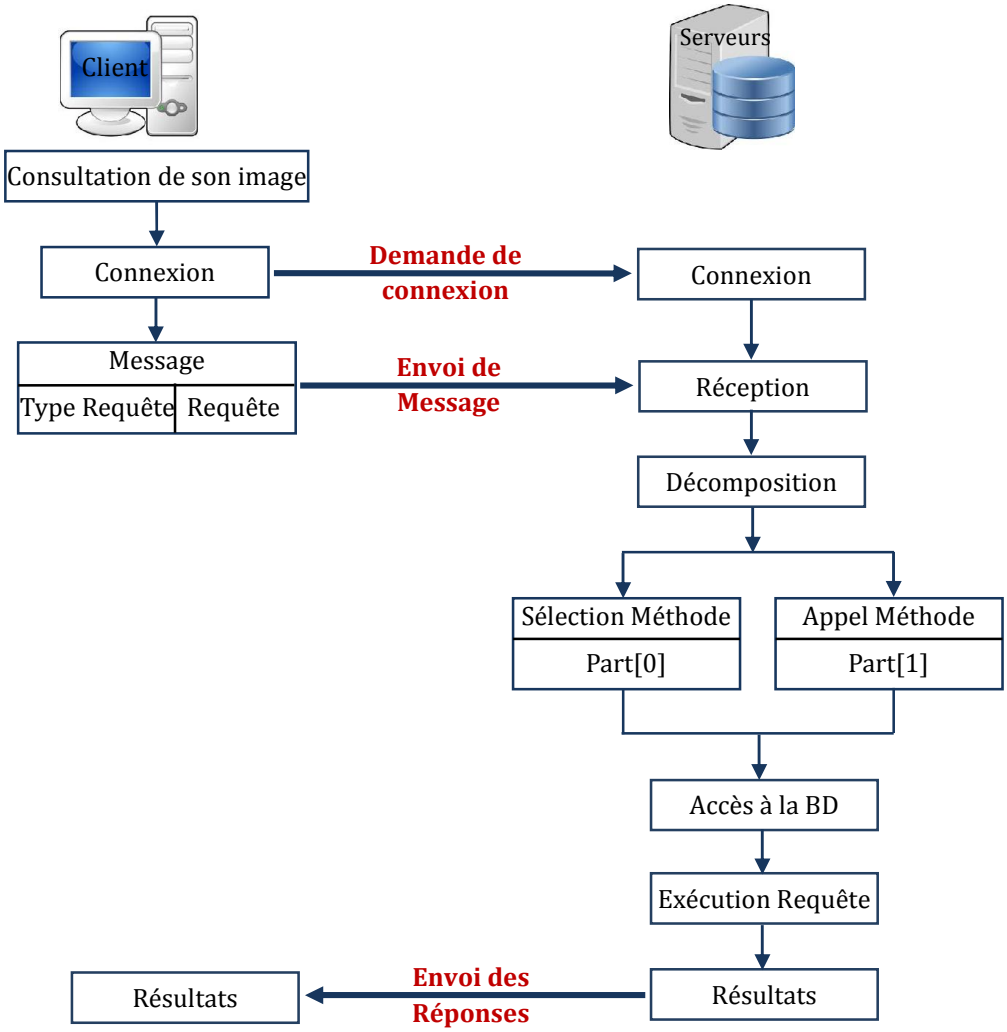
3) Accéder à sa table locale afin d'exécuter la recherche demandée.

4) Traiter la requête en se connectant à sa table locale afin d'identifier les enregistrements correspondants, et préparer une réponse.

5) Renvoie sa réponse au client, incluant les enregistrements trouvés ou des messages d'absence si aucun enregistrement ne correspond.

Côté client :

Le client fusionne et regroupe les résultats reçus des différents serveurs pour produire une réponse consolidée.



FigureV.18 : Scénario d'exécution d'une requête de recherche d'un ensemble d'enregistrements

V.8.3. Requête d'ajout sur la table de fait.

Lorsqu'un client souhaite insérer un ou plusieurs enregistrements dans la table de faits, il commence par consulter sa propre image du système pour identifier le serveur actif et en extraire ses paramètres (adresse et port). La communication s'établit alors directement entre le client et le serveur, sans intervention du coordinateur.

Deux cas peuvent exister :

i) L'espace de stockage disponible sur le serveur actif est suffisant pour accueillir tous les enregistrements.

Dans ce cas, le client consulte son image pour extraire les paramètres nécessaires à la communication avec le serveur actif, notamment l'adresse du serveur et le numéro du port. Cela permet au client d'établir une connexion directe avec le serveur afin de transmettre les enregistrements à insérer.

L'exécution de cette requête d'ajout suit les étapes suivantes :

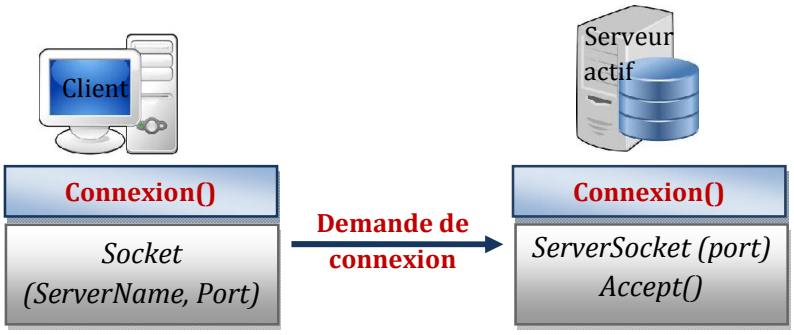
a. Connexion entre le client et le serveur actif

Côté client :

Comme pour toute communication, le client engage une demande de connexion avec le serveur actif à l'aide de la procédure *Connexion()*.Il crée un objet *Socket*, qu'il transmet à ce serveur via le port dédié, afin d'établir la liaison.

Côté serveur actif :

Le serveur actif reste en écoute sur un port spécifique. Lorsqu'il reçoit une demande de connexion de la part du client, il établit la communication en utilisant la méthode *accept()* de la classe *ServerSocket*.



FigureV.19 : Connexion entre le client et le serveur actif

b. Echange de données entre le client et le serveur actif

Côté Client :

Le client envoie au serveur, via le flux *OutputStream*, la requête d'ajout contenant les enregistrements à insérer, structurée de manière identique aux requêtes précédentes.

Côté serveur actif :

A la réception du message provenant du client, le serveur actif effectuera les opérations suivantes :

1) Analyser le message pour extraire le nom de la méthode à exécuter (contenu dans Part[0]).
2) Invoker la méthode appropriée en lui passant la requête (Part[1]) pour exécution.
3) Se connecter à sa table locale afin de réaliser l'opération d'ajout.
4) Envoyer une réponse au client indiquant que les enregistrements ont été insérés avec succès.

ii) L'espace de stockage disponible sur le serveur actif est insuffisant pour insérer tous les enregistrements.

Le client consulte son image pour récupérer les paramètres requis pour communiquer avec le serveur actif, notamment l'adresse du serveur et le numéro du port, lui permettant ainsi d'établir une connexion directe avec ce dernier.

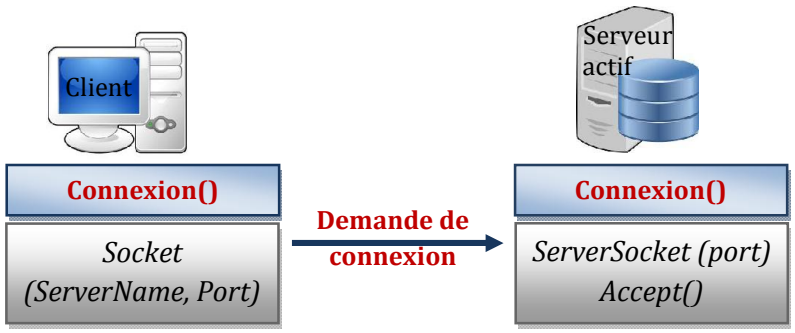
a. Connexion entre le client et le serveur actif

Côté client :

Le client initie une demande de connexion avec le serveur actif en utilisant la procédure *Connexion()*. Pour ce faire, il crée un objet *Socket*, qu'il utilise pour établir une communication avec le serveur actif via le port spécifié dans son image. Cette connexion permet un échange direct entre le client et le serveur, garantissant la transmission des données.

Côté serveur actif :

Le serveur actif reste en écoute sur un port spécifique. Lorsqu'il reçoit une demande de connexion de la part du client, il établit la communication en utilisant la méthode *accept()* de la classe *ServerSocket*.



FigureV.20 : Connexion entre le client et le serveur actif

b. Echange de données entre le client et le serveur actif

Lorsque le serveur actif reçoit la requête d'ajout, il constate que sa capacité de stockage disponible est insuffisante pour insérer tous les nouveaux enregistrements, ce qui provoque un débordement.

Dans ce cas, le serveur actif contacte le coordinateur en lui envoyant un message informatif signalant la situation. Ce message permet au coordinateur de mettre en œuvre les actions nécessaires pour redistribuer les enregistrements vers d'autres serveurs.

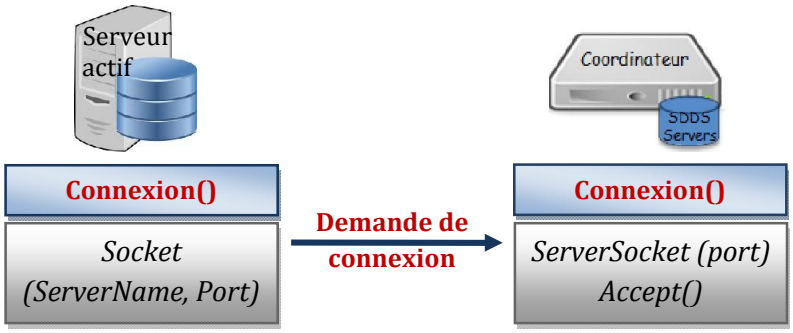
c. Connexion entre le serveur actif et le coordinateur

Côté serveur actif :

Le serveur actif établit une connexion avec le coordinateur en utilisant la procédure *Connexion()*. Il crée un objet *Socket*, qu'il transmet au coordinateur via le port approprié pour permettre la communication.

Côté coordinateur :

Lorsqu'il reçoit une demande de connexion de la part du client, le coordinateur valide la communication en appelant la méthode *accept()* de la classe *ServerSocket*.



FigureV.21 : Connexion entre le serveur actif et le coordinateur

d. Echange de données entre le serveur actif et le coordinateur

Côté serveur actif :

Le serveur actif envoie un message informatif au coordinateur via le port spécifié pour signaler qu'un débordement a eu lieu. Ce message permet au coordinateur de prendre connaissance de la situation et d'intervenir pour gérer le débordement.

Côté coordinateur :

Le coordinateur consulte sa table *SDDSServers* pour extraire l'adresse et le port de communication du serveur suivant. Ces informations sont ensuite envoyées au serveur actif.

e. Connexion entre le coordinateur, le serveur actif et le serveur suivant

Côté coordinateur :

Le coordinateur établit une connexion avec le serveur suivant en se connectant à son port de communication.

Le serveur suivant est celui désigné par le pointeur n, c'est-à-dire le pointeur qui indique l'adresse du prochain serveur à éclater. Ainsi, le serveur suivant est celui qui va être éclaté, et non le serveur actif où le débordement a eu lieu.

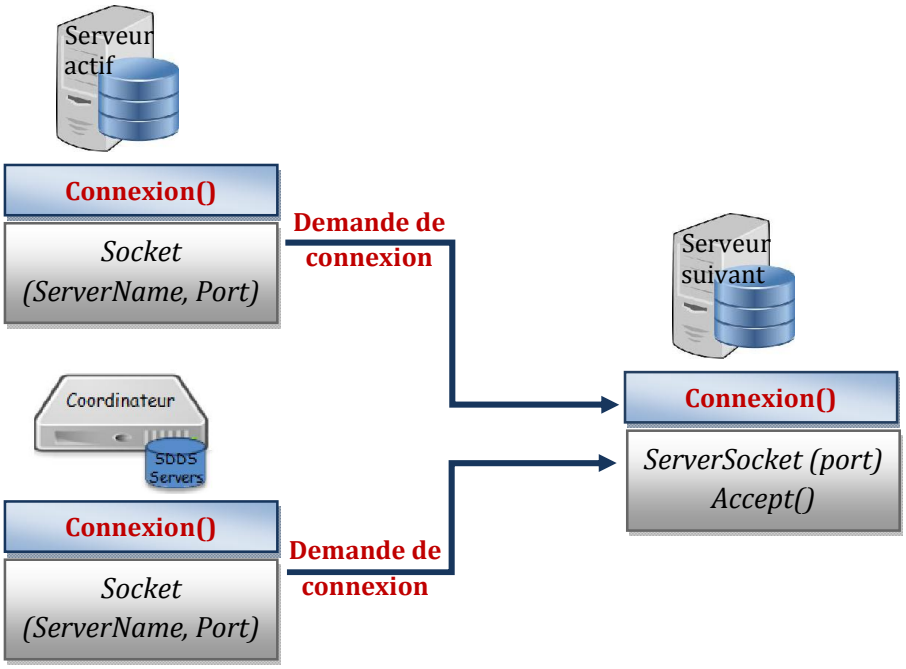
Le coordinateur dispose désormais de deux sockets : l'un pour communiquer avec le serveur actif et l'autre avec le serveur suivant. Par ailleurs, le premier socket, créé pour la communication avec le client, peut maintenant être fermé.

Côté serveur actif :

Le serveur actif envoie également une demande de connexion au serveur suivant, en utilisant un port de communication distinct de celui employé par le coordinateur. Ainsi, deux objets *Socket* sont créés : l'un pour établir la connexion avec le coordinateur et l'autre pour la connexion avec le serveur suivant. Cette séparation des canaux de communication assure une gestion efficace des différentes connexions au sein du système.

Côté serveur suivant :

Le serveur suivant, une fois les demandes de connexion reçues, accepte la connexion avec le coordinateur et le serveur actif. Cela se fait en utilisant la procédure *Connexion()*, où deux objets *Socket* sont créés : l'un pour se connecter avec le coordinateur et l'autre avec le serveur actif.



FigureV.22 : Connexion entre le serveur actif, le coordinateur et le serveur suivant

f. Echange de données entre le coordinateur, le serveur actif et le serveur suivant

Côté coordinateur :

Le coordinateur envoie une demande d'éclatement au serveur suivant, accompagnée des paramètres nécessaires pour établir la communication avec le nouveau serveur. Cette demande comprend les détails requis pour que le serveur suivant puisse gérer la répartition des enregistrements.

Le nouveau serveur est celui qui accueillera les enregistrements transférés depuis le serveur suivant.

Côté serveur actif :

Le serveur actif renvoie la requête d'ajout initialement adressée par le client au serveur suivant. En plus de cette requête, le serveur actif transmet également les paramètres du client, incluant son adresse et le numéro de port de communication. Cela permet au serveur suivant de savoir d'où provient la requête et d'assurer la gestion appropriée des données.

Côté serveur suivant :

Le serveur suivant reçoit la requête d'ajout et les informations associées. Il se charge d'exécuter cette requête, en recalculant les adresses de tous les enregistrements (nouveaux et déjà présents dans sa table locale) en utilisant la fonction de hachage $\mathbf{h_i = C \bmod N * 2^i}$.

Afin de rééquilibrer la charge entre les serveurs, le serveur suivant transfère la moitié des enregistrements vers le nouveau serveur. Cette opération permet d'assurer que la charge de travail soit bien distribuée, évitant ainsi les débordements ou surcharges sur un serveur spécifique.

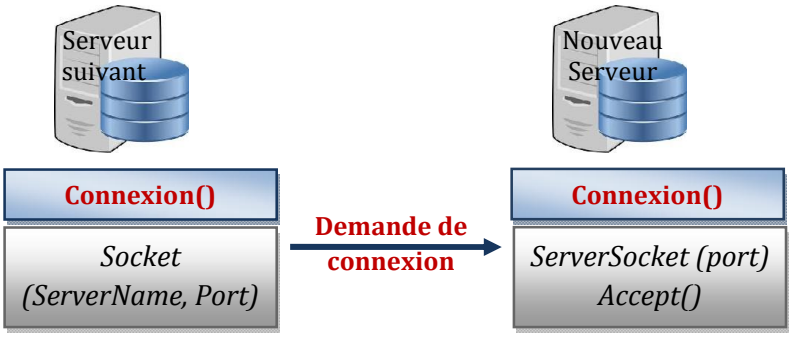
g. Connexion entre le serveur suivant et le nouveau serveur

Côté serveur suivant :

Le serveur suivant envoie une demande de connexion au nouveau serveur en utilisant le port de communication fourni par le coordinateur. Cette étape permet d'établir une connexion entre les deux serveurs pour transférer les enregistrements.

Côté nouveau serveur:

Le nouveau serveur accepte la connexion avec le serveur suivant via la procédure *Connexion()*, permettant ainsi d'établir une communication entre les deux.



FigureV.23 : Connexion entre le serveur suivant et le nouveau serveur

h. Echange de données entre le serveur suivant et le nouveau serveur

Côté serveur suivant :

Après avoir recalculé les adresses de tous les enregistrements (anciens et nouveaux) à l'aide de la fonction de hachage $\mathbf{h_i}$, le serveur suivant procède à l'insertion des enregistrements dans sa table locale, et dont leur adresse est inférieure à **n**. Les enregistrements restants, dont l'adresse dépasse **n**, seront transférés au nouveau serveur pour y être insérés. (*Voir AlgorithmeIII.4)*

Rappelons que n est le pointeur désignant le serveur à éclater.

Côté nouveau serveur :

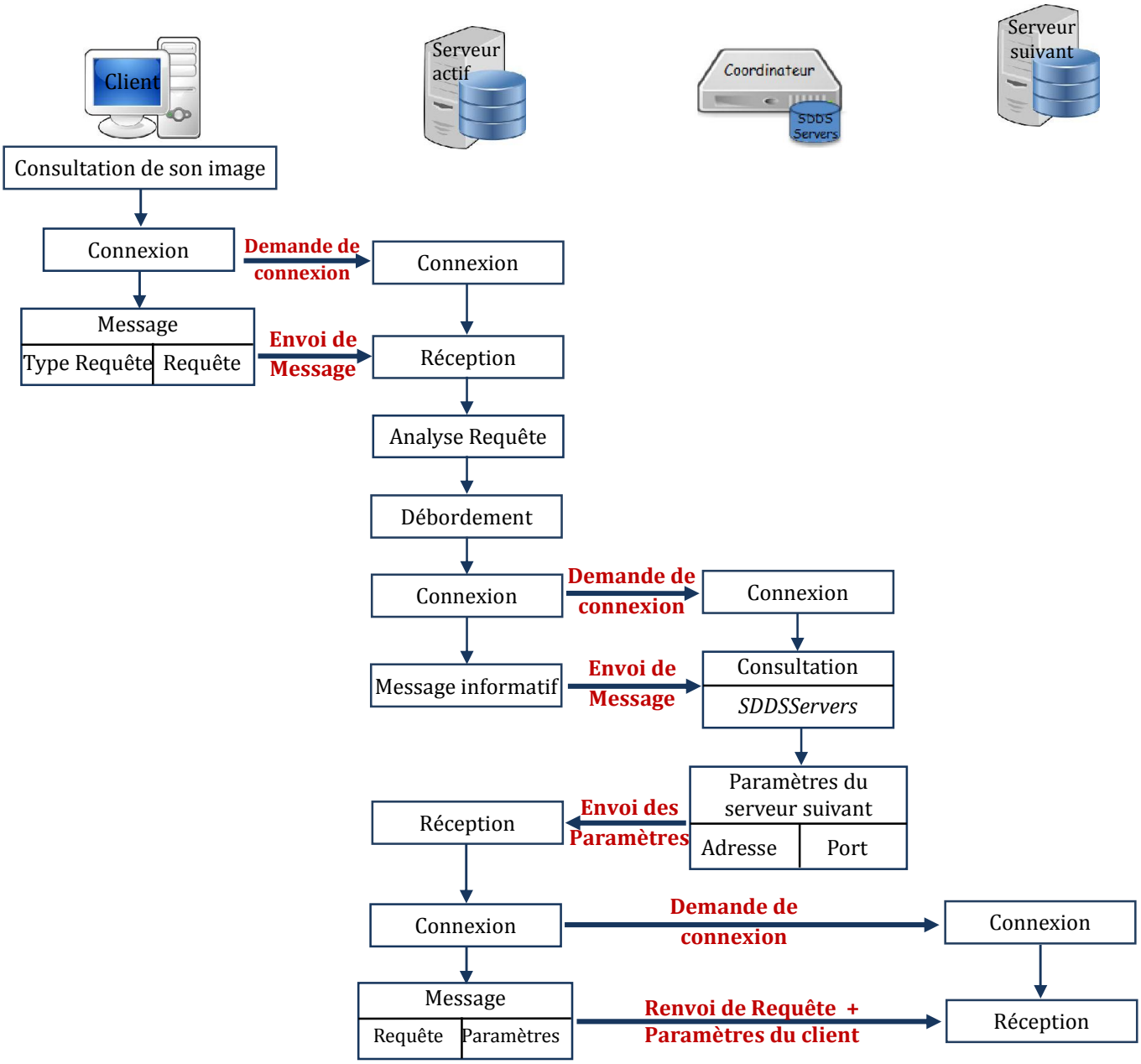
Une fois la connexion établie, le nouveau serveur reçoit les enregistrements transmis par le serveur suivant. Il se connecte alors à sa propre table pour y insérer ces enregistrements.

Après que tous les enregistrements soient insérés, tant sur le serveur suivant que sur le nouveau serveur, le serveur suivant adresse un message de confirmation d'ajout au client, indiquant que l'insertion des enregistrements a été réalisé avec succès.

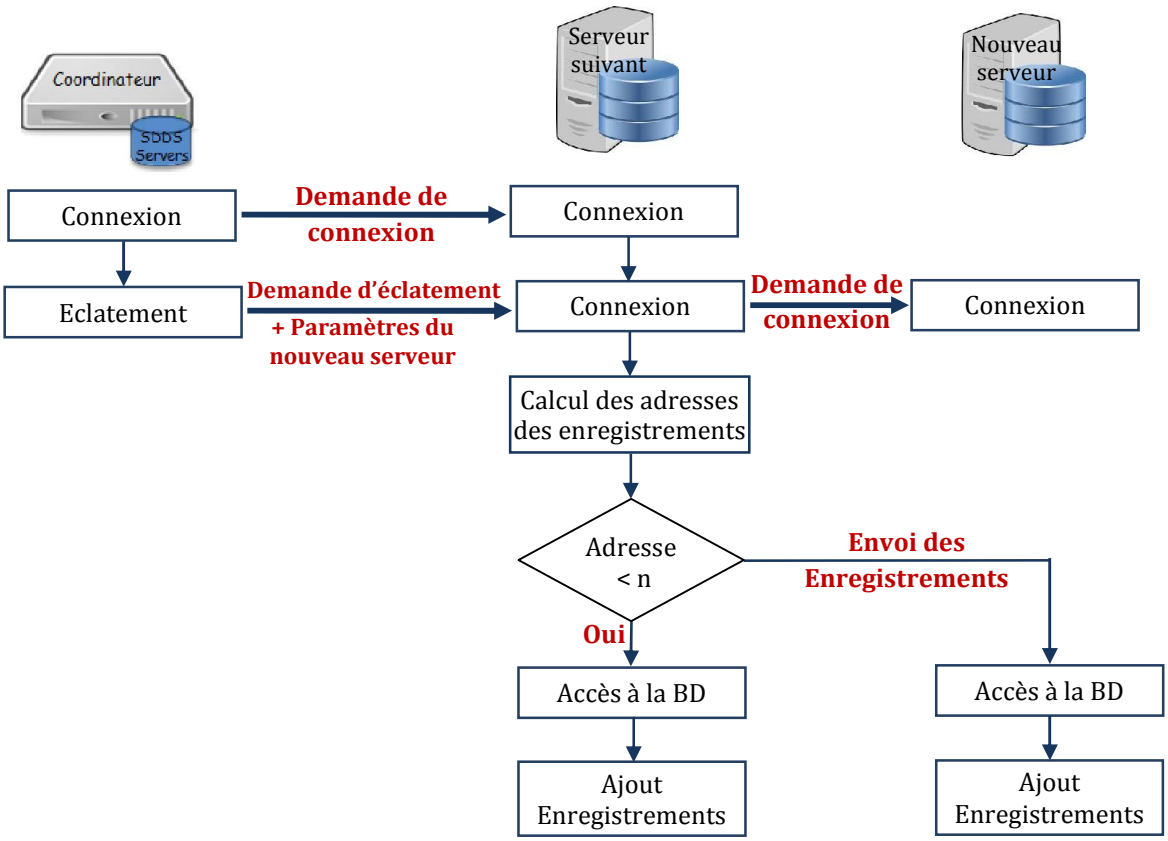
i. Mise à jour des paramètres de la table *Parameters*

Après chaque éclatement, le coordinateur met à jour sa table *Parameters*, il incrémente le compteur n de 1 pour indiquer l'adresse du prochain serveur à éclater (le serveur suivant), ainsi que le niveau de fichier i. (*Voir AlgorithmeIII.2)*

Si $n = 2^i * N$, cela signifie que le nombre total de serveurs a été atteint, n est alors réinitialisé à 0 pour pointer vers le premier serveur.



FigureV.24 : Phase1 : Débordement du serveur actif



FigureV.25 : Phase2 : Eclatement du serveur suivant pointé par n

V.9. Conclusion

Le chapitre précédent et en cours qui sont dédiés à la conception et à la mise en place des deux entrepôts de données ont permis de poser les bases solides de notre travail.

Dans un premier temps, la phase de conception a été essentielle pour définir les structures, les modèles de données et les mécanismes nécessaires pour répondre aux besoins spécifiques du système. Cette étape a permis de formaliser les schémas conceptuels et logiques de chaque entrepôt.

Ensuite, la mise en place technique de ces entrepôts a permis de transformer ces modèles en systèmes opérationnels. Cette phase a couvert des aspects cruciaux tels que la configuration des environnements, les mécanismes d'accès et l'intégration des données. Elle a également impliqué des choix technologiques pour garantir une implémentation conforme aux attentes.

Ces deux chapitres ont donc permis d'assurer une transition fluide entre la théorie et la pratique, en fournissant des entrepôts pleinement fonctionnels. Le chapitre suivant s'appuiera sur ces travaux pour analyser les résultats obtenus lors de l'exécution des requêtes sur chaque entrepôt. Cette analyse comparative mettra en lumière leurs

performances respectives et fournira un éclairage détaillé sur les points forts et les limites de chaque architecture.

Chapitre VI

Résultats et discussion

VI.1. Introduction

Après la conception et de la mise en œuvre d'un entrepôt de données classique et d'un DW_SDDS, ce chapitre expose les résultats expérimentaux obtenus en termes de messages échangés et du temps requis pour l'exécution des requêtes.

Une analyse comparative approfondie des deux architectures sera ensuite menée dans le but de déterminer celle offrant les meilleures performances pour l'implémentation d'un entrepôt de données.

Enfin, une solution sera proposée pour pallier les limites identifiées dans l'architecture DW_SDDS.

VI.2. Comparaison entre les deux architectures

Les deux critères essentiels pour évaluer la performance d'une opération donnée dans le cadre d'un système réparti sont, d'une part, le nombre de messages point-à-point échangés entre les différents sites du réseau afin de réaliser l'opération, et d'autre part, le temps requis à son accomplissement.

VI.2.1. Nombre de messages échangés

Le premier indicateur de performance à examiner, est le nombre de messages échangés entre les différents composants du système d'entrepôt de données réparti, à savoir le client, les serveurs et le coordinateur. Ce volume de messages, reflétant l'efficacité des échanges au sein de l'architecture distribuée, est illustré dans les deux tableaux suivants :

	Client - Coordinateur	**Serveur - Coordinateur**	**Total**
Requête de recherche	2	2*nb	2+2*nb
Requête d'insertion sans débordement	2	2	4
Requête d'insertion avec débordement	2	2 ou 4	4 ou 6

TableauVI.1 : Nombre de messages échangés dans un entrepôt de données classique

	Serveur - Serveur	Client - Serveur	Serveur - Coordinateur	Total
Requête de recherche	--	2*nb	--	2*nb
Requête d'insertion sans débordement	--	2	--	2
Requête d'insertion avec débordement	2	3	3	8

TableauVI.2 : Nombre de messages échangés dans un DW_SDDS

nb : représente le nombre de serveurs impliqués dans l'exécution d'une requête.

Les deux tableaux ci-dessous présentent le nombre de messages échangés au sein d'un entrepôt de données pour des requêtes de recherche et d'insertion, avec ou sans débordement. Le premier tableau se réfère à un entrepôt classique, tandis que le second illustre la même mesure pour un entrepôt utilisant l'approche DW_SDDS.

VI.2.1.1. Requête de recherche

La première ligne de chaque tableau correspond à une requête de recherche. On voit que dans un entrepôt classique, le nombre de messages échangés est supérieur de deux unités par rapport à un entrepôt DW_SDDS. Cette différence s'explique par le fait que, dans un entrepôt classique, le client doit impérativement transmettre sa requête au coordinateur. Celui-ci se charge alors de la relayer aux serveurs, de collecter leurs réponses, de les fusionner en une seule, puis de la renvoyer au client.

En revanche, dans un entrepôt DW_SDDS, la communication est plus directe : le client interagit directement avec les serveurs, sans passer par le coordinateur, ce qui a pour avantage de réduire le nombre total de messages échangés et optimiser l'efficacité des échanges.

VI.2.1.2. Requête d'ajout sans débordement

La deuxième ligne des tableaux présente le nombre de messages échangés lors d'une requête d'insertion sans débordement. On trouve que ce nombre est deux fois plus élevé dans un entrepôt classique que dans un DW_SDDS, avec respectivement 4 messages contre 2.

Cette différence s'explique par le même principe que celui observé lors d'une requête de recherche, où dans un entrepôt classique, toute communication entre le client et le serveur doit obligatoirement transiter par le coordinateur. Ainsi, aux deux messages échangés entre le client et le coordinateur, s'ajoutent deux autres messages entre le coordinateur et le serveur, doublant ainsi le volume total des échanges.

VI.2.1.3. Requête d'ajout avec débordement

La dernière ligne des deux tableaux correspond au nombre de message échangés lors d'une requête d'insertion avec débordement, où l'espace de stockage disponible au niveau du serveur actif est insuffisant pour accueillir tous les nouveaux enregistrements.

Dans un entrepôt classique, ce nombre est de 4 dans le cas où il n'y a plus d'espace disponible sur le serveur actif. La communication se fait alors entre le coordinateur et le serveur suivant, qui devient actif. En revanche, ce nombre atteint 6 lorsque la répartition des enregistrements se fait entre les deux serveurs, actif et suivant.

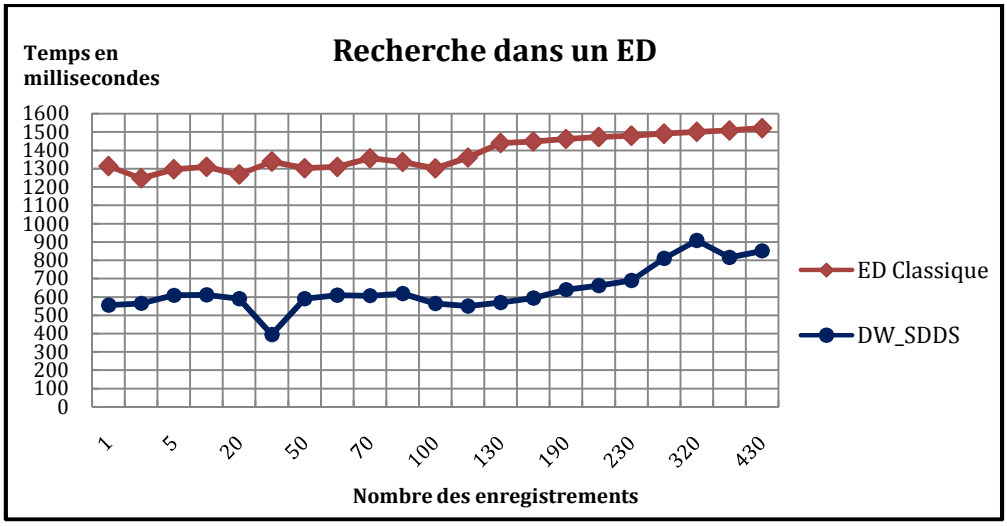
Dans un DW_SDDS, le nombre de messages s'élève à 8, car la communication implique plusieurs composants : le client, le coordinateur, le serveur en débordement, le serveur pointé, ainsi qu'un nouveau serveur qui va accueillir les nouveaux enregistrements, chacun échangeant des messages pour gérer l'insertion et le débordement.

VI.2.2. Temps d'exécution

La deuxième mesure permettant d'évaluer la performance d'un système d'entrepôt de données réparti (ou tout autre système) est le temps nécessaire pour l'exécution d'une requête, depuis son envoi par le client jusqu'à la réception des réponses par celui-ci.

Les courbes ci-dessous illustrent le temps requis pour l'exécution de chacune des trois requêtes (recherche, ajout avec débordement et ajout sans débordement) dans un entrepôt classique et DW_SDDS.

VI.2.2.1. Requête de recherche



FigureVI.1 : Estimation du temps requis pour l'exécution d'une requête de recherche

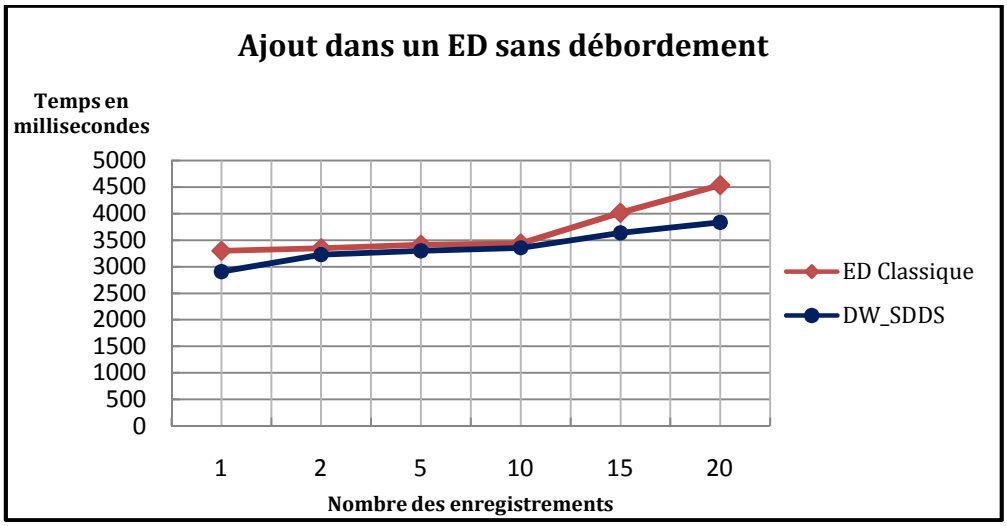
On analysant la première courbe, qui concerne une requête de recherche, on constate que le temps nécessaire pour son exécution dans un entrepôt classique est plus de deux fois supérieur à celui observé dans un DW_SDDS.

Cette différence s'explique par la première mesure de performance : le nombre de messages échangés. En effet, dans un entrepôt classique, il y a un plus grand volume de messages échangés entre le client, le coordinateur et les serveurs, ce qui entraîne un temps d'exécution plus long par rapport à un DW_SDDS, où la communication se fait directement entre le client et les serveurs, réduisant ainsi le nombre de messages et le temps global d'exécution.

Et bien sûr, il ya une relation de corrélation entre le nombre des enregistrements à rechercher et le temps d'exécution, ie : plus il y a des enregistrements à rechercher, i y aura plus de serveurs à participant, est plus le temps d'exécution est long.

Il ne faut pas oublier qu'il existe également une relation de corrélation entre le nombre d'enregistrements à rechercher et le temps d'exécution. Autrement dit, plus il y a d'enregistrements à rechercher, plus le nombre de serveurs impliqués dans le processus sera élevé, ce qui entraîne une augmentation du temps d'exécution. Cette relation est due au fait que la recherche doit être effectuée sur un plus grand volume de données réparties sur plusieurs serveurs, ce qui implique plus de communications et, par conséquent, un temps de réponse plus long.

VI.2.2.2. Requête d'ajout sans débordement

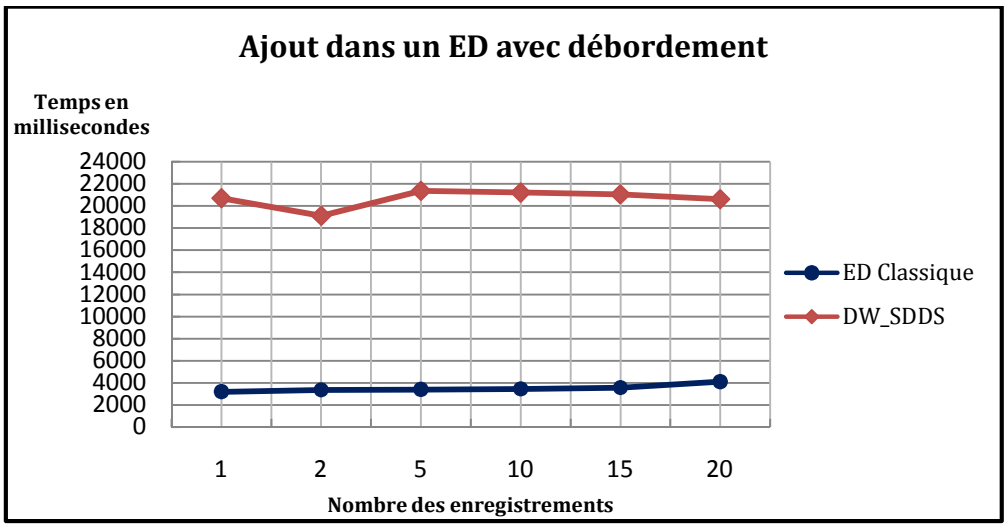


FigureVI.2 : Estimation du temps requis pour l'exécution d'une requête d'ajout sans débordement

Pour la deuxième courbe, qui représente le temps d'exécution d'une requête d'insertion sans débordement, on observe également que ce temps est plus long dans un entrepôt classique que dans un DW_SDDS.

Cela s'explique par le fait que dans un entrepôt classique, le nombre de messages échangés est plus élevé que dans un DW_SDDS. Dans un entrepôt classique, la communication doit passer par le coordinateur, ce qui ajoute des étapes et des messages supplémentaires. En revanche, dans un DW_SDDS, la communication se fait directement entre le client et les serveurs, réduisant ainsi le volume de messages et, par conséquent, le temps d'exécution.

VI.2.2.3. Requête d'ajout avec débordement



FigureV.3 : Estimation du temps requis pour l'exécution d'une requête d'ajout avec débordement

Dans la dernière courbe, il est évident que l'opération d''insertion avec débordement est nettement plus rapide dans un entrepôt classique que dans un DW_SDDS.

Cette différence s'explique par le fait qu'en cas de débordement dans un DW_SDDS, il est nécessaire de recalculer les adresses des enregistrements, tant les nouveaux que ceux déjà insérés au niveau du serveur en débordement. Ces enregistrements doivent ensuite être redistribués vers un nouveau serveur. Cette étape supplémentaire de recalcul et de redistribution des enregistrements entraîne un temps d'exécution plus long, contrairement à un entrepôt classique où aucun recalcul ni redistribution des enregistrements déjà insérés n'est nécessaire.

VI.3. Conclusion

Les résultats expérimentaux issus de la mise en œuvre des deux systèmes, l'entrepôt classique et le DW_SDDS, montrent que le DW_SDDS se révèle plus performant pour les requêtes de recherche et d'insertion sans débordement. Cette supériorité se manifeste tant par la réduction du nombre de messages échangés que par la diminution du temps d'exécution global.

Cependant, cette efficacité se trouve remise en question lors des requêtes d'insertion avec débordement. Bien que le nombre de messages échangés dans un DW_SDDS ne dépasse celui d'un entrepôt classique que de deux unités, le temps d'exécution y est nettement plus élevé. Ce ralentissement s'explique par la gestion décentralisée du débordement, impliquant la redistribution des enregistrements entre plusieurs serveurs, un processus plus complexe et coûteux en temps.

Cette différence de comportement trouve son origine dans l'architecture même des deux systèmes. Dans un entrepôt classique, le coordinateur est fortement sollicité, toute communication ou échange de messages entre un client et un serveur devant obligatoirement transiter par lui. En cas de trafic élevé, cette centralisation peut entraîner un goulot d'étranglement, limitant les performances du système. De plus, une panne du site coordinateur bloque complètement les échanges entre les clients et les serveurs, paralysant ainsi l'ensemble du système.

À l'inverse, l'un des principaux atouts du DW_SDDS réside dans le fait que la communication s'effectue directement entre le client et le serveur, sans passer par un site intermédiaire. Cette approche réduit significativement le nombre de messages échangés et, par conséquent, le temps nécessaire à l'exécution d'une requête.

Pour remédier à la problématique de l'insertion avec débordement et optimiser davantage les performances, il est proposé d'établir un seuil de stockage pour chaque serveur. Une fois ce seuil atteint, une redistribution des enregistrements vers un nouveau serveur serait déclenchée avant même la réception d'une nouvelle requête d'insertion. Cette approche permettrait ainsi d'effectuer des mises à jour périodiques de l'entrepôt de données de manière proactive, indépendamment des requêtes d'insertion initiées par les clients, réduisant ainsi les risques de surcharge et les délais d'exécution prolongés.

Chapitre VII

Conclusion et perspectives

VII.1. Résultats

Cette thèse démontre une étude approfondie sur la conception et la mise en place de deux types d'entrepôt de données distribués : les entrepôts classiques et les entrepôts en SDDS (DW_SDDS).

La distribution des données de DW_SDDS s'appuyait sur l'algorithme de hachage linéaire LH*, appliqué sur des cubes ROLAP. Dans ces cubes, les données sont stockées dans des bases de données relationnelles, et les requêtes multidimensionnelles sont traduites en relationnelles.

La comparaison des deux architectures a révélé que le DW_SDDS offre des performances supérieures par rapport à un entrepôt classique, notamment pour l'exécution des requêtes de recherche et d'ajout sans débordement. Cependant, un point faible a été identifié lors des opérations d'ajout avec débordement : celles-ci requièrent un temps supplémentaire pour recalculer les adresses des enregistrements et pour les redistribuer vers de nouveaux serveurs. Ce processus, bien que central dans les SDDS pour assurer la scalabilité et un équilibrage de charge entre les serveurs, peut impacter les performances globales du système.

Pour pallier cette limitation, une solution a été proposée consistant à définir un seuil de stockage pour chaque serveur. Dès que ce seuil est atteint, une redistribution anticipée des enregistrements vers un nouveau serveur est déclenchée, avant la réception d'une nouvelle requête d'ajout, réduisant ainsi le temps de traitement.

VII.2. Travaux futurs

Les travaux futurs viseront à étendre cette étude en implémentant le même cube de données avec d'autres techniques de SDDS, notamment le partitionnement par intervalles (comme l'algorithme RP*) et le hachage digital (CTH*). Une comparaison des résultats obtenus permettra d'identifier la solution la plus optimale et la plus performante.

Un autre aspect important qui n'a pas été abordé dans cette thèse concerne la gestion des erreurs d'adressage, bien que celles-ci puissent avoir un impact significatif sur la robustesse et l'efficacité des mécanismes présentés. Cette problématique, qui mérite une analyse approfondie, sera envisagée dans le cadre de recherches futures afin d'explorer des solutions adaptées et d'améliorer davantage la fiabilité des méthodes proposées.

Références bibliographiques

[1] T. ARCHAMBEAU, "Cours SQL. Base du langage SQL et des bases de données", Cours en ligne, https://sql.sh/.

[2] M. ARIDJ, "LH*TH: New fast Scalable Distributed Data Structures (SDDS s)", IJCSI International Journal of Computer Science Issues, Volume 11, Issue 6, No 2, November 2014. ISSN (Print): 1694-0814 | ISSN (Online): 1694-0784.www.IJCSI.org. pp. 123-128.

[3] M. ARIDJ, "Intégration des structures de données distribuées et scalables (SDDS) ordonnées dans les systèmes distribués", Thèse de doctorat, Ecole Supiereur en Informatique ESI(ex INI) Oued Smar Alger Algerie, 2013.

[4] M. ARIDJ, D. E. ZEGOUR, "TH*: Scalable Distributed Trie Hashing", IJCSI International Journal of Computer Science Issues, Vol. 7, Issue 6, November 2010, ISSN (Online): 1694-0814,

[5] M. ARIDJ, "hachage digitale compact multidimesionnel avec expansion partielle" , these de Mgister -INI 2000.

[6] M. BEDLA et K. SAPIECHA, "Scalable Store of Java Objects Using Range Partitioning", in Advances in Software Engineering Techniques, Berlin, Heidelberg, 2012, pp. 84–93, https://doi.org/10.1007/978-3-642-28038-2_7.

[7] M. BELGUIDOUM, "Programmation réseau en Java : Les sockets", Support de cours, Université Mentouri, Constantine.

[8] Y. BEN ABDEL KADER NDIAYE, "Mise en œuvre de l'architecture de communication des Structures de Données Distribuées et Scalables RP*N et RP*C", Mémoire de DEA, Université Cheikh Anta Diop de Dakar, 1998.

[9] S. BENBELGACEM: "Perspectives d'utilisation des SDDS pour l'implémentation du niveau physique d'une base de données relationnelle", Thèse de Magister, Université El Hadj Lakhder – Batna, 2011.

[10] F. BENNOUR SAHLI, "Un Gestionnaire de Structures de Données Distribuées et Scalables. Pour les Multiordinateurs Windows: Application à la Fragmentation par Hachage", Thèse de Doctorat, Université Paris IX Dauphine, 30 Juin 2000.

[11] F. BENNOUR, A. Diène, Y. Ndiaye, and W. Litwin, "Scalable and Distributed Linear Hashing LH*LH under Windows NT", SCI-2000 Orlando, Florida, USA, July 23-26, 2000.

[12] K. BOUKHALFA, "Entrepôts et fouille de données", Support de cours, Laboratoire des Systèmes Informatiques, Université des sciences et de la Technologie Houari Boumediene USTHB – Alger.

[13] Dj. BOUKHELEF, D. E. ZEGOUR, "IH*: A New Hash-Based Multidimensional SDDS", Workshop on Distributed Data And Systems, WDAS 2002, Université Paris Dauphine, 20-23 March 2002.

[14] J. CHABKINIAN, J.E. THOMAS et Sj. SCHWARZ, "Fast LH∗," Int J Parallel Prog (2016) 44:709–734 DOI 10.1007/s10766-015-0371-8.

[15] L. CHOUDER, "Entrepôt Distribué de Données. Une méthodologie pour l'emplacement optimal des données dans les entrepôts distribués", Thèse de Magister, Institut National d'Informatique, INI, Alger, Algérie, 2007.

[16] D. CIESLICKI, S. SCHAECKELER et T. SCHWARZ, "Maintaining and checking parity in highly available Scalable Distributed Data Structures", Journal of Systems and Software, Vol. 83, Issue. 4, pp. 529-542, April 2010.

[17] A. DI PASQUALE et E. NARDELLI, "A Very Efficient Order Preserving Scalable Distributed Data Structure," H.C. Mayr et al. (Eds.): DEXA 2001, LNCS 2113, pp. 186–199, 2001, Springer-Verlag Berlin Heidelberg 2001.

[18] A. DI PASQUALE et E. NARDELLI, "Scalable Distributed Data Structures: a Survey", In 3rd International Workshop on Distributed Data and Structures (WDAS'00), pages 87-111, L'Aquila, Italy, June 2000.

[19] B. DJAIL, "Validation atomique de transactions réparties pour un système de stockage à base de LH* (Distributed Linear Hashing) ", Thèse de Magister, Ecole Supérieure d'Informatique (ESI), 2010/2011.

[20] H. DJELLALI, "Bases de Données Réparties", Cours BD Avancées, Fevrier 2021.

[21] J. M. DOUDOUX, "Développons en Java", Livre en ligne, Version 1.30. 27/03/2010, Copyright (C) 1999-2010 DOUDOUX Jean Michel.

[22] E. ERTURK et K. JYOTI, "Perspectives on a Big Data Application: What Database Engineers and IT Students Need to Know", Engineering, Technology & Applied Science Research, Vol. 5, Issue. 5, pp. 850-853, Oct 2015, https://doi.org/10.48084/etasr.

[23] B. ESPINASSE, "Introduction aux entrepôts de données", Ecole Polytechnique Universitaire de Marseille, Septembre 2013.

[24] M. FARIDI MASOULE, M. A. AFSHAR KAZEMI, M. ALBORZI et A. TOLOIE ESHLAGHY, "Genetic-Firefly Hybrid Algorithm to Find the Best Data Location in a Data Cube", Engineering, Technology & Applied Science Research, Vol. 6, No. 5, pp. 1187-1194, 2016.

[25] T. HAMON, "Bases de Données Avancées, DataWareHouse", Cours, Institut Galilée, Université Paris 13, 2012.

[26] T. HAMON, "Entrepôts de données Première partie", Cours, Institut Galilée, Université Sorbonne Paris Nord, 2023.

[27] T. HAMON, "Entrepôts de données Deuxième partie", Cours, Institut Galilée, Université Sorbonne Paris Nord, 2023.

[28] C. HERBY, "Tutoriel : Apprenez à programmer en Java", livre en ligne, https://pub.phyks.me/sdz/sdz/apprenez-a-programmer-en-java.html.

[29] M. HERSCHEL, "Bases de Données OLAP", Université Paris sud, Groupe Base de données, LRI Laboratoire de recherche en informatique, 2013-2014.

[30] F. HUMBERT, "Réseau en Java", Livre en ligne, https://humbert-florent.developpez.com/java/reseau/avance/ .

[31] W.H. INMON, "Building the DataWarehouse", 4th Edition, Wiley Publishing, Inc., ISBN: 978-0-764-59944-6, October 2005.

[32] M.N. ISSAOUI et R. BOUAZIZ, "SDDS LH* TT: Une solution pour la scalabilité d'une relation temporelle de transaction standard", SETIT 2009, 5th International Conference: Sciences of Electronic, Technologies of Information and Telecommunications, Tunisie, March 22-26, 2009.

[33] H. JERBI, "Entreposage, analyse en ligne et fouille de données", Journée COMPIL " Bases de Données", 14/12/2010.

[34] H. JERBI, "Personnalisation d'analyses décisionnelles sur des données multidimensionnelles", Thèse de Doctorat, Université de Toulouse, 2012.

[35] R. KIMBLALL, "Concevoir et déployer un data warehouse", Editions Eyrolles, ISBN 2-212-09165-6, 2000.

[36] S. KRAKOWIAK, "Programmation client-serveur Sockets –RPC", Support de cours, Université Joseph Fourier, http://sardes.inrialpes.fr/people/krakowia, 2004.

[37] J.S. KARLSSON, W. LITWIN et T. RISCH, "LH*LH: A Scalable High Performance Data Structure for Switched Multicomputers", EDBT'96, pp. 573-591, Avignon, France, March 1996.

[38] R. LACHAIZE et S. KRAKOWIAK, "Communication par sockets TCP/IP: Illustration avec Java", Support de cours, Université Grenoble Alpes, Janvier 2020.

[39] W. LITWIN, R. MOUSSA et S. J. THOMAS SchWarz, "LH*RS: A Highly Available Distributed Data Storage", Proceedings of the 30th VLDB Conference, Toronto, Canada, 2004.

[40] W. LITWIN, R. MOUSSA et S. J. THOMAS SCHWARZ, "LH*RS: A Highly Available Distributed Data Structure", ACM Transactions on Database Systems, Vol. 30, No. 3, Sep 2005, pp. 1–23.

[41] W. LITWIN, M. A. NEIMAT, et D. A. SCHNEIDER, "RP*: A Family of Order Preserving Scalable Distributed Data Structures", in Proceedings of the 20th International Conference on Very Large Data Bases, San Francisco, CA, USA, Sep. 1994, pp. 342–353.

[42] W. LITWIN, S. SAHRI et S. J. THOMAS SCHWARZ, "An Overview of a Scalable Distributed Database System SD-SQL Server", D. Bell and J. Hong (Eds.): BNCOD 2006, LNCS 4042, pp. 16–35, 2006, Springer-Verlag Berlin Heidelberg 2006.

[43] W. LITWIN, H. YAKOUBEN et S. J. THOMAS SCHWARZ, "LH*RS P2P: A Scalable Distributed Data Structure for P2P Environment", 8th annual international conference on New Technologies of Distributed Systems, Lyon, France, 23-27 June 2008.

[44] G. LUKAWSKI et K. SAPIECHA, "Fault Tolerant Record Placement for Decentralized SDDS LH*", R. Wyrzykowski et al. (Eds.): PPAM 2007, LNCS 4967, pp. 312–320, 2008, Springer-Verlag Berlin Heidelberg 2008.

[45] M. MAABED, N. DENNOUNI et M. ARIDJ, "Optimizing Data Availability and Scalability with RP*-SD2DS Architecture for Distributed Systems", Engineering, Technology & Applied Science Research. Vol. 14, Issue. 5, pp. 16178-16184, Oct 2024.

[46] R. MOKADEM, F. MORVAN et A. HameUrlain, "SDDS Based Hierarchical DHT Systems for an Efficient Resource Discovery in Data Grid Systems", Springer-Verlag Berlin Heidelberg 2015. E. Simperl et al. (Ed.): ESWC 2012 Satellite Events, LNCS 7540, pp. 327–342, 2015, DOI: 10.1007/978-3-662-46641-4_25.

[47] S. NAIT BAHLOUL, "Les entrepôts de données pour le décisionnel : Concepts et notions de base", Support de cours.

[48] E. NEGRE, "Entrepôt de données", Cours en ligne, Université Paris-Dauphine, 2015-2016.

[49] M. TAMER ÖZSU et P. VALDURIEZ, "Principles of Distributed Database Systems", 3rd edition, © Springer Science+Business Media, LLC 2011, ISBN 978-1-4419-8833-1, ISBN 978-1-4419-8834-8 (eBook), DOI 10.1007/978-1-4419-8834-8

[50] M. TAMER ÖZSU et P. VALDURIEZ, "Principles of Distributed Database Systems," 4th edition, © Springer Nature Switzerland AG 2020, ISBN 978-3-030-26252-5, ISBN 978-3-030-26253-2 (eBook), https://doi.org/10.1007/978-3-030-26253-2.

[51] N. PASQUIER, "Fouillede Données: OLAP & Data Warehousing", Cours, Université de Nice Sophia-Antipolis, Laboratoire I3S.

[52] N. PRAKASH, D. PRAKASH, "Data Warehouse. Requirements Engineering. A Decision Based Approach", ISBN 978-981-10-7018-1, ISBN 978-981-10-7019-8 (eBook), https://doi.org/10.1007/978-981-10-7019-8, Springer Nature Singapore Pte Ltd, 2018.

[53] Y. RAMDANE, "Big Data Warehouse : Modèle de distribution de données dans des cubes à la volée", Thèse de Doctorat de l'université de Lyon, 21 Novembre 2019.

[54] X. REN et X. XU, "EH*RS: A High-Availability Scalable Distributed Data Structure", H. Jin et al. (Eds.): ICA3PP 2007, LNCS 4494, pp. 188–197, 2007, Springer-Verlag Berlin Heidelberg 2007.

[55] C. L. RONCANCIO, C. LABBÉ, "Bases de données réparties : fragmentation et allocation", Support de cours, www-lsr.imag.fr/Les.Personnes/Cyril.Labbe/M2P.

[56] K. SAPIECHA et G. LUKAWSKI, "Fault-Tolerant Protocols for Scalable Distributed Data Structures", R. Wyrzykowski et al. (Eds.): PPAM 2005, LNCS 3911, pp. 1018–1025, 2006.

[57] K. SAPIECHA et G. LUKAWSKI, "Scalable Distributed Two-Layer Data Structures (SD2DS)", International Journal of Distributed Systems and Technologies (IJDST), vol. 4, no. 2, pp. 15–30, Apr. 2013, https://doi.org/10.4018/jdst.2013040102.

[58] A. SEBAA, "Bases de Données Distribuées", Support de cours, École supérieure en Sciences et Technologies de l'Informatique et du Numérique – ESTIN, Enseignant au Département d'Informatique, Faculté des Sciences Exactes, Université Abderrahmane Mira de Bejaïa, Septembre 2019.

[59] A, SEDIKI, "Entrepôts de données", Support de cours. Institut Supérieur des Etudes Technologique de Kef, 2012.

[60] L. SOLER, "Les entrepôts de données", Document diffusé sous licence Creative Commons by-nc-nd, http://creativecommons.org/licenses/by-nc-nd/2.0/fr/, Janvier 2008.

[61] A. VAISMAN et E. ZIMÁNYI, "Data Warehouse Systems, Design and Implementation", Springer Heidelberg, New York Dordrecht London, Library of Congress Control Number: 2014943455, ©Springer-Verlag Berlin Heidelberg 2014, ISBN 978-3-642-54654-9 ISBN 978-3-642-54655-6 (eBook), DOI 10.1007/978-3-642-54655-6.

[62] C. VANGENOT, "Data warehouse", Cours en ligne, Laboratoire de bases de données, database laboratory.

[63] S. VIALLE, "Prog. réseau et systèmes distribués, Programmation par sockets-Java", Support de cours, http://www.metz.supelec.fr/~vialle.

[64] Y. HANAFI et S. SOROR, "LH*RS P2P: a fast and high churn resistant scalable distributed data structure for P2P systems", Int. J. Internet Technology and Secured Transactions, Vol. 2, Nos. 1/2, 2010

[65] w3schools, "Learn to code", https://www.w3schools.com.

[66] D. E. ZEGOUR, "Scalable distributed compact trie hashing (CTH*)", Information and Software Technology Vol 46, Issue 14, pp.923-9351, Nov 2004.

[67] D. E. ZEGOUR, "Cours, Structures de Données Avancées", Cours en ligne, http://zegour.esi.dz/Site%20secondaire/Cours-pg/Cours-pg.html.