# Recommandation de placement de données pour les traitements dans des lacs de données Smartgrids

Asma Zgolli

▶ **To cite this version:**

Asma Zgolli. Recommandation de placement de données pour les traitements dans des lacs de données Smartgrids. Energie électrique. Université Grenoble Alpes [2020-..], 2021. Français. NNT : 2021GRALM076 . tel-04310181

**HAL Id: tel-04310181**

**https://theses.hal.science/tel-04310181**

Submitted on 27 Nov 2023

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

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

Université Grenoble Alpes

# THÈSE

Pour obtenir le grade de

## DOCTEUR DE L'UNIVERSITE GRENOBLE ALPES

Spécialité : **Informatique**

Arrêté ministériel : 25 mai 2016

Présentée par


## Asma ZGOLLI


Thèse dirigée par **Genoveva VARGAS-SOLAR,** CR, Université Grenoble Alpes, **Christine COLLET**, PR, Grenoble INP et Co-dirigée par **Christophe BOBINEAU,** Grenoble INP
préparée au sein du **Laboratoire d'Informatique de Grenoble**
dans **l'École Doctorale Mathématiques, Sciences et technologies de l'information, Informatique**

# Recommandation de placement de données pour les traitements dans des lacs de données Smartgrids

# Data placement recommendation for the workloads in Smartgrids Data lakes

Thèse soutenue publiquement le **23 août 2021**,
devant le jury composé de :

**Madame GENOVEVA VARGAS SOLAR**
CHARGE DE RECHERCHE HDR, CNRS DELEGATION RHONE AUVERGNE, Directrice de thèse
**Madame IOANA MANOLESCU**
DIRECTEUR DE RECHERCHE, INRIA CENTRE DE SACLAY-ILE-DEFRANCE, Rapporteure
**Monsieur LADJEL BELLATRECHE**
PROFESSEUR, ECOLE SUP DE MECANIQUE ET AEROTECHNIQUE, Rapporteur
**Madame FREDERIQUE LAFOREST**
PROFESSEUR, INSTITUT NATIONAL SC. APPLIQUEES DE LYON, Présidente
**Monsieur NICOLAS TRAVERS**
DOCTEUR EN SCIENCES, DE VINCI RESEARCH CENTER, Examinateur

**Remerciements**

Je tiens à remercier tous ceux qui m'ont aidé à accomplir ce travail de recherche. Je remercie particulièrement mes encadrants au laboratoire d'informatique de Grenoble : (i) Madame Christine Collet qui m'a offert cette opportunité, qui m'a encadré, qui a diriger ma thèse pendant les 3 premières années et qui a travaillé avec moi pour construire une méthodologie de recherche et publier mes articles scientifiques. (ii) Monsieur Christophe Bobineau pour son encadrement pendant la fin de ma thèse et son aide précieuse dans la rédaction de ce mémoire. (iii) Madame Génoveva pour son aide dans la correction de ce mémoire et dans la direction de ma thèse en 2021. Et finalement (iv) Monsieur Houssem Chihoub pour ses bonnes idées qui m'ont aidé à définir ma contribution.

Je remercie chaleureusement Enedis pour le financement de ma thèse et particulièrement Monsieur Hervé Ngangmeni et Monsieur Pierre Sébastien Verdier pour leur collaboration lors de plusieurs réunions de travail, de suivi d'avancement et de mise en place des cas d'utilisation industriels.

Finalement, je remercie ma famille qui sans elle je n'aurai pas eu le courage d'aller jusqu'au bout : un grand merci à mes parents qui ont insisté sur l'importance de mes études et qui ont fait de moi la personne que je suis aujourd'hui et à mon mari qui m'a supporté moralement tout au long de ma thèse.

**Dedicace**

A ma fille Inès -née en juillet 2021- qui a donné le courage à sa maman de bien terminer sa thèse. A la mémoire de mon père qui nous a quitté très tôt (2019), qui a cru en moi et qui m'a appelé docteur bien avant la validation de ma thèse.

# Résumé

Cette thèse porte sur l'optimisation de l'accès aux masses de données générées et/ou exploitées dans la gestion des réseaux de distribution électriques intelligents ou smart grids. Ces masses de données (mesures brutes, données raffinées, historiques, etc.) sont en pratique représentées dans des modèles de données très variés (relationnel, clé-valeur, documents, graphes, etc.) et stockées dans des systèmes Big Data très hétérogènes. En effet, ces systèmes offrent des fonctionnalités variés (par exemple, certains ne peuvent effectuer une jointure), des structures de données (pour le stockage, l'indexation), des algorithmes et des performances très différentes.

L'objectif de cette thèse est d'aborder l'optimisation de workflow de traitements sur ces ensembles de données par la recommandation de placement des données sur les systèmes les plus appropriés de façon à minimiser le temps total d'exécution et en se basant sur des métadonnées décrivant les ensembles de données, les workflows et les systèmes de stockage et de traitement. Le temps total d'exécution est composé du temps de transformation et de déplacement des données, et du temps d'exécution des requêtes réécrites en fonction de ces transformations. En effet, nous explorons également la possibilité de déplacer les données d'un système à un autre s'il offre des caractéristiques intéressantes pour favoriser l'exécution des requêtes des workflows.

L'étude des techniques utilisées dans les systèmes de gestion de masses de données et des systèmes d'intégration/de médiation de masses de données nous a convaincus de l'impossibilité de définir un modèle d'estimation de coût d'exécution de plans de requêtes universel qui permette de comparer les performances des différents systèmes. Une approche intéressante est d'utiliser des techniques d'apprentissage automatique pour cela.

Nous proposons donc une approche, nommé DWS – pour *Data, Workloads and Systems* –, qui explore les différentes combinaisons de systèmes pour exécuter un workflow en éliminant les solutions où les systèmes ne peuvent exécuter tous les opérateurs d'une requête (condition de *faisabilité*) et qui respecte les règles métier quant au point de stockage des données initiales, intermédiaires ou finales (condition de *conformité*). L'estimation du temps d'exécution des différentes requêtes (de transformation de données ou extraites du workflow) se base sur l'injection de statistiques dans les systèmes, pour simuler l'exécution et ainsi récupérer les plans optimaux et éventuellement les estimations de coût, et l'estimation du temps d'exécution par apprentissage en incluant toutes les métadonnées utiles concernant les jeux de données, les workloads et les systèmes.

Pour ce faire, nous présentons dans ce document (i) un modèle de métadonnées unifié sur les données (tailles, distribution de valeurs, disponibilité, emplacement, schémas, tailles, statistiques, etc.), sur les workloads (requêtes, opérateurs, applications, statistiques, etc.) et sur les systèmes (APIs, modèles de données, modèles de distributions, modèles de stockage, etc.) ; (ii) Une architecture et des algorithmes pour le support de la recommandation de placement de données pour des workloads ; et (iii) les résultats des expérimentations effectuées à l'aide de notre prototype.

# Abstract

This thesis deals with optimising access to the masses of data generated and/or exploited in the management of smart grids. These masses of data (raw measurements, refined data, historical data, etc.) are in practice represented in very varied data models (relational, key-value, documents, graphs, etc.) and stored in very heterogeneous Big Data systems. Indeed, these systems offer various functionalities (e.g., some cannot perform a join), data structures (for storage, indexing), algorithms and performances.

This thesis aims to address the optimisation of processing workflows on these datasets by recommending data placement on the most appropriate systems to minimise the total execution time and based on metadata describing the datasets, workflows, and storage and processing systems. The total execution time is composed of the time it takes to transform and move data, and the time it takes to execute rewritten queries based on these transformations. Indeed, we are also exploring the possibility of moving data from one system to another if it offers interesting characteristics to favour the execution of workflows queries.

The study of the techniques used in data management systems and data integration/mediation systems has convinced us that it is impossible to define a universal model for estimating the cost of executing query plans that would allow us to compare the performance of different systems. An interesting approach is to use machine learning techniques for this.

We, therefore, propose an approach named DWS - for Data, Workloads and Systems. DWS explores different combinations of systems to execute a workflow by eliminating solutions where the systems cannot execute all the operators of a query (feasibility condition) and respect the business rules regarding the storage point of the initial, intermediate or final data (compliance condition). The estimation of different queries' execution time (data transformation or extracted from the workflow) is based on the injection of statistics in the systems. The objective is to simulate the execution and thus retrieve the optimal plans and possibly the cost estimates and the estimation of the execution time by learning, including useful metadata concerning the datasets, the workloads, and the systems.

# Table des matières

**CHAPITRE 1 INTRODUCTION .......... 16**

1.1 CONTEXTE ET MOTIVATION .......... 16
1.2 BIG DATA .......... 17
1.3 PROBLEME ET OBJECTIF .......... 20
1.4 APPROCHE ET CONTRIBUTIONS .......... 22
1.5 ORGANISATION DU DOCUMENT .......... 23

**CHAPITRE 2 ÉTAT DE L'ART – PLACEMENT DE DONNEES ET TRAITEMENT EFFICACE DU BIG DATA 24**

2.1 ÉCOSYSTEMES DE GESTION DU BIG DATA .......... 24
2.2 STOCKAGE DISTRIBUE .......... 33
2.3 EVALUATION DE REQUETES .......... 43
2.4 INTEGRATION/MEDIATION .......... 68
2.5 DISCUSSION ET CONCLUSIONS .......... 79

**CHAPITRE 3 META-MODELE D'UN ECOSYSTEME DE GESTION DE BIG DATA .......... 82**

3.1 ENSEMBLES DE DONNEES .......... 82
3.2 APPLICATIONS ET REQUETES .......... 89
3.3 SYSTEMES DE GESTION ET DE TRAITEMENT DE DONNEES .......... 93
3.4 DISCUSSION ET CONCLUSIONS .......... 122

**CHAPITRE 4 SYSTEME DE RECOMMANDATION DE PLACEMENT (DWS) .......... 124**

4.1 PRINCIPE .......... 127
4.2 ARCHITECTURE DE DWS .......... 128
4.3 TRANSFORMATION DES DONNEES .......... 133
4.4 OPTIMISATION DU PLACEMENT D'UN WORKLOAD .......... 142
4.5 SIMULATION D'EXECUTION DE REQUETES .......... 158
4.6 ALGORITHMES .......... 161
4.7 RECAPITULATION ET ARCHITECTURES .......... 164
4.8 RESUME DU CHAPITRE .......... 166

**CHAPITRE 5 PROTOTYPE ET EXPERIMENTATIONS .......... 168**

5.1 ARCHITECTURE DU PROTOTYPE DWS .......... 168
5.2 EXPERIMENTATION ET RESULTATS .......... 173
5.3 ARCHITECTURE LOGICIELLE DETAILLEE .......... 200
5.4 RESUME DU CHAPITRE .......... 213

**CHAPITRE 6 CONCLUSION ET PERSPECTIVES .......... 214**

6.1 RESUME ET CRITIQUE .......... 214
6.2 PERSPECTIVES .......... 215

**- ANNEXE 1 : MODELE DE DONNEES RELATIONNEL .......... 218**

**- .......... 219**

**- ANNEXE 2 : COMPARAISON DES STANDARDS DU LANGAGE SQL ET SON ADAPTATION EN BIG DATA .......... 219**

# Table des figures

Figure 1.1 Exemple d'application du traitement en flux en Big Data : cas des applications Smart Grids ....19

Figure 1.2 Exemple illustrant le problème de placement de données ....21

Figure 2.1 Exemple de données d'entrée à l'algorithme de prédiction de la demande [Hal] ....26

Figure 2.2 Visualisation de données de consommation d'énergie : variation de la moyenne des nombres de point de soutirage en fonction des plages de puissance ....27

Figure 2.3 Exemple d'un cube OLAP calculé par Apache Kylin à partir de données stockées dans HDFS [ino15] ....27

Figure 2.4 Disposition des données dans un fichier SequenceFile [GZH+19] ....34

Figure 2.5 Stockage des données dans MongoDB [Li14] ....35

Figure 2.6 Stockage physique d'un graph interrogé par la librairie Graphx d'Apache Spark ....36

Figure 2.7 Stockage physique d'un graphe de données dans Neo4j ....36

Figure 2.8 Comparaison des architectures de systèmes d'interrogation de données distribués en utilisant un modèle de référence ....37

Figure 2.9 Réplication à maîtres multiples ....39

Figure 2.10 Architecture de distribution de données paire à paire ....39

Figure 2.11 Exemple d'une grappe de données composée de deux racks [snh] ....41

Figure 2.12 Partitionnement horizontal par plage de valeurs ....42

Figure 2.13 Partitionnement par hachage et collision des partitions ....42

Figure 2.14 Partitionnement par hachage consistant ....43

Figure 2.15 Architecture fonctionnelle d'un optimiseur de requêtes [Bal12] ....44

Figure 2.16 Pseudo algorithme de l'opérateur nested loop join ....46

Figure 2.17 Pseudo algorithme de l'opérateur hash join [Zur] ....47

Figure 2.18 Exemple d'un index ayant une structure de données arbre binaire ....50

Figure 2.19 Exemple d'index bitmap (a) ....51

Figure 2.20 Exemple d'index bitmap (b) ....51

Figure 2.21 Etapes d'optimisation de requêtes dans une base de données ....53

Figure 2.22 Exemple d'analyse et de réécriture de requête SQL ....54

Figure 2.23 Exemple d'espace de recherche de plan optimal ....54

Figure 2.24 Pseudo-code de l'algorithme simulated annealing ....59

Figure 2.25 Pseudocode de l'algorithme DPsize [Feg98] ....60

Figure 2.26 Pseudocode de l'algorithme DPsub [ZCHC11] ....................61
Figure 2.27 Modèles de coûts dans les systèmes distribués ....................63
Figure 2.28 Architecture d'un système d'évaluation de requête distribué ....................66
Figure 2.29 Exemple d'évaluation de requête distribué sur deux machines en utilisant le partitionnement vertical ....................67
Figure 2.30 Exemple d'une architecture d'un médiateur [GMPQ+04] ....................69
Figure 2.31 Architecture haut niveau d'Apache Spark ....................71
Figure 2.32 Architecture distribuée d'une application Spark [Spa] ....................71
Figure 2.33 Exemple de coupe d'un graph ....................75
Figure 2.34 Architecture d'Apache calcite [BCRH+18] ....................77
Figure 2.35 Architecture du système d'intégration BigDAWG ....................78
Figure 3.1 Diagramme UML de classes des métadonnées niveau ensemble de données ....................83
Figure 3.2 Propriétés des classes pour les métadonnées de type statistiques ....................84
Figure 3.3 Diagramme de classe pour les métadonnées : statistiques sur les ensembles des données ....................86
Figure 3.4 Propriétés des classes pour les métadonnées sur les versions des données du lac ....................88
Figure 3.5 Diagramme de classe des métadonnées du niveau applicatif ....................90
Figure 3.6 Structure de la classe DataPipeline ....................91
Figure 3.7 Classification des métadonnées pour le niveau système ....................93
Figure 3.8 Diagramme de classe des «modèles de partitionnement» ....................95
Figure 3.9 Hachage consistant ....................98
Figure 3.10 Diagramme de classe des «Requêtes et API d'accès» ....................108
Figure 3.11 Diagramme de classe des métadonnées «data storage» ....................114
Figure 3.12 Modèle physique de MongoDB [Ho12] ....................116
Figure 3.13 Processus d'exécution d'une requête ....................118
Figure 3.14 Diagramme de classes «physical operators» ....................119
Figure 4.1 Première partie du workload issue du cas d'utilisation : "construction d'un bilan de la consommation d'énergie dans un smart grid" ....................124
Figure 4.2 Code du workload issue du cas d'utilisation : "construction d'un bilan de la consommation d'énergie dans un smart grid" ....................125
Figure 4.3 Plan logique du workload du cas d'utilisation : "construction d'un bilan de la consommation d'énergie dans un smart grid" ....................125

Figure 4.4 Exemple de métadonnées utilisées dans l'algorithme de construction de l'espace de recherche ....126

Figure 4.5 Diagramme d'architecture de DWS ....128

Figure 4.6 Exemple de l'espace de recherche dans DWS ....144

Figure 4.7 Exemple de deux solutions de placement de données dans DWS ....145

Figure 4.8 Réécriture de requête dans un écosystème Big Data ....146

Figure 4.9 Graphe de traitement du workload : « calcul de bilan de consommation d'énergie des clients enedis» ....147

Figure 4.10 Architecture du module de comparaison de performance dans DWS ....154

Figure 4.11 Modélisation des transformations sous formes de requêtes ....157

Figure 4.12 Algorithme de recherche de la solution de placement optimale ....162

Figure 4.13 Exemple de transformation d'un plan d'exécution en inversant l'ordre de la jointure ...163

Figure 4.14 Exemple d'espace de recherche généré par un optimiseur de placement de données ..163

Figure 4.15 Architecture détaillée de DWS ....164

Figure 4.16 Architecture du composant d'estimation du temps de réponse des requêtes du workload ....165

Figure 5.1 Diagramme d'architecture en micro services de DWS ....168

Figure 5.2 Architecture technique du module de recommandation....170

Figure 5.3 Architecture technique du micro service « manipulation et accès aux métadonnées »....172

Figure 5.4 Courbe ROC du modèle de classification selon la conformité....181

Figure 5.5 Distribution des valeurs des variables "taille des données" et "temps de réponse" pour l'expérience de sélection de données pour Apache HBase/ Phoenix) ....183

Figure 5.6 Nuage de points de la variable "temps de réponse" en fonction de la variable "nombre d'enregistrement" ....184

Figure 5.7 Comparaison de la corrélation des nombres d'enregistrements avec la corrélation avec la taille de données ....184

Figure 5.8 Distribution des valeurs des attributs "cout" et "avg" qui ont été collectées des résultats de l'expérience 2 ....186

Figure 5.9 Matrice de corrélation des attributs observés pour l'expérience numéro 2 : influence des fonctions d'agrégation ....186

Figure 5.10 *Matrice de corrélation des variables : "number_of_groups" et "execution_time"* ....187

Figure 5.11 Diagrammes de distribution des valeurs des caractéristiques générées pour le problème de prédiction ....189

Figure 5.12 Diagrammes de distribution de données pour quelques variables de l'expérience 3 (2) 190

Figure 5.13 Générer l'ensemble de données d'apprentissage à partir des requêtes et des ensembles de données historiques ........................................................................................................191

Figure 5.14 Transformations effectuées sur une requête pour obtenir le tableau des caractéristiques ........................................................................................................................................192

Figure 5.15 Comparaison du temps de réponse prédit et mesuré....................................................197

Figure 5.16 Hiérarchie du prototype de DWS ...................................................................................201

Figure 5.17 Diagramme de classes du paquet : "fr.uga.lig.hadas.placement.connection.catalog" ....202

Figure 5.18 Diagramme de classes du paquet : "fr.uga.lig.hadas.placement.feasability" .................203

Figure 5.19 Diagramme de classes du paquet : "fr.uga.lig.hadas.placement.integration.model.postgres " .......................................................................................................................................204

Figure 5.20 Diagramme de classes du paquet : "fr.uga.lig.hadas.placement.simulation" .................206

Figure 5.21 Diagramme de classes du paquet : "fr.uga.lig.hadas.placement.plan.search.space.exhaustive".....................................................207

Figure 5.22 Structure de la classe QueryGraph .................................................................................209

Figure 5.23 Diagramme de classes qui implémentent la réécriture de requêtes et un exemple de transformation .................................................................................................................210

Figure 5.24 Diagramme de classes qui implémentent la génération du code à partir d'un plan de requête ........................................................................................................................................212

# Table des tableaux

Tableau 2.1 Avantages et inconvénients des modèles de données en big data ..................................29

Tableau 2.2 Comparaison des APIs des modèles de données en Big Data ..........................................31

Tableau 2.3 Tableau comparatif des solutions de réplication...............................................................40

Tableau 2.4 Algorithmes d'agrégation et leurs complexités ...............................................................48

Tableau 2.5 Complexité de quelques opérateurs physiques d'accès aux données..............................52

Tableau 2.6 Résumé des algorithmes de recherche de la solution optimale.......................................61

Tableau 2.7 Récapitulation des modèles de coûts dans les systèmes Big Data ...................................64

Tableau 2.8 Comparaison des différentes versions des APIs de Spark SQL .........................................73

Tableau 2.9 Exemple de fonctions de l'API RDD de Spark....................................................................74

Tableau 2.10 Exemples d'opérateurs de l'API Graphx de Spark...........................................................76

Tableau 2.11 Comparaison des systèmes d'intégration en Big Data ..................................................79

Tableau 4.1 Table de transformation entre les modèles de données.................................................139

Tableau 4.2 Types d'indexes et compatibilité .....................................................................................140

Tableau 4.3 Types de partitionnement et compatibilité .....................................................................141

Tableau 4.4 Exemples d'opérateurs physique et leurs complexités en fonction des métadonnées systèmes ............................................................................................................................................150

Tableau 4.5 Résumé des variables utilisées dans le problème de prédiction de temps de réponse de quelques travaux existants.................................................................................................................156

Tableau 4.6 Résumé des statistiques utilisés dans l'optimiseur de requête de quelques moteurs d'exécution .........................................................................................................................................159

Tableau 5.1 Détails sur les ensembles de données utilisés dans l'expérimentation...........................175

Tableau 5.2 Requêtes du cas d'utilisation étudié................................................................................177

Tableau 5.3 Un exemple de caractéristiques utilisés dans le problème de prédiction de la faisabilité ...........................................................................................................................................................179

Tableau 5.4 Métriques de performance de la prédiction conduisant au choix de l'algorithme le plus adéquat ..............................................................................................................................................179

Tableau 5.5 Un exemple de caractéristiques utilisés dans le problème de prédiction de la conformité ...........................................................................................................................................................180

Tableau 5.6 Transformation des caractéristiques et initiation de la variable à prédire......................181

Tableau 5.7 Distribution des valeurs des caractéristiques générées pour le problème de prédiction ...........................................................................................................................................................189

Tableau 5.8 Corrélations avec l'objectif de prédiction pour quelques variables de l'expérience 3 ....190

Tableau 5.9 Résultats de l'apprentissage du temps de réponse d'un workloads exécuté dans un magasin de données orienté graph : Neo4J ....... 193

Tableau 5.10 Résultats de l'expérience numéro 2 d'apprentissage du temps de réponse d'un workloads exécuté dans un magasin de données orienté graph : Neo4J ....... 195

Tableau 5.11 Résultats de l'apprentissage du temps de réponse d'un workloads exécuté dans un magasin de données orienté famille de colonnes : Apache HBase ....... 196

Tableau 5.12 Résultats de l'apprentissage du temps de réponse d'un workloads exécuté dans un moteur de traitements de données distribuées : Apache Spark ....... 197

Tableau 5.13 Résultat du déroulement de l'algorithme de recommandation sur un exemple simple ....... 200

# Chapitre 1 Introduction

## 1.1 Contexte et motivation

Les Smart Grids sont des réseaux de distribution d'électricité intelligents offrant une communication bidirectionnelle entre les clients et les centres de distribution d'énergie. Contrairement aux réseaux électriques traditionnels, ce type de réseau génère des données massives. En effet, pour transformer un réseau de distribution électrique en Smart Grids, les services publics déploient massivement des capteurs et actuateurs distribués sur tous les composants du réseau : les transformateurs, les concentrateurs de données, directement sur les lignes électriques et surtout sur des compteurs intelligents à l'échelle d'un pays ou bien d'un continent. Par exemple, Enedis – le premier opérateur de distribution d'électricité en France et un des plus importants en Europe – a prévu de déployer en 2021, plus de 35 millions de compteurs intelligents.

Dans ce contexte, plus de données sont collectées sur les profils de consommation des clients et le comportement du réseau (ou grille) d'électricité. Ces données sont le cœur de ce type de réseau. Elles sont critiques pour la gestion efficace des ressources d'énergie et elles offrent de nouveaux niveaux d'efficacité pour les applications métiers. Ces applications comme la maintenance prédictive ou bien la prédiction de la demande sont principalement de nature analytique.

La collecte des mesures dans les Smart Grids peut être réalisée chaque seconde sur l'amplitude du réseau. Ces mesures subissent des traitements en continu ou bien elles sont stockées directement sans prétraitement.

En plus des données des capteurs et des mesures effectuées sur le réseau, les Smart Grids gèrent et traitent des données variées comme les données météorologiques, les données des réseaux sociaux et les données des profils de clients afin d'affiner analyses et prédictions.

L'écosystème des données pour les Smart Grids est par conséquent un écosystème de masses données hétérogènes. Il est basé sur un lac de données et plusieurs espaces de projet dédiés aux besoins des utilisateurs. Ces utilisateurs ont des niveaux d'expertise différents. Ils peuvent être des spécialistes de données, des informaticiens, des scientifiques ou bien des experts métier.

Deux problèmes majeurs existent dans ce type d'écosystème. Le premier est une conséquence du rôle principal (de la définition même) du lac de données. En effet, ce dernier permet une ingestion de données dans leurs états bruts sans effort de transformation ni d'intégration. Le manque d'information sur l'organisation de données, sur leurs structures ou schémas ainsi que sur l'historique de l'évolution de ces schémas rend difficile l'accès et l'exploitation des données et contribue à l'apparition de silos de données entraînant une perte de productivité, la duplication de données et traitements, et de mauvaises manipulations.

Des travaux existants ont adressés ces problèmes dans le cadre d'un lac de données en : (i) proposant une architecture par zones qui sépare les données du lac en données en données brutes, données des applications et données archivés [SD21] ou bien (ii) en utilisant des modules de gestion de métadonnées pour contrôler l'ingestion et la qualité des données dans un lac de données [HQZ18]. Dans cette thèse on s'intéresse plus à la deuxième solution, et on propose d'utiliser les métadonnées qui sont transversales à l'écosystème pour localiser et contextualiser les données.

Le deuxième problème et celui qui nous intéresse le plus est l'optimisation de l'architecture de l'écosystème dans l'objectif d'assurer l'efficacité des traitements dans ce dernier.

Les motivations pour travailler sur ce sujet sont multiples. Les écosystèmes Big Data sont complexes, ils sont composés d'utilisateurs divers, ayant des expertises différentes et qui ne maitrisent pas

totalement les technologies Big Data. Il est donc nécessaire de proposer un tel outil qui permettra d'optimiser leurs traitements. D'un autre côté les modèles de données et les systèmes en Big Data sont variés. Ils ne supportent pas tous les types d'opérateurs des workloads d'analyse de données OLAP. Il est donc évident de considérer dans la recommandation, différentes variantes de moteurs de stockage et de moteurs de traitement pour le placement effectif des données. Traditionnellement pour déterminer la meilleure solution de placement les propriétaires des données utilisent la technique du benchmark. Par contre, le benchmark n'est pas une solution générique et adaptable pour garantir la performance de l'exécution du workload à long terme. Cette méthode est spécifique à l'application testée (elle n'évalue la performance que pour le domaine de valeurs des ensembles de données testé ou bien au workload défini). Notre méthode est basée sur les statistiques et les métadonnées pour évaluer la performance. Ce qui la rend plus adaptive aux changements du domaine de valeurs. Mais la motivation principale pour ce projet de recherche est que le domaine d'optimisation de requêtes en Big Data est un sujet d'actualité, nos travaux sur le critère de performance est une contribution intéressante dans le domaine. Comment peut-on donc trouver la meilleure combinaison de systèmes qui permet d'exécuter un workload particulier de bout en bout dans un écosystème Big Data en respectant le critère de performance ? Selon quels critères sélectionner les systèmes et comment modéliser la performance dans un système Big Data ? Et pour finir comment doit-on utiliser les métadonnées d'un écosystème Big Data pour tirer des recommandations sur l'optimisation du traitement ?

## 1.2 Big Data

Le déluge des données qui a vu le jour avec le progrès des domaines de l'Internet des objets (IOT) et de la communication machine à machine (M2M), du web et des réseaux sociaux, des réseaux télécom et de la 5G, et de l'industrie et des Smart Grids. Les solutions de gestion de données traditionnelles ne sont plus capables de gérer des masses aussi importantes de données d'où le besoin de créer de nouvelles solutions qui passent à l'échelle avec un moindre coût. Pour répondre à ce besoin, le Framework Hadoop et les solutions NoSQL ont vu le jour pour révolutionner les projets de gestion, de traitement et d'analyse de données, surtout en entreprise, mais aussi dans le domaine scientifique. Le point fort de ces solutions est leur capacité à gérer des données plus massives et plus variées en utilisant des serveurs commodes et sans avoir besoin à acquérir des ressources de calculs couteuses.

Dans les sous sections suivantes, nous présentons la définition du Big Data de laquelle découle les « Vs » qui caractérisent ce domaine, les différentes applications du Big Data et les processus de traitements qui constituent le cycle de vie de ce type de projets. Je conclus la section par une énumération des problèmes qu'on peut trouver dans ce domaine.

### 1.2.1 Définition générale

La fondation TechAmerica [PBHP18] définit le Big Data comme « un terme qui décrit de larges volumes de données, ayant une vélocité (vitesse) élevée, une complexité et une variété importantes, au point d'avoir besoin de techniques avancées pour la capture, le stockage, la distribution, la gestion et l'analyse d'information ».

Cette définition couvre principalement les caractéristiques principales du Big Data. Ces derniers sont généralement connus comme les « Vs » du Big Data. Au départ, ils été présentés comme les 3 V et ils incluent le volume, la vélocité et la variété. De nos jours, on parle même de 7 V en ajoutant la véracité, la valeur, la visualisation et la variabilité.

- *Volume*

Depuis quelques années la taille des données a augmenté exponentiellement [Kah11]. Les systèmes d'informations actuels génèrent des Petabytes, voire des Zetabytes, de données quotidiennement. Par exemple, dans le web chaque utilisateur est capable de générer le double de sa production chaque 18 à 24 heures [Cor15]. Dans les Smart Grids, l'ensemble de l'infrastructure générera des Petabytes en 2020 [adlC]. En effet, les compteurs communicants installés par Enedis génèrent quotidiennement des millions de mesures par seconde à partir de capteurs installés massivement partout en France. Ces données sont collectées puis envoyées à un système d'information centralisé et plusieurs traitements de gestion et d'analyses sont effectué sur une vue globale de ces données.

Les solutions de gestion de données classiques ne sont plus capables de gérer plus que des Terabytes de données, il a fallu inventer de nouvelles solutions qui sont capables de gérer et traiter les données à grandes échelles dans des délais raisonnables.

- *Variété*

Les applications Big Data traitent une riche diversité de données. Les ensembles de données dans ce type d'applications sont structurés, semi-structurés ou non structurés. Les données sont traitées sous forme de flux, en flots ou bien d'une manière réactive (par des requêtes).

Dans les écosystèmes de données Smart Grids les sources de données sont très variées ; on y trouve des données des réseaux sociaux, qui peuvent inclure des données semi- ou non-structurés (par exemple des images ou bien du texte brut), des données météo qui sont de type spatiau-temporel ou bien des données du réseau qui peuvent avoir un modèle de données orienté graphe. De plus, dans ce type d'écosystèmes, il existe des applications de traitement de données en flux (par exemple les applications de collecte de mesures électriques) ainsi que de traitement de données au repos.

- *Véracité*

En Big Data, il existe toujours des problèmes de valeurs manquantes, de données dont on ne connaît pas l'origine, de données dupliquées, … Il est nécessaire de mettre en place des processus d'assurance qualité pour garantir la fiabilité des décisions prises suite à l'analyse faite sur ces données massives et éviter de présenter des résultats erronés dans les applications Big Data. L'intérêt de vérifier l'aspect de la véracité de données est surtout perçu pour les données collectées des sources externes ou bien des mesures des capteurs. Effectivement, pour ce type de données, il est possible d'avoir des valeurs manquantes ou bien erronées. Ce risque croît dans le cadre des applications de collecte de mesures physiques en temps réel.

- *Vélocité*

Les utilisateurs exigent de plus en plus de réactivité et de rapidité du traitement de leurs masses de données. Mais principalement, les nouvelles applications de collecte en temps réel de données des dispositifs de l'internet des objets (Iot), par exemple des compteurs communicants des Smart Grids, des capteurs installés dans le réseau du Smart Grids, sous forme de flux reflètent parfaitement la caractéristique de vélocité en Big Data. Ce type d'applications a besoin de technologies qui sont capables de gérer et traiter des données massives en continu.

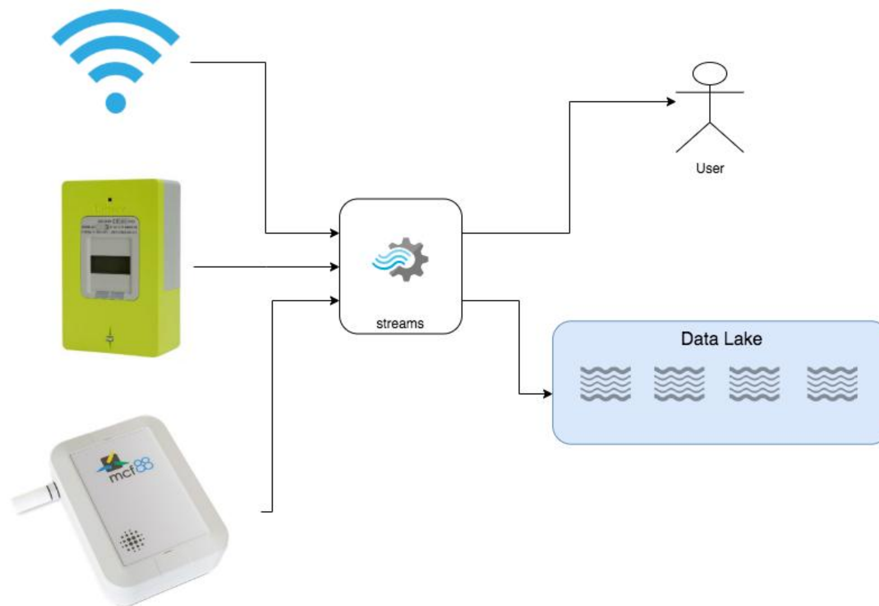



*Figure 1.1 Exemple d'application du traitement en flux en Big Data : cas des applications Smart Grids*

- *Valeur*

Le but ultime de la Big Data est de produire une valeur métier à l'entreprise. On souhaite conduire des prises de décision pilotées par les données (en anglais : data-driven). D'où l'intérêt de gérer et collecter des masses aussi importantes de données. Dans ce cadre, on trouve en Big Data des applications d'analyse de données, de business intelligence (BI) et d'intelligence artificielle.

En reprenant notre exemple sur les Smart Grids, on trouve plusieurs applications qui utilisent les masses de données générées par le réseau électrique, par exemple dans le cadre de l'analyse et de la prédiction de la demande en énergie, la construction de bilans, …

- *Variabilité*

En Big Data, avec la multiplication des sources de données et l'amplitude des projets (et les équipes) dans l'écosystème, les données ne sont plus standardisées et ne suivent plus une définition stable. Ceci résulte en la création de plusieurs convenions ; par exemple référencer les valeurs manquantes par « NaN » selon les usages en python, ou bien en utilisant « null » ou encore le nombre « 0 ».

La variabilité peut être aussi au niveau des types des données et des sources des données. Ceci est un résultat de l'absence de structure et de schéma prédéfinis des données et des projets d'intégration.

Finalement, cette caractéristique est aussi perçue dans la performance des systèmes d'accès aux données et de traitement. Le temps de réponse est très variable et ceci est dû au temps de chargement des masses de données et de transmission dans le réseau qui est généralement inconsistant.

- *Visualisation*

On parle parfois du défi de visualisation en Big Data. Ce défi est impliqué par les autres aspects, notamment par le manque ou l'absence de structure. Par exemple la visualisation de données à plusieurs dimensions, et spécifiquement dans le cas de valeurs manquantes ou de structure non-régulière des données, peut mener à des difficultés de visualisation ou bien à des interprétations qui ne sont pas fiables. La visualisation a besoin généralement de mécanismes de traitement en mémoire (en anglais : in-memory processing), ce qui n'est pas évident avec l'augmentation du volume des masses de données.

## 1.3 Problème et objectif

L'objectif général du travail de recherche que nous menons, sur la gestion de données pour les Smart Grids est de proposer et évaluer un modèle d'architecture de système de lac de données mieux adapté pour collecter, nettoyer, stocker et traiter les ensembles de données variés des Smart Grids. Dans cette architecture la gestion des métadonnées est alors nécessaire pour orchestrer et maîtriser les processus de l'écosystème et garder trace des ensembles de données.

Disposer de métadonnées permet également de mieux maîtriser la complexité de l'architecture due aux volumes importants de données, la variété des ensembles de données (brutes et raffinées, courantes et historiques, données de mesure compteurs et données sociales, etc.), leurs provenances, et la variété des systèmes de gestion et de traitement de données utilisés (organisation des données, modèle de configuration, choix de partitionnement de données, monitoring, métriques, etc.). Une bonne gestion de ces derniers présente une solution aux défis liés à l'organisation des données dans cet écosystème.

### 1.3.1 Performance du traitement

Le deuxième problème est en relation avec la performance du traitement. Effectivement, les workloads peuvent reposer sur un système de gestion de données comme les systèmes relationnels tels que Teradata [AKGC+13], magasins de documents comme MongoDB [Aut21], des magasins de stockage en colonnes comme Cassandra [Cas], magasins de paires clés / valeurs tels que Riak [ZS17] ou magasins de graphes des données tels que Neo4J [Neo]. Les workloads peuvent également s'appuyer sur les moteurs de traitement basés sur MapReduce comme Apache Hive ou sur les moteurs de traitement basés sur MPP comme Apache Spark [HK17]. Cependant, ces systèmes ne garantissent pas de la même manière les performances, la latence, l'évolutivité, la cohérence et la disponibilité.

La définition des workloads nécessite une connaissance approfondie des systèmes de gestion des données. Cela peut être un problème important étant donné que tous les utilisateurs ne possèdent pas nécessairement de telles compétences. À titre d'exemple d'utilisateurs de l'écosystème Smart Grid, nous considérons les scientifiques des données. Ces utilisateurs dans leur travail quotidien ont besoin d'accéder, de croiser et de stocker des jeux de données, des résultats d'expérimentation, … Dans ce contexte, nous avons identifié le problème de placement de données qui vise à assister ces utilisateurs dans la prise de décision vis-à-vis le système de gestion de données le plus adapté pour leurs besoins.

Une description semi-formelle peut être proposé pour le problème de placement comme suit. On considère un ensemble de données S composé de n tuples $d_i$ : S= $\{d_1 .. d_n\}$ et une requête Q composé de m opérateurs $op_i$ (e.g select / project / join …) telle que Q=$\{op_1 .. op_m\}$.

Pour un ensemble de k requêtes W=$\{Q_1..Q_k\}$ on souhaite déterminer les systèmes de stockage et de traitement de données qui sont capables (i) d'exécuter le workload W (la charge de travail), (ii) de garantir l'efficacité de l'exécution et (iii) de respecter des contraintes de conformité avec l'écosystème de données exprimé par les utilisateurs ou bien par l'unité de gouvernance des données.

Pour illustrer le problème d'une manière plus concrète, considérons l'exemple de requête suivant :

Q= select predicates from D1 join D2 join D3 on join_keys

L'utilisateur souhaite la meilleure combinaison de systèmes de stockage et de traitement de données qu'il doit utiliser pour garantir l'exécution optimale de sa requête. La figure suivante illustre le principe. Pour les sous requêtes $Q_0$ et $Q_1$ de Q, la solution énumère différentes configurations de placement de

données, en faisant varier pour les ensembles de données les systèmes de stockage et pour les sous-requêtes les moteurs d'exécution. Elle évalue aussi d'autres propriétés selon les capacités des différents systèmes comme leurs algorithmes d'indexation, leurs schémas de partitionnement et leurs propriétés physiques de stockage. Dans cet exemple nous considérons les systèmes : MongoDB, Teradata , PostgreSQL et SparkSQL.

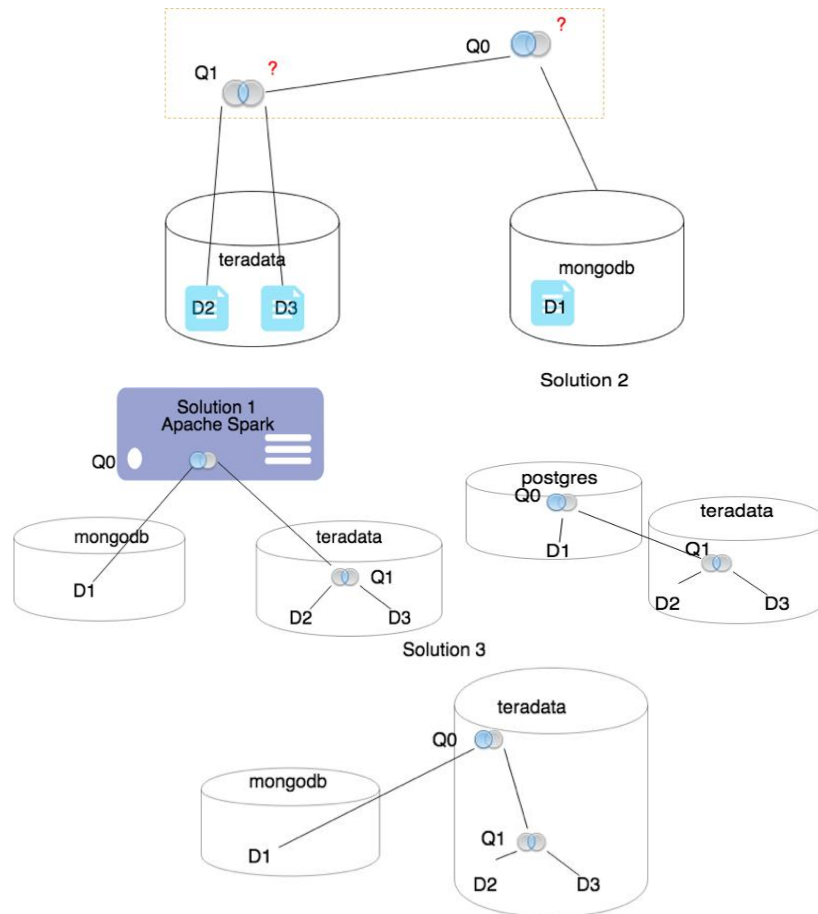



*Figure 1.2 Exemple illustrant le problème de placement de données*

Ce problème placement des ensembles de données est basé sur le workload dans un écosystème de gestion de données massives et on vise dans notre étude à aborder les problèmes suivants : (i) Les solutions Big Data d'un écosystème de données smart grids – notamment les systèmes de fichiers distribués, les bases de données NoSQL et les bases de données relationnelles – ont des interfaces très spécifiques et diverses. Ces solutions ne suivent plus le principe « one size fits all » pour l'accès et le traitement des données, ils sont dédiés aux workloads et aux cas d'usage et nécessitent de faire des compromis dans l'objectif de supporter l'exécution sur de grands volumes de données. (ii) En outre, le placement initial des ensembles de données est d'une grande importance, car nous voulons minimiser les mouvements de données entre les systèmes. En effet, la migration des données est très coûteuse, le chargement de 1 441 548 enregistrements stockés dans le magasin de données Apache HBase en utilisant le moteur d'exécution Apache Phoenix coute à l'utilisateur 793.256 sec(s) dans un serveur dédié qui n'est pas utilisé par d'autres workloads. Nous souhaitons identifier, à un stade précoce, le placement qui corresponde au mieux au workload et qui garantisse l'efficacité du traitement. Une solution à ce problème consiste à prendre en compte les propriétés des workloads pour recommander le placement optimal des ensembles de données. Considérer uniquement des statistiques sur les ensembles de données représente une solution incomplète à ce problème. (iii) le troisième problème adressé concerne l'exécution hybride des requêtes. En effet, dans les requêtes qui

croisent deux ou plusieurs ensembles de données, l'exécution dans des systèmes qui supportent nativement les opérateurs de croisement (jointure) est plus performante que dans les systèmes qui chargent les données en mémoire et par la suite exécutant la requête hybride. Une solution possible à ce problème consiste à analyser des métadonnées des workloads pour identifier les ensembles de données accédés par une même requête. Le stockage de ce type d'ensembles de données corrélés dans le même magasin de données garantit de meilleures performances pour les workload qui sont composés de requêtes de croisement de données. (iv) Le quatrième problème adresse un aspect fonctionnel d'un écosystème basé sur un lac de données. Effectivement, le résultat du placement peut être sémantiquement incorrect : le fait de permettre à l'utilisateur de choisir librement le système de données en fonction de ses besoins peut entraîner des erreurs de placement et des incohérences avec les règles de l'entreprise. Intégrer des règles métier dans l'algorithme de placement pour vérifier la conformité avec l'écosystème de données permet d'adapter davantage le résultat du placement dans l'environnement. (v) Et finalement le dernier problème porte sur la cohérence de l'exécution d'un workload complet. L'exécution d'un workload sur des données interdépendantes peut faire appel à des opérations de transformation, de déplacement ou de copie de données. Il est nécessaire de tenir compte de ces opérations dans l'algorithme de recherche de placement optimal. Pour contourner ce problème, nous proposons de considérer la solution suivante dans notre étude : dans le cas de déplacement / duplication d'un jeu de données pour améliorer l'exécution de la charge de travail, l'estimation du coût de l'opération de déplacement / duplication doit être combinée à l'estimation du coût total du workload.

### 1.3.2 Objectif général

L'Objectif de cette thèse est de trouver une réponse à ces questions et surtout de résoudre le problème de recommandation sur le placement des données en se basant sur les caractéristiques des systèmes, des workloads et des ensembles de données en respectant les règles de conformité et en assurant l'efficacité de l'exécution du workload. La Recommandation considère des applications Big Data qui sont exécutées **fréquemment** dans l'écosystème de données Smart Grids. Au repos, on collecte des informations sur les données, les systèmes et surtout les workloads puis on utilise une stratégie de recherche exhaustive pour trouver la solution optimale de placement.

La résolution de notre problème fait intervenir deux fonctionnalités majeures : le traitement des métadonnées et la recommandation sur le placement des données massives. Ces deux briques sont complémentaires, mais présentent des défis différents et indépendants. Pour répondre à ces défis, nous avons utilisé une méthodologie composée par 4 étapes : (i) l'Identification des métadonnées de l'écosystème. (ii) l'élaboration d'un état de l'art sur les systèmes de gestion de métadonnées et trouver un service original de métadonnées. (iii) l'étude des travaux existant sur le placement des données. (iv) et finalement la conception et le développement d'un prototype. En effet, la première partie de ce travail a comme objectif d'identifier les métadonnées dans les lacs de données, les modules qui les manipulent et à concevoir un schéma de métadonnées pour le lac de données et son écosystème qui vont être une source de données pour l'algorithme de placement. La deuxième partie du travail était consacré à définir le problème de placement de données, de formaliser ce problème, de concevoir l'algorithme et l'architecture et d'identifier les méthodes nécessaires pour implémenter la solution. L'expérimentation et l'évaluation du prototype a été faite sur une machine locale, un serveur distant et une grille de calcul expérimentale : Grid5000.

## 1.4 Approche et contributions

- Conception d'un schéma de métadonnées pour les niveaux systèmes, ensemble de données et workloads modulaire et extensible.

- Formulation du problème de placement de données dans un écosystème Big Data complexe.
- Proposition des critères de faisabilité, conformité et performance.
- Conception de l'architecture de placement de données.
- Conception de l'architecture de prédiction de la performance du workload en utilisant les algorithmes d'apprentissage.

## 1.5 Organisation du document

- **Chapitre 1 : Introduction**
  Dans ce chapitre, nous avons le contexte général du travail. Nous avons par la suite présenté le problème et l'intérêt de travailler dessus et finalement nous avons décrit la méthode suivie pour résoudre le problème.

- **Chapitre 2 : État de l'art - Écosystèmes Big Data**
  Le second chapitre est un état de l'art sur le domaine du Big Data. Nous présentons en général le domaine et les écosystèmes Big Data et leurs architectures ensuite nous détaillons les modèles de données et les systèmes existants. On termine le chapitre par une présentation des différentes solutions d'intégration de données en Big Data et d'une étude sur l'évaluation de requête en Big Data et un focus sur les techniques de l'optimisation de requête et ses limites en Big Data

- **Chapitre 3 : Méta-modèle d'un écosystème de gestion de Big Data**
  Dans ce chapitre, nous décrivons la première contribution de ce travail qui est un modèle de données pour les métadonnées qu'un écosystème Big Data et son lac de données auront besoin. Ce modèle est composé de trois niveaux : un modèle de métadonnées pour les ensembles de données, un modèle de métadonnées pour le workloads et un modèle de métadonnées pour les systèmes.

- **Chapitre 4 : Système de recommandation de placement (DWS)**
  Ce chapitre est le chapitre le plus important de la thèse. Dans ce chapitre, nous détaillons l'architecture de DWS, un système de recommandation original sur les solutions de stockage et d'interrogation de données massives. Nous décrivons dans ce chapitre la conception et les algorithmes des modules d'optimisation, de simulation de l'exécution, et de réécriture de requête.

- **Chapitre 5 : Prototype et expérimentation**
  Le chapitre suivant est composé de deux parties : une première partie sur l'architecture technique du prototype de notre solution et une deuxième partie sur l'expérimentation et les résultats obtenu pour notre prototype.

- **Chapitre 6 : Conclusion et perspectives**
  Le dernier chapitre résume nos contributions et présente les perspectives pour ce travail de recherche.

# Chapitre 2 État de l'art – Placement de données et traitement efficace du Big Data

Les systèmes de gestion et de traitement de données dans un écosystème Big Data sont divers et proposent des solutions différentes pour répondre au problème de l'exécution efficace des workloads. Ce chapitre a pour objectif de présenter un état de l'art sur ces systèmes en détaillant trois piliers principaux : le modèle de représentation et de stockage des données massives, l'optimisation des traitements et l'intégration et la distribution de ces données.

Le chapitre est organisé comme suit : (i) la section 2.1 détaille les écosystèmes de gestion du Big Data, (ii) la section 2.2 présente les techniques d'évaluation de requêtes dans le domaine du Big Data, (iii) la section 2.3 résume les techniques de l'intégration et médiation des données et (iv) finalement la section 2.4 présente une discussion et la conclusion du chapitre.

## 2.1 Écosystèmes de gestion du Big Data

Dans cette section nous présentons une architecture de référence à partir de laquelle nous déclinons les différentes propriétés d'un système Big Data et nous distinguons les divers types de solutions d'interrogation distribuée des données.

Notre architecture de référence pour les systèmes de gestion des données est composée de quatre couches : (i) la couche API, (ii) la couche modèle de représentation, (iii) la couche moteur d'interrogation et de traitement de données et (iv) la couche stockage.

Dans cette sous-section, nous présentons ses différents composants et des exemples pour mettre en évidence son utilité.

Les couches *API* et *Modèle de représentation* sont celles à partir desquelles l'utilisateur interagit avec le système. L'*API* est l'interface d'accès qui nous permet de manipuler les données qui sont représentées en utilisant les concepts définis dans la couche *Modèle de représentation*. La première couche offre un ensemble de fonctions (appelés généralement des opérateurs) qui expriment une manipulation des données en respectant le modèle de représentation.

Ces opérateurs seront exécutés par un *Moteur* sur des données qui sont stockées dans la dernière couche. Cette dernière couche peut aussi permettre de gérer la distribution et la tolérance aux pannes d'une manière encapsulée pour l'utilisateur final qui n'interagit qu'avec les couches supérieures.

### 2.1.1 Traitement de données

Le terme traitement au repos en Big Data désigne les types d'applications dont le temps de réponse n'est pas critique. Le traitement peut prendre plusieurs minutes, voir des heures entières. Ils sont exécutés sous forme de lots de traitement. Ce type de traitement est une solution au problème de traitement des volumes très importants de données et d'analyse de données offline (par exemple générer des rapports, calculer des résultats intermédiaires). Les technologies qui ont été créées pour résoudre ce problème sont les datastore NoSQL, les systèmes de fichiers distribués et les systèmes de traitement distribué en mémoire (en anglais : distributed In memory processing).

#### *2.1.1.1 Analyse de données (pull)*

L'analyse de données a comme but de tirer des conclusions à partir de données collectées sur un domaine d'intérêt. C'est une étude statistique des données qui fait intervenir plusieurs techniques, comme l'exploration de données, la visualisation et la description de données, ou bien l'étude de

corrélations. L'analyse de données peut être faite par consultation des données existantes ou par extrapolation de données historiques

L'exploration de données (également appelé fouille de données et en anglais *data mining*) est un domaine qui utilise des outils mathématiques et algorithmiques pour générer des connaissances à partir de données existantes. Les méthodes d'exploration de données englobent l'apprentissage automatique et les systèmes à base de règles et de raisonnement. Dans cette partie on s'intéresse à la première catégorie d'application et on présente le principe et les applications de la classification de données, l'analyse prédictive et le groupement des données.

La visualisation et la description des données est une technique de génération de rapports et de bilans à partir des sources de données existantes. Pour visualiser les données il est nécessaire d'agréger, de transformer ou bien de filtrer les données. Ces opérations sont des requêtes complexes qui peuvent être interactives (OLAP), consommées par les applications de visualisation, ou bien *batch* qui produisent des rapports / bilans.

En Big Data les ensembles de données ont un grand nombre d'attributs. Dans ce cas, il est nécessaire de réduire la dimension de l'ensemble de données et sélectionner les attributs les plus représentatifs des données de l'ensemble de données.

- Analyse prédictive

Les algorithmes de prédiction visent à trouver une relation ($f$) entre un ensemble massif de données appelé ensemble de caractéristiques ($X$) et une valeur cible dite valeur de prédiction ($p$). L'objectif de la prédiction est d'inférer de nouvelles valeurs $p_1$ pour la variable de prédiction à partir de la fonction $f$ et d'un ensemble de caractéristiques $X_1$ qui n'ont pas été vus par l'algorithme d'apprentissage en vérifiant que $f(X_1) = p_1$.

Dans l'industrie (ou bien dans les environnements académiques) les applications types de l'analyse prédictive sont utilisées dans les stratégies marketing, dans la maintenance, dans la fidélisation des clients, dans l'analyse de fiabilité des produits, ou dans la R&D et l'innovation de production.

En marketing par exemple, les applications modernes utilisent ces algorithmes pour la prévision de la demande, des ventes et celle des stocks. La méthode de prévision dans les processus industriels (par exemple la chaîne d'approvisionnement) est importante, car elle influence l'optimisation de ces processus et le choix de leurs paramètres [Hub13]. La prévision de la demande sert à déterminer rapidement une estimation des besoins des clients dans un point éloigné dans le temps, et à répondre à ces demandes d'une manière immédiate. En particulier dans les Smart Grids les applications de prévision de demande concernant la demande en énergie électrique.

Prévoir la vente est d'une aussi grande importance que l'application qui la précède. La vente a influencé plusieurs autres secteurs comme l'approvisionnement des matières premières, la production et surtout la finance. Avoir des connaissances sur l'état de la vente face aux changements saisonniers est une fonctionnalité clé pour toute industrie qui souhaite être proactive par rapport à ses concurrents.

Quant à la prévision de l'approvisionnement et du stock, ils sont nécessaires pour la prévention de la rupture de stock, la gestion de logistique et la perte de matières premières.

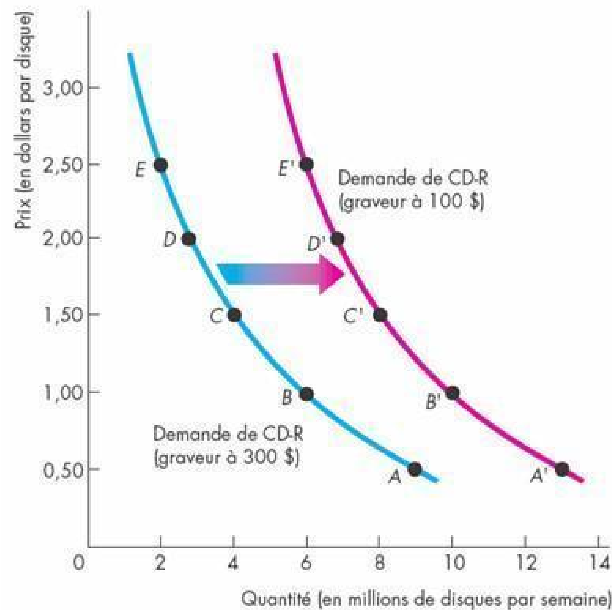



*Figure 2.1 Exemple de données d'entrée à l'algorithme de prédiction de la demande [Hal]*

La figure 2.1 montre un exemple d'application de prévision de la demande qu'on peut trouver dans l'industrie informatique. La demande est modélisée dans cet exemple par la différence de la quantité de CD-ROM vendu entre deux événements de ventes. Cette valeur dépend du prix du graveur, celui du disque lui-même et d'autres facteurs. L'objectif des applications de prévision de la demande est de prédire la relation entre ces facteurs et la demande. La prédiction est d'autant plus exacte que l'augmentation de la taille de données. D'où le besoin de technologies Big Data pour gérer et accéder à ces données.

- Interrogation de données (pull)

Tout système d'information a besoin d'applications d'accès et d'interrogation de données pour ces services de gestion ou de traitement. Dans les applications d'interrogation des données à l'opposé des applications transactionnelles, les opérations de lecture sont exécutées sur des tailles de données plus importantes et les calculs en mémoire sont plus complexes. De ce fait, ces applications ont des limites de performance dans plusieurs systèmes. Un exemple type de cette catégorie d'applications est la gestion des données des ressources d'énergie, le calcul d'agrégats pour la visualisation et la transformation des données.

L'agrégation permet de combiner des données et effectuer des calculs sur les valeurs des attributs des groupes similaires d'enregistrements dans un objectif d'analyse, de présentation ou bien de visualisation. Dans les Smart Grids, les applications d'agrégation de données sont utilisées par exemple pour produire des bilans de consommation et de production d'énergie, la multiplication de simulation ou bien pour la visualisation et la comparaison des courbes de charges suivant les catégories de consommateurs.

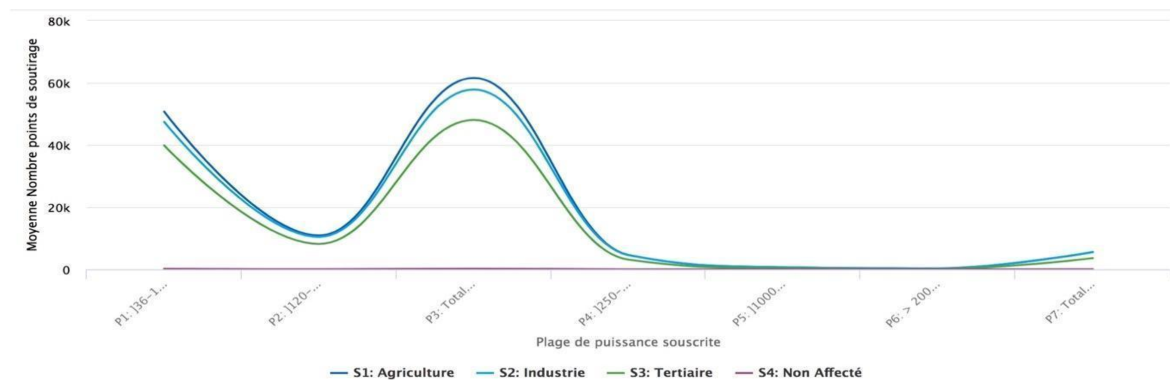



*Figure 2.2 Visualisation de données de consommation d'énergie : variation de la moyenne des nombres de point de soutirage en fonction des plages de puissance*

Le deuxième type d'application est le croisement des données. Croiser des masses très importantes de données est un vrai défi : dans plusieurs systèmes existants l'opération n'est pas faisable ou bien engendre des inconvénients de performance. Le problème de croisement de données (opération de jointure) a été étudié dans plusieurs travaux sur le modèle relationnel. Dans les systèmes distribués, ce problème est d'autant plus présent à cause du calcul distribué : la communication entre les serveurs, principalement la taille de données à envoyer sur le réseau, et dans certains modèles de traitement la matérialisation massive des résultats intermédiaires provoque des latences importantes. Toutefois, ce type d'application est très important dans le cadre des applications d'analyse de données (business intelligence) et ils sont très utilisés pour construire des vues en Étoile, des cubes OLAP, …

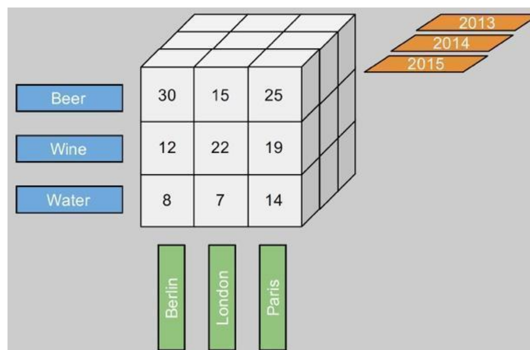



*Figure 2.3 Exemple d'un cube OLAP calculé par Apache Kylin à partir de données stockées dans HDFS [ino15]*

### 2.1.1.2 Traitement réactif (push)

Pour répondre au problème de vélocité et pour permettre aux utilisateurs de traiter les données dès leur arrivée et de manière continue, des applications de traitement réactif sur des flux de données utilisant des technologies spécialisées de traitement temps réel ont vu le jour.

Ces applications sont surtout utilisées dans les domaines de l'internet des objets (IoT), du web, des maisons/villes intelligentes et les réseaux électriques intelligents, etc. Les types de calculs effectués dans le cadre de ces applications peuvent être simples, comme des calculs d'agrégats continus, ou complexes, comme l'analyse et l'apprentissage automatique. Les résultats de ces calculs sont stockés en permanence à chaque réception du signal. Ainsi on qualifie ce type de traitement par traitement *push* puisqu'ils sont déclenchés par la réception des données. Les opérateurs de traitement sont exécutés sur des volumes très réduits de données (à l'ordre de quelques octets) mais ils sont collectés en parallèle à partir de milliers de sources de données et il existe plusieurs défis adressés par ces technologies de traitement réactifs qui sont levés par ce type d'applications [AWS].

Les défis principaux sont le traitement parallèle et continu de données, le besoin de temps minimal de latence (à l'ordre de la microseconde) et l'exécution le traitement sur une partie des données (des fenêtres temporelles ou sur les enregistrements les plus récents).

#### 2.1.1.3 Récapitulation (Applications Big Data dans les Smart Grids)

Les premières générations de systèmes Big Data avaient comme objectif de gérer plus de données sans se soucier de temps de réponse. Le souci principal était de passer à l'échelle et de garantir l'exécution de traitement sur des données plus volumineuses et plus variées. Par contre, de nos jours les utilisateurs exigent plus de réactivité et plus de performance (traitement en quasi-temps réel).

Nous avons présenté dans cette section les applications existantes dans le domaine du Big Data. Nous avons fait un focus sur les applications de gestion et de traitement de données Smart Grids. L'ingestion et la gestion des mesures des compteurs qui sont générés très fréquemment et gérées sous forme d'ensembles de données très volumineux est une classe d'applications typique de la gestion de données massives et réactive. Pour la gestion des données de description des compteurs, capteurs et généralement de l'infrastructure des Smart Grids, on a besoin de technologies Big Data. Ces dernières sont caractérisées par leur variété et leur véracité et leur gestion ainsi que leur interrogation a besoin d'opérateurs spécialisés qui ne sont pas supportés par défaut dans tous les systèmes. Les opérations de jointure et d'agrégation sont d'autant plus complexes. Ils sont aussi très présents dans les applications d'analyse de données (OLAP) des écosystèmes de données Smart Grids comme les applications de calculs des bilans pour les clients et pour les industriels. Dans les sections suivantes, nous détaillons les processus nécessaires pour mettre en place ce type d'applications et surtout les architectures adaptées pour résoudre les problèmes du Big Data.

### 2.1.2 Modèles de données

Dans cette section, nous détaillons les différents modèles de données en Big Data. Le modèle de données est un ensemble de concepts tels que : entités, attributs, relations, documents, pour définir d'une manière formelle les structures des données. Ces concepts sont utilisés pour la couche *Modèle de représentation* et pour la couche de stockage qui peut utiliser un modèle de donnée spécifique. Un système de gestion de données supporte généralement un modèle de données et gère des schémas – les descriptions des structures (d'ensembles) de données. Plusieurs modèles de données ont été proposés.

Par contre, dans un cadre généralisé, Dans un système de gestion de données, le modèle de représentation possède un modèle de données qui peut être différent de celui ou ceux du stockage physique et qui est géré totalement par le système lui-même. Les détails du modèle de données de cette dernière couche est masqué à l'utilisateur.

Dans quelques solutions NoSQL, le modèle de représentation est absent. L'utilisateur manipule directement les données selon leurs modèles de stockage à travers des API d'accès de données. Il n'y a généralement pas de langage déclaratif pour manipuler les données de la couche de stockage d'où l'absence de processus d'optimisation.

Pour répondre à la problématique de la variété en Big Data, plusieurs types de modèles de données ont été inventés. Chaque type varie en termes de complexité et de performance.

Dans ce qui suit, nous comparons les différents modèles de données en Big Data et nous présentons pour chaque modèle les avantages et les inconvénients. Dans les exemples, on utilise des données de consommation d'électricité dans les Smart grids.

| Data model | Principle | Advantage | Disadvantage | Example |
|---|---|---|---|---|
| Key-value | Data stored and accessed as a key-value pair: Key a unique identifier and Complex data as value | ✔ Efficient index on data<br>✔ Simple data model | Limited access API | key / value: Reference_Profil: PRO2 (+ PRO2WE + PRO6); Reference_Horodate: 2018-11-30T06:00:00+01:00; Reference_Indice_représentativité_Courbe1(%): 1; Reference_Nb_points_soutirage: 2; Reference_Courbe_Moyenne1_2(Wh): 1121 |
| Document oriented | Aggregates stored as documents | ✔ Complex queries<br>✔ Rich filters | Slow writes, lack of consistency | |
| Column family oriented | Data organised as columns. Columns store aggregates and are organised as column families | ✔ Evolving data schema<br>✔ Efficient column oriented storage | Slow writes, bad performance with small data and row based access | |
| Graph oriented | Representing the connexions between data records | ✔ Graph algorithms for efficient connected data querying | Does not support well distribution | |

*Tableau 2.1 Avantages et inconvénients des modèles de données en big data*

Le tableau 2.1 représente une comparaison des modèles de données les plus populaires en soulignant leurs avantages et leurs inconvénients.

Le modèle de données **clé-valeur** qui est le modèle de données le plus simple est composé par deux attributs seulement : la *clé* qui est généralement une clé hachée et la valeur. Dans ce modèle, la valeur peut être de type textuel, structuré, multivalué (tableau) ou atomique. Elle est traitée sous forme opaque et les seules opérations autorisées sont exécutées sur la clé (Modèle d'interrogation basé sur la clé).

La manipulation des données dans le modèle clé valeur est faite directement sur le modèle de stockage des données comme le montre l'exemple du tableau 2.1. Cette architecture facilite considérablement la lecture et l'écriture des données dans les environnements distribués vu que le système maintient un index efficace sur la clé. Mais elle rend le développement d'applications complexe vu les limites d'opérateurs supportés par le système. Ce modèle de données a donc une API qui ne supporte pas toutefois l'accès par plage de valeur et l'accès par correspondance exacte. Un autre modèle d'interrogation supporté par les modèles clés valeurs est le modèle Map-Reduce [DG08].

Il est rare de trouver des systèmes ayant des modèles de données clés-valeurs pures. Il existe plusieurs systèmes qui ont été conçus par référence à ce modèle et qui ont des APIs d'accès plus développées. D'autres exemples de systèmes qui ont le modèle clé valeur sont : Riak kv [ZS17] et Dynamo [Siv12, KSMH17].

Le modèle de données **orienté document** sert à stocker des agrégats et de les interroger en appliquant des filtres complexes. Il associe à chaque document (un agrégat) une clé. L'agrégat est composé d'attributs qui à leurs tours sont composés d'un nom d'attribut et d'une valeur. Cette valeur peut être de type simple ou peut être à son tour un agrégat.

Dans l'exemple du tableau 2.1 : l'attribut *cd_prof* est un exemple de valeur de type structuré. Dans ce modèle de données, ces agrégats sont appelés aussi des sous-documents ou des documents imbriqués. Il est possible d'accéder aux valeurs de ce type d'attributs en utilisant un opérateur spécifique par

exemple l'opérateur « *MapValue* » *[*https://calcite.Apache.org/apidocs/org/Apache/calcite/sql/fun/SqlMapValueConstructor.html*]* de l'API du système Apache Calcite ou l'opérateur « *[]* » de rethinkdb. En effet, contrairement au modèle clé-valeur où l'API d'accès à une vue opaque sur la valeur, la structure de la valeur (un document) est connue ; ainsi les agrégats dans ce modèle sont facilement manipulables en utilisant des opérateurs de filtrage, de sélection par plage de valeurs et de correspondance exacte. Cette organisation logique sous forme d'attributs ayant des valeurs structurées qualifie ce modèle de données de modèle hiérarchique.

Le modèle de données **orienté familles de colonnes** est très similaire au modèle relationnel : Les attributs sont représentés sous forme de colonnes. De plus, ce modèle permet de grouper les attributs en familles de colonnes et avoir pour chaque attribut une valeur non structurée (par exemple texte, images jpeg, gif, …) ou ayant un type complexe. Les familles de colonnes sont prédéfinies à la création du schéma des données, mais les colonnes sont quant à elles ajoutées au fur et à mesure de l'exécution de requêtes de mise à jour.

Le tableau 2.1 donne un exemple d'ensemble de données organisé sous forme de familles de colonnes. L'exemple est composé de deux familles de colonnes : *Description de consommation* et *mesures*. Chacune de ces familles de colonnes contient des colonnes groupées dans ce cas de figure selon une logique métier.

L'accès aux données est fait une colonne à la fois, il est optimisé pour des opérations sur une partie d'un ensemble de données ayant une grande dimension. La complexité de projection sur $s$ attributs est à l'ordre de $s * n << t * n = N$. avec $t$ le nombre total des attributs à sélectionner, $n$ le nombre d'enregistrements total et $N$ la taille totale des données.

Les exemples de systèmes qui supportent le modèle de données orienté familles de colonnes sont : Google Bigtable [CDG+08], Apache Hbase [Voh16], Amazon Simple DB [CH10].

L'analyse et l'interrogation des liens entre des données massives est un défi qui n'a pas été adressé dans plusieurs modèles de données tels que le modèle clé valeur, le modèle orienté document et le modèle orienté familles de colonnes. Ces derniers ont omis de leurs API les opérateurs de jointure et d'interrogation de graphes pour passer à l'échelle et préserver l'efficacité des traitements. Ainsi, pour interroger des liens entre des ensembles de données dans le domaine Big Data, la solution qui a été proposée est de créer un modèle de données (modèle de données orienté graphes), qui soit capable de matérialiser ces liens entre entités et de les interroger en utilisant des algorithmes de parcours de chemins, algorithmes qui ont prouvé leur efficacité pour des données massives par rapport aux solutions de calcul des liens entre données au moment de l'exécution. Ce modèle de données représente donc les données sous forme de graphes. Il supporte le stockage et l'interrogation d'ensembles de données ayant une infinité de liens. Les applications usuelles de ce type de modèle de données sont la capture de données composées de liens complexes, tels que les réseaux sociaux, les préférences de produit, l'analyse de routes et de réseaux géospatiaux, la recommandation basée sur la similarité de profils, l'interrogation de données relationnelles ou des graphes de connaissance, comme les ontologies [ME16] à très grande échelle. Il est possible d'améliorer l'efficacité du traitement en transformant les croisements complexes de données en parcours de graphes.

Il existe deux types de modèles de données orientés graphes : (i) le premier modèle concerne les graphs d'attributs qui sont représentés sous forme d'agrégats (des nœuds avec des propriétés) reliés entre eux en utilisant des liens. Les exemples de systèmes qui utilisent ce type de modèle de données qu'on a étudié sont : Neo4j [LC15] et Spark Graphx [ME16]. (ii) Quant au 2ème, il donne des concepts pour représenter les données sous forme de graphes de triplets <sujet, prédicat, objet> (par exemple,

le modèle RDF [CSPG16], qui est supporté par les systèmes Virtuoso [EM10], Allegrograph [FB18] …). Il n'existe pas de séparation entre vue logique et physique dans ce modèle, les données sont stockées et interrogées sous forme de triplets. Il faut noter que cette étude se limite à la première sous-catégorie de modèle de données orienté graphe.

On propose une autre comparaison entre les modèles de données. Le tableau 2.2 met en évidence la différence des *APIs* entre les modèles de données. Nous présentons pour chaque modèle les langages d'interrogation et les APIs d'accès supportés. Dans notre étude on s'intéresse à la comparaison de la capacité de l'exécution des différents opérateurs pour chaque système d'où l'intérêt de ce tableau.

| **Operator / data model** | **Key value (e.g Amazon DynamoDB)** | **Document based (e.g MongoDB)** | **Column family (e.g Apache Hbase)** | **Graph based (e.g Neo4J)** |
|---|---|---|---|---|
| Scan | XXX | X | XX | X |
| Filter | X | XX | X | XX |
| Join | - | X | - | XXX |
| Aggregate | - | XXX | X | XX |
| Group by | - | XXX | X | XX |
| Order by | - | X | X | X |

*Tableau 2.2 Comparaison des APIs des modèles de données en Big Data*

Le tableau 2.2 nous permet de comparer les différents modèles de données en indiquant la performance de l'exécution des opérateurs d'interrogation de données de base.

Pour manipuler des données dans leur format brut et sans exécuter un prétraitement pour les raffiner et les restructurer, des modèles de données basés sur les agrégats ont été inventés. Ces modèles de données permettent le stockage et la manipulation de valeurs structurées et les APIs des systèmes facilitent l'accès aux valeurs composites. Il existe trois types de modèles de données basés sur l'agrégat : le modèle clé-valeur, le modèle orienté document (appelé aussi clé-document) et le modèle orienté famille de colonnes. En effet, cette catégorie de modèles de données supporte les valeurs de type structuré [SF12] (par exemple, les tables hachées, en anglais *hashmap*) et les valeurs multivaluées, comme les tableaux. L'opération de jointure n'est souvent pas supportée par le moteur d'exécution (comme le montre le tableau 2.2 pour le modèle clé-valeur) ou bien est très coûteuse (cas des modèles orientés documents ou familles de colonnes). Ainsi, Les associations (*one-to-one*, *one-to-many* et *many-to-many*) sont généralement transformées en une table avec des attributs de type structurés. Dans ces modèles les schémas physiques et logique sont souvent confondus. L'API interroge les données selon la manière dont elles sont stockées.

En effet, dans le modèle clé valeur, l'application doit connaitre la clé pour accéder à la valeur. Les opérateurs de filtrage et de projection sur la valeur ne sont pas supportés, car cette dernière ne peut être interprétée par le système. L'opérateur « Filter » n'est possible que pour la clé et les opérations de jointure et de l'agrégation ne sont pas possibles. Néanmoins, le parcours des données (opérateur scan) est très efficace d'où l'intérêt d'utiliser ce modèle pour accéder à des données massives dans un environnement distribué en assurant la performance.

Par exemple dans le data store Voldemort [NB15] opérations que l'API offre sont : *get(key)*, *put(key, value)* et *delete(key)*. Ce modèle d'interrogation garantit l'efficacité du traitement et assure la prédictibilité de la performance dans les environnements distribués, mais une grande partie des traitements (interprétation des valeurs) doit être faite directement par les applications.

Le modèle de données orienté documents permet à l'utilisateur de lancer des requêtes complexes sur les valeurs. L'accès aux données se fait donc en connaissant les attributs qu'on souhaite manipuler ou en précisant les motifs selon lesquels le résultat est obtenu. Un motif peut être une correspondance exacte ou partielle. Il est optimisé pour les opérations de filtrage (par valeur ou bien par plage de valeurs) et le groupement des enregistrements par attributs pour calculer des agrégats. Un autre type de requête avec ce modèle d'interrogation est l'accès aux sous documents des valeurs structurées et l'accès aux éléments des valeurs de type tableau. Ce modèle permet d'indexer les attributs non-clés de l'ensemble de données et ainsi permettre des opérateurs de filtrage et de projection qui n'était pas possibles dans le modèle de données orienté clé-valeur

Le modèle d'interrogation est spécifique aux modèles de données orientés graphes est appelé : modèle d'interrogation basé sur la connexion. Les opérateurs de ce modèle n'interrogent pas seulement les enregistrements de données, mais aussi les liens qui sont matérialisés dans le graphe. Ces opérateurs sont spécifiques au modèle de données orienté graphes et ils ne sont pas définis dans d'autres modèles de données, comme le modèle relationnel. L'accès aux données se fait par traversée de graphes et en connaissant les liens qui existent entre les enregistrements. À partir d'une référence du nœud de départ, le moteur d'exécution parcourt le fichier des liens à la recherche des nœuds liés qui répondent à la requête. Plusieurs algorithmes ont été proposés pour développer les opérateurs de ce modèle. Ils sont basés sur la théorie des graphes et ils servent à résoudre des problèmes comme : la recherche dans un graph [VVDG+11], [CB1] la recherche de chemins [HSL16] [CB3], particulièrement du plus court chemin [KMJ13] [CB5] et la recherche des voisins les plus proches [SDG09] [CB7].

### 2.1.3 Conclusion

Dans cette section, nous avons proposé une architecture de référence pour la description des systèmes en Big Data. Pour mieux mettre en évidence l'importance de cette architecture, on reprend l'exemple de l'interrogation de données stockées dans HDFS [Bea15] (acronyme pour *hadoop distributed file system*). La fonctionnalité offerte par HDFS est le stockage des données de manière répliquée afin d'assurer la tolérance aux pannes. Ces données sont stockées dans un format binaire. L'utilisateur a besoin d'utiliser d'autres outils pour accéder à ces derniers. Un système de la couche moteur d'exécution comme le Framework MapReduce [DG04] permet aux utilisateurs d'écrire des programmes de traitement des données stockées dans HDFS d'une manière distribuée et parallèle. HDFS n'est par contre pas capable de décoder la structure des données telles qu'elles sont stockées. Il n'a pas d'informations sur le modèle de représentation. Il a donc besoin d'un système de sérialisation qui transforme un bloc de données binaires en un ensemble de clés-valeurs que le moteur d'exécution peut traiter.

La deuxième partie de cette section concerne les modèles de données. Généralement, les modèles de données émergents ont été conçus pour couvrir les limites pour le passage à l'échelle du modèle relationnel en offrant des alternatives à la façon dont les données sont représentées physiquement ou logiquement. Toutefois, il existe d'autres modèles de données moins populaires répondant à de besoins très spécifiques que nous n'avons pas couvert. Par exemple, il existe des modèles de données basés sur les fichiers [IT15], qui manipulent des ensembles de données sous forme de fichiers. Le modèle de données objet offre des constructeurs et des opérateurs de manipulation pour des objets tels qu'ils sont définis dans le domaine de la programmation orienté objet. Le modèle de données des séries temporelles permet de manipuler et gérer des valeurs indexées par des horodatages (timestamps).

## 2.2 Stockage distribué

La deuxième étape du cycle de vie correspond au stockage des données. C'est un processus qui correspond à la dernière couche de notre architecture de référence. Dans cette section, on décrit les différentes solutions de stockages dans les data stores et les systèmes de fichiers distribués dans un écosystème Big Data et les techniques de distributions associées à ces systèmes.

La technique de stockage des données sans transformations complexes et sans conception pointue d'un schéma a vu le jour avec l'avènement du domaine du Big Data. Pour faciliter la manipulation des données variées, ce modèle permet le stockage de données semi- ou non-structurées directement sans se soucier du schéma et limiter l'utilisation d'un schéma unique pour représenter les données.

Le stockage en Big Data peut être distribué entre plusieurs systèmes. Ainsi au lieu d'utiliser une seule solution de stockage, il est possible de répartir les données entre plusieurs systèmes pour des raisons de performance ou bien selon des règles métier et utiliser une solution d'intégration de données massives pour interroger les données. Dans cette section, nous commençons par expliquer le cas de stockage le plus simple : dans un seul système de stockage, ensuite nous présentons les différentes techniques d'intégration de données et de stockage dans ce qu'on appelle des polystores.

### 2.2.1 Datastore

On appelle data store, dans le domaine Big Data, tout système de gestion de bases de données (SGBD), système de fichiers distribué (par exemple HDFS [Bea15]) ou bien système NoSQL. Le stockage dans des solutions NoSQL a été proposé comme solution alternative aux RDBMS pour pallier les limites de ces derniers au niveau du passage à l'échelle et à la performance liée à la complexité de ces API. Ces solutions offrent un large choix de modèle de données (cf. modèles de données et manipulations), une grande flexibilité dans la gestion des transactions et des structures et des fonctionnalités pour le traitement de données qui ne sont généralement pas supportées dans les SGBD relationnels. Les systèmes de fichier distribués sont dédiés au stockage de données massives, généralement dans leurs formats bruts. HDFS, un exemple de ces systèmes, isolé n'offre pas de langage de requête et de gestion de données supporté par défaut. Il offre par contre des plugins pour plusieurs autres systèmes de l'environnement Big Data comme SparkSQL [AGZ+15], Hive [Gup17], HBase… Il permet aussi de développer une API d'accès et de manipulation de données en utilisant le framework Map/Reduce [Bea15]. Son modèle de données est de type fichier, ce qui lui offre une élasticité pour supporter toute structure de données, mais avec une limite au niveau de la facilité de manipulation. Ce système n'impose non plus les mécanismes d'administration du stockage de données et de distribution, et ne permet pas l'accès à ces fonctionnalités d'une manière intuitive et user friendly.

Dans cette section, on étudie les solutions de stockage physique, dans cet objectif, on détaille les solutions de sérialisation sur le disque et les formats de compression en se focalisant sur les systèmes de fichier distribués, car ils offrent plus de choix et de liberté à ce niveau. Le schéma physique a été expliqué dans la section "modèles de données et manipulations".

### 2.2.2 Modèles de stockage

Il existe trois types de solution de stockage en Big Data : le stockage des données sous formes de fichiers, le stockage des données sous format binaire et le stockage des objets massifs binaires. Le stockage binaire peut être très spécifique d'un système à un autre. Cette propriété peut présenter une limite sur la compatibilité entre les différents systèmes au niveau échange et transfert de données [SDRL16]. Quant au stockage sous forme de fichiers, il est utilisé par les applications qui intègrent les systèmes de fichiers distribués (par exemple HDFS). Le traitement des données généré sous forme de fichiers brut peut être un défi. Dans ce travail on ne s'intéresse pas au troisième type de stockage qui concerne surtout le stockage des données non structurées telles que des images.

#### 2.2.2.1 Stockage des données dans les systèmes de fichier distribués

Les deux techniques utilisées pour améliorer le stockage dans ces types de systèmes sont la compression et la sérialisation.

La compression sert à réduire l'espace occupé par les données et permet ainsi de diminuer la taille de stockage sur disque et d'allocation en mémoire ainsi que la taille des données échangées entre les serveurs, et de réduire les coûts de lecture et d'écriture (IO) [MPB17]. Diminuer la taille des données a un impact direct sur le temps de transfert et des opérations IO, d'où l'intérêt de la compression pour l'amélioration de la performance des Workloads Big Data. Dans HDFS, pour la compression des données, il est recommandé d'utiliser le format le plus compact pour l'archivage de données et les formats de compression divisibles (*splittable*) pour la compression des données à utiliser dans les programmes Map-Reduce. Dans Apache Hbase, une des premières configurations à considérer lors de l'amélioration de l'efficacité du traitement de données massives est la compression des blocs sur disque [Tea16], vu qu'il supporte plusieurs formats de compression différents tels que Gzip (l'algorithme Deflate) [SS19], LZO [KY12] ou Snappy [Che18], ou bien des formats de compression fractionnables comme Lz4 et LZO indexé

Le choix du format de sérialisation nécessite aussi une connaissance sur sa compatibilité avec le type de fichiers manipulés ainsi que le type de Workload. Dans le cadre des applications de traitement et de gestion de données stockées dans des systèmes de fichier distribués, et plus spécifiquement dans l'écosystème Hadoop, il existe un vaste choix de solutions pour la sérialisation des ensembles de données. On peut en fait utiliser la librairie *SequenceFiles [ESBES19]* pour le stockage des fichiers de type général avec une taille du block supérieure à 64 MO, surtout pour les fichiers binaires et massifs comme les images. Le Framework Map-Reduce utilise aussi comme format de sérialisation pour les données qu'il traite plusieurs librairies, comme la librairie *SequenceFiles* qui a été créée exclusivement pour cet objectif. Un fichier de type *SequenceFiles* a un format binaire et il stocke les données sous forme de séquence de couples <clé, valeur>. La figure 2.4 montre une représentation schématique de la disposition de données dans un fichier de ce type. La compression peut être appliquée sur la totalité de l'enregistrement selon une configuration intitulée : compression par blocs, ou sur la valeur uniquement pour la compression par enregistrement.

*Figure 2.4 Disposition des données dans un fichier SequenceFile [GZH+19]*

Quant aux fichiers textes structurés, par exemple XML ou bien JSON, leur traitement est difficile, mais il existe des bibliothèques spécialisées intégrées à quelques systèmes de manipulation (exemple la sérialisation de XML dans Apache PIG [SA17] est possible avec le format *XMLLoader* de la librairie *PiggyBank* et celle de *JSON* dans la librairie Apache PIG intitulé *elephant-bird* avec le format *LzoJsonInputFormat*). Pour stocker des données de volume important et de format tabulaire, il est possible d'utiliser un format de stockage structuré. Les deux formats les plus utilisés sont le format de fichier *Avro [Voh16a]* qui utilise la disposition orientée enregistrement et le format de fichier *Parquet [Voh16c]* dont la disposition de données est orientée colonnes.

Apache Avro est un système de sérialisation de données qui permet aux moteurs de traitement qui le supportent de stocker et gérer les données sous format structuré, orienté enregistrement. La caractéristique qui le différencie de ses concurrents est la fonctionnalité d'évolution des données. Il permet aussi de stocker des enregistrements qui ont des attributs de type structuré. Quant à Apache Parquet, c'est une librairie qui permet de sérialiser des ensembles de données structurées et qui ont une disposition de données (en anglais *data layout*) orientée colonne qui favorisent les workload de

lecture des données, vu que le moteur traite moins de données que dans le cas du stockage orienté enregistrement.

#### 2.2.2.2 Stockage des données dans les magasins de données

Le stockage de données dans ce type de systèmes est plus varié que les modèles de données de la représentation logique.

Le stockage physique des enregistrements dans les magasins de données ayant un modèle de données orienté document est fait horizontalement, sous forme d'une séquence contiguë d'enregistrements. Ce type de représentation physique des données est appelée disposition de données orienté enregistrement (en anglais *row oriented data layout*) et l'accès à l'enregistrement à la position *p* se fait après la lecture et le chargement dans le processeur de *p* enregistrements entiers. Cet accès n'est pas optimisé pour les opérations et la projection d'un sous-ensemble d'attributs. Il n'est pas donc le meilleur choix pour stocker des ensembles de données éparses ou ayant un nombre de dimensions élevé. MongoDB [Aut21], un système de stockage dont le modèle de données est orienté document, utilise la structure de données en listes chaînées pour le stockage physique et la structure de données Btree [GB10] pour l'indexation. La figure suivante présente un schéma du modèle physique des données dans ce système ; la complexité d'accès aux données sans indexes est *O(n)*, par contre la complexité d'un accès sur des données indexées est celui d'une recherche dans un arbre binaire balancée et il est égale à $O(\log_2(n))$.

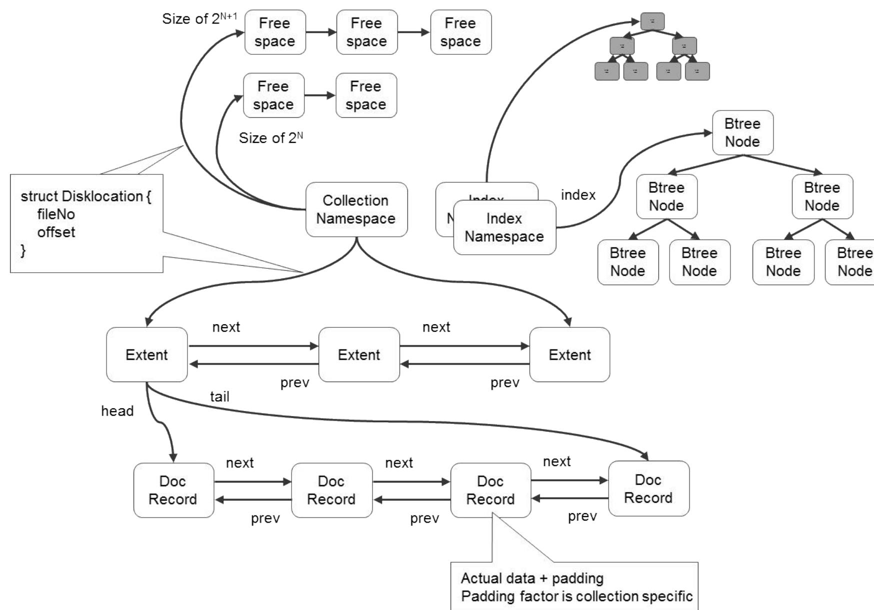



*Figure 2.5 Stockage des données dans MongoDB [Li14]*

CouchDB [Len09], une autre solution NoSQL de modèle de données orienté document permet aux applications d'utiliser des index géospatiaux grâce à leurs index de structure R-tree [QTCZ18].

Dans les magasins de données NoSQL orientés familles de colonnes la représentation des données au niveau physique le stockage est organisé par colonnes. Les familles de colonnes qui regroupent une ou plusieurs colonnes représentent l'entité unitaire de distribution de données. Cette optimisation du stockage permet d'améliorer la performance des opérations de projection et d'agrégation faites sur un sous-ensemble d'attributs pour les ensembles de données massifs.

Au niveau physique du modèle de données orienté graphes d'attributs, il existe plusieurs solutions pour stocker les données. Une solution largement adoptée par les moteurs de traitement de requête

sur des graphes consiste à stocker séparément les nœuds et les liens dans des supports de stockage physiques (en anglais *data store*) différents. Par exemple dans le moteur de traitement Spark GraphX [ME16] la création du graphe se fait généralement à partir de deux fichiers plats stockés sur un système de fichier distribué tel que HDFS (un premier pour les nœuds et leurs attributs et un deuxième pour les liens et leurs attributs) comme le montre la figure (Figure 2.6). La première étape d'une application de traitement de graphes consiste donc à lire ces fichiers dans le modèle de données de Spark. Ensuite, en utilisant l'API spécifique au traitement des graphes, l'application devient capable d'interroger les données en utilisant des algorithmes spécifiques.

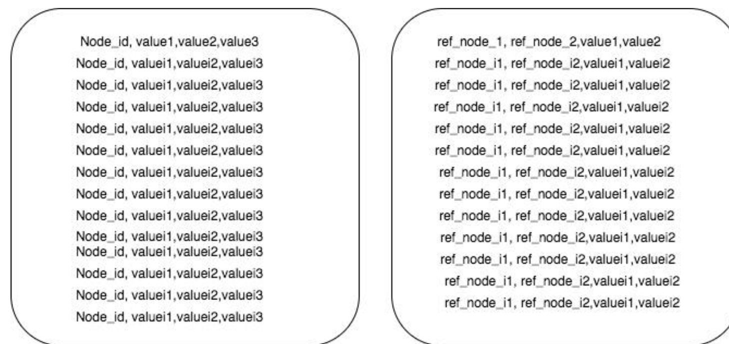



*Figure 2.6 Stockage physique d'un graph interrogé par la librairie Graphx d'Apache Spark*

Quant au système NoSQL orienté graphe Neo4J [Mil13], il stocke séparément les propriétés, les nœuds et les liens, chacun dans un fichier de stockage ayant une structure de liste chaînée (comme le montre la Figure 2.6).

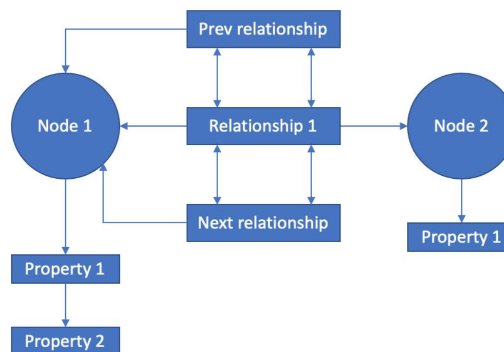



*Figure 2.7 Stockage physique d'un graphe de données dans Neo4j*

Le dernier exemple de stockage physique que nous présentons concerne le stockage dans les bases de données relationnelles. La représentation des données dans la couche stockage de ces derniers n'est pas standardisée. Plusieurs alternatives se présentent selon le système et elles offrent des performances variées. Les structures de données utilisées dans les systèmes existants pour ce modèle sont multiples et peuvent être par exemple : un *heap file* [AG15] ou un arbre binaire trié (B-tree).

### 2.2.3 Distribution des données

Dans cette section, nous présentons les techniques de distribution de données dans les systèmes Big Data. Nous distinguons deux types de distributions : la distribution des données entre différents systèmes de gestion de données autonomes et la distribution interne des données effectuée dans un seul système en utilisant des techniques de partitionnement, de réplication et de parallélisme.

### 2.2.3.1 Architectures de distribution de données

En reprenant notre architecture de référence, on peut décrire différents modèles de distribution de données pour les systèmes d'intégration ainsi que pour les systèmes monolithiques.

Les systèmes d'intégrations permettent d'interroger des données gérées dans multiples bases de données ou magasin de données différents, généralement hétérogènes et autonomes en utilisant une interface uniforme et en masquant à l'utilisateur la complexité de l'architecture. Plusieurs types de systèmes ont été proposés pour permettre l'interrogation de multiples sources de données : (i) les systèmes d'intégration proposent des fonctionnalités de transformation de données et de conversion de schémas afin de rendre compatibles la couche « moteur d'exécution » et la couche « modèle de représentation ». (ii) Les systèmes de médiation offrent des fonctionnalités qui facilitent le passage de la couche API et celle du modèle de représentation. Et finalement la fédération permet d'accéder à plusieurs systèmes à travers une communication l'application et plusieurs API locales.

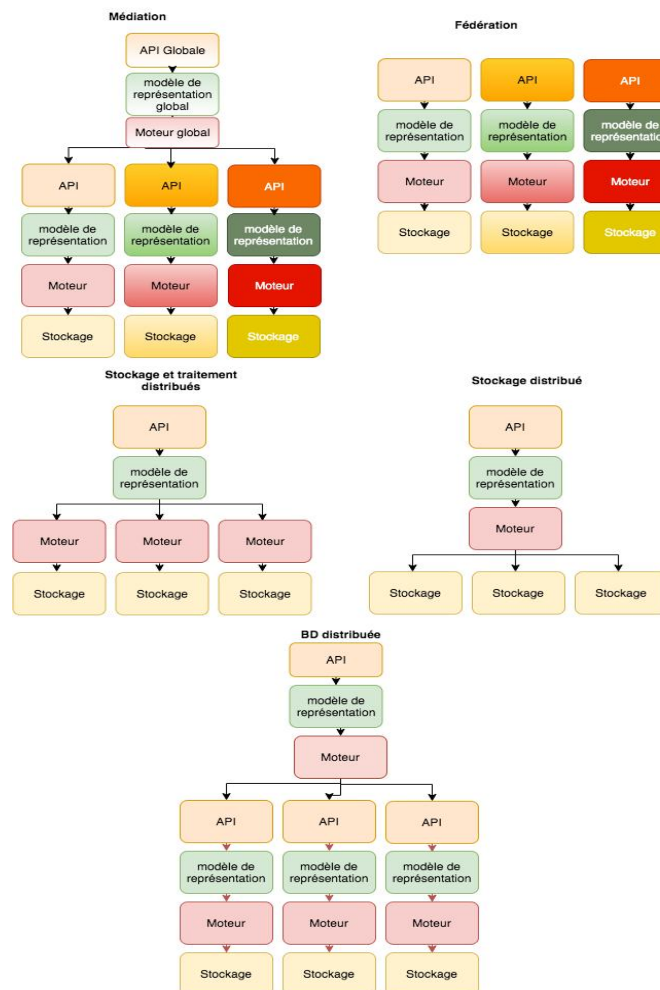



*Figure 2.8 Comparaison des architectures de systèmes d'interrogation de données distribués en utilisant un modèle de référence*

La figure 2.8 distingue les différentes solutions d'interrogation de données en respectant notre architecture de référence. La différenciation des couleurs dans figure reflète la possibilité que les couches (API, modèles de représentation, Moteurs et Stockage) puissent être différentes et qu'il faut gérer la compatibilité entre deux couches successives.

La première architecture concerne les systèmes de médiation. Ces systèmes permettent d'interroger des sources hétérogènes en offrant une interface entre ces derniers et l'utilisateur. L'API globale

contient un schéma global qu'il faut adapter pour pouvoir interroger les schémas locaux des sources. Il existe deux techniques utilisées dans ce domaine pour résoudre ce problème : (i) la première *global-as-a-view* (GAV), dans laquelle le schéma global est l'intégration des différents schémas locaux, nécessite l'adaptation du schéma global à l'ajout de nouvelles sources de données, mais facilité l'interrogation, et (ii) *local-as-a-view* (LAV), où les schémas locaux sont des sous-parties du schéma global, permet d'ajouter facilement des sources de données qui suivent le schéma global défini dans la couche API.

La deuxième architecture correspond aux systèmes de fédération. Elle permet aux applications de gérer des systèmes hétérogènes en utilisant leurs propres APIs (hétérogènes). Elle délègue entièrement à l'utilisateur /l'application la gestion de la distribution et l'orchestration de l'interrogation de multiples sources hétérogènes de données.

Quant aux trois dernières architectures, elles concernent : (i) des systèmes dont les différentes couches sont homogènes comme les bases de données relationnelles distribuées. Des systèmes (ii) dont l'architecture possèdent une solution de stockage distribué, comme dans le cas de l'interrogation de données stockées dans HDFS en utilisant un moteur de traitement centralisé, possèdent une architecture homogène uniquement pour le stockage et un seul moteur. Et (iii) des systèmes qui ont une architecture basée sur le stockage et le traitement distribué (plusieurs moteurs associés à des supports de stockage) mais qui restent homogènes, comme dans le cas de l'interrogation de données stockées dans HDFS en utilisant le système d'interrogation de données Apache Spark.

#### 2.2.3.2 Distribution des données dans les systèmes monolithiques

Pour assurer le passage à l'échelle et la performance du traitement deux solutions ont été proposées : (i) l'évolution verticale qui consiste à améliorer le matériel du serveur où les données sont stockées en ajoutant de la puissance de calcul (processeurs) et de l'espace de stockage, et (ii) l'évolution horizontale qui permet de traiter des masses plus importantes de données en ajoutant à l'architecture des serveurs supplémentaires moins coûteux. La première solution a été adaptée dans les premières générations de bases de données, elle permet une exécution centralisée des données ce qui est avantageux pour éviter les problèmes de communication réseau. Mais elle présente plusieurs limites comme le coût très élevé du matériel et l'incapacité de supporter le volume très important qui caractérise les données des applications modernes. Il est donc nécessaire d'utiliser la deuxième solution et de répartir les données et les traitements entre les serveurs disponibles dans le réseau. Ce processus est appelé la distribution de données et il existe plusieurs architectures utilisées en Big Data pour cet objectif : l'architecture Maître/Esclave, l'architecture pair-à-pair et l'architecture basée sur des clusters [Shr17], et l'architecture de traitement massivement parallèle (MPP). Dans l'architecture Maître/Esclave (en anglais Master/Slave), on distingue deux rôles pour les serveurs du système distribué : le maître et l'esclave. Le maître a le rôle d'orchestrer les traitements et les communications entre esclaves et l'esclave a le rôle d'exécuter les tâches dédiées. Dans le contexte des bases des données, cette architecture est surtout utilisée dans la réplication des données. Dans ce cadre d'application, le serveur maître exécute les mises à jour et l'écriture dans la base, et les esclaves sont limités à la lecture des données. Les techniques de réplication qui utilisent cette architecture sont divisibles en deux groupes : (i) la réplication de données maître unique (en anglais *single-master*) et (ii) réplication de données maîtres multiples (en anglais *multi-master*). La première catégorie est l'architecture traditionnelle qu'on a expliquée ci-dessus. Concernant la deuxième solution, elle permet de désigner plusieurs serveurs maitres qui vont se comporter comme pairs entre eux. Cette architecture autorise l'écriture sur plusieurs serveurs (les serveurs maîtres) et nécessite une étape de synchronisation entre ces derniers pour propager les modifications dans tous les membres de l'architecture en garantissant la cohérence des données (comme le montre la figure 2.9).

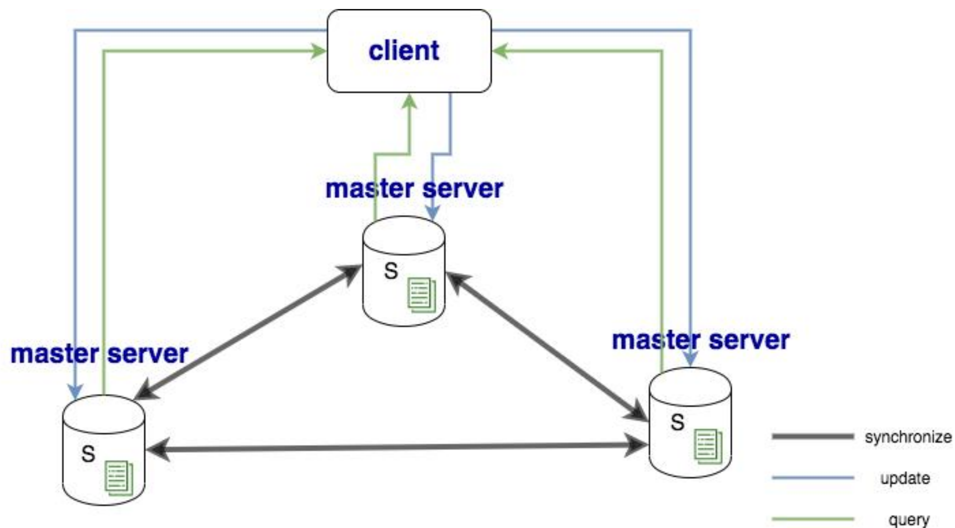



*Figure 2.9 Réplication à maîtres multiples*

L'architecture pair-à-pair est une architecture décentralisée qui permet d'organiser les serveurs dans un réseau distribué sous forme d'agents qui ne détiennent aucun rôle particulier et qui contrôlent d'une manière égale la communication entre eux ainsi que l'exécution des traitements [YHKS08].

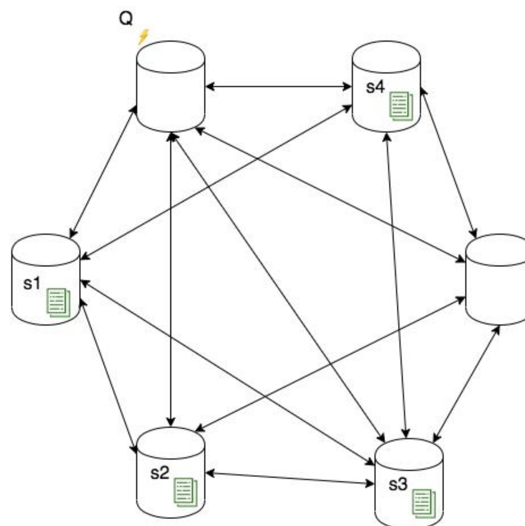



*Figure 2.10* Architecture de distribution de données paire à paire

Comme le montre la figure 2.10, à la réception d'une requête le serveur de départ communique cette dernière à tous les serveurs du réseau à la recherche des fragments $s_i$ de données qui, une fois reconstituées, forment l'ensemble de données *S* qui a été distribué. Ensuite dans chaque serveur, la requête est exécutée localement pour produire une partie du résultat. Finalement, le résultat est restitué à partir des fragments $s_i$ et puis envoyé au service qui a émis la requête.

L'architecture pair-à-pair est aussi utilisée dans les solutions de réplication. Pour résumer et pour illustrer ces principes en termes d'exemples, nous proposons de comparer les techniques de réplication dans les solutions « Big » et « Small data » dans le tableau 2.3.

| Système de données | Distribution de données | Architectures de réplication | Type d'architecture | Stratégie de placement des réplicats | Unité de réplication | Architecture en grappes de données |
|---|---|---|---|---|---|---|
| HDFS (framework Hadoop) | Vrai | « master / slave » | « multi-master » , « single master » | « hdfs replica placement policy » | Block | Vrai |
| Hive (framework Hadoop) | Vrai | « master / slave » | « multi-master » , « single master » | « hdfs replica placement policy » | Block | Vrai |
| Hbase | Vrai | « master / slave » | « multi-master » , « single master » | « hdfs replica placement policy » | Block | Vrai |
| MongoDB | Vrai | « master / slave » | « single master » | « asynchronous replication » | Document | Vrai |
| Cassandra | Vrai | « peer to peer» | « consistent hashing» | « simple strategy » , « network topology strategy » | Sstable | Vrai |
| DynamoDB | Vrai | « peer to peer» | « consistent hashing» [AWSb] | « geographic replication » | Table | Vrai |
| Neo4J | Faux | « master/ slave» | « single master » [Neo] | « causal stratery using raft protocol » | Graph | Vrai |

*Tableau 2.3* Tableau comparatif des solutions de réplication

La dernière architecture pour la distribution de données que nous présentons est l'architecture basée sur les grappes de serveurs (en anglais clusters). C'est un type d'architecture pour l'organisation physique des serveurs dans un réseau et qui est entrée en vigueur avec le domaine Big Data. Une grappe de calcul [Cas] est un ensemble de serveurs géographiquement proches (par exemple, ils appartiennent au même réseau local) et qui partagent des ressources de stockage, par exemple une baie de disques durs. Dans cette architecture les groupes de serveurs sont intitulés des centres de données (plus connus en anglais comme datacenter) et les serveurs sont appelés nœuds, ils stockent généralement une partie des données. La proximité des nœuds dans un centre de données fait en sorte que la communication entre eux est plus facile et plus efficace qu'entre ceux qui ne font pas

partie du même centre. Comme le montre la figure ci-dessous, qui présente un exemple de grappe de données, les nœuds dans une grappe de calcul sont organisés sous forme de racks. Une grappe de données peut être composée d'un ou plusieurs racks de données. Cette organisation sous forme de racks a l'avantage d'améliorer la performance puisque le temps de transfert réseau est minimisé et améliore la bande passante dans la totalité du centre de données.

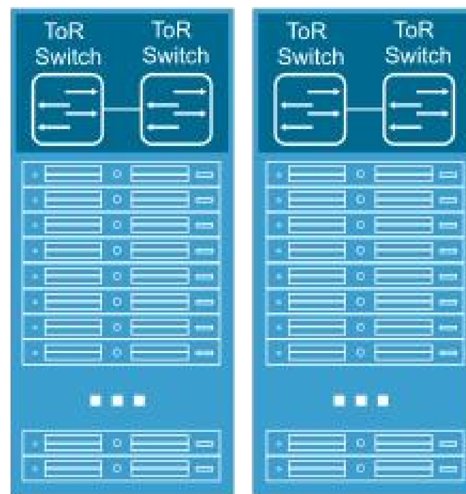



*Figure 2.11* Exemple d'une grappe de données composée de deux racks [snh]

Toutes ces architectures sont mises en place dans le domaine de gestion et de traitement de données en utilisant des techniques de distribution de données dont principalement : le partitionnement, la réplication et le parallélisme. Dans cette sous-section, je présente les solutions existantes pour le partitionnement et le parallélisme ainsi que leurs sous catégories.

- Partitionnement

Le partitionnement [SF12] est une méthode de division d'un ensemble de données et de partage de ce dernier entre multiples bases de données physiques. Les partitions - sous-ensembles créés par cette technique – sont accédées et manipulées comme une seule base de données logique tout en assurant que les détails de l'implémentation physique soient totalement gérés par le système et cachés à l'utilisateur/application final(e). L'avantage principal du partitionnement est le support du passage à l'échelle des données et la performance des opérations de lecture. Avec cette technique, un système Big Data est capable de gérer des ensembles plus volumineux et d'exécuter les requêtes plus rapidement vu qu'il a moins de données à lire. Dans la littérature, on distingue trois types de techniques de partitionnement : le partitionnement horizontal, le partitionnement vertical et le partitionnement hybride.

La première technique (le partitionnement horizontal) est définie comme un partitionnement en ensembles ayant un nombre plus réduit d'enregistrements selon un critère particulier sur leurs valeurs. Les sous catégories de ce type de partitionnement sont caractérisées par le critère de partitionnement et le schéma de partitionnement.

Le partitionnement par plage de valeurs est un schéma de partitionnement qui selon une plage de valeurs d'un ou de plusieurs attributs permet de fragmenter horizontalement un ensemble de données. L'attribut sélectionné et la valeur max de cette plage constituent le critère de partitionnement. La figure suivante illustre le principe de cette catégorie. Elle montre qu'à partir du critère composé d'un attribut (différent de la clé primaire) et une valeur max, il est possible de diviser l'ensemble de données en deux partitions.

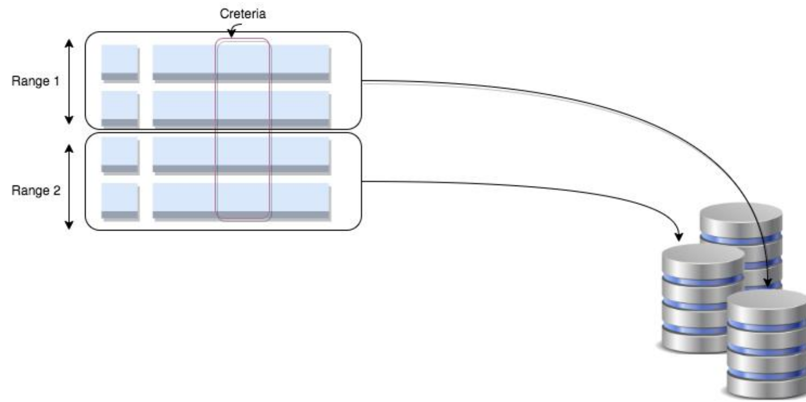



*Figure 2.12* Partitionnement horizontal par plage de valeurs

Une variante de ce partitionnement est le partitionnement basé sur les annotations (en anglais *tag aware partitioning*). Ce dernier définit des annotations pour représenter les plages de valeurs du partitionnement. La définition au préalable de ces tags permet à l'utilisateur de décider de l'emplacement des collections de données. Et vu que l'efficacité de l'interrogation d'un ensemble de données distribué dépend de la proximité des serveurs (par exemple appartenir à un même rack), utiliser cette technique se confirme être avantageuse. Elle améliore la localité des données. Dans le cadre de la distribution de données dans une grappe de données très vaste, il existe une technique de partitionnement intitulée partitionnement géographique [CSSC17] qui est plus spécifique que la précédente et qui répartit les partitions selon une indication de la zone géographique du support de stockage physique.

Une autre technique de partitionnement aussi populaire utilise une technique de hachage pour créer les fragments de données. Les techniques de partitionnement basées sur le hachage sont caractérisées par l'algorithme de hachage et les attributs de partitionnement. Une fonction de hachage simple peut être appliquée pour partitionner les données. La qualité de cette fonction dépend du risque de collision des valeurs hachées (Figure 2.13). Ce dernier influence le nombre d'accès disque et l'uniformité de la disposition des données partitionnés (en anglais *partitionning layout*).

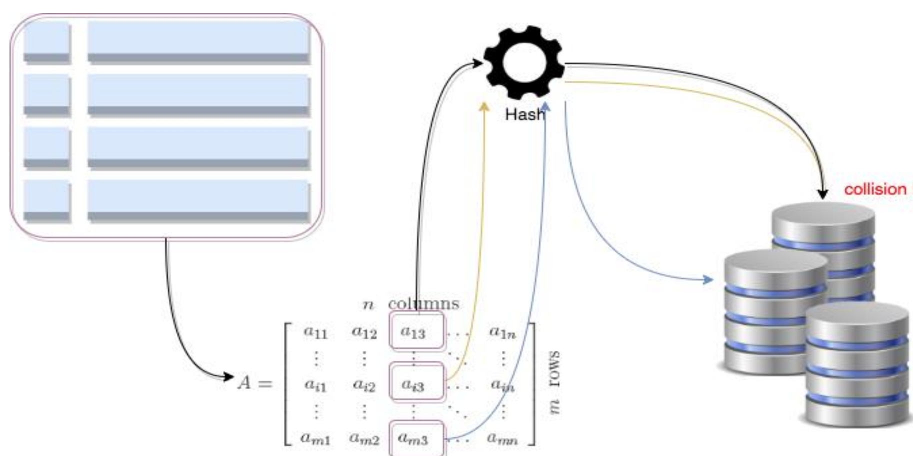



*Figure 2.13* Partitionnement par hachage et collision des partitions

Une technique de partitionnement plus avancée et qui trouve du succès dans les bases de données distribuées est le partitionnement par hachage cohérent (en anglais *consistent hashing*). Ce partitionnement utilise une fonction de hachage distribuée ainsi qu'une structure abstraite pour partitionner les données. Il associe à chaque clé d'un enregistrement des coordonnés $P_x$ (un angle géométrique) d'un cercle abstrait entre deux points $P_i$ et $P_j$. Ces points correspondent à des serveurs

de partitionnement et l'enregistrement (dont le résultat de hachage de sa clé est le point $P_x$) est stocké dans le serveur $P_j$. Ce mécanisme a comme avantage la résilience au changement de disposition (en anglais *layout*) de la distribution. En effet, il ne dépend pas du nombre de serveurs de distribution et ne nécessite pas une redistribution en cas de panne d'un serveur.

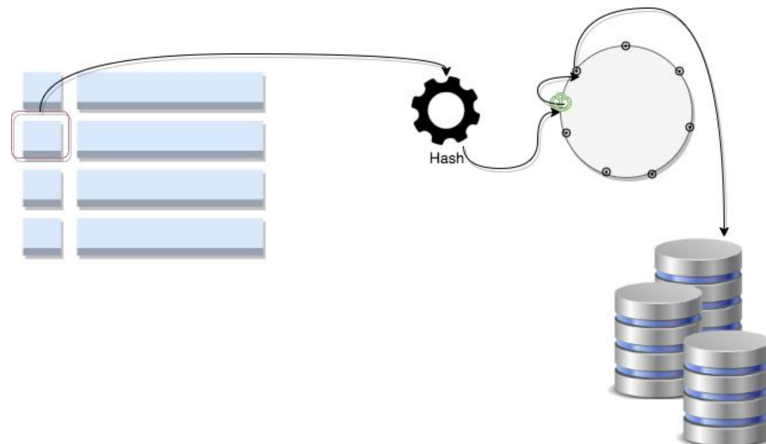



*Figure 2.14* Partitionnement par hachage consistant

- Exécution parallèle

Le parallélisme est défini comme l'exécution simultanée du même ensemble de tâches sur plusieurs ressources en même temps. C'est une application du principe algorithmique « diviser pour régner » : un problème de taille est décomposé en plusieurs sous-problèmes qui seront exécutés en même temps sur différents processeurs (localement ou en mode distribué). En Big Data, il existe plusieurs types de modèles de parallélisme : (i) le parallélisme des données qui consiste à distribuer les données et exécuter le même Workload sur toutes les données. Les systèmes qui utilisent ce type de parallélisme sont par exemple Apache Spark, Apache MapReduce, et les bases de données MPP ; (ii) le parallélisme des tâches qui permet au moteur d'exécution de partager les tâches sur le même ensemble de données. Ce modèle d'exécution est supporté par exemple par le système Apache Storm ; et finalement (iii) le parallélisme hybride qui est supporté, entre autres, par Apache Yarn et qui permet de distribuer les données et les traitements.

### 2.2.4 Conclusion

Pour atteindre une scalabilité optimale en lecture, il est toutefois recommandé de combiner les techniques de partitionnement et de réplication. Quant au stockage, il existe plusieurs possibilités de configuration des Data stores par exemple le changement du format de sérialisation ou bien l'ajout de la compression.

Définir la meilleure architecture pour une application Big Data est une série de compromis qui se multiplient avec la complexité de l'écosystème. Quelle est donc la solution optimale pour le stockage ? Peut-on développer un modèle de coût qui soit capable d'estimer la performance du traitement en variant les paramètre de stockage ? Est-il possible de modéliser avec précision la distribution (notamment le parallélisme et le partitionnement) dans les modèles de coût ?

## 2.3 Evaluation de requêtes

L'architecture des moteurs d'évaluation des requêtes a été définie dans des travaux de recherche sur le modèle relationnel et principalement pour le langage SQL. Ce type de requête permet à l'utilisateur de décrire le résultat souhaité, par exemple afficher toutes les données d'une table et les modules d'évaluation doivent définir les étapes et les méthodes nécessaires pour obtenir ce résultat. Ces étapes sont illustrées dans la figure 2.15.

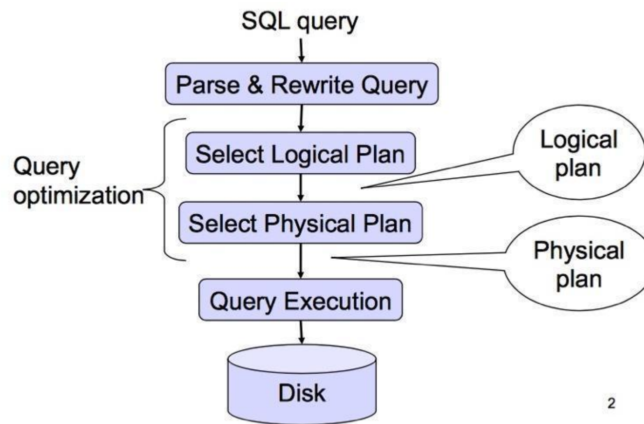



*Figure 2.15* Architecture fonctionnelle d'un optimiseur de requêtes [Bal12]

D'abord, le module d'analyse syntaxique construit un arbre d'opérateurs appelé arbre d'analyse qui représente l'enchaînement logique de traitements nécessaires pour obtenir le résultat exprimé en entrée. Dans cette étape, il utilise des outils de transformation de texte qui se basent dans leurs processus d'analyse sur une grammaire bien définie, ainsi que sur une analyse de structure de données. Par la suite, le module d'analyse sémantique [Pos] identifie les variables (résolution des variables) et élimine les erreurs de sens comme les variables ambiguës ou des appels d'opérateurs sur des variables de types incompatibles. Le résultat de cette étape est un arbre de requêtes, il est composé d'un ensemble d'opérateurs logiques qui correspondent aux opérateurs de l'algèbre relationnelle (cf SQL) plus quelques opérateurs pour les groupements et calculs d'agrégats. Ce plan logique sera ensuite envoyé au module d'optimisation qui génère à partir de ce dernier plusieurs plans équivalents produisant le même résultat, mais qui varient au niveau de l'efficacité d'exécution. La génération des plans équivalents génère l'espace de recherche en utilisant des transformations dans la structure de l'arbre d'analyse par substitution des opérateurs ou bien de l'ordre des opérateurs binaires. Selon l'architecture du moteur de traitement de requêtes, il existe plusieurs solutions de transformation de plan de requête, ces solutions sont détaillées dans la suite de cette section. L'espace de recherche est le terme utilisé pour identifier l'ensemble de plans de requêtes équivalents qui sont générés tout au long du processus d'évaluation de requête. Il est envoyé à la fin de l'analyse au module d'optimisation de requête qui, en utilisant un objectif d'optimisation et une stratégie de recherche, sélectionne le plan optimal et l'envoie au module d'exécution. Finalement, ce dernier module, en utilisant si c'est possible des techniques de parallélisme ou des techniques d'exécution en mémoire, exécute l'algorithme des opérateurs du plan de requête et envoie le résultat à l'utilisateur.

Dans la suite de cette section, nous présentons d'abord les différents opérateurs physiques dans le cas général et les principes d'optimisation des requêtes en se focalisant sur le cas le plus simple qui est le cas d'exécution centralisée. Par la suite, nous concluons par le principe d'évaluation de requêtes dans les systèmes d'évaluation de requêtes distribuées.

### 2.3.1 Opérateurs

Les opérateurs en Big Data sont généralement des opérateurs relationnels, mais pas uniquement. Il existe d'autres opérateurs spécifiques qui ont été définis pour manipuler et interroger des modèles de données plus complexes. Les opérateurs relationnels sont divisés en 3 catégories : opérateurs ensemblistes (union, intersection, différence, produit cartésien), les opérateurs spécifiques (sélection/restriction, projection, jointure) et les opérateurs dérivés de ces opérateurs de base (division, jointure externe, complément, fermeture transitive). En Big Data d'autres opérateurs ont été ajoutés pour supporter de nouveaux modèles d'interrogation comme les modèles d'interrogation

basés sur les connexions ou bien des opérateurs pour des types de données spécifiques, comme les données spatio-temporelles ou les opérateurs de recherche dans le texte.

Ces opérateurs dans les architectures des solutions d'évaluation de requêtes sont appelés des opérateurs logiques. Dans ce processus, les opérations les plus importantes, et qui sont les plus étudiées dans le domaine, sont les opérateurs d'accès aux données, les opérateurs d'agrégation de données et les opérateurs de jointure.

En effet, dans les bases de données et en Big Data, selon le modèle de stockage des données, le modèle de données et le support du parallélisme par le système, les opérateurs utilisés peuvent être implémentés différemment. Comme on l'a vu précédemment, l'accès aux données peut être basé sur la clé, sur l'agrégat ou bien sur les connexions. Il peut aussi dépendre d'un index qui est une technique qui permet d'accélérer le traitement en utilisant des données supplémentaires qui référencent les données de base. L'implémentation de l'opérateur et l'efficacité de son exécution dépend de la structure physique des données et d'autres caractéristiques comme l'existence d'un index, etc.

Dans cette section on détaille différents opérateurs et on présente les algorithmes existants et qui sont nécessaires pour exécuter ces opérateurs. L'organisation de ces opérateurs dans cette partie est faite selon la structure physique des données. Cette organisation facilite la mise en évidence des facteurs qui influencent l'efficacité de l'exécution d'une requête.

#### *2.3.1.1 Opérateurs avec accès séquentiel*

Cette opération même dans le cadre d'une exécution centralisée est très coûteuse. Elle peut engendrer des accès sur disques importants qui ralentissent le traitement. L'unité de l'accès aux données est traditionnellement faite par page disque (*bloc*) qui est un ensemble d'enregistrements qui sont stockés d'une manière contiguë. L'objectif de cette organisation physique est de minimiser les entrées/sorties nécessaires pour récupérer les données. Les opérateurs utilisent généralement cette technique pour charger les données en mémoire avant d'exécuter le traitement souhaité.

- Sélection par accès séquentiel

Le moteur d'exécution lit la table ligne par ligne pour récupérer les données (à partir de la mémoire si l'exécution est faite en mémoire ou bien sur le disque dans le cas échéant) qui satisfont la requête. Ce mode d'accès est très coûteux et généralement les optimiseurs visent à l'éviter en utilisant un index ou bien en utilisant le parallélisme. L'intérêt d'utiliser cet opérateur est dans la lecture d'un ensemble de données de taille réduite, vu que cette solution permet d'éviter les latences dues à la lecture de l'index ou bien à la distribution des données pour l'exécution parallèle des données. La complexité temporelle est à l'ordre de $O(n)$ où $n$ représente le nombre d'éléments à lire.

- Nested Loop join

L'opérateur de jointure est l'opérateur le plus coûteux en bases de données et le plus difficile à implémenter efficacement [ME92]. Le problème de l'optimisation des jointures est toujours un sujet d'actualité en recherche.

Il existe plusieurs types de jointures : (i) les jointures internes (*inner join*) [CM17] qui permettent de mettre en correspondance deux tables et retournent les enregistrements qui vérifient la condition de la jointure. Le résultat calculé par ces opérateurs n'élimine pas les enregistrements dupliqués. (ii) les jointures externes (*outer join)* retournent les enregistrements qui vérifient la condition en plus des enregistrements de la première table dans le cadre de la jointure externe gauche ou bien la deuxième table dans le cadre de la jointure externe droite.

En interne dans les moteurs d'exécution des requêtes, ces opérateurs sont traduits en un ensemble d'opérateurs physiques qui varient selon l'architecture du système ainsi que la structure des données utilisée dans le schéma physique.

L'implémentation la plus évidente de cet opérateur utilise un algorithme qui parcourt les enregistrements des tables un par un en utilisant des boucles imbriquées. Le coût temporel de l'exécution est $O(\prod_{i<r} N_i)$, avec $r$ le nombre de tables à croiser, et $N_i$ le nombre d'enregistrements dans la table i. cet opérateur est très coûteux, son pseudo algorithme est décrit dans la figure 2.16.

```
for each t1 in T1 do
  for each t2 in T2 do
    if t1.A == t2.B
       put t1° t2 in output relation
    fi
  od
od
```

*Figure 2.16* Pseudo algorithme de l'opérateur nested loop join

La complexité de l'algorithme de cet opérateur est *O(card(R)*card(S))*. De plus, le coût de transfert de la relation *R* (la relation externe) et la relation *S* vers la mémoire est égale au nombre de pages disque de ces dernières ($b_r$ et $b_s$). Finalement, dans son opération de recherche le système doit faire $R+b_r$ accès sur disque au pire cas (si la taille de la mémoire disponible ne peut supporter qu'une page disque). Si une des relations peut être stockée entièrement en mémoire le nombre d'accès est $R+b_s$, et dans le meilleur cas où les deux relations peuvent tenir en mémoire le nombre de recherches est 2. D'où la complexité totale au pire cas pour cet opérateur : $O(R*S+2(R*b_2+b_1))$ [Zho09].

#### 2.3.1.2 Opérateurs basés sur le hachage

Les techniques de hachage (utiliser une fonction à sens unique d'un ensemble de données quelconque vers un intervalle fermé d'entiers) peuvent être avantageusement exploitées pour accélérer les calculs d'opérateur de requête.

- Accès par hachage

La lecture des données par un opérateur de ce type permet au moteur d'exécution de récupérer les données en exécutant une fonction de hachage sur une clé. Dans le cas du calcul distribué en utilisant cet opérateur, l'algorithme utilise une structure de données abstraite pour localiser les partitions. La complexité temporelle pour récupérer un enregistrement en utilisant cet opérateur est constante (*O(1)*). Dans le cas de l'utilisation d'une table de hachage, cette complexité devient égale à celle d'une recherche séquentielle des clés dans la table. Il est donc nécessaire d'éviter le cas où la taille de cette table est très large. Il est donc judicieux d'avoir une fonction de hachage qui ne risque pas de produire trop de collisions des valeurs calculées. Dans le cas du calcul distribué, il est nécessaire d'ajouter la complexité temporelle de la communication des données entre les serveurs qui est égale à la taille des données multipliée par le débit du transfert.

Dans les bases de données et les solutions NoSQL, les opérateurs d'accès hachés sont supportés si le système supporte le stockage physique sous forme de clé-valeur par exemple les data stores Riak et Redis ainsi que la base de données PostgreSQL en utilisant l'extension Hstore [Pos15].

**Correspondance exacte par clé primaire**

La clé primaire est caractérisée par son unicité, utiliser cette dernière dans l'opération d'accès garantit que le premier enregistrement trouvé est le résultat de la requête.

### Jointure par hachage

Cet opérateur utilise une fonction de hachage pour organiser les tables sous forme de groupes d'enregistrements référencés par la valeur hachée obtenue sur les attributs de jointure (égalité uniquement). Finalement, l'algorithme construit résultat à partir des groupes obtenus en comparant ces valeurs hachées. Le pseudo algorithme est comme suit :

```
/* Hash relation R */
foreach tuple r ∈ R do
    put r in bucket no. h(r.A)
od

/* Probe relation Q */
foreach tuple q ∈ Q do
    foreach tuple r in bucket no. h(q.A) do
        if r.A = q.B then
            put r ∘ q in the output relation
        fi
    od
od
```

*Figure 2.17* Pseudo algorithme de l'opérateur hash join [Zur]

D'après ce pseudo algorithme on voit que le calcul se fait en utilisant deux boucles. Dans cet algorithme *h* est le symbole de la fonction de hachage, les variables *R* et *Q* sont des relations et les attributs des tuples *A* et *B* sont ceux de la condition de jointure. Sa complexité est estimée à $O(T_2*S+T_1)$, avec $T_1$ cardinalité de la table *R*, *S* la sélectivité de la condition de jointure, et $T_2$ cardinalité de table *Q*.

### *2.3.1.3 Opérateurs sur les données triées*

L'opération de tri de données est une des opérations coûteuses dans le domaine d'évaluation de requête. Il existe deux types de fonction de tri : le Tri rapide (*quicksort*) qui a comme *complexité* $O(R \log(R)+b_r)$ en moyenne et $O(R^2+b_r)$ au pire cas ou un Tri fusion externe, dont la complexité est estimé à $O(2[b_r/M]+ [b_r/b_b]( \log_{M-1}(b_r/M))-1)$. Comme dans le cadre de l'agrégation, la technique de tri est aussi utilisée pour la jointure.

### Jointure par fusion sur les données tries (Sort merge join)

L'opérateur physique « *sort merge join* » utilise un algorithme de tri sur les attributs de la condition de jointure pour organiser les données de telle sorte que la mise en correspondance des enregistrements à joindre soit plus facile à exécuter. En commençant par les tables qui sont triées sur les attributs de jointure, l'algorithme de l'opérateur parcourt les tables et à chaque fois que la comparaison est évaluée à vrai ce dernier fusionne les deux enregistrements. L'opération de tri préliminaire est très coûteuse surtout si la taille des ensembles de données est supérieure à la capacité mémoire, ce qui implique une opération de tri externe [Zho09]. Ainsi on peut conclure que la complexité temporelle est estimée à $O(R\log(R)+S\log(T)+T+R)=O(N\log(N)+N)$

### *2.3.1.4 Opérateurs d'agrégation*

Tout comme la jointure, l'agrégation est une opération coûteuse dans l'exécution en mode distribué. Cette opération a comme objectif d'exécuter une fonction d'agrégation sur les valeurs d'un ou plusieurs attributs et élimine les valeurs dupliquées d'un ou plusieurs autres attributs [CM17]. Les algorithmes utilisés pour calculer les valeurs distinctes sont généralement des algorithmes de tri ou de hachage. Il existe trois méthodes pour calculer l'agrégation. Dans la première, l'algorithme commence par regrouper les attributs selon la clé spécifiée dans la requête. Ensuite, il applique la fonction d'agrégation sur chaque groupe. La deuxième méthode utilise les techniques de tri ou de hachage qui sont utilisées dans l'algorithme d'élimination des doublons. La dernière méthode est l'agrégation à la volée qui remplace les valeurs des attributs à agréger par leurs agrégats chaque fois qu'il détecte un

nouvel enregistrement qui appartient au même groupe. Dans ce qui suit, nous présentons un tableau (tableau 2.4) qui résume ces algorithmes et donne leur complexité.

| **Algorithme** | **Principe** | **Complexité** |
|---|---|---|
| Agrégation par tri | L'algorithme de tri peut être un algorithme de tri rapide ou un tri fusion externe. Il est appelé tri externe dans le cas où la taille de l'ensemble de données ne tient pas en mémoire. Ainsi le moteur d'exécution doit trier les données sur des blocks de données en effectuant des accès sur disque pour récupérer chaque page disque. Ensuite, l'algorithme construit à partir des pages disques ordonnées des données intermédiaire en fusionnant deux à deux ces dernières. En supposant que $M$ est le nombre de pages disques en mémoire et $b_r$ est le nombre de pages disques totales de l'ensemble de données à agréger. Le nombre de fusions est égal à $log_{M-1}(b_r/M)$. | -Tri rapide : $O(R\ log(R)+b_r)$ en moyenne et $O(R^2+b_r)$ au pire cas<br>-Tri fusion externe : $O(2[b_r/M]+[b_r/b_b](\ log_{M-1}(b_r/M))-1)$ |
| Agrégation par hachage | De même que l'opérateur précédent, l'algorithme utilisé dans cet opérateur utilise la technique de hachage pour organiser les données par groupes avant l'exécution de la fonction d'agrégation. | - hachage en utilisant une fonction : $O(b_r+2b_s)= O(b_r+2[b_r/M])$. Dans la complexité temporelle de l'agrégation, il y a deux recherches en mémoire des pages disques, car la deuxième recherche permet d'exécuter l'agrégation.<br>-hachage en utilisant une table de hachage : $O(b_r+[b_r/M]+R_{dist,pred})$. Avec $R_{dist,pred}$ le nombre d'accès à la table de hachage qui est égal au nombre de valeurs distinctes des attributs de groupement.<br>-hachage en utilisant une table de hachage distribuée (DHT) : $O(b_r+[b_r/M]+R_{dist,pred}+S_{dist,pred}*débit)$ avec $S_{dist,pred}$ la sélectivité du prédicat de groupement. |
| Agrégation à la volée | Calculer à la volée des fonctions *count*, *sum*, *min*, *max*, et calculer la valeur moyenne (*avg*) à partir de *count* et *sum*. Un des bases de données qui supportent cet opérateur est SQL Server [Amo11]. | |

*Tableau 2.4* Algorithmes d'agrégation et leurs complexités

#### 2.3.1.5 Opérateurs sur les données indexées

Dans cette sous-section, nous présentons les opérateurs sur les données indexées.

- Accès par index

Le troisième type d'opérateur utilisé pour l'accès aux données exploite les index pour accéder aux données. Ce type d'accès est utile dans le cadre d'une requête de correspondance exacte (*point query*)

ou par intervalle (*range query*). Les index permettent au moteur d'exécution de lire uniquement les données nécessaires : celles qui correspondent au prédicat de la requête. Ainsi comparé à l'opérateur d'accès séquentiel qui lit tous les enregistrements (n) pour déterminer parmi eux les *s* qui vérifient le prédicat, l'accès par index fait *s* accès sur disque en plus de *p* accès pour récupérer les indexes. Généralement $p+s \ll n$. L'indexation est donc une technique qui vise à diminuer le nombre d'accès au disque lors de la recherche d'un enregistrement de données. En Big Data, les indexes secondaires ne sont pas supportés par défaut. Cette fonctionnalité peut être omise ou considérée comme un anti-pattern (par exemple dans Cassandra ou Apache Spark [ME16]).

Il existe plusieurs algorithmes de recherche de données en utilisant des indexes. L'opérateur (i) full *index scan* ou *index only scan* qui ne lit les indexes des attributs non clés (*indexes secondaires* qui sont stockés dans une structure de données séparées des données), puis les enregistrements identifiés par ces derniers. Le deuxième type d'opérateurs d'accès par index est (ii) l'*index skip scan* [*Zur*](qui peut se traduire : balayage par saut d'index). Cet opérateur lit les données en sautant les enregistrements indiqués par l'index. Il concerne le type d'indexation composée (multi-attributs) et il commence le balayage en accédant aux enregistrements par les attributs qui ne sont pas dans le prédicat de filtrage et qui appartiennent à la tête de l'index composé. Ces attributs doivent généralement avoir des valeurs denses (non distinctes). Ensuite, il parcourt les données en utilisant les indexes du prédicat. Le troisième opérateur (iii) est utilisé quand le prédicat cherché concerne une colonne qui a une contrainte d'unicité. Cet opérateur peut être aussi utilisé avec le filtrage sur les clés primaires et il ne retourne qu'un seul enregistrement. Le quatrième opérateur d'accès par index (iv) utilise le parallélisme pour accélérer l'accès en utilisant le *full index scan*. Finalement, il existe un 5ème type d'index : l'*index range scan* ou recherche basée sur index (*index seek*), permet au moteur de traitement d'améliorer la performance de l'opération de filtrage sur une plage de valeurs. Ce type d'accès est efficace dans le cas où les données qui seront retournées par le filtre sont peu nombreuses. Cet opérateur commence par parcourir l'index selon les valeurs de la condition range et détermine à partir de ce dernier les identifiants des enregistrements recherchés.

Un index dans une base de données/une solution NoSQL peut être simple ou bien composé de plusieurs attributs et est appelé ainsi index composite. Il peut être aussi partiel et définit sur un sous-ensemble de données. Il existe aussi dans la littérature plusieurs structures de données qui sont utilisées dans le schéma physique des indexes.

- Index B-tree

La structure arbre trié équilibré *B-tree* a été utilisée dans les techniques d'indexation pour rendre le parcours de l'index et des données plus efficace. L'exemple de la figure 2.18 représente un index structuré comme un arbre binaire. Cet index sert à améliorer l'accès à des données stockées sur disque sous forme d'une liste chainée en utilisant la structure matérielle de fichier en tas. L'avantage d'utiliser ce type d'index [GB10b] est son organisation hiérarchique qui limite les accès sur disque pour récupérer les données et son maintien de l'ordre des données ce qui lui permet d'assurer un temps de réponse logarithmique *O(log(n))* pour les requêtes et les opérations de gestion de données.

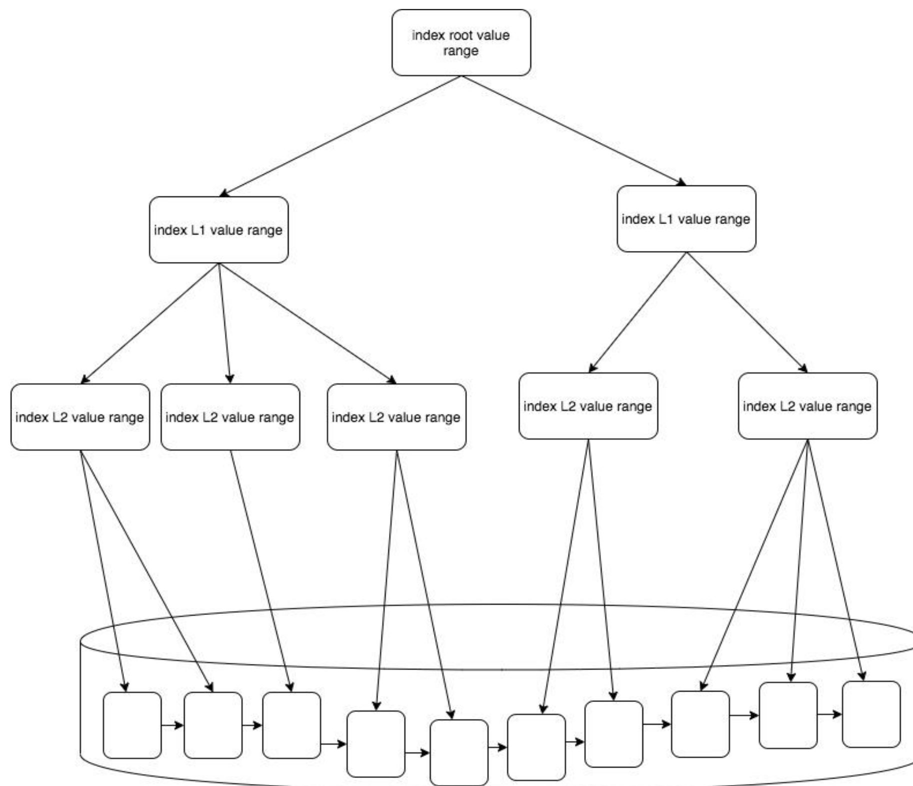



*Figure 2.18* Exemple d'un index ayant une structure de données arbre binaire

- Indexe spécifiques

Il existe aussi en Big Data, tout comme en systèmes de gestions de bases de données classiques, des solutions d'indexation qui accélèrent l'accès et les traitements sur des données spécifiques. Un exemple de cette catégorie serait les index de recherche dans le texte brut comme les indexes GIN [Pos21] et GIST [Posa] de PostgreSQL.

Le premier type d'index est utilisé avec les index composés et facilite la recherche de mots spécifiques dans un texte. L'opérateur d'accès aux données en utilisant ce type d'index effectue une recherche sur les mots clés qui sont stockés sous forme d'index et qui référencent les attributs où se trouvent les données. Ce type d'index est appelé index inversé et il trouve son utilisation principalement dans les moteurs de recherche et dans le domaine de la recherche d'information. Le deuxième type d'index est caractérisé par son inexactitude. Cette méthode utilise une fonction de hachage sur les clés, mais elle vérifie, à chaque recherche d'enregistrement par index, par un accès séquentiel aux données pour être sûre d'avoir le bon enregistrement (les erreurs sont dues aux collisions de hachage). L'accès en utilisant l'index GIN est plus efficace qu'en utilisant sa solution alternative. La complexité temporelle du premier est à l'ordre de $O(log(U))$ avec $U$ le nombre de mots uniques dans le dictionnaire de l'index. Par contre, la deuxième solution dépend de la performance de la lecture d'une quantité importante de données en mémoire à chaque vérification de l'exactitude de la fonction de hachage.

Nous présentons maintenant un deuxième type d'index intitulé *index bitmap* et qui permet de croiser les enregistrements avec les valeurs de l'attribut indexé sous forme binaire.

| Horodate | Cd_prof | Plag_puis | Nb_points_s | Tot_nrj_s |
|---|---|---|---|---|
| 2018-09-25T03:30:00+02:00 | PRO5 | P1: ]0-3] kVA | 312655 | 120944217 |
| 2018-09-25T03:30:00+02:00 | PRO5 | P7: ]18-24] kVA | 3041 | 15570754 |
| 2018-09-25T03:30:00+02:00 | RES1 (+ RES1WE) | P2: ]3-6] kVA | 12507439 | 882744147 |
| 2018-09-25T03:30:00+02:00 | RES11 (+ RES11WE) | P3: ]6-9] kVA | 2042916 | 217876153 |
| 2018-09-25T03:30:00+02:00 | RES2 (+ RES5) | P4: ]9-12] kVA | 1967285 | 488229885 |

*Figure 2.19* Exemple d'index bitmap (a)

Index bitmap sur *cd_prof* :

| Horodate | PRO5 | RES1 (+ RES1WE) | RES11 (+ RES11WE) | RES2 (+ RES5) |
|---|---|---|---|---|
| 2018-09-25T03:30:00+02:00 | 1 | 0 | 0 | 0 |
| 2018-09-25T03:30:00+02:00 | 1 | 0 | 0 | 0 |
| 2018-09-25T03:30:00+02:00 | 0 | 1 | 0 | 0 |
| 2018-09-25T03:30:00+02:00 | 0 | 0 | 1 | 0 |
| 2018-09-25T03:30:00+02:00 | 0 | 0 | 0 | 1 |

*Figure 2.20* Exemple d'index bitmap (b)

Dans la figure 2.20 nous présentons un exemple d'un index bitmap sur la colonne *cd_prof* de l'ensemble de données de consommation des particuliers en énergie électrique. L'algorithme d'indexation a calculé un encodage binaire pour cet attribut. Il associe à chaque valeur possible une colonne de type binaire. La complexité temporelle d'une recherche en utilisant cet index est *0(n)* mais il souffre d'une limite pour la taille de l'index. Il est conseillé de l'utiliser dans le cas où le nombre de valeurs distinctes de l'attribut indexé est petit.

Un autre type d'index spécifique concerne les données géospatiales. Il est supporté par PostgreSQL et il est appelé BRIN acronyme pour « *block range index* » qui permet d'organiser les attributs d'une table très massive de telle sorte que les attributs qui sont physiquement stockés ensemble sont accédés plus rapidement. Cet algorithme fixe des plages de valeurs pour chaque bloc et référence les pages stockées sur disque.

#### *2.3.1.6 Jointure parallèle*

Comme on l'a vu précédemment, la jointure nécessite le transfert des données de toutes les tables qui participent à l'opération vers le nœud de calcul. En Big Data, faire une opération sur des données *n* fois plus volumineuses (avec *n* le nombre d'ensembles de données qui participent à l'opération de jointure) rend le calcul plus coûteux. En effet, ces systèmes utilisent les techniques de distribution pour assurer la faisabilité du calcul.

La communication réseau est un problème majeur dans ce type d'application. Plusieurs systèmes éliminent la possibilité d'exécuter des opérateurs binaires comme la jointure et l'union à cause de leurs coûts. Mais il existe d'autres systèmes qui supportent ces types d'opérations et proposent des opérateurs spécifiques. Ils peuvent aussi définir de nouveaux opérateurs pour la distribution des données comme les opérateurs : *Split [JBK+20]* et *Shuffle [VMP17]* que l'on trouve dans les APIs des

moteurs de traitements massifs, comme Apache Spark ou Apache MapReduce. Dans ce qui suit nous présentons les opérateurs spécifiques de jointure.

- « Broadcast » et « shuffle » hash join

Dans les systèmes distribués comme Apache Spark, l'optimiseur de requêtes peut choisir à la place de l'opérateur *hash join* un *broadcast hash join* ou un *shuffle hash join*. Le premier type de jointure par hachage distribuée un des ensembles des données mis en jeu dans la requête sur les nœuds (serveurs) qui contiennent les partitions de la deuxième relation. La table à distribuer est généralement la plus petite, vu que la transmission des données à travers le réseau est très coûteuse. L'exécution de la jointure dans chaque nœud cible est faite comme une exécution locale et finalement le résultat est reconstruit à partir des différents résultats intermédiaires obtenus de l'étape précédente. Quant au *shuffle hash join* [HK17], il partitionne une ou les deux ensembles des données sur la clé de jointure (vu que l'heuristique de la taille de l'ensemble de données n'est pas satisfaite), exécute la fonction de jointure par hachage sur chaque partition et finalement reconstruit le résultat. Ces opérateurs souffrent d'un coût élevé, lié à la distribution des données pour permettre l'exécution de la jointure sur des volumes de données aussi importants et à la reconstitution du résultat à retourner à l'utilisateur.

- Problème de skew dans la jointure distribuée

La jointure selon une clé qui a des valeurs qui ne sont pas distribuées crée un déséquilibre dans la distribution des données dans les serveurs lors du traitement. Cette distribution inégale des données sur les nœuds de distribution a lieu dans le cas de hachage sur la valeur de la clé de jointure. Ce problème existe aussi dans le cadre du partitionnement des données en utilisant la technique de hachage. La performance de l'accès et de la recherche des données dépend du nœud qui contient le plus de données.

#### *2.3.1.7 Conclusion*

Pour conclure on présente ce tableau qui met en relation les opérateurs et leurs caractéristiques et on propose une estimation de la complexité temporelle des opérateurs en utilisant la notation grand O. Comme dit précédemment, l'accès dans les moteurs d'exécutions est fait par pages disque de données et le nombre d'accès peut être exprimé comme le nombre d'enregistrements (/d'index) accédés divisé par la taille d'une page disque. Mais vu que dans un système de stockage la taille de page disque est la même (elle est configurable et elle a une valeur par défaut, par exemple dans Oracle elle est égale à 8kb [Oraa]), on exprime la complexité par le nombre d'enregistrements.

| Opérateur | Distribué | Index | Parallèle | Complexité temporelle |
|---|---|---|---|---|
| Lecture séquentielle (*sequential scan*) | _ | _ | _ | O(R) |
| Lecture séquentielle distribuée (*sequential scan*) | X | _ | _ | $O(R+t_{net_io})$=O(R+R*débit) |
| Sélection en utilisant l'accès par hachage | _ | _ | _ | O(1) |
| Sélection en utilisant l'accès par hachage distribué | X | _ | _ | $O(t_{net_io})$=O(R*débit) |
| Sélection en utilisant l'accès par index btree | _ | X | _ | O(Slog(S)) |
| Sélection en utilisant l'accès par index haché | _ | X | _ | O(S) |

*Tableau 2.5* Complexité de quelques opérateurs physiques d'accès aux données

Le tableau 2.5 montre qu'il est plus efficace de lire des données à partir d'un index. Vu que le nombre de données qui satisfont le prédicat de l'index (*S*) est plus réduit que le nombre total de données, il est plus judicieux d'utiliser un index.

### 2.3.2 Optimisation classique (dans les systèmes NoSQL)

Nous avons présenté brièvement les étapes de traitement de requête. Dans cette section, nous expliquons le principe général de l'optimisation de requête (troisième étape de la figure 2.21) qui est déclenchée par l'étape de réécriture de requête et qui est suivie par celle de la génération du code, puis de l'exécution.

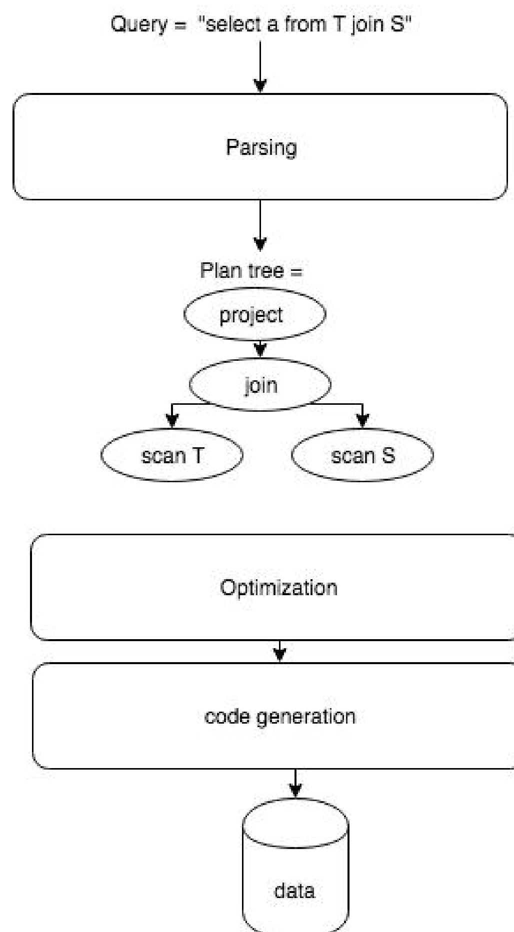



*Figure 2.21* Etapes d'optimisation de requêtes dans une base de données

La réécriture de requête est une technique qui permet de générer à partir d'un plan de requête analysé reçue de la part de l'analyseur syntaxique et sémantique un ensemble de plan logiques équivalents appelés aussi l'espace de recherche. Ensuite le moteur d'exécution remplace les opérateurs logiques par toutes les combinaisons possibles d'opérateurs physiques dans le plan. Cette étape est appelée génération de plan physique et les opérateurs physiques représentent différentes implémentations qui à l'exécution différente dans leur performance mais retournent le même résultat.

L'objectif de l'optimisation est de sélectionner le plan de requête qui s'exécute en un temps minimal et qui minimise l'utilisation des ressources. Cette étape exécute une recherche dans un ensemble de plan d'opérateurs (l'espace de recherche) et utilise plusieurs stratégies pour déterminer le plan optimal. Cette section résume les différents travaux dans le domaine et présente les techniques principales de l'optimisation des requêtes.

### 2.3.2.1 Principe général

Pour mieux maîtriser la première étape qui est l'analyse de l'expression de la requête, nous considérons l'exemple suivant illustré dans la figure 2.22.

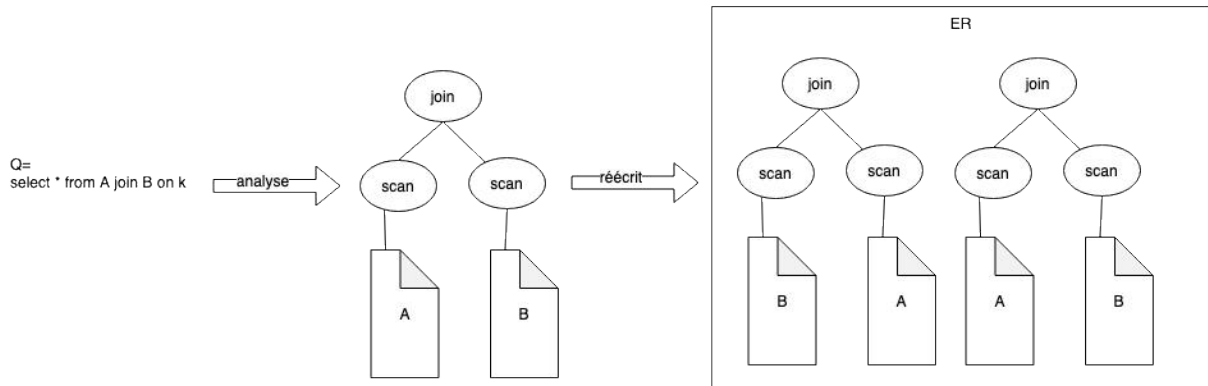



*Figure 2.22* Exemple d'analyse et de réécriture de requête SQL

Dans cet exemple, une requête de jointure est analysée et un premier plan (dans ce cas structuré comme un arbre) est construit. Il est constitué des opérateurs logiques *: join et scan*. Par la suite intervient un processus de validation des paramètres associés à cet arbre (notamment les ensembles de données, leurs références et leurs attributs) pour finalement obtenir un plan logique analysé.

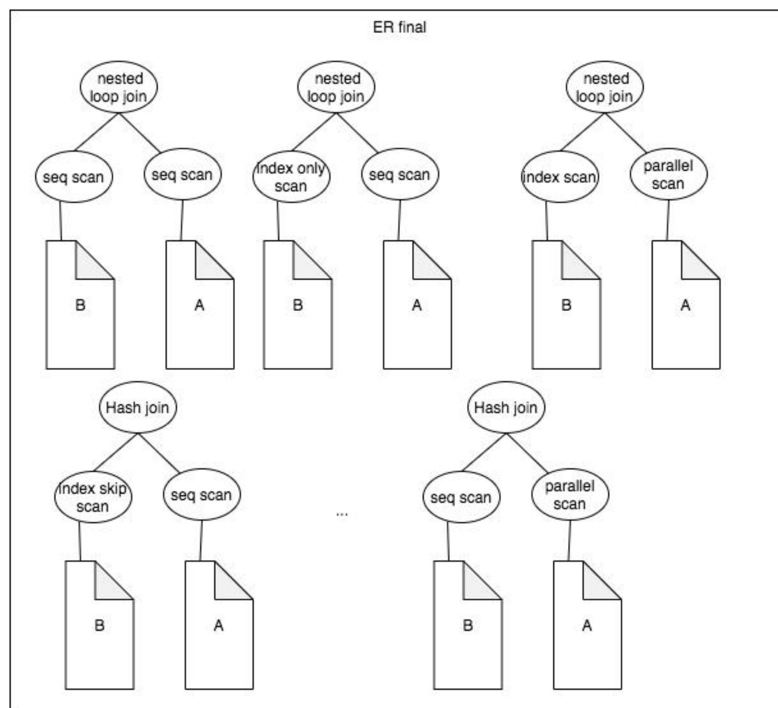



*Figure 2.23* Exemple d'espace de recherche de plan optimal

En gardant le même exemple, on explique la deuxième phase de l'optimisation. La figure 2.23 présente l'ensemble de plans physiques correspondant au premier plan de l'étape précédente. Elle montre que l'optimiseur a sélectionné les différents algorithmes d'accès aux données et de jointures qui sont supportés par le moteur d'exécution, et que pendant la génération des plans physiques plusieurs algorithmes sont considérés pour les différents opérateurs logiques, ce qui multiplie davantage le nombre de plans de requête dans l'espace de recherche *ER*. Par exemple, le choix de la jointure par *nested loop join* ou *hash join* dépend des caractéristiques des données exprimées sous forme de

métadonnées. Dans cet exemple, on remarque aussi que seulement l'ensemble de données *B* est indexé. Le chemin d'accès à ce dernier comporte des opérateurs d'accès par index.

Finalement, en répondant à un objectif d'optimisation, par exemple minimiser une valeur de coût, l'optimiseur cherchera le plan optimal en comparant les coûts des plans de l'espace de recherche. Dans ce qui suit, nous présentons l'espace de recherche du module d'optimisation, nous détaillons les structures utilisées pour les plans de requête physiques et logiques et nous introduisons les stratégies de recherche.

### Spécification de l'espace de recherche

L'expression de l'algèbre relationnelle de la requête, obtenue à l'étape de *parsing* et envoyée à l'optimiseur sous format textuel est appelée le plan logique initial. Ce plan est ensuite transformé en plusieurs combinaisons de structures possibles du plan initial. Cette analyse est nécessaire, car chaque structure sera exécutée d'une manière différente. À titre d'exemple, on considère la requête suivante :

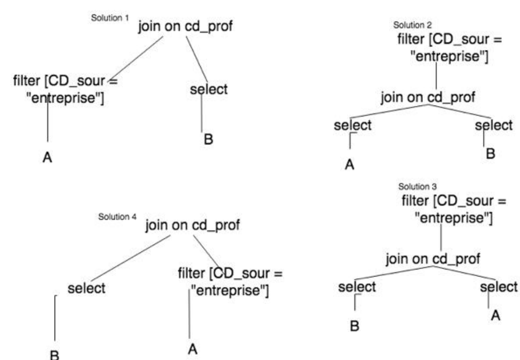
SELECT A.*,B.FAM_PROF

FROM asma_AGG_CONSO_INF36_SUP36 A

JOIN HNI_TRANSCO_FAM_PROF B

ON A.CD_PROF=B.CD_PROF

Where CD_SOUR ="Entreprise"




L'espace de recherche (*ER*) est un ensemble de plans de requêtes équivalents qui a été généré à partir du plan logique envoyé par l'analyseur de syntaxe au module de réécriture de requête. Dans cette sous-section, on s'intéresse à la structure de l'espace de recherche et plus précisément aux plans qui peuvent se trouver dans ce composant.

Le plan logique est composé d'un ensemble d'opérateurs de l'algèbre relationnelle. L'algèbre relationnelle est une algèbre fermée où chaque opérateur consomme des tables et produit une table. C'est une algèbre complète qui permet d'exprimer tous les expressions en utilisant les opérateurs de base (sauf pour la fermeture transitive). Dans la deuxième phase d'optimisation les opérateurs logiques sont remplacés par des opérateurs physiques correspondants à une implémentation particulière de ces opérateurs logiques dans les systèmes. On obtient donc l'arbre du plan physique qui est une combinaison d'algorithmes.

La représentation des plans des requêtes a deux modèles possibles dans la littérature : les arbres *abstract syntax tree* (AST) ou les graphes orientés acycliques (DAG). Les AST ont été adoptés dans la majorité des systèmes existants vu la simplicité de la manipulation de ces structures comparées aux DAG.

Un arbre AST est une structure de données qui a été utilisée dans les logiciels d'interprétation des langages et des compilateurs [FLG07]. Cette structure est un arbre qui est composé de deux types de nœuds : les nœuds non feuille et qui correspondent à des opérateurs dans notre cas et les nœuds feuille pour les symboles (*token*) qui représentent les variables (dans notre domaine les ensembles de données) ou bien les valeurs constantes.

Selon cette définition, un AST est un cas particulier de DAG où les nœuds du graphe ne peuvent avoir au plus qu'un seul prédécesseur. Dans le domaine de l'exécution des requêtes, les DAG peuvent être

utilisés comme une alternative aux AST, vu qu'ils permettent de réutiliser les résultats de calculs qui ont été déjà fait pour d'autres opérateurs, par exemple scanner une seule fois une table, pour être utilisés dans plusieurs opérateurs et ainsi limiter le temps d'exécution et l'espace mémoire occupé pour le traitement.

Ce plan logique subira une autre transformation qui permettra de déterminer pour chaque opérateur logique l'algorithme avec lequel il sera exécuté. On obtient ainsi un plan physique qui a généralement la même structure que le plan logique. Dans ce qui suit nous présentons les concepts reliés à l'exploration de l'espace de recherche notamment les solutions d'énumération des plans logiques et ceux de la restriction des plans physiques.

#### *2.3.2.2 Exploration de l'espace de recherche*

En utilisant un modèle de coût, une heuristique ou un ensemble de règles, on parcourt tous les plans de l'espace de recherche pour trouver la solution optimale qui permet d'exécuter la requête. Les modèles de coûts sont des formules mathématiques qui servent à estimer une valeur qui représente le coût d'exécution d'une requête. Cette estimation est calculée à partir d'une modélisation de l'utilisation des ressources matérielles disponibles pour le moteur d'exécution et elle est exprimée sous formes d'une unité (par exemple le nombre de pages disques traités ou accédés) qui dépend des caractéristiques du système de stockage et du moteur d'exécution. D'un autre côté, les heuristiques sont des choix arbitraires déterminés par les expériences et le savoir-faire des experts dans le domaine. Un exemple d'heuristique serait le choix de l'ordre des jointures en se basant sur la taille des ensembles de données mis en jeu. Ces heuristiques sont aussi utilisées dans les optimiseurs de requêtes basés sur les règles. Ces règles sont exprimées généralement en utilisant une structure conditionnelle « if – then » et ils peuvent par exemple choisir entre plusieurs chemins d'accès qui sont ordonnées comme dans le cas de la base de données Oracle [Orab] si les conditions des règles sont vérifiées.

Mais avant d'explorer l'espace de recherche, deux étapes primordiales sont utilisées pour générer ce dernier. Une première étape est l'énumération des plans équivalents pour découvrir de nouvelles solutions qui potentiellement s'exécutent plus efficacement que le plan, et une deuxième étape de restriction de l'espace de recherche dans l'objectif de limiter la recherche et de diminuer le temps nécessaire pour chercher la solution optimale.

- Enumérer les plans

À l'aide de transformations algébriques, l'optimiseur de requête explore les différentes structures d'arbres d'opérateurs logiques pour générer des plans plus ou moins efficaces. Un exemple de transformation serait d'inverser l'ordre des opérateurs binaires. Dans la littérature, construire un espace de recherche à partir des requêtes analysées est fait en utilisant des règles d'équivalences qui ont été proposées et améliorées en exploitant les propriétés des opérateurs de l'algèbre relationnelle. Dans ce qui suit nous présentons des règles d'équivalence qui sont utilisées dans l'énumération des plans logiques de *ER*.

- Règle de commutativité : les opérateurs binaires relationnels sont commutatifs, l'ordre de l'exécution des opérateurs n'influence pas le résultat. Par exemple ; l'expression : « join(A,B) » est équivalente à l'expression « join(B,A) ». Par contre, le temps d'exécution peut dépendre de cet ordre, d'où l'intérêt d'explorer toutes les combinaisons possibles générées par cette règle.

- Règle d'associativité : tout groupement d'opérateurs est valable. Par exemple, les deux expressions suivantes sont équivalentes : « join(join(B,A) ,C)» et « join(B,join(A,C))». Cette règle concerne aussi les opérateurs binaires.

- Envoi des opérations de sélection aux feuilles de l'arbre : cette règle n'est pas générale, il existe des cas où appliquer cette règle ne conduit pas toujours à l'optimalité, surtout dans le cas de prédicats ayant des conditions complexes [GMUW14].
- Distribution des projections vers les feuilles de l'arbre en ne gardant que les attributs utiles pour la suite du calcul afin de minimiser le volume des résultats intermédiaires
- Équivalence entre jointures et produits cartésiens suivis de sélections.

En exploitant ces règles, l'optimiseur de requête génère à partir du plan logique un espace de recherche dense qui ne peut pas être exploré en un temps polynomial d'où l'importance des techniques de restriction de l'espace de recherche.

- Restriction de l'espace de recherche

L'optimiseur peut décider de ne plus explorer une partie de l'espace de recherche si une heuristique ou bien des règles de restriction indique que l'optimum n'existe pas dans cette partie. Cette stratégie est nécessaire pour des requêtes dont l'optimisation peut prendre plus de temps que l'exécution (exemple : l'optimisation d'une jointure de *N* relations qui génère *N!* plans équivalents en appliquant uniquement les règles d'associativité et commutativité des plans de requêtes).

Les heuristiques sont utilisées dans le domaine de l'optimisation de requête pour trouver une solution approchée à un problème NP-complet par une approche moins complexe [OV11]. En termes d'exemples de l'heuristique largement utilisée dans le domaine, on trouve l'évaluation des opérateurs unaires, par exemple la sélection et la projection, le plus tôt possible en descendant ces opérateurs dans le bas de l'arbre de requête. Une autre heuristique dans l'optimisation des jointures limite les optimiseurs à ne produire que des plans ayant une forme unique, par exemple des arbres linéaires.

Ces heuristiques ne conduisent pas forcément à trouver l'optimum global, mais ils évitent les pires cas et surtout ils accélèrent l'optimisation. Elles sont généralement implémentées dans les moteurs d'évaluation de requêtes en utilisant des règles logiques.

Dans le cadre de l'évaluation de requête distribuée, il existe aussi des techniques de restriction de l'espace de recherche en utilisant des règles, par exemple celles qui permettent d'éviter de créer, à partir de prédicats incompatibles, des relations vides, ou celles qui évitent la production de résultats intermédiaires ayant des attributs inutiles à partir de fragments distribués horizontalement ou verticalement.

Vu que notre solution ne s'appuie pas sur la restriction de l'espace de recherche et en particulier sur les heuristiques, nous nous sommes limité à ne présenter que quelques heuristiques. Dans ce qui suit, nous présentons les techniques de recherche du plan optimal suivi par un état de l'art sur les modèles de coûts en Big Data.

#### *2.3.2.3 Stratégies d'exploration de l'espace de recherche*

Dans cette section, nous expliquons les différentes stratégies existantes basées sur le calcul de coût d'exécution d'une requête et nous présentons brièvement celles qui sont utilisées pour la recherche du plan optimal. Le modèle de données de référence utilisé pour expliquer les notions abordées dans cette partie est le modèle relationnel (sauf mention spécifique). Ce choix est motivé par le fait que ces techniques ont été d'abord utilisées avec ce modèle.

- Stratégies déterministes

Les stratégies déterministes (*greedy algorithms*) constituent une catégorie d'algorithmes qui garantissent la sélection du plan optimal. Dans cette catégorie, on trouve la recherche exhaustive qui

traverse la totalité de l'espace de recherche pour trouver la solution optimale ou les recherches basées sur les heuristiques, qui sont aussi déterministes, car elles ont démontré leur efficacité en pratique.

Les algorithmes de recherche exhaustive comme leur nom l'indique parcourent la totalité de l'espace de recherche et finissent toujours par trouver l'optimum global. Pourtant, elles sont caractérisées par une faible performance, vu que sa complexité algorithmique est égale à *O(N)*. Avec l'augmentation de la taille des requêtes et leur complexité, la recherche exhaustive devient inefficace [IK90] d'où l'intérêt de l'améliorer.

L'approche naïve parcourt l'espace de recherche séquentiellement et trouve le plan optimal. Pourtant, il est possible d'utiliser les heuristiques pour identifier les plans coûteux et limiter le temps de parcours comme dans l'algorithme *A* [LG11]*. Cet algorithme est précis dans le cadre de l'optimisation d'une requête qui fait intervenir un nombre réduit d'ensembles de données (relations dans le modèle relationnel) [JK84]. Il choisit des solutions sous-optimales, vu qu'il souffre du problème de sélection d'un optimum local [GTS06]. L'algorithme *A** a été inventé dans le domaine d'intelligence artificielle pour résoudre le problème de recherche du plus court chemin dans un réseau sans avoir à parcourir toutes les solutions possibles. Depuis le nœud de départ, il commence par parcourir les nœuds voisins et sélectionner le nœud suivant qui satisfait une heuristique bien définie et en s'assurant que la distance parcourue entre le premier nœud jusqu'au nœud suivant est minimale.

Le problème de minimisation est – comme le montre le pseudo algorithme de la figure [Caz] ci-dessous – faite sur une valeur $f$ pour chaque nœud $n$, tel que la valeur de $f(n) = d(n)+h(n)$ sachant que $h$ est une fonction qui calcule une heuristique.

```
void AStar() {

bool outOfMemory = false;

Node * node = nodeWithSmallestf(); while(!node->final() && !outOfMemory){ outOfMemory = develop(node);
```

*Figure 2.23.b. Pseudo code de l'algorithme A**

Dans le domaine de l'optimisation de requêtes, les stratégies pour générer l'arbre des opérateurs [GTS06] (comme le choix de l'ordre des jointures ou le choix des opérateurs physiques) sont modélisés comme des chemins, les coûts des opérateurs sont modélisés comme la distance et les heuristiques utilisées correspondent à des heuristiques utilisés couramment dans le domaine des bases de données.

- Stratégies aléatoires

Ce type de stratégie résout un problème de recherche d'optimum local. Ce dernier commence par faire une recherche locale en profondeur d'abord (*depth first search* DFS) [LV91]. Ensuite les points de départ de la recherche sont améliorés en utilisant des transformations. Un exemple de point de départ serait un opérateur physique et une transformation serait la réécriture de l'arbre de la requête en effectuant des choix pour les algorithmes alternatifs des opérateurs. L'arbre de requête initial et les ensembles des arbres créés par les transformations sont appelés des états. L'algorithme parcourt ses états aléatoirement [Ka99] guidé par les transformations et en utilisant un modèle de coût jusqu'à atteindre l'état optimal ou la fin du temps alloué à l'optimisation.

Il existe plusieurs catégories d'algorithmes de recherche aléatoire, et ces catégories ont été améliorées dans plusieurs travaux de recherche. La première catégorie, recherche locale ou bien recherche *Monte-Carlo [Ros99]* représente l'algorithme le plus simple de recherche aléatoire.

Cet algorithme de recherche local est itératif. Il génère à chaque itération l'ensemble des solutions voisines $S_i$ [DCM19] et sélectionne comme point courant l'état qui a un coût inférieur à celui de l'état examiné. L'algorithme se termine si tous les états ont des coûts supérieurs à l'état examiné. La solution générée par cet algorithme est un optimum local vu qu'elle n'explore pas les états générés à partir des états qui ont des coûts supérieurs à l'état actuel d'une itération donnée et qui peuvent contenir l'optimum global. Pour trouver cet optimum global il faut sacrifier de la performance mais sans avoir à générer tous les solutions possibles.

D'autres algorithmes ont été proposés dans cette logique comme recherche aléatoire par « *Simulated annealing* ». Cette stratégie de recherche utilise une méthode méta-heuristique pour résoudre le problème d'optimisation globale : en utilisant une heuristique, elle décide d'éliminer une partie de l'espace de recherche et de ne pas l'explorer. En pratique, certaines de ces heuristiques ont montré leur efficacité et cette méthode est plus efficace d'éliminer arbitrairement une partie de l'espace de recherche. Cette stratégie est mieux adaptée aux requêtes complexes accédant à plusieurs relations. L'algorithme détermine plusieurs optimums locaux [IW87] et permet de sélectionner une solution plus proche de l'optimum global que les autres méthodes. Comme le montre le pseudo code de la figure 2.24, Il utilise une approche probabiliste pour permettre l'exploration des états qui ont un coût supérieur au coût de l'état actuel et ainsi pouvoir vérifier l'existence dans l'espace de solution voisine de ce dernier un état qui minimise la fonction de coût.

**Simulated annealing**

1. **Initialization** $i := i_{start}$, $k := 0$, $c_k = c_0$, $L_k := L_0$);
2. **Repeat**
3. **For** $l = 0$ **to** $L_k$ **do**
   - **Generate a solution** $j$ **from the neighborhood** $S_i$ **of the current solution** $i$;
   - **If** $f(j) < f(i)$ **then** $i := j$ ($j$ **becomes the current solution);**
   - **Else,** $j$ **becomes the current solution with probability** $e^{\left(\frac{f(i)-f(j)}{c_k}\right)}$;
4. $k := k + 1$;
5. **Compute**($L_k, c_k$);
6. **Until** $c_k \simeq 0$

*Figure 2.24* Pseudo-code de l'algorithme simulated annealing

Quant à la recherche aléatoire par « *Iterative improvement* » [ABM07] elle exécute un ensemble de recherches locales de l'état optimal. L'algorithme utilisé dans cette technique est l'algorithme *hill climbing*. Il commence par un état initial aléatoire à partir duquel il effectue une recherche de l'optimum local [Feg98]. L'algorithme *hill climbing* a comme objectif de trouver le maximum (par défaut) ou bien le minimum d'une fonction *f* bien déterminée. Il permet de parcourir l'ensemble des solutions état par état, et remplace l'état courant par son successeur si le résultat de *f* pour cet état est supérieur (/inférieur) à celui de *f* pour l'état courant. A partir de cet algorithme, on a défini l'algorithme du *gradient descendant[Rud17]* qui minimise une fonction en calculant sont gradient *delta(f)*. Le calcul du gradient permet de déterminer le sens de parcours pour sélectionner l'état voisin dans la prochaine itération. Cette méthode a été utilisée aussi dans le domaine d'apprentissage automatique pour l'optimisation et la minimisation de l'erreur d'apprentissage.

- Programmation dynamique

Tout comme les stratégies aléatoires, la programmation dynamique utilise une approche algorithmique pour trouver la solution optimale sans explorer la totalité de l'espace de recherche. Le domaine de la programmation dynamique utilise le principe diviser pour régner pour arriver à résoudre un problème complexe. Ce dernier est divisé en sous problèmes élémentaires qui conduisent à atteindre l'objectif. La génération des problèmes dans ce type de stratégies se fait d'une manière

ascendante (*bottom-up fashion*) et non récursive. Les sous-solutions sont combinées une à une jusqu'à créer un espace de recherche complet (ou restreint à l'aide d'heuristiques).

Ces algorithmes ont été prouvés efficaces pour trouver l'ordre optimal des jointures. Ils diffèrent du point de vue de leurs limites, de la qualité de sélection de la solution optimale et de l'efficacité de l'algorithme.

- DPsize [MN08]

Cet algorithme est basé sur une heuristique dont l'hypothèse est la suivante : l'ordre de jointure optimal est celui qui commence de la relation la plus petite.

```
DPsize(R = {R1,...,Rn})
for each Ri ∈ R
    dpTable[1][{Ri}] = Ri
for each 1 < s ≤ n ascending // size of plan
    for each 1 ≤ s1 < s // size of left subplan
        for each S1 ∈ dpTable[s1], S2 ∈ dpTable[s − s1]
            if S1 ∩ S2 ≠ ∅ continue
            if ¬(S1 connected to S2) continue
            p = dpTable[s1][S1]⋈dpTable[s − s1][S2]
            if dpTable[s][S1 ∪ S2]=∅ ∨ cost(p)<cost(dpTable[s][S1 ∪ S2])
                dpTable[s][S1 ∪ S2] = p
return dpTable[n][{R1,...,Rn}]
```

*Figure 2.25* Pseudocode de l'algorithme DPsize [Feg98]

Comme le montre la figure 2.25, l'algorithme génère pour chaque ensemble de relations, toutes les combinaisons telles qu'il y a un prédicat de jointure entre ces relations et en vérifiant les propriétés d'associativité et de commutativité. Ensuite, il stocke ses sous-plans de requête dans une collection de correspondance clé-valeur, appelée une table de programmation dynamique, de sorte que la clé soit une combinaison de relation et la valeur soit le plan de la requête. La construction de la structure *dptable* est faite en itérant sur les sous-plans dans l'ordre ascendant de la taille estimée de leurs résultats et en générant les combinaisons de ces sous-plans, s'il existe un prédicat de jointure entre ces sous-plans. Si la nouvelle combinaison de sous plan existe dans *dpTable* le plan résultat qui est gardé est celui qui a le coût le plus faible. À la fin de l'algorithme on obtient un plan sans produit cartésien, qui minimise le coût et qui se trouve à la dernière position de *dpTable*.

- DPsub

Comme l'algorithme précédent, DPsub [MN08] est un algorithme qui utilise la programmation dynamique d'une manière ascendante.

Le pseudo code de la figure 2.26 montre que l'algorithme commence par énumérer les sous-ensembles de plans à partir de la requête de jointure initiale. Il commence par initialiser la structure *dpTable* (dans la figure *BestPlan*) et puis énumère toutes les combinaisons possibles d'ensemble de relation en itérant jusqu'à $2^n$-1. Cette boucle s'exécute plus rapidement que le parcours dans l'algorithme précédant, puisqu'itérer sur un nombre occupe moins de mémoire. Sauf que les plans générés peuvent être erronés vu que des relations qui ont été combinés par l'algorithme peuvent ne pas avoir un prédicat de jointure entre elles. Par exemple, en considèrent la jointure suivante : $A_a \triangleright\triangleleft_a B_c \triangleright\triangleleft_c C$, l'algorithme peut générer la combinaison invalide suivante : $A_a \triangleright\triangleleft_c C$ et qui aurait résulté en un produit cartésien. Ainsi les créateurs de l'algorithme ont inventé la notion de connectivité qui permet d'éviter ce cas de figure comme on le voit dans le pseudo code. Finalement, si les relations de gauche $S_1$ et de droite $S_2$ sont connectées, l'algorithme met à jour la structure *dpTable* par le coût de la jointure entre $S_1$ et $S_2$ s'il est inférieur à l'ancienne valeur.

```
DPsub
Input: a connected query graph with relations R = {R_0, ..., R_{n-1}}
Output: an optimal bushy join tree
for all R_i ∈ R {
    BestPlan({R_i}) = R_i;
}
for 1 ≤ i < 2^n − 1 ascending {
    S = {R_j ∈ R|(⌊i/2^j⌋ mod 2) = 1}
    if not (connected S) continue;    // *
    for all S_1 ⊂ S, S_1 ≠ ∅ do {
        ++InnerCounter;
        S_2 = S \ S_1;
        if (S_2 = ∅) continue;
        if not (connected S_1) continue;
        if not (connected S_2) continue;
        if not (S_1 connected to S_2) continue;
        ++CsgCmpPairCounter;
        p_1 = BestPlan(S_1);
        p_2 = BestPlan(S_2);
        CurrPlan = CreateJoinTree(p_1, p_2);
        if (cost(BestPlan(S)) > cost(CurrPlan)) {
            BestPlan(S) = CurrPlan;
        }
    }
}
OnoLohmanCounter = CsgCmpPairCounter / 2;
return BestPlan({R_0, ..., R_{n-1}});
```

*Figure 2.26* Pseudocode de l'algorithme DPsub [ZCHC11]

Le plan optimal est donc obtenu en récupérant la valeur de coût associée à la combinaison optimale de toutes les relations à partir de *dpTable*. La structure de ce plan ne contient donc pas de produit cartésien.

- Conclusion

Dans cette section, nous avons présenté les différentes stratégies de recherche du plan optimal. Ces stratégies diffèrent dans la qualité du résultat et dans la performance d'exécution des algorithmes. Le tableau 2.6 résume les notions que nous avons explorées.

| Catégorie | Algorithme | Technique | Complexité de la requête | Efficacité | Qualité | Limite |
|---|---|---|---|---|---|---|
| Aléatoire | Marco polo | Aléatoire | Simple | + | - | Recherche locale |
| | SA | Restriction de l'espace de recherche en utilisant une heuristique | Complexe | Moyen | Moyen | Recherche locale |
| | II | Aléatoire | | | | Recherche locale |
| Programmation dynamique | Dpsize | Buttom-up | Inner join | + | - | Compatible avec inner join uniquement |
| | Dpsub | Buttom-up | Inner join | + | - | Compatible avec inner join uniquement |
| | Dpcpp | Buttom-up | Inner join | + | - | |
| | Dphyp | Buttom-up | Complexe, inner join | + | - | |
| Déterministe | Exhaustive | DFS,BFS | Simple | - | + | Souffre de l'explosion combinatoire |
| | A* | Heuristique | Complexe | + | - | Heuristique, sélection de solution sous optimale |

*Tableau 2.6* Résumé des algorithmes de recherche de la solution optimale

Dans notre approche, on a choisi la stratégie exhaustive vu qu'elle garantit l'optimalité et que notre problème est un problème de recommandation sans contrainte sur temps de réponse.

#### 2.3.2.4 Modèles de coût

Un modèle de coût est une formule mathématique qui permet d'estimer la performance d'une requête et de l'exprimer en termes de temps de réponse, de nombre de pages ou sous forme d'une valeur abstraite. Un tel modèle est défini dans le seul objectif de comparer la performance d'exécution des plans de l'espace de recherche pour une même requête. Ainsi dans chaque base de données / moteur de traitement, il existe une formule mathématique différente pour chaque opérateur pour estimer son coût d'exécution. On prend l'exemple [Sch19] de l'opérateur de sélection séquentielle. Le modèle de coût de cet opérateur dans la base de données relationnelle PostgreSQL est le suivant : pour une projection de 40 000 enregistrements qui ont une taille sur disque égale à 177 127 424 et en reprenant la configuration par défaut de cette base de données :

*Cost(sequential scan)= seq_page_cost * nb_pages + cpu_tuple_cost * nb_rows+ cpu_operator_cost*nb_rows*

*= seq_page_cost * (relation_size/page_size) + cpu_tuple_cost * nb_rows+ cpu_operator_cost*nb_rows*

*=1 *(177127424 /8192)+0,01*4000000+0,0025*4000000*

*=71622*

L'unité de cette valeur de valeur est une unité arbitraire, elle n'est significative que pour comparer les plans de requêtes SQL évalués dans PostgreSQL. Dans le contexte de l'évaluation de requêtes SQL sur le système MongoDB, il est possible d'utiliser le multi-store Apache Calcite. Ce dernier est capable d'optimiser les requêtes SQL sur des magasins de données non relationnels, comme MongoDB, en utilisant les modèles de coût. Par contre, son modèle de coût est très simple et il ne tient pas compte de l'exécution des opérateurs physiques dans le système de la couche de stockage (MongoDB). Ainsi il propose un modèle de coût pour une exécution de requête en mémoire avec un coût d'entrée sortie égal à zéro et qui se base surtout sur le coût du calcul (CPU). Ce coût d'exécution CPU est égal à $\sum(estimated_rows_i * ne_i)$ sachant que les variables : $estimated_rows_i$ représentent le nombre estimé d'enregistrement à retourner par l'opérateur, et $ne_i$ le nombre d'expressions dans chaque opérateur. En revenant à notre exemple, on trouve que le coût de sélection séquentielle par Apache Calcite de 4 000 000 enregistrements est égal à 4 000 000. D'après cet exemple, on peut conclure que les modèles de coût peuvent être très différents d'un système à un autre. Ils sont la plupart du temps incomparables et n'ont pas été créés dans l'objectif de comparer la performance de l'exécution de requêtes dans différents systèmes hétérogènes.

Pour pouvoir comparer la performance de l'exécution, il est théoriquement possible de créer un modèle de coût générique qui soit capable d'estimer les coûts d'exécution d'un opérateur selon le moteur d'exécution et le système de stockage. Cette solution devra faire face à plusieurs problèmes : (i) le premier inconvénient de cette approche est que les systèmes sous-jacents sont autonomes et qu'à chaque changement de l'API ou du langage de requête de ces systèmes, il faudra adapter le modèle de coût global proposé. Ce qui n'est pas réalisable en pratique si on veut supporter plusieurs systèmes de traitement et de stockage de données. (ii) Il n'est pas possible d'imposer aux systèmes sous-jacents de suivre le contrat suggéré par ce modèle de coût global. (iii) Les modèles de coût peuvent faire des erreurs de précision pour comparer les plans d'exécution dans un même moteur d'évaluation de requête, créer un modèle à plus grande échelle sera moins précis vu les simplifications qu'on devra faire pour limiter la complexité du problème.

En outre, dans notre contexte, les techniques utilisées pour l'évaluation de requêtes sont hétérogènes : en Big Data, les écosystèmes de traitement de données imposent une coexistence de système d'évaluation de requêtes centralisées et distribués. Ces deux catégories de systèmes ont des modèles de coûts qui sont définis d'une manière très différente.

Dans les bases de données centralisées, minimiser le temps revient à diminuer l'utilisation des ressources CPU, d'entrée et/ou sortie de mémoire. Pour cette raison les modules d'optimisation estiment ces différentes métriques, les représentent sous forme de vecteur de valeurs pour, au final, calculer le coût de la requête en utilisant une formule mathématique qui pondère les unités de ce vecteur. Par contre, dans le cadre de l'optimisation de requêtes, il est important d'introduire le coût de communication des données à travers le réseau. Ce coût est un résultat de l'envoi des résultats intermédiaires au site de contrôle pour les combiner ou du coût d'échange des partitions entre les serveurs pour ordonner ou hacher les données, ce qui est une opération indispensable dans plusieurs opérateurs comme le *sort-merge join* ou les *hash-join*.

Estimer les unités du vecteur du modèle de coût est fait en utilisant des formules mathématiques et en exploitant les statistiques générées par le système de données. Dans les écosystèmes Big Data distribués, le modèle de coût doit être généralisé et il doit tenir compte des composantes du vecteur de coût spécifiques aux opérateurs exécutés en mode centralisé, ainsi que du coût de communication réseau. Les catégories d'opérations qu'on peut trouver dans les plans d'évaluation de requêtes distribués et leurs coûts sont schématisés (Figure *2.27*) et expliqués dans la partie suivante [KV13, TLRG08] :

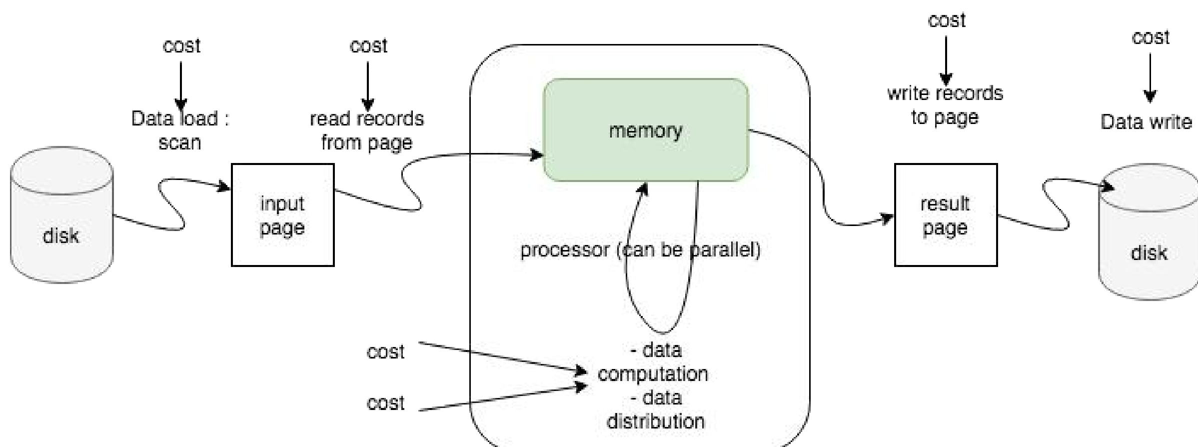



*Figure 2.27* Modèles de coûts dans les systèmes distribués

- *Opérateur de chargement de données* (IO) : cet opérateur permet de lire les données à partir du stockage physique. Son coût correspond au coût de recherche / lecture de données à partir du disque. L'unité de ce coût est généralement le nombre de pages, le nombre d'extraits de disque, le nombre de lignes….
- *Opérateur de stockage* : coût de stockage des résultats intermédiaires importants dans l'exécution de certains opérateurs en fonction de la stratégie du système de données.
- *Opérateur de calcul* : coût de traitement des données en mémoire et d'exécution des agrégations / jointures / filtres nécessaires. Il comprend aussi le coût d'utilisation de la mémoire : nombre de pages en mémoire tampon.
- *Opérateur de communication* : le coût de la distribution des données entre serveurs, de la reconstitution du résultat de la requête et de sa communication au client. Ce coût est très

important dans le contexte d'un workload Big Data, car une simple jointure ou bien une agrégation peut être un frein d'optimisation du workload

| Engine | Cost Vector | Cost Unit | Statistics | Cost attribute semantics |
|---|---|---|---|---|
| Postgres | Start-up cost | 1 page scan | number of pages | Cost unit in Postgres is equal the cost for scan 1 page of relation sequentially. the default value is 1 |
| | Total cost | cost unit | number of tuples | |
| | Number of rows | Row | fraction of null in a column | |
| | Size of a row | Byte | average width of an entry of a column | |
| | | | number of distinct values in a column | |
| | | | most common values | |
| | | | most common frequencies | |
| | | | histogram bounds | |
| | | | Correlation | |
| | | | most common elements | |
| | | | most common elements frequencies | |
| | | | elements count histogram | |
| | | | selectivity | |
| MongoDB with Calcite | Number of rows | Row | number of rows | CPU cost unit is the cpu cost for scanning a row of a table |
| | CPU | usage of CPU resources | number of field in a row | |
| | IO | usage of I/O resources | row size | |
| | | | Selectivity | |
| HBase + Phoenix | CPU | Byte | number of rows | |
| | IO | Byte | size of scanned data | |
| | Memory | Byte | size of a row | |
| | | | parallel level | |
| Neo4j | Cardinality Cost | 1 node scan | Cardinality | Cost unit in Neo4j is the cost for scanning a node by its label |
| | | | Selectivity | |

*Tableau 2.7* Récapitulation des modèles de coûts dans les systèmes Big Data

- *Opérateur d'écriture des données* (IO): coût de l'écriture des données sur disque; il est similaire au coût du chargement de données à partir du disque.

Dans les systèmes distribués, les composants les plus importants sont le coût de calcul et le coût de communication. Dans le tableau 2.7 on compare différents modèles de coût appartenant à des magasins de données / bases de données diverses.

Par exemple, et en ne considérant que les optimiseurs dans le SGBDR, nous avons constaté que les coûts dans Oracle et les coûts dans PostgreSQL sont exprimés à l'aide de différentes unités. En effet, les unités de coût dans PostgreSQL sont généralement arbitraires, mais elles font généralement référence aux chargements en nombre de pages de disque. D'autre part, le coût en oracle est exprimé en nombre de lectures de disques physiques. Pour les modèles de coûts dans les systèmes de données volumineuses (s'ils existent), ils sont représentés dans des vecteurs totalement différents. Par

exemple, le modèle de coût de Calcite est exprimé en un vecteur composé d'un coût IO, d'un coût CPU et d'un nombre de lignes estimées. Quant à Apache Phoenix, le modèle de coût est exprimé à l'aide d'un vecteur composé du coût IO, du coût mémoire et du coût CPU.

Un dernier défi concernant l'optimisation des requêtes dans les systèmes Big Data consiste à intégrer les opérateurs spécifiques pour le parallélisme et la répartition du traitement, à l'instar de l'opérateur *shuffle* dans Spark. L'opérateur *shuffle* est caractérisé par son processus aléatoire d'exécution et donc par l'imprévisibilité de la performance.

D'après l'analyse présentée dans cette section, il est clair que les modèles de coûts ne sont pas suffisants et ne présentent pas une solution optimale pour le problème de comparaison de la performance d'un même workload dans des systèmes hétérogènes. Dans cet objectif, nous proposons l'étude de l'optimisation des requêtes en utilisant les techniques d'apprentissage automatique et d'adapter ces méthodes.

### 2.3.3 Optimisation de requêtes par apprentissage

Les travaux récents en bases de données et en optimisation de requêtes ont prouvé que les modèles mathématiques de coûts sont inefficaces pour estimer la performance des requêtes [WCZ+13]

Étant donné que les modèles de coût visent à estimer le temps de réponse, les systèmes modernes profitent de l'avancée dans le domaine de l'intelligence artificielle pour obtenir des prédictions de performance encore plus précises. Il existe plusieurs travaux qui utilisent les techniques de l'apprentissage pour améliorer différents sous domaine de l'optimisation des requêtes, par exemple :

- Rejoin [MP18] qui utilise l'apprentissage par renforcement profond pour optimiser les requêtes. Il énumère les différentes combinaisons de plans de requêtes en changeant l'ordre des jointures.

- Neo: A Learned Query Optimizer [MNM+19], qui apprend les plans exécutés par les requêtes précédentes pour gagner du temps pour l'optimisation de requêtes similaires.

- DeepQuery: Deep Reinforcement Learning for Query Optimization,[OBGK18] qui estime les cardinalités des requêtes et apprend la meilleure représentation des sous requêtes en utilisant les techniques d'apprentissage profond par renforcement.

La prédiction du temps de réponse dépend des propriétés systèmes de l'expérience (par exemple les propriétés matérielles comme le système d'exploitation utilisé, le type de système, le nombre de cœurs par serveur, de l'espace mémoire disponible, …).

Comme il est difficile de faire varier ces propriétés, nous proposons de :

(i) développer des nœuds de prédiction spécifiques à l'environnement d'exécution du workload,

(ii) et de réapprendre quand on veut déployer notre solution dans un nouvel environnement (il existe aussi des techniques de *Transfer Learning* [WKW16] qui permettent d'adapter les données d'apprentissage dans de nouveaux environnements qui peuvent être exploités). Toutefois, même les systèmes traditionnels utilisent les techniques de calibration pour trouver une approximation du temps de réponse en utilisant le modèle de coût. La méthode de calibration engendre la dépendance du temps de réponse aux propriétés systèmes de l'expérience.

### 2.3.4 Évaluation distribuée des requêtes

Dans la première partie, nous avons expliqué le principe d'évaluation de requête dans une base de données centralisée. Les étapes d'envoi de requête et de reconstruction de résultat dans ce cas sont

simples et ne nécessitent pas de communication réseau. Par contre, dans le cas distribué l'exécution de la requête a généralement trois étapes supplémentaires : la localisation de données (identifier les fragments/réplicas pertinents), l'exécution locale des opérateurs physiques et finalement la reconstitution du résultat à partir des résultats intermédiaires.

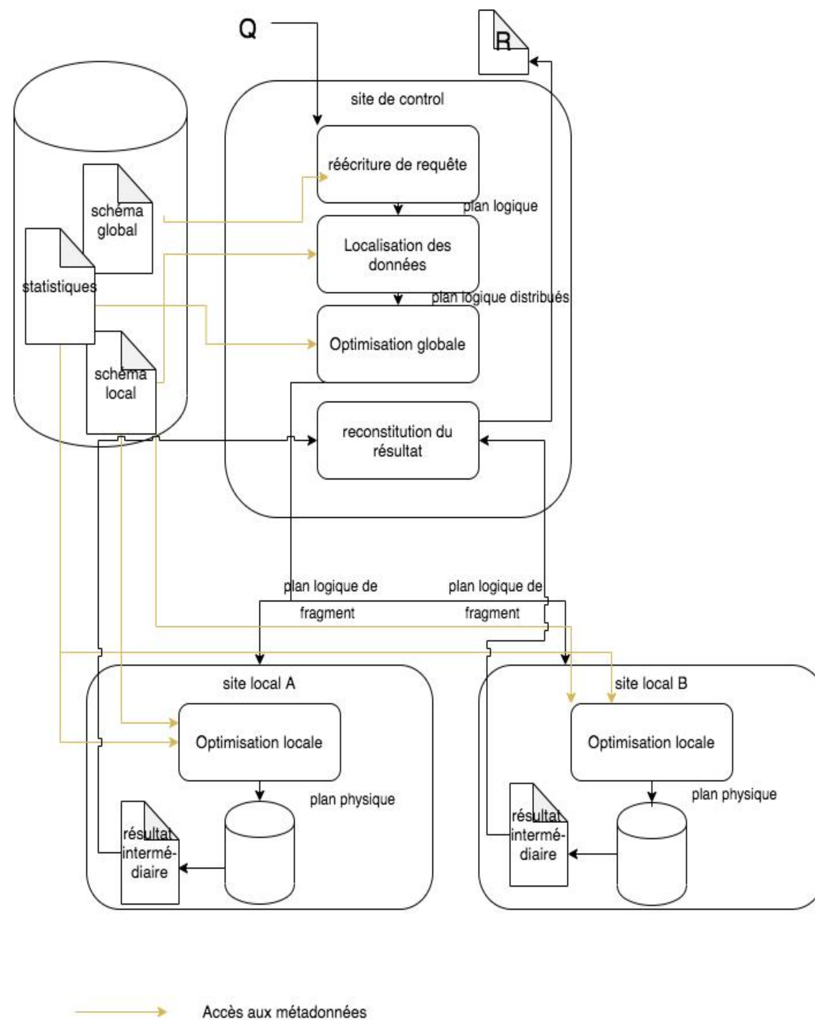



*Figure 2.28* Architecture d'un système d'évaluation de requête distribué

La figure 2.28 illustre le principe de l'évaluation de requêtes distribuées. La première étape est appelée la réécriture de requête, comme en centralisé. La distribution des données est masquée à l'utilisateur final. L'étape de localisation des données vise à traduire une requête exprimée sur une vue globale vers une requête exprimée sur les fragments/réplicats des sites locaux. Elle prend en entrée une requête exprimée par exemple en SQL et génère à partir de cette dernière un plan d'exécution qui tient compte des opérateurs algébriques de parallélisme et de calcul distribué. Ensuite, le site de contrôle détermine l'emplacement des fragments de données à partir des schémas locaux et décompose le plan distribué envoyé par l'étape précédente en plusieurs sous plans qui tiennent compte de l'information sur le site local des fragments [Bre09]. Le système d'évaluation de requêtes peut être enrichi par une fonctionnalité d'intégration de données.

L'étape suivante consiste à trouver l'ordre optimal des opérateurs en utilisant une méthode d'optimisation comme celle décrite dans la section précédente et qui est généralement basée sur le coût. L'objectif de cette étape est de trouver une combinaison d'opérateurs logiques qui minimise la communication réseau. Les plans optimaux déterminés dans cette étape sont envoyés aux sites locaux

et ils subissent une deuxième étape d'optimisation pour trouver le plan physique optimal pour chaque fragment. Finalement les sites participants envoient des résultats partiels et le site de contrôle reconstruit le résultat final. Pour mieux comprendre la différence entre ces types d'exécutions, on considère ces cas de figure : une relation *R* est partitionnée verticalement entre deux serveurs qui sont lancés sur la même machine et une copie de *R* est partitionnée verticalement sur deux serveurs (figure *2.29*).

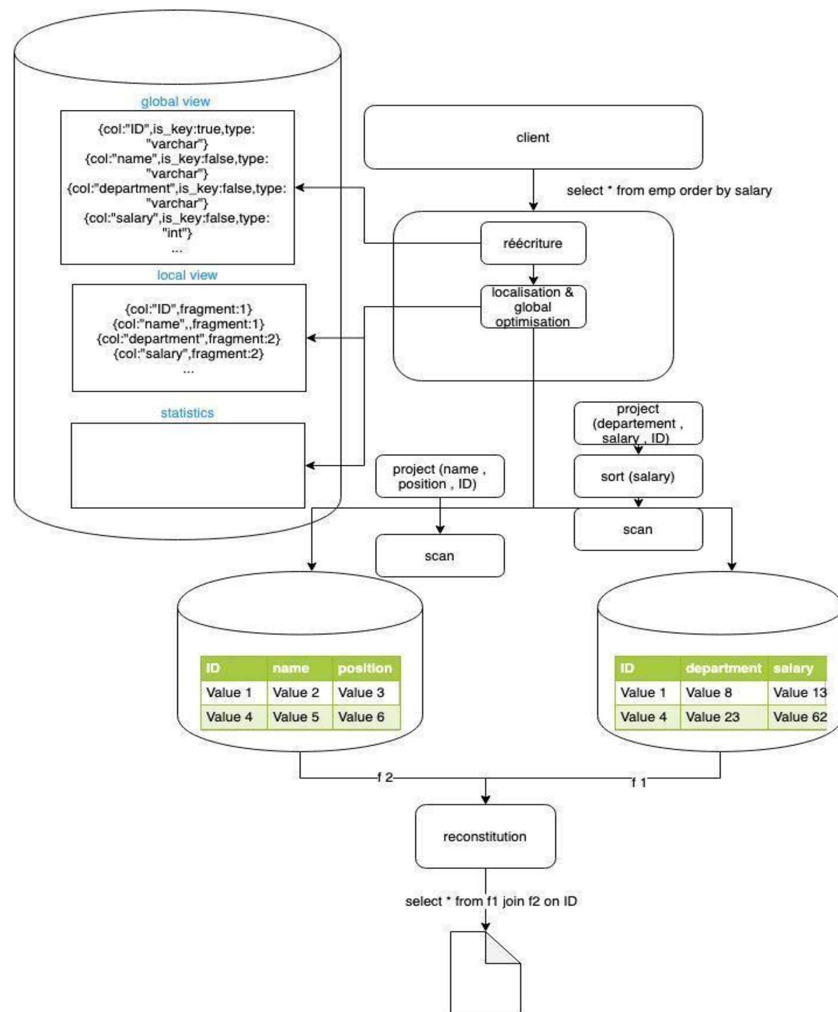



*Figure 2.29* Exemple d'évaluation de requête distribué sur deux machines en utilisant le partitionnement vertical

Dans l'exemple ci-dessous, si on suppose que les machines sont dans un réseau WAN et que le débit est de l'ordre de 10 Mbit par seconde, la transmission de $f_1$ et $f_2$ peut prendre plusieurs secondes dans l'étape de transmission de données comparée à l'étape de calcul. En effet, si un ensemble de données qui a le schéma suivant :

> *ConsoSup36(h_date, cd_prof, sect_act, plag_puis, nb_point_s, tot_nrj_s, cour_moy_1, indc_rep_cour_1, cour_moy_2, indc_rep_cour_2, cour_moy_1_2, indc_rep_cour_1_2, jour_max_mois, sem_max_mois)*

Le premier fragment *vertical conso_sup_*$f_1$ sera composé des colonnes *nb_point_s*, *tot_nrj_s*, *jour_max_mois* et *sem_max_mois* en plus de la clé, et le deuxième fragment *conso_sup_*$f_2$ contiendra les colonnes de la courbe de charge en plus de la clé. Dans cet exemple, le nombre d'enregistrements est égal à 981 100, la taille du premier fragment est égale à 137 MB et celle du deuxième est égale à 145 MB. En considérant un modèle de coût simpliste et si on suppose le temps que prennent une

instruction CPU et une instruction entrée sortie sont 1 ms ; le temps d'exécution de la requête de l'exemple est égale à **2*981 100 *log(981 100) + 2* 981 100 = 29 033 s** (si l'algorithme de jointure de recomposition est *hash join* et en supposant que la taille mémoire dans le serveur de recomposition du résultat est infinie). Le temps de communication est égal à **(145+137)/10 = 28 s** et le temps total de réponse sera donc égal à **29 061 s**. Par contre l'exécution locale ne nécessite pas une jointure et elle aura comme temps de réponse **981 100 *log(981 100) + 981100 = 14 517 s**. Ce résultat montre que l'efficacité de l'exécution sur une seule machine est très supérieure au traitement distribué. Ceci n'est pas principalement dû au débit de communication réseau (surtout dans cet exemple où la taille des données est moyenne) mais plutôt à l'étape de reconstruction du résultat en utilisant une opération de jointure. Le résultat observé dans l'approche centralisée est aussi limité par les ressources matérielles, la taille de la mémoire influence la performance de l'exécution car elle limite la taille de données à lire dans chaque étape.

Par conséquent, on peut conclure que dans les SGBD traditionnels et centralisés la reconstruction le résultat d'une requête est relativement simple comparée à la mise en place de la distribution. En effet, vu que l'échange des données est limité, le coût de communication des données à travers le réseau est inexistant d'où l'efficacité de cette solution par rapport de l'exécution distribuée. En plus, dans ce type de systèmes le modèle de données est homogène ainsi que les solutions de stockage physique des données. Par contre, dans les systèmes NoSQL et les systèmes de fichiers distribués la distribution des données est naturelle, mais l'étape de reconstitution du résultat n'est pas évidente. Le transfert de données à travers le réseau peut être un goulot d'exécution. Dans les applications du monde réel, la gestion de données est généralement faite dans des systèmes différents et autonomes et l'utilisateur peut être dans une situation qui nécessite d'exécuter une requête hybride sur des données gérées indépendamment dans le cadre d'un projet d'intégration de données (distribution applicative). Ainsi les ensembles de données, peuvent être stockés dans des data stores distincts et accès en utilisant des systèmes d'intégration.

## 2.4 Intégration/médiation

L'intégration des données est l'une des phases les plus importantes dans un projet de gestion de données à très large échelle. À partir de sources de données séparées, autonomes et parfois hétérogènes, on souhaite donner à l'utilisateur la possibilité d'interroger les données de ces sources et de les manipuler tout en masquant la complexité du schéma physique des données et leurs éventuelles distributions. Dans cette partie nous présentons brièvement les techniques d'intégration, vu que ce domaine n'est pas le cœur de notre travail.

### 2.4.1 Fédération

L'intégration de données vise à offrir à l'utilisateur les outils nécessaires pour manipuler des données stockées dans des sources indépendantes de façon transparente. L'objectif de ce domaine est d'offrir une vue sur ces données à partir de laquelle l'utilisateur peut envoyer des requêtes d'interrogation ou de mise à jour de données. Une première solution intègre les sources de données en créant un entrepôt de données (*datawarehouse*) et en matérialisant toutes les données utiles dans le système cible. Il est possible aussi de ne pas utiliser une solution d'entrepôt de données et matérialiser tous les données dans un même magasin de données en effectuant les transformations nécessaires pour être conforme au modèle de données utilisé dans le système cible. On perd ainsi les avantages des choix du modèle de données parmi ceux utilisés dans les sources. En effet, le choix de modèles de données peut impacter l'efficacité du traitement. Par exemple, les magasins de données dont le modèle de données est orienté clé-valeur sont caractérisés par une interrogation de données très riche. Par contre, ils possèdent des APIs très simples.

Ce type d'intégration souffre aussi de la redondance de données et des limites techniques liées au transfert des données entre différentes sources, surtout dans le cadre de projets Big Data. Pour éviter ces problèmes, il est possible de garder les données dans leurs sources d'origine et d'utiliser une technique appelée l'intégration virtuelle. Ainsi plusieurs solutions d'architecture ont été proposées pour répondre au besoin d'intégrer virtuellement des sources autonomes (la couche d'intégration ne doit pas impacter les fonctions indépendantes des sources), hétérogènes (ayants différents modèles de données, leurs données ayants différents schémas, par exemple la définition d'un attribut « date de production » comme chaîne de caractères dans une source et date dans une autre, et ils ont plusieurs degrés de structuration (« non structurés », « semi-structurés », « structurés ».) et multiples).

### 2.4.2 Médiation

La médiation est une première architecture qui a été proposée. Cette dernière est composée d'un ou plusieurs médiateurs qui reçoivent les requêtes de l'utilisateur, effectuent les réécritures nécessaires et les envoient aux sources. Comme le montre la figure 2.30, il est possible de définir une interface entre la source et le médiateur appelé le « *wrapper* ». Ce dernier permet de transformer la requête du langage du médiateur vers celui des sources, voire d'effectuer une partie des calculs lui-même si la source a une API très simple.

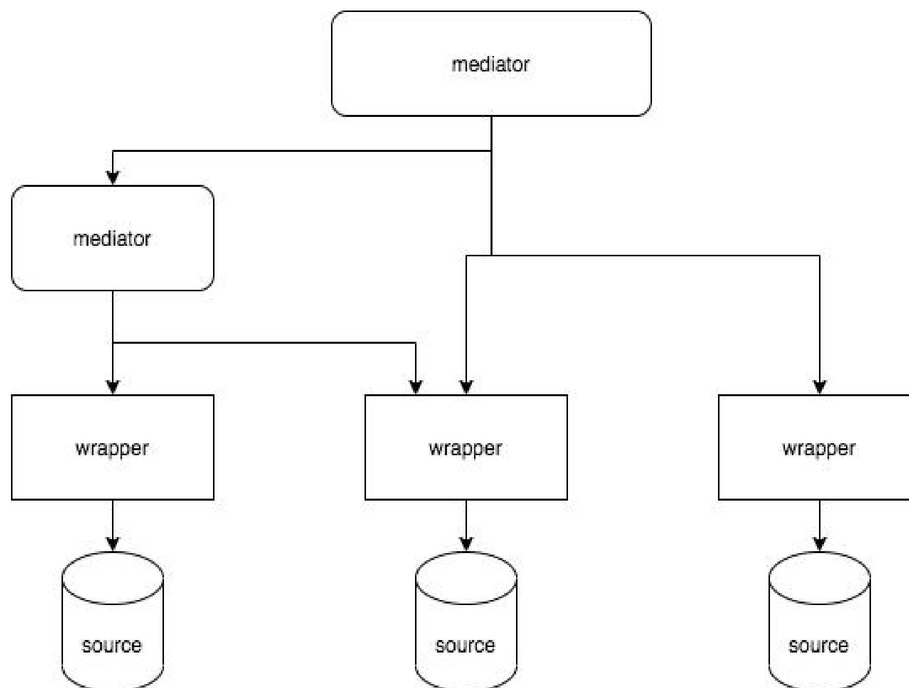



*Figure 2.30* Exemple d'une architecture d'un médiateur [GMPQ+04]

Le médiateur réécrit la requête de l'utilisateur en respectant les schémas hétérogènes des sources en se basant sur un schéma global et des correspondances entre le schéma global et les schémas locaux. Il existe deux techniques pour définir ce schéma :

- Globas-as-view (GAV) : l'approche la plus simple dans laquelle le schéma global est le résultat de l'intégration des schémas locaux. La difficulté principale de cette méthode est la maintenance du schéma global à chaque changement d'un des schémas locaux ou à chaque ajout d'une nouvelle source de données.
- Local-as-view (LAV) où le médiateur construit le schéma global à partir des schémas locaux et les schémas locaux sont vus comme des sous-parties du schéma global. Pour construire la requête, il doit trouver la meilleure combinaison des différents schémas locaux. Le schéma global ne décrit

que les sources mises en jeu dans l'architecture et le système reformule automatiquement la requête selon les informations des sources [XE02]. Cette méthode monte plus à l'échelle, mais est caractérisée par une performance moindre.

Dans le médiateur, le modèle de représentation utilisé est le même dans les médiateurs et les « wrappers ».

### 2.4.3 Polystores

Dans cette section, nous présentons des exemples de systèmes d'intégration de données en Big Data. Dans ce domaine, les solutions existantes ne supportent pas totalement la fonctionnalité d'intégration de données, comme la définition de vues globales. Mais ils offrent des techniques pour l'interrogation hybrides de données et ils supportent les architectures de type polystores / multi-stores.

#### *2.4.3.1 Spark [ME16]*

Apache Spark est un multi-store qui offre une API riche pour le traitement distribué des données à large échelle. Il ne supporte pas uniquement les workloads d'interrogation de données, mais il a aussi une librairie qui permet de développer des applications d'analyse de graphes et une librairie d'apprentissage automatique.

Dans ce travail, on se limite aux librairies de traitement de données au repos et au caractère multi-store de ce système. Ce système supporte le parallélisme hybride. Il supporte le partitionnement des données et permet l'exécution parallèle de certains opérateurs. Le traitement dans Apache Spark est modélisé sous forme d'un graphe orienté acyclique (DAG). L'architecture de Spark est décrite dans la figure ci-dessous et ces composants principaux sont :

- *Resiliant Distributed Dataset* (RDD) : c'est la représentation physique d'un ensemble de données dans Spark. Un RDD est une collection d'enregistrements partitionnée sur la grappe et pour permettre un traitement parallèle. Il est possible d'exécuter les traitements en mémoire ou de stocker cette collection dans le cache pour améliorer l'efficacité, puisque le coût d'accès sur disque (IO) n'est pas pris en compte (réduit à zéro).

- *Dataframe API* : les concepts *Dataset* et *Dataframe* représentent la vue logique des données dans Spark. Ces concepts permettent aux applications de représenter les données sous forme de collections distribuées qui peuvent être interrogées en utilisant le dialecte *Spark SQL* ou transformées en utilisant une API spécifique à ces modèles de données.

- Sources de données : Spark est compatible avec toutes les sources de données supportées par le framework Hadoop et plus spécifiquement avec la librairie *InputFormat [VKKS17]* de Hadoop MapReduce. Des plugins peuvent aussi permettre à Spark d'accéder à des sources de données différentes de l'écosystème Hadoop, comme le magasin de documents MongoDB.

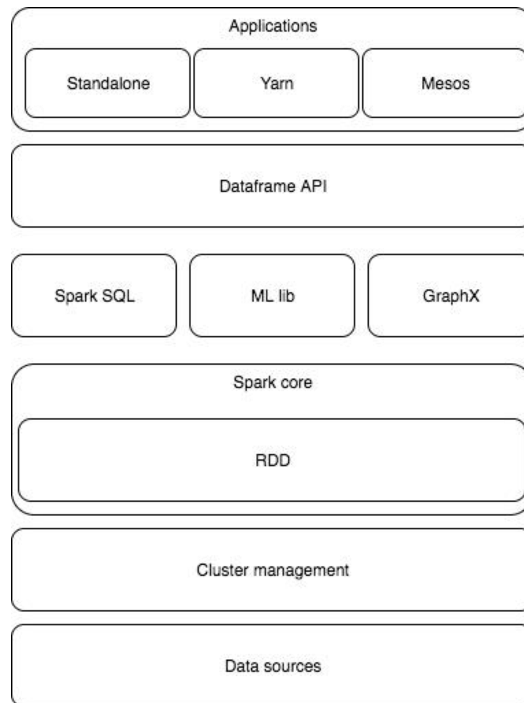



*Figure 2.31* Architecture haut niveau d'Apache Spark

### ○ Modèle de traitement

Le moteur de traitement Spark a une architecture maître-esclave. Une application envoie un workload au nœud maître sous forme d'un 'job', ce dernier l'envoie au gestionnaire de la grappe (en anglais *cluster manager*) qui se charge de distribuer le traitement entre les serveurs esclaves (appelés *worker servers* dans la terminologie Spark).

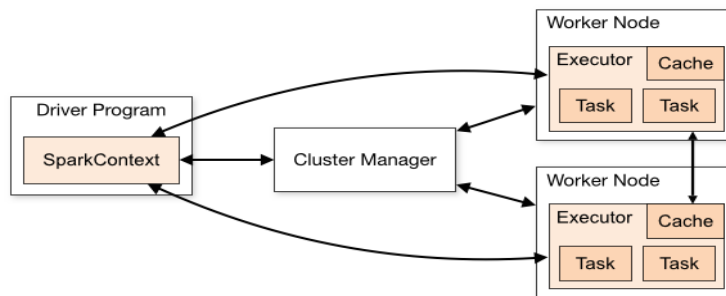



*Figure 2.32* Architecture distribuée d'une application Spark [Spa]

Les jobs dans Spark sont structurés sous forme d'un DAG de taches. Les liens entre ces taches représentent des dépendances entre ces derniers ; pour deux taches A et B mises en relation, la tache B ne peux être exécutée que si la tache A a été exécutée. Ces taches sont des unités de calculs physiques qui correspondent à des transformations des opérateurs sur les RDD définies dans les applications de plus haut niveau. Ces taches sont aussi groupées sous forme d'étapes (en anglais *stages*) qui seront exécutées en parallèle. L'intérêt de planifier l'exécution en utilisant comme structure de données les DAG est : (i) la tolérance aux fautes puisque le graphe d'exécution permet de détecter la tâche qui a échoué et de la signaler au gestionnaire des ressources ; (ii) la récupération des données en cas de panne puisque l'exécution est tracée et le système peux déterminer quelles sont les données intermédiaires qui pourront être réutilisés dans le calcul ; et finalement (iii) avec cette stratégie de planification des taches, l'exécution devient plus efficace [Spa] vu qu'il est possible de réutiliser des étapes du calcul.

### Discussion sur la performance de Spark

L'intérêt de l'utilisation de Spark par rapport à MapReduce pour traiter les données massives est l'amélioration du traitement vérifiée par ce système. Cette amélioration est due à la technique d'écriture des données intermédiaires nécessaires pour le traitement en mémoire (dans le cache) au lieu de l'écriture sur disque, l'utilisation de plusieurs formes de parallélisme, comme la technique du multithreading et l'organisation des tâches en plusieurs étages (staging), et finalement le modèle d'exécution en utilisant la structure de données DAG. L'inconvénient principal de ce système est l'utilisation excessive de la mémoire, qui est limitée.

### Spark SQL

L'API Spark SQL ne suit aucune version exacte des standards du langage SQL, mais suit partiellement la version 2011.

Dans le tableau 2.8, on résume les opérateurs supportés par Spark SQL. On présente pour chaque version de l'API de Spark SQL les opérateurs physiques qui représentent l'implémentation des opérateurs de l'algèbre du langage et les arguments de chaque opérateur, par exemple les fonctions d'agrégation et les types spécifiques d'argument. On souligne aussi les expressions (du standard SQL 2011) qui ne sont pas supportées dans les différentes versions de cette API et on présente des exemples de l'utilisation des expressions de Spark SQL.

| **Fonctionnalité SQL** | **Expression SQL 2011** | **Arguments** | **Fonction** | **v[1]1.6** | **v 2.0** | **v 2.2** | **Remarque** | **Exemple** |
|---|---|---|---|---|---|---|---|---|
| Sélection | Select where « argument » | Egalité (*EqualTo*, *EqualNullSafe*), Inégalité (GreaterThan, GreaterThanOrEqual, LesserThanOrEqual, LesserThan), conjonction(*And*), disjonction (*Or*) , In , between , IsNull,IsNotNull,Not, StringContains, StringEndsWith, StringStartsWith, LIKE | _ | X | X | X | _ | select * from `profile_fam_csv` where cd_prof between "PRO2" AND "PRO7" |
| Sélection | Select Exists(« argument ») | Fonction de map, udf | Udf | X | X | X | _ | select * from `profile_fam_csv` where cd_prof between "PRO2" AND "PRO7" |
| Projection | Select « argument » from | Liste d'attributs, *, [], opérateur point (*dot* .) pour accéder aux valeurs structurés | _ | X | X | X | | select compounda.attribute from test_profiles |
| | Select « function » from | _ | fonctions de tableaux, explode , getItem , etc | x | X | x | _ | select sort_array(profiles) from test_profiles |
| Agrégation | Select « function » from | _ | Sum ,avg , min , max | X | X | x | _ | |

[1] V : version

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | Select « function » from group by | _ | Sum ,avg , min , max ,etc | x | X | x | _ | |
| | Select « function » from group by having | _ | Sum ,avg , min , max, etc | X | X | x | _ | |
| Jointure | Select « arguments de projection » from « type de jointure » join | Type de jointure = inner , left (outer) , right (outer), full outer , cross, left anti , left semi | _ | x | X | X | Naturel , equi ou theta | |
| | Select « arguments de projection » from where « argument » | _ | _ | X | X | x | _ | |
| | Select « arguments de projection » from where in « sous requête» | _ | _ | x | X | x | _ | |
| Tri | Select « arguments de projection » from order by “type de l’ordre” | type de l’ordre= asc , dsc | _ | x | X | X | _ | |
| Elimination des duplicat | Select distinct « arguments de projection » from | _ | _ | X | X | x | _ | |
| Union | Select « arguments de projection » from union | _ | _ | x | X | x | _ | |
| Produit cartésien | Select « arguments de projection » from cross join | _ | _ | X | X | _ | | |

*Tableau 2.8* Comparaison des différentes versions des APIs de Spark SQL

- RDD API

Au niveau physique, les données sont représentées sous forme d’une collection immuable et distribuée d’objets appelée des RDDs [RDD]. L’API associée à cette représentation de données permet aux utilisateurs d’exécuter des traitements parallèles en utilisant des opérateurs de type *action* ou *transformation*. Ce modèle de données est aussi utilisé dans le moteur d’exécution pour traiter les opérateurs physiques envoyés par le module d’optimisation qui est utilisé par les modules de plus haut niveau : *Spark SQL* et *Dataframes API*. À l’exécution, les RDDs sont distribués entre nœuds esclaves de la grappe de calcul. Ensuite en parallèle, les opérateurs physiques qui sont dans cette étape se traduisent en un ensemble de tâches qui seront finalement exécutées sur chaque partition du RDD. Le degré de parallélisme est configurable dans l’application, il est aussi possible de spécifier le nombre de réducteurs pour l’opérateur *Shuffle* qui redistribue les données entre les nœuds à chaque opération d’agrégation ou de croisement de données et qui a une valeur par défaut égale à 200.

Le tableau 2.9 résume les opérateurs de l’API des RDD. Cette API est inspirée du modèle de programmation fonctionnel à base des fonctions *map* et *reduce* qui est adopté au domaine par Hadoop. Mais, dans ce système de traitement, l’API d’accès physique aux données est plus riche.

| Fonction | Arguments | Type | T / A[2] | Résultat | Avant v1.3 | v[3] 1.3 | v 1.6 | v 2.2 | exemple[4] |
|---|---|---|---|---|---|---|---|---|---|
| map | Fonction | projection | T | RDD | _ | X | X | X | |
| select | Ensemble de colonnes | projection | T | DataFrame | _ | X | X | X | |
| selectExpr | Ensemble de chaines de caractères | projection | T | DataFrame | _ | X | X | X | |
| filter | Condition | Filtrage | T | DataFrame | _ | X | X | X | |
| where | Condition | Filtrage | T | DataFrame | _ | X | X | X | |
| sort | Ensemble de colonnes | Tri | T | DataFrame | _ | X | X | X | |
| sort | Ensemble d'expressions sur les colonnes | Tri | T | DataFrame | _ | X | X | X | |
| orderby | Ensemble de colonnes | Tri | T | DataFrame | _ | X | X | X | |
| orderby | Ensemble d'expressions sur les colonnes | Tri | T | DataFrame | _ | X | X | X | |
| groupBy | Ensemble de colonnes | agrégation | T | GroupedData | _ | X | X | X | _ |
| agg | Ensemble d'expressions sur les colonnes | agrégation | T | valeur | _ | X | X | X | df.groupBy("_c1").agg(max("_c2"), sum("_c3")) |
| rollup | Ensemble de colonnes | agrégation | T | GroupedData | _ | _ | X | X | _ |

*Tableau 2.9* Exemple de fonctions de l'API RDD de Spark

### ○ Graphx

La librairie Graphx de Spark permet à ce moteur de traitement de supporter les applications ayant un autre modèle de données : le modèle basé sur les graphes. Cet outil a été inspiré de l'article de recherche de Google inc. qui traite le même problème et qui propose un algorithme intitulé *Pregel [Spa]*. Contrairement aux solutions NoSQL ayant un modèle de données orienté graphe comme Neo4J, GraphX ne permet pas les requêtes de type mise à jour, mais il propose plusieurs algorithmes d'interrogation de graphes de données.

Dans cette librairie, il est possible de distribuer les données et les traitements en utilisant les techniques de partitionnement et de parallélisme. En effet, il est possible de partitionner horizontalement les nœuds (*vertices*) entre plusieurs serveurs. Mais ceci n'est pas la solution optimale vu que des nœuds qui peuvent avoir des liens entre eux peuvent être stockés dans des serveurs différents ce qui rend le parcours des liens dans le graphe très couteux.

---

[2] T : Transformation , A : Action
[3] v : version
[4] Les exemples sont faits sur des fichiers générés par le benchmark tpc-ds

Une deuxième méthode est plus adaptée aux techniques de stockage des graphes en mode distribué. Elle divise le graphe selon les liens (*edges*). Ce type de partitionnement partage les nœuds de telle sorte que le nombre de liens entre les groupes de serveurs (les partitions) sont minimisés, ce qui permet l'optimisation du coût de la communication réseau dans le cluster.

Le problème dans ce type de partitionnement est de trouver la coupe optimale pour préserver cette stratégie d'optimisation, comme le montre la figure 2.33.

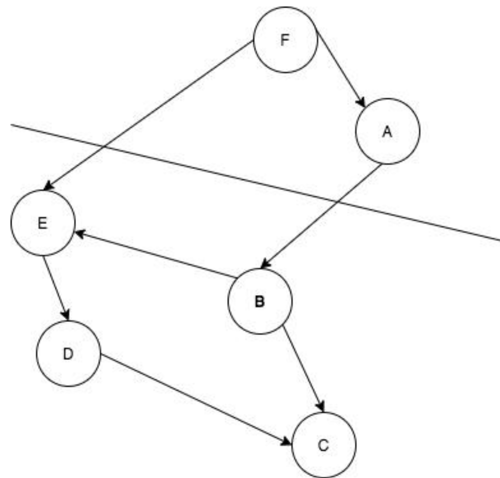



*Figure 2.33* Exemple de coupe d'un graph

Dans cette librairie, deux types de solutions ont été adoptées [Gra] : la recherche optimale de coupe de bords (*edge-cut)* ou celle de la coupe de nœuds. La première catégorie coupe les liens du graphe, répartit les partitions entre les serveurs et réplique le lien coupé dans les partitions concernées. Le premier algorithme est intitulé : *EdgePartition1D* et permet de regrouper les liens qui appartiennent à la même source dans le même serveur [AXF18].

*EdgePartition2D* est un second algorithme du même type qui utilise une matrice d'adjacence et les deux sommets d'un lien pour calculer les partitions. Cet algorithme utilise une méthode de partitionnement hybride qui calcule les partitions du graphe en utilisant une fonction de hachage sur les colonnes et les lignes de la matrice d'adjacence.

La deuxième catégorie d'algorithmes, partitionne le graphe en effectuant une coupe au niveau d'un nœud et la réplication de ce dernier dans les deux partitions associées. Dans cette catégorie on trouve les algorithmes *CanonicalRandomVertexCut* [Gra] et *RandomVertexCut* [Gra] qui attribuent les nœuds aux partitions en hachant les ID des deux sommets, ce qui entraîne une coupure de sommets aléatoire qui Co-localise toutes les arêtes entre ces derniers. La différence entre ces deux algorithmes est que le premier dépend de la direction des liens.

L'API de Spark GraphX supporte un ensemble d'opérateurs spécifiques qui implémentent des algorithmes de traitement et de parcours de graphe. Le tableau 2.10 résume ces opérateurs ainsi que des exemples de leurs utilisations.

| Opérateur de graphe | Arguments | Type | T / A | Résultat | v1.3 | v1.6 | v2.2 | Remarque et limites | exemple[5] |
|---|---|---|---|---|---|---|---|---|---|
| numEdges | _ | projection | T | Long | X | X | X | | |
| select | Ensemble de colonnes | projection | T | DataFrame | X | X | X | | |
| selectExpr | Ensemble de chaines de caractères | projection | T | DataFrame | X | X | X | | |
| Filter | Condition | Filtrage | T | DataFrame | X | X | X | | |
| Where | Condition | Filtrage | T | DataFrame | X | X | X | | |
| Sort | Ensemble de colonnes | Tri | T | DataFrame | X | X | X | | |
| Sort | Ensemble d'expressions sur les colonnes | Tri | T | DataFrame | X | X | X | | |
| Orderby | Ensemble de colonnes | Tri | T | DataFrame | X | X | X | | |
| orderby | Ensemble d'expressions sur les colonnes | Tri | T | DataFrame | X | X | X | | |
| groupBy | Ensemble de colonnes | Agrégation | T | GroupedData | X | X | X | | _ |
| Agg | Ensemble d'expressions sur les colonnes | Agrégation | T | Valeur | X | X | X | L'argument | df.groupBy("_c1").agg( max("_c2"), sum("_c3")) |
| Rollup | Ensemble de colonnes | Agrégation | T | GroupedData | _ | X | X | Depuis 1.4 | _ |

*Tableau 2.10* Exemples d'opérateurs de l'API Graphx de Spark

### Optimisation

Apache Spark s'inspire de l'avancée en recherche dans le domaine de base de données pour son moteur d'optimisation. Dans toutes ses APIs d'interrogation de données, il permet l'optimisation de l'exécution en se basant sur des heuristiques exprimées sous forme de règles et des coûts calculés à partir de formules mathématiques qui estiment la performance des requêtes/des opérations sur les données en fonction de la consommation des ressources matérielles (CPU, IO, Communication, …). L'optimisation basée sur les coûts a été intégrée dans le projet *Catalyst* depuis sa version 2.2. Le projet qui le précède : projet *Tungsten* était basé uniquement sur des heuristiques et, comme on a vu précédemment, l'optimisation basée sur les heuristiques est très limitée et ne permet pas de trouver la solution optimale dans la majorité des cas.

En utilisant Catalyst, Spark permet de collecter des statistiques sur les sources de données et d'utiliser ces statistiques pour estimer les coûts des requêtes. Ce modèle de coût est très simple. Ce dernier se contente de calculer les cardinalités des opérateurs. Dans l'optimisation des requêtes de jointure, l'un des plus importants problèmes dans le domaine d'interrogation de données, il sélectionne le plan optimal en comparant la cardinalité des sous-arbres droits et gauches qui correspondent à la taille la plus petite.

### Conclusion

Les caractéristiques des RDDs, et principalement leurs immutabilités, l'exécution paresseuse (*lazy*) des workloads lancés sur ces derniers, l'optimisation du traitement en utilisant une exécution structurée comme en flux de donnée (*data flow*) et le stockage des données en mémoire contribuent à la haute performance des applications Spark. Mais l'aspect de distribution massive et aléatoire des données en utilisant par exemple l'opérateur *Shuffle* est un frein à l'efficacité dans Spark. Comme tous les systèmes

[5] Les exemples sont faits sur des fichiers générés par le benchmark tpc-ds

de traitement parallèle et distribué, l'étape de communication des données dans le réseau domine le temps de réponse. La performance de cette étape dépend de l'état du réseau et il peut engendrer des délais très variés selon le type de matériel utilisé dans l'architecture et du nombre de taches concurrentes. De plus, l'organisation des opérateurs du workload permet l'efficacité du traitement en utilisant un moteur d'optimisation compatible avec toutes les APIs de Spark.

#### *2.4.3.2 Apache calcite*

Apache Calcite est un système de gestion de données dynamiques qui permet d'interroger les données Big Data en utilisant le langage SQL (standard 1992), de transformer les plans de requête en utilisant des heuristiques et de les optimiser en utilisant une approche basée sur le coût. Pourtant, il ne permet ni le stockage de données (c'est un médiateur), ni le stockage de métadonnées. Il est principalement utilisé en Big Data comme module d'optimisation.

Utilisé dans un large nombre de systèmes d'interrogation de donnée massifs Apache comme Hive [SGMH18], Phoenix [AM17] et dans des systèmes d'intégration comme Drill [HN13]

L'architecture de ce système respecte l'architecture de référence pour les moteurs d'évaluation de requêtes SQL. Comme le montre la figure 2.34, elle est composée d'un module d'analyse syntaxique et un module d'optimisation de requête.

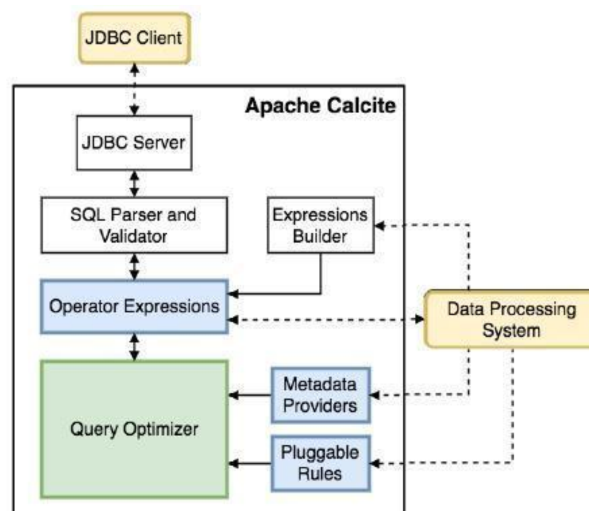



*Figure 2.34* Architecture d'Apache calcite [BCRH+18]

Le modèle de données d'Apache calcite est le modèle relationnel. En utilisant le module d'analyse syntaxique qui supporte le standard SQL92, étendue par les concepteurs d'Apache Calcite pour supporter une variété de modèles de données.

#### *2.4.3.3 MuSQLe [GPTK16]*

MuSQLe est un Médiateur construit au-dessus de Spark. C'est un système homogène qui permet l'intégration des données dans des magasins de données qui ne supportent que le langage SQL. Son API et son modèle d'interrogation de données hérite de spark, ainsi il supporte le traitement massivement parallèle MPP et la version 2011 du Standard SQL. L'innovation de ce projet consiste à proposer un modèle de coût pour Apache Spark qui est un sujet de recherche d'actualité. La modélisation du coût des requêtes dans Spark n'est pas encore résolu car modéliser l'exécution distribuée est un problème difficile du fait de l'utilisation de l'opérateur *shuffle* pour l'exécution parallèle. Cet opérateur a un comportement aléatoire. Il est difficile de modéliser mathématiquement son coût. Pour cette raison MuSQLe fait une simplification qui limite beaucoup son exploitation dans des scénarii réels : Les auteurs proposent un modèle de coût destiné à un déploiement centralisé de

Spark. Or l'intérêt principal de Spark comme tout système Big Data est le calcul distribué. D'où l'inconvénient principal de cette solution.

Le module d'optimisation de MuSQLe trouve le plan logique optimal et découplé l'optimisation physique de l'optimisation logique. Il s'occupe de l'optimisation logique du plan hybride en inversant l'ordre des jointures du plan optimal retourné par l'optimiseur de Spark (Apache Catalyst) et il délègue l'optimisation physique aux systèmes de la couche inférieure. Il calcule le coût des requêtes hybrides en réutilisant les coûts retournés par les systèmes de la couche stockage.

Dans ce système, on est face aux problèmes de comparaison de coût de requêtes hétérogènes. Les coûts des systèmes supportés par MuSQLe n'expriment pas le coût en utilisant les mêmes unités. MuSQLe utilise une approche basée sur l'apprentissage automatique pour transformer le coût retourné par PostgreSQL et Spark pour les exprimer sous forme de temps de réponse. Les auteurs indiquent que les coûts varient linéairement avec le temps de réponse.

#### 2.4.3.4 BigDawg

BigDawg est un système d'intégration de données massives qui a la particularité de proposer trois apis spécifiques associés à 3 différents modèles de données : l'api BDrel pour accéder des magasins de données ayant le modèle relationnel, l'api bdarray pour les magasins de données ayant un modèle de données orienté tableau et l'api bdtext pour les magasins de données ayant un modèle de données orienté clé-valeur. Ces API's accèdent aux magasins de données à travers leurs propres moteurs de requête. Son architecture est composée de 4 composants principaux (comme le montre la figure 2.35): un module de planification de requête, un module d'exécution de requête, un module de supervision de performance et un module de migration.

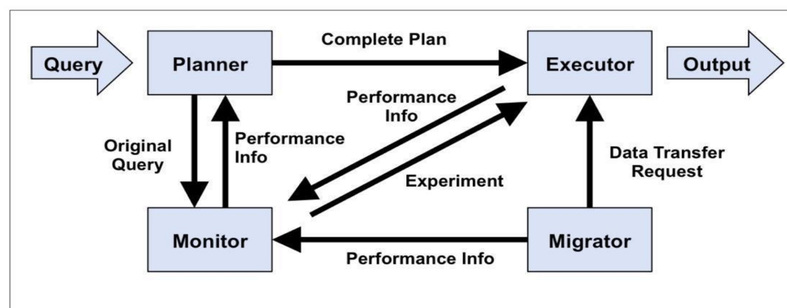



*Figure 2.35* Architecture du système d'intégration BigDAWG

Le module de planification transforme la requête vers un ensemble de plans logiques qui contiennent une information sur le magasin de données qui va exécuter le plan. Il envoie ensuite la requête au superviseur pour récupérer une estimation du temps de réponse de cette dernière. Pour son estimation ce dernier utilise une méthode de prédiction par étude de similarité. Il fait l'hypothèse que pour deux requêtes exécutées sur le même système ayant des plans de requêtes similaires ont le même temps de réponse. Cette mesure de similarité est faite en utilisant une fonction de hachage qui transforme la requête et les informations sur le magasin de données qui va faire l'exécution ainsi qu'une fonction de distance qui compare les valeurs hachées obtenus. Dans le cas où le module de planification ne trouve pas une requête similaire à la requête en entrée il exécute la requête et met à jour le module de supervision. Le module de migration sert à déplacer les données en utilisant les informations encodées dans le plan généré par le module de planification.

#### 2.4.3.5 Synthèse

Dans cette section, nous avons présenté l'intégration et la médiation et nous avons détaillé des exemples proposés dans la littérature pour les systèmes de d'intégrations de données en Big Data. Le

problème principal dans l'optimisation des requêtes dans ces types de système est l'estimation du coût d'exécution des requêtes. On peut passer des années à construire un modèle de coût générique mais les systèmes sous-jacents peuvent faire évoluer leurs modèles de coûts, leurs opérateurs ou leurs modèles de données. L'apprentissage automatique est plus rapide et plus efficace (*accuracy*). Le coût de l'exécution des requêtes dans chaque système est exprimé en utilisant des unités différentes. Par exemple, dans Oracle l'unité de calcul est le nombre d'accès disque, par contre dans PostgreSQL l'unité est plutôt le nombre de pages manipulées. Pour cette raison, il faut trouver une unité commune pour les modèles de coûts dans les systèmes d'intégration hétérogènes et nous avons décidé de travailler avec de temps d'exécution (seconde) qui exprime mieux l'efficacité sous forme de latence. En termes de conclusion, nous présentons un tableau de synthèse des différentes solutions qui existent pour résoudre ce problème.

| **System** | **Evaluation of workload performance** | **workload execution** | **Execution engine** | **Cost models** | **Comparing costs** |
|---|---|---|---|---|---|
| Bigdawg | Learning on execution stats | Logical plan (DAG) .. | Bigdawg's | None | Execution time comparing of queries with similar signatures. |
| MuSQLe | Cost models | Sql partial queries to be executed in sparksql | SparkSQL | Yes (+reuse) | Yes (calibrating queries) |
| Miso | Materialized view | Logical Plan splits .. | Moves partial plans to the stores execution engine | Reuse | Yes (calibrating queries) |
| Apache calcite | Cbo | Logical plan (AST) | Calcite | Yes | No |

*Tableau 2.11* Comparaison des systèmes d'intégration en Big Data

Le tableau 2.11 montre que les techniques d'estimation de la performance et la comparaison de ces derniers sont variées dans les systèmes d'intégration de données massifs. Certains utilisent des techniques d'apprentissage automatiques pour prédire le temps d'exécution d'une requête et d'autres utilisent des techniques de calibration sur les coûts estimés par les moteurs d'exécution de requête de la couche de stockage. La deuxième solution est donc suffisante pour des architectures qui n'utilisent que des moteurs d'exécution qui possèdent des modèles de coût dans leurs modules d'optimisation. Par contre, la solution basée sur l'apprentissage est plus complète dans le cadre d'un écosystème Big Data composé de systèmes hétérogènes qui ne sont pas capables généralement d'estimer le coût d'exécution des requêtes en utilisant des modèles de coût.

## 2.5 Discussion et conclusions

Dans ce chapitre, nous avons présenté le domaine du Big Data. Nous avons présenté une définition générale du domaine, un exemple d'application de ce domaine et une architecture générale qui permet de décliner les systèmes Big Data, notamment leurs modèles de données, leurs techniques de stockage et de distribution, ainsi que leurs moteurs d'interrogation de données.

Nous avons montré que ces systèmes ont des modèles de données et de traitements qui sont très variés et leur utilisation est spécifique à un cas d'utilisation particulier. Par exemple, les systèmes de modèles de données clé-valeurs ont été créés pour un accès très rapide aux données, mais ils proposent des APIs limitées. L'intérêt principal de cette hétérogénéité est l'adaptation des systèmes aux données et aux workloads pour améliorer la performance du traitement à très grande échelle.

Une catégorie de systèmes qui utilisent cette hétérogénéité et adresse leurs défis sont les systèmes d'intégration de données massifs. Ces derniers interrogent des données stockées et gérées dans des magasins de données autonomes et assurent l'exécution optimisée de ces workloads.

En Big Data, l'optimisation de l'exécution des traitements est un défi important vu la variété des systèmes utilisés. Deux techniques principales sont adaptées dans ces systèmes pour atteindre cet objectif. La première consiste à concevoir un algorithme d'optimisation dans le module d'exécution des traitements, qui sont généralement des requêtes déclaratives. Cette technique a été inventée avec les bases de données relationnelles et elle est utilisée dans quelques systèmes NoSQL, comme Neo4J. La deuxième technique délègue l'optimisation à l'utilisateur. Ce dernier, en utilisant une API doit connaître la meilleure solution pour exécuter ces requêtes. Cette solution est utilisée dans plusieurs solutions NoSQL, comme Apache HBase et Apache MapReduce. Elle demande un niveau d'expertise élevé, ce qui rend complexe le développement de workloads efficaces.

Les systèmes de la première catégorie utilisent trois stratégies distinctes afin d'optimiser l'exécution du workload. La première stratégie est la plus simple et elle consiste à développer des heuristiques pour améliorer les performances. Les travaux existants ont montré l'insuffisance de cette méthode puisqu'elle sélectionne des plans sous-optimaux. Une alternative à cette stratégie permet au système de sélectionner un plan à exécuter en minimisant une fonction de coût. Cette deuxième stratégie a comme objectif de comparer des plans générés à partir d'une seule requête dans un seul système. Elle est efficace pour résoudre le problème d'optimisation dans un système. Dans les systèmes d'intégration, cette solution n'est pas adaptée pour comparer la performance du workload entre différents systèmes. En effet, ces systèmes utilisent des modèles de coût ayant des dimensions différentes et ne sont donc pas comparables. De plus, il est extrêmement difficile de générer un modèle de coût universel, du fait de l'évolution indépendante et très rapide de ces systèmes. La troisième stratégie utilise des techniques d'apprentissage automatique pour estimer un coût d'exécution unifié pour les différents systèmes de l'écosystème. Cette solution est la plus adaptée pour le problème de sélection du plan optimal dans un écosystème Big Data hétérogène. Par contre l'un des défis majeurs reste à déterminer les caractéristiques sur lesquelles se basent la prédiction et à construire l'architecture à utiliser pour résoudre ce problème.

Dans ce travail, nous nous intéressons à la problématique de la recommandation de la meilleure combinaison de systèmes - de stockage de données et d'exécution – pour les workloads (/applications) utilisateurs. Il s'agit d'un problème d'optimisation dont l'objectif principal est de minimiser le temps d'exécution des workloads. Dans notre solution on s'inspire des travaux existants dans le domaine de l'optimisation des requêtes dans les systèmes d'intégration, qui est un sujet de recherche d'actualité. Il existe par exemple des systèmes multi-bases qui envoient des sous-requêtes aux systèmes sous-jacents et utilisent un modèle de coût pour comparer les plans qui manipulent des résultats intermédiaires obtenus dans le moteur d'exécution du système d'intégration. Ils ne tiennent pas compte dans leurs calculs du coût de l'exécution dans les sources. Ils ne répondent donc pas exactement à notre besoin. Des exemples de ces systèmes sont Apache Spark et Apache Calcite. Quant aux systèmes de médiations, comme MuSQLe, ils ajoutent la fonctionnalité de sélection des sources et tiennent compte dans leurs modèles de coût de l'exécution dans les sources. MuSQLe permet de transformer les coûts récupérés des sources en utilisant l'apprentissage automatique. Un des inconvénients de MuSQLe est que son API est homogène : il ne supporte que les systèmes relationnels qui peuvent estimer un coût d'exécution à partir de statistiques injectées. De plus, son modèle d'apprentissage ne tient compte que du coût des sous-requêtes, ce qui peut être une information insuffisante. Finalement ce système utilise la technique d'optimisation basée sur la programmation dynamique qui ne garantit pas d'obtenir l'optimum global. Une telle approche est intéressante si le

temps d'optimisation est limité, ce qui n'est pas le cas dans notre contexte. Ces systèmes d'intégration/médiation en Big Data sont limités par rapport à notre besoin. Nous proposons d'utiliser les bonnes idées proposées par ces systèmes et les enrichir en : (i) utilisant l'apprentissage automatique pour prédire la performance des workloads à partir de métadonnées riches et variées, incluant une estimation de coût si possible ; (ii) en gérant l'hétérogénéité des systèmes de gestion de données sous-jacents par la prise en compte des transformations de données entre les différents modèles (structures de données et schéma physique) ; et (iii), utiliser une technique de recherche exhaustive de la solution optimale, ce qui est possible vu que la recommandation de placement des données est effectuée hors-ligne.

Le chapitre suivant présente le modèle de métadonnées décrivant les jeux de données, les workloads et les systèmes, qui seront exploités pour prédire les coûts d'exécution des différents plans de requête. Le chapitre 4 présentera l'architecture et le fonctionnement de notre solution de recommandation de placement DWS (pour *Data, Workloads and Systems*). Le Chapitre 5 présentera notre prototype et les expérimentations.

# Chapitre 3 Méta-modèle d'un écosystème de gestion de Big Data

Disposer de métadonnées permet dans un écosystème basé sur les lacs de données de mieux maîtriser la complexité de l'architecture due aux volumes importants de données, la variété des ensembles de données (brutes et raffinées, courantes et historiques, données de mesure compteurs et données sociales, etc.), leurs provenances, et la variété des systèmes de gestion et de traitement de données utilisés (modèle de configuration, choix de partitionnement de données, monitoring, métriques, etc.) La démarche choisie a consisté à identifier les besoins en termes de représentation et traitement de métadonnées au sein des fonctionnalités ou services pour des ensembles de données [ZCM20]. Nous avons choisi le standard UML (Unified Modeling Language) pour la spécification et la modélisation des métadonnées. Le résultat de cette étape de modélisation est un schéma de métadonnées modulaire et extensible. Il est utilisé dans différents services que nous détaillons dans les chapitres suivants.

Les métadonnées du niveau « Data » décrivent les structures des (ensembles de) données stockées dans le lac, les annotations sémantiques ainsi que les statistiques descriptives qui caractérisent ces données. Dans le niveau « applications and workloads », on s'intéresse aux métadonnées sur les workloads. Les métadonnées du niveau « Systems » concernent non seulement la description d'une manière générale des systèmes de gestion des données mais aussi des propriétés des systèmes à l'exécution. Dans notre travail, nous ne détaillons pas les métadonnées des systèmes en cours d'exécution.
Dans ce chapitre, nous discutons les métadonnées du niveau données, workloads et systèmes. Le chapitre est divisé en trois parties : une première partie qui concerne les métadonnées sur les données et qui leur structure ainsi que les statistiques sur les répartitions de valeurs, une deuxième partie qui concerne les métadonnées sur les applications – description des workloads et des structures des requêtes les composant, et une troisième partie qui concerne le modèle des métadonnées du niveau systèmes. On conclut notre chapitre par des perspectives sur la gestion des métadonnées et une discussion sur les perspectives de travailler sur la problématique d'intégration de métadonnées.

Le chapitre est organisé comme suit : (i) la section 3.1 résume les métadonnées conçus pour le niveau « ensembles de données », (ii) la section 3.2 présente les métadonnées du niveau « applications et requêtes », (iii) la section 3.3 détaille les métadonnées du niveau « systèmes de gestion et de traitement de données dans un écosystème basé sur un lac de données » et finalement la section 3.4 présente une discussion et la conclusion du chapitre.

## 3.1 Ensembles de données

Dans cette partie nous détaillons le modèle de données des métadonnées sur les ensembles de données en présentant le diagramme de classe, et quelques exemples. Ces métadonnées décrivent les identifiants des ensembles (par exemple le chemin d'accès dans le système de fichiers distribué HDFS, les noms des ensembles de données, ...), leurs annotations, leurs statistiques descriptives (comme les valeurs médianes, min, max, la variance, covariance, moyenne), la liste des métadonnées sur leurs partitions (par exemple, le schéma de distribution des partitions qui peut être uniforme, ou bien qui peut favoriser certaines propriétés …), sur leurs index et sur leurs structures (notamment la liste des attributs, le nombre d'enregistrements, la structure de données, le type de fichier) et finalement leurs métadonnées administratives et de provenance définis comme la version (de type «timestamps»), le propriétaire, la licence, la source des données, la date de création, la date de modification et la description.

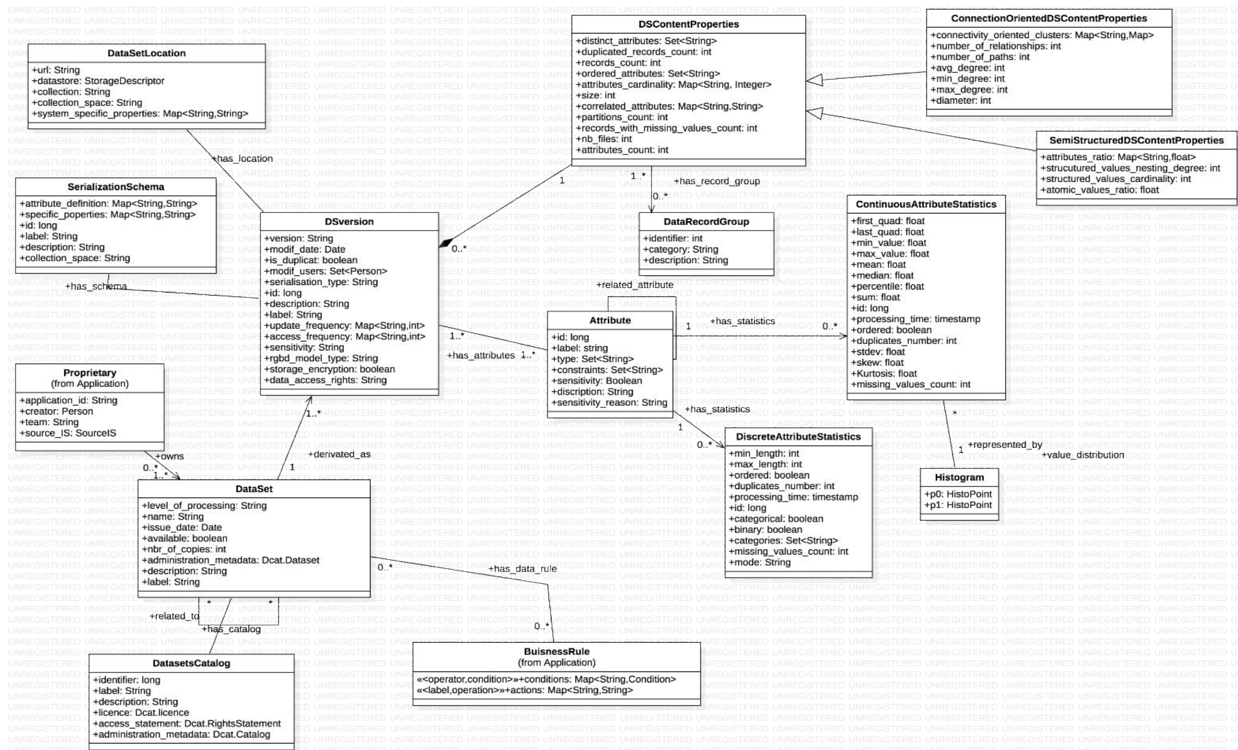



*Figure 3.1 Diagramme UML de classes des métadonnées niveau ensemble de données*

La Figure 3.1 présente les classes de métadonnées du niveau ensembles de données. Ce modèle permet de décrire d'une part les ensembles de données et leurs évolutions dans un écosystème basé sur un lac de données et d'autre part les statistiques qui sont nécessaires pour les modules d'optimisation des requêtes et la recommandation de placement. La classe « Dataset (ensemble de données) représente les métadonnées qui distinguent un ensemble de données d'une manière générale comme le nom, la description et la licence. Cette classe est liée avec une association un à plusieurs avec la classe « DSversion » qui permet de donner plus d'informations sur les versions des ensembles de données qui sont stockés dans l'écosystème complexe basé sur un lac de données. Cette dernière présente des attributs comme le numéro de version, les utilisateurs qui ont créé une version et qui l'ont modifié et des informations sur la sécurité (par exemple les droits d'accès à une version, la sécurité du stockage et le degré de sensibilité d'une version). La classe « Dataset » est aussi associé à des classes qui permettent de donner un contexte applicatif aux ensembles de données : (i) la classe « Proprietary » permet d'associer aux ensembles de données les applications qui les gèrent ainsi que leurs systèmes d'information et la classe « BuisnessRules » permet d'exprimer des règles métier sur les ensembles de données par exemple les règles de conformité que nous avons identifiées dans le problème de placement de données. Ces ensembles peuvent aussi être organisés en catégories. Ceci est modélisé par la relation « has-catalog » entre les classes « Dataset » et « DatasetsCatalog ». Dans notre modèle il est possible aussi de mettre en relation 2 ensemble qui n'appartiennent pas à la même catégorie de données en utilisant la relation réflexive « related_to ».

Dans ce qui suit, nous détaillons les classes de statistiques concernant des ensembles des données et leurs attributs, les classes sur la structure de ces ensembles et finalement les métadonnées sur les métadonnées d'administration de ces derniers.

### 3.1.1 Statistiques

Les classes des métadonnées de type statistiques sont très importantes pour DWS. Ils représentent une vue globale sur les statistiques des données dans l'écosystème et ils sont nécessaires pour notre

module de recommandation de placement de données ainsi que pour les modules d’optimisation des requêtes Big Data.

Ces statistiques sont de type descriptif. Ils concernent les attributs qui peuvent être discrets ou bien continus ainsi que les enregistrements. Le premier intérêt de gérer ce type de statiques est qu’ils peuvent être envoyés à l’optimiseur des requêtes qui pourra les exploiter pour l’estimation du coût d’exécution des requêtes. Notre solution utilise dans son architecture Apache Calcite qui est une boîte à outils pour l’optimisation de requêtes en Big Data, mais qui a besoin de statistiques fournies par un service tiers (cf. chapitre état de l’art). Un deuxième avantage de gérer ce type de métadonnées est son utilité dans les applications de qualité et plus précisément les solutions de nettoyage et de préparation de données [EEI+13,noad]

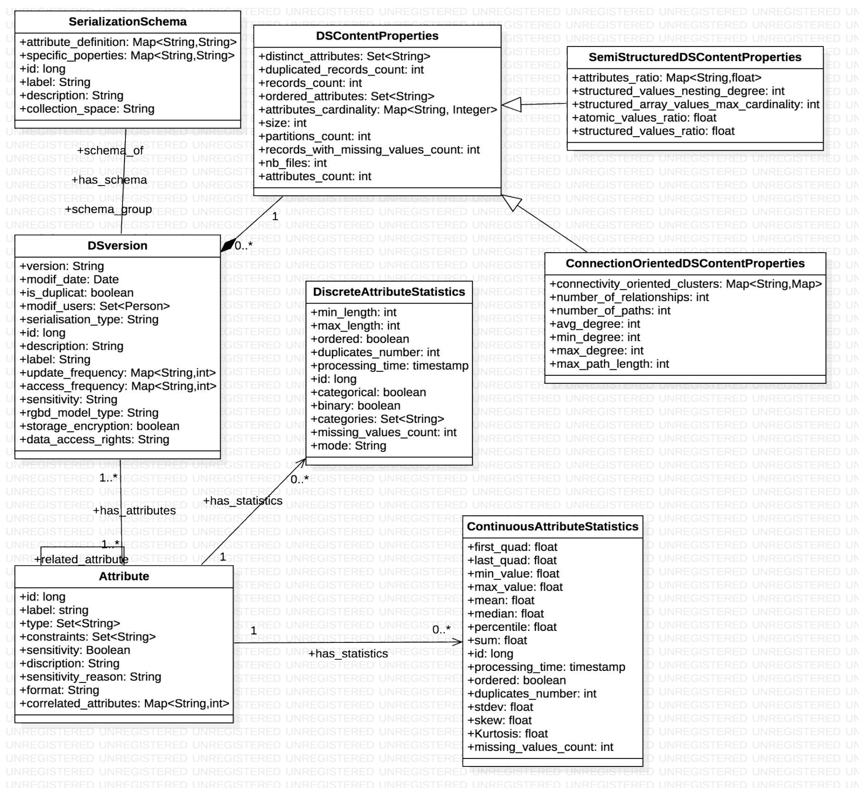



*Figure 3.2 Propriétés des classes pour les métadonnées de type statistiques*

La classe « DSContentProperties » représente les statistiques en générales sur les ensembles de données et les statistiques sur les enregistrements. (i) Les attributs « distincts_attributes » et « ordered_attributes » permettent de référencer respectivement les attributs distincts et ordonnées d’un ensemble de données. Le type de ces attributs est un ensemble de texte. (ii) Les attributs « attributes_count », « records_count », « duplicated_records_count » et « records_with_messing_values_count » sont des attributs de type entiers qui représentent le nombre des attributs d’un ensemble de données, des enregistrements, des enregistrements dupliqués et des enregistrements ayant des valeurs manquantes. (iii) La cardinalité des attributs dans un ensemble de données qui représente le nombre de valeurs uniques est représenté par l’attribut « attributes_cardinality » qui a un type dictionnaire de paires clef/valeur : texte/entier. (iv) finalement les attributs « size », « nb_files » et « partitions_count » sont des statistiques sur le stockage physique des données. Ils représentent respectivement la taille, le nombre de fichiers et le nombre de partitions d’un ensemble de données. En utilisant la propriété d’héritage offerte par le langage de modélisation unifié (UML) on étend cette classe et on permet à l’utilisateur de définir des statistiques sur des

données semi-structurées et les données de types graphes. Les statistiques sur les données semi-structurées sont définis par la classe « SemiStructuredContentProperties ». Elle contient 4 attributs :

- **attributes_ratio** : attribut qui permet de représenter pour chaque attribut le ratio nombre d'occurrences dans chaque enregistrement sur nombre total des enregistrements d'un ensemble de données semi-structuré.
- **atomic_values_ratio** : attribut qui permet de calculer le ratio entre le nombre d'attributs atomiques dans un ensemble de données
- **structured_array_values_max_cardinality** : attribut qui représente la cardinalité maximale des attributs multivalués, comme les tableaux.
- **structured_values_ratio** : attribut qui permet de calculer le ratio entre le nombre d'attributs structurés dans un ensemble de données
- **structured_values_nesting_degree** : attribut de type entier qui représente le degré d'imbrication des attributs de valeurs structurés.

Quant aux statistiques sur les données de type graphes, ils sont représentés par la classe « ConnectionOrientedContentProperties ». Ses attributs sont listés ci-dessous :

- **connectivity_oriented_clusters** : un attribut qui représente les clusters dans un ensemble de données de type graph. Il a comme type Map<String,Map> ce qui lui permet de stocker le nom du cluster ainsi que ces propriétés comme le nombres de nœuds et le nombre de relations.
- **number_of_relationships** : un entier qui représente le nombre de relations dans un graph
- **number_of_paths** : un entier qui représente le nombre de chemins dans un graph
- **avg_degree** (**min_degree** / **max_degree**) : un entier qui représente le degré moyen (ou bien minumum/maximum) des nœuds
- **max_path_length** : un entier qui représente le nombre de relations dans le plus long chemin.

Ces statistiques sont utiles pour résumer et donner plus d'information sur un ensemble de données. Ils sont utilisés aussi pour tirer des conclusions à partir d'analyse de ces statistiques comme dans le cadre d'applications de prédiction du temps de réponses d'une requête ou bien dans un rapport de Business Intelligence sur la qualité d'un ensemble de données.

### 3.1.2 Structure de données et statistiques sur les attributs

Le deuxième type de statistiques sur les ensembles de données défini dans notre système sont les statistiques sur les attributs. Dans cette sous-section, nous présentons ces statistiques ainsi que les classes qui décrivent la structure des ensembles de données.

La classe « Attribute » représente les attributs d'un ensemble de données. Cette dernière a une association avec la classe « DSversion » de multiplicité plusieurs à plusieurs, deux associations « has_statistics » de multiplicité un à plusieurs avec les classes de statistiques sur les attributs « DiscreteAttributeStatistics » et « ContinuousAttributeStatistics » et une association réflexive qui sert à relier les attributs qui ont un lien métier.

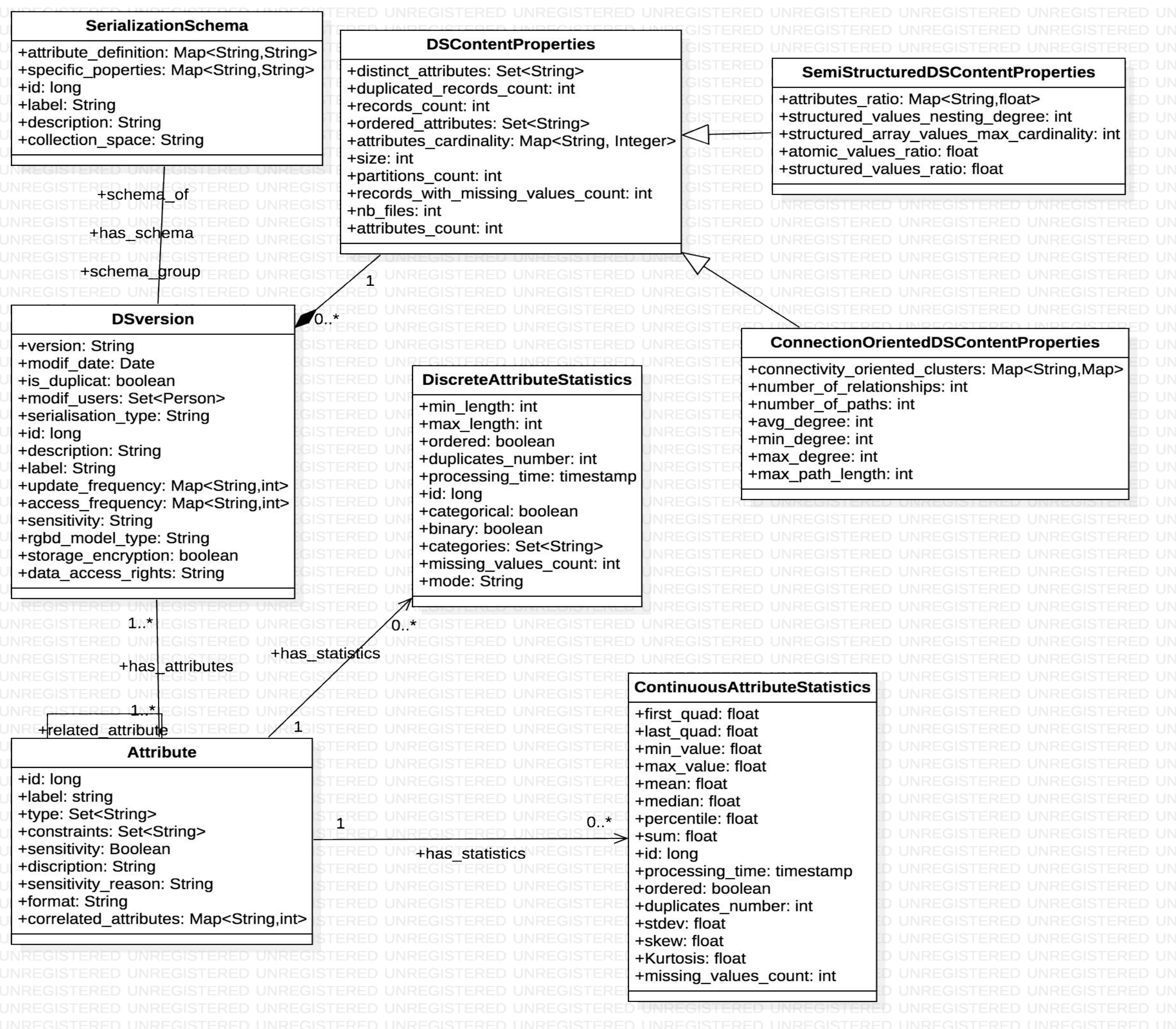



*Figure 3.3 Diagramme de classe pour les métadonnées : statistiques sur les ensembles des données*

Comme le montre le diagramme de classe de la figure 3.3, la classe « DiscreteAttributeStatistics » est composé des attributs suivants :

- **min_length** et **max_length** : attributs de type entier qui permet d'indiquer la taille maximale et minimale des valeurs d'un attribut de type discret.
- **ordered**, **binary**, **categorical** : attributs de type booléen qui indique si les valeurs de l'attribut sont ordonnés ou binaires ou qui représente une catégorie
- **categories** : attribut de type ensemble de texte qui représente la plage de valeurs possible pour cet attribut
- **missing_values_count** : attribut de type entier qui représente le nombre de valeurs manquante dans un attribut
- **mode** : attribut de type texte qui représente la valeur modale d'un attribut discret.

Quant à la classe « ContinuousAttributeStatistics » elle partage quelques attributs avec la classe précédente notamment les attributs : **id** et **processing_time** qui sont des attributs d'administration ainsi que l'attribut **ordered**. Elle contient en plus les attributs suivants :

- **first_quad**, **last_quad**, **median**, **mean** : attributs de type réel qui représentent des statistiques descriptives. Ils sont utiles pour avoir une vue résumée sur la distribution de données complétés par les histogrammes.
- **min_value**, **max_value** : les valeurs minimale et maximale. Ils ont un type réel.
- **percentile**, **sum**, **stdev**, **skew**, **kurtosis** : des statistiques avancées sur la distribution de données. **percentile** est le point à partir duquel les valeurs d'un attribut sont supérieures ou égales à la valeur qui se trouve à x pourcent de la population. **sum** est la somme de toutes les valeurs d'un attribut. **stdev** est l'écart type. C'est une mesure [noac] de la dispersion des valeurs d'un attribut qui est calculée à partir de la variance. Cette dernière est calculée à partir de la formule suivante : var = $\frac{\sum_{k=1}^{n} \quad (xi-avg(x))\text{^}2}{n-1}$. Quant aux attributs **kurtosis** et **skew**, ils concernent la mesure de l'aplatissement de la courbe de distribution des valeurs et l'asymétrie de cette dernière.
- **missing_values_count** : le nombre des valeurs manquantes. Cet attribut est nécessaire comme indicateur pour la qualité de données.

La distribution des données est calculée dans notre travail sous forme d'un histogramme. Dans notre conception, on calcule les points de l'histogramme et on utilise des instances de la classe « HistoPoint » pour stocker le résultat. Cette classe a une association avec la classe attribut dont la multiplicité est un à plusieurs. Pour terminer cette partie on présente les attributs de la classe « Attribute ». Ils servent surtout à donner un contexte métier aux attributs d'un ensemble de données. En plus des attributs d'administration **id**, **label**, **description**, cette classe contient des attributs sur le type comme : **type**, **format**, **constraints**. L'attribut **type** désigne le type des valeurs représenté par l'attribut de l'ensemble de données et dans un contexte Big Data ou les données sont semi-structurées ou non structurées ; ce dernier peut être multiple ou évolutif. **constraints** sert à exprimer des contraintes comme l'unicité. Quant au **format**, c'est un attribut qui permet à l'utilisateur de préciser le format attendu des attributs qui ont un type semi-structuré (comme le type texte libre). Utiliser cette pratique ou bien spécifier des métadonnées pour des exemples de valeurs attendues facilite le travail sur la qualité de données dans le contexte d'un écosystème Big Data. Un exemple de format pour le type attribut est pour un attribut adresse : « numero_voie quartier ville code_postale ». Le dernier attribut de type statistiques est **correlated_attributes** , il représente les attributs qui sont corrélés à l'attribut décrit par l'instance ainsi que le coefficient de corrélation. Notre modèle permet de caractériser la sensibilité d'un attribut. En respectant les consignes de la RGPD, il est possible de désigner des attributs comme sensibles dans l'objectif de protégés les données de l'entreprise ou des clients des usages malveillants des données. La sensibilité désigne les données qui contiennent des informations qui peuvent identifier une personne comme son numéro de téléphone ou bien ses opinions politiques, ou bien ce type d'information mais pseudonymisé dans des objectifs d'analyse marketing ou autre. L'attribut **sensitivity** de type booléen permet d'indiquer si un attribut est sensible et l'attribut **sensitivity_category** représente le motif de sensibilité d'un attribut. Ce dernier peut prendre comme valeurs : « personal information », « identification information », « pseudonomised personal information ». Cette classe a été modélisée en se basant sur les consignes de la RGPD publiés par la CNIL [noab]. Quant à l'attribut **has_personal_categorie**. Finalement, on permet aux utilisateurs de caractériser le type sémantique de l'attribut de l'ensemble de données (qui est différent du type technique comme le type entier). On définit ainsi l'attribut **attribute_kind_of** qui est de type texte et ses valeurs possibles peuvent être par exemple : « phone number », « address », « free text », « image », …

L’intérêt de définir les attributs **attribute_kind_of** est par exemple de pouvoir prédire les valeurs des attributs de sensibilité des données à partir de cet attribut, de l’attribut format et des données. Ce cas d’utilisation est une perspective intéressante à notre travail.

### 3.1.3 Métadonnées sur les versions des ensembles de données

Dans DWS il est important de garder une trace de l’évolution des données, de leurs versions et de leurs schémas. Avoir une description extensive des différents ensembles de données et associer à cette description des références à leurs emplacements physiques facilite les opérations de gestion du lac de données et optimise l’architecture de ce dernier.

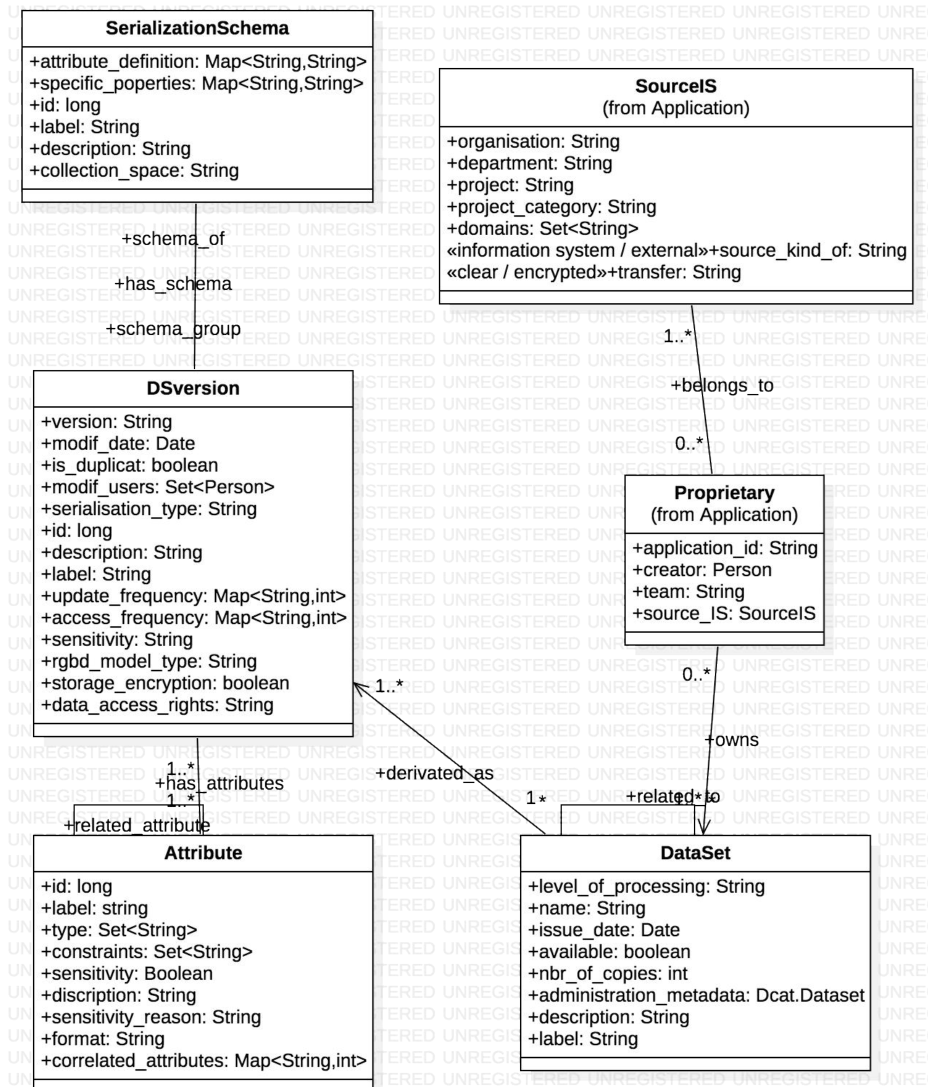



*Figure 3.4 Propriétés des classes pour les métadonnées sur les versions des données du lac*

La classe « DSversion » permet le suivi des versions des ensembles de données dans un écosystème basé sur un lac de données. L’association « derived_as » permet de relier chaque instance de cette classe à la classe « DataSet » qui regroupe les métadonnées administratives des ensembles de données notamment des métadonnées qui les identifient comme l’identifiant , le label et la description et des métadonnées spécifiques qui résument les versions disponibles dans l’écosystème comme le degré de traitement d’un ensemble (« level of processing ») qui peut être raffiné, brute, etc., la disponibilité d’un ensemble de données, la date de création de l’ensemble de données dans l’écosystème, une information sur la possibilité que la version soit une copie et le nombre de copies total. Dans les

attributs de la classe « DSversion » on décrit en détail les métadonnées techniques des versions des ensembles de données stockés dans l'écosystème. Les attributs **modif_date** et **modifs_users** permettent d'indiquer les dates de modifications d'un ensemble de données ainsi que les utilisateurs qui ont réalisé ces modifications. Dans ce travail la création d'une version est considérée comme modification initiale. Les attributs **sensitivity** (booléen), **sensitivity_categories** (ensemble de texte), **storage_encryption** (booléen), **data_access_rights** (texte dont les valeurs possibles sont par exemple a+r », « a+w », « a+x ») concerne les droits d'accès et la sensibilité des données. La sensibilité exprimée dans les deux premiers attributs et leurs valeurs dépendent de la valeur de la sensibilité dans les instances de la classe « Attribute » (une union des valeurs de **sensitivity_category**). Les derniers types de métadonnées qu'on trouve dans cette classe sont les métadonnées techniques représentées par les attributs : **serialisation_type**, un attribut de type texte qui indique le type de sérialisation dans l'écosystème de données (par exemple fichier de type CSV, données binaires, fichier avro), **update_frenquency** qui indique la fréquence de mise à jour de la version (par exemple <hourly, 4>) et **access_frequency** pour spécifier la fréquence d'accès).

Cette classe a une association « has_schema » avec la classe « SerializationSchema ». Cette dernière permet de représenter les schémas des données, de les stocker et de suivre leurs évolutions.

### 3.1.4 Conclusion

Il est possible d'utiliser des schémas de métadonnées existants. L'intégration de métadonnées avec d'autres schémas existants est une problématique importante, mais qui n'est pas dans le cœur de notre étude. Pour cette raison on utilise une méthode simple d'intégration : « intégration pilotée par l'utilisateur ».

Les sources de métadonnées que nous souhaitons intégrer dans notre solution ont un schéma bien connu, fixe et qui n'est pas évolutif. Ce sont des schémas standardisés ou bien qui sont extraits d'une API bien connue (dans notre cas Apache Atlas) la solution la plus appropriée pour l'intégration des données est le Global as View. On crée une vue globale et on utilise à partir de cette vue globale les autres métadonnées.

## 3.2 Applications et requêtes

Les métadonnées sur les workloads correspondent aux informations sur les requêtes usuelles (disponibles en gardant un historique des requêtes) ainsi qu'aux métadonnées d'administration des tâches (jobs). On trouve ainsi : la description des patrons des requêtes (« attribute oriented », « range oriented », …), la fréquence des requêtes (« continuous stream », « days of the week », « hours », …), la géolocalisation des clients (les clients qui accèdent une partition particulière doivent être à proximité du serveur qui gère cette partition), les plans logique et physique de la requête, et finalement des statistiques (particulièrement les mesures de « selectivity », « cardinality » et du « cost »).

Les métadonnées d'administration des workloads sont relatives aux informations sur le suivi d'exécution et du parallélisme et on peut trouver par exemple pour le système Hive les informations concernant le nom du système de parallélisation (par exemple Tez), le nom du système qui a soumis le job (dans ce cas de figure il s'agit de Hive), les timestamps, le nom de l'application métier qui a créé le job, etc. Dans ce travail on s'intéresse aux métadonnées du niveau applicatif qui décrivent les requêtes et les workloads des applications Big Data. Le diagramme de classe suivant regroupe les classes de ce niveau.

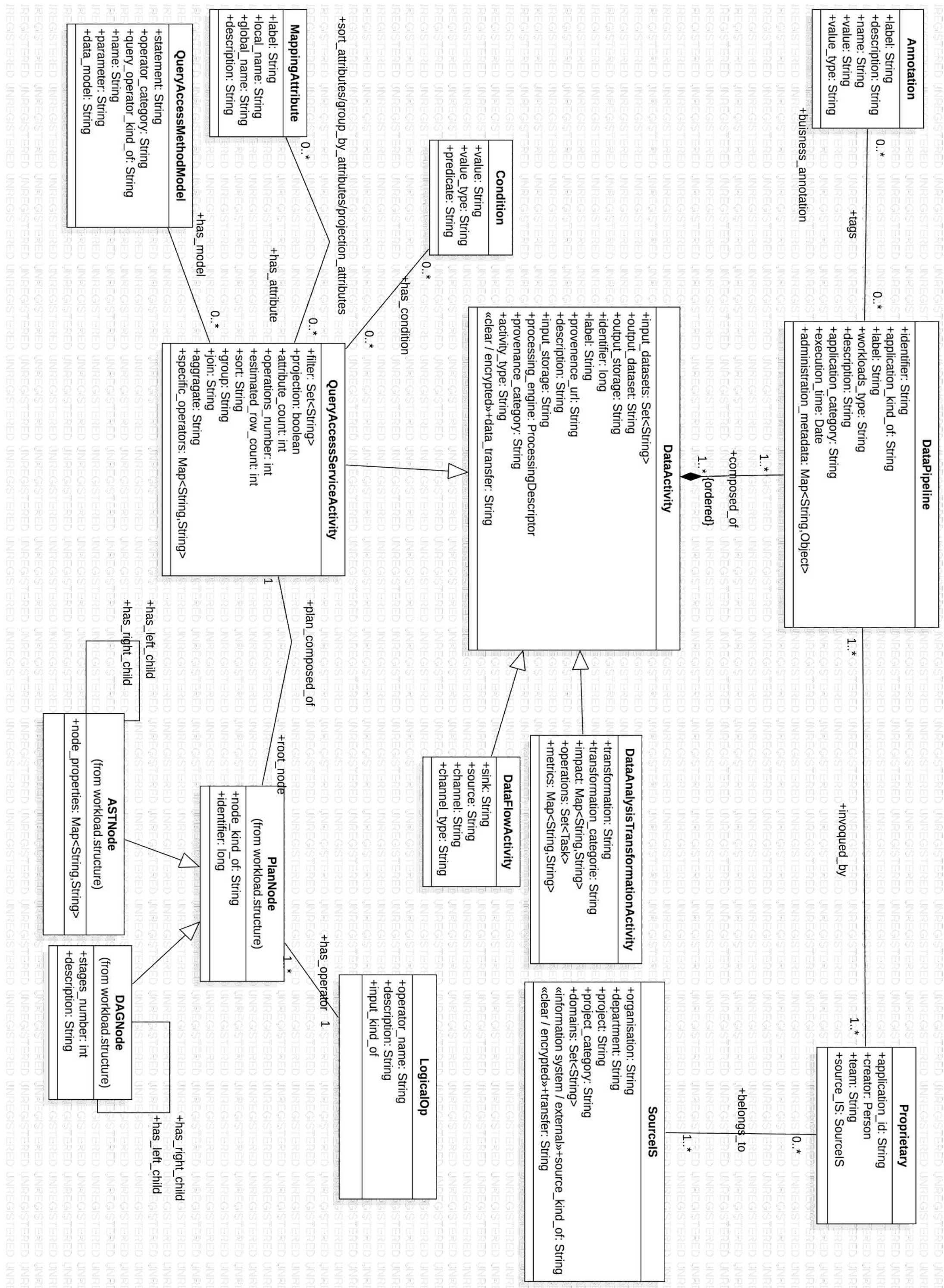



*Figure 3.5 Diagramme de classe des métadonnées du niveau applicatif*

Dans notre modèle, nous considérons un workload dans une application Big Data comme une séquence d'activités sur des données et qui peuvent être des requêtes, des fonctions de transformation de données ou bien des fonctions de traitement de flux de données. Cette spécification d'un workload se traduit par la classe « DataPipeline » et la relation de composition « composed_of » de type n à n qui la relie avec la classe « DataActivity ». On associe aussi à cette classe, des classes de métadonnées applicatives : la classe « Proprietary » qui permet de suivre les utilisateurs qui ont lancé un workload et la classe « Annotation » qui permet d'associer des annotations métiers à un workload.

Les liens entre ces dernières et la classe « DataPipeline » sont des liens plusieurs à plusieurs comme le montre la figure ci-dessus. Les propriétaires des workloads sont organisés en plusieurs systèmes d'informations. Cette modélisation peut préparer les briques de base pour mettre en place une stratégie de confidentialité et de sécurité des données et des applications dans l'écosystème. Les attributs de la classe « Proprietary » sont définis comme suit :

- **creator** : attribut de type « Person » une classe qu'on réutilise du standard FOAF et qui permet de spécifier les utilisateurs qui peuvent s'approprier des données
- **team** : attribut de type chaîne de caractères qui permet d'indiquer le label de l'équipe à laquelle appartient le propriétaire de l'ensemble de données.
- **source** : attribut de type « SourceIS ». La classe « SourceIS » permet de préciser la source de laquelle provient la donnée.

À partir de la classe « DataActivity » on peut créer d'autres sous types en utilisant l'héritage comme la classe « QueryAccessServiceActivity ». Dans les trois sous-sections suivantes nous détaillons d'abords les propriétés de la classe « DataPipeline » qui représente le workload, ensuite la classe « DataActivity » et ces descendants et finalement les classes que nous avons conçues pour les opérateurs du workload.

### 3.2.1 Propriétés des workloads

La classe « DataPipeline » permet de caractériser un workload. Le concept modélisé par cette classe est celui des pipelines de données. Comme on l'a expliqué plus haut, dans notre contexte la définition de pipeline de données est généralisée. C'est une séquence d'activités qui peut être une requête, un flux de données ou bien tout autre service de traitement de données. Dans d'autres plateformes comme AWS un pipeline n'est composé que des services de traitement de flux de données [noaa]. Dans la figure suivante, on fait un zoom sur les attributs de cette classe.

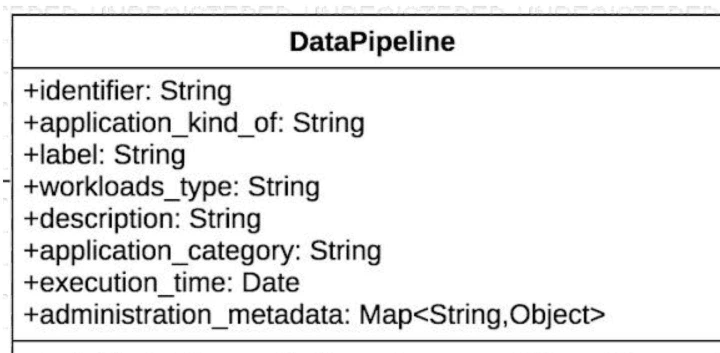



*Figure 3.6 Structure de la classe DataPipeline*

Les attributs **identitifier** et **label** servent à identifier d'une manière unique un workload. Le premier est un numéro généré aléatoirement et le deuxième est le nom de l'application. L'attribut **application_category** permet de classifier les workloads dans un écosystème Big Data en plusieurs types comme les applications d'analyse de données, les applications d'interrogation de données ou bien les applications d'apprentissage automatique. Par contre l'attribut **application_kind_of** sert à associer le type à l'application qui a créé le workload. Il peut prendre comme valeur : « streaming », « data management », « data analysis », … Quant aux attributs **execution_time** et **workload_duration** ils concernent respectivement l'instant de début d'exécution du workload et sa durée. Ils ont le type « DateTime ». Finalement, notre model conceptuel permet d'enrichir les métadonnées avec des

instances de modèles conceptuels existants concernant des détails sur l'administration d'une ressource.

### 3.2.2 Structures des requêtes

Le deuxième concept que nous allons détailler est celui des activités sur les données et nous allons nous limiter ici aux requêtes de type SQL. Cette restriction est motivée par notre définition du problème et le choix du langage SQL pour exprimer les workload en entrée du système de recommandation DWS. Dans cette sous-section nous présentons la classe « DataActivity » et la classe enfant de la relation d'héritage à laquelle participe cette classe.

La classe « DataActivity » est composée d'attributs qui concernent généralement le suivi des versions des données dans les workloads. Quant aux attributs de la classe « QueryAccessServiceActivity », ils concernent surtout les attributs sur lesquels on peut avoir des statistiques nécessaires pour l'optimisation des requêtes. Comme le montre la figure ci-dessus on trouve dans la première classe les attributs suivant :

- **input_datasets**, **input_storage** » : attributs de type ensemble de valeurs textuelles qui représente les références des ensembles de données en entrées d'une activité sur les données et la référence de leurs systèmes de stockage.
- **processing_engine** : attribut de type texte qui permet d'indiquer le moteur de traitement qui a exécuté l'activité de données.
- **output_storage** : attribut de type texte qui représente la référence d'un ensemble de données en sortie (s'il existe) d'une activité sur les données et la référence de son système de stockage.
- **activity_type** : attribut de type texte. Il sert à classer les activités selon leurs types.
- **provonence_url**, **provonence_categorie** : deux attributs de type texte qui permettent de renseigner l'url du service qui a créé le workload et sa catégorie. La catégorie prend une des valeurs suivantes : « batch job », « application », « web service », …
- **data_tranfer** : attribut de type texte qui permet d'indiquer si le transfert de données est fait en clair ou bien en crypté dans cette activité de données.

Cette classe contient aussi des métadonnées descriptives comme les attributs **description** et **label**. En revanche, la classe « QueryAccessServiceActivity » contient les attributs suivants :

- **filter**, **join**, **sort** : attributs de type ensemble de texte. Ils permettent de renseigner les attributs qui ont été accédés dans une requête / service d'accès aux données. À partir de ces attributs, il est facile de déterminer la sélectivité de ce filtre en utilisant des formules connues dans le domaine d'interrogation de données.
- **project**, **group**, **aggregate**: attributs de type ensemble de texte qui sert à déterminer les attributs qui ont été accédés par ces opérateurs.
- **estimated_row_count** : attribut de type entier qui représente le nombre estimé d'enregistrements retournés
- **operations_number** : attribut de type entier qui représente le nombre d'opérateurs.
- **specific_operator** : attribut de type dictionnaire de paires clé-valeur. Il permet de représenter les attributs des opérateurs spécifiques d'une activité sur les données.

La classe qui représente les activités sur les données à une relation d'association (« plan_composed_of ») avec la classe « PlanNode ». Elle a aussi une relation « has_attribute » avec

une classe « MappingAttribute » qui permet de spécifier les correspondances entre le schéma global des données et local. La relation « plan_composed_of » permet de stocker les opérateurs de ces activités dans des plans d'opérateurs de deux types. Les plans ayant une structure de données de type AST, acronyme pour les arbres asymétriques de syntaxe, et celle de type « dag_acronym » pour les graphs acycliques orientés (cf. chapitre 2).

La classe « PlanNode » contient trois attributs : (i) un identifiant unique pour chaque nœud du graph de type numérique. (ii) la description d'un nœud et (iii) le type du nœud (**node_kind_of**). Cette classe est spécifiée par les deux classes « ASTNode » et « DAGNode ».

Ces deux classes possèdent une relation réflexive « has_left_child », « has_right_child » qui permettent de construire le graphe. La classe « PlanNode » est reliée à la classe « LogicalOp » par une relation un à plusieurs. Cette dernière permet de définir un ensemble d'opérateurs réutilisables et qui servent à être référencés dans les plans d' activités de données.

## 3.3 Systèmes de gestion et de traitement de données

Chaque système possède une multitude de caractéristiques et répond à un ensemble de besoins spécifiques. Par conséquent, le choix du modèle de données et de traitement, du schéma de distribution des données et du moteur de stockage devient un problème de décision qui nécessite l'intervention d'un expert.

Ces informations couplées aux métadonnées sur les systèmes en cours d'exécution et les spécifications des besoins fonctionnels aussi bien que non fonctionnels permettent de dégager des recommandations sur par exemple le placement optimal des données qui contribue significativement à l'amélioration de l'organisation du lac de données.

Les métadonnées sur l'exécution dans ces systèmes correspondent à des métadonnées collectées sur l'infrastructure, à des statistiques et métriques sur l'exécution des workloads (charge de travail), ainsi que des configurations des systèmes et leurs schémas d'organisation. Elles dépendent d'une architecture et d'une utilisation particulière d'un lac de données. D'où leur caractérisation comme propriétés dynamiques.

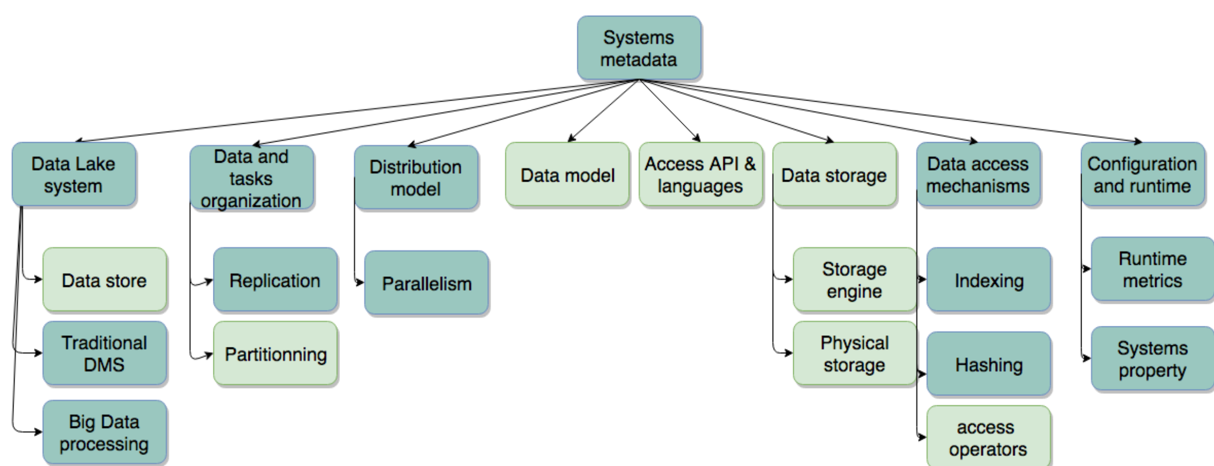



*Figure 3.7 Classification des métadonnées pour le niveau système*

Les métadonnées de ce niveau sont regroupées sous forme de classes organisées en hiérarchie et en agrégats et qui caractérisent respectivement le modèle de partitionnement, le modèle de données, le

modèle de requêtes et le modèle de stockage. Dans les sous-sections suivantes on détaille ces métadonnées.

### 3.3.1 Lacs de données

Les métadonnées «data lake systems» décrivent les composants principaux des architectures de systèmes de lacs de données. Elles représentent les entités de liaison entre les différents groupes de métadonnées du niveau systèmes. Dans notre solution le lac de données est composé de :

- «data stores» comme les systèmes NoSQL et les systèmes de fichier
- «Big Data processing engines» comme Spark, Hadoop MapReduce
- «traditional dbms» comme les SGBD relationnels

Dans cette thèse, on s'intéresse surtout aux data stores. Les métadonnées des deux autres systèmes ne sont pas détaillées.

#### *3.3.1.1 Classe DataStore*

Les métadonnées sur les data stores sont représentées dans la classe « DataStore ». Cette classe décrit les systèmes de gestion de données de lacs de type magasin de données. Elle a comme attributs :

- **schema_management** : attribut de type texte qui décrit la possibilité de définir au préalable un schéma de données et le système utilisé pour le gérer (par exemple HCatalog pour Hive)
- **load_balancing** : attribut de type texte qui renseigne sur les caractéristique du load balancing dans un système data store.
- **data_locality**: dans un environnement distribué, les informations sur l'emplacement du stockage physique sont importantes. La modélisation de l'architecture de distribution des données en ayant au préalable une connaissance sur cette propriété permet d'améliorer le temps de réponse des requêtes dans l'environnement distribué ainsi que le passage à l'échelle des charges de lecture et écriture. Par exemple, cette notion est concrétisée par la fonctionnalité de « rack awareness » dans Hadoop, et elle est implantée dans MongoDB sous forme de « data locality » au niveau du sharding.
- **data_exchange_protocol** : attribut de type liste qui regroupe les différents protocoles d'échange des données. Par exemple, il contient la valeur « thrift » pour le système Cassandra. Cette classe a aussi des associations « has_data_model », « has_data_query_component », « has_default_storage_engine », « has_other_storage_engines » et « has_distribution_model », respectivement avec les classes : « DataModel », « DataQuerySystem », « StorageEngine » et « DistributionModel ».

L'instance « HdfsHBaseSystem » pour cette classe, décrivant les propriétés de HBase [Ser13], est :

> (HdfsHBaseSystem, DataStore ( schema_management : «default not supported, but possible with a serialization tool like Apache avro. Column families and tables are also persisted along with informations of region servers in the table hbase:meta and managed by the user»
> load_balancing : «hbase has its own load balancer but can be configured to use hdfs load balancer»
> data_locality : «HDFS rack awareness»
> data_exchange_protocol : { «http/rest», «thrift»} ))

#### 3.3.1.2 Classe DistributionModel

Les systèmes de lacs de données possèdent des modèles de distribution décrits par des instances de la classe « DistributionModel ». Ces attributs sont :

- **cluster_architecture** : un attribut booléen qui indique si l'architecture du système est de type cluster.
- **replication_kind_of** : un attribut de type texte qui renseigne le type de réplication de données.
- **replication_architecture** : un attribut de type texte qui indique l'architecture de la réplication du système. Il peut avoir comme valeur : «P2P» ou bien «master/slave».
- **replication_factor** : un attribut de type numérique qui représente le nombre de réplicas par défaut dans le modèle de distribution.
- **partitioning_model** : un attribut qui contient une instance de la classe « Partitioning ».

### 3.3.2 Partitionnement des données

Le partitionnement d'une base de données [SF12] est une méthode qui consiste à partitionner un ensemble de données dans de multiples bases de données physiques. Les partitions – sous-ensembles créés par cette technique – sont accédées et manipulées conjointement comme une seule base de données logique (un seul ensemble) tout en assurant que les détails de l'implémentation physique soient totalement gérés par le système et cachés à l'utilisateur final (application). L'avantage principal du partitionnement est le passage à l'échelle des données et la performance des opérations de lecture. Avec cette technique, un système du lac de données est capable d'exécuter les requêtes plus rapidement puisque l'opération d'accès sera faite sur un nombre plus réduit de données. Pour atteindre un passage à l'échelle optimal en lecture. Il est toutefois recommandé de combiner les techniques de partitionnement et de réplication.

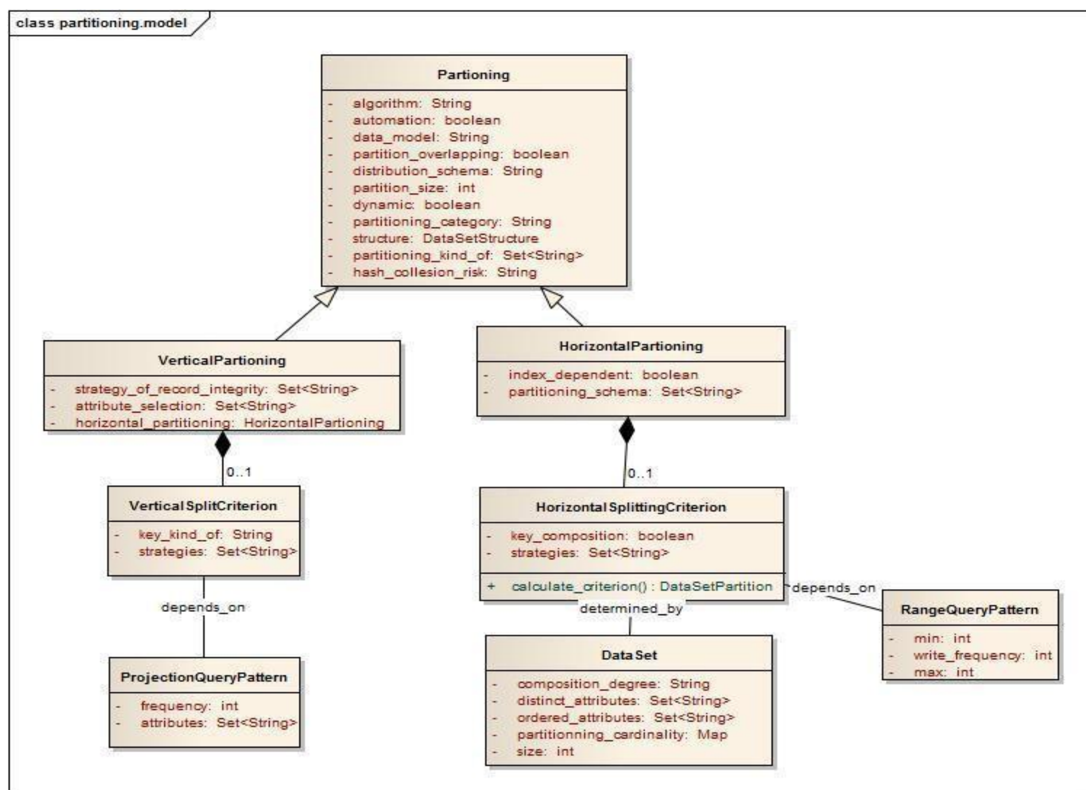



*Figure 3.8 Diagramme de classe des «modèles de partitionnement»*

Le diagramme de classe illustré dans la figure 3.8 décrit la classe « Partitioning » qui représente les techniques de partitionnement et ses interactions. Cette classe est héritée par deux classes correspondant aux différents types de partitionnement qui existent dans les systèmes de gestion de lacs de données : « VerticalPartitioning » et « HorizontalPartitioning ».

Ces classes concernent respectivement les techniques de partitionnement horizontal et vertical. Elles sont décrites plus précisément dans les sections 3.3.2.2 et 3.3.2.

#### *3.3.2.1 Classe Partitioning*

La classe « Partitioning » représente les propriétés générales des techniques de partitionnement. Elle a comme attributs :

- **partitioning_kind_of** : un attribut de type ensemble de textes qui décrit le type de partitionnement qui peut avoir comme valeurs: «key based», «hash based», ...
- **partitioning_categorie** : c'est un attribut de type texte qui indique la catégorie de partitionnement. Ses valeurs possibles sont :
- **size-only** pour les partitionnement naïfs qui ne dépendent que de la taille de l'ensemble de données, ils ont l'inconvénient de ne supporter que les opérateur physique d'accès de type «full scan» et ne permettent pas le «skip scan» avec le partitionnement
- **workload based** ou bien **adaptative** qui dépend de l'historique des workloads (des requêtes statiques connues au préalable). Ce type de partitionnement souffre de performances très limités au premières phases d'utilisation, l'accès est plus lent dans ce qu'avec l'opérateur «full scan»
- **ad-hoc** pour le partitionnement qui ne dépend pas d'une spécification à priori des workloads (par exemple le partitionnement robuste détaillé dans le papier [SJM+17] qui partitionne un ensemble de données selon une structure arbre et un algorithme de comparaison basé sur des métadonnées de partitionnement).
- **structure** : attribut de type « DataSetStructure ». Il représente la structure de partitionnement. Un exemple de valeurs possible pour cet attribut est «treeDS».
- **algorithm** : attribut de type texte qui renseigne sur l'algorithme utilisé pour le partitionnement. Ses valeurs peuvent être : «hashing algorithm», «none», «comparaison algorithm»
- **hash_collesion_risk** : attribut de type texte qui renseigne sur la possibilité de la collision de la fonction de hachage. Cet attribut a pour valeur «élevé» pour le hachage naïf, «moyen» pour le hachage basé sur une structure de hachage et des stratégies de collision (cf. data access mechanism classe hachage). Par exemple dans MongoDB le sharding à base de hachage d'index utilise une fonction trunc pour hacher les clés d'index qui possèdent des nombres à virgule flottante en entiers 64 bits avant le hachage. Un index haché stockait la même valeur pour un champ qui contient une valeur de 2,3, 2,2 et 2,9. Pour éviter les collisions, il est recommandé de ne pas utiliser d'index haché pour les nombres à virgule flottante qui ne peuvent pas être convertis de manière fiable en entiers 64 bits (puis revenez au flottant). Les index hachés MongoDB ne prennent pas en charge les valeurs à virgule flottante supérieures à 253.
- **data_model** : une valeur qui représente le modèle de données d'une technique de partitionnement pour indiquer qu'un algorithme de partitionnement est dépendant

d'un modèle de données particulier. Par exemple : le partitionnement de graphe, le partitionnement d'ensembles et le partitionnement de tables ont des techniques de partitionnement différents (et même différents algorithmes mathématiques)

- **automation** : attribut de type booléen qui indique si le partitionnement est fait automatiquement ou par intervention de l'utilisateur. En effet, certains systèmes offrent la fonctionnalité de partitionnement automatique. C'est le système qui détermine le critère de partitionnement qui est représenté par la classe « HorinzontalSplitCriterion ». Par exemple, dans MongoDB, la plage de partition (attribut range de la classe « SplittingCriterion ») est déterminée automatiquement.
- **dynamic** : attribut booléen qui indique si le (re)partitionnement est supporté, dans le cas échéant il s'agit d'un partitionnement statique.
- **distribution_schema** : information sur l'équilibrage de la charge des lectures et des écritures, il peut avoir comme valeur «uniform» or «unbalanced» or «none». En effet, certains types de partitionnement sont plus susceptibles de produire une distribution non équilibrée que d'autres. Par exemple le cas du range « based sharding » de MongoDB comparé au «hashed sharding» du même système.
- **partition_overlapping** : un attribut booléen qui indique si les intersections de partitions sont permises. Il prend la valeur « faux » si chaque sous-ensemble des attributs ou des enregistrements est associé à une partition séparée.
- **partition_size** : un attribut numérique qui représente la taille d'une partition.

#### *3.3.2.2 Class HorizontalPartitioning*

Cette classe représente le partitionnement horizontal qui est un partitionnement en collections (attribut **data_collection** de la classe « DataModel ») ayant un nombre plus réduit d'enregistrements (classe « DataRecord ») selon un critère particulier. Toutefois il est important de noter que la notion de collection ici n'est pas propre au modèle de données du système MongoDB elle est plutôt relative aux métamodèles « DataModel » qu'on a établi pour couvrir un modèle de données de tous les systèmes gestion de lacs de données.

Il existe plusieurs catégories de partitionnement horizontal qui dépendent de la structure de partitionnement, de l'algorithme, du modèle de données.

L'attribut **partitioning_schema** décrit ces catégories. Il est de type ensemble de textes et il prend une ou plusieurs des valeurs citées ci-dessous. Une instance de la classe « HorizontalPartitioning » dont la taille l'attribut **partitioning_schema** est supérieur à 1 indique que le partitionnement est composite et qu'il supporte la combinaison de valeurs possibles suivantes :

- «range based partitioning» : une valeur qui correspond à la création de sous-ensembles de données qui vérifient un critère défini au préalable. En effet, ce partitionnement dépend des valeurs d'un attribut sélectionné ; il divise la collection de données sources selon une plage de la valeur de cet attribut. L'attribut sélectionné et la valeur max de cette plage constitue le critère de partitionnement. Ce critère subit un ensemble de règles pour aboutir à une division optimale définie dans la classe « SplittingCriterion ». Les partitions dans ce type de partitionnement n'ont aucune intersection d'où la valeur de **partition_overlapping** est false pour ce type de partitionnement.

- «tag aware partitioning» : une valeur pour un type de partitionnement similaire au type précédent. Ce partitionnement définit des annotations pour représenter les plages de valeurs relatives à l'attribut qui détermine partition. Il divise les partitions selon un ou plusieurs annotations. La valeur de l'attribut **partition_overlapping** est true pour ce type de partitionnement. La définition au préalable de ces tags permet à l'utilisateur de décider l'emplacement des collections de données, ce qui améliore la localisation des données (data locality)
- « list based partitioning» : une valeur associée à un partitionnement qui est équivalent au range based partitioning avec la seule différence qu'il partitionne les données selon des valeurs stockées dans une liste fixée au préalable au lieu de l'utilisation de plages de valeur
- «hash based partitioning» : cette valeur est relative à une technique de partitionnement qui attribue pour chaque enregistrement une partition cible selon une fonction de hachage exécutée sur la clé de partitionnement. L'accès se fait dans ce cas avec un index haché.
- «consistent hashing» : c'est une valeur qui réfère une technique de partitionnement basé sur la clé de l'enregistrement de données (« DataRecord »). Elle utilise une fonction de hachage distribuée pour partitionner les données ainsi qu'une structure abstraite pour son hachage. Elle a l'avantage de ne pas dépendre du nombre de serveurs de distributions et ne nécessite pas une redistribution en cas de panne d'un serveur. La fonction de hachage calcule pour chaque enregistrement des coordonnées (un angle géométrique) dans un cercle abstrait entre 2 points $P_i$ et $P_j$. Ces points correspondent à des serveurs de partitionnement et l'enregistrement haché est stocké dans le serveur $P_j$. L'intérêt de cette fonction de hachage est qu'elle ne dépend pas du nombre de serveurs dans le schéma de distribution et du nombre d'entrées dans la structure de hachage. Des index supplémentaires peuvent être ajoutés entre les identifiants de serveurs pour accélérer l'accès aux données.

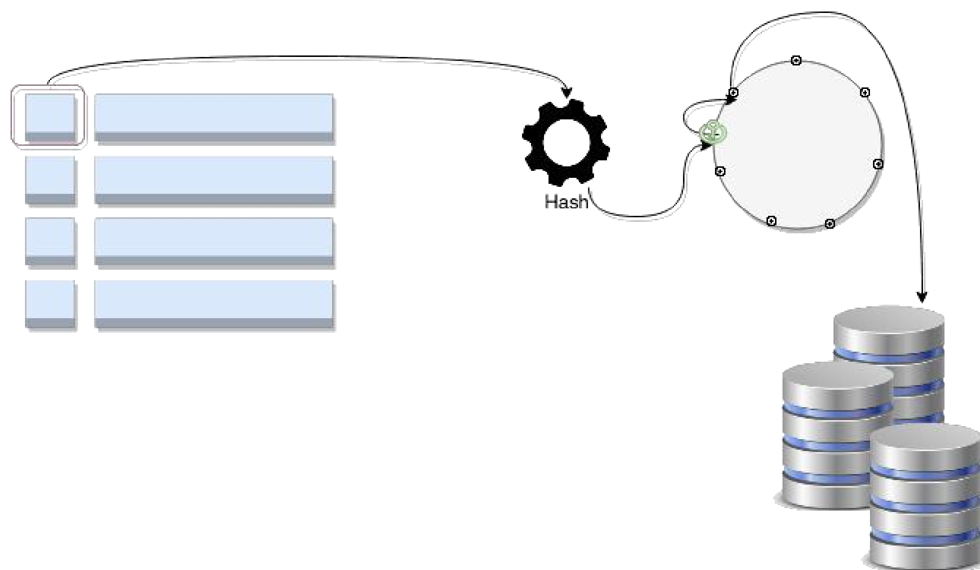



*Figure 3.9 Hachage consistant*

- «modulo based hashing» : c'est une valeur réfère une technique de partitionnement qui utilise une fonction de hachage dépendant du nombre de serveur du schéma de partitionnement. L'identifiant du partitionnement est retrouvé selon la fonction de hachage (key%num-serveur)
- «graph_partitioning» : c'est une valeur qui correspond à la catégorie de partitionnement qui transforme une structure de données tabulaire en une matrice, puis, utilise des techniques (algorithmes) de partitionnement de graphes pour partitionner les données.

En effet, il existe dans la littérature plusieurs algorithmes de partitionnement de graph qui sont utilisés après transformation de la structure. Il existe d'autres types de partitionnement plus avancés comme le partitionnement basé sur la structure que nous ne détaillons pas à ce niveau.

Le deuxième attribut de la classe « HorizontalPartitioning » est **index_dependent**. Il est de type booléen et il indique si le partitionnement nécessite une étape d'indexation au préalable (comme le cas du range partitioning de MongoDB).

La classe « HorizontalPartitioning » permet de définir des instances correspondant au partitionnement horizontal utilisées dans les systèmes de gestion de lacs. Soit par exemple le sharding dans MongoDB, le bucketing dans Hadoop et le table partitioning dans les systèmes supportant des modèles de données basés sur une structure de données tabulaire.

On détaille ci-dessous l'instance de la classe « HorizontalPartitioning » pour le partitionnement basé sur les plages dans MongoDB (en anglais *range based sharding* )

(MongoDBRangeBasedSharding, HorizontalPartitioning (
partitioning_kind_of : {«key based»},
partitioning_categorie : «size-only»,
structure : «none»,
algorithm : «comparaison algorithm»,
data_model : «any»,
automation : True,
dynamique : False,
distribution_schema : «unbalanced»,
partition_overlapping : False,
partition_size : 64 megabytes,
partitioning_schema : {«range based partitioning»},
index_dependent : True,
hash_collesion_risk :«collision risk is high in case of a
partition key of type float»,
key_design : rangeShardingSplitCriterion
))

#### *3.3.2.3 Classe HorizontalSplitCriterion*

La classe « HorizontalSplittingCriterion » représente les concepts caractérisant les critères de partitionnement horizontal. Elle a un lien de composition avec la classe « HorizontalPartitioning » avec la cardinalité 0..1.

Effectivement, pour partitionner, un utilisateur ou le système (dans le cas de partitionnement automatique) détermine l'attribut (ou l'ensemble des attributs) à considérer comme clé. Les contraintes et les règles de choix d'une clé de partitionnement sont modélisées dans cette classe sous forme des attributs suivants :

- **key_composition** : attribut booléen qui indique si la clé de partitionnement horizontale peut être composé de plusieurs attributs
- **strategies** : un attribut de type ensemble de textes qui stocke les bonnes pratiques pour concevoir une clé de partitionnement, il a des valeurs comme : «si DataAttribute.value_is_ordered, il faut choisir cet attribut de données particulier pour la clé de partitionnement», «recalculer l'attribut identifiant dans l'application pour l'utiliser comme clé de partitionnement», «privilégier les attributs qui ont un niveau de cardinalité haut pour l'utiliser comme clé de partitionnement», «un attribut qui a des valeur nuls ne peut pas être choisir pour le partitionnement»

Le partitionnement optimal dépend des requêtes usuelles, de la structure des ensembles de données ; il produit d'autres métadonnées relatives aux caractéristiques des partitions créées à partir des ensembles de données source (type de retour de la méthode « calculate_criterion »).

Ainsi on peut déterminer qu'il existe des dépendances entre cette classe avec les classes « QueryPattern » du niveau workload et « Dataset du niveau Data, mais on ne détaille pas ces associations puisqu'elles mettent en jeux des classes des niveaux non couverts par ce document.

#### *3.3.2.4 Classe VerticalPartitioning*

Le partitionnement vertical est un type de partitionnement qui divise une collection de données selon ses attributs en sous-groupes physiquement distinguables. C'est un partitionnement en collection de données avec des data records de tailles plus petites. Le partitionnement vertical est optimal pour les systèmes performants pour les jointures (par exemple les systèmes qui ont un type de modèle de données « StructuredDataModel ») et qui autorisent la manipulation de plusieurs collections de données à la fois. Toutefois, ce type de partitionnement nécessite une stratégie / contrainte d'intégrité sur les enregistrements. L'attribut **strategy_of_record_integrity** décrit cette information. Il est de type ensemble de textes et peut contenir comme valeurs possibles :

- «partition sorting» pour le maintien de l'ordre des enregistrement à travers les partitions
- «identifier partition duplication» qui consiste à associer pour chaque n-uplet logique d'une table fractionnée le même identifiant dans toutes les partitions que celui de la collection source. Par exemple, une jointure des n-uplets ayant le même identifiant de chaque table fractionnée construit le n-uplet d'origine.
- «referential integrity» Ce type de partitionnement nécessite une stratégie pour la sélection d'attributs à partitionner. Le choix des attributs est un problème de complexité élevé [BCN17] et les approches pour le faire sont renseignées dans l'attribut **attribute_selection**. Cet attribut est de type ensemble de textes et peut avoir les valeurs : «data driven», «workload dependent», «workload and data dependent».

Généralement, le partitionnement vertical est couplé avec un partitionnement Horizontal. Cette propriété est décrite dans « VerticalPartitioning » par l'attribut **horizontal_partitioning** qui prend comme valeur un objet de la classe « HorizontalPartitioning ».

Les techniques utilisées par le partitionnement vertical sont la normalisation ou bien la division des lignes (rows splitting). Ces techniques sont représentées sous forme d'instances de la classe « VerticalPartitioning ».

L'instance «normalisation» : La normalisation est un processus standard de base de données consistant à supprimer des informations redondantes d'une relation et à les placer dans des relations secondaires liées à la relation primaire par le biais de l'opérateur de jointure. Cette instance a comme propriétés:

(normalisation, VerticalPartitioning ( strategy_of_record_integrity :{ «referential integrity»},
attribute_selection :{},
horizontal_partitioning: NULL
))

L'instance «rowSplitting» : La division des n-uplets divise verticalement la relation d'origine en relations avec moins de colonnes.

(rowSplitting, VerticalPartitioning (
strategy_of_record_integrity : { «identifier partition duplication», «partition sorting»},
attribute_selection :{«data driven», «workload dependent», «workload and data dependent»},
horizontal_partitioning: NULL
))

#### *3.3.2.5 Classe VerticalSplitCriterion*

De manière similaire à la classe « HorizontalSplittingCriterion », la classe « VerticalSplitCriterion » représente les concepts caractérisant les critères de partitionnement vertical. Elle est associée à la classe « VerticalPartitioning » par un lien de composition de cardinalité 0..1.

Les attributs de cette classe sont :

- **key_kind_of** : attribut texte qui a comme valeurs possible : «heuristic-based» qui représente un type d'algorithmes utilisé dans le partitionnement vertical surtout pour les bases de données relationnelles ou bien «cost-based».
- **strategies** : attribut de type texte qui décrit les stratégies utilisées pour avoir un partitionnement vertical optimal.

Cette classe est associée à la classe « QueryPattern » du niveau workload puisque le choix du critère de partitionnement dépend de ces éléments.

### 3.3.3 Modèles de données

#### *3.3.3.1 Classe DataModelType*

La classe «DataModelType» est utilisée pour décrire les différents types de modèles qu'on trouve dans les systèmes de gestion de lacs. Elle a 3 attributs :

- **schema_kind_of** qui décrit le type de schéma de données supporté par le type de modèle. Un schéma est une spécification qui décrit comment les données sont structurées et utilisées. Les valeurs de cet attribut sont : «rigide», «flexible» et «non-applicable».

- **data_structure** qui est de type chaîne de caractères et qui modélise le fait qu'un type de modèle de données supporte un certain type de structures de données bien déterminé.
- **data_connectivity** qui indique à quel degré le modèle de données supporte les données reliées. Ses valeurs possibles sont «high», «low», «non-applicable».
- **normalization** qui renseigne si un type de modèle de données favorise la normalisation. «non-applicable», «denormalization is advised», «supported», «required».

La classe DataModelType possède 4 sous-classes, représentant chacune un type de modèle de données.

#### *3.3.3.2 Classe AggregateDataModel*

La classe « AggregateDataModel » caractérise les modèles qui se réfèrent au concept d'agrégation [SF12] issu du domaine de la conception pilotée par les modèles. Un agrégat est une collection d'attributs/éléments reliés et traités comme une seule unité. Quand les données ont besoin d'être manipulées comme un seul objet alors un modèle de données de type agrégation est le plus approprié pour représenter l'information. Un agrégat simplifie aussi la gestion de la cohérence et permet d'exécuter des opérations atomiques sur des attributs de données qui sont logiquement reliés. (Cf. le modèle de cohérence de MongoDB).

Cette classe a un attribut **dimension**. Sa valeur numérique de 1 à n symbolise le niveau de composition de l'enregistrement de données d'un modèle de données (cf classe « DataModel »).

Par exemple, HBase se base sur un modèle de données tridimensionnel. Ses attributs de données sont représentés dans l'espace de la 3ème dimension. Cette représentation est due au versionnement des attributs de données par horodatage.

Un autre exemple est celui de MongoDB qui utilise l'imbrication pour créer un modèle de données de dimension n.

Elle a aussi un attribut **aggregate** qui est de type texte et qui correspond au constructeur qui représente un ensemble de données qui vont être accédés ensemble. Un exemple de valeur pour cet attribut est « document » pour les modèles de données basés sur les documents, «value» pour les modèles de données clé valeur et «column family» pour les modèles de données basés sur les colonnes.

La classe « AggregateDataModel » a 3 instances possibles :

- L'instance « keyValueDataModel » : caractérise un modèle de données simple mais puissant basé sur le concept de paires de données clés-valeur. Son point fort est que son constructeur valeur (l'aggrégat) est de type «blob» (binary large object). Une valeur/objet est retrouvée en utilisant une clé. Dans ce type de modèle l'agrégat n'est pas typé (par exemple dans HBase il n'y a pas de types de données pour le champ valeur). Le point fort des modèles de données de ce type et l'avantage de cette opacité est la possibilité de supporter des ensembles de données non structurés : on peut stocker tout ce qu'on veut dans l'agrégat. Le lac de données peut toutefois imposer des règles de limite de taille. Le deuxième avantage majeur de ce modèle est que le stockage des données sans typage améliore la performance d'accès. Néanmoins l'interface de manipulation liée à ce type de modèle de données est de type

« keyBasedQuery », ce modèle d'interrogation de données possède des opérations logiques limitées en termes de manipulations complexes (cf métadonnées « access API and query »). Une conséquence de ces caractéristiques, est l'utilisation des modèles de ce type par des systèmes de caching par exemple. Une définition de l'instance « keyValueDataModel » peut être :

> (keyValueDataModel, AggregateDataModel( schema_kind_of : «non-applicable»,
> data_structure : «All»,
> dimension : 1,
> data_connectivity : «non-applicable»,
> aggregate : «value»,
> normalization : «non-applicable»,
> ))

- L'instance « columnBasedDataModel » représente un type de modèle qui est utilisé pour modéliser l'information comme des colonnes de données reliées entre elles (familles de colonnes). Un objet de ce modèle peut être comparé à un tableau ayant pour chaque ligne une paire de clés-valeurs. Les valeurs regroupent un ensemble de colonnes où chaque colonne est un triplet constitué par un nom de colonne, une valeur et une valeur d'Horodatage (**timestamp**). UNe définition de l'instance « columnBasedDataModel » peut être :

> (columnBasedDataModel, AggregateDataModel(schema_kind_of : «flexible»,
> data_structure : «tabular data»,
> data_connectivity : «non-applicable»,
> dimension : 3,
> aggregate : «column family»,
> normalization : «denormalization is advised»
> ))

- L'instance « documentBasedDataModel » est utilisée pour décrire des structures de données comme des paires de clés-valeurs dont la structure de la valeur (un document) est connue. En effet, le type de la valeur d'un champ est généralement atomique, document, collection ou ensemble vide. Elle peut être définie par :

> (documentBasedDataModel, AggregateDataModel(schema_kind_of : «flexible»,
> data_structure : «hierarchie tree data structure»,
> data_connectivity : «low»,
> dimension : n,
> aggregate : «document»,
> normalization : «denormalization is advised»
> ))

Le système MongoDB a un modèle de données agrégat. Son modèle de données est en même temps clé-valeur et basé sur les documents. L'instance de la classe « DataModel » pour le système MongoDB

aura une association avec les instances : « documentBasedDataModel » et « keyValueBasedDataModel ». Les autres propriétés de cette instance sont détaillées dans la section décrivant la classe « DataModel » (cf. section 3.3.3.6).

#### 3.3.3.3 Classe GraphDataModel

La deuxième classe qui hérite de « DataModelType » est nommée « GraphBasedDataModel ». Elle est utilisée pour décrire un modèle de données en graphe idéal pour la capture de données composées de relations complexes tels que les réseaux sociaux, les préférences de produit ou les règles d'éligibilité.

Elle a un attribut **kind_of** utilisé pour donner plus de précisions du modèle en tant que :

o « property_graph » qui est un type de modèle de données qui donne des concepts pour représenter les données sous forme de graphes de nœuds avec des propriétés. Les nœuds sont liés à des relations pouvant être dirigées.

o « triplet_graph » qui est un type de modèle de données qui donne des concepts pour représenter les données sous forme de graphes de triplets (sujet, prédicat, objet).

Une instance de la classe « GraphBasedDataModel » décrivant le modèle de données basé sur les graphes tel que RDF est défini comme suit :

> (RDF, GraphBasedDataModel ( schema_kind_of: «non-applicable»,
> data_structure: «triplet»,
> data_connectivity : «high»,
> kind_of : «Triplet_graph»,
> normalization : «supported»,
> ))

#### 3.3.3.4 Classe FileDataModel

La classe « FileDataModel » est utilisée pour décrire un type de modèles de données offrant le plus haut degré de liberté. Aucune structure n'est imposée, les données sont stockées dans leurs formats bruts. Ainsi, la gestion de ces données nécessite généralement des outils supplémentaires (cf Section 4. Métadonnées «Api d'accès et requêtes»). Prenons comme exemple la gestion des datasets dans HDFS.

Ce système n'impose aucune directive de conception de schéma mais la littérature [Gro15] propose quelques bonnes pratiques. Ces pratiques concernent surtout le nommage d'un jeu de données à stocker dans HDFS et elles sont représentées dans l'attribut **key_design** dont le type est un ensemble de textes.

#### 3.3.3.5 Classe StructuredDataModel

La dernière sous classe de « DataModelType » est « StructuredDataModel » qui est utilisée pour décrire un type de modèle de données offrant des constructeurs de haut niveau (comme les agrégats, ensembles, listes, tuples, références) pour décrire des données. Le modèle de données relationnel, le modèle de données orienté objet, le modèle de données XML ... sont des instances de cette classe. Les propriétés de l'exemple d'instance « relationalDataModel » est :

> (relationalDataModel, StructuredDataModel ( schema_kind_of : «rigide»,
> data_structure : «tabular data»,
> data_connectivity : «high»,

normalization : «required»,
))

#### 3.3.3.6 Classe DataModel

Cette classe regroupe, par le biais de plusieurs attributs, les constructeurs d'un modèle de données. Elle offre plus d'informations sur la façon dont les données sont nommées, consultées, groupées, structurées dans un système de stockage de données utilisant un modèle de données spécifique.

Elle possède les attributs :

- **data_space**: attribut de type liste qui spécifie la façon dont est nommé l'espace dans lequel résident les données. Il peut être vu comme une connotation de domaine d'application ou un paquet (en adoptant le jargon Java par exemple). Dans MongoDB, l'espace de noms est appelé «database».
- **data_collection** (groupe de données ou en d'autres termes collection de données) : attribut de type ensemble de texte qui spécifie la manière dont les objets de données sont logiquement liés dans des ensembles. Dans le modèle de données relationnel, cet attribut aura comme valeur {«table»}. Dans le modèle de données de HBase il aura la valeur {«table»,«column family»}.
- **data_constraints** : attribut de type ensemble de textes qui décrit la possibilité de typage et de définition de contraintes sur les données. Par exemple, dans le système Neo4J, on a un constructeur de contraintes «Exists» qui indique qu'un attribut doit nécessairement exister dans tous les enregistrements d'une collection particulière dans le cadre d'une fonctionnalité de définition de schéma. Cet attribut peut avoir comme valeur l'ensemble vide.
- **data_types** : attribut de type ensemble de textes qui renseigne sur les types supportés par un modèle de données bien déterminé (par exemple : les systèmes qui ne supportent pas les types « timestamps » voire même les types numériques sont susceptibles d'être rejetés par le système de recommandation dans le cadre de cas d'utilisation de gestion d'ensembles de données de séries temporelles). Cet attribut peut avoir comme valeur l'ensemble vide.
- **data_record** (enregistrement de données ou en d'autres termes objet de données) : cet attribut est de type « DataRecord » . Il indique le type de constructeur utilisé pour agréger différents attributs de données. En termes d'exemple, nous citons l'enregistrement de données de HBase qui est appelé «row». La structure de la classe de cet attribut est composée de :
  - **name** : une valeur de type chaîne de caractères représentant le nom d'un enregistrement de données dans le modèle de données cible
  - **identifier** : une valeur de type chaîne de caractères représentant l'identifiant unique d'un enregistrement de données (s'il existe). Il prend une des valeurs suivantes : «ordered», «auto-generated», «explicit definition»,«none». En termes d'exemples, on cite le modèle de données de type «triple data model» qui a comme valeur «none» pour cet attribut et celui du système HBase qui a la valeur «ordered».
  - **content_type**: avec des valeurs correspondant à des types de données tels que: «binary», «triplet», «structured»

- **data_attribute** (attribut de données) : cet attribut décrit le concept le plus élémentaire dans un modèle de données (par exemple une colonne (column) dans le modèle de données de HBase ou un champ (field) dans celui de MongoDB). Il est en général représenté par le couple valeur ($A_i$ :$V_i$) dans les modèles de données où $A_i$ est un champ de données ou attribut. Il est de type « DataAttribute ». La classe « DataAttribute » a les attributs suivants :
    - **name** : une valeur de chaîne représentant le nom d'un attribut de données dans le modèle de données cible
    - **attribute_kind_of** ayant comme valeurs possibles : (i) «predefined» (par exemple: le cas du schéma de jeu de données pour le modèle relationnel ou celui des clés de colonnes prédéfinies dans la famille de colonnes pour le modèle de données basé sur une colonne) typage fort et tous les attributs doivent figurer dans tous les enregistrements d'une collection données. (ii) «strong typing» typage fort , (iii) «no typing» pas de typage ,(iv) «flexible definition» l'espace des attributs possibles est défini au préalable (par exemple création des nom des colonne dans des column families) mais à l'insertion des données l'utilisateur a une liberté de choisir les attributs de son enregistrement. (v) «optional definition» (dans certains data store il est optionnel de définir un schéma et spécifier les attributs qui doivent nécessairement exister pour chaque enregistrement, par exemple le cas du système Neo4J) et (vi) «dynamic definition» : aucune prédéfinition . Les valeurs sont spécifiées lors de l'opération d'insertion de données et les attributs ne sont pas forcément les mêmes dans tous les enregistrements
    - **value_kind_of** avec des valeurs possibles: «atomic», «structured», «collection»
    - **value_ is_nullable** : est un booléen qui symbolise si le modèle de données accepte des valeurs nulles
    - **value_ is_ordered** : un booléen qui symbolise si les valeurs peuvent être ordonnées ou non
    - **value_is_versionned** : est un booléen qui symbolise si les valeurs sont versionnées par horodatage ou non (par exemple, cas du modèle de données Hbase)

En continuant avec l'exemple d'instanciation du système MongoDB (cf 3.5.2), le modèle de données de

MongoDB est représenté par l'instance :

(MongoDBDataModel, DataModel( data_space : {«database»},
data_collection : {«collection»},
data_constraints : «validator operator»,
data_record.name : «document»,
data_record.identifier : «auto-generated»,
data_record.content_type : «structured»,
data_attribute.name : «field»,
data_attribute.attribute_kind_of : «dynamically defined»,

data_attribute.value_kind_of : «structured»,
data_attribute.value_ is_nullable : True,
data_attribute. value_ is_ordered : True,
data_attribute. value_ is_versionned : False))

Avec ce type de représentation on peut répondre à la requête : donner les data stores qui ont un type de modèle de données document et qui ont des attributs de données de type «structured» (pour la recherche des systèmes qui supportent les types complexes). Parmi les réponses possibles on aura l'instance MongoDB citée dans notre exemple.

##### *3.3.3.7 Classe DataSetStructure*

La classe DataSetStructure caractérise les structures de données des ensembles de données et leurs utilisations standards dans les applications métiers. Elle est utilisée dans les classes des trois niveaux et contribue à associer à chaque cas d'utilisation le modèle de données le plus adéquat. Elle définit les attributs suivants :

- **data_set_structure_category** : attribut de type texte qui indique la catégorie des structures de données représentées dans cette classe.
- **data_set_structures** : attribut de type ensemble de textes qui regroupe les types abstraits relatifs à une structure de données bien déterminé.
- **use_case** : attribut de type ensemble de textes qui renseigne sur les cas d'utilisations standard d'une structure de données particulière.
- **non_supported_operations_properties** : attribut de type ensemble de textes qui représente les caractéristiques des opérations non supportées par la structure de données en question.

Les propriétés de l'exemple d'instance « treeDS » de cette classe sont [SF12] :

(treeDS, DataSetStructure( data_set_structure_category: «hierarchical tree structure»,
data_set_structures: {«map», «collection», «scalar values»},
use_case: {«Event logs», «CMS», «blog», «web analytics», «e-commerce»},
non_supported_operations_properties: {«complex transactions»,
«querying aggregate with varying structures»}))

### 3.3.4 Interfaces de programmation et langages d'accès

Les métadonnées « Access API & languages » permettent de caractériser les APIs et les langages de requêtes des systèmes de gestion de lacs de données. Elles détaillent les opérations logiques supportées pour chaque système ainsi que les problèmes qu'elles visent à résoudre. Elles nous permettent aussi de renseigner les opérations de prétraitement qui permettent d'optimiser l'accès aux données. Par exemple, en sachant qu'une liste a été triée au préalable, nous pourrons effectuer une opération de recherche en temps logarithmique avec la technique de recherche binaire. Cette information peut être intégrée aux métadonnées des ensembles de données pour décrire si les données qui vont être accédées sont ordonnées ou pas et pour recommander sur le type de traitement en se basant sur cette métadonnée de l'ensemble des données.

Cette section décrit les opérateurs (logiques) d'un plan de requêtes. Les opérations physiques correspondantes sont présentées dans la sous-section 3.3.6.

Le diagramme de classe de la figure 3.10 présente les classes utiles à la description des systèmes d'interrogation des données ainsi que leurs modèles associés et leur spécificité sous forme de langages et d'api d'accès.

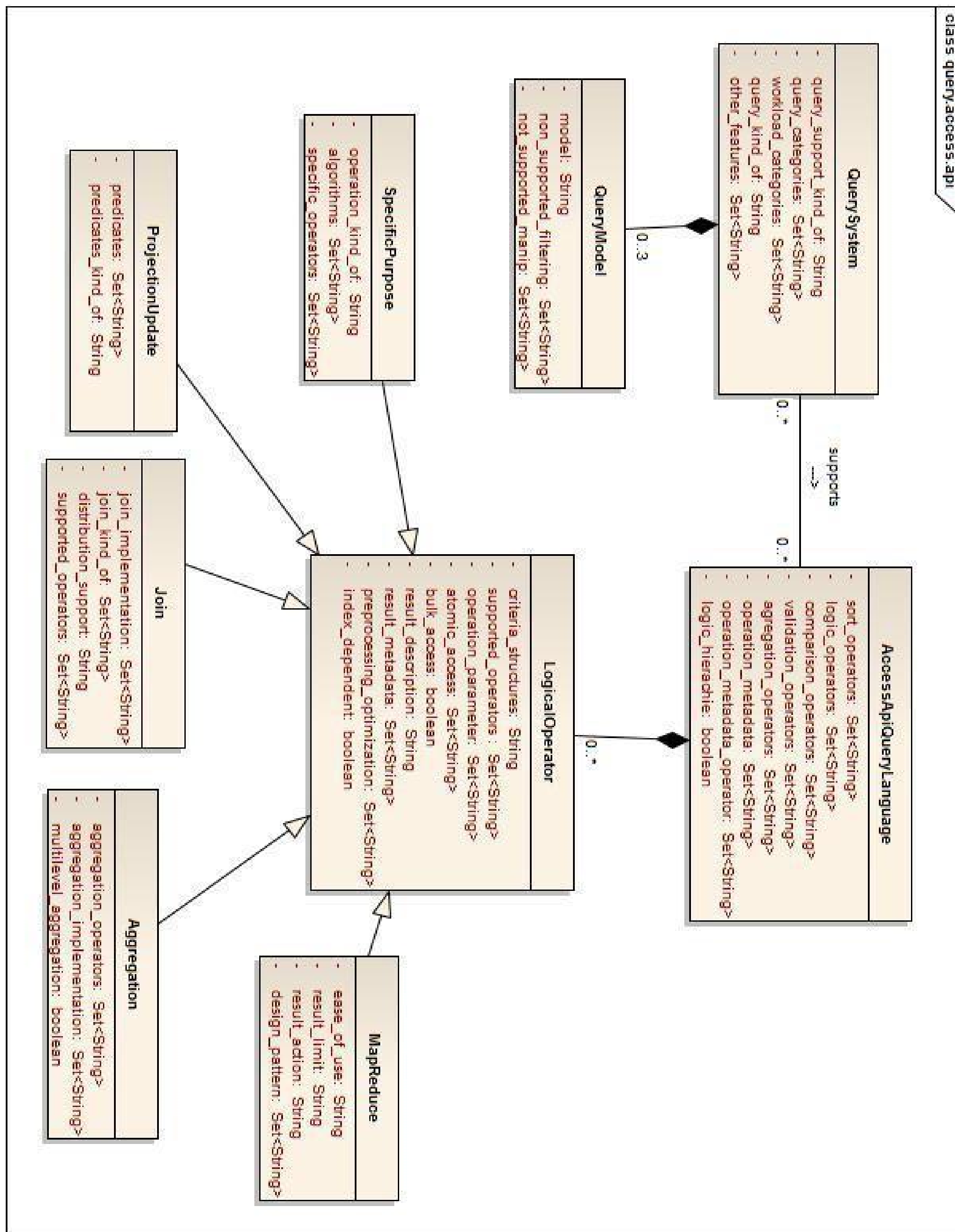



*Figure 3.10 Diagramme de classe des «Requêtes et API d'accès»*

La classe qui représente les systèmes d'interrogation des données est « QuerySystem ». Elle a un lien de composition et de cardinalité 0..3 avec la classe « QueryModel » pour refléter que chaque système d'interrogation de données possède au moins un modèle de requête. Elle est en association avec la classe « AccessApiQueryLanguage ». L'association «supports» de cardinalités 0..* décrit que ces systèmes peuvent exposer plusieurs API ou langages de requête. La classe « AccessApiQueryLanguage » est à son tour en lien de composition avec la classe « LogicalOperator »

qui est héritée par plusieurs classes qui représentent différentes catégories d'opérations logiques sur les données. La suite de cette section détaille ces éléments.

#### *3.3.4.1 Classe QuerySystem*

La classe « QuerySystem » décrit les propriétés des systèmes (/modules) d'interrogation des données des lacs. Elle a comme attributs :

- **query_support_kind_of** : attribut de type texte qui indique si la manipulation de données est supportée nativement par le système de gestion de données ou bien si elle l'est par un système externe (3rd party).
- **query_categories** : attribut de type ensemble de textes qui décrit les catégories de requêtes supportées. Il comprend comme valeurs : «CRUD», «geospatial», «sql», ...
- **workload_categories** : attribut de type ensemble de textes qui renseigne les types de workload supportés par le système. Son champ de valeurs est compris dans la liste suivante : «batch», «stream», «interactif».
- **query _kind_of** : attribut de type ensemble de textes qui indique si l'interrogation des données est faite via une API ou bien un langage de requête déclaratif ou bien HTTP/REST-based ou bien un langage de script (par exemple le système Pig). Ses valeurs sont comprises dans la liste suivante : «API», «declarative query language», «HTTP/REST-based» ou bien «script language».
- **other_features** : attribut de type ensemble de textes qui renseigne sur des caractéristiques spécifiques sur les requêtes que le système offre. Son champ de valeurs est défini comme suit : « Cursor-based Queries», «JOIN-style queries», «Complex Data Types manipulation», «Restrict Query Result Set Size», «Key Matching Options», «Triggers», «View», «PreparedStatement», «Data Object Expiry»

Une instance de cette classe pour MongoDB est définie comme suit :

(MongoDBAccessApiModule, QuerySystem(
query_support_kind_of : «natif»,
query_categories : {«CRUD»,«geospatial»},
workload_categories : {«interactive»,«batch»},
query _kind_of: «Api»,
other_features: {«Cursor-based Queries»,«JOIN-style queries»,«Complex Data Types manipulation»,«Restrict Query Result Set Size»,«Key Matching Options»,«View»,«udf»,«Data Object Expiry»}
))

#### *3.3.4.2 Classe QueryModel*

La classe « QueryModel » informe sur les modèles de référence d'une requête supportés par les systèmes. Ses attributs sont :

- **model** : attribut de type ensemble de textes qui peut avoir une ou plusieurs des valeurs suivantes : «key based», «pattern based», «connection based».
- **non_supported_filtering** : attribut ensemble de textes qui renseigne sur les catégories d'opérations non supportées par le modèle. Ses valeurs sont comprises dans la liste suivante : « equality », « range » (bornée ou infini d'un côté ou des deux côtés), …

- **non_supported_manip**: attribut ensemble de textes qui renseigne sur les catégories d'opérations de manipulation non supporté par le modèle.

Les instances de cette classe : keyBasedQueryModel, aggregateBasedQueryModel, connectionBasedQueryModel seront détaillées ultérieurement.

#### *3.3.4.3 Classe AccessApiQueryLanguage*

Cette classe décrit les opérations supportées par API ou un langage relatif à un système du lac de données. Ses attributs sont :

- **sort_operators** : un attribut de type ensemble de textes qui représente les types supportés d'ordre du résultat par le système. Ses valeurs peuvent être : «ascending», «descending»
- **logic_operators** : un attribut de type ensemble de textes qui représente les opérateurs supportés par le système pour réaliser les opérations logiques. Il peut avoir comme valeur : «AND», «OR», …
- **logic_hieararchy**: un attribut booléen qui indique si la hiérarchie des opérateurs logiques sont supportés.
- **comparison_operators** : un attribut de type ensemble de textes qui représente les opérateurs supportés par le système pour réaliser les comparaisons.
- **validation_operators** : un attribut de type ensemble de textes qui représente les opérateurs supportés par le système pour la validation des données.
- **agregation_operators**: un attribut de type ensemble de textes qui représente les opérateurs supportés par le système pour l'agrégation des données.
- **operation_metadata** : attribut de type ensemble de textes qui regroupe les métadonnées qu'un utilisateur peut spécifier sur une requête.
- **operation_metadata_operator** : attribut de type ensemble de textes qui renseigne les opérateurs d'interrogation et de modification des métadonnées sur les requêtes.

Ci-dessous un exemple d'instance de cette classe :

(MongoDBAccessAPI, AccessApiQueryLanguage(
sort_operators:{«ascending»,«descending»},
logic_operators : {«AND», «OR», «NOT», «NOR»},
comparison_operators : {«<=», «<», «=», «=>», «>», «<>», «in», «not in»},
logical_hierarchy : True, validation_operators : {«validate»},
agregation_operators :{«count», «distinct», «group», «mapReduce»},
operation_metadata : {«comment»},
operation_metadata_operator : {«meta», «comment[6]» }
))

[6] Opérateur utile pour les fonctions d'administration. Il enregistre des commentaires sur les requêtes dans le journal de profil d'un utilisateur. En ajoutant un commentaire, on rend les données de profil plus faciles à interpréter et à tracer.

#### 3.3.4.4 Classe LogicalOperator

La classe « LogicalOperator » décrit les composants principaux du plan logique d'une requête. Elle est héritée par les différentes classes qui décrivent les langages de requêtes et les api d'accès ; elle a comme attributs :

- **filter_structures** : un attribut de type ensemble de textes qui décrit les structures de données supportées comme attributs de la requête. Un ensemble vide pour cet attribut signifie que le système ne supporte pas des attributs structurés pour ses requêtes
- **supported_operators** : un attribut de type ensemble de textes qui décrit les catégories d'opérateurs supportés (exemple : comparaison, filtrage).
- **operation_parameter** : un attribut de type ensemble de textes qui permet de décrire les paramètres spécifiques que le système offre pour l'opération en question
- **bulk_access** : un attribut de type booléen qui indique si l'opération accepte l'accès en masse.
- **result_description** : un attribut de type texte qui représente le modèle de données du résultat de l'opération.
- **result_metadata** : attribut ensemble de textes qui regroupe les nom des métadonnées de type statistiques ou information sur le statut de la requête exécutée.
- **index_dependent** : un attribut de type booléen qui indique si une opération nécessite une étape d'indexation au préalable (par exemple les requêtes CQL)
- **preprocessing_optimization** : un attribut de type ensemble de textes qui indique les techniques de prétraitement préconisées pour améliorer l'exécution d'une requête.

#### 3.3.4.5 Classe ProjectionUpdate

La classe « ProjectionUpdate » décrit les composants principaux des requêtes de projection ou de modification des données. Elle définit les attributs suivants :

- **predicates** : un attribut de type ensemble de textes qui décrit les noms des prédicats des opérateurs supportés.
- **predicates_kind_of** : un attribut de type texte qui indique si la requête est de type «projection» ou «update»

Exemple d'instances de cette classe pour le système MongoDB :

(MongoDBInsert, ProjectionUpdate (
predicates : «insertMany,insertOne»,
predicates_kind_of : «update»,
creteria_structures : «Json (document)»,
supported_operators : {},
operation_parameter : {«write_concern», «ordered»},
bulk_access : true,
atomic_access : {«document»},
result_description : «None»,
result_metadata : {«nInserted», «writeConcernError», «writeConcernError»}
preprocessing_optimization : {}
))

#### 3.3.4.6 *Classes «Analytic and aggregation»*

Les classes « Aggregation MapReduce », « SpecificPurpose » et « Join » décrivent les fonctionnalités de traitements pour l'analyse de données offertes par les systèmes d'interrogation. Ces classes sont en lien de composition avec la classe « AccessApiQueryLanguage » avec la cardinalité 0..*. Elles sont décrites ci-dessous.

- Classe Aggregation

Cette classe représente la fonctionnalité permettant de combiner et grouper plusieurs ensembles de données pour produire un seul résultat. Elle a 3 attributs :

- **aggregation_operators** : un attribut de type ensemble de textes qui renseigne sur les fonctions d'agrégation (par exemple «sum», «group by», …)
- **multilevel_aggregation** : un attribut de type booléen qui indique la possibilité de faire une cascade d'agrégation. Le système MongoDB offre par exemple la possibilité de faire du mutilevel aggregation avec la fonctionnalité « aggregation pipeline ». Cette fonctionnalité inspirée du concept de «pipelines de traitement de données» permet à l'utilisateur de d'exécuter une séquence d'opérations d'agrégation de données. L'instance des requêtes de ce type pour MongoDB a donc la valeur true pour cet attribut.
  **Remarque** : dans le système MongoDB, cette fonctionnalité est plus performante que MapReduce. En effet, le framework d'agrégation de MongoDB est implémenté en C++ donc ses applications sont sujet à une exécution de code compilé. Par contre le framework MapReduce de ce système est implémenté en Javascript et il est limité par les performances de l'exécution d'un code interprété.
- **aggregation_implementation** : un attribut de type ensemble de textes qui regroupe les types d'implémentations de l'agrégation. Par exemple, dans MangoDB la valeur de cet attribut est : «agregation pipeline».

- Classe MapReduce

La classe « MapReduce » décrit les systèmes qui utilisent le paradigme de programmation MapReduce. Ses attributs sont :

- **design_pattern**: un attribut de type ensemble de textes qui décrit l'ensemble des modèles de problèmes que le paradigme MapReduce résout. Il a comme valeurs : «top n», «structure pattern», ...
- **ease_of_use** : un attribut de type texte qui indique le degré de développement nécessaire pour utiliser ce système. Il peut avoir comme valeurs possibles : «high» ou bien «limited»
- **result_limit** : un attribut de type texte qui décrit les limites sur le résultats des requêtes MapReduce. Par exemple dans l'implémentation MapReduce de MongoDB, le résultat est soumis à une limite sur la taille (exprimée dans la métadonnée de statistiques : document « size limit » est égale à 16 megabytes dans la version actuelle).
- **result_action** : un attribut de type texte qui indique les opérations possibles à faire sur les résultats du traitement MapReduce. Les valeurs possibles pour cet attribut sont : «merge», «replace», «reduce» ou «none»

### Classe SpecificPurpose

Cette classe décrit les opérateurs logiques qui sont associés à des cas d'utilisations spécifiques (par exemple analyse de données géo spatiales) ou bien qui traitent des structures de données particulières (par exemple des graphes de données). Ses attributs sont définis comme suit :

- **operation_kind_of** : un attribut de type texte qui décrit le type de ces opérations logiques spécifiques. Il peut avoir comme valeurs : «spacial», «temporal», «bitemporal» ou «graph»
- **algorithms** : un attribut de type ensemble de textes qui indique les problèmes algorithmiques supportés par ce modèle d'opération logique. Cet attribut peut avoir une ou plusieurs de ces valeurs : «nearest neighbour» pour l'exemple des requêtes de type spatial ou bien «traversal» ou «inference» pour le type de requêtes basées sur les graphes.
- **specific_operators** : attribut de type ensemble de textes qui renseigne les opérateurs pour une utilisation spécifique.

### Classe Join

La classe « Join » représente les opérations logiques de jointure. Elle a comme attributs :

- **join_implementation** : dans les systèmes de gestion de lacs de données l'opération de jointure n'est pas toujours supporté. Or il existe certaines stratégies pour contourner cette problématique et implémenter la jointure issue du domaine de modélisation des données, d'où le rôle de cet attribut de type ensemble de textes dans la classe « Join ». Les valeurs possibles pour cet attribut sont : «nested collections»,«list structures»
- **join_kind_of** : un attribut de type ensemble de textes qui décrit les types de jointure supportés. Par exemple «equijoin»...
- **distribution_support** : attribut de type texte qui indique si l'opération de jointure est possible dans le cadre d'un schéma de distribution des ensembles de données. Par exemple dans MongoDB l'opérateur «lookup» permet d'exécuter une opération Outer Join sur les collections qui n'ont pas subi de partitionnement horizontal (sharding)
- **supported_operators** : attribut de type ensemble de textes qui renseigne les opérateurs supportés.

L'instance de cette classe, soit l'implémentation de l'opération de jointure dans le système MongoDB est décrite par :

(MongoDBJoin,Join (
join_implementation :{ «nested collections»,«list structures»,«db_ref»,«lookup operator»},
join_kind_of:{},
distribution_support : «no partitioning»,
supported_operators :{«lookup»},
creteria_structures : «Json (document)»,
supported_operators : {},
operation_parameter : {«write_concern», «ordered»},
bulk_access : true,
atomic_access : {«document»},
result_description : «None»,

result_metadata : {«nInserted», «writeConcernError», «writeConcernError»}
preprocessing_optimization : {}
))

### 3.3.5 Stockage des données

Le modèle de stockage des données au niveau physique est différent de celui du niveau logique. En effet, les données sont stockées dans un format non typé et subissent des traitements de compression et de sérialisation qui contribuent à l'amélioration de la performance et l'utilisabilité des systèmes de gestion d'un lac de données. Les métadonnées «data storage» caractérisent le modèle du composant de stockage d'un système et sont présentées dans le diagramme de la figure 3.AA

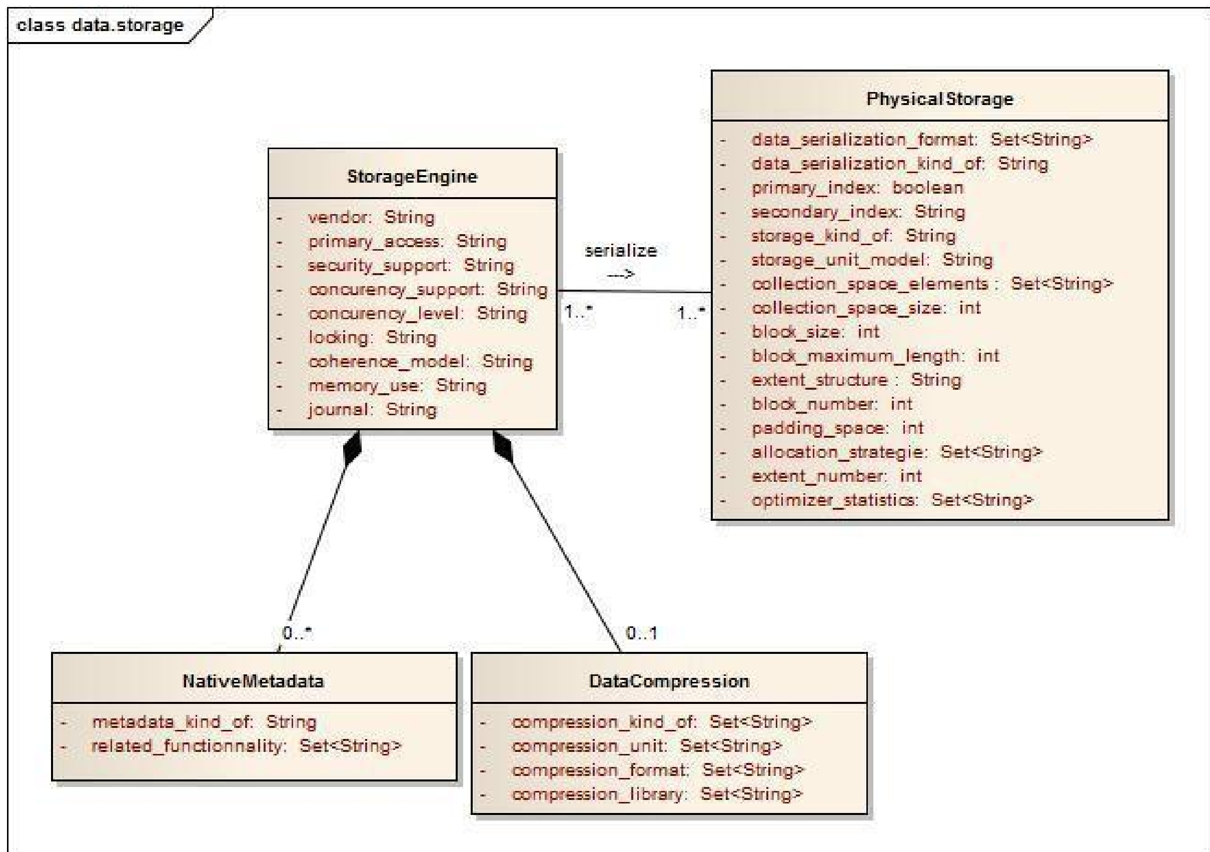



*Figure 3.11 Diagramme de classe des métadonnées «data storage»*

Le diagramme de classe des métadonnées du stockage des données présente deux classes principales « PhysicalStorage » et « StorageEngine » ainsi que leurs composants. Elles sont liées par l'association «serialize» dont les cardinalités sont 1..*. La classe « PhysicalStorage » décrit globalement le stockage physique des données dans un système. Elle concerne d'une part la sérialisation sous forme de fichiers et d'autre part les éléments de stockage unitaires. La classe « StorageEngine » présente les caractéristiques spécifiques aux moteurs de stockage qui sont susceptibles d'intéresser l'utilisateur et l'amener à choisir un moteur particulier pour un système de gestion de données. Cette classe utilise les classes « DataCompression » et « NativeMetadata » qui détaillent respectivement les propriétés relatives à la compression et à la gestion des métadonnées faite en interne par le système.

#### *3.3.5.1 Classe PhysicalStorage*

La classe « PhysicalStorage » définit les informations sur le stockage physique des données. Elle décrit les formes de sérialisation par les attributs suivants :

- **data_serialization_format** : un ensemble de textes qui indique les formats physiques des données supportées par le système. Ci-dessous on donne des exemples de valeurs pour cet attribut qu'on peut trouver dans des instances de systèmes différents :
  - pour le système Hadoop HDFS : « avro », « parquet », « orc »
  - pour le système MongoDB : « bson »
  - format polyvalents : « EXT4 »
- **data_serialization_kind_of** : un attribut de type texte qui renseigne les type de fichiers manipulés dont les valeurs sont par exemple : «binaire», «texte», ...
- **storage_kind_of** : un attribut de type texte qui indique si le stockage est «in_memory» ou «on_disk»
- **primary_index** : un attribut booléen qui représente la possibilité de stockage des index primaires par le système.
- **secondary_index** : un attribut de type texte qui indique si l'indexation est supportée et s'il est fait automatiquement par le système ou manuellement par l'utilisateur. Les valeurs possibles pour cet attribut sont : «automatic», «manual» ou «none».

Les autres attributs de cette classe décrivent le modèle physique de stockage de données. En effet certains systèmes de stockage physique utilisent la technique de «blockage» pour la gestion des données au niveau physique qui consiste à manipuler des séquences de bits ou d'octets comme éléments unitaires de stockage physique nommés «blocks». Un ensemble de «blocks» contigus alloués sur disque est appelé «extent». Ce constructeur physique permet de présenter une entité logique composite -- qui correspond généralement à l'entité data collection du modèle de données (cf. classe «DataModel») ou bien à un index. Les «extents» peuvent eux-mêmes être regroupés à leur tour dans des segments qui ne sont pas forcément contigus. Entre le niveau physique et le niveau logique il existe une autre couche d'abstraction qui se nomme les «tablespace» dans le modèle relationnel. Dans MongoDB cette couche est référée comme «namespace» (index namespace, collection namespace). On appelle ce niveau intermédiaire : collection space3.

Les attributs qui permettent de caractériser le modèle physique de stockage sont :

- **storage_unit_model** : un attribut de type texte qui renseigne sur l'unité élémentaire du modèle de donnée qui va être stockée sur le support physique.
- **collection_space_elements** : un ensemble de textes des éléments gérés par le système au niveau physique . Par exemple, une collection « space » correspond dans le modèle relationnel au tablespace et les valeurs peuvent être : « index », « table », ...
- **collection_space_size** : c'est un attribut de type numérique qui renseigne sur la taille d'un «collection space»
- **block_size** : un attribut de type numérique pour indiquer la taille de l'unité de stockage alloué sur disque
- **block_maximum_length** : un attribut de type numérique qui indique la taille maximale de l'unité de stockage allouée sur disque
- **extent_structure** : attribut de type texte qui indique la structure de données de stockage ; par exemple dans MongoDB sa valeur est «linked list»
- **block_number** : attribut numérique qui renseigne sur le nombre de blocks dans un extent

- **padding_space** : attribut numérique qui renseigne sur la taille de l'espace libre entre les blocks de données.
- **allocation_strategie** : attribut de type ensemble de textes qui représente les différents stratégies d'allocation d'espace disque pour une unité de stockage. Par exemple, les valeurs possibles pour MongoDB de cet attribut sont :
    - «Power of 2 Sized Allocations» : une stratégie d'allocation qui favorise les workloads «insert/update/delete». Il s'agit de la valeur par défaut depuis la version 3.0 de MongoDB
    - «No Padding Allocation Strategy» : une stratégie d'allocation qui favorise les workloads des collections sans mise à jour.
- **extent_number** : attribut numérique qui renseigne sur le nombre d'extents dans un segment
- **optimizer_statistics** : attribut de type ensemble de textes qui comprend la liste des métadonnées statistiques sur les constructeurs du modèle de données.

Généralement un système de gestion de données supporte plusieurs moteurs de stockage. Dans la section suivante, on décrit les métadonnées pour ces moteurs et leurs propriétés.

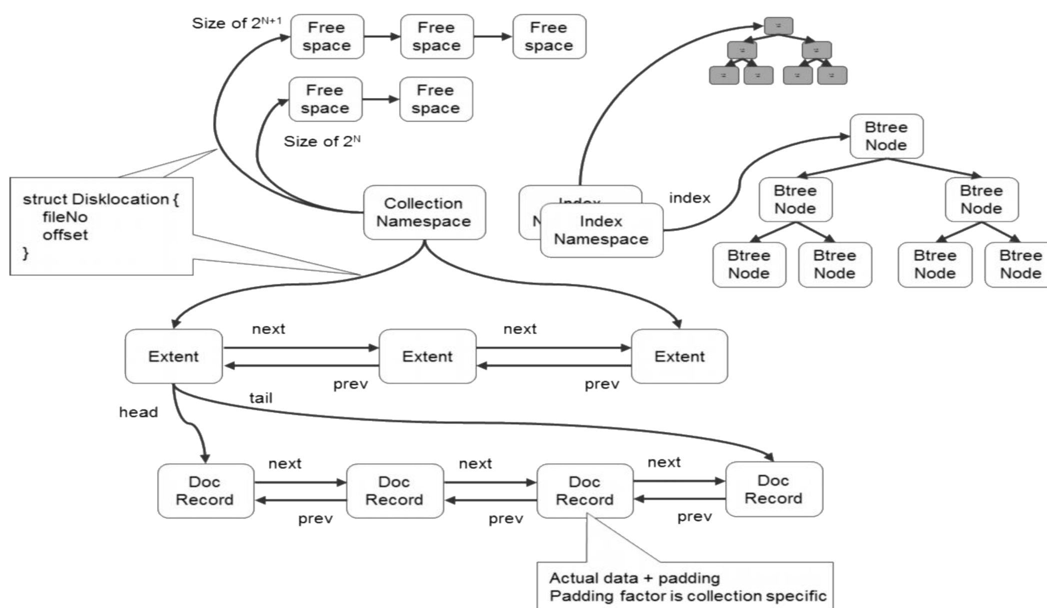



*Figure 3.12 Modèle physique de MongoDB [Ho12]*

#### 3.3.5.2 Les classes «Storage engine»

Dans cette section, nous décrivons les classes « Storage Engine » et « Data compression » décrivant les moteurs de stockage.

- Classe StorageEngine

Cette classe correspond au concept du moteur de stockage physique des données. Ces moteurs sont caractérisés par un ensemble de propriétés qu'on exploite sous la forme des attributs suivants :

- **vendor** : un attribut de type texte qui indique le fournisseur du moteur de stockage. Ce dernier peut avoir comme valeur «native» ou bien le nom d'un fournisseur tiers.
- **primary_access** : un attribut de type texte qui indique le type d'accès primaire. Il prend comme valeur : «hash based» ou bien «index based»

- **security_support** : attribut de type texte qui indique les caractéristiques supportées par le moteur de stockage pour garantir la sécurité. Il représente par exemple des informations sur le chiffrement.
- **concurency_support** : attribut de type texte qui renseigne sur la technique de gestion des accès concurrents offertes par le moteur de stockage par exemple dans wiredTiger de MongoDB l'accès concurrent est géré par MVCC.
- **concurency_level** : attribut de type texte qui sert à représenter l'unité élémentaire du modèle de données pour l'accès concurrent. Il s'agit généralement de la valeur de l'attribut **data_record** de la classe « DataModel ». Par exemple, dans le moteur de stockage de MongoDB wiredTiger la concurrence est réalisée au niveau Document.
- **locking**: attribut de type texte qui caractérise le niveau de verrouillage assuré par le moteur. Sa valeur dans le moteur MongoDB est «write locking»
- **coherence_model** : un attribut de type texte qui représente sur le type de cohérence supporté par le moteur.
- **memory_use** : un attribut de type texte qui informe sur le type de caching. Il peut avoir comme valeurs : «in-memory», «filesystem cache», ...
- **journal** : un attribut de type texte qui décrit le type de journalisation effectué par le moteur de stockage pour assurer la durabilité des transactions.

- Classe DataCompression

« DataCompression » décrit les caractéristiques des techniques de compression supportées par un moteur de stockage. La compression est une fonctionnalité nécessaire pour la réduction du volume de données et l'archivage des données auxquelles on accède rarement. Cette fonctionnalité contribue au passage à l'échelle des données par incrémentation de l'espace de stockage des systèmes. Les attributs de cette classe sont :

- **compression_kind_of** : attribut de type ensemble de textes qui décrit les types de compression réalisées sur les données. Par exemple dans le moteur wiredTiger de MongoDB sa valeur est : {«block compression»,«prefix compression»}
- **compression_unit** : attribut de type ensemble de textes qui indique les unités élémentaires qui subissent la compression. Dans le moteur de MongoDB, sa valeur est : {«index»,«collection»}
- **compression_format** : attribut de type ensemble de textes qui renseigne sur les formats de compression supportés par le moteur de stockage (par exemple, dans Hadoop, les valeurs pour le format de compression sont « bzip », ...)
- **compression_library** : attribut de type ensemble de textes qui indique les bibliothèques utilisées pour la compression. Cette fonctionnalité est généralement fournie de manière native dans le système ou via une librairie tierce comme Snappy.

### 3.3.6 Opérateurs d'accès

La figure 3.13 synthétise les différentes étapes de traitement d'une requête pour son évaluation. La première étape consiste à analyser la requête et construire un ensemble d'alternatives de plans logiques équivalents qui découlent de la requête reçue en entrée. Par la suite l'optimiseur choisit parmi les plans de l'espace de recherche généré par l'analyseur l'alternative qui présente le moindre coût à l'exécution. Ce choix peut être compromis par des paramètres qui exigent l'utilisation d'un plan particulier et il est généralement fait par rapport à un modèle de coût.

Finalement, le module de génération de code transforme le plan élu en un plan physique constitué d'opérateurs physiques.

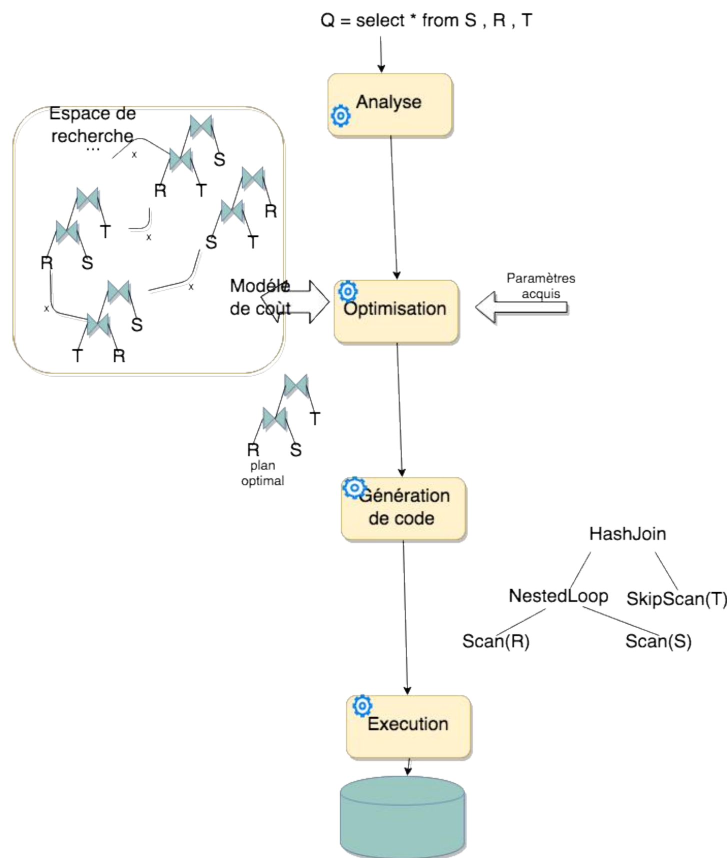



*Figure 3.13 Processus d'exécution d'une requête*

Cette section décrit ces opérateurs dans un objectif de construction des instructions (ou recommandations) susceptibles d'être exploitées par l'optimiseur des requêtes sur un lac de données. Décrire ces opérateurs nous permet aussi d'estimer les coûts et les performances des workloads sur les systèmes voir de donner des instructions explicites à l'optimiseur sur l'utilisation d'un opérateur particulier au lieu d'un autre vu les avantages qu'il procure.

Ces opérateurs sont décrits dans les classes du diagramme de la figure 3.13:

La classe « PhysicalOperator » représente les opérateurs d'accès physique. Elle a un attribut **limit** de type ensemble de chaînes de caractères qui décrit les limites de l'utilisation d'un opérateur physique. Trois sous-classes : « SequentialScanOperator », « IndexOnlyScanOperator » et « JoinOperator » héritent de cette classe.

La classe « SequentialScanOperator » décrit les opérateurs d'accès séquentiel, la classe « IndexOnlyScanOperator » décrit les opérateurs d'accès basés sur les index et finalement la classe « JoinOperator » décrit les opérateurs de jointure des ensembles de données. Elle a deux associations avec la classe « PhysicalOperator » pour représenter l'opérateur de jointure imbriquée. La première association : « inner_dataset_operator » a une cardinalité 0..1, de même que la seconde association : « outer_dataset_operator ».

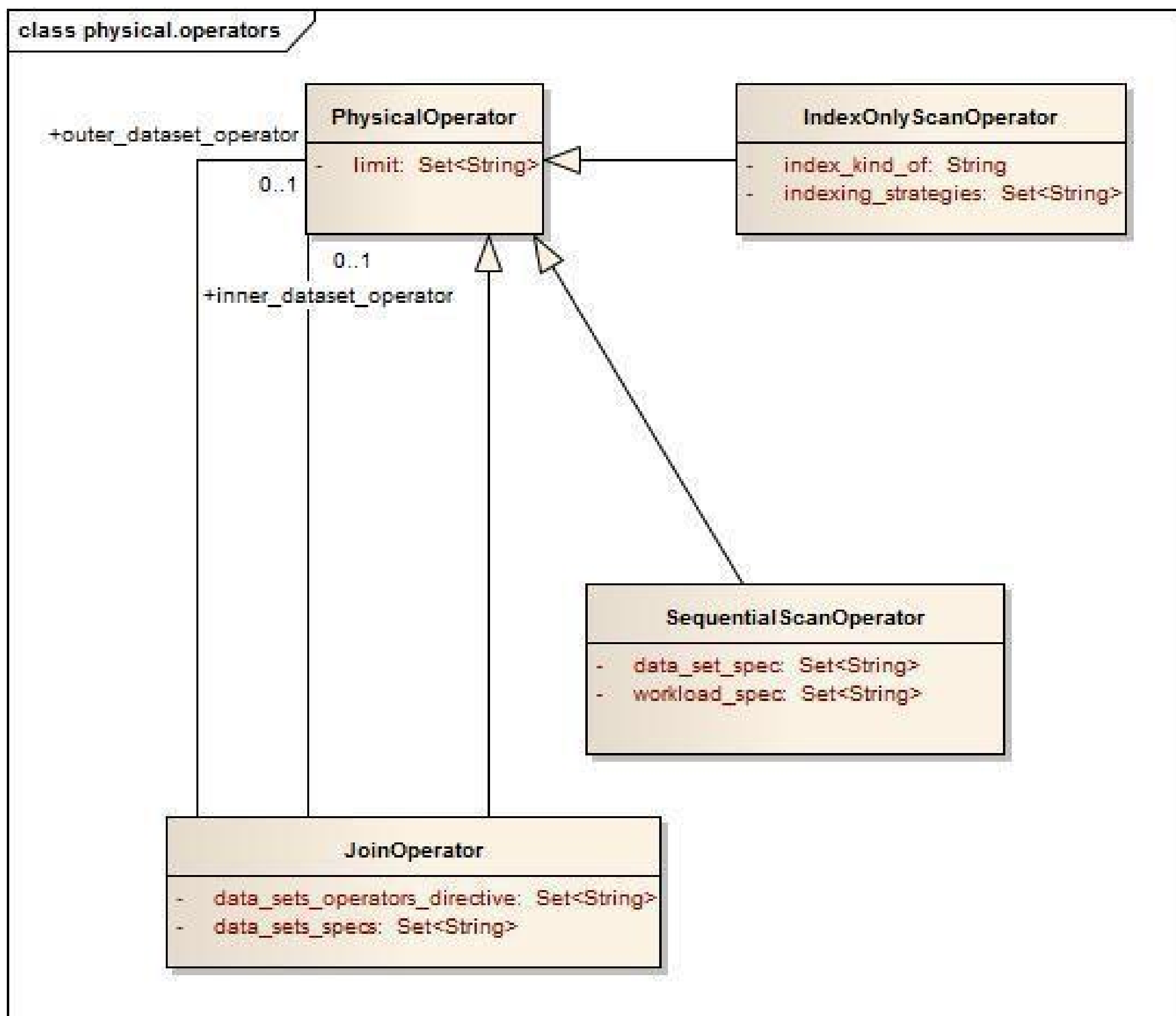



*Figure 3.14 Diagramme de classes «physical operators»*

#### 3.3.6.1 Classe « SequentialScanOperator »

Cette classe représente les opérateurs d'accès séquentiel à un ensemble de données. Avec ces opérateurs, aucun index n'est utilisé, l'accès aux données est le plus lent par rapport aux opérateurs équivalents. Ce type d'accès n'est donc pas optimal, néanmoins il existe des cas particuliers qui favorisent son utilisation. Elle est décrite par les attributs suivants :

- **data_set_spec** : ensemble de textes représentant différentes spécifications sur les ensembles de données gérés dans un système et qui favorisent l'utilisation de cet opérateur.
- **workload_spec** : ensemble de texte représentant différentes spécifications sur les workloads qui favorisent l'utilisation de cet opérateur.

Dans ce qui suit on présente deux instances de la classe « SequentialScanOperator ».

#### 3.3.6.2 Instance « fullTableScan »

Cette instance correspond à l'opérateur qui lit (au pire) l'intégralité de l'ensemble de données dans un ordre séquentiel pour atteindre la donnée recherchée. Il est utilisé dans des cas particuliers, surtout quand :

- la taille de la relation est réduite. L'accès séquentiel devient dans ce cas plus performant qu'une analyse d'une plage d'index
- la requête n'utilise pas de filtrage
- l'ensemble de données est accédé en parallèle
- l'utilisateur a imposé l'utilisation de cet opérateur explicitement

L'instance de « SequentialScanOperator » est définie comme suit :

> (fullTableScan, SequentialScanOperator (
> data_set_spec : {«small size», «doesn't have secondary index»},
> workload_spec : {«parallel access», «doesn't use filter»}
> limit : {«slow access»}
> ))

#### *3.3.6.3 Instance « fullCollectionScan »*

Le second exemple d'instance de cette classe correspond à l'opérateur séquentiel relatif au modèle de données basé sur les documents. C'est la solution équivalente du full table scan pour les collections.

#### *3.3.6.4 Classe « IndexOnlyScanOperator »*

Cette classe décrit les opérateurs d'accès basés uniquement sur l'indexation. Généralement l'accès par indexation est plus performant que l'accès séquentiel.

Elle a comme attributs :

- **index_kind_of** : un attribut de type texte dont la valeur renseigne le type de l'index. Il a comme valeurs possibles : «structured», «atomic» ou «any»
- **indexing_strategies** : un attribut de type ensemble de texte qui regroupe les stratégies adéquates de construction de l'index de scan.

Nous présentons les opérateurs « indexSkipScan », « indexRangeScan » et « indexFastFullScan » correspondant à trois instances de cette classe.

#### *3.3.6.5 indexSkipScan*

L'index skip scan (analyse par saut) est un opérateur d'accès basé sur les index de type composés (cf métadonnées «Data access mechanisms»). Il permet de diviser logiquement un index composite en sous-index plus petits. Le nombre de sous-index est déterminé par le nombre de valeurs distinctes du premier attribut de l'index. Le choix de cet attribut est déterminé à partir de l'attribut **distinct_attributes** de la classe « DataSet ».

L'utilisation de l'opérateur «index skip scan» est avantageuse s'il existe peu de valeurs distinctes dans le premier attribut de l'index composite et plusieurs valeurs distinctes dans la clé non linéaire de l'index. L'instance de la classe « IndexOnlyScanOperator » correspondant à cet opérateur est :

> (indexSkipScan, IndexOnlyScanOperator ( index_kind_of : «structured»,
> indexing_strategies : {«number of distinct values in the
> first attribute of the structured index shall be
> limited»,«the number of distinct values in the nonlinear
> key of the index shall be high» },
> limit:{«requires structured indexes» }
> ))

#### 3.3.6.6 indexRangeScan

« Index range scan » est un opérateur utilisé pour les requêtes de type sélection sur un attribut avec filtrage sur une plage de valeurs (range) relative à l'opérateur. Cette plage peut être bornée (délimitée des deux côtés) ou illimitée (d'un côté ou des deux côtés). Les résultats d'une requête qui utilise cet opérateur sont retournés dans l'ordre croissant des valeurs des attributs d'index et en cas de valeurs identiques pour un attribut particulier l'ordre est établi selon la clé des enregistrements de données (attribut **data_record** de la classe « DataModel ») dont les valeurs de l'attribut de l'index sont égaux.

L'instance de la classe « IndexOnlyScanOperator » qui représente cet opérateur est définie par:

(indexRangeScan, IndexOnlyScanOperator (
index_kind_of : «structured»,
indexing_strategies : {«index the dataset based on frequent range queries» },
limit:{«requires structured indexes» }
))

#### 3.3.6.7 indexFastFullScan

L'optimiseur utilise la valeur d'une métadonnées sur les statistiques de l'exécution de requêtes historiques basée sur les opérateurs de type «sequential scan» par exemple «fullTableScan» pour indexer une collection de données (par exemple une relation). Si la valeur de cet index est élevée indiquant un coût moins élevé pour les analyses complètes de relations (full table scan), alors l'optimiseur peut choisir l'opérateur full table scan au lieu d'un opérateur basé sur l'index. L'instance de la classe « IndexOnlyScanOperator » représentant cet opérateur est :

(indexFastFullScan, IndexOnlyScanOperator (
index_kind_of : «atomic»,
indexing_strategies : {«if index has high statistics then use sequential index» },
limit :{«requires statistics from historical workloads» }
))

#### 3.3.6.8 Classe « JoinOperator »

Cette classe décrit les opérateurs physiques liés à une opération logique de jointure. Elle a diverses instances chacune représentant une opération de jointure que l'optimiseur choisit en fonction d'une expression logique de jointure. Elle a comme attributs :

- **data_sets_spec** : ensemble de texte représentant différents spécifications sur les ensembles de données gérés dans le système et qui favorisent l'utilisation de cet opérateur
- **data_sets_operators_directives** : ensemble de texte représentant différentes directives sur les opérateurs d'accès des ensembles de données membres de la jointure. Cet attribut peut avoir comme valeur l'ensemble vide.

Dans ce qui suit on présente deux exemples d'instances de cette classe décrivant respectivement deux implantations de la jointure : « nested loops » & « hash join »

#### 3.3.6.9 Instance « nestedLoop »

Cette instance décrit l'opérateur nested loops (boucles imbriquées) qui correspond aux opérateurs logiques : « inner_join » (jointure interne), « left_outer_join » (jointure externe gauche),

« left_semi_join » (semi-jointure gauche) et « left_anti_semi_join » (anti semi-jointure gauche). Ses propriétés sont :

( nestedLoop, JoinOperator(
data_sets_spec: {«small», «inner dataset depends on outer dataset»},
limit: {«less performant than hash join with independent data sets»}
data_sets_operators_directives: {«outer dataset access should be optimized»}
))

#### *3.3.6.10 Instance « hashJoin »*

L'opérateur d'accès hash join est l'implémentation de la jointure par comparaison des clés des jointure en utilisant une fonction de hachage. L'optimiseur exécute la fonction de hachage sur l'attribut de l'ensemble de données le plus petit relatif à la clé de jointure pour construire en mémoire une table de hachage. L'opérateur hash join sert à exécuter l'opération logique « equi_join » si les conditions indiquées dans la propriété **data_sets_spec** sont vérifiées :

( hashJoin, JoinOperator(
data_sets_spec: {«large», «inner dataset and outer
dataset are independent»},
limit: {«poor performance», «memory greedy»}
data_sets_operators_directives: {}
))

## 3.4 Discussion et conclusions

Dans cette section nous avons présenté le modèle de métadonnées sur les ensembles de données, les workloads et surtout les systèmes. Ce modèle est validé sous forme d'instances présentées dans le chapitre dédié au prototype et à l'expérimentation. Nous avons présenté également les perspectives sur l'intégration des métadonnées dans les premières sous sections. Dans le chapitre suivant nous présentons le système de recommandation de placement DWS qui est basé sur ces métadonnées et qui met en évidence l'importance de les gérer.

# Chapitre 4 Système de recommandation de placement (DWS)

Dans la conclusion du chapitre 2 nous avons présenté un résumé de notre contribution : la recommandation sur le placement de données en utilisant des métadonnées et en exploitant des techniques d'apprentissage automatique. Dans ce chapitre nous présentons l'architecture et les algorithmes de ce système de recommandation. Ce dernier est appelé DWS acronyme pour « Data », « Workloads » et « Systems » les trois niveaux de métadonnées sur lesquelles se base la recommandation.

DWS est un système de recommandation qui utilise une approche par apprentissage automatique pour recommander la meilleure combinaison de solutions de stockage et de traitement pour un workload en utilisant la technique de simulation. Cette technique permet de récupérer des informations sur l'exécution du workload dans un système de traitement de données sans avoir à y stocker réellement la totalité de l'ensemble de données. Notre méthode consiste à envoyer au système de traitement / de stockage le schéma des données, des statistiques sur ces derniers et un extrait de ces données afin de pouvoir récupérer de ce dernier un plan de l'exécution du workload et une estimation de son coût si c'est possible. L'objectif de cette simulation est d'obtenir des métadonnées sur l'exécution de la requête dans différents systèmes hétérogènes ce qui est indispensable pour comparer la performance dans ces derniers sans avoir à stocker réellement des masses de données très importantes. Pour mieux comprendre le principe de notre proposition, on considère l'exemple du workload présenté à la fin du chapitre 2 (c.f conclusion chapitre 2) et on détaille les étapes de l'exécution de l'algorithme de recommandation du placement de données DWS.

Dans notre étude on s'intéresse aux applications exécutées fréquemment dans les écosystèmes Big Data des smart grids. Un exemple d'application consiste à construire des bilans pour la consommation des données d'énergies des clients.

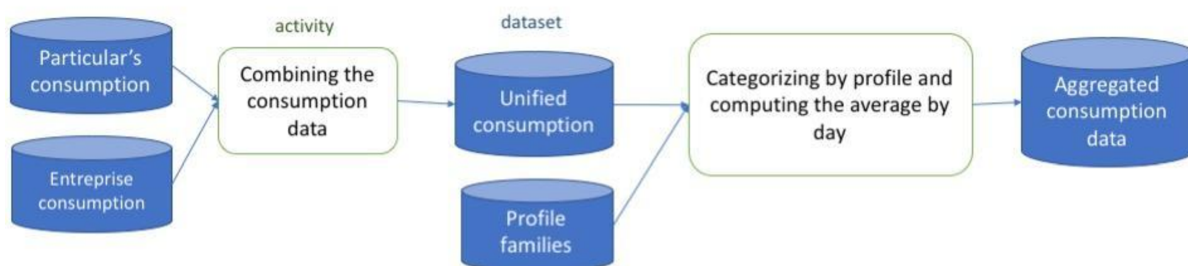



*Figure 4.1 Première partie du workload issue du cas d'utilisation : "construction d'un bilan de la consommation d'énergie dans un smart grid"*

À partir des données de départ (qui concernent la consommation de particuliers et d'entreprises) l'application propose quelques règles de manipulation des données qui sont nécessaires à agréger la consommation :

- faire d'abord l'union des deux jeux de données

- ensuite on croise ces données avec une nouvelle source : « Famille de Profil » (qui décrit les profils de consommation et les catégorise en profils : ENTreprise, PROducteur, RESident)

- ce même traitement calcule aussi la Consommation moyenne journalière et l'ajoute aux données pour des fins de comparaison entre la consommation journalière et la valeur observée toutes les 5 min

Ces manipulations sont exécutées séquentiellement sous forme d'activités lancées sur les données et l'objectif principal de notre étude consiste à recommander aux utilisateurs la meilleure combinaison de systèmes « de stockage et de traitements des données » qui permettent l'exécution la plus efficace de l'application dans son ensemble (de bout en bout). Toutefois un ensemble de contraintes s'impose : on souhaite que le temps de réponse des applications soit minimal et souhaite que les systèmes recommandés respectent les consignes des architectes et des utilisateurs de l'écosystème de données.

Comment fait- on fait cette recommandation ? Les applications sont d'abord collectées de l'écosystème et notre système de recommandation analyse et détermine la solution optimale au repos en utilisant un algorithme de recherche exhaustif.

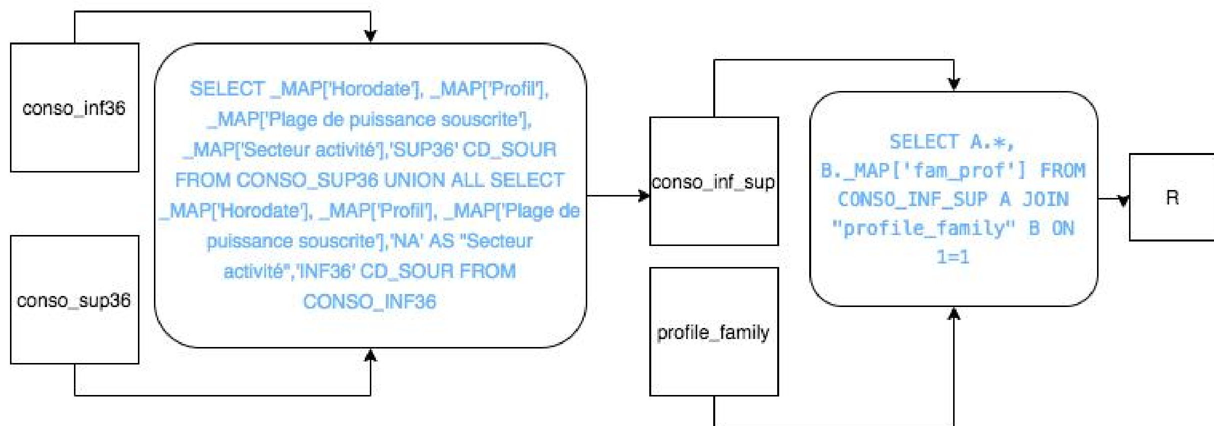



*Figure 4.2 Code du workload issue du cas d'utilisation : "construction d'un bilan de la consommation d'énergie dans un smart grid"*

Commençons par regarder en détails les opérateurs de la première requête. Comme le montre la figure 4.2, la première requête du workload contient les opérateurs suivants : *Union*, *Scan* et *Project*. La deuxième contient un opérateur *Join* qui est un opérateur très coûteux en Big Data.

La première étape de notre algorithme correspond à l'extraction des plans logiques à partir des requêtes des workloads. Un exemple du graph du workload est illustré dans la figure 4.3. Ce graph contient des métadonnées de requête comprenant les opérateurs de ces derniers et des métadonnées sur les ensembles de données : les références des ensembles de données.

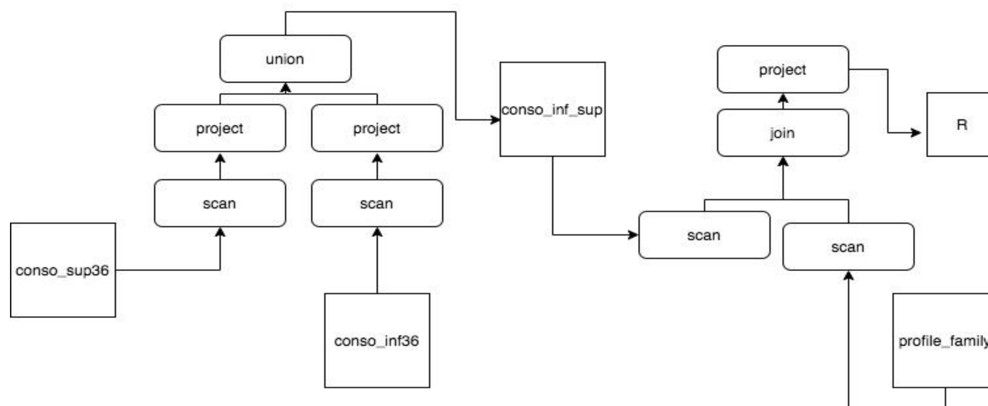



*Figure 4.3 Plan logique du workload du cas d'utilisation : "construction d'un bilan de la consommation d'énergie dans un smart grid"*

Par la suite, il interroge le magasin de métadonnées pour récupérer les métadonnées sur les systèmes disponibles dans l'écosystème et qui ont été renseignées par un expert métier ou bien le propriétaire

des données. On suppose pour notre exemple qu'on a limité les candidats de placement à Hbase, Neo4J et Spark. Ces métadonnées seront croisées avec les requêtes générées par la première étape pour construire un espace de recherche intermédiaire qui contient tous les combinaisons possibles de d'association (dataset -> data store), (subquery -> execution engine). La figure 4.4 présente un exemple de métadonnées système que l'algorithme utilise pour construire l'espace de recherche associé à la première requête.

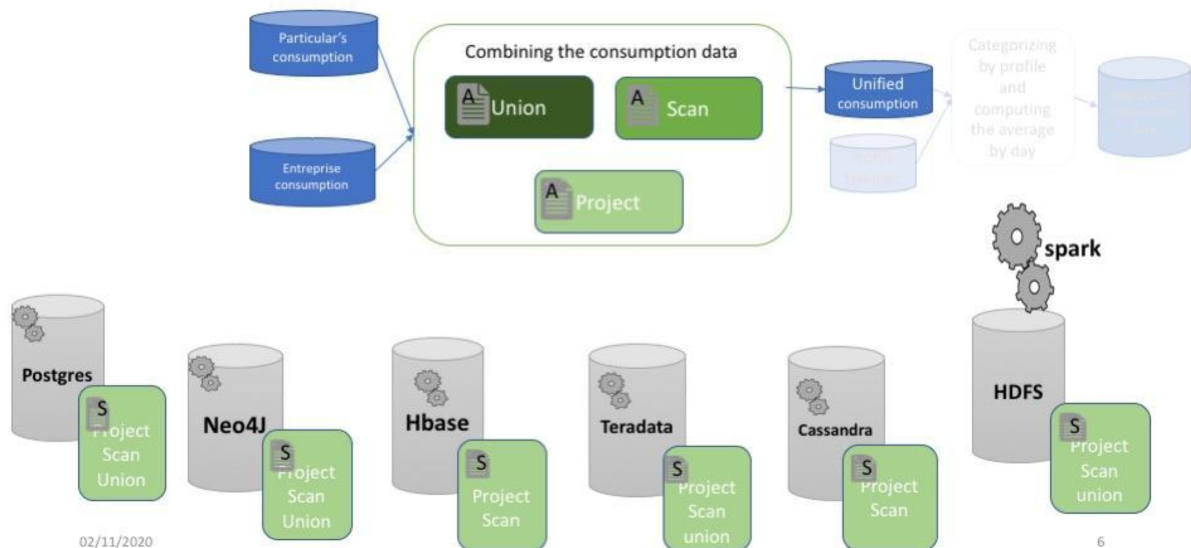



*Figure 4.4* Exemple de métadonnées utilisées dans l'algorithme de construction de l'espace de recherche

Cet algorithme extrait à partir de la base de métadonnées les propriétés sur les systèmes, comme les opérateurs supportés par chacun. Dans l'exemple de la figure on peut constater que le magasin de données HBase supporte uniquement les opérateurs suivants de la requête : *scan*, *project*. À la génération de l'espace de recherche on souhaite évaluer la performance de l'exécution d'une partie de la requête dans Hbase. Pour cette raison on propose de décomposer la requête, exécuter une partie dans Hbase selon quels opérateurs il supporte (après avoir réécrit la requête vers la syntaxe de l'API de Hbase) et finalement le résultat de cette requête sera envoyé vers un système qui supporte l'opérateur *union*. Une autre alternative de transformation sera de pré-calculer l'union des deux ensembles de données de départ et de stocker le résultat dans Hbase qui exécutera par la suite ce qui reste de la requête. Le résultat de la requête calculé peut être utile dans d'autres workloads.

La dernière étape de l'algorithme est la simulation de l'exécution du workload pour chaque transformation de l'espace de recherche. La section ci-dessous expliques les principes cités dans cet exemple avec plus de détails.

Le chapitre est organisé comme suit : (i) la section 4.1 présente le principe de notre recommandation, (ii) la section 4.2 présente l'architecture de DWS, (iii) la section 4.3 résume les techniques de transformations considérés dans DWS, (iv) la section 4.4 détaille les techniques d'optimisation de notre contribution, (v) la section 4.5 présente la solution de simulation de l'exécution de requête que nous avons utilisé dans DWS ,(vi) la section 4.6 résume les algorithmes développés dans le cadre de ce projet et finalement (vii) la section 4.7 récapitule le chapitre et présente une architecture détaillée de DWS.

## 4.1 Principe

À partir d'une description du workload collectée sous forme de métadonnées sur la requête que l'utilisateur a lancée, des métadonnées extraites des ensembles de données, notre système de recommandation est capable de choisir la combinaison de systèmes Big Data qui permet de stocker les données et d'exécuter le traitement d'une manière optimale.
La recommandation passe par trois étapes : (i) DWS sélectionne d'abord les instances de métadonnées du niveau systèmes Big Data disponibles. (ii) Par la suite, l'algorithme vérifie un ensemble de contraintes de conformité avec les décisions du propriétaire des données ou bien du responsable sur les données. Ces contraintes sont définies par l'utilisateur sous forme de métadonnées du niveau applicatif. (iii) la troisième étape correspond à l'inférence de la faisabilité du placement de données et la génération de plusieurs transformations possibles basées sur des métadonnées définies sur le niveau système par un expert du domaine. (iv) Finalement, pour l'espace de solutions généré, le placement de données sera simulé et le module de prédiction de temps de réponse associé à chaque transformation une estimation du temps de réponse.

Dans notre approche le coût calculé sous forme de temps de réponse et les modèles de coûts sont des modèles de prédiction du temps de réponse. L'originalité du travail est que l'optimisation de DWS tient compte des coûts de transformations et de communication dans les systèmes Big Data. Il considère aussi des transformations entre les différents modèles de données en utilisant des techniques de réécriture de requête.

Par contre nous avons fixé un ensemble d'hypothèses pour faciliter notre démarche expérimentale, pour limiter le cadre de cette recherche et pour ouvrir de nouvelles perspectives. On suppose dans cette étude que les schémas des données utilisés dans DWS sont homogènes. On n'a pas besoin d'utiliser des techniques avancées pour assurer leur intégration. Les problématiques de la mise en correspondance des schémas est donc considéré comme une perspective à ce travail.

Dans le chapitre 3, qui présente le modèle des métadonnées, nous avons présenté la conception détaillée de ce dernier pour un écosystème de gestion et de traitement des données massifs. Ces métadonnées doivent a priori (théoriquement) être utilisées intégralement dans DWS. Par contre, on va se limiter à l'utilisation uniquement d'une partie de ces métadonnées. Par conséquent, nous prenons dans ce projet comme hypothèse que les droits d'accès sur les données sont vérifiés par l'utilisateur qui utilise DWS. Ainsi les métadonnées d'administration ne seront pas utilisées dans la preuve de concept. Il est toutefois possible de les utiliser pour la vérification de la conformité. De plus, les métadonnées de configuration dynamique des systèmes, comme le nombre total des serveurs ou la capacité totale CPU, peuvent être ajoutées dans le cadre de la prédiction du temps de réponse dans certains systèmes distribués comme Apache Spark par exemple. Finalement on utilise des métadonnées qui caractérisent les APIs des systèmes ainsi que le modèle de données pour vérifier le critère de faisabilité.

Dans ce chapitre, nous présentons l'architecture de DWS : dans la première partie, nous expliquons le rôle des différents composants du système de recommandation proposé. Ensuite, nous détaillons le modèle de données des trois niveaux : ensemble de données, workloads et systèmes ainsi que les critères pour déterminer la solution optimale de placement de données. Finalement nous présentons l'algorithme de recommandation du placement de données.

## 4.2 Architecture de DWS

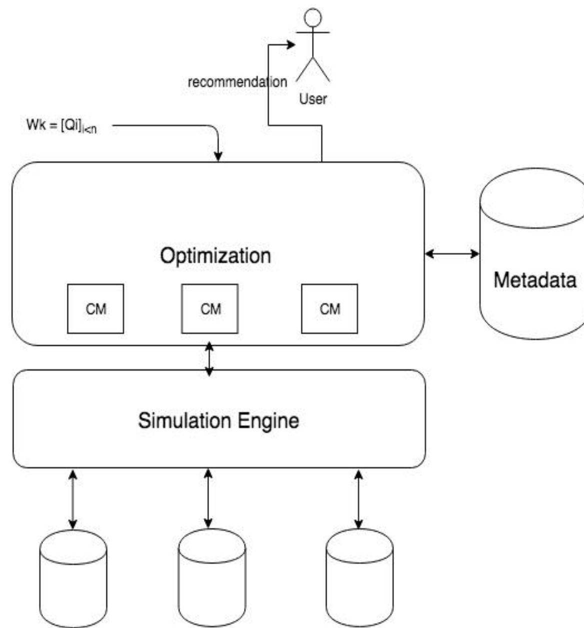



*Figure 4.5* Diagramme d'architecture de DWS

Le composant principal de notre système de recommandation DWS est le module d'optimisation. Comme le montre la figure ci-dessus, ce module contient plusieurs boîtes noires (CM) qui permettent d'estimer le coût de l'exécution du workload. Chacune de ces boîtes noires est composée d'un modèle d'apprentissage automatique qui est spécifique à l'environnement d'exécution des workloads qui ont été utilisés pour constituer la base d'apprentissage et aux systèmes utilisés pour le stockage des ensembles de données et l'exécution du workload. Ce module d'optimisation permet, à partir d'un workload décrit par l'utilisateur sous forme d'un graphe de requêtes dont les liens sont des échanges de données (un ensemble de données en entrée, un ensemble intermédiaire ou bien un résultat final), de générer pour chaque ensemble de données et chaque requête du workloads les transformations possibles en exécutant des réécritures de plans de requêtes ou bien en simulant des transformations du schéma physique des données. Cette étape est appelée une étape de construction de l'espace de recherche et elle retourne un ensemble de chemins équivalents mais qui peuvent avoir des temps de réponses variés. Par la suite l'optimiseur en utilisant un algorithme de recherche exhaustif détermine la meilleure transformation qui permettra d'exécuter le workload efficacement. Dans cette recherche, DWS fait appel à des modèles de coûts propres à chaque système et à chaque configuration. Ces modèles estiment la performance en utilisant un autre module de simulation de l'exécution du workload. Ce dernier envoie les requêtes aux systèmes d'exécution et récupère si c'est possible le coût de la requête calculé par le système sous-jacent. La simulation dans notre travail consiste à l'injection de statistiques sur les ensembles de données au moteur d'exécution pour connaître le coût du traitement estimé sans stocker physiquement les données. L'intérêt de cette étape est de gagner le temps de chargement des données à chaque simulation et d'augmenter la précision de l'estimation du temps de réponse en utilisant les coûts retournés par le moteur d'exécution.

Dans ce qui suit nous présentons les modèles de données de chaque entité de notre système de recommandation ainsi que les différents critères qui nous permettent de déterminer la solution optimale de placement.

### 4.2.1 Modèle de données pour les ensembles de données

Le modèle de données pour les ensembles de données dans DWS sert à définir d'une manière unique la représentation des données dans un environnement hétérogène. Pour la représentation interne des ensembles de données dans notre module de placement, nous adoptons le modèle de données ODMG. Ce modèle propose deux constructeurs pour organiser les valeurs en ensembles de données : le constructeur de collection qui organise les valeurs en ensembles, listes, sacs, tableaux et dictionnaires, et le constructeur d'agrégation qui organise les valeurs en objets ou agrégats. Dans ce modèle, les valeurs $v_i$ sont soit des littéraux, soit des agrégats. Ces valeurs sont également classées en atomiques, structurées ou collections. Nous représentons les valeurs nulles et les valeurs non structurées (par exemple, les gros objets binaires : blobs) respectivement sous forme de valeur *Null* et de valeur agrégée. Pour construire un ensemble de données, nous utilisons d'abord le constructeur de collection. Il regroupe les agrégats liés d*i* dans une collection S: $S = <d_0..d_n>$.

Le constructeur d'agrégats crée des objets qui sont soit structurés (par exemple, un tuple d'un ensemble de données relationnelles), semi-structurés (par exemple, un document) ou non structuré (par exemple, un blob). Dans notre modèle, nous gardons la même structure pour les agrégats et nous représentons l'absence d'une paire attribut-valeur comme valeur manquante. Par conséquent, les valeurs manquantes et les valeurs nulles sont deux représentations différentes dans notre modèle, et nous associons à chaque cas un littéral différent. La variété des données manipulées dans notre écosystème nous motive à envisager ce modèle flexible. Avec cette représentation, nous pouvons traiter des ensembles de données gérés dans les magasins de données ayant un modèle de données relationnel, valeurs-clés, document ou graphes. Par exemple, pour une table relationnelle, nous représentent les relations avec le constructeur de collection et les tuples avec le constructeur d'agrégat. Dans les magasins de documents et agrégats sont représentés comme des documents et organisés en collections. Nous représentons les graphes d'attributs par deux ensembles de données : ensembles de données des nœuds et ensembles de données des arêtes. Nous utilisons deux constructeurs de collection distincts : une pour représenter les nœuds d'un graphe et une pour les relations entre ces nœuds. Des constructeurs d'agrégats sont également utilisés pour représenter les nœuds et les arêtes. Enfin, nous représentons des ensembles de données stockés dans les magasins valeurs-clés avec le constructeur de la collection et les paires clé-valeur avec un agrégat (un agrégat unaire).

La mise en évidence de l'intérêt de ce modèle abstrait de données est réalisée dans le chapitre 3 section 1 qui présente les métadonnées du niveau données. En utilisant ce modèle, nous avons conçu un modèle de métadonnées qui est capable de supporté les formats variés d'un écosystème de données massives.

### 4.2.2 Modèle pour le workload

Une charge de travail dans l'écosystème du Big Data quantifie le traitement effectué sur les systèmes de données. Dans notre contexte, nous considérons les charges de travail de requête dérivées d'applications et de scripts SQL utilisés dans le cas d'utilisation d'agrégation de consommation. Selon le modèle proposé, les charges de travail de requête sont regroupées sous forme de flux d'opérations (ou d'activités) et elles sont exécutées séquentiellement dans des pipelines de données. Ainsi, nous représentons une charge de travail de requête sous forme de graphique $G = <V, E>$. V est l'ensemble des sommets représentant un opérateur logique, une variable ou une valeur et E est l'ensemble des arêtes représentant les dépendances entre les sommets. Dans le graphe des charges de travail de requête, nous utilisons la représentation intermédiaire de la charge de travail de requête pour la structure d'activité des données. Cette représentation intermédiaire peut être un arbre de syntaxe asymétrique (AST), par exemple sous forme de charges de travail de l'API d'accès MongoDB ou d'un

graphe acyclique orienté (DAG). Le choix de représenter la charge de travail à l'aide de la représentation intermédiaire des requêtes facilite l'exécution de l'algorithme de placement indépendamment de l'environnement et du langage de requête cible. Dans notre expérience et pour des raisons de simplification, nous utilisons des AST. La figure 4.2 présente un exemple de pipeline de données de vue compacte d'une application de traitement de données smart grid. Ce pipeline est structuré comme un DAG d'activité de données. Les activités de données smart grid regroupent généralement différents ensembles de données. Ils évoluent généralement à l'aide d'un partitionnement horizontal et appliquent de nombreux filtres et transformations pour obtenir la sortie souhaitée. Comme exemple d'activité de données, nous considérons le graphique suivant (figure 4.3). Cette activité est la première étape du pipeline. Elle enrichit l'ensemble de données du périmètre avec la date du calcul de l'indicateur et stocke le résultat dans une table intermédiaire qui sera utilisée comme entrée pour le reste du pipeline.

### 4.2.3 Modèle pour les systèmes

Un système de données dans les écosystèmes de Big Data est soit un système de stockage de données comme des magasins de données, des bases de données ou des systèmes de fichiers distribués (DFS) ou un moteur de traitement, par exemple des moteurs de traitement basés sur MPP. Nous modélisons pour les systèmes deux entités : (i) l'abstraction d'un système de données qui détaille les caractéristiques de ces systèmes modélisés à partir de l'étude de l'état de l'art. Et (ii) les descripteurs systèmes qui représentent les systèmes de données disponibles dans l'écosystème de données smart grid. Nous représentons un descripteur de système comme un objet composite. Nous associons à cet objet deux autres objets : un descripteur de stockage et un descripteur de traitement. Le descripteur de stockage est un objet qui définit des attributs tels que le modèle de données, le partitionnement modèle, modèle de distribution, ... De même, le Descripteur de Traitement a en tant qu'attribut : API de requête et d'accès prises en charge, opérateurs physiques, modèle de données, ... Les descripteurs sont représentés selon le modèle ODMG en tant que classes ; les relations entre ces classes sont soit composition et association ou héritage.

### 4.2.4 Critère de faisabilité

Cette étape permet de vérifier si le système cible est capable d'exécuter le workload, ou une de ses activités. On propose de vérifier deux propriétés : si le système cible est capable (i) de traiter des volumes importants, et (ii) d'exécuter les opérateurs du workloads c'est-à-dire si les opérateurs des requêtes du workloads sont supportés par l'API du moteur d'exécution ou son langage de requête.

On vérifie aussi la compatibilité entre les systèmes en utilisant des règles logiques. Cette compatibilité est une compatibilité des versions des systèmes par exemple dans le cas de HDFS version 1 et Hive version 2 qui ne sont pas compatibles.

#### *4.2.4.1 Faisabilité basée sur la taille des données*

Pour le stockage de volumes massifs de données, il est nécessaire de vérifier si le système est capable de stocker une taille aussi importante : si le système supporte la distribution et le partitionnement (surtout le partitionnement horizontal) sur plusieurs serveurs on présume que ce système supporte le stockage d'un volume infini. Pour les systèmes qui ne sont pas conçus pour passer à l'échelle, on propose de découvrir les limites de ces systèmes en utilisant des techniques d'inférence. D'après la littérature les systèmes centralisés peuvent supporter des terabytes de données [Tay]. Dans DWS, on souhaite ne pas nous limiter à des heuristiques qui déterminent ces limites mais on veut prendre en considération la taille mémoire disponible, le nombre de serveurs dans les nœuds et, à partir des expériences sur le système distribué, déterminer les requêtes qui s'exécutent et celles qui sont interrompues faute de ressources.

Dans cette expérience, on collecte des mesures exécutées sur différents systèmes. Les systèmes considérés sont ceux qui sont décrits dans ce module par les classes : « StorageDiscriptor » qui représente un descripteur de stockage et « ProcessingDescriptor » pour les descripteurs de traitement. Pour diversifier les cas d'étude, on utilise des benchmarks et on génère des requêtes complexes qui sont proches de nos cas d'utilisation.

Dans cette étude, on considère une requête qui prend un temps d'exécution très important comme non faisable ainsi que celles qui ne s'exécutent pas et qui retournent des erreurs. Dans ce qui suit, on résume et on visualise les données collectées pour l'étude de faisabilité qui a été faite sur plusieurs serveurs et pour différents systèmes.

##### *4.2.4.2 Faisabilité basée sur la compatibilité des opérateurs*

Pour préparer les données pour cette méthode, on commence par sélectionner les propriétés des systèmes à partir des métadonnées, ensuite nous les croisons avec les métadonnées du niveau système que nous avons préparées pour concevoir le modèle d'inférence de la compatibilité des opérateurs avec les systèmes cibles.

Nous proposons d'utiliser une méthode d'apprentissage automatique pour déterminer la faisabilité du placement. Après le croisement des métadonnées des deux niveaux *workloads* et *systems* nous attribuons le label « 1 » à l'attribut **feasable** si l'opérateur logique du workload est supporté par le système et « 0 » sinon. Un exemple d'observation qui est obtenue après cette étape est représenté comme suit :

hbaseApiFeasabilityJoin (**logical_operator** : « join »,
**data_model** : [« key value based data model», « column based data model »],
**query_categories** : [« crud », « time series »]
**identifier_kind_of :** [« compound », « user defined », « sorted »],
**supported_logical_operators:** [« scan », « project», « filter », « count »,« exists »,« insert »],
**workloads_kind_of:** [« interactive»],
**parallelism_supported :** true**,**
**query_access_kind_of :** « Api »,
**query_system_kind_of :** « natif »,
**feasable : 0**
)

Pour construire la base d'apprentissage, on interroge les métadonnées et on les transforme à partir de leurs modèles de données d'origine (orienté graphe) vers un modèle de données compatible avec la vaste majorité des algorithmes d'apprentissage (format de données tabulaire).

### 4.2.5 Critère de performance

Dans cette sous-section, nous expliquons la motivation derrière le choix du temps de réponse comme critère de performance et par la suite nous présentons la méthode utilisée pour développer le module de prise de décision qui sélectionne la solution optimale par rapport à ce critère.

L'intérêt à l'estimation de la performance dans le domaine de l'évaluation de requête se manifeste souvent dans les problèmes d'optimisation de requête. Comme on a vu dans le chapitre 2, l'optimisation des requêtes a comme objectif de minimiser un modèle de coût. Ce dernier est un ensemble de formules mathématiques qui permettent de modéliser la performance de l'exécution des

opérateurs en estimant des statistiques sur le résultat de la requête, ou bien sa consommation en ressources disponibles, comme la quantité de calculs (CPU), mémoire ou entrées/sorties vers le disque. Ces modèles expriment le coût d'une requête en utilisant des unités arbitraires ou bien en nombre d'unités CPU nécessaires pour le calcul (par exemple dans MySQL) ou bien le nombre de pages de données traitées (par exemple PostgreSQL). En effet, si le système utilise moins de ressources, il produit le résultat plus efficacement (rapidement).

Les problèmes des modèles de coût dans les systèmes Big Data concernent surtout les problèmes liés au parallélisme et à l'exécution distribuée. Par contre les travaux de recherche dans ce domaine sont relativement limités vu la difficulté du domaine et l'impossibilité de prédire la performance de l'exécution dans un milieu distribué. L'évaluation des systèmes selon leurs performances est aussi présente dans les techniques de benchmarks. Cette performance peut être modélisée comme une estimation de la latence (temps de réponse) ou bien une estimation de la consommation des ressources (par exemple en CPU, mémoire ou entrée/sortie).

L'idée centrale de notre approche est d'identifier le placement optimal des données sans recourir à l'exécution du workload. Dans notre étude nous avons identifié trois méthodes intéressantes pour récupérer une estimation de la performance de la requête en respectant ces conditions. La première méthode (i) correspond à la modélisation des coûts pour les opérateurs des requêtes de tous les systèmes Big Data de notre écosystème et trouver les statistiques pertinentes pour tous ces modèles. La deuxième méthode (ii) est appelée recommandation de placement par simulation et elle a comme objectif d'injecter les statistiques dans l'optimiseur de requête afin que ce dernier retourne une estimation du coût. Finalement, (iii) une troisième méthode à considérer consiste à utiliser des techniques de machine Learning pour estimer la performance. Dans les écosystèmes Big Data, ces trois méthodes présentent des avantages et des inconvénients variés. Dans la suite, chaque méthode sera présentée et détaillée afin de justifier les choix d'architecture de DWS.

On peut parler en perspective sur le niveau de concurrence et le débit comme un autre critère pour le placement de données mais on se limite dans cette étude au temps de réponse du workload.

### 4.2.6 Critère de conformité

L'architecte de données ou bien les utilisateurs peuvent interdire l'utilisation d'un ou plusieurs systèmes dans l'architecture ou bien imposer à l'utilisation un ou plusieurs systèmes pour des données particulières. Nous proposons dans ce cas un critère de vérification de la conformité du placement et il concerne deux scénarios types.

- Le premier scénario à prendre en considération est celui d'un utilisateur qui souhaite prendre le contrôle complet de cette décision et imposer une solution précise de placement. Ce choix peut être arbitraire ou bien motivé par les compétences du propriétaire de l'ensemble de données.
- Le choix peut également être imposé par l'architecte du lac de données. Ce type de conformité limite l'utilisation de data stores dans l'écosystème Big Data.

Le développement des règles du critère de conformité peut être fait en utilisant un moteur de règles métiers par exemple Apache Drools [drwdD]. Dans notre travail on se limite à une implémentation simple qui ne fait pas appel à l'inférence et on propose une étude de ce sujet comme perspective à ce travail.

## 4.3 Transformation des données

Dans cette partie on étudie les différentes transformations de point de vue performance ensuite on présente la méthode et les résultats de l'inférence de la faisabilité de l'exécution du workload. Dans notre étude des différentes solutions de placement dans le lac de données et son écosystème global, il est nécessaire de considérer les cas de placement dans tous les systèmes de l'architecture de l'écosystème, qui peuvent être très hétérogènes. Il est important d'énumérer en détails les différentes transitions d'un modèle de données à un autre. Dans notre étude, les données à transformer sont des données semi-structurées. Elles sont stockées dans des fichiers qui ont une disposition (en anglais *layout*) tabulaire généralement sérialisés sous format CSV.

La transformation de données est un processus qui convertit les données d'un modèle de données à un autre. Dans les écosystèmes Big Data, non seulement les ensembles de données sont ciblés par la transformation de données, mais également leurs workloads associées. La transformation des ensembles de données fait partie des techniques les plus largement utilisées dans les applications de base de données et d'analyse. C'est le bloc de base des processus d'intégration de données ETL [drwdT] et ELT [drwdO]. Il a d'abord été utilisé pour convertir les ensembles de données du format source au format cible dans des entrepôts de données (par exemple, le processus ETL d'un ensemble de données CSV vers un RDBMS). Un défi important dans ce type de transformation consiste à créer un schéma pour la base de données cible, à le mapper avec les schémas sources et convertir les types de données. En Big Data, les transformations de données sont plus complexes. Les modèles de données des data stores sont hétérogènes, les attributs peuvent être atomiques, structurés ou à valeurs multiples. Le schéma est généralement absent, ce qui complique davantage la transformation vers le modèle relationnel (par exemple). Dans ce type de scénario, la gestion globale des métadonnées dans les écosystèmes Big Data devient un service clé. Caractériser les systèmes Big Data en spécifiant les types de données pris en charge, le niveau d'atomicité de chaque attribut de données et les informations relatives au schéma permet une transformation plus facile des données d'un modèle de données à un autre. Cette transformation utilise les métadonnées au niveau des systèmes.

Dans le cas de la transformation de données non structurées ou semi-structurées en un modèle relationnel, un autre service de métadonnées est d'une grande importance : l'inférence de schéma et l'enregistrement de schéma dans un catalogue de métadonnées global. Ces transformations ne sont pas couvertes par notre étude car les ensembles de données de notre cas d'utilisation sont assimilables à du relationnel.

Dans le placement des données, nous considérons également d'autres transformations fondamentales pour améliorer les performances de la requête. Nous proposons d'étudier la distribution des ensembles de données ainsi que les solutions d'indexation dans notre étude de transformation. Nous utilisons des métadonnées de systèmes caractérisant les techniques de partitionnement et d'indexation prises en charge par les systèmes Big Data.

Le deuxième type de transformation concerne la transformation des workloads. Les systèmes Big Data prennent en charge un ensemble large et inégal de langages de requête et de systèmes d'accès aux données. La réécriture des requêtes pour correspondre au schéma du jeu de données cible, mais surtout à l'API du moteur de requête est la fonctionnalité principale du composant de transformation des données dans les infrastructures de données modernes. Un autre défi auquel notre infrastructure de placement de données est confrontée à ce niveau est la décomposition des requêtes et le choix du moteur de traitement qui exécutera chaque opérateur. A titre d'exemple, considérons la requête de jointure hybride suivante qui combine des jeux de données stockés de manière respectueuse dans les systèmes PostgreSQL, MongoDB et HDFS:

$$S_1 \bowtie_1 S_2 \bowtie_2 S_3$$

Dans les systèmes d'intégration des systèmes Big Data (par exemple Spark SQL), les jointures sont généralement exécutées en mémoire par le moteur d'exécution du système d'intégration. Il existe des systèmes d'intégration qui supportent l'optimisation par envoi des prédicats à la couche physique (couche de stockage) et ceci dépend des versions de systèmes d'intégration et des systèmes supportés. L'approche d'exécution hybride des jointures dans le système d'intégration sous-utilise les capacités des magasins de données sous-jacents. En effet, cette exécution hybride engendre généralement des coûts réduits de communication réseau et d'accès disque local et l'exécution locale est ainsi plus efficace que l'exécution distribuée. Dans notre approche, nous souhaitons évaluer toutes les alternatives possibles et nous considérons comme une transformation l'exécution d'une partie de la requête et le stockage du résultat intermédiaire dans le système qui a exécuté la sous-requête. Pour cet exemple, nous présentons quelques transformations de requête possibles :

- En exécutant $S_1 \bowtie_1 S_2$ dans PostgreSQL, considérons alors le résultat $R_1$ en tant qu'entrée de la requête. La requête transformée devient alors : $R_1 \bowtie_2 S_3$. Seulement $\bowtie_2$ est dans cette solution exécutée dans le multi-magasin qui minimise le réseau et le coût de la charge.

- Exécution de $S_2 \bowtie_2 S_3$ dans PostgreSQL, considérons alors le résultat $R_2$ comme entrée de la requête. La requête transformée devient alors : $S_1 \bowtie_1 R_2$. Seulement $\bowtie_1$ est dans cette solution exécutée dans le multi-magasin.

Notez que les systèmes MongoDB et HDFS ne prennent pas en charge les jointures. La sous-requête de jointure n'est pas faisable dans ces systèmes et ils ne sont pas pris en compte par le constructeur du simulateur de transformation de requête.

La transformation des requêtes est également guidée par les métadonnées des systèmes et impactée par le critère de faisabilité. Notre objectif de générer toutes les transformations de requête possibles est l'évaluation de toutes les alternatives dans le placement du workload en tenant compte des moteurs de traitement disponibles de l'écosystème.

Les techniques utilisées pour la transformation de requêtes complexes sont la réécriture de requête et la matérialisation des résultats. Notre système étant optimisé, nous tirons parti des résultats intermédiaires de ce moteur.

La transformation finale adressée par DWS est le partitionnement et l'indexation des données. Pour le partitionnement vertical (s'il est pris en charge par le système), nous identifions à partir de la charge de travail les attributs auxquels qui sont généralement accédés ensemble et nous associons une étiquette au regroupement de ces attributs. Chaque groupe est stocké sur le même serveur afin d'obtenir un partitionnement vertical dépendant de la charge de travail et la localité des attributs.

Pour l'indexation, à partir de notre base de requêtes, nous identifions la fréquence d'accès sur les attributs des tables et nous indexons les données en conséquence. Enfin, pour le partitionnement horizontal, nous observons les filtres appliqués aux données et décidons de l'organisation la plus appropriée.

### 4.3.1 Transformation entre modèles de données

Dans la couche de métadonnées nous exprimons des correspondances (en anglais *mappings*) entre le modèle de référence, qui est le modèle relationnel, et les modèles de données Big Data. En se basant sur le modèle relationnel nous avons déterminé les compatibilités entre 3 grandes catégories de modèles : les modèles basés sur l'agrégation : document et clés valeurs, les modèles basés sur les graphees et les modèles relationnels.

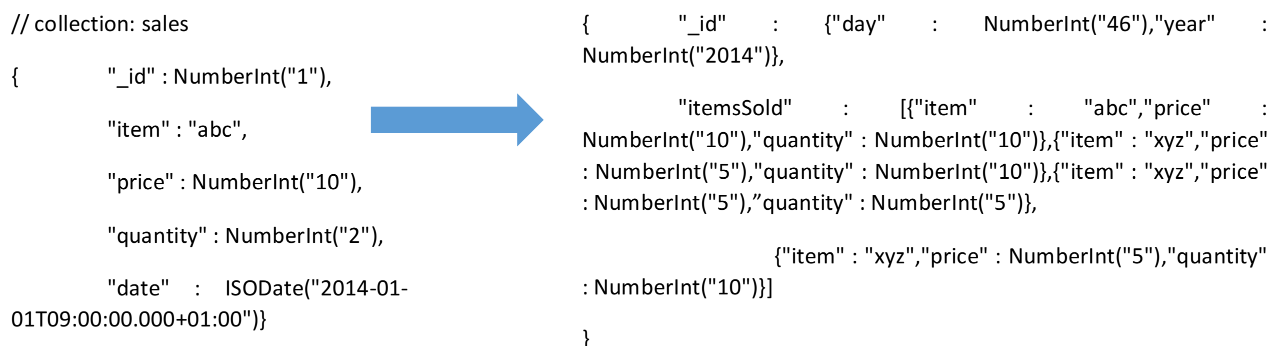
Ces correspondances sont utilisées dans notre logique de transformation entre les différents modèles de données. Elles sont établies entre deux modèles physiques de données et elles guident les transformations.

Pour transformer les données nous considérons deux fonctions : l'agrégation sous forme de valeurs structurées et la création de liens entre entités.

La transformation par agrégation transforme des attributs atomiques en attributs structurés et multivalués si le système le permet.

Cette transformation nécessité une réécriture de requête pour permettre à l'utilisateur d'accéder aux attributs des valeurs structurés.

Exemple :




On utilise des transformations par association pour trouver des liens entre les relations et créer les solutions associées à la représentation des liens selon le modèle de données cible. Les liens peuvent être déterminés à partir d'une abstraction du modèle entité-association dans notre solution. Mais il n'existe pas de compatibilité directe entre le modèle entité-association et l'ensemble des modèles de données existants. C'est un modèle théorique qui a été adapté partiellement dans les systèmes existants.

Une autre consiste à découvrir des correspondances entre les tables automatiquement en utilisant des algorithmes d'apprentissage [BCN17].

Et la troisième méthode, qui correspond mieux à notre approche guidée par les métadonnées des workload, consiste à découvrir les jointures des métadonnées qui décrivent les requêtes (notamment leurs structures et l'existence d'opérateurs de jointure). Dans ce cas, les jointures sont transformées en liens et les clés de jointures en attributs de liens si le système cible supporte les attributs sur les liens.

Dans le tableau 4.1, je présente le schéma de transformation pour différents modèles de données et les règles proposées pour passer d'un modèle à un autre et ceci pour les transformations de données qui ne sont pas interrogées en utilisant l'opérateur de jointure avec d'autres données, pour des données qui sont interrogés avec l'opérateur de jointure et que la clé de la jointure est unique pour les deux ensembles en question, qui sont unique pour un des ensembles ou bien qui ne sont pas uniques. L'unicité de la clé de jointure détermine si le produit par la jointure est un tuple ou bien plusieurs.

| Transformation | Case target system is | Entity to | 1-1 association | 1-n association | n-n association |
|---|---|---|---|---|---|
| Transformation to an aggregate data model | document store (eg MongoDB) | Transformation_to_target_schema_1 = ( data_collection : collection, data_record : document, data_attribute : field, Referencial_constrain :[ « dbref (not recommended) » , « denormalization using documents nesting »] null_values : [« missing fields » , « null »] ) Transformation rules from entity association model: Entity-> collection Object (or instance) -> document Attribute-> field Association (1-1)-> dbref (not recommended) \| documents nesting as a structured field \| lookup null_values -> missing fields \|null | Transformation_to_target_schema_9 = ( source_collection : document, Target_collection: field, Target_data_attribute : a map of fields, Referencial_constrain :[ « dbref (not recommended) » , « lookup», « denormalization using documents nesting »] Target_null_values : [« missing fields » , « null »] ) Transformation 9 rules from entity association model : Source_entity-> transformation 1 rules Target_Attribute-> aggregate(attribute) Association (1-1)-> dbref (not recommended) \| lookup \| documents nesting as a structured field Target_null_values -> missing fields \|null | Transformation_to_target_schema_14 = ( source_collection : document, Target_collection: element of array in a field, Target_data_attribute : an array of document (for every referenced collection), Referencial_constrain :[ « dbref (not recommended) » , « lookup», « denormalization using documents nesting »] Target_null_values : [« missing fields » , « null »] ) Transformation 14 rules from entity association model : Source_entity-> transformation 1 rules Target_Attribute-> collection(aggregate(attribute) ) Association (1-n)-> dbref (not recommended) \| lookup \| documents nesting as a structured field Target_null_values -> missing fields \|null | Transformation_to_target_schema_19 = ( source_collection : document, Target_collection: document, Intermediate_collection : element of array in a field, intermediate_data_attribute : an array of document (for every referenced collection), Referencial_constrain :[ « dbref (not recommended) » , « lookup», « denormalization using documents nesting »] Target_null_values : [« missing fields » , « null »] ) • Transformation 19 rules from entity association model : Source_entity-> transformation 1 rules Target_entity-> transformation 1 rules Intermediate_collection -> collection(aggregate(attribute) ) Association (n-n)-> dbref (not recommended) \| lookup \| documents nesting as a structured field for the intermediate collection in both association entities Target_null_values -> missing fields \|null |

| | | | | | |
|---|---|---|---|---|---|
| | column family store (eg hbase) | Transformation_schema_2 = (<br>Table -> collection<br>Row -> row<br>Column -> column_family<br>Referencial_constrain -> denormalization and storage in a single table \| denormalization using structured values (binary)<br>null_values -> missing fields \|null<br>)<br><br>Transformation_schema_3 = (<br>Table -> collection<br>Row -> row<br>Column -> column<br>Referencial_constrain -> denormalization using column families structure \| denormalization and storage in a single table \| denormalization using structured values (binary)<br>null_values -> missing fields \|null<br>) | Transformation_to_target_schema_10 = (<br>source_collection : table,<br>Target_collection : column family,<br>Target_data_attribute : column,<br>Referencial_constrain :[« denormalization using column families structure»,« denormalization and storage in a single table” , “denormalization using structured values (binary) »]<br>Target_null_values : [« missing fields » , « null »]<br>)<br>Transformation 10 rules from entity association model :<br>Source_entity -> transformation 2,3,4 rules<br>Target_Attribute- > column<br>Association (1-1)-> denormalization using column families structure \| denormalization and storage in a single table \| complex attribute having structured value (binary)<br>Target_null_values -> missing fields \|null | Transformation_to_target_schema_15 = (<br>source_collection : table,<br>Target_collection : column,<br>Target_data_attribute : array of map,<br>Referencial_constrain :[« denormalization using column families structure»,« denormalization and storage in a single table” , “denormalization using structured values (binary) »]<br>Target_null_values : [« missing fields » , « null »]<br>)<br>Transformation 15 rules from entity association model :<br>Source_entity -> transformation 2,3,4 rules<br>Target_Attribute- > collection(aggregate(attribute)) as column<br>Association (1-n)-> denormalization using column families structure \| denormalization and storage in a single table \| complex attribute having structured value (binary)<br>Target_null_values -> missing fields \|null | Transformation_to_target_schema_20 = (<br>source_collection : table,<br>Target_collection : table,<br>intermediate_data : column,<br>intermediate_data_attribute : array of map,<br>Referencial_constrain :[« denormalization using column families structure»,« denormalization and storage in a single table” , “denormalization using structured values (binary) »]<br>Target_null_values : [« missing fields » , « null »]<br>)<br>• Transformation 20 rules from entity association model :<br>Source_entity -> transformation 2,3,4 rules<br>target_entity -> transformation 2,3,4 rules<br>intermediate_Attribute- > collection(aggregate(attribute)) as column<br>Association (n-n)-> denormalization using column families structure \| denormalization and storage in a single table \| complex attribute having structured value (binary) of intermediate collection in |

| | | | | | |
|---|---|---|---|---|---|
| | | | | | association collections Target_null_values -> missing fields \|null |
| | key value store (eg hbase) | Transformation_schema_4 = ( Table -> key_value map Row -> key_value pair, key = timestamp + primary , value = all the attributes stored as a blob Column -> None Referencial_constrain -> denormalization and storage of all attributes of a row as a value \| denormalization using structured values (binary) null_values -> absence of attribute value in the blob \|null ) Transformation_schema_5 = ( Table -> key_value map Row -> key_value pair, key = timestamp + primary + attribute_name, value = the attribute value Column -> None | Transformation_to_target_schema_12 = ( source_collection : key_value map, Target_data_attribute : value = all the target attributes stored as a blob, Referencial_constrain :[ « denormalization and storage of all attributes of a row as a value" \| "denormalization using structured values (binary)»] Target_null_values : [« absence of attribute value in the blob » , « null »] ) Transformation 12 rules from entity association model : Source_entity > transformation 5,6 rules Target_Attribute- > aggregate(attribute) as value Association (1-1)-> denormalization and storage of all attributes of a row as a value Target_null_values -> absence of attribute value in the blob \|null | Transformation_to_target_schema_16 = ( source_collection : key_value map, Target_data_attribute : value = all the target attributes stored as a blob, Referencial_constrain :[ « denormalization and storage of all attributes of a row as a value" \| "denormalization using structured values (binary)»] Target_null_values : [« absence of attribute value in the blob » , « null »] ) Transformation 16 rules from entity association model : Source_entity > transformation 5,6 rules Target_Attribute- > collection(aggregate(attribute)) as value Association (1-n)-> denormalization and storage of all attributes of a row as a value Target_null_values -> absence of attribute value in the blob \|null | Transformation_to_target_schema_21 = ( source_collection : key_value map, target_collection : key_value map, intermediate_data_attribute : value = all the target attributes stored as a blob, Referencial_constrain :[ « denormalization and storage of all attributes of a row as a value" \| "denormalization using structured values (binary)»] Target_null_values : [« absence of attribute value in the blob » , « null »] ) • Transformation 21 rules from entity association model : Source_entity > transformation 5,6 rules target_entity > transformation 5,6 rules intermediate_Attribute- > collection(aggregate(attribute)) as value Association (n-n)-> denormalization and storage of all attributes of a row as a value Target_null_values -> absence of attribute value in the blob \|null |

| | | Referencial_constrain -> denormalization by storing everything as key value pairs (binary) null_values -> missing key value pair \|null ) | | | |
|---|---|---|---|---|---|
| graph data model | Attribute graph (e.g neo4j) | Transformation_schema_6 = ( Table -> node label Row -> node Column -> field Referencial_constrain -> no referential constrains but associations are created as relationships null_values -> absence of attribute value\|null ) Transformation_schema_7 = ( Table -> node label Row -> node Column -> field Referencial_constrain -> denormalization using nodes nesting null_values -> absence of attribute value\|null ) | Transformation_schema_13 = ( source_collection -> node label Target_data_attribute -> map Referencial_constrain -> no referential constrains but associations are created as relationships Target_null_values -> absence of attribute value\|null ) Transformation 13 rules from entity association model : Source_entity -> transformation 7,8 rules Target_Attribute- > aggregate(attribute) as field Association (1-1)-> no referential constrains but associations are created as relationships null_values -> absence of attribute value\|null | Transformation_schema_17 = ( source_collection -> node label Target_data_attribute -> array of map Referencial_constrain -> no referential constrains but associations are created as relationships Target_null_values -> absence of attribute value\|null ) Transformation 17 rules from entity association model : Source_entity -> transformation 7,8 rules Target_Attribute- > collection(aggregate(attribute)) as field Association (1-1)-> no referential constrains but associations are created as relationships null_values -> absence of attribute value\|null | Transformation_schema_22 = ( source_collection -> node label target_collection -> node label intermediate_data_attribute -> array of map Referencial_constrain -> no referential constrains but associations are created as relationships Target_null_values -> absence of attribute value\|null ) Transformation 22 rules from entity association model : Source_entity -> transformation 7,8 rules target_entity -> transformation 7,8 rules intermediate_Attribute- > collection(aggregate(attribute)) as field Association (n-n)-> no referential constrains but associations are created as relationships null_values -> absence of attribute value\|null |

*Tableau 4.1 Table de transformation entre les modèles de données*

### 4.3.2 Transformations des attributs de types texte

Le cas d'utilisation peut nécessiter des transformations d'attributs complexes sérialisés sous forme textuelle en des attributs structurés. La première approche consistait à inférer à partir des données le

coût des transformations par l'API de transformation de données de Apache Spark. Mais vu que cette méthode n'est pas précise et qu'il est facile d'obtenir le plan de transformation avec Spark on propose d'utiliser la seconde approche.

Pour agréger les données et les exporter sous formes de bilans ou d'agrégats, il existe une solution qui consiste à stocker le résultat des requêtes dans des bases de données raffinées. On souhaite exploiter l'avancée en Big Data et en variant les modèles de données inspecter le stockage des données agrégés en utilisant la propriété des attributs complexes à la place du stockage en champ de texte. Notre point de départ est un ensemble de données complexe sous forme de texte. On utilise l'API de transformation de données de Spark pour transformer les données.

### 4.3.3 Transformations du schéma physique

Généralement les techniques les plus utilisées en gestion de données pour améliorer la performance sont les techniques d'indexation et de partitionnement.

#### *4.3.3.1 Indexation*

L'une des techniques les plus connues dans l'optimisation du traitement de la requête est l'indexation (cf. état de l'art).

| Système | Type d'index | Physical data layout | Fonctionnalité |
|---|---|---|---|
| PostgreSQL | Btree | Btree | (i) Utilisé pour les conditions ayant une comparaison d'inégalité. |
| | Hash index | Hash | (i) Utilisé pour les conditions ayant une comparaison d'égalité |
| | Gin | Indexe inversé | (i) permet d'indexer des attributs de type chaîne de caractère et permet de réaliser une recherche dans un texte |
| | Gist | Arbre de recherche généralisé | |
| Neo4j | single-property index | Linked list | Indexe sur un seul attribut |
| | Composite index | Linked list | Indexe sur un ensemble d'attribut |

*Tableau 4.2 Types d'indexes et compatibilité*

Selon notre méthode, nous associons à chaque système les indexes qu'il supporte comme le montre le tableau 4.2 (non exhaustif). Le module de transformation de notre architecture permet de simuler l'indexation d'un attribut après consultation de la couche de métadonnées et vérification des compatibilités. Finalement le module de prédiction prend compte de cette transformation et prédit son temps de réponse.

#### *4.3.3.2 Partitionnement*

Le partitionnement [SF12] est une méthode de division d'un ensemble de données et le partager entre de multiples bases de données physiques. Les partitions - sous-ensembles créés par cette technique – sont accédées et manipulées comme une seule base de données logique tout en assurant que les détails d'implémentation physique sont totalement gérés par le système et transparents pour l'utilisateur final. L'avantage principal du partitionnement est le passage à l'échelle des données et la performance des opérations de lecture. Avec cette technique, un système Big Data est capable de

gérer des ensembles plus volumineux et d'exécuter les requêtes plus rapidement vu qu'il y a moins de données à scanner par serveur.

Il existe deux catégories principales de partitionnement : le partitionnement vertical et le partitionnement horizontal. Il existe aussi un type de partitionnement dérivé : le partitionnement hybride.

| Système | Type de partitionnement | Catégorie | Distribué |
|---|---|---|---|
| PostgreSQL | Plage de valeurs | Horizontale | Pas nécessairement |
| | Liste | Horizontale | Pas nécessairement |
| | Hachage | Horizontale | Pas nécessairement |
| | Verticale | Verticale | Pas nécessairement |
| Phoenix | Salting | Horizontale | Oui |
| MongoDB | Hachage | Horizontale | Pas nécessairement |
| | Range | Horizontale | Pas nécessairement |
| Dynamodb | Consistent hashing | Horizontale | Oui |

*Tableau 4.3 Types de partitionnement et compatibilité*

Nous proposons comme perspective d'étudier les configurations du schéma physique en relation de la compression et la sérialisation des données (avro , parquet ou json pour le stockage dans un système de fichier distribué comme HDFS). Ces configurations sont représentées dans les métadonnées systèmes et il est facile en suivant notre méthode d'étendre la recommandation à ce niveau de granularité.

### 4.3.4 Transformation par répartition des données

Il existe un autre type de transformation qui consiste à fragmenter les collections de données verticalement. Si le système supporte le partitionnement vertical, la transformation est considérée comme transformation du schéma physique. Sinon, la transformation est une séparation de la collection de départ en plusieurs sous-collections distinctes en constituant des groupes d'attributs de la collection.

Par exemple, pour un nœud de Neo4j composé de n = p+q attribut on propose la transformation en deux nœuds un premier avec p attributs et un deuxième avec q attributs.

### 4.3.5 Transformation de plan logique du workload et réécriture de requête

Dans cette section, nous présentons la méthode adaptée pour transformer un plan logique extrait à partir du workload et de le réécrire en tenant compte de la diversité des systèmes considérés dans l'espace de solution.

#### *4.3.5.1 Ordre des opérateurs binaires*

Dans notre approche exhaustive, nous examinons tous les cas de plan de requête possibles. En s'inspirant des techniques d'optimisation dans les systèmes d'intégration, nous énumérons les plans en considérant différents ordres pour les opérateurs binaires, notamment les jointures et les unions.

L'optimisation des jointures est un des défis les plus connus et les plus étudiés dans le domaine de l'optimisation de requêtes dans les bases de données. Plusieurs heuristiques ont vu le jour, parmi lesquelles la variation de l'ordre des opérateurs.

#### *4.3.5.2 Vues matérialisées*

Apache Calcite permet de créer des vues matérialisées dans les systèmes qu'il supporte. Ce type de transformation nous permet d'exécuter le placement même dans les systèmes qui ne vérifient pas le

critère de faisabilité pour une partie du workload. Les vues matérialisées sont des techniques qui ont été inventées dans les systèmes relationnels pour optimiser la performance des requêtes. Une vue est généralement un sous ensemble d'une relation de départ créée pour des accès fréquents en lecture exclusivement (une vue correspond au résultat d'une requête). Le sous ensemble de données dans cette technique est stocké sur le disque, d'où la notion de matérialisation. Cette technique permet de traiter moins de données et ainsi minimise le temps de réponse. L'intérêt principal d'utiliser cette méthode est de remplacer les calculs complexes comme une jointure n-aire ou bien une agrégation complexe par une simple lecture de données pré-calculées.

##### *4.3.5.3 Envoi des prédicats à la source de données (Push down predicate)*

Dans les travaux existants sur les systèmes d'interrogation de requête hybrides les seuls opérateurs qui sont envoyés à la couche inférieure sont des opérateurs de filtrage et de projection. Dans notre approche et pour examiner exhaustivement tous les cas possibles de placement de données nous proposons d'envoyer également les opérateurs de jointure et d'agrégation pour une exécution au niveau physique et comparer le résultat avec l'exécution hybride.

### 4.3.6 Transformation de métadonnées

Il est possible de transformer une partie des métadonnées qui sont nécessaires pour l'estimation du temps d'exécution des requêtes et qui sont ainsi indispensables pour le module de recommandation de placement.

## 4.4 Optimisation du placement d'un workload

Dans cette section, nous détaillons le module d'optimisation dans DWS.

### 4.4.1 Principe

Dans notre approche nous proposons d'optimiser le schéma physique des données après l'optimisation logique et hybride du workload. Les solutions d'optimisation que nous utilisons sont : (i) le choix de l'ordre optimal des jointures, (ii) l'envoi des opérations d'agrégations, de jointure et de filtrage à la couche inférieure pour minimiser le coût de transfert des données entre les systèmes et favoriser l'exécution non hybride d'une partie de la requête par rapport à l'exécution hybride. Les transformations par indexation et partitionnement améliorent davantage la performance et optimisent ainsi le placement.

### 4.4.2 Défis d'optimisation dans DWS

Les défis d'optimisation dans DWS sont surtout générés par le problème d'hétérogénéité des modèles des données et le besoin de réécriture de requête qui en découle. Il faut être capable de traduire les opérations du langage utilisé dans DWS (on a choisi le langage SQL) vers tous les langages et les APIs des systèmes supportés et on a besoin de trouver comment on passe d'une structure de données à une autre (par exemple d'une table à un graphe).

L'hétérogénéité des systèmes à comparer est un autre problème majeur : les coûts estimés de performance sont exprimés d'une manière différente suivant les systèmes et certains d'entre eux ne calculent pas de coût de requête et utilisent des approches d'optimisation plus simples. En effet, l'interrogation des données dans plusieurs datastores est basée sur les APIs. Elle n'est pas déclarative et son architecture est différente de celle des architectures d'évaluation de requête traditionnels qui passent par les étapes parsing, optimisation et exécution. L'exécution du workload dans ces data stores est un simple appel de fonctions. D'où l'utilisation des approches traditionnelles d'optimisation n'est pas évidente pour certains systèmes. Dans certains cas, le modèle de coût proposé par ces systèmes est limité (par exemple, dans le cas du modèle de coût de Catalyst ou l'optimiseur de requête de Spark) ou bien très simplifié (par exemple, le plugin Calcite-MongoDB). Finalement la transformation des

données et la génération de celles-ci dans DWS sont des défis principaux vu la diversité des solutions en Big Data. Il faut adresser ce problème et trouver une solution efficace pour la génération des transformations.

### 4.4.3 Espace de recherche de DWS

Le parsing des requêtes entrées dans DWS est la première étape d'optimisation. Dans cet étape, le système transforme l'expression de requête en un ensemble de graphes d'opérateurs appelés *plans de requêtes logique*s (cf état de l'Art). L'exécution de chacun de ces plans donne les mêmes résultats mais ils n'offrent pas les mêmes performances.

Cette étape est appelée l'étape de génération de plans de requête ou bien de construction de l'espace de recherche de plan. Dans notre approche on utilise le parseur d'un moteur de requêtes hybride de l'état de l'art (celui d'Apache Spark), et à partir des plans logiques générés par ce parseur, nous construisons des combinaisons plus complexes de plans en variant les opérateurs et les propriétés des systèmes pour obtenir à la fin un espace de recherche pour identifier le placement optimal qui tient compte des systèmes de stockage, des traitements, des schémas de partitionnement, de la solution d'indexation et de distribution des données.

Les critères de faisabilité et de conformité permettent de limiter l'espace de recherche. Ils correspondent à des heuristiques que nous utilisons pour éliminer les plans de requêtes qui ne sont pas intéressants pour l'application. Par contre, les transformations permettent d'explorer davantage l'espace de recherche. Ils permettent de chercher davantage des solutions d'exécution du workload. On décompose aussi le plan des requêtes du workload pour explorer d'autres solutions de placement et/ou de transformation.

#### *4.4.3.1 Spécification de l'espace de recherche*

L'espace de recherche est un ensemble de n schémas de placements. Chaque schéma de placement est relatif à un workload. Il a comme structure un graphe orienté asymétrique (DAG) et chaque nœud du graphe contient les propriétés du candidat de placement relatifs aux ensembles de données et aux requêtes.

En considérant l'exemple de départ qui était décrit dans le cas d'utilisation des travaux de nos recherches (cf. introduction), je présente un exemple d'espace de recherche pour le problème de placement de données (figure 4.6). Ce graphe a comme point de départ un des ensembles de données de départ (sélectionné aléatoirement) et comme point d'arrivée le résultat du workload. Les nœuds du graphe sont des requêtes ou bien des transformations. Ils sont représentés par un schéma de placement et modifient des propriétés de ce schéma crée une nouvelle solution. Dans l'espace de recherche il existe des solutions qui ne sont pas valides. L'algorithme doit être capable d'éliminer ces solutions le plus tôt que possible.

Dans cet exemple on considère le placement pour seulement trois systèmes : PostgreSQL, Hbase et Neo4j. Par souci de simplification, on va limiter les transformations à l'indexation sur tous les attributs, le partitionnement horizontal et quelques transformations du modèle de données.

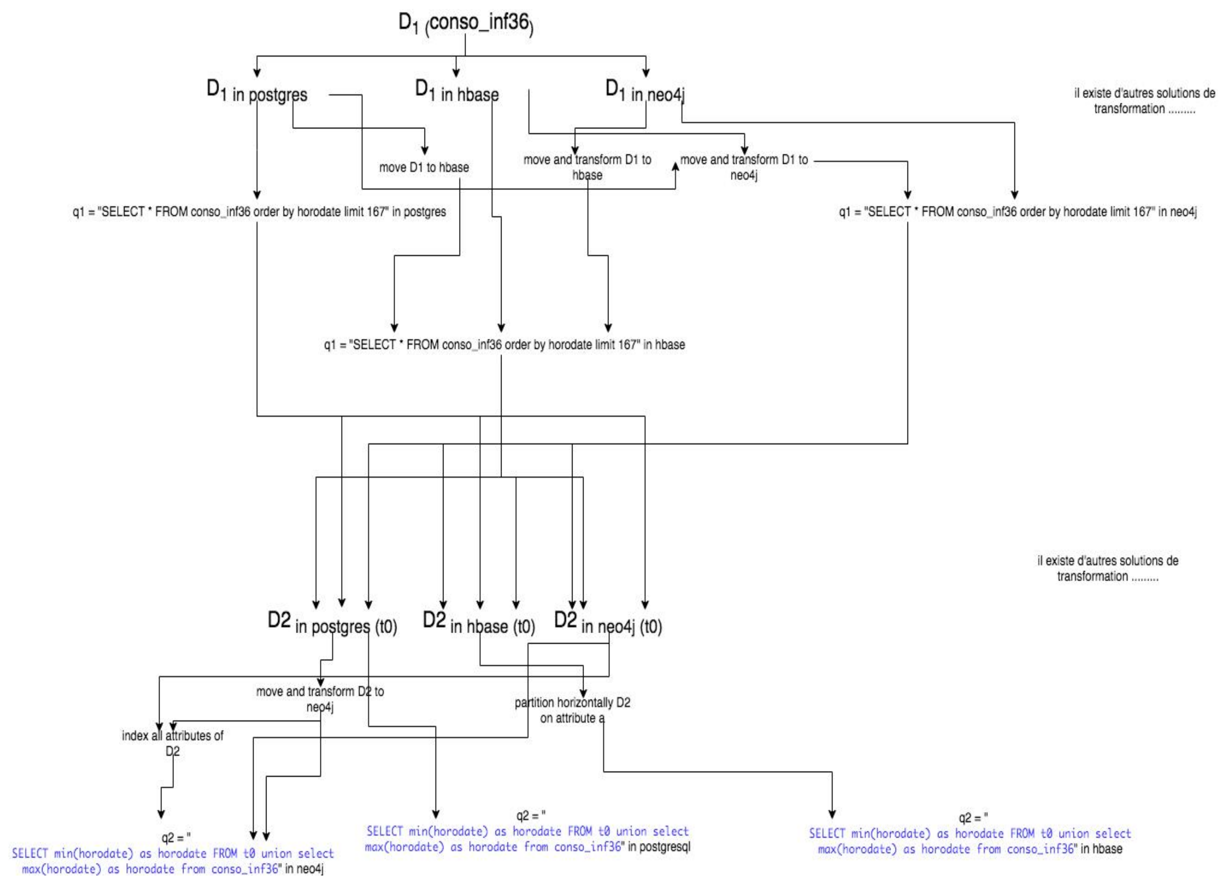



*Figure 4.6* Exemple de l'espace de recherche dans DWS

#### *4.4.3.2 Enumérer les schémas de placement*

Les étapes sont les suivantes :

1. Construire un schéma de placement initial pour workload à partir des requêtes de l'application. Les nœuds du schéma initial sont composés seulement d' informations relatives aux requêtes et aux ensembles de données qui composent le graphe.
2. Récupérer à partir de la base de métadonnées les propriétés des candidats de placement
3. Associer à chaque nœud les candidats de placement
4. Parcourir le graphe une seconde fois pour découvrir les transformations complexes possibles
5. Ajouter les nœuds de transformation, renseigner les liens transformation et calculer les coûts

### 4.4.4 Recherche de solution optimale pour un workload

Pour trouver la solution optimale il faut parcourir le graphe et sélectionner la solution optimale qui produit le résultat du workload et garantit la performance du traitement.

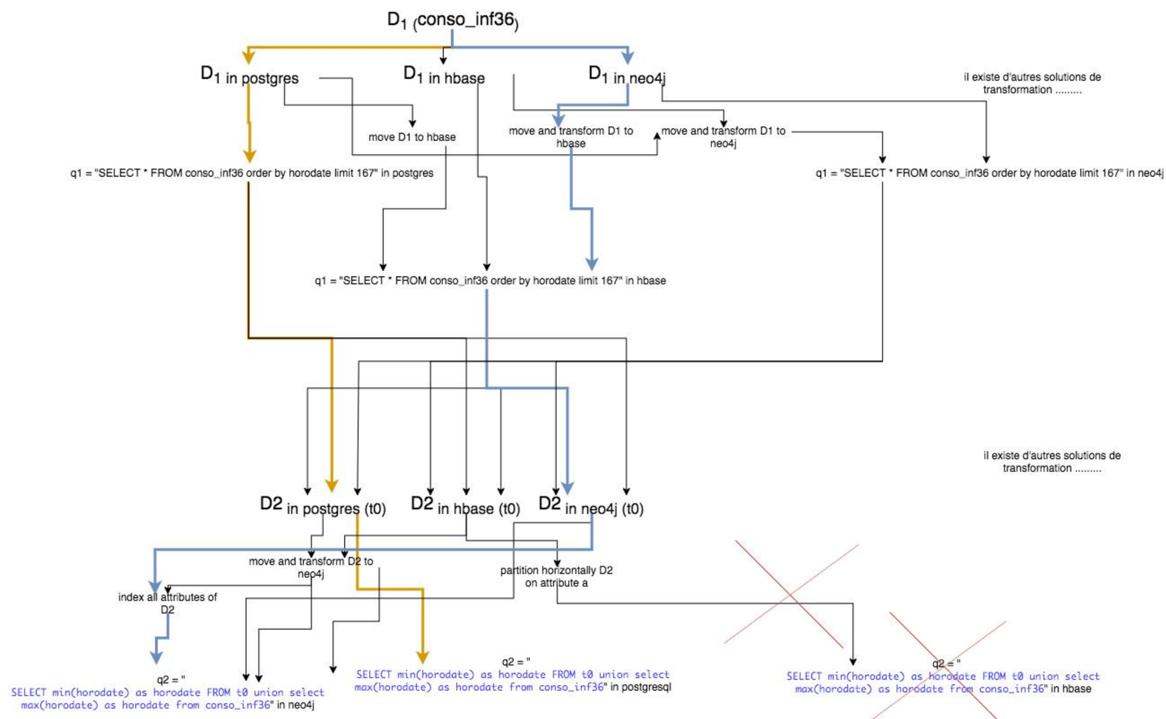



*Figure 4.7* Exemple de deux solutions de placement de données dans DWS

La figure 4.7 illustre deux exemples de solutions possibles de placement de données pour un workload. La solution optimale, qui présente le moindre coût, peut être la première solution (en orange) qui est située dans une profondeur égale à 4 ou bien la deuxième (en bleu) qui se trouve à un degré de profondeur égale à 6.

L'algorithme de recherche de solution optimale doit parcourir les chemins du graphe un à un jusqu'à atteindre son objectif. Pourtant, il existe plusieurs stratégies de recherches. Laquelle doit-on sélectionner ?

#### 4.4.4.1 Recherche exhaustive

Dans le deuxième chapitre nous avons comparé les techniques de recherche de solution optimale dans le domaine d'évaluation de requête et nous avons sélectionné la recherche exhaustive. Pour un workload W, dans l'étape de construction de l'espace de recherche ER l'algorithme génère tous les graphes d'opérateurs logiques qui correspondent au workload d'entrée. Ensuite l'algorithme parcourt tous les éléments de ER, compare les performances des différentes combinaisons des candidats de placement et détermine la solution optimale.

Dans la littérature il existe plusieurs algorithmes de recherche. Le choix d'utiliser la recherche exhaustive à la place de méthodes d'optimisations plus légères comme la programmation dynamique est principalement motivé par la nature de l'application. On est dans un cadre où le temps passé pour recommander le meilleur placement n'est pas critique (off-line), contrairement aux solutions d'optimisations on-line qui doivent sélectionner le plan optimal et l'exécuter en un temps minimal. On peut tolérer le choix d'algorithmes qui demandent plus de temps. Pour l'optimisation des requêtes on-line, où l'espace de recherche est plus petit, les algorithmes tels que la programmation dynamique sont plus appropriés.

#### 4.4.4.2 Réécriture de requête et envoie à la couche de stockage

Dans les solutions NoSQL l'optimisation des requêtes est quasi inexistante car généralement l'optimisation est spécifique aux requêtes déclaratives. L'efficacité de l'exécution du workload dans ce

cas est donc corrélé avec le code d'accès aux données ou bien par la requête déclarative (exemple requête Cypher de Neo4j) spécifiés par l'utilisateur.

La réécriture des requêtes est une brique très importante dans notre contexte puisque les requêtes en Big Data sont hétérogènes et parfois la performance d'exécution de la requête dépend de l'expression de celle-ci. Ce résultat est observé dans des systèmes NoSQL qui n'utilisent pas de modèle de coût pour l'optimisation des requêtes comme MongoDB ou bien si leur modèle de coûts n'est pas raffiné, voire simpliste comme Neo4j.

En effet, les techniques de réécriture de requêtes interviennent généralement pour transformer des plans logiques ou bien des requêtes vers une structure ou une expression qui facilitent son optimisation. Ils sont généralement utilisés dans des systèmes multi-stores qui proposent d'exécuter des requêtes sur des données distribuées entres plusieurs solutions de stockages différentes (voir état de l'art)

Dans notre approche, nous récupérons le plan logique des requêtes et, à partir des opérateurs de ce dernier, nous générons l'expression de la requête dans les systèmes de placement cibles.

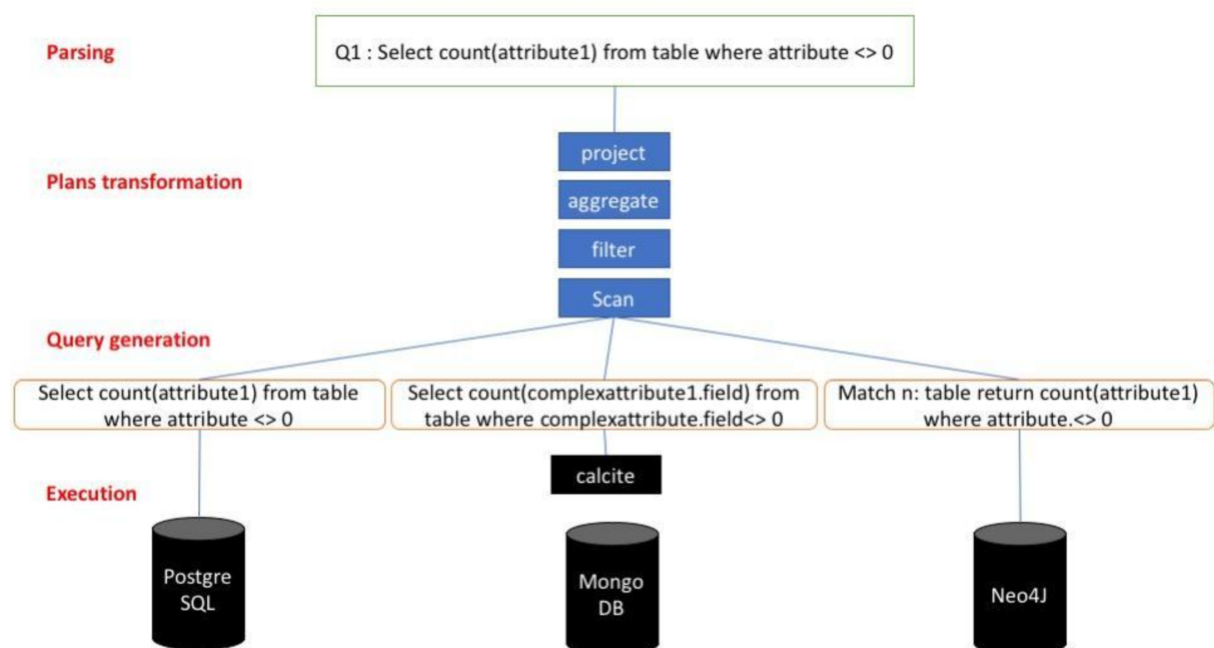



*Figure 4.8* Réécriture de requête dans un écosystème Big Data

DWS s'adapte avec différents modèles de données et il prend en entrée des workloads qui ont différents modèles de requêtes. Ainsi, nous proposons trois types d'interfaces : (i) une interface de réécriture de requête vers le langage SQL standard, par exemple le SQL supporté par SparkSQL, (ii) une interface de réécriture de requête vers une version étendue du langage SQL, comme la version supportée par Apache Calcite et (iii) une interface de réécriture de requête vers un langage ou une API NoSQL, comme le langage Cypher de Neo4j.

#### *4.4.4.3 Problème de réconciliation*

L'un des défis les plus importants dans notre étude est de garantir la performance de l'exécution dans un écosystème Big Data de tout un workload qui présente un ensemble de requêtes qui sont dépendantes, par précédence, ou bien qui font intervenir plusieurs ensembles de données dans des requêtes différentes.

Un exemple de workload sera l'application de calcul de bilan de consommation d'énergie des clients Enedis que nous avons présentée dans notre cas d'utilisation industriel (cf- motivation – Cas d'usage). L'implémentation de cette application est un workload qui est exécuté sous forme de script SQL et qui est représenté dans la figure suivante :

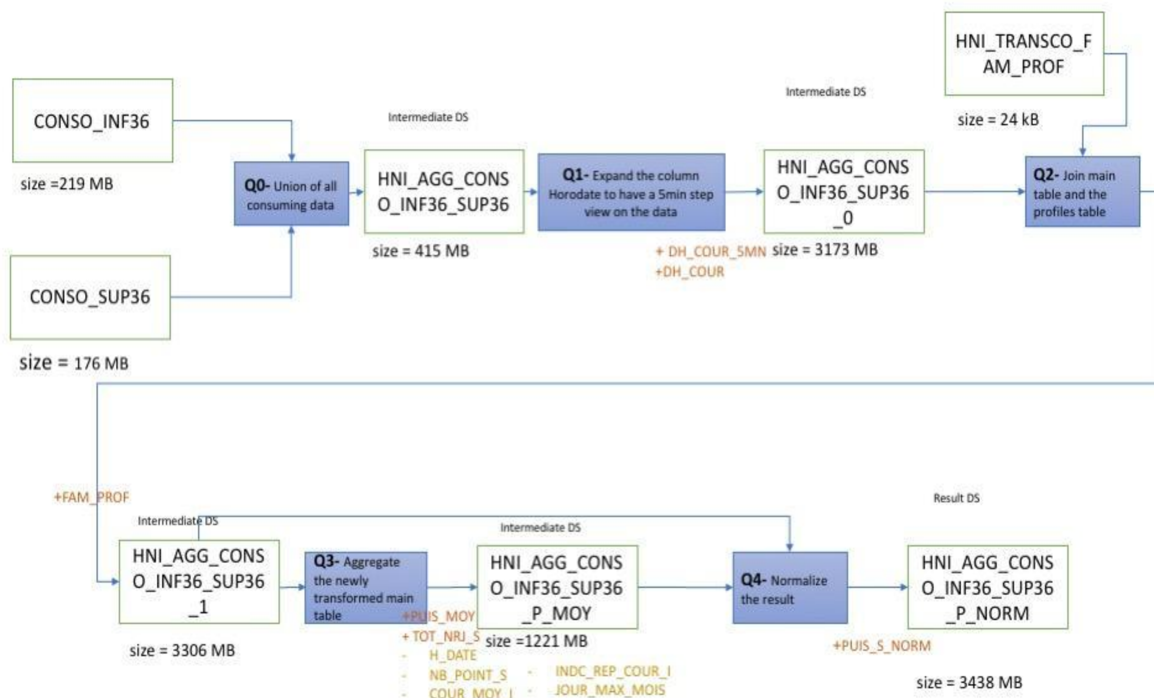



*Figure 4.9* Graphe de traitement du workload : « calcul de bilan de consommation d'énergie des clients enedis»

Dans cet exemple, les requêtes du workload communiquent des données entre elles. Ces ensembles sont appelés des « résultats intermédiaires » et ils sont généralement nécessaires pour la traçabilité et l'analyse de la cohérence des applications.

En effet, en tant que bonnes pratiques dans la gouvernance des données, il est nécessaire d'avoir une vision évolutive sur la donnée. Cette pratique nous permet de répondre à la question : « d'où viennent mes données ? ». Renforcé avec un service de persistance et de visualisation de la lignée des données (en anglais *data lineage*) – cf. section services de métadonnées –, elle assure le suivi du cycle de vie des données de leur création à leur stockage, ainsi qu'à leur utilisation. Son but est de faciliter la gestion de la qualité des données et la reprise en cas d'erreurs.

Dans notre approche, on propose de stocker les résultats intermédiaires dans les cas suivants :

- à la demande du gestionnaire des données : (i) dans le lac de données et dans un système qui garantit la performance de la lecture et de l'écriture des données, ou bien (ii) dans un espace projet si le résultat intermédiaire est intéressant du point de vue métier ;
- si la création de ce résultat intermédiaire améliore considérablement la performance du traitement.

Pour intégrer les pratiques de traçabilité et optimiser davantage l'exécution des applications Big Data de l'écosystème, on permet aux utilisateurs de créer des règles de conformité qui exigent la création d'un résultat intermédiaire et en même temps on donne à l'algorithme de placement le contrôle de la création ou non des autres résultats intermédiaires dans le but d'optimiser l'exécution ou bien le stockage.

Pour illustrer notre démarche, on considère le workload de la figure 4.9 et on suppose que la persistance de l'ensemble : « HNI_AGG_CONSO_INF36_SUP36 », qui représente les données de consommation unifiés, est nécessaire et imposé par une règle de conformité.

Suite à cette hypothèse, la création de cet ensemble de données est donc obligatoire et notre graphe de traitement peut subir par exemple les transformations suivantes :

<CONSO_INF36 , CONSO_SUP36> - (Q0)- <HNI_AGG_CONSO_INF36_SUP36>-(Q1)-< HNI_AGG_CONSO_INF36_SUP36_0, HNI_TRANSCO_FAM_PROF>-(Q2)-< HNI_AGG_CONSO_INF36_SUP36_1>-(Q3)-< HNI_AGG_CONSO_INF36_SUP36_P_MOY,HNI_AGG_CONSO_INF36_SUP36_1>-(Q4)-< HNI_AGG_CONSO_INF36_SUP36_P_NORM>

<CONSO_INF36 , CONSO_SUP36> - (Q0)- <HNI_AGG_CONSO_INF36_SUP36, HNI_TRANSCO_FAM_PROF >-(Q1,2)-< HNI_AGG_CONSO_INF36_SUP36_1>-(Q3)-< HNI_AGG_CONSO_INF36_SUP36_P_MOY,HNI_AGG_CONSO_INF36_SUP36_1>-(Q4)-< HNI_AGG_CONSO_INF36_SUP36_P_NORM>

<CONSO_INF36 , CONSO_SUP36> - (Q0)- <HNI_AGG_CONSO_INF36_SUP36, HNI_TRANSCO_FAM_PROF >-(Q1,2)-< HNI_AGG_CONSO_INF36_SUP36_1>-(Q 3,4)-< HNI_AGG_CONSO_INF36_SUP36_P_NORM>

…

En plus de ces contraintes de conformité, il est important de noter que le placement des données pour un workload peut faire intervenir plusieurs ensembles de données dans des requêtes différentes. Optimiser individuellement ces ensembles n'est pas toujours optimal. En effet, si pour un ensemble de donnée S une requête Q1 est faisable dans le système A et B, mais le système A offre une meilleure performance pour l'exécution de Q1, et si pour ce même ensemble S une requête Q2 doit être exécutée, mais elle n'est faisable que dans le système B, alors pour exécuter les deux requêtes Q1 puis Q2 on doit placer S dans A exécuter Q1, déplacer S dans B puis exécuter Q2. On aurait pu avoir une meilleure performance si on avait choisi dès le début d'exécuter la requête Q1 dans B, car on économiserait le transfert de S et l'éventuelle transformation qui en découle.

Un autre cas d'étude similaire consiste à déterminer pour un ensemble de données la solution de placement de données qui garantit l'efficacité du traitement pour un premier workload – par exemple l'application de bilan de consommations – mais qui résulte en une dégradation de la performance pour un autre workload – par exemple une application de mise à jour des données de consommation.

On appelle ce sous-problème la réconciliation des solutions de placement pour tout un workload. Comme solution pour la réconciliation des solutions de placement de données pour les requêtes d'un workload, on propose de développer un algorithme de recherche de placement pour tout un workload. On propose aussi d'intégrer les opérateurs de copie et de déplacement des données dans le plan du workload, et l'intégration des coûts de ces mouvements dans l'estimation de la performance du workload pour chaque placement candidat. On fait appel à l'opérateur de copie si un opérateur logique du workload est exécuté souvent sur un ensemble de données à placer et que cet opérateur n'est pas supporté couramment dans les autre systèmes (par exemple un opérateur de calcul de données orienté graphe), ou bien il existe une règle de conformité qui restreint le déplacement de cet ensemble. On utilise l'opérateur de déplacement des données si le premier emplacement n'est pas fréquemment optimal.

Les transformations ont généralement un coût important surtout dans le cas de déplacement des volumes importants de données d'un système à un autre. Dans l'algorithme, il faut trouver la bonne

modélisation pour prendre en compte les transformations, mais ne pas pénaliser les solutions qui ont besoin de ce type de configuration. Il faut choisir soigneusement la bonne modélisation. Dans ce travail nous n'avons pas examiné en détail les méthodes d'estimation de coût des transformations et nous considérons cette partie comme une perspective importante pour ce travail.

### 4.4.5 Estimation de la performance pour le placement

Pour estimer l'impact du placement des données sur les performances de la charge de travail, deux approches possibles sont suggérées :

(i) La première proposition consiste à définir un modèle de coûts de haut niveau prenant en charge tous les systèmes Big Data de DWS.

(ii) La deuxième utilise les coûts calculés par la couche de stockage et de traitement dans DWS et trouve une solution pour comparer ces coûts hétérogènes.

Les deux approches sont également complexes. La première approche consiste à trouver la bonne abstraction et à concevoir à partir de zéro un modèle de coûts élastique capable d'estimer et de comparer les performances d'exécution d'opérateurs hétérogènes. Cette approche a besoin d'abord d'une unification des coûts hétérogène (cf. Coût du placement de données) :

- Structure des vecteurs de coûts : homogénéiser les vecteurs de coûts
- Unités de coût : Trouver une approche pour réécrire les valeurs de coût dans une unité de coût unifiée
- Statistiques d'optimisation
- Sémantique d'attribut de coût
- Formules mathématiques de modèle de coût

Le modèle de coût est un ensemble de formules mathématiques associées à chaque opérateur physique du plan de requête (cf. chapitre 2, modèles de coût). Il est utilisé pour estimer les performances de l'exécution de la requête. Les modèles de coûts sont composés de différents paramètres et sont généralement exprimés sous la forme d'un vecteur composé de plusieurs dimensions : coût CPU, coût d'entrée/sortie, coût de communication, cardinalité des résultats intermédiaires, … Toutefois, lorsque nous examinons l'état de l'art dans le domaine des Big Data et des bases de données, nous constatons que les modèles de coûts sont très spécialisés et très différents d'un système à l'autre. Leur objectif est de comparer les coûts d'exécution de différents plans d'exécution d'une requête dans un seul magasin de données. Ils ne sont pas conçus pour comparer les performances d'exécution des requêtes entre des systèmes différents, et ils sont hétérogènes en termes d'unités permettant d'exprimer la valeur de coût, de formules de coût et de représentation du vecteur de coût.

Dans cette approche, une solution réalisable consiste à définir pour chaque groupe d'opérateurs physiques une formule de coût et à utiliser cette formule pour estimer la performance de la charge de travail des systèmes cibles, en connaissant les caractéristiques de ces systèmes. Dans ce cas, l'optimiseur extrait des métadonnées du système : les opérateurs physiques pris en charge par le système cible, les propriétés de stockage des données et leurs statistiques (par exemple, la taille de la mémoire de bloc, la structure de données de l'index, l'accès principal pouvant être un hachage, basique ou index…).

Dans le tableau suivant, nous présentons pour un ensemble d'opérateurs physiques Big Data des exemples de métadonnées de systèmes influant sur le coût de performances et la complexité de l'opérateur :

| Physical operator | Systems metadata | Complexity |
|---|---|---|
| Full collection scan | _ | O(n) |
| Hash scan | Access type = hashing Hashing type = consistent hashing | O(N/k) ou k est le nombre de nœuds dans la grappe (*cluster*) |
| Index scan | Index data structure = btree | O(log(n)) |
| Sort | _ | O(nlog(n)) |
| Aggregate | _ (In the absence of indexes the cost of aggregates are equivalent to the cost of sort) | O(nlog(n)) |

*Tableau 4.4 Exemples d'opérateurs physique et leurs complexités en fonction des métadonnées systèmes*

Trouver les modèles de coûts pour chaque opérateur, pour chaque système n'est pas une solution réaliste : c'est une approche exhaustive, elle souffre de plusieurs limites, principalement : (i) elle ne peut pas passer à l'échelle. La recommandation sera limitée aux systèmes supportés dans DWS et il sera difficile d'ajouter d'autres systèmes, (ii) cette solution peut engendrer des erreurs de comparaison de performance vu que les techniques basées sur les coûts sont caractérisés par une inexactitude / imprécision (*inaccurancy*) surtout dans les moteurs d'exécution immatures et émergeants, (iii) finalement, le modèle de coût produit sera non réutilisable. Il serait très spécifique à DWS et il ne serait pas possible de l'intégrer dans d'autres projets.

La seconde approche, et celle que nous adoptons, permet de comparer les coûts des requêtes qui sont calculés par les systèmes qui les ont exécutées. Dans cette approche, on doit faire face à la difficulté de comparer des modèles de coûts hétérogènes qui n'ont pas été conçus pour estimer la performance inter systèmes. Il est nécessaire dans ce cas d'extraire des différents systèmes les coûts de l'exécution d'une même requête, transformer ces coûts en valeurs comparables et conclure par la sélection du plan d'exécution qui présente le moindre coût.

Deux solutions sont envisageables pour transformer le coût des charges de travail (*workloads*) hétérogènes :

- Utiliser des formules mathématiques pour obtenir des valeurs de coûts ayant la même sémantique et la même unité

- Utiliser des techniques d'apprentissages pour prédire le temps de réponse qui est la variable choisie pour représenter la performance. L'algorithme d'apprentissage utilisera les différents attributs du vecteur de coût associé à d'autres valeurs pour déterminer la valeur de la variable de performance que nous avons choisie.

Pour favoriser l'extensibilité et la réutilisation des travaux existants, nous proposons d'exploiter la deuxième approche pour l'évaluation du critère de performance dans DWS. Certains problèmes subsistent néanmoins lorsqu'on utilise des modèles de coûts en Big Data. En effet, dans les solutions Big Data, la prise en charge de l'optimisation basée sur les coûts n'est pas généralisée. Les moteurs de requête / traitement en Big Data ne supportent pas nécessairement l'optimisation de requêtes déclaratives basée sur des modèles de coûts et de statistiques sur la répartition des valeurs. Par exemple, certains systèmes tels qu'Apache Spark planifient l'exécution de la requête en utilisant principalement des méthodes heuristiques et des estimations des cardinalités des résultats intermédiaires dans le cas de l'optimisation de l'ordre de jointure. D'autre part, de nombreux systèmes comme Apache Hbase ne prennent pas en charge l'optimisation native des requêtes en fonction de l'estimation des coûts, mais offrent la possibilité d'utiliser des systèmes externes pour activer le support SQL. Des exemples de ces systèmes externes sont multistores comme Apache Drill, des

systèmes spécialisés comme Apache Phoenix, et des optimiseurs hybrides comme Apache Calcite. La plupart de ces systèmes utilisent le langage de requête SQL pour assurer l'interface avec le magasin, car l'utilisation de ce langage est largement maîtrisée par la communauté. D'ailleurs, l'exécution des requêtes dans des data stores comme Hbase n'est pas faite en utilisant un langage déclaratif mais plutôt une API d'accès. L'optimisation ne serait plus une optimisation de requête qui a une architecture classique composé d'un parseur, un optimiseur et un générateur de code mais plutôt un optimiseur de code qui a comme objectif de choisir les opérateurs physiques correspondant à des appels de l'API du niveau logique. Elle n'utilise pas la technique d'estimation de coût des opérateurs pour évaluer leurs performances et sélectionner les plus efficaces.

Pour les systèmes dont la stratégie d'exécution de requête n'a pas de modèle de coût, nous proposons de prédire les performances de l'exécution de la requête par apprentissage à partir d'une analyse des métadonnées. L'hétérogénéité des modèles de coûts (voir l'architecture DWS) est un autre défi que pose l'utilisation des modèles de coûts dans l'évaluation des performances.

Comme nous en avons discuté précédemment, la comparaison de ces coûts hétérogènes est un défi important qui ne sera probablement pas résolu rapidement. Une approche possible consiste à développer un modèle de coût prenant en charge divers systèmes de données volumineuses. Un cadre existant qui adapte cette approche est Apache Calcite. La deuxième approche consiste à utiliser des techniques d'apprentissage automatique pour une estimation comparable des performances de moteurs de traitement de données hétérogènes.

Dans cette expérience et pour évaluer les implications en termes de performances du placement de données, nous utilisons la seconde approche. Le choix de travailler avec la deuxième approche est motivé par l'architecture de l'écosystème de données de notre cas d'utilisation.

Apache Hadoop, Apache Spark, Apache Hbase et Neo4J font partie des domaines d'intérêt de cet écosystème. Pour préparer l'algorithme d'évaluation des performances, nous proposons d'enregistrer les coûts d'un ensemble de requêtes ainsi que leurs temps d'exécution dans une base de code. Dans ce contexte, nous utilisons la base de code pour obtenir un temps d'exécution de requêtes ayant des plans logiques similaires à la charge de travail simulée dans les systèmes de placement candidats.

Les performances d'une requête sont généralement mesurées à partir d'informations relatives à l'utilisation des ressources et au temps d'exécution de workload. Elles sont évaluées à l'aide d'estimations de « executorRunTime », « executorCpuTime », « io time » et du temps de communication. Les statistiques d'horloge réseau et entrées/sorties sont importantes car le transfert et la lecture de grandes quantités de données sont ce qui fait la différence dans les systèmes Big Data. Par exemple, MapReduce souffre de cette utilisation intensive du réseau et des entrées/sorties. A chaque étape du traitement, le serveur distribue et matérialise sur disque le résultat intermédiaire. Spark améliore cela en structurant l'exécution dans les DAG et en favorisant la réutilisation de certaines parties de la charge de travail afin d'améliorer l'exécution et la communication.

Dans les systèmes qui ne prennent pas en charge le parallélisme et la distribution, les performances sont généralement mesurées à l'aide du temps d'exécution, du temps de calcul et du temps d'entrée/sortie. Même si ces systèmes ont des représentations similaires de leurs vecteurs de coûts, ils sont toujours hétérogènes en termes de structure : les vecteurs de modèles de coûts ont des attributs différents, des attributs différents, et en termes de formule de calcul et de l'unité.

Pour prédire le critère de performance que nous avons sélectionné (temps de réponse d'une requête), deux solutions sont alors envisageables :

- Une approche en boîte blanche où nous connaissons le modèle de coût du système de données et la seule difficulté qui reste est de trouver une relation entre ces modèles afin de pouvoir comparer leurs valeurs. Les optimiseurs exposent les coûts des requêtes en utilisant deux types de méthodes : « explain » ou bien « profile ». « Explain » est utilisé pour ne faire que la phase d'optimisation d'une requête et produit le plan optimal ainsi que le coût des requêtes estimé en utilisant les statistiques collectés par le système, et cela sans exécuter la requête. « Profile » exécute réellement la requête et collecte des métriques, comme le temps d'exécution dans Apache Drill, ou bien dB hits (unité d'accès au stockage) dans Neo4j.
- Dans notre approche, nous utilisons Apache Calcite pour estimer le coût d'exécution dans les systèmes Big Data qui n'utilisent pas l'optimisation basée sur le coût pour trouver le plan optimal ou bien on récupère le coût directement du système lui-même. Le modèle de coût d'Apache Calcite est un vecteur composé de trois dimensions : <Coût Cpu, Coût d'entrée/sortie, cardinalité estimés>. Apache Calcite ne prend pas en charge un ensemble de data stores comme Apache HBase et Neo4j.
- Une approche boîte noire pour estimer le coût d'exécution de la charge de travail dans un système de données qui ne prend pas en charge l'optimisation basée sur les coûts (CBO). Dans cette approche, nous exploitons pour chaque requête
    - les statistiques de jeu de données et de requête,
    - la durée d'exécution de la requête
    - la durée d'exécution de chaque opérateur
    - Cette approche prend en entrée un plan de requête et génère une comparaison juste sans estimer le coût de la requête. Une approche d'apprentissage automatique peut être envisagée dans ce cas.

#### *4.4.5.1 Algorithme d'apprentissage automatique pour prédire le temps de réponse pour tous les systèmes Big Data*

La prédiction du temps de réponse est la solution qu'on a sélectionnée comme méthode pour estimer le coût d'exécution d'un plan de requête. Les variables sur lesquelles se basent la prédiction varient d'un système à un autre. En effet, dans le cas d'exécution d'un workload dans un système qui n'utilisent pas les méthodes d'optimisation des requêtes basés sur les coûts, on propose de baser notre étude sur les métadonnées qui influencent l'exécution des requêtes et qui permettent au modèle de prédiction de déterminer leurs temps de réponse.

Par contre, dans le cadre des optimiseurs de requêtes qui supportent l'estimation des coûts de l'exécution, nous proposons d'utiliser ces coûts en plus des métadonnées pour prédire le temps de réponse. Dans cette section nous détaillons la méthode adaptée pour déterminer les variables d'apprentissage.

Contrairement aux SGBD relationnels où l'optimisation de requête basé sur l'estimation de coût est largement adaptée et étudiée avec le langage SQL (cf chapitre 2), les systèmes de traitement en Big Data n'utilisent pas forcément cette approche pour estimer la performance de l'exécution des requêtes.

Ainsi, on peut considérer les systèmes de traitement de données en 4 catégories :

(i) des systèmes qui ont un optimiseur basé sur un modèle de coût mature et dont les estimations sont généralement correctes, par exemple les systèmes relationnels matures comme Oracle ou PostgreSQL,

(ii) des optimiseurs dont le modèle de coût est simpliste et le développement sur ces systèmes est un projet en cours, comme le projet Apache Calcite et son intégration avec MongoDB,

(iii) des systèmes qui basent leur optimisation par estimation de cardinalités et des mesures de nombre d'accès disque, comme Neo4j ou l'optimiseur de requête du module SQL d'Apache Spark intitulé « Catalyst »,

(iv) et des systèmes qui utilisent les heuristiques ou des règles d'optimisation, comme l'ancienne version d'Apache Spark.

À partir de ces observations on peut conclure qu'utiliser un seul modèle de prédiction n'est pas une solution faisable vu l'hétérogénéité des différents moteurs d'exécution supportés par notre solution :

(i) les modèles de coût sont représentés sous forme de vecteurs ayant des dimensions différentes (cf . Modèle de coût de la section optimisation)

(ii) le temps de réponse ne varie pas avec les variables de prédiction indépendamment des configurations du contexte expérimental.

Les autres facteurs qui dépendent de la configuration des systèmes et de l'environnement de déploiement de l'expérience ont une forte influence sur les valeurs de la variable à prédire.

Finalement on peut mentionner le fait qu'il faut adapter le modèle de prédiction à chaque ajout d'un nouveau système dans notre solution ce qui peut mettre à risque le fonctionnement de l'algorithme.

Le type d'algorithme d'apprentissage qu'on a déterminé pour ce problème est un algorithme de régression. Ce choix est évident car la variable à prédire (le temps de réponse d'une requête) est une variable continue.

Pour utiliser cette approche, la première étape consiste à trouver l'architecture appropriée. Ensuite, il est nécessaire de déterminer les métadonnées qui influencent directement la valeur de temps de réponse (c'est-à-dire sélectionner les caractéristiques qui vont déterminer la valeur à prédire), et finalement il est important de déterminer l'algorithme de prédiction à utiliser.

#### *4.4.5.2 Proposition : Des nœuds de prédiction pour estimer la performance du placement*

Dans cette sous-section nous présentons l'architecture de haut niveau du module de prédiction de temps de réponse dans DWS.

En se basant sur la discussion précédente, nous avons déterminé trois nœuds de prédictions pour supporter l'hétérogénéité des systèmes supportés dans notre solution.

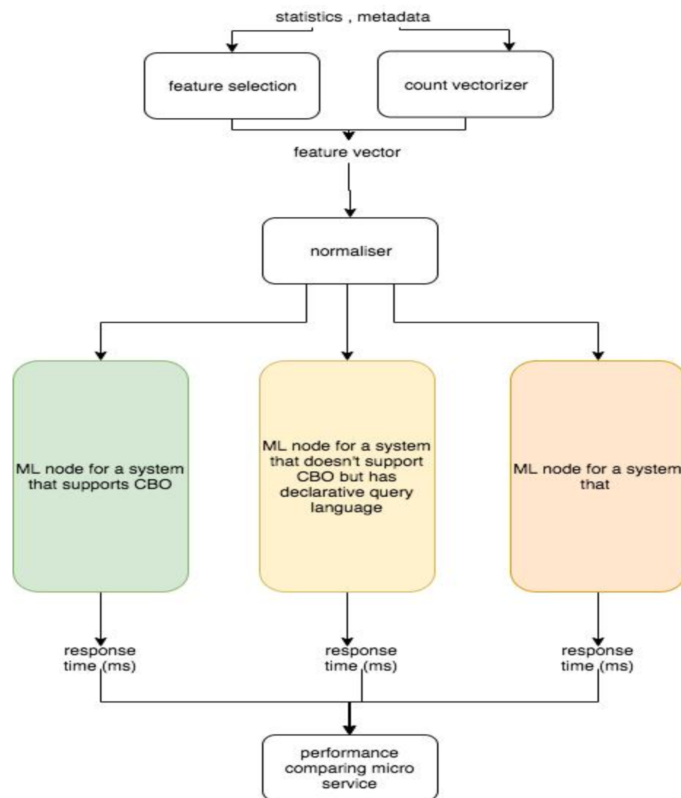



*Figure 4.10* Architecture du module de comparaison de performance dans DWS

Nous proposons donc des nœuds pour trois types de systèmes (comme le montre la figure 4.10) :

- Nœuds d'apprentissage automatique pour les systèmes qui sont capables d'estimer le coût des requêtes comme les RDBMS.
- Nœuds d'apprentissage automatique pour les systèmes qui n'utilisent pas les modèles de coûts pour l'optimisation du traitement, mais qui sont supportés par un médiateur qui satisfait la première condition comme le système Apache Calcite
- Nœuds d'apprentissage automatique pour les systèmes qui utilisent les modèles de coûts mais leur modèle de coût n'est pas intéressant, très simple ou erroné.

Avant d'identifier les algorithmes d'apprentissage pour ces nœuds, nous proposons d'expliquer la méthode utilisée pour sélectionner les caractéristiques du modèle de prédiction et les transformations qu'il faut exécuter pour préparer les données à l'étape d'apprentissage.

#### *4.4.5.3 Transformation des requêtes d'entrée en sous-requêtes simples (atomiques)*

A l'entrée de l'architecture de la figure précédente, des métadonnées sur les ensembles des données, par exemple la taille des données et les histogrammes de répartition des valeurs des attributs, et des métadonnées sur les requêtes comme les opérateurs du plan logique et physique de la requête. Les deux premiers composants : celui qui est responsable de la sélection des caractéristiques (*feature selection*) et celui qui transforme les opérateurs d'une requête en un format compréhensible par l'algorithme de prédiction (*count vectorizer*), reçoivent les métadonnées en entrée et génèrent un vecteur numérique.

Le défi principal de notre approche est la collecte de mesures de performance sur une granularité fine du workload. Toutefois le temps de réponse de chaque opérateur est une information qui n'est pas

toujours disponible. Les outils utilisés pour acquérir cette information sont les API *explain* et *profile* des optimiseurs des bases de données / des moteurs de traitement Big Data.

Dans ce cas, le contexte expérimental impose la collecte de mesures par requête et on doit trouver une méthode de transformation de fine granularité pour garantir la précision dans la collecte et la prédiction du temps d'exécution. Pour essayer d'adapter notre algorithme à des requêtes diverses complexes et simples, à partir d'une base de requêtes on extrait leur plan logique et ensuite on le décompose en sous plans simples composés d'un opérateur de projection, de lecture des données et d'un autre opérateur de transformation comme le filtrage, le tri, etc.

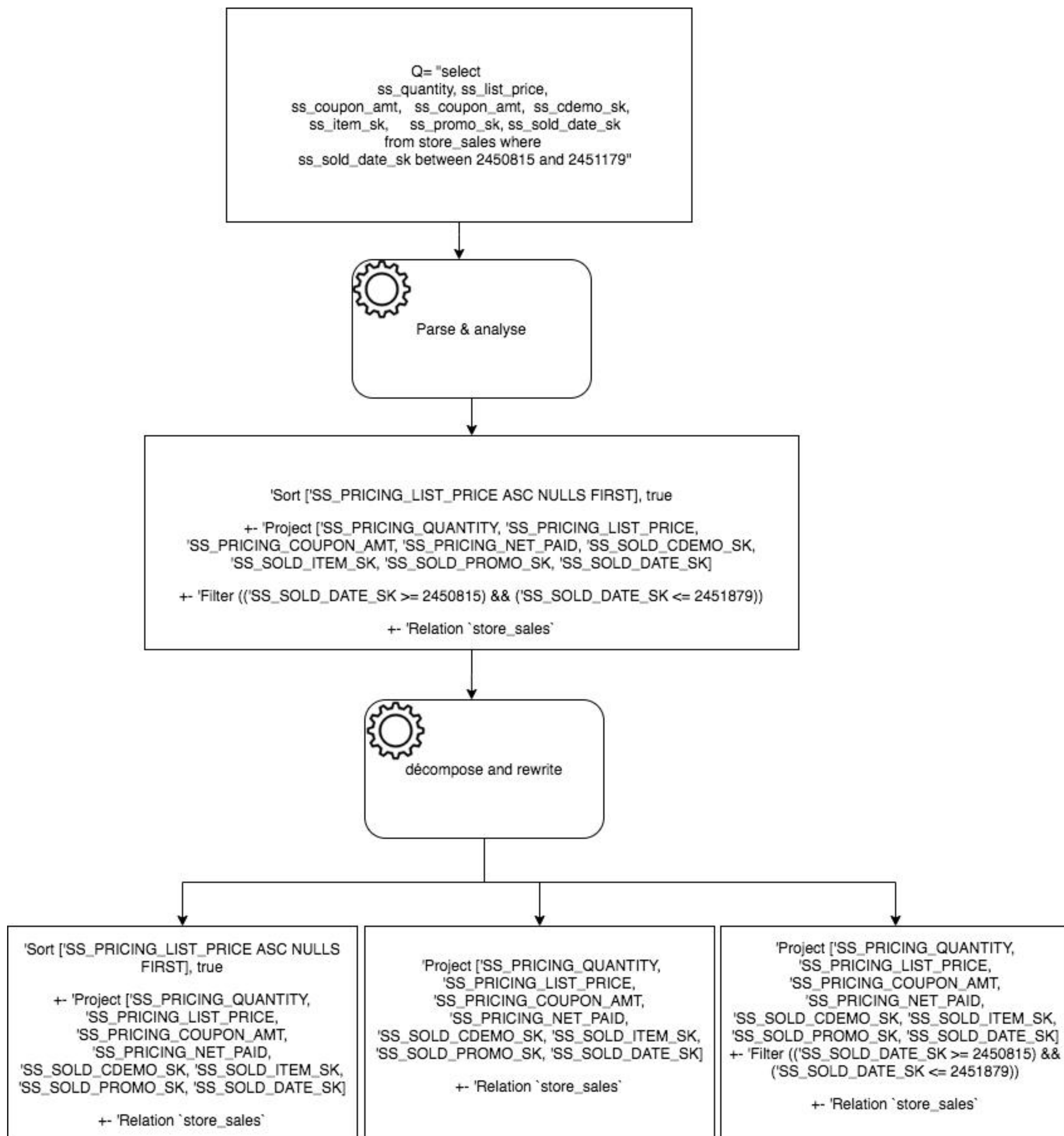



*Figure 4.11 Exemple de transformation par réécriture de requête*

La figure 4.11 présente un exemple des transformations que l'on propose. Le premier composant « *parse & analyse* » permet de transformer une requête SQL complexe en un plan logique analysé et optimisé. Le deuxième « *décompose and rewrite* » génère à partir de la requête d'entrée des sous requêtes atomiques qui contiennent un seul opérateur.

#### 4.4.5.4 Méthode utilisée pour sélectionner les variables de prédiction

La méthode adaptée dans ce travail est de s'inspirer du fonctionnement des moteurs d'exécution des systèmes Big Data pour déterminer les variables qu'on va utiliser pour résoudre le problème de prédiction. Comme première étape, nous avons examiné les travaux existant dans le domaine. Ceci nous a permis de construire ce tableau de synthèse qui compare les différents papiers de recherche qui étudient les variables de prédiction (tableau 4.5).

| Papier | Système | Variables |
|---|---|---|
| ACR+12 | postgresQL | **Modélisation au niveau du plan** : Coût total estimé du plan, Coût de démarrage estimé du plan, Nombre estimé de tuples de sortie, Largeur moyenne estimée d'un tuple de sortie (en octets), Nombre d'opérateurs de requête dans le plan, Nombre total estimé de tuples d'entrée et de sortie vers / depuis chaque opérateur, Taille totale estimée (en octets) de tous les tuples d'entrée et de sortie, Le nombre d'opérateurs <operator_name> dans la requête, Le nombre total de tuples sortis des opérateurs <operator_name><br>**Modélisation au niveau de l'opérateur** : E / S estimée (en nombre de pages), nombre estimé de tuples de sortie, nombre estimé de tuples d'entrée (à partir de l'opérateur enfant gauche), Nombre estimé de tuples d'entrée (à partir de l'opérateur gauche droite), Sélectivité estimée de l'opérateur, Heure de début de l'opérateur enfant gauche, Durée d'exécution de l'opérateur enfant gauche, Heure de début de l'opérateur enfant droit, Durée d'exécution de l'opérateur enfant droit. |
| WCHN13 | postgresQL | - Nombre d'E/S séquentiels<br>- Nombre d'E/S aléatoires<br>- Nombre de balayages |

*Tableau 4.5* Résumé des variables utilisées dans le problème de prédiction de temps de réponse de quelques travaux existants

En étudiant les travaux existants, on peut conclure que la modélisation au niveau de l'opérateur est plus appropriée pour les workloads dynamiques contrairement à la modélisation au niveau du plan qui convient mieux aux applications usuelles qui sont composés de requêtes similaires (comme dans le cadre de notre projet). Notre approche consiste à apprendre un modèle prédictif à partir d'un workload exécuté off-line qui est similaire aux workloads qui se trouvent dans le lac de données. La modélisation au niveau du plan de la requête correspond parfaitement à notre contexte et il n'est pas nécessaire de choisir une approche complexe comme celle de la modélisation au niveau des opérateurs.

Les variables des travaux existants représentés dans le tableau ci-dessous, sont enrichies dans notre travail par d'autres variables que nous jugeons nécessaires, comme la sélectivité accumulée des opérateurs de filtrage et de jointure, la taille des attributs agrégés, etc. Nous proposons aussi d'utiliser des variables supplémentaires spécifiques pour les solutions Big Data comme les magasins de données NoSQL.

La deuxième étape consiste à déterminer d'autres variables en s'inspirant du fonctionnement des systèmes. Intuitivement on sélectionne des statistiques comme la taille des données. Cette variable influence le temps d'exécution pour la majorité des opérateurs des langages de requêtes. Les variables déterminées d'après cette observation sont la taille de l'ensemble des données calculé en utilisant

cette formule : « taille d'un enregistrement * nombre d'enregistrements » et sa taille sur disque récupérée à partir du système de stockage. Par la suite, nous avons étudié les moteurs de stockage et d'exécution des différents systèmes. Par exemple on a remarqué dans le magasin de données NoSQL dont le modèle est orienté graphe que le stockage des nœuds du graphe et celui des relations se fait dans 2 fichiers différents (cf. chapitre 2). Si la requête est faite en utilisant l'opérateur « ScanByNode , le moteur d'exécution permet de balayer le fichier des nœuds et le lire en mémoire afin d'exécuter l'opérateur suivant. De la même manière, l'opérateur « ScanByRel » qui est utilisé dans les requêtes de traverse du graphe lit les données en commençant par le fichier des relations ensuite celui des nœuds accédés dans la requête. A partir de ces observations on détermine les variables « nodes_by_label » et « links_by_label » qui représentent le nombre de nœuds / relations stockés dans leurs fichiers associés et qui seront accédé par l'opérateur de balayage de données. La justification des choix des variables spécifiques aux magasins de données NoSQL sont expliqués dans le chapitre de présentation des résultats et du prototype. La deuxième étape de la sélection des variables consiste à calculer statistiquement la dépendance du temps de réponse (l'objectif de la prédiction) avec les variables identifiées dans cette étape. Les outils utilisés dans cette étape sont la visualisation et l'estimation du coefficient de corrélation des variables étudiées avec l'objectif de la prédiction.

Les données utilisées dans l'apprentissage ont été générés à partir des mesures sur des requêtes lancés dans un environnement contrôlé. Le serveur qu'on a utilisé est isolé et il n'est pas accédé par d'autres utilisateurs. L'avantage de cette configuration est de limiter les latences produites par les accès concurrents qui peuvent exister dans des environnements partagés.

#### *4.4.5.5 Estimation de coût pour un workload*

Pour estimer le critère de performance dans DWS, nous proposons d'étudier les coûts suivant : (i) le coût d'exécution de la requête sur le candidat de placement cible, (ii) le coût de déplacement de l'ensemble de données d'un magasin de données à un autre, (iii) le coût de transformation des données et (iv) coût du transfert des données (coût du réseau et du parallélisme)

Dans notre approche on fait une distinction logique entre les transformations. Mais physiquement les fonctions de transformation sont généralement équivalentes à des requêtes. Par exemple les requêtes d'indexation ou de partitionnement qu'on peut envoyer aux couches inférieures, des fonctions de transformation customisée qu'on peut développer en utilisant l'API de transformation de Spark ou bien en utilisant des DSL (*domain specific language*), ou bien des transformations de requêtes par réécriture. On peut donc conclure que le coût de transformation est facilement calculable à partir du plan de la fonction de transformation et en utilisant le même principe.

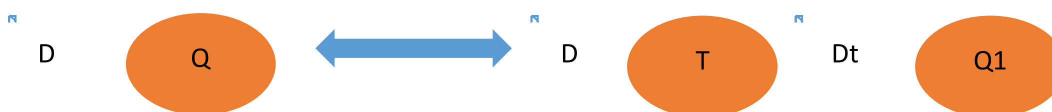



*Figure 4.11* Modélisation des transformations sous formes de requêtes

La sélection du plan optimal doit prendre en compte le coût de transformation. Mais elle doit aussi prendre en compte le fait que la transformation va être utilisée dans des futurs workloads. Par conséquent, le coût du workload doit prendre en compte des réutilisations des transformations et coûts associés.

- Coût de transformation de modèle de données et de communication des données

Dans notre modélisation de la performance on souhaite prendre en compte en plus du temps d'exécution le temps nécessaire pour charger les données depuis le disque (si le traitement l'exige), le

temps de communication des données d'un serveur à un autre, le temps d'exécution de la requête de transformation des données et finalement le temps nécessaire pour charger les données dans le serveur cible.

Le calcul du coût de communication réseau est dans le cadre de déplacement de données d'une source à une autre est simplifié dans le cadre de cette étude. Il suffit de multiplier la taille des données à transférer par le débit de transmission. Dans le cas d'une exécution distribuée et en parallèle du workload le coût de communication devient égal au coût de communication du fragment le plus volumineux de données. Donc dans notre modèle simplifié de coût de communication on considère le coût = taille du fragment le plus volumineux de données à transférer dans le réseau.

Le temps de chargement des données en mémoire varie d'un système à un autre. Il est possible de l'observer dans les outils de reporting de la performance du traitement (comme dans le cas de Spark qui permet cette fonctionnalité dans l'interface web UI).

##### 4.4.5.6 Conclusion

Dans cette section nous avons présenté la méthode ainsi que les arguments qui la supportent. Nous avons présenté aussi l'architecture du module de prédiction du temps de réponse qui est composée de différents nœuds propres à chaque système Big Data intégré dans ce module de recommandation et à chaque serveur où a eu lieu le déploiement de ce système. Dans le chapitre suivant nous présenterons les résultats de cette méthode qui concernent des modèles de prédiction développés à partir des expériences effectuées dans ce travail de recherche. Dans la section suivante nous présentons le module de simulation de l'exécution du workload, qui est nécessaire à un type bien déterminé des moteurs d'exécution de requête des systèmes Big Data étudiés.

## 4.5 Simulation d'exécution de requêtes

Notre approche de recommandation est basée sur la simulation pour pouvoir estimer les coûts d'exécution des différentes solutions de placement. On envoie au système un extrait des données ou bien des métadonnées pour déterminer la performance. Cette approche a été utilisée dans des systèmes Polystores existants et elle s'appelle la méthode : « what-if » [GPTK16, LSH+14]. Dans notre approche, on envoie les statistiques aux systèmes et on récupère le coût estimé de la requête, ou bien d'autres estimations calculées par ce système (par exemple la sélectivité, la taille des données, …). Parfois, le système n'est pas capable de retourner une estimation de coût ou bien des statistiques sur l'exécution de la requête ou alors il est impossible d'envoyer des statistiques à ce système. Dans ce cas de figure, on envoie au système des métadonnées sur la structure des données telles qu'elles seraient stockées dans la solution de placement et on récupère du système seulement le plan physique d'exécution de la requête. A partir des informations que le système est capable de retourner, on construit un vecteur de métadonnées sur les cas de placement étudiés et en y associant des métadonnées récupérées du catalogue de DWS, on estime la performance de l'exécution de requête. On détaille par la suite les deux cas de simulation possible : simulation de placement par injection de statistiques et simulation de placement par matérialisation d'un extrait des données. Dans le chapitre prototype et évaluation, nous allons discuter de l'implémentation technique de ces fonctionnalités.

### 4.5.1 Injection de statistique

Une idée intéressante consiste à envoyer à l'optimiseur de requête les statistiques sur les ensembles de données pour lui faire croire que les données sont bien présentes et à récupérer directement des systèmes une estimation du coût d'exécution de la requête si l'ensemble de données a été stocké dans ce système. Cette méthode présume que le système est capable d'estimer et de retourner le coût d'exécution des requêtes et de gérer des statistiques sur les données. Ce qui est vrai pour les SGBD classiques mais qui présente une limite pour certains systèmes Big Data. De plus, pour les systèmes

qui utilisent l'optimisation basée sur les coûts, la comparaison directe de ces coûts n'est pas évidente. Les coûts des requêtes dans ces systèmes ont été calculés dans l'objectif de comparaison des plans d'exécution des requêtes dans un seul système et la comparaison des coûts inter systèmes est un défi conditionné par l'hétérogénéité de ces modèles abstraits : les modèles de coûts ont des vecteurs divers (sur le plan sémantique et sur le plan dimensionnel) et ils sont exprimés en utilisant des unités différentes. Vu la diversité des modèles de coûts, on ne peut pas comparer les coûts par transformation mathématiques. Cette approche est difficile à mettre en place et son utilité à long terme est limitée vu la rapidité de l'évolution des modèles de coûts.

Le seul intérêt d'injecter des statistiques est l'obtention de l'estimation du coût de la requête calculée par l'optimiseur. Pour mettre en place une solution basée sur l'injection des statistiques il est nécessaire que le système gère des métadonnées sur les ensembles de données surtout un catalogue et des statistiques et que ce système utilise ces métadonnées dans l'estimation du coût d'exécution de la requête. Cette solution a été proposée par le système MuSQLe qui nous a inspiré dans la conception de DWS, mais elle n'a été proposée que pour PostgreSQL et elle n'a pas été vérifié expérimentalement. Dans DWS nous avons implémenté un module d'injection de statistiques pour PostgreSQL, pour le plugin d'Apache Calcite pour MongoDB et pour Apache Phoenix. Le choix de ces trois systèmes pour la mise en place de cette méthode a été justifié par la possibilité d'accéder et de modifier les métadonnées du catalogue et les statistiques, et par la pertinence de ces métadonnées dans l'estimation des coûts des requêtes. Le tableau 4.6 résume la différence de quelques systèmes Big Data pour cette fonctionnalité et souligne notre choix.

| **Système** | **Statistiques** | **Collection de statistiques** | **Catalog** | **Estimation de coût** | **Optimisation de requête** |
|---|---|---|---|---|---|
| Apache Phoenix | - Taille d'un chunk (GUIDE_POSTS_ WIDTH)<br>- Nombre d'enregistrements | System.stat s | System. catalog | Oui | Modèle de coûts simple |
| PostgreSQL | - Statistiques sur les colonnes<br>- Statistiques sur les relations<br>- Statistiques sur les pages | Oui | Oui | Oui | Modèle de coûts mature |
| Apache calcite | -Nombre d'enregistrements<br>- Clés<br>- Contraintes référentiels | Non | Non | Oui | Modèles de coûts spécifiques à chaque système et en cours de développement |

*Tableau 4.6* Résumé des statistiques utilisés dans l'optimiseur de requête de quelques moteurs d'exécution

L'utilisation du coût des requêtes est important dans le cas où l'optimiseur de requête a une estimation précise. La valeur du coût dans ces cas permet de retrouver des métriques complémentaires aux statistiques présentes dans le catalogue de métadonnées de DWS.

On propose d'utiliser une approche par apprentissage automatique pour l'estimation du temps de réponse qui selon notre méthode reflète la performance dans chaque système. Toutefois l'estimation des coûts retournée par les systèmes et qui est utilisée pour le choix le meilleur plan d'exécution de la requête est une métadonnée importante pour la prédiction du temps de réponse. Elle est intéressante pour accélérer l'apprentissage et rendre la prédiction plus précise [GPTK16].

#### 4.5.1.1 Injection de statistiques dans PostgreSQL

Cette méthode a été inspirée du système d'intégration MuSQLe. Le système simule l'exécution d'une requête en envoyant les statistiques que le moteur de requête utilise dans son module d'optimisation pour estimer le coût de l'exécution. Le plugin est intitulé : *pg_dbms_stats*.

MuSQLe injecte les statistiques uniquement pour les colonnes *reltuples* et *relpages* du catalog_pg_class. Nous explorons cette approche pour simuler l'exécution d'une requête dans PostgreSQL et ainsi récupérer le coût estimé par l'optimiseur.

#### 4.5.1.2 Injection de statistiques dans Apache calcite

Dans le 2ème type de nœuds de recommandation on propose d'utiliser Apache Calcite – le système d'interrogation dynamique des données Big Data – pour obtenir le coût d'exécution des requêtes dans les systèmes Big Data qui ne supportes pas l'optimisation de requêtes basée sur les coûts ou bien l'interrogation des données en utilisant des requêtes déclaratives. MongoDB utilise une API spécifique pour l'accès aux données. Il ne gère pas de statistiques pour l'optimisation de requête et il utilise des heuristiques pour sélectionner le plan optimal pour la fonction d'accès aux données lancée par l'utilisateur. Or il existe un plugin Calcite-MongoDB qui nous permet d'étudier le coût de l'exécution des requêtes lancés sur MongoDB et récupérer des plans de requêtes composés d'opérateurs qui correspondent à des fonctions de l'API de ce système.

Le plugin Calcite-MongoDB utilisé dans cette expérience a la version 1.19.0 Il permet d'interroger les données dans MongoDB en utilisant le langage SQL. Il permet aussi d'optimiser les requêtes en utilisant la méthode basée sur les coûts et estimer pour chaque opérateur de requête : le coût CPU, le coût E/S et le nombre de résultats retournés (cardinalité de l'opérateur).

Le coût d'entrée sortie dans MongoDB n'est pas calculé par Apache calcite. Pour optimiser les coûts des requêtes dans MongoDB, Apache Calcite ne compare pas la performance de l'exécution dans la source de données (MongoDB). Il estime le coût d'exécution des opérateurs relationnels d'Apache Calcite en éliminant les coûts d'accès à la couche de stockage pour simplifier le processus d'optimisation.

Calcite utilise des métadonnées pour estimer les coûts des requêtes. Mais il n'offre pas la fonctionnalité de stockage de métadonnées. Il propose néanmoins des interfaces qui interagissent avec une source externe de métadonnées.

La statistique injectée dans notre preuve de concept est le nombre d'enregistrements vu que cette propriété a une grande influence dans le calcul du coût des opérateurs d'Apache calcite (cf état de l'art). Mais il est possible d'injecter d'autre métadonnées, comme les colonnes qui appartiennent à la clé primaire, les contraintes référentielles, …

Dans le cas de l'utilisation d'Apache calcite avec le data store MongoDB, une instance de la classe « Statistic » est créée dans la classe « MongoTable » et elle contient des statistiques sur les ensembles de données obtenues du catalogue de statistiques globales.

Dans l'objectif d'avoir des estimations de coût de requête plus pertinents, nous implémentons une fonctionnalité de sélectivité qui sera utilisé dans plusieurs opérateurs comme *Sort*, *Join*, *Filter,* … Dans la partie suivante, nous présenterons nos alternatives pour le calcul de la sélectivité.

### 4.5.2 Récupérer des statistiques sur les ensembles et les requêtes

Dans cette sous-section, nous présentons les méthodes utilisées pour récupérer des statistiques sur les workloads et les ensembles de données.

#### *4.5.2.1 Estimation de la sélectivité*

L'idée initiale était d'utiliser Apache Calcite pour l'estimation de la sélectivité des requêtes car Apache Calcite supporte une large variété de systèmes Big Data.

Ces limites nous ont dirigé vers (i) développer nos propres algorithmes d'estimation de sélectivité, ou bien (ii) l'utilisation de la sélectivité de la requête du système PostgreSQL puisque l'injection de statistique dans ce système est complète et vu que la sélectivité ne dépend pas du système, les valeurs de sélectivité sont toujours les mêmes dans tous systèmes de traitement de requêtes. La solution (ii) est plus élégante et en harmonie avec notre approche basée sur la simulation.

#### *4.5.2.2 Estimer les cardinalités et les statistiques des résultats intermédiaires*

*Explain* est une fonctionnalité fournie par la majorité des systèmes d'interrogations de données. Son objectif est de retourner à l'utilisateur les détails des plans d'exécution des requêtes sous forme d'arbres d'opérateurs pondérés à par une estimation de la performance de l'exécution de la requête. Avec *explain* on est capable de récupérer des statistiques sur les requêtes et sur les données comme les cardinalités. On peut aussi récupérer la taille des résultats par ce moyen.

## 4.6 Algorithmes

Dans cette section nous présentons l'algorithme de recherche de solution optimale de placement de données et les algorithmes annexes qui mènent à construire l'espace de recherche. L'algorithme de recherche en profondeur d'abord (*Depth first*) et son alternative rechercher en largeur (*Width first*) sont des exemples de recherche exhaustive (celle que nous avons sélectionnée dans cette étude car elle garantit l'optimalité et que notre algorithme de recommandation n'a pas des contraintes de rapidité d'exécution particulières).

### 4.6.1 Algorithme de construction de l'espace de recherche

L'algorithme de construction de l'espace de recherche de DWS est composé de 3 étapes principales. La première étape consiste, pour chaque requête, à énumérer les différents plans logiques équivalents de cette requête en utilisant des heuristiques et en variant entre l'exécution hybride et l'exécution dans les data stores directement. La deuxième étape a comme objectifs d'examiner les différentes solutions (i) de stockage dans les data stores et (ii) d'exécution dans les moteurs d'exécution. La troisième étape consiste à parcourir l'espace de recherche, détecter et appliquer les transformations par réécriture de requête qui sont nécessaires pour la recommandation sur le placement. Finalement, l'algorithme simule l'exécution de chaque transformation du workload selon la configuration que le constructeur de l'espace de recherche lui a associée et estime le coût d'exécution de ces transformations jusqu'à trouver la solution optimale.

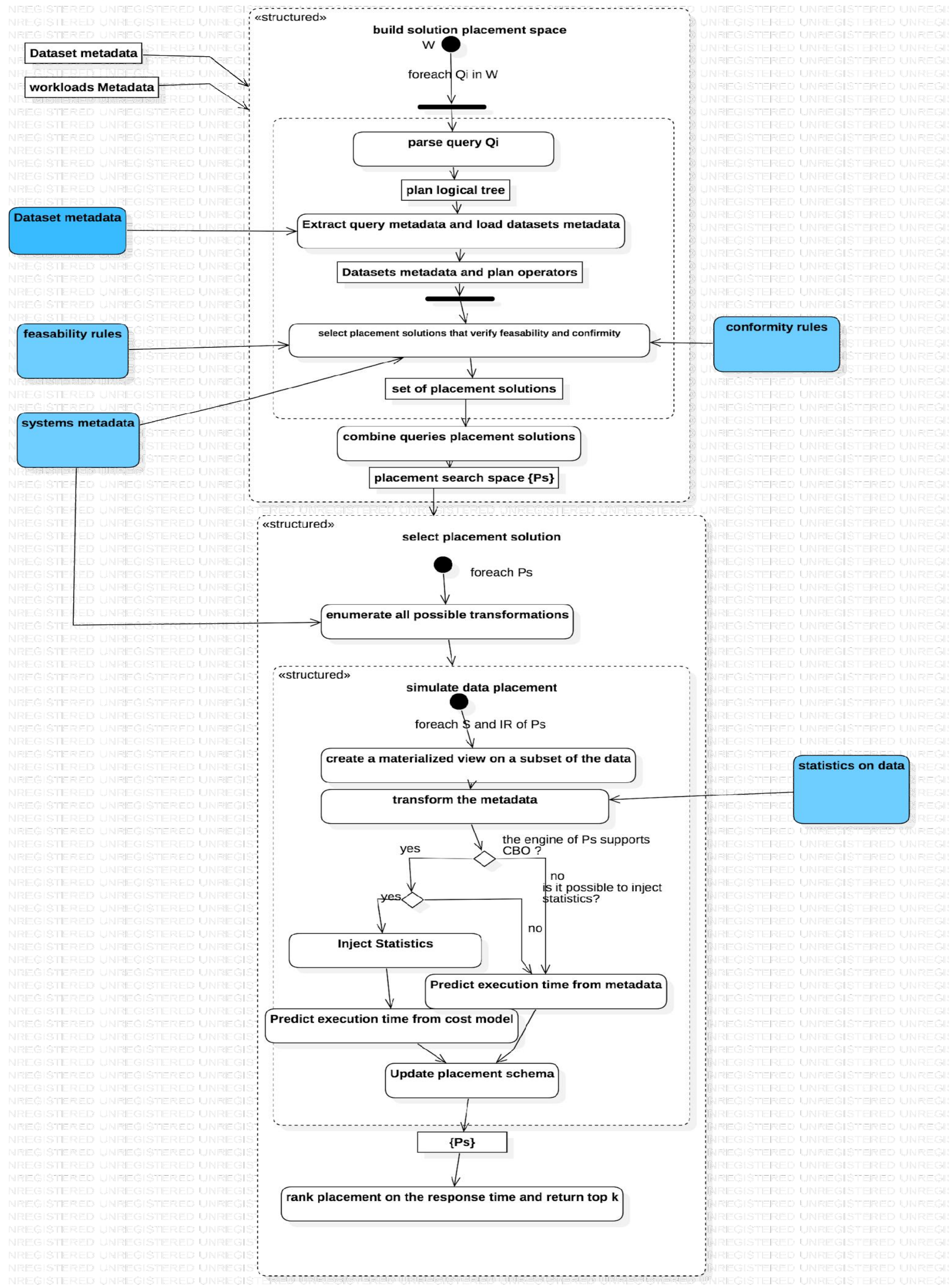



*Figure 4.12* Algorithme de recherche de la solution de placement optimale

La figure 4.12 représente l’algorithme complet de recherche de la solution optimale de placement de données. Dans ce qui suit, nous détaillons davantage les composants de cet algorithme.

- *Enumération des plans logiques pour une requête hybride*

La première étape de l’algorithme construit un espace de recherche initial avec les requêtes décomposées à partir du workloads qui sont associées à différentes possibilités de placement de données.

La deuxième étape de cet algorithme consiste à développer à partir du plan logique de départ, qui correspond à chaque requête (Q) du workload (W), les plans équivalents qui produisent le même résultat mais qui sont caractérisés par des efficacités différentes. Il utilise (i) des heuristiques pour construire des plans équivalents en modifiant l'ordre des opérateurs binaires et en envoyant les opérateurs de jointure, agrégation et ordre à la couche inférieure et (ii) des métadonnées niveau système qui décrivent les transformations physiques possibles.

Les heuristiques utilisées dans cet algorithme sont des règles qui permettent de déléguer l'exécution plus vers les sources de données que dans les moteurs d'exécution hybrides. Parmi ces heuristiques on utilise : (i) « *predicate push down* » une heuristique qui permet d'envoyer les attributs aux sources et ainsi permettre au moteur d'exécution hybride de charger en mémoire moins de données. (ii) « *filter push down* » une 2ème heuristique qui envoie l'opérateur de filtrage vers la source de données. (iii) Et finalement « *join order* » qui permet d'inverser comme le montre la figure ci-dessous l'ordre des jointures.

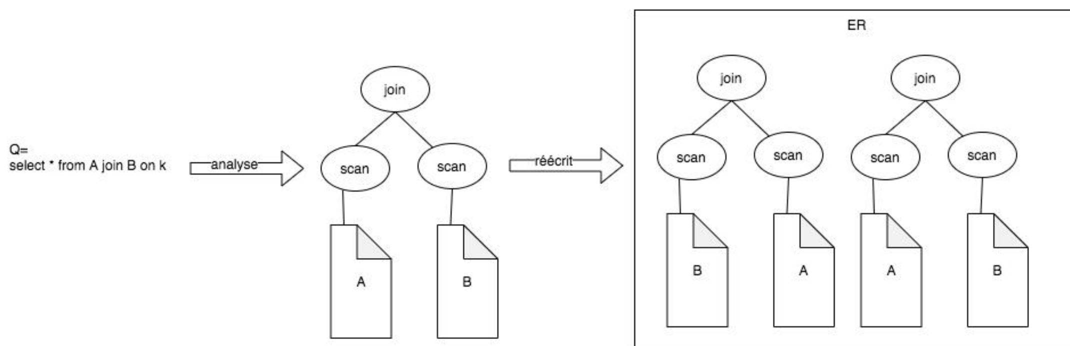



*Figure 4.13 Exemple de transformation d'un plan d'exécution en inversant l'ordre de la jointure*

La deuxième transformation associe à chaque ensemble de données une transformation physique supportée par le magasin de données dans lequel on a prévu de stocker ces données.

#### 4.6.1.1 Espace de recherche obtenu pour un workload

A la fin de l'étape de l'énumération de toutes les solutions possibles de placement de données, on obtient un espace de recherche étendu qui a une structure qui ressemble à la structure suivante :

> (table source avec une transformation physique) -> requête -> communication -> transformation résultat intermédiaire (eg insertion) -> (next tables)

Un exemple de cet espace de recherche pour un workload composé d'une seule requête est représenté dans la figure 4.14.

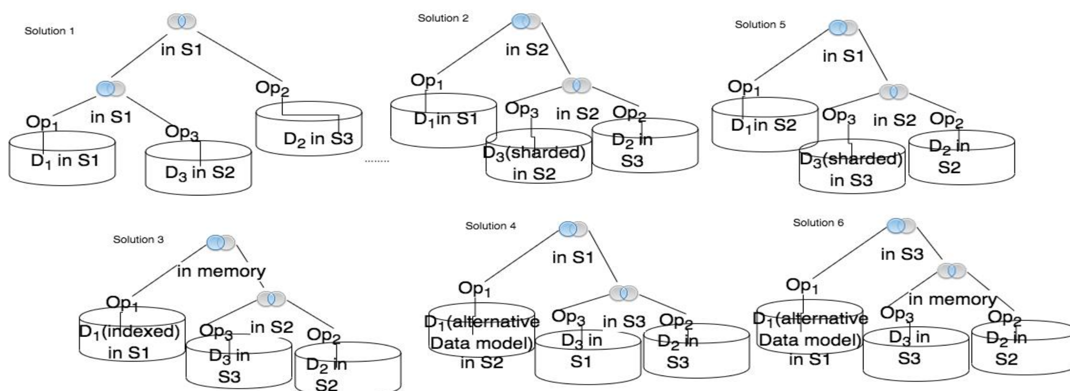



*Figure 4.14 Exemple d'espace de recherche généré par un optimiseur de placement de données*

### 4.6.2 Algorithme de recherche de placement optimale

Comme nous l'avons expliqué précédemment, l'algorithme de recherche de la solution optimale est un algorithme exhaustif. Cet algorithme génère les solutions de l'espace de recherche ensuite il teste si la condition de sélection de l'optimale est vérifiée. Il est possible d'utiliser des heuristiques pour limiter l'exploration.

Un exemple d'heuristique dans notre contexte serait une heuristique qui limite la communication réseau. Cette heuristique peut être significative vu que la distribution des données massives est la cause principale des latences dans l'exécution d'une requête Big Data.

Dans ce projet nous ne considérerons pas d'heuristiques supplémentaires aux règles de conformité et faisabilité expliquées au début de ce chapitre mais ils peuvent être considérées comme des perspectives à ce projet.

L'exploration de l'espace de recherche dans cet algorithme est en profondeur d'abord. En effet chaque élément du workload subit une transformation en commençant par l'élément le plus à gauche et en lui affectant toutes les transformations possibles.

## 4.7 Récapitulation et architectures

Dans cette section on propose une synthèse de ce qu'on a présenté précédemment sous forme une architecture plus détaillée.

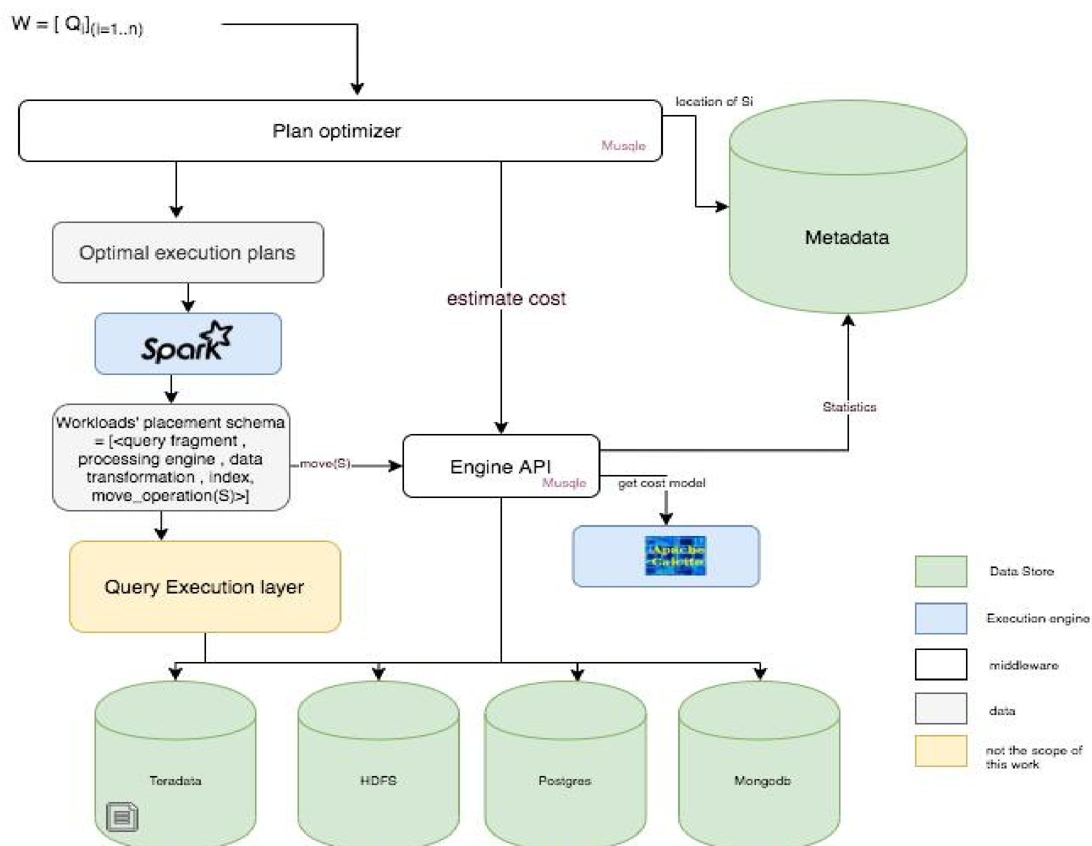



*Figure 4.15* Architecture détaillée de DWS

L'architecture décrite dans la figure 4.15 est composé principalement par : (i) un optimiseur de plan du workload qui a un rôle central et qui permet de décomposer le plan du workload, le transformer et chercher sa meilleure transformation, (ii) une couche d'intégration de nouveaux systèmes qui permet

d'étendre le système et supporter plusieurs systèmes, et finalement (iii) une couche d'exécution qui permet d'exécuter le plan optimal sélectionné par l'optimiseur.

En plus de l'algorithme de recherche de solution optimale et du simulateur d'exécution de requête, le composant d'optimisation du workload est composé d'un estimateur du temps de réponse des requêtes.

Ce composant utilise le principe de l'apprentissage continue et la figure ci-dessous illustre son architecture.

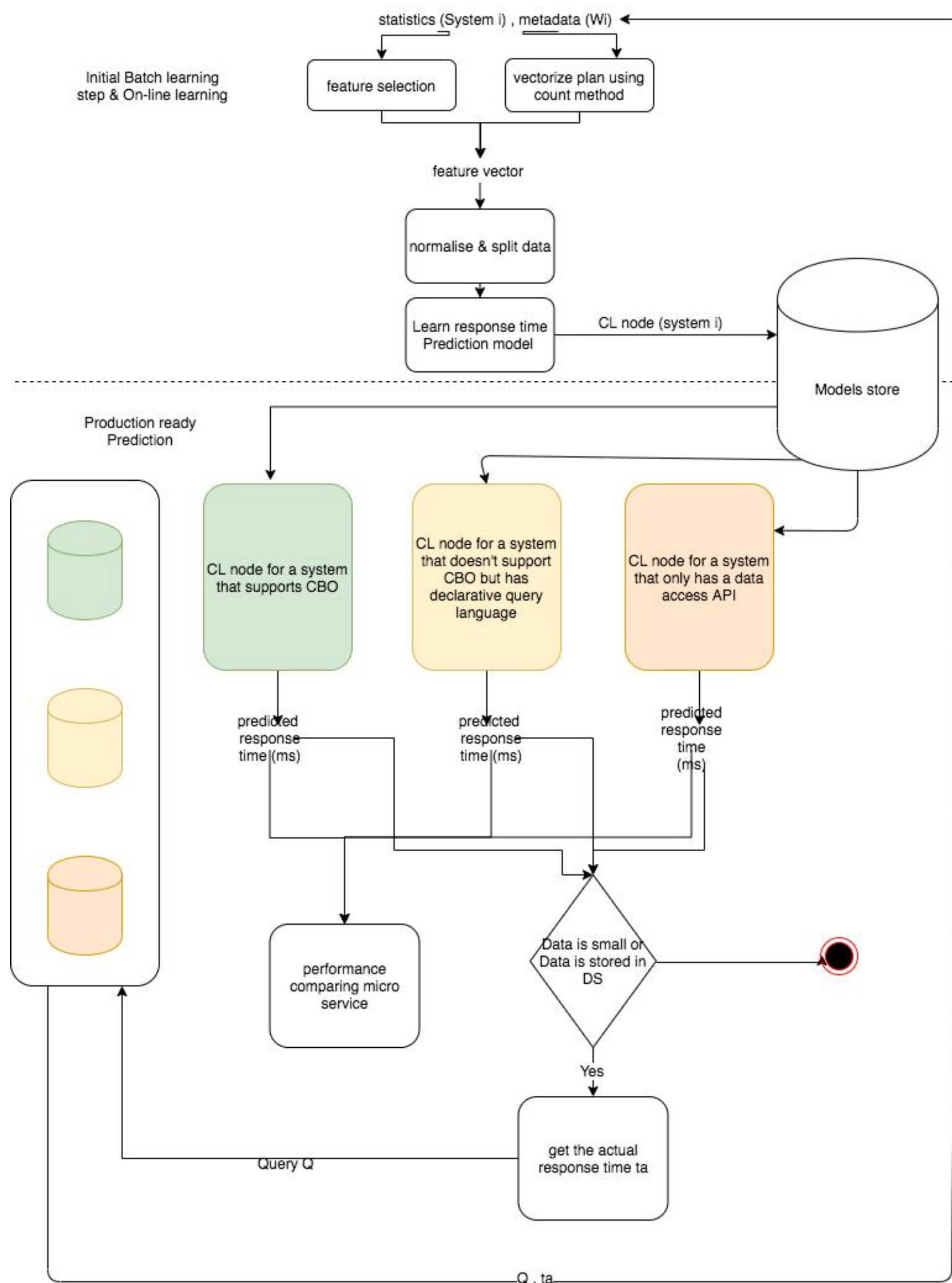



*Figure 4.16* Architecture du composant d'estimation du temps de réponse des requêtes du workload

La figure 4.16 illustre l'architecture du module le plus important dans ce système de recommandation.

### 4.7.1 Interface d'intégration

L'interface d'intégration est comme son nom l'indique une API qui permet à DWS d'intégrer de nouveaux systèmes Big Data d'une manière extensible et intuitive. Cette interface offre un point unique pour la transformation des données, l'interrogation de données hétérogènes et l'exécution des opérations de déplacement ou de copie des données. Elle permet d'injecter des métadonnées dans la couche de stockage physique pour simuler le placement.

## 4.8 Résumé du chapitre

Dans ce chapitre, nous avons présenté une architecture détaillée d'un système de recommandation de placement des données ainsi que la fonctionnalité de ses différents composons. Ce système de recommandation est basé principalement sur un module d'optimisation qui s'inspire des travaux existant en optimisation de requête. L'algorithme d'optimisation est exhaustif vu qu'il est mise en place dans le cadre d'un système de recommandation. Trouver la solution optimale est dans ce contexte plus important que la rapidité de l'exécution de l'algorithme de recommandation. La variable à optimiser dans notre application est le temps de réponse et on utilise des techniques d'apprentissage automatique pour l'estimer.

En termes de conclusion, nous proposons des opérateurs pour le placement de donnée et l'exécution optimisée du workload.

Il existe trois types d'opérateurs pour le placement des données dans un écosystème de Big Data complexe basé sur un lac de données : l'opérateur « *move* », l'opérateur « *copy* » et l'opérateur « *transform* ». Le premier opérateur sert à déplacer un ou plusieurs ensembles de données d'un système à un autre. Le deuxième opérateur crée une copie de l'ensemble de départ et la stocke dans le système cible. Il sera géré comme une nouvelle version et on aura création d'une nouvelle instance de métadonnées qui renseignent les versions des ensembles de données dans l'écosystème. L'opérateur de copie est surtout nécessaire dans le cadre de l'exigence par une règle de conformité que l'ensemble de données de départ ne soit pas déplacé. Le troisième opérateur est un opérateur de transformation, il sélectionne les données de leurs sources les agrège en exécutant une partie du traitement et stocke les données dans leurs systèmes cibles.

Le coût de mouvement entre les systèmes est estimé pour les trois étapes de lecture des données de la source, de l'écriture des nouveaux ensembles de données dans les systèmes cibles et la transmission des données d'un nœud à un autre en cas de besoin.

# Chapitre 5 Prototype et expérimentations

Dans ce chapitre, nous présentons ce qu'on a réalisé comme preuve de concept (un prototype) de notre Framework DWS décrit théoriquement dans les chapitres précédents. Nous présentons également les démarches expérimentales que nous avons utilisées pour sélectionner les variables d'apprentissage. Les expérimentations ainsi que la validation de la démarche scientifique ont été faits sur les données et le workloads du cas d'utilisation décrit dans le chapitre 4.

Le but de ces expérimentations étant de produire des modèles d'apprentissage précis qui s'intégreront dans le module de recommandation de placement de données. Ces derniers nous ont permis de valider notre étude théorique et fournir des résultats. Finalement, nous concluons par les résultats et leur analyse. La présentation du prototype dans ce chapitre suivra une méthode de conception UML. Ainsi nous allons présenter les diagrammes pour la conception du prototype réalisé au cours de cette thèse.

Le chapitre est organisé comme suit : (i) la section 5.1 détaille l'architecture du prototype développé à l'occasion de cette thèse, (ii) la section 5.2 résume la démarche expérimentale adapté et présente les résultats obtenus, (iii) quant à la section 5.3, elle détaille l'architecture logicielle du prototype et finalement (vii) la section 5.4 récapitule le chapitre.

## 5.1 Architecture du prototype DWS

Dans cette section, nous présentons les diagrammes du prototype réalisés au cours de ce projet. La première architecture proposée est une architecture en micro services qui décrit les interactions entre les composants de DWS.

*Figure 5.1* Diagramme d'architecture en micro services de DWS

La figure 5.1 représente les trois modules principaux de DWS organisés en micro services :

- *Metadata processing* : ce micro service prend en charge l'estimation des statistiques des requêtes et des ensembles des données qui sont des entrées importantes pour notre algorithme de recommandation de placement et de son modèle d'apprentissage du temps de réponse du workload. Il accède aux données du magasin de données des statistiques et des requêtes historiques (« statistics store, historical queries »).

- *Metadata injection and access* : ce micro service permet un accès modulaire aux métadonnées système en offrant des services REST aux micro services clients (par exemple le micro service de recommandation de placement) et encapsule la complexité des requêtes qui sont envoyées au magasin de métadonnées. Ce micro service prend en charge aussi l'injection des statistiques dans la couche de stockage des données afin de pouvoir simuler l'exécution du workload (cf. chapitre 4).

- *Recommendation on data placement* : le module le plus important de l'architecture est celui de la recommandation. Il utilise les deux autres micro services pour récupérer les métadonnées nécessaires pour son fonctionnement et accède à la couche de stockage en cas de besoin (par exemple pour la récupération des plans d'exécution des requêtes).

Dans ce suit qui nous détaillons chacun de ces micro services en commençant par le micro service de recommandation.

### 5.1.1 Micro service « Recommendation on data placement »

La recommandation dans DWS est basée sur la réécriture de requête et la simulation de l'exécution d'une requête dans plusieurs magasins de données candidats pour récupérer des métadonnées sur les requêtes nécessaires à la recommandation. Dans le chapitre 2 nous avons comparé plusieurs systèmes d'intégration de données. Il aurait été possible de concevoir et d'implanter un système d'intégration dédié pour notre solution. Mais vu qu'il existe plusieurs solutions qui satisfont notre besoin et qui offrent l'efficacité et les fonctionnalités visées par notre projet, nous avons décidé d'utiliser une solution existante et de l'enrichir pour atteindre notre objectif. Comme le montre la figure 5.2 nous utilisons dans ce prototype Apache Spark comme moteur d'exécution et API de transformation de requêtes pour plusieurs raisons. Il permet d'intégrer plusieurs sources de données de la distribution Apache, son API de transformation de données est riche, et finalement il supporte le langage SQL que nous avons choisi comme langage de requête pour nos workloads. Le langage de programmation du module de transformation du workload et d'une grande partie du module de l'optimisation est Scala. Ce langage a été choisi car il correspond au choix le plus performant pour Apache Spark. De plus, il offre une fonction de *Pattern matching* très utile puisqu'on travaille avec la réécriture de requête et manipulation de métadonnées.

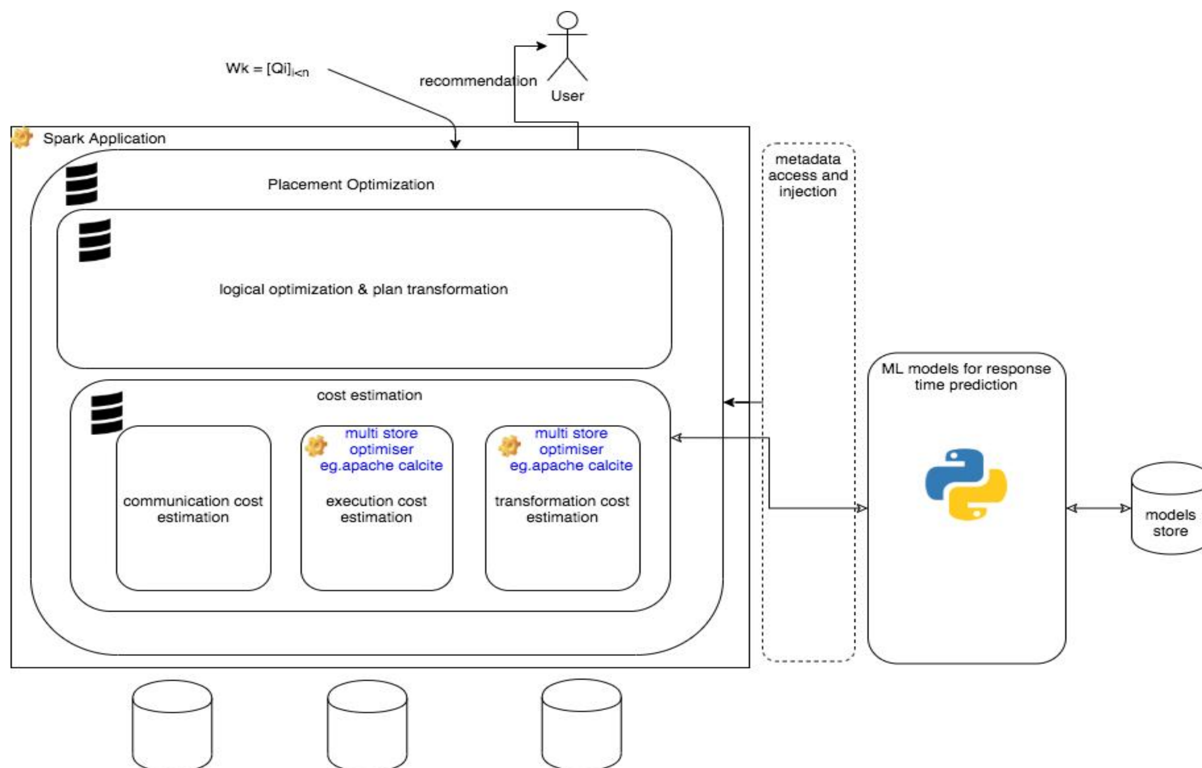



*Figure 5.2* Architecture technique du module de recommandation

Comme nous l'avons expliqué dans le chapitre précédent, dans la mesure du possible nous utilisons le coût d'exécution de requêtes estimés par l'optimiseur dans l'apprentissage du temps de réponse. Dans notre projet, nous avons choisi Apache Calcite vu qu'il supporte l'optimisation des requêtes basée sur les coûts pour plusieurs magasins de données.

La recommandation utilise des métadonnées de DWS sur les ensembles des données, les systèmes et les workloads. L'accès et la manipulation de ces métadonnées sont encapsulé dans un micro services : « *Metadata injection and access* ».

### 5.1.2 Micro service « Metadata injection and access »

Le micro service « Metadata injection and access » a comme objectif de permettre l'accès aux métadonnées pour les autres composants du système. Les métadonnées dans notre projet sont hétérogènes du point de vue structurel et sémantique. Il est important de sélectionner la solution de stockage la plus appropriée pour chaque type de métadonnées. Dans cette section, nous commençons par justifier le choix de la technologie de stockage pour ce micro service, par la suite nous présentons l'architecture technique complète du micro service.

#### *5.1.2.1 Implantation et représentation de métadonnées*

Les métadonnées de type statistique sont représentées sous forme d'agrégats ayant des attributs complexes et elles sont généralement accédées en utilisant des requêtes orientées sur les clés. Les accès par comparaison des attributs ne sont pas fréquents pour ces métadonnées et elles ne sont retrouvées que si on connait la référence de l'ensemble de données sur lequel on les cherche. Ces statistiques sont utilisées pour l'optimisation de placement par l'estimation de performance, pour l'optimisation de requêtes hybrides – si par exemple on utilise le système Apache Calcite pour cet objectif on peut intégrer des statistiques à partir d'une source externe – ou bien pour le profilage des ensembles de données. Tous ces cas d'utilisations ont un type d'accès par clé aux statiques. Pour cette

raison on favorise les systèmes de stockage dont le modèle est basé sur l'agrégat au lieu des systèmes de stockage de triplets.

Pour les métadonnées administratives et techniques nous proposons d'utiliser le format RDF et de les faire persister dans un système de stockage de type triplets. Ces métadonnées concernent les ensembles de données, les applications et leurs workloads, ainsi que les informations sur les systèmes en cours d'exécution. L'intérêt principal d'utiliser un modèle de données orienté graphe est de connecter les trois niveaux de description des métadonnées dans l'écosystème. Dans ce qui suit, nous détaillons les arguments qui justifient le choix de **RDF + RDFS** pour la gestion de cette catégorie de métadonnées.

- **standard**: c'est un standard reconnu par le W3C et qui a une syntaxe standardisée pour l'échange de données. Il propose un langage de requête standard orienté connexion (SPARQL) et qui permet de traverser facilement un graphe de métadonnées.

-**favorise la réutilisation** : il permet aussi de réutiliser d'autres schémas de métadonnées d'une manière flexible, ce qui est un atout fondamental. De plus il existe plusieurs travaux et des standards sur les métadonnées qui sont facilement intégrables dans notre schéma de métadonnées.

- **modèle de données orienté graphe labélisé** : ce modèle de données nous permet de représenter nos métadonnées DWS des différents niveaux, de faciliter et de garantir la performance des requêtes les métadonnées et les objets qui leurs sont associés.

**- extensibilité :** utiliser cette technologie facilite l'extensibilité. Le schéma des métadonnées est représenté en RDF. On n'a pas besoin de développer une application dédiée pour gérer les métadonnées. On peut se contenter d'utiliser des éditeurs de données RDF pour mettre à jour et étendre un schéma d'ontologie.

- **inférence** : l'inférence nous permet de découvrir des liens sémantiques implicites entre les métadonnées. Elle nous permet aussi de découvrir la structure des métadonnées, par exemple inférer les types des attributs du schéma de métadonnées. Finalement l'inférence nous permet de vérifier l'intégrité des métadonnées et de déterminer si la définition des objets est cohérente avec le schéma, les contraintes imposées et les restrictions.

Une alternative à RDF/RDFS pour représenter les métadonnées consiste à utiliser les graphes de propriétés. Les avantages de cette approche sont associés à son modèle de données qui (i) différencie les propriétés et les liens, (ii) permet plus facilement de représenter et d'interroger les agrégats et (iii) permet la représentation des liens avec des attributs.

#### 5.1.2.2 Architecture technique du micro service

Dans cette sous-section, nous présentons l'architecture technique du micro service manipulation des métadonnées.

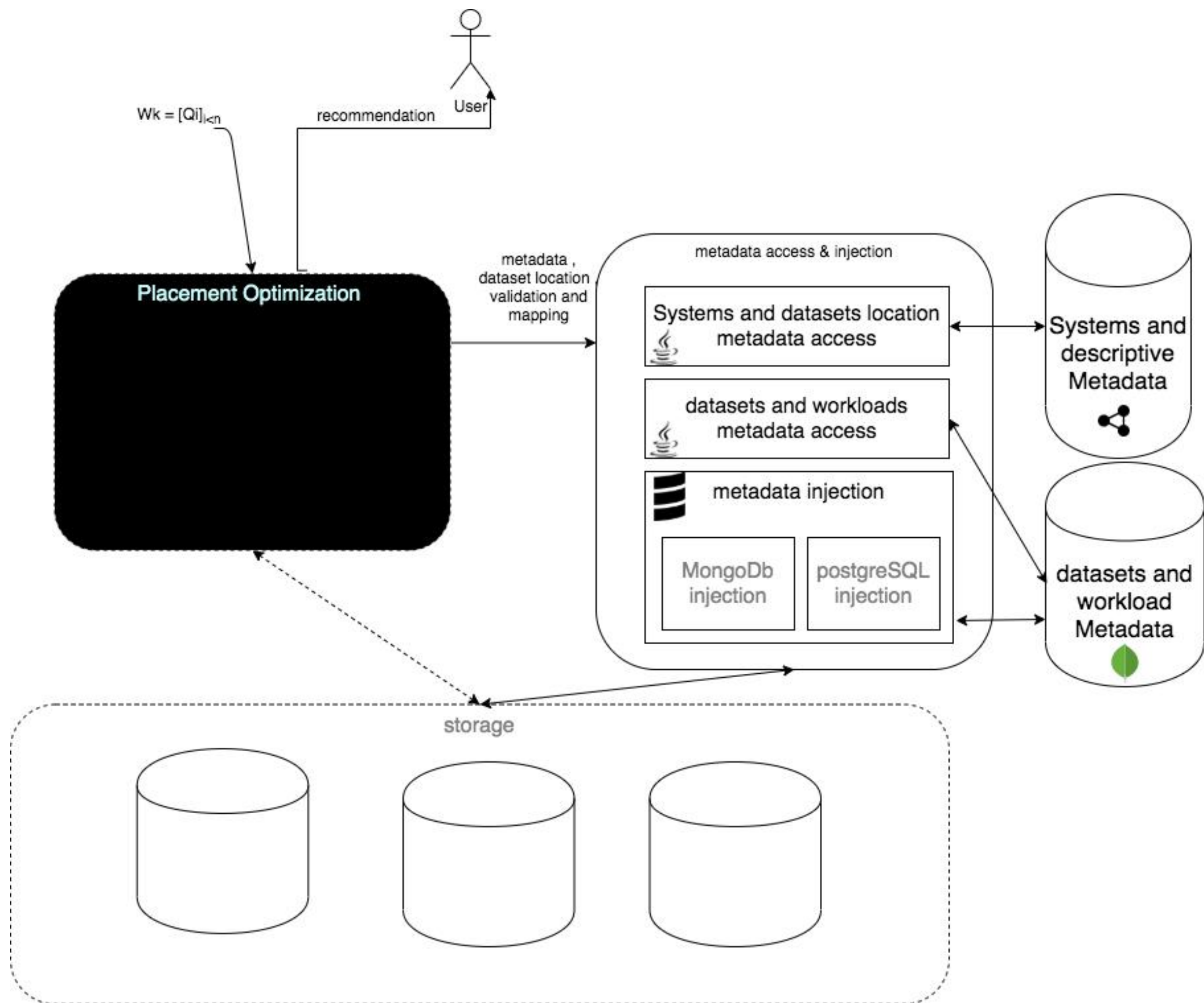



*Figure 5.3* Architecture technique du micro service « manipulation et accès aux métadonnées »

Comme le montre la figure 5.3, le module d'optimisation utilise ce micro service pour identifier les ensembles des données de l'écosystème (à partir des métadonnées indiquant la localisation de ces ensembles) et leurs statistiques. Il l'utilise aussi pour l'injection de statistiques qui est nécessaire pour permettre la simulation du placement des données. Finalement le module d'accès aux métadonnées est nécessaire aussi pour définir et mettre à jour les métadonnées. Ce module est composé de méthodes de gestion de type CRUD.

Le stockage des métadonnées est organisé comme suit : (i) dans un magasin de données RDF on stocke les métadonnées systèmes et un sous ensemble des métadonnées workload et ensembles de données, et (ii) dans un magasin de données orienté documents ou relationnel pour le stockage les statistiques. Dans la première version du prototype, le choix de la technologie de stockage a été pour le data store MongoDB.

Trois modules principaux composent le micro service « *Metadata injection and access* ». Dans ce qui suit nous présentons ces modules.

- Inférence, stockage et interrogation de schéma local

L'intégration de données et la transformation de données ne sont pas possibles sans connaissance préalable du schéma. En Big Data, pour simplifier les tâches de gestion et pour plus d'agilité dans ce type de projets, la définition de schéma au préalable n'est pas obligatoire.

Dans notre approche, on utilise Apache Spark pour inférer les types atomiques et on utilise notre propre algorithme pour améliorer la précision de l'inférence.

SparkSQL propose une API pour l'inférence de schéma basé sur l'API *Reflection*. Cette API utilise la technique de l'échantillonnage (*sampling*) pour déterminer le type des colonnes d'une dataframe. Ce n'est pas une méthode très précise. Il est recommandé de ne pas l'utiliser en production.

##### *5.1.2.3 Manipulation de données pour extraire des statistiques*

L'optimiseur de requêtes a besoin de statistiques riches et précises pour estimer efficacement le coût des plans d'exécution. Dans cet objectif, il est nécessaire d'avoir des services dédiés pour le calcul des statistiques dans notre solution. Les statistiques utilisées pour l'optimisation sont généralement des histogrammes, les cardinalités des opérateurs des plans de requêtes et la sélectivité de quelques opérateurs. Nous extrayons aussi d'autres statistiques descriptives comme le nombre d'attributs, la taille des attributs, le nombre d'enregistrements, etc. dans l'objectif de construire une base de statistiques qui sera utilisée dans le module d'apprentissage du temps de réponse.

- Algorithmes utilisés pour calculer les histogrammes

Les histogrammes sont des outils de statistiques puissants qui représentent d'une manière précise et concise la distribution des valeurs des données. Pour les attributs de type discret (cf. Statistiques) il est facile de calculer leurs histogrammes. Il s'agit de déterminer le nombre des valeurs distinctes pour chaque attribut. Pour les attributs de type continu il faut déterminer le nombre d'occurrences des valeurs de cet attribut dans des intervalles bien déterminés. Pour calculer ces intervalles il faut d'abord partitionner les valeurs de l'attribut et il existe plusieurs stratégies pour déterminer ces intervalles.

- Génération de métadonnées pour l'apprentissage

Pour transformer un plan de requêtes en un ensemble de données compréhensible par l'algorithme d'apprentissage, on exploite des techniques de préparation de données qui sont capables de générer des colonnes de type numérique à partir de données textuelles. Ces techniques sont appelées les « *Transformers* » et dans ce prototype on utilise :

- La méthode d'extraction « *Count vectoriser* » pour déterminer les propriétés du plan logique. Nos travaux sur la sélection de caractéristiques pour l'apprentissage ont déterminé que la représentation idéale du plan logique d'exécution est sous forme d'un ensemble de colonnes qui contiennent le nombre d'apparitions des opérateurs dans le plan.
- La méthode d'extraction « *multi label binarizer* » pour transformer les attributs de types chaînes de caractères en colonnes de type binaire.

D'autres fonctions ont été créées pour générer des statistiques à partir d'un plan de requête et d'un ensemble de données. Principalement des méthodes utilisées pour générer des statistiques sur des ensembles de données ayant des modèles particuliers comme des graphes :

- Calculer le nombre de nœuds dans un graph.
- Calculer le nombre de nœuds ayant des labels particuliers dans un graph.
- Calculer le nombre de nœuds entre deux labels dans un graph.

## 5.2 Expérimentation et résultats

Dans cette section, nous présentons les résultats expérimentaux obtenus en appliquant la méthode décrite dans le chapitre précédent. Nous commençons par la présentation des résultats de l'algorithme de faisabilité, ensuite ceux de la prédiction du temps de réponse et finalement on donne un exemple du déroulement de l'algorithme de recherche de solution optimale.

### 5.2.1 Description de la démarche expérimentale adaptée

Les expérimentations ont été faites principalement sur les données de notre cas d'utilisation industriel. Ces derniers représentent des mesures de consommation électrique recueilli dans le smart grids. Pour enrichir notre validation expérimentale nous présentons un exemple de déroulement de l'algorithme de placement de données sur un benchmark pour des systèmes « Big data » disponible en licence ouverte. Le tableau ci-dessous résume les données utilisées dans les différentes expériences réalisées.

| Ensemble de données | Source | Types de données | Taille | Description | Utilisation |
|---|---|---|---|---|---|
| conso_inf36 | Données ouvertes enedis (cas d'utilisation enedis) | Ensemble de données initial | 103,4 Mo | Données de consommation des clients privés de la smart grids | Démonstration et apprentissage automatique |
| conso_sup36 | Données ouvertes enedis (cas d'utilisation enedis) | Ensemble de données initial | 125,4 Mo | Données de consommation des clients Entreprises de la smart grids | Démonstration et apprentissage automatique |
| HNI_AGG_CONSO_INF36_SUP36 | Données ouvertes enedis (cas d'utilisation enedis) | Ensemble de données intermédiaire | 268,5 MO | Données de consommation totale des clients de la smart grid | Démonstration et apprentissage automatique |
| HNI_AGG_CONSO_INF0 | Données ouvertes enedis (cas d'utilisation enedis) | Ensemble de données intermédiaire | 2,56 GO | Données de consommation de la des clients de la smart grid étendue pour produire un ensemble de | Démonstration |
| HNI_AGG_CONSO_INF1 | Données ouvertes enedis (cas d'utilisation enedis) | Ensemble de données intermédiaire | 2,62 GO | Données de la consommation des | Démonstration |

| Ensembles de données du benchmark TPC-DS | HNI_AGG_CONSO_INF _PNORM | HNI_AGG_CONSO_IN F _PMOY |
|---|---|---|
| Le benchmark TPC-DS | Données ouvertes enedis (cas d'utilisation enedis) | Données ouvertes enedis (cas d'utilisation enedis) |
| – | Ensemble de données intermédiaire | Ensemble de données intermédiaire |
| Supérieure à 1 GO et inférieure à 5 GO | 2,73 GO | 1,01 GO |
| – | Données de la consommation des clients de la smart grid normalisées | Valeur moyenne de la consommation des clients de la smart grid |
| Apprentissage automatique | Démonstration | Démonstration |

*Tableau 5.1* Détails sur les ensembles de données utilisés dans l'expérimentation

Dans ce qui suit nous présentons le workload utilisé dans l'expérimentation. Ce dernier a été utilisé dans la démonstration de la recommandation produite par DWS.

| Identifiant | Requête | Résultat intermédiaire | Description |
|---|---|---|---|
| Q1 | SELECT H_DATE,CD_PROF,PLAG_PUIS ,SECT_ACT,NB_POINT_S,TOT_NRJ_S ,COUR_MOY_1,INDC_REP_COUR_1,COUR _MOY_2 ,INDC_REP_COUR_2,COUR_MOY_1_2,IND C_REP_COUR_1_2 ,JOUR_MAX_MOIS,SEM_MAX_MOIS,'SUP36 ' CD_SOUR FROM CONSO_SUP36 UNION ALL SELECT H_DATE,CD_PROF,PLAG_PUIS ,'NA' AS SECT_ACT,NB_POINT_S,TOT_NRJ_S | HNI_AGG_CONSO_ INF36_SUP36 | Une requête pour unir les deux ensembles de données de consommation |

| | | | |
|---|---|---|---|
| | ,COUR_MOY_1,INDC_REP_COUR_1,COUR_MOY_2 ,INDC_REP_COUR_2,COUR_MOY_1_2,INDC_REP_COUR_1_2 ,JOUR_MAX_MOIS,SEM_MAX_MOIS,'INF36' CD_SOUR FROM CONSO_INF36 | | |
| Exp | select g.*, generate_series(DH_COUR,DH_COUR + 30 * '1 minute'::interval, '5 minutes') as DH_COUR_5MN from (select CAST(H_DATE AS TIMESTAMP(0) ) DH_COUR , A.* from HNI_AGG_CONSO_INF36_SUP36 A limit (select count(*) from HNI_AGG_CONSO_INF36_SUP36)) g | HNI_AGG_CONSO_ INF36_SUP36_0 | Requête qui augmente la taille des données en dupliquant les données et éclate les horodates (attribut H_date) afin de passer d'un pas de 30mn à un pas de 5 mn |
| Q2 | SELECT A.*,B.FAM_PROF FROM HNI_AGG_CONSO_INF36_SUP36_0 A JOIN HNI_TRANSCO_FAM_PROF B ON A.CD_PROF=B.CD_PROF | HNI_AGG_CONSO_ INF36_SUP36_1 | Jointure entre la table principale et la table de transcodification des familles de profil |
| Q3 | SELECT AVG(SUM_TOT_NRJ_S) PUIS_MOY,DH_COUR_5MN ,CD_PROF,PLAG_PUIS,SECT_ACT FROM ( SELECT SUM(CAST(TOT_NRJ_S AS FLOAT)) SUM_TOT_NRJ_S ,DH_COUR_5MN,CD_PROF,PLAG_PUIS,SECT_ACT FROM HNI_AGG_CONSO_INF36_SUP36_1 WHERE 1=1 GROUP BY 2,3,4,5 ) T WHERE 1=1 GROUP BY 2,3,4,5 | HNI_AGG_CONSO_ INF36_SUP36_P_MOY | Calcul de la puissance moyenne de la consommation |
| Q4 | SELECT CASE WHEN PUIS_MOY <>0 THEN CAST(TOT_NRJ_S AS FLOAT)/PUIS_MOY ELSE CAST(TOT_NRJ_S AS FLOAT) END PUIS_S_NORM,A.* FROM HNI_AGG_CONSO_INF36_SUP36_1 A JOIN HNI_AGG_CONSO_INF36_SUP36_P_MOY B ON A.DH_COUR_5MN = B.DH_COUR_5MN | HNI_AGG_CONSO_INF36_SUP36_P_NORM | Normalisation de la consommation par la valeur de puissance moyenne calculée |

| | --(DATE) = B.DT_COUR<br>AND A.CD_PROF = B.CD_PROF<br>AND A.PLAG_PUIS= B.PLAG_PUIS<br>AND A.SECT_ACT = B.SECT_ACT<br>WHERE 1=1 | | |
|---|---|---|---|

*Tableau 5.2* Requêtes du cas d'utilisation étudié

### 5.2.2 Algorithmes d'inférence de la faisabilité

Dans la première partie nous avons expliqué brièvement le concept de faisabilité de placement de données et les méthodes utilisées pour les inférer. Dans ce qui suit, nous sélectionnons l'algorithme d'inférence de faisabilité et nous évaluons les différentes caractéristiques sur lesquelles se base notre apprentissage.

La première étape de l'apprentissage du critère de faisabilité est la lecture des métadonnées « Systems » des magasins de données disponible dans la base de métadonnées.

La requête utilisée pour cette étape est la suivante :

```
select ?ss ?lquerysystem ?querysystem (concat("[",group_concat(distinct
?datamodeltype;separator=","),"]") as ?dmt) (concat("[",group_concat(distinct
?qc;separator=","),"]") as ?qcc) (concat("[",group_concat(distinct
?identifier;separator=","),"]") as ?ids) (concat("[",group_concat(distinct
?llop;separator=","),"]") as ?llops) (concat("[",group_concat(distinct
?logop;separator=","),"]") as ?logops) ?qsk ?qssk (concat("[",group_concat(distinct
?wcat;separator=","),"]") as ?wkcat) ?para where { ?ss a
<http://www.enedis.fr/ontologies/2017/5/datalake#DataSystems> .?ss
<http://www.enedis.fr/ontologies/2017/5/datalake#has_data_model>/<http://www.enedis.
fr/ontologies/2017/5/datalake#type_of>/<http://www.w3.org/2000/01/rdf-schema#label>
?datamodeltype.?ss <http://www.enedis.fr/ontologies/2017/5/datalake#has_query_system>
?querysystem. ?querysystem <http://www.w3.org/2000/01/rdf-schema#label>
?lquerysystem. optional{ ?querysystem
<http://www.enedis.fr/ontologies/2017/5/datalake#query_categories> ?qc .} optional{ ?ss
<http://www.enedis.fr/ontologies/2017/5/datalake#parallelism> ?para.} optional{ ?ss
<http://www.enedis.fr/ontologies/2017/5/datalake#has_data_model>/<http://www.enedis.
fr/ontologies/2017/5/datalake#composed_of> ?record. ?record a
<http://www.enedis.fr/ontologies/2017/5/datalake#DataRecord>. optional{ ?record
<http://www.enedis.fr/ontologies/2017/5/datalake#identifier> ?identifier.}} optional{
?querysystem <http://www.enedis.fr/ontologies/2017/5/datalake#query_kind_of> ?qsk.}
optional{ ?querysystem
<http://www.enedis.fr/ontologies/2017/5/datalake#query_support_kind_of>?qssk.}
optional{ ?querysystem
<http://www.enedis.fr/ontologies/2017/5/datalake#workload_categories> ?wcat.} optional{
?querysystem
<http://www.enedis.fr/ontologies/2017/5/datalake#has_access_query>/<http://www.enedi
s.fr/ontologies/2017/5/datalake#has_logical_operator> ?rf . ?rf
<http://www.enedis.fr/ontologies/2017/5/datalake#correspondent_logical_operator>
?logop. optional{ ?logop
<http://www.enedis.fr/ontologies/2017/5/datalake#operator_name> ?llop.}} optional{
?querysystem <http://www.enedis.fr/ontologies/2017/5/datalake#has_access_query>?k .?rf
<http://www.enedis.fr/ontologies/2017/5/datalake#logical_operator_of> ?k . ?rf
```

<http://www.enedis.fr/ontologies/2017/5/datalake#correspondent_logical_operator> ?logop. optional{ ?logop <http://www.enedis.fr/ontologies/2017/5/datalake#operator_name> ?llop.}}} GROUP BY ?querysystem ?ss ?lquerysystem ?qsk ?qssk ?para

Avec la requête ci-dessus on sélectionne les propriétés des systèmes Big Data de notre problème. Ces derniers sont définis comme suit :

- **Ss** : type chaîne de caractère, désigne le nom du système.
- **Querysystem** et **Lquerysystem** : attributs de types chaîne de caractères qui représentent le moteur d'exécution et son label.
- **Dmt** : liste de chaînes de caractères, représente les types de modèles de données supportés par le système.
- **Qcc** : liste de chaînes de caractères, représente les catégories d'application d'interrogation de données supportés.
- **Ids** : liste de chaînes de caractères, désigne les caractéristiques des *ids* (un élément du modèle de données du système).
- **Llops :** liste de chaînes de caractères, représente les opérateurs physiques supportés.
- **Logops** : liste de chaînes de caractères, représente les opérateurs logiques supportés.
- **Qsk** : chaine de caractère qui renseigne le type du module d'interrogation de données : une API ou bien un langage de requête.
- **Qssk** : chaine de caractère, permet d'indiquer si le système intègre le module d'interrogation de données ou bien si on utilise un système tiers pour accéder aux données.
- **Wkcat** : liste de chaînes de caractères, représente les catégories de workloads qui sont supportés par le système.
- **Para** : booléen, indique si le système supporte le parallélisme.

Dans le tableau 5.3 nous présentons un exemple des métadonnées retrouvé en résultat de cette étape.

| Ss | lquerysystem | Querysystem | Dmt | Qcc | Ids | Llops | Logops | Qsk | Qssk | Wkcat | Para |
|---|---|---|---|---|---|---|---|---|---|---|---|
| neo4j | cypher query engine | neo4j_cypher_query_engine | [attribute based graph, document based data model] | [graph,crud] | [autogenerated] | [project] | [logical_project] | query language | natif | [interactive] | F |
| Teradata | teradata SQL engine | teradata_processing_engine | [relational data model] | [sql,Analytic,bi-tempora | [not mandatory] | [join,scan] | [logical_join,logical_scan] | query language | natif | [interactive] | F |

| | | | | l,tempor al] | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| hbase Hdfs | hbase access module | hbaseAccessApiModule | [key value based data model, column based data ... | [crud,time series] | [compound,user defined,sorted] | [scan,project] | [logical_count,logical_exist, logical_scan,inse... | api | natif | [interactive] | T |

*Tableau 5.3* Un exemple de caractéristiques utilisés dans le problème de prédiction de la faisabilité

La 2ème étape consiste à croiser ces données avec la liste des opérateurs logiques supportés par DWS et d'indiquer pour chaque système si l'exécution de chaque opérateur est supportée par ce dernier. Par la suite, nous transformons les variables de types complexes en type numérique. Pour ceci nous utilisons la technique d'encodage *multilabel encoder* pour les variables de type liste de chaine de caractères et la technique *label encoder* pour les variables de type chaîne de caractères.

L'apprentissage a été mené sur un ensemble de données ayant 270 enregistrements et 14 colonnes. La division entre ensemble d'apprentissage et de test était de 70% pour le premier groupe et 30 % pour le 2ème. Le tableau 5.4 résume la performance des différents algorithmes évalués.

| **Algorithme** | **Paramètres** | **Qualité d'apprentissage** |
|---|---|---|
| Stochastic Gradient Descent **(**SGDClassifier**)** | alpha=0.0001 | Précision = 0,73 |
| Support Vector Machines (SVM) | GridSearchCV sur les différentes configurations des paramétres suivants : 'C' : [0.01, 0.1, 1, 10,100],'kernel':('linear', 'rbf') et la métrique de selection de la meilleure configuration : score f1 | Précision = 0.84 |

*Tableau 5.4* Métriques de performance de la prédiction conduisant au choix de l'algorithme le plus adéquat

### 5.2.3 Algorithmes d'inférence de la conformité

Le critère de conformité dépend des règles métier définies par un expert du domaine ou bien par le propriétaire des données pour imposer une solution particulière de placement ou bien pour la restreindre. Dans ce prototype nous avons défini trois types de règles métiers :

- Restriction d'une solution de stockage pour une application
- Imposer une solution de stockage pour une application
- Imposer une solution de stockage pour un ensemble de données

Ces règles sont représentées sous forme de métadonnées qui ont la structure suivante :

StoreRestriction (application, dataset, store, exclude_store)

En suivant cette spécification nous avons créé trois objets pour la proposition d'un modèle d'inférence de la conformité en suivant le cas d'utilisation métier de notre problématique de recherche.

Restrict_conso_sup: **StoreRestriction** (**store :** "postgres",
**dataset** :"conso_sup36")

profiling_app_restriction : **StoreRestriction** (**exclude_store** :"HDFS", **application** :"profiling")

application_restriction : **StoreRestriction** (**store** **:** "HDFS", **application** :" anonymised _application" )

Le problème d'inférence est une classification de toutes les solutions possibles de placement selon leurs conformités avec les contraintes définies sous forme de métadonnées. On vérifie pour chaque n-uplet (ensemble de données, application, store) qui représente les candidats de placement s'ils sont conformes avec les contraintes. Le type de classification pour ce problème est une classification binaire et on définit une variable « conf » qui représente la classe.

La première étape de l'apprentissage consiste à lire tous les objets (métadonnées) de type « StoreRestriction ». On utilise la requête Sparql suivante :

```
select * where { ?s a <http://www.enedis.fr/ontologies/2017/5/datalake#StoreRestriction> . optional{ ?s <http://www.enedis.fr/ontologies/2017/5/datalake#dataset> ?ds.}  optional{ ?s <http://www.enedis.fr/ontologies/2017/5/datalake#application> ?appli.} optional{ ?s <http://www.enedis.fr/ontologies/2017/5/datalake#store> ?store.} optional{ ?s <http://www.enedis.fr/ontologies/2017/5/datalake#exclude_store> ?exclude_store.} }
```

*Le résultat de cette requête est représenté dans le tableau 5.5.*

| s | ds | Appli | store | exclude_store |
|---:|---:|---:|---:|---:|
| restrect_conso_sup | conso_sup36 | NaN | postgres | NaN |
| application_restriction | NaN | Anon_app | hdfs | NaN |
| profiling_app_restriction | NaN | Profiling | NaN | HDFS |

*Tableau 5.5* Un exemple de caractéristiques utilisés dans le problème de prédiction de la conformité

Par la suite, on croise ces métadonnées avec d'autres métadonnées : on souhaite obtenir pour chaque application les ensembles de données associés aux versions manipulées dans ces applications (voir les classes « DSVersion », « DataSet » du chapitre 3).

La requête utilisée est la suivante :

```
select * where { ?s a <http://www.enedis.fr/ontologies/2017/5/datalake#Application> .  ?s <http://www.enedis.fr/ontologies/2017/5/datalake#uses> ?dsversion. ?ds <http://www.enedis.fr/ontologies/2017/5/datalake#derived_as> ?dsversion.}'
```

Finalement on sélectionne toutes les solutions de stockage et d'interrogation de données qu'on peut trouver dans l'écosystème en utilisant la requête suivante :

```
select * where { VALUES ?p { <http://www.enedis.fr/ontologies/2017/5/datalake#integrates_data_store> <http://www.enedis.fr/ontologies/2017/5/datalake#uses_processing_engine> } <http://www.enedis.fr/ontologies/2017/5/datalake#asma_test_datalake> ?p ?s.}'
```

Par la suite on croise le résultat des trois requêtes et on définit la variable de conformité (qui est initiée à zéro) comme suit :

si exclude_store != ecosystem_store alors conf =1

si store = ecosystem_store alors conf = 1

si store != ecosystem_store alors conf = 0

Après ces transformations l'ensemble de données d'apprentissage devient :

| ecosystem_store | Ds | appli | Store | exclude_store | Conf |
|---|---|---|---|---|---|
| S3 | conso_sup36 | | postgres | | 0 |
| Hdfs | conso_inf36 | Profiling | | Hdfs | 0 |
| Hdfs | conso_sup36 | Profiling | | Hdfs | 0 |

*Tableau 5.6* Transformation des caractéristiques et initiation de la variable à prédire

Vu que les algorithmes d'apprentissage sont adaptés majoritairement aux variables numériques, on utilise l'encodage des labels pour transformer ces données vers un ensemble plus adapté à l'étape finale. On utilise la technique : *OneHotEncoder* pour les variables : **ecosystem_store** , **store** et **exclude_store** et *LabelEncoder* pour les variables **appli** et **ds**.

L'ensemble d'apprentissage obtenu a 323 enregistrements et 60 colonnes.

L'algorithme choisi pour le problème est l'algorithme d'arbre de décision. L'entraînement a été fait sur 133 enregistrements et le test sur 123 enregistrements et le score du modèle produit est 1 (score parfait vu la simplicité des règles de prédiction de la conformité).

La figure 5.4 montre la courbe ROC du modèle de classification.

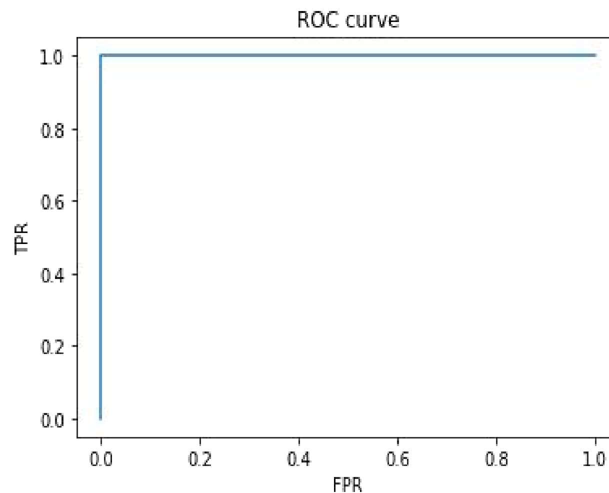



*Figure 5.4* Courbe ROC du modèle de classification selon la conformité

L'intérêt de cette méthode est la facilité de définir les contraintes de conformité et permet les utilisateurs d'éviter de définir des règles métiers complexes. Mais il reste nécessaire de réapprendre à chaque ajout de nouvelles contraintes de conformité.

### 5.2.4 Apprentissage automatique pour prédire le temps de réponse pour les systèmes Big Data

Dans cette section nous détaillons les variables qu'on a choisies pour la prédiction pour chaque type de moteur de traitement de données.

#### 5.2.4.1 Sélection des variables pour le problème de régression

Les travaux précédents nous ont permis de déterminer une nouvelle méthode pour profiler les systèmes et déterminer de nouvelles variables sur lesquelles on exécutera l'apprentissage automatique pour prédire le temps d'exécution.

Notre approche pour déterminer les variables de prédiction consiste à explorer les modèles de coûts existants et à déterminer les paramètres de données, de requête et de systèmes qui sont utilisés dans ces modèles. Dans cette section nous présentons les variables que nous avons sélectionnées pour chaque nœud d'apprentissage automatique et nous validons ces variables en utilisant des tests statistiques adaptés au problème.

Les variables suivantes seront les plus déterminant dans la prédiction du temps de réponses vu qu'ils sont utilisés dans l'estimation du coût de l'exécution dans les optimiseurs de l'état de l'art (cf. chapitre 1) : (i) la taille des données représentée par le nombre d'enregistrements et le nombre d'attributs. (ii) le plan de la requête physique et logique qu'on peut représenter par le nombre d'occurrences de chaque opérateur dans les plans et finalement (iii) la sélectivité du filtre de la clé de jointure ou bien de l'index.

Dans ce qui suit, nous vérifions expérimentalement la corrélation entre ces variables et la variable à prédire : le temps d'exécution.

- Vérification expérimentale

**<u>But de l'expérimentation</u> : Déterminer les variables, les algorithmes et les paramètres nécessaires pour le problème de la prédiction du temps de réponse**

Dans notre approche, nous avons collecté des métadonnées, des statistiques et des mesures de performance dans une base de requêtes historique. A partir de ces enregistrements et en se basant sur une méthodologie d'analyse expérimentale et une étude théorique des modèles de coûts des optimiseurs des systèmes existant, nous déterminons l'ensemble de caractéristiques qui influencent significativement la performance.

L'étude expérimentale a montré que les paramètres de données pertinents sont la taille des données et (ii) le nombre d'enregistrements.

Pour les paramètres de requête nous avons déterminé :

- Le ratio de sélectivité des requêtes
- Le ratio de projection des requêtes : *width of proj attributes / width of table*
- Le nombre de groupes calculés pour les opérateurs d'agrégations et le type d'agrégations
- La représentation vectorielle des opérateurs des requêtes

Dans cette étude, on n'a pas déterminé les paramètres des systèmes. On peut toutefois confirmer que le nombre de processeurs et le nombre de serveurs (entre autres) influencent la performance des requêtes. Dans les sous sections suivantes, nous présentons en détail les paramètres de données et de requêtes que nous avons identifiés comme pertinents. Pour justifier leur importance dans l'estimation du temps d'exécution, nous utilisons la méthode de corrélation statistique.

- Etude de corrélation

Dans cette partie nous étudions l'influence des métadonnées DWS sur les mesures de performance collectés des systèmes du lac de données. Pour chaque paramètre qu'on examine, nous visualisons la variation du temps d'exécution en fonction de la valeur de ce paramètre. Ensuite nous mesurons la

corrélation entre le paramètre et le temps de réponse et finalement nous déterminons les paramètres nécessaires pour le problème de prédiction en éliminant ceux qui sont corrélés entre eux.

Les données utilisées dans cette expérience ont été collectées sur un serveur linux qui a un processeur à 8 cœurs et une taille mémoire de 32Go.

- *Cas d'un magasin de données de type « NoSQL »*

L'exécution des requêtes dans les magasins de données de ce type est faite via une interface spécifique en utilisant des appels de fonctions et rarement en utilisant un langage déclaratif. Ainsi, l'optimisation de l'exécution des traitements n'est pas faite d'une manière traditionnelle comme on a expliqué dans notre étude de l'état de l'art. Ce type de système n'estime pas le coût de l'exécution. On ne peut donc pas utiliser cette mesure dans la prédiction du temps de réponse et il faut déterminer donc d'autres paramètres qui présentent une corrélation forte avec le temps d'exécution.

Dans notre méthode, on fait varier les statistiques sur les données et sur les requêtes pour chaque opérateur identifié et supporté par le magasin de données. Nous allons étudier pour ce cas d'exécution des workload d'abord dans le magasin de données NoSQL Hbase et puis dans le magasin de données orienté graphes Neo4j.

Corrélation des métadonnées de l'opérateur « scan »

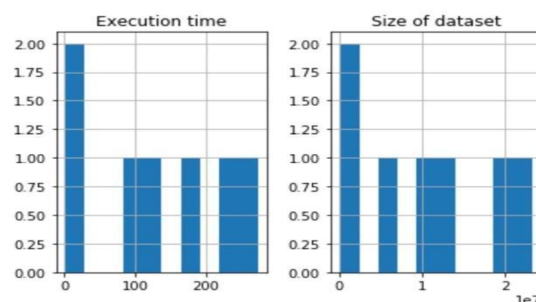
L'opérateur scan permet de balayer les données en mémoire à partir de leurs supports de stockage. Il existe plusieurs types d'implémentation de cet opérateur :

(i) Le balayage total des données,

(ii) Le balayage des données en utilisant un index

(iii) Et le balayage des données en utilisant le hachage.

Il existe des paramètres en communs pour ces différentes implémentations et des paramètres qui sont spécifiques à des types bien déterminées. Le premier paramètre à examiner est la taille des données. Dans une première expérience nous avons fait varier la taille des données et nous avons collecté le temps de réponse de quelques requêtes de sélection de données. Le système de stockage utilisé dans cette expérience est Apache Hbase et le moteur d'exécution est Apache Phoenix. La figure 5.5 illustre la distribution des mesures collectées :




| | count | mean | std | min | 25% | 50% | 75% | max |
|---|---|---|---|---|---|---|---|---|
| **Size of dataset** | 20.0 | 6.340496e+06 | 7.781611e+06 | 28.000 | 159665.250 | 2866890.000 | 1.056249e+07 | 2.327930e+07 |
| **Execution time** | 20.0 | 7.975995e+01 | 8.873268e+01 | 0.218 | 1.776 | 51.831 | 1.336302e+02 | 2.723430e+02 |

*Figure 5.5* Distribution des valeurs des variables "taille des données" et "temps de réponse" pour l'expérience de sélection de données pour Apache HBase/ Phoenix)

La fiigure 5.5.b montre la variation du temps de réponse en fonction de ce paramètre pour le balayage total des données (scan) ainsi que leur corrélation.

| | Size of dataset | Execution time |
|---|---|---|
| Size of dataset | 1.00000 | 0.97294 |
| Execution time | 0.97294 | 1.00000 |

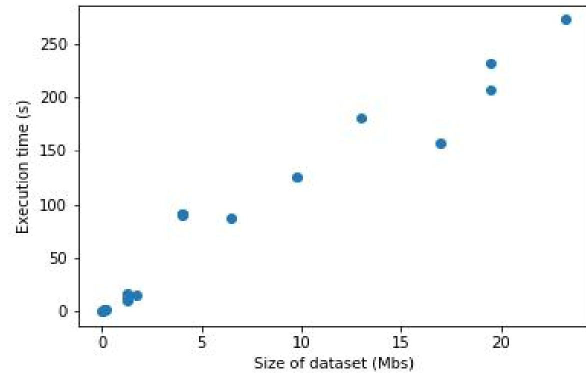



*Figure 5.5.b Matrice de corrélation et nuage de points de la variable "temps de réponse" en fonction de la variable "taille des données"*

L'expérience montre que le temps de réponse varie linéairement avec la taille des données. L'indice de corrélation très proche de 1 renforce cette constatation et indique que le paramètre étudié sera déterminant dans le modèle de prédiction.

Le deuxième paramètre est le nombre d'enregistrements. Le nuage de point illustré ci-dessous (Figure 5.6) montre que le nombre d'enregistrements n'est pas parfaitement linéaire avec le temps de réponse. Ce constat est renforcé avec les valeurs de corrélation des deux variables qui présentent des valeurs inférieures à celles de la taille de données.

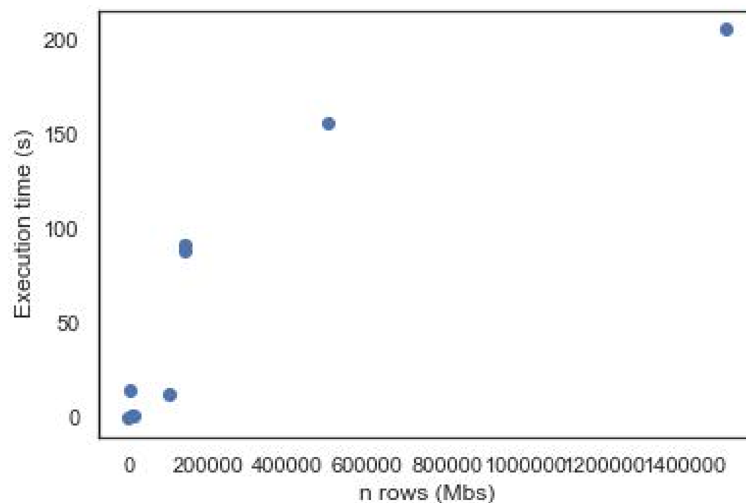



*Figure 5.6* Nuage de points de la variable "temps de réponse" en fonction de la variable "nombre d'enregistrement"

| | Size of dataset | Execution time | n rows |
|---|---|---|---|
| Size of dataset | 1 | 0.939586 | 0.893748 |
| Execution time | 0.939586 | 1 | 0.851789 |
| n rows | 0.893748 | 0.851789 | 1 |

*Figure 5.7* Comparaison de la corrélation des nombres d'enregistrements avec la corrélation avec la taille de données

En effet, la matrice de corrélation précédente (figure 5.7) permet de comparer la corrélation des variables utilisées comme observations dans notre problème d'apprentissage automatique. D'après les résultats de notre expérience on observe que les paramètres « taille des données » et « nombre d'enregistrements » sont très corrélés entre eux. Ils montrent aussi que le paramètre « taille de données » à un coefficient de corrélation plus proche de 1 avec la variable à prédire (le « temps de réponse »). Par conséquent, on peut conclure que la variable à prédire dépend plus de la variable « taille de donnée » et il est sélectionné comme caractéristique (*feature*) dans notre problème d'apprentissage.

Dans l'expérience précédente, nous avons aussi mesuré le nombre d'attributs dans chaque ensemble de données interrogé. Le coefficient de corrélation entre cette variable et la variable à prédire est égale à 0.376220. La valeur de corrélation est inférieure à 0.6. On peut donc conclure que la relation linéaire est trop faible entre ces derniers pour considérer le nombre d'attribut comme caractéristique dans le problème d'apprentissage.

- Corrélation des métadonnées de l'opérateur agrégation

Dans cette section, nous souhaitons déterminer les variables spécifiques aux opérateurs d'agrégation. Nous proposons d'abord d'étudier les différentes fonctions d'agrégation et leurs influences sur le temps d'exécution de la requête. Dans cet objectif, nous réalisons l'expérience numéro 2 suivante. Pour plusieurs ensembles et sur différents attributs de ces derniers, on lance des requêtes de la forme suivante :

```
Select attribut_i , aggregation_function(attribut_j)
from ensemble_k group by  attribut_j
```

Sachant qu'on désigne par *aggregation_function* la fonction d'agrégation qui prend plusieurs valeurs comme *avg* , *min* , *count*, … (ces valeurs dépendent du langage d'interrogation de données ou bien de l'API utilisée pour l'accès aux données) , *attribut_i* les attributs accédés au cours de la requête et *ensemble_k* l'ensemble de donnée avec *i* , *j* et *k* des nombres entiers. Dans cet exemple on a un attribut de projection : *attribut_i* et un attribut d'agrégation : *attribut_k*.

Les résultats de cette étude sont collectés à partir de mesures faites sur des requêtes exécutées dans le magasin de données orienté graphes Neo4J. Le langage d'interrogation de données de ce dernier (Cypher) supporte les fonctions d'agrégations suivantes : *avg* , *min* , *max* , *count*. Dans cette expérience sur un seul ensemble de données, on exécute un workload composé de requêtes d'agrégations sur un seul attribut à la fois et en utilisant des fonctions d'agrégation différentes. Pour l'étude des fonctions on se limite aussi aux fonctions *avg* de calcul de moyenne et *count* de dénombrement pour la preuve du concept. La première étape d'analyse des résultats consiste à reporter la distribution des résultats collecté ensuite à étudier l'interdépendance des variables pour arriver finalement à sélectionner les caractéristiques nécessaires pour notre problème de régression. Dans notre méthode nous avons représenté chacune de ces fonctions par un attribut qui prend comme valeur la taille des attributs d'agrégation. Pour calculer cette valeur on commence par associer à chaque attribut une valeur binaire qui indique si une fonction existe dans la requête. Par la suite, on multiplie cette valeur par la taille de chaque attribut qui a été accédé par l'opérateur d'agrégation. Le calcul de cette taille se fait en utilisant des informations sur la taille en mémoire des types supportés par le système de traitement de données. Cette information est stockée dans la couche de métadonnées. La figure ci-dessous (Figure 5.6) représente la distribution des différents attributs qui représentent les fonctions d'agrégation. L'expérience a produit 750 mesures sur lesquelles se base notre analyse.

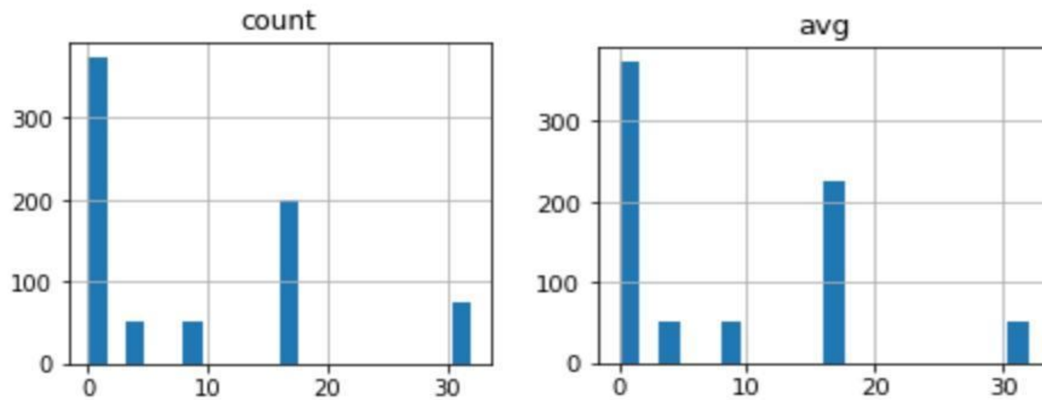



*Figure 5.8* Distribution des valeurs des attributs "cout" et "avg" qui ont été collectées des résultats de l'expérience 2

La figure 5.9 présente une matrice de corrélation des différentes fonctions d'agrégation et le temps de réponse dans le magasin de données Neo4j.

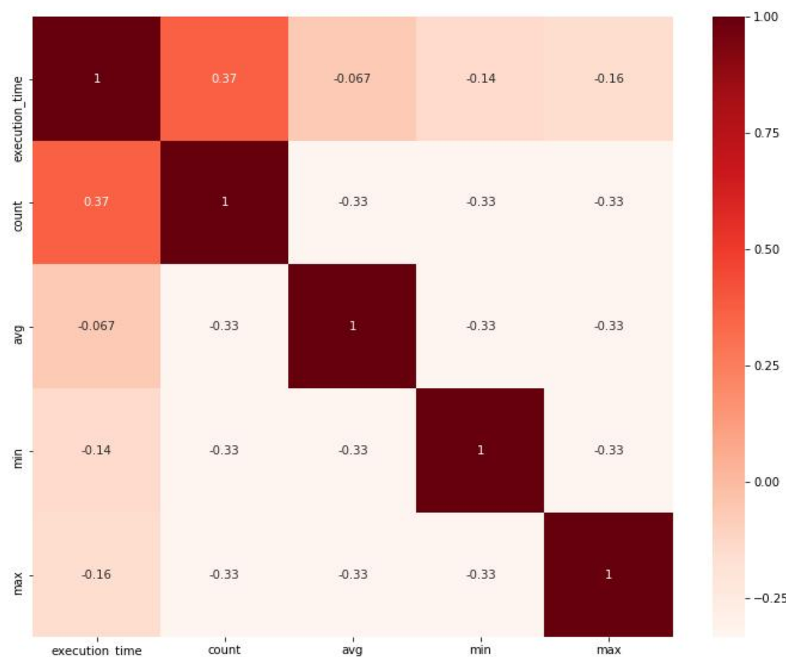



*Figure 5.9* Matrice de corrélation des attributs observés pour l'expérience numéro 2 : influence des fonctions d'agrégation

Le résultat de cette expérience montre que les fonctions d'agrégation dans Neo4j n'ont pas d'impact sur le temps de réponses d'où la corrélation faible entre les attributs étudiés et la variable de prédiction (le temps de réponse). Dans notre méthode on ne sélectionne que les caractéristiques qui ont un coefficient de corrélation supérieur à 0.6 avec la variable objective. On peut donc conclure que les fonctions d'agrégation ne sont pas déterminantes pour prédire le temps de réponse.

On propose ensuite de mesurer le nombre de groupes calculés par l'opérateur d'agrégation. Ce nombre de groupes correspond en réalité à la cardinalité du résultat intermédiaire retourné par cet opérateur. Il est facile de le déterminer à partir du plan logique de la requête si le système est capable d'estimer cette statistique (comme dans le cas du moteur d'exécution de Neo4j). Dans l'expérience suivante on utilise les sous-ensembles de données créés dans Neo4j à partir de l'ensemble de données de départ (de notre cas d'utilisation) pour l'étude de l'influence de la taille des données sur le temps

de réponse et qui possèdent des distributions variées des attributs. La figure suivante (Figure 5.10) présente la matrice de corrélation de l'attribut étudié et du temps d'exécution.

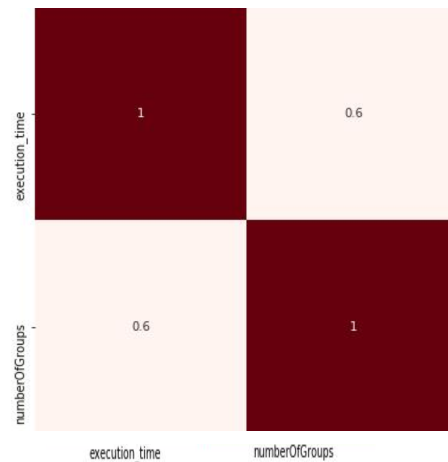



Figure 5.10 *Matrice de corrélation des variables : "number_of_groups" et "execution_time"*

La matrice montre que le coefficient de corrélation entre l'attribut « *number_of_groups* » et « *execution_time* » est égale à 0,6. On peut conclure que cet attribut est une caractéristique convenable pour notre problème.

- *Cas d'un moteur de médiation*

Dans cette section on détermine les caractéristiques du problème de prédiction du temps de réponse pour un moteur de médiation largement utilisé dans le domaine du Big Data : Apache Spark (cf chapitre 2). Un autre système que nous avons présenté dans les chapitres précédents (Apache Calcite) peut être étudié dans cette sous-section mais pour éviter la redondance dans notre étude expérimentale nous considérons la caractérisation du problème de prédiction de ce système comme perspective.

- Caractéristiques du problème de prédiction dans Apache Spark

Cette expérience a été réalisée sur une seule machine qui est caractérisée par un processeur i7, 16 GO de mémoire totale et le système d'exploitation Mac OS. Dans ce cadre expérimental nous avons généré 15558 mesures ayant 13 attributs définis comme suit :

- **attributes_count**: type int64, une variable qui représente le nombre d'attributs dans l'ensemble de données.
- **cardinality** : type int64, une variable qui représente le nombre d'enregistrements
- **records_count** : variable de type réel qui représente le nombre total d'enregistrements
- **size**: type float64 , variable qui représente la taille des données sur disque exprimé en méga octets.
- **sum_max_rows_partition**: variable de type réel qui représente le nombre maximum d'enregistrements dans chaque partition.
- **logical_aggragate** ,**logical_count**, **logical_project**, **logical_scan**, **logical_sort**: type int64, variables binaires qui correspondent à l'encodage du plan logique de la requête
- **exchange**, **file**, **scan**, **project**, **scan_collection**, **sort**: type int64, variables binaires qui correspondent à l'encodage du plan physique de la requête

- **count**, **grouping**, **groupingaggregate** : type int64, variables binaires qui correspondent à l'encodage du type d'agrégation
- **execution_time** : type float64, une variable qui représente le temps de réponse mesuré dans le cadre de cette expérience.

Le tableau 5.7 et la figure 5.11 représentent une description de la distribution des valeurs générées.

| | Count | Mean | Std | Min | 25% | 50% | 75% | Max |
|---|---|---|---|---|---|---|---|---|
| attributes_ count | 15558.0 | 2.919668e+01 | 1.234971e+01 | 1.0 | 30.0 | 34.0 | 34.0 | 4.820000e+02 |
| Cardinality | 15545.0 | 3.157500e+05 | 2.395793e+05 | 1.0 | 1.0 | 499969.0 | 499969.0 | 2.345805e+06 |
| execution_ time | 15558.0 | 1.044686e+09 | 1.755080e+09 | 33122451.0 | 206323149.0 | 283812642.5 | 566552046.0 | 4.656694e+10 |
| records_ count | 15558.0 | 7.390584e+05 | 2.259209e+06 | 137.0 | 78096.0 | 499969.0 | 499969.0 | 1.574923e+07 |
| Size | 15558.0 | 1.021667e+08 | 1.884569e+08 | 1001.0 | 33961749.0 | 100006677.0 | 100006677.0 | 1.794836e+09 |
| logical_aggragate | 15558.0 | 3.200926e-02 | 1.760303e-01 | 0.0 | 0.0 | 0.0 | 0.0 | 1.000000e+00 |
| logical_count | 15558.0 | 3.636071e-01 | 4.810529e-01 | 0.0 | 0.0 | 0.0 | 1.0 | 1.000000e+00 |
| logical_ project | 15558.0 | 8.699704e-01 | 3.363468e-01 | 0.0 | 1.0 | 1.0 | 1.0 | 1.000000e+00 |
| logical_scan | 15558.0 | 1.000000e+00 | 0.000000e+00 | 1.0 | 1.0 | 1.0 | 1.0 | 1.000000e+00 |
| logical_sort | 15558.0 | 4.118139e-01 | 4.921776e-01 | 0.0 | 0.0 | 0.0 | 1.0 | 1.000000e+00 |
| Exchange | 15558.0 | 4.806530e-01 | 4.996416e-01 | 0.0 | 0.0 | 0.0 | 1.0 | 1.000000e+00 |
| file scan | 15558.0 | 9.998714e-01 | 1.133768e-02 | 0.0 | 1.0 | 1.0 | 1.0 | 1.000000e+00 |
| key : aggregate, value :count | 15558.0 | 3.969019e-01 | 4.892711e-01 | 0.0 | 0.0 | 0.0 | 1.0 | 1.000000e+00 |
| key : aggregate, value :grouping , label : groupingaggregate | 15558.0 | 3.136650e-02 | 1.743118e-01 | 0.0 | 0.0 | 0.0 | 0.0 | 1.000000e+00 |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **Project** | 15558.0 | 5.468569e-01 | 4.978156e-01 | 0.0 | 0.0 | 1.0 | 1.0 | 1.000000e+00 |
| **scan collection** | 15558.0 | 2.056820e-03 | 4.530696e-02 | 0.0 | 0.0 | 0.0 | 0.0 | 1.000000e+00 |
| **Sort** | 15558.0 | 1.707160e-01 | 3.762727e-01 | 0.0 | 0.0 | 0.0 | 0.0 | 1.000000e+00 |

*Tableau 5.7* Distribution des valeurs des caractéristiques générées pour le problème de prédiction

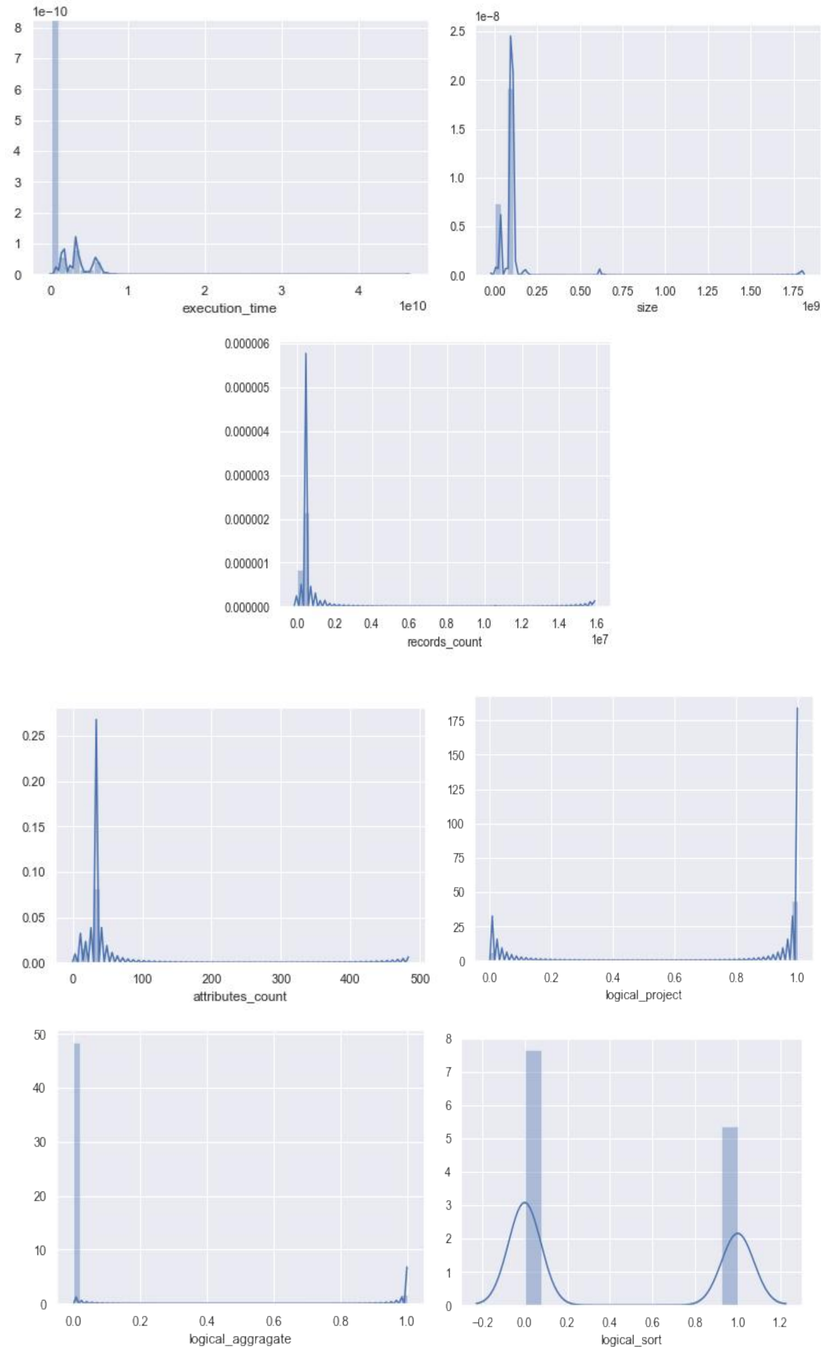



*Figure 5.11* Diagrammes de distribution des valeurs des caractéristiques générées pour le problème de prédiction

Ces observations montrent que les mesures ont été faites sur des caractéristiques variées. Les benchmarks ont concerné des données massives mais aussi de tailles moyennes et réduites. La distribution des variables *attributes_count* et *records_count* montrent que nous avons diversifié les

expériences en considérant des nombres différents pour les enregistrements ainsi que pour les attributs des ensembles des données. Les types d'opérateurs du workload sont aussi variés comme les montre les distributions des figures ci-dessus. La distribution de la variable *logical_sort* montre que l'opérateur du tri existe dans presque la moitié des requêtes. Celle de la variable *logical_aggregate* montre qu'elle existe dans 10% de ces derniers, et celle de la variable *logical_project* montre qu'elle existe dans 90% des requêtes.

L'étape suivante consiste à observer la variation de l'objectif de la prédiction (le temps d'exécution) avec les autres variables pour pouvoir sélectionner le type d'algorithme d'apprentissage il faut utiliser. On détermine d'abords le coefficient de corrélation des variables avec l'objectif de la prédiction. Ce dernier nous donne une information sur l'existence d'une relation linéaire entre ces derniers.

| | attributes_count | Cardinality | execution_time | records_count | size | logical_aggregate | logical_count | logical_project | logical_sort | Exchange | key : aggregate, value :count | key : aggregate, value :grouping | Project | Sort |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **execution_ti** | 0.155819 | 0.217715 | 1.000000 | 0.194068 | 0.172773 | 0.042285 | 0.124215 | -0.402089 | 0.373037 | 0.352417 | 0.133857 | 0.038368 | -0.036283 | 0.767696 |

*Tableau 5.8* Corrélations avec l'objectif de prédiction pour quelques variables de l'expérience 3

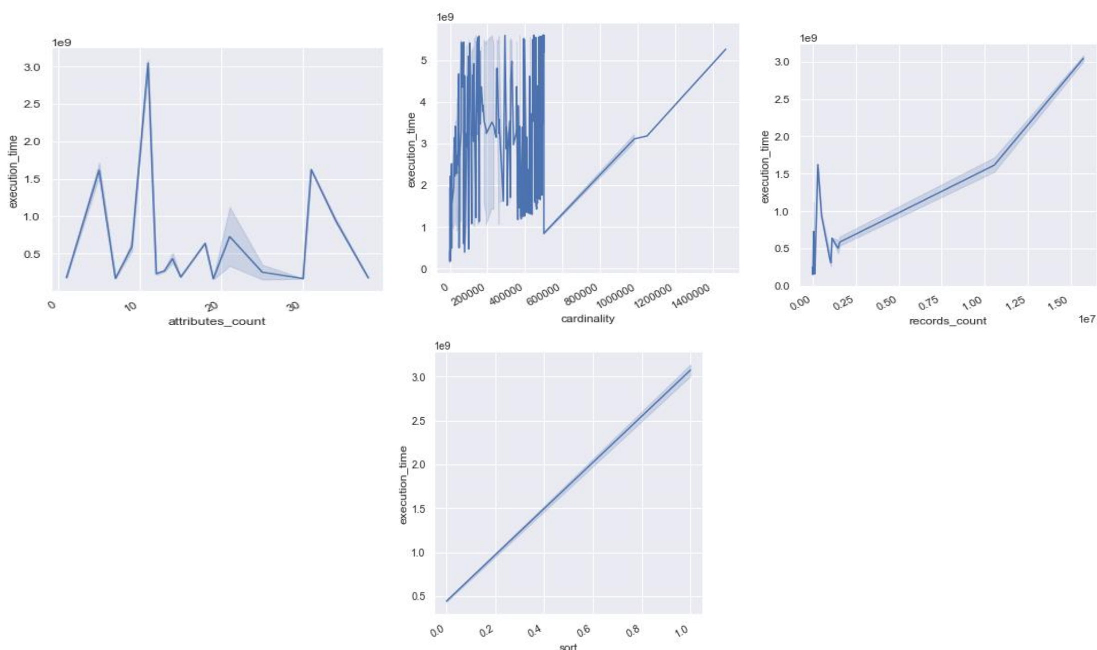



*Figure 5.12* Diagrammes de distribution de données pour quelques variables de l'expérience 3 (2)

D'après ces observations on peut conclure que la dépendance entre les variables et l'objectif de la prédiction n'est pas linéaire et que l'utilisation d'un modèle de prédiction non linéaire est nécessaire dans ce cas.

- Représentation vectorielle du plan de la requête

Le choix de la transformation adéquate des plans de requêtes et leurs représentations sous forme de vecteur intégrable avec le vecteur des autres caractéristiques est un facteur important dans la réussite d'un problème de prédiction par apprentissage. Dans notre méthode, on extrait les opérateurs du plan de requête et on compte le nombre d'occurrences des opérateurs en utilisant des transformations simples. Ces transformations permettent de générer un vecteur de nombres réels à partir d'un attribut de type texte ou bien de type tableau et ils sont appelés des encodeurs.

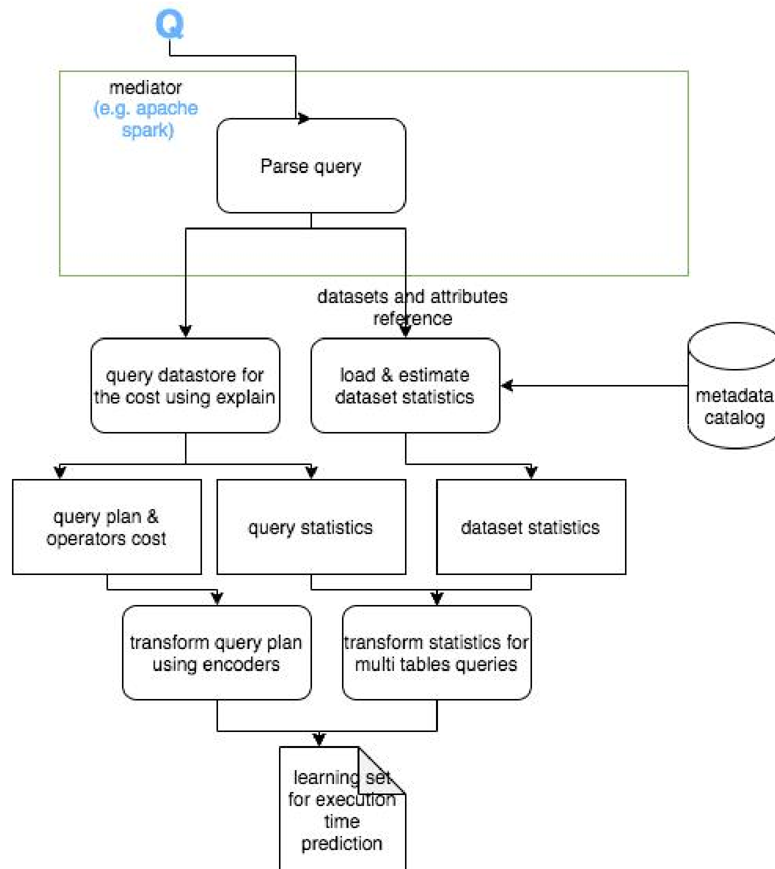



*Figure 5.13* Générer l'ensemble de données d'apprentissage à partir des requêtes et des ensembles de données historiques

Comme le montre la figure 5.13, le premier composant de notre transformateur de plans de requête extrait les opérateurs du plan en entrée, les ensembles de données (par exemple les tables accédées par une requête SQL), les coûts des opérateurs calculés par l'optimiseur s'ils existent et des statistiques sur les opérateurs (comme le nombre d'enregistrements) s'elles existent. Toutes ces informations seront stockées dans des attributs de type tableaux et ils seront à leurs tours transformés. Par la suite, le deuxième composant sert à construire à partir de l'attribut de type tableau d'opérateur, une matrice qui a comme colonnes les différents types d'opérateurs supportés par le système étudié et qui pour chaque enregistrement de l'ensemble des mesures associé aux colonnes des opérateurs le nombre d'occurrences de ce dernier dans le tableau d'opérateur calculé à l'étape précédente. L'attribut suivant à transformer est celui des ensembles des données. Un croisement de la table des mesures avec la table des statistiques sur les ensembles de données nous permet d'enrichir les caractéristiques par ces statistiques sur les données. Pour finir, on transforme les attributs sur les coûts et les statistiques des opérateurs en utilisant une opération mathématique simple comme la somme des coûts ou bien la valeur maximale des coûts suivant les modèles de coût utilisés par les systèmes.

L'exemple suivant illustre une transformation obtenue du processus décrit ci-dessus (figure 5.14). Cet exemple correspond à une observation de l'ensemble de données qui est utilisé pour l'apprentissage dans notre problème de prédiction du temps d'exécution d'une requête dans le magasin de données Neo4j. Il correspond à une extraction de métadonnées à partir d'une requête et de l'ensemble de données qui lui est associé. Le schéma du n-uplet d'entrée est : « *query, dataset, execution_time* » et l'étape finale produit à la sortie un n-uplet dont le schéma est le suivant :

*«stat1 , stat2, nb_joins, nb_agg, max_cost, min_cost, mean_cost,filter, join,project,scan,filter,join,project,scan,Filter,AllNodesScan,Expand,NodesByLabelScan,ProduceResults,Projection»*

Les transformations effectuées générer le n-uplet final de caractéristiques sont décrites dans la figure ci-dessous.

<MATCH (n:ConsoInfSup) RETURN count(n.JOUR_MAX_MOIS), ConsoInfSup>

Benchmark et mesures

<"dataset": "ConsoInfSup", "execution_time": NumberLong("223907811"),"unit": "ms", "processing_engine": "neo4j", "QL_ACCAPI": "cypher", "operators": [ "ProduceResults", "EagerAggregation", "NodeByLabelScan"], "arguments": [ "[count(n.JOUR_MAX_MOIS)]", "[count(n.JOUR_MAX_MOIS)]", "[n]"], "{KeyNames=\"\", EstimatedRows=1.0}", "{EstimatedRows=62812.0, LabelName=\":ConsoInfSup\"}"], "rowcount": ["1.0", "1.0", "62812.0" ]>

Transformations et croisement avec les statistiques

| AllNodesScan | Expand All | NodeByLabelScan | ProduceResults | Projection | stat_1 | stat_2 |
|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 1 | 1 | 103357026.0 | 0.0 |

| nb_joins | nb_agg | **max_cost** | **min_cost** | **mean_cost** | **Filter** | **Join** | **project** | **scan** | **Filter** |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1436641 | 1436641 | 1.436641e+06 | 0 | 0 | 1 | 1 | 0 |

*Figure 5.14* Transformations effectuées sur une requête pour obtenir le tableau des caractéristiques

#### 5.2.4.2 *Résultats et modèles d'apprentissage*

Dans cette section nous présentons les résultats obtenus de l'étape d'apprentissage pour quelques systèmes que nous comparons. La collecte des données de mesures de performances est faite sur un serveur local ou sur une grille de calcul distribuée (Grid5000) selon le type du système étudié.

- Résultats de l'apprentissage du temps de réponse de l'exécution dans « Neo4j » :

Dans cette expérience nous avons collecté 30 enregistrements qui ont comme attributs :

- **costs** : type *object* , un tableau de valeurs réelles (*float64*) qui désigne la liste des coût de chaque opérateur dans le plan de la requête exécuté. Cette variable sera transformé en 3 variables réelles : « cumulated_cost » , « max_cost » , « min_cost » , « mean_cost » , « median_cost » qui désignent , le coût cumulé du plan calculé comme la somme des coûts unitaire , la valeur maximale des coûts du plan de la requête , sa valeur minimale , sa valeur moyenne et sa valeur médiane.

- **execution_time** : type *float64*, un réel qui représente le temps de réponse de la requête.
- **experience** : type *object* , une chaine de caractère qui sert à attribuer un label qui représente le cadre expérimental dans laquelle les mesures ont été collectées
- **stat_1** et **stat_2** : type *float64* , cardinalité des ensembles de données interrogés.
- **nb_joins** et **nb_agg**: type *int64*, nombre de jointures et nombre d'agrégations dans la requête.
- **filter** : type *int64* , une variable binaire qui correspond à l'encodage du plan logique de la requête : 1 représente la présence de l'opérateur « *filter* » et 0 son absence.
- **join** : type *int64*, une variable binaire qui correspond à l'encodage du plan logique de la requête : 1 représente la présence de l'opérateur « *join*» et 0 son absence.
- **project** : type *int64*, une variable binaire qui correspond à l'encodage du plan logique de la requête : 1 représente la présence de l'opérateur « *project*» et 0 son absence.
- **scan** : type *int64*, une variable binaire qui correspond à l'encodage du plan logique de la requête: 1 représente la présence de l'opérateur « *scan*» et 0 son absence.
- **AllNodesScan** : type *int64*, une variable binaire qui correspond à l'encodage du plan physique de la requête : 1 représente la présence de l'opérateur « *AllNodesScan* » et 0 son absence.
- **ExpandAll** : type *int64*, une variable binaire qui correspond à l'encodage du plan physique de la requête : 1 représente la présence de l'opérateur « *ExpandAl l*» et 0 son absence.
- **Filter** : type *int64*, une variable binaire qui correspond à l'encodage du plan physique de la requête : 1 représente la présence de l'opérateur « *Filter* » et 0 son absence.
- **NodeByLabelScan** : type *int64*, une variable binaire qui correspond à l'encodage du plan physique de la requête : 1 représente la présence de l'opérateur « *NodeByLabelScan* » et 0 son absence.
- **ProduceResults** : type *int64*, une variable binaire qui correspond à l'encodage du plan physique de la requête : 1 représente la présence de l'opérateur « *ProduceResults* » et 0 son absence.
- **Projection** : type *int64*, une variable binaire qui correspond à l'encodage du plan physique de la requête : 1 représente la présence de l'opérateur « *Projection* » et 0 son absence.
- **Results** : type *int64*, une variable binaire qui correspond à l'encodage du plan physique de la requête : 1 représente la présence de l'opérateur « *Results* » et 0 son absence.

Le nombre d'enregistrements total de cette expérience est 30 et le nombre d'attributs est 20. Le tableau 5.9 présente les résultats qu'on a trouvés au cours de cette expérience :

| Algorithme | Paramètres | Résultat |
|---|---|---|
| SVR | _ | MAE = 0.16s<br>Coefficient de détermination (r2_score) = -0,5 |
| Random forest regression | _ | MAE = 0.045 s<br>Coefficient de détermination (r2_score) = 0,97 |

*Tableau 5.9* Résultats de l'apprentissage du temps de réponse d'un workloads exécuté dans un magasin de données orienté graph : Neo4J

Vu le nombre réduit d'enregistrements nous n'avons pas exploré les algorithmes d'apprentissage basés sur les réseaux de neurones. D'après les métriques d'évaluation des résultats nous sélectionnons l'algorithme « *random forest regression* » [LW01]. En effet, en utilisant métrique R2 en observe bien que cet algorithme produit un modèle de prédiction ayant une précision égale à 97%.

Nous répétons l'expérience en utilisant les variables déterminées dans l'étape de sélection des variables et en générant plus de mesures de performance.

- **agg_attributes_size**, **scan_attributes_size** : attribut de type *int64* qui représente le nombre des attributs référencés par l'opérateur d'agrégation / de scan
- **agg_rows** / **proj_rows** : attributs de type *int64* qui représentent les cardinalités des opérateurs d'agrégation / de projection.
- **agg_width** / **proj_width** : attributs de type *int64* qui représentent la taille des attributs référencés par les opérateurs d'agrégation / de projection. Cette taille est calculée en multipliant le nombre d'attributs de chaque type et la taille en octets des types. La taille de référence des types sont stockés sous forme de métadonnées.
- **all_label_nodes** / **avg_label_nodes** : attributs de type *int64* qui correspondent aux nombres totaux / moyen des nœuds appartenant aux labels référencés dans la requête.
- **all_label_rels** : type *int64* , cet attribut représente le nombre total de relations.
- **all_label_rels_between_nodes** : type *int64* , cet attribut représente le nombre de relations entre les nœuds référencés dans la requête.
- **all_nodes** : type *int64* , cet attribut représente le nombre total de nœuds.
- **attributes_size** : type *int64* , taille des attributs référencés dans la requête.
- **execution_time** : type *float* , le temps de l'exécution de la requête.
- **is_directed** : type *booléen*, la projection du chemin du graphe suit le sens des liens définis ou non.
- **nodes_labeled** : type *int64*, le nombre de nœuds ayant un label
- **numberOfGroups** : type *int64*, cardinalité de l'opérateur d'agrégation
- **cardinality** : type *int64*, cardinalité de la requête
- **project_ratio** : type *float*, taux des attributs projetés par rapport à ceux qui ont été balayés par l'opérateur scan.
- **AllNodesScan**, **EagerAggregation**, **Expand**(**All**), **Filter**, **NodeByLabelScan**, **ProduceResults**, **Projection** : type *int64*, variables binaires qui correspondent à l'encodage du plan physique de la requête

En utilisant comme repère de sélection des variables le coefficient de corrélation avec la variable à prédire, on détermine les variables suivantes pour le problème de régression : (**all_label_nodes**, **attributes_size**, **avg_label_nodes**, **execution_time**, **numberOfGroups**, **scan_attributes_size**, **cardinality**, **project_ratio**, **EagerAggregation**, **NodeByLabelScan**, **Projection**)

L'expérience précédente nous a montré que l'algorithme le plus adapté à notre problème est la régression en utilisant les forêts aléatoires. On répète l'expérience en utilisant les caractéristiques

sélectionnées au lieu des coûts calculés par l'optimiseur de requête et on remarque une amélioration dans la performance de la prédiction. Le nombre d'enregistrements total de cette expérience est : 787. Le tableau 5.10 présente les résultats qu'on a trouvés au cours de cette expérience :

| Algorithme | Paramètres | Résultat |
|---|---|---|
| Random forest regression | _ | MAE = 1.27 s Coefficient de détermination (r2_score) = 0, .92 |
| Random forest regression | Grid search param_grid={ 'max_depth': range(3,7), 'n_estimators': (10, 50, 100, 1000) }, cv=5, scoring='neg_mean_squared_error' | MAE = 2.2 s Coefficient de détermination (r2_score) = 0.92 |

*Tableau 5.10* Résultats de l'expérience numéro 2 d'apprentissage du temps de réponse d'un workloads exécuté dans un magasin de données orienté graph : Neo4J

- Résultats de l'apprentissage du temps de réponse de l'exécution dans « Apache Phoenix » :

Dans cette expérience nous avons exécuté un workload sur des données stockées dans Hbase en utilisant le système Apache Phoenix. Les mesures collectées sont égales à 130 enregistrements qui ont comme attributs :

- **execution_time**, **experience**, **stat_1**, **stat_2**, **nb_joins**, **filter**, **join**, **project**, **scan** et **nb_agg** : des attributs qui ont le même type et la même sémantique des attributs qui portent le même nom dans l'expérience précédente.
- **sort**, **outer_join**, **left_join**, **aggregate** : type *int64*, variables binaires qui correspondent à l'encodage du plan logique de la requête
- **filter_ selectivity** : attribut de type *réel* qui représente la valeur de sélectivité du filtre.
- **IO_cost** : attribut de type *réel* qui représente le coût d'entrée/sortie de la requête.
- **SERVER_AGGREGATE_INTO_SINGLE_ROW, MERGE_SORT, PARALLEL_1-WAY_FULL_SCAN, PARALLEL_1-WAY_ROUND_ROBIN_FULL_SCAN, PARALLEL_1-WAY_ROUND_ROBIN_FULL_SCAN, PARALLEL_1-WAY_ROUND_ROBIN_RANGE_SCAN_OVER_INDEX, PARALLEL_3-WAY_ROUND_ROBIN_FULL_SCAN, PARALLEL_INNER-JOIN, PARALLEL_INNER-JOIN_TABLE, PARALLEL_INNER-JOIN_TABLE_0, PARALLEL_LEFT-JOIN_TABLE_0, SERVER_AGGREGATE_INTO_DISTINCT_ROWS, SERVER_AGGREGATE_INTO_SINGLE_ROW, SERVER_FILTER, SERVER_FILTER_BY_FIRST_KEY_ONLY :** type *int64*, variables binaires qui correspondent à l'encodage du plan physique de la requête
- **aggregation_group** : un *entier* qui représente la cardinalité du résultat de l'opérateur d'agrégation (le nombre de groupes construit)
- **filter_predicate_encoded** : type *int64*, variable binaire qui correspond à l'encodage du prédicat de filtrage.
- **count**, **groupby**, **sum** : type *int64*, variables binaires qui correspondent à l'encodage du type d'agrégation.

Le nombre d'attributs total est 36. Le tableau 5.11 présente les résultats qu'on a trouvés au cours de cette expérience :

| Algorithme | Paramètres | Résultat |
|---|---|---|
| Random forest regression | _ | MAE = $1.04 * 10^{-5}$ s<br>Coefficient de détermination (r2_score) = 0.74 |
| Lasso | _ | MAE = $2* 10^{-5}$s<br>Coefficient de détermination (r2_score) = 0.66 |
| gradient boosting | 'n_estimators': 500,<br>'max_depth': 4,<br>'min_samples_split': 5,<br>'learning_rate': 0.01,<br>'loss': 'ls' | MAE = $9* 10^{-6}$s<br>Coefficient de détermination (r2_score) = 0.82 |
| Xgboost | 'colsample_bytree': 0.7, 'silent': 1, 'learning_rate': 0.03, 'nthread': 4, 'min_child_weight': 4, 'n_estimators': 500, 'subsample': 0.7, 'objective': 'reg:linear', 'max_depth': 6 | MAE = $1* 10^{-5}$s<br>Coefficient de détermination (r2_score) = 0, 71 |

*Tableau 5.11* Résultats de l'apprentissage du temps de réponse d'un workloads exécuté dans un magasin de données orienté famille de colonnes : Apache HBase

- Résultats de l'apprentissage du temps de réponse de l'exécution dans « Apache Spark» :

Dans une première expérience nous avons exécuté un workload sur des données stockées dans un système de fichier en utilisant le système Apache Spark. Les mesures ont 15558 enregistrements et on a sélectionné 13 attributs. Le tableau 5.12 présente les résultats qu'on a trouvés au cours de cette expérience :

| Algorithme | Paramètres | Résultat |
|---|---|---|
| Réseau de neurones : MLP | - Type du modèle : séquentiel<br>- Nombre de couches : 3 (une première couche ayant la fonction d'activation relu, 4 couches cachés ayant la fonction d'activation relu et une dernière couche composée d'une seule neurone)<br>- Nombre de neurones : 8,16, 32<br>- Drop out : 0.1, 0.2<br>- Optimiseur : 'adam', 'sgd'<br>Les paramètres du modèle sélectionné sont : 16 neurones par couche, 0,1 pour le paramètre drop out et le choix de l'optimiseur est Adam. | RMSE = 0.01 s<br>MAE = 0.008 s<br><br>Coefficient de détermination (r2_score) = 0.9999 |
| Gradient boosting | | MAE =0.1 s |

| | | Coefficient de détermination (r2_score) = 0.9322<br>RMSE = 303528199.5364 ns |
|---|---|---|
| RFR | _ | MAE =0.08s<br>Coefficient de détermination (r2_score) = 0,56 |
| RFR + grid search | -'max_depth': range(3,7)<br>- 'n_estimators': (10, 50, 100, 1000) | MAE = 0.08 s<br>Coefficient de détermination (r2_score) = 0,62 |
| Régression linéaire | | MAE = 0.08 s<br>Coefficient de détermination (r2_score) = 0,57 |

*Tableau 5.12* Résultats de l'apprentissage du temps de réponse d'un workloads exécuté dans un moteur de traitements de données distribuées : Apache Spark

La figure 5.15 compare le temps de réponse prédit (en orange) et le temps de réponse mesuré du premier modèle prédictif. On remarque que les deux courbes sont presque identiques et que les erreurs de prédictions concernent surtout les mesures sur des ensembles de données de petites tailles.

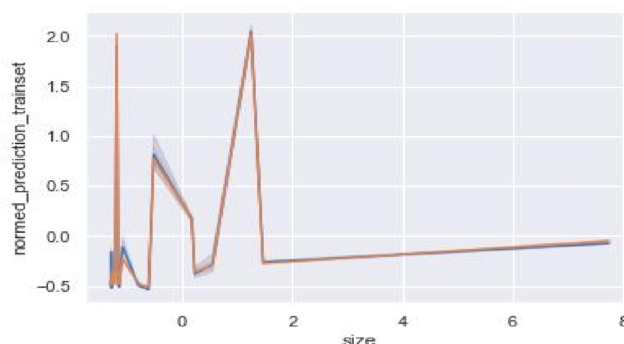



*Figure 5.15* Comparaison du temps de réponse prédit et mesuré

La méthode utilisée pour déterminer les paramètres du modèle sont la grille de recherche dans l'algorithme de régression en utilisant les forêts aléatoires. Dans le premier réseau de neurones nous avons itéré l'apprentissage et fait varier les hyper-paramètres.

### 5.2.5 Validation expérimentale de la recommandation

**<u>But de l'expérience :</u> validation sur un exemple le module de recommandation du placement de données.**

Dans cette section, nous présentons un déroulement de l'exécution de l'algorithme de recommandation sur le cas d'utilisation industriel étudié dans cette thèse. Dans cet exemple, nous avons utilisé un ensemble de simplifications ((i) on ignore le temps d'insertion des données intermédiaire puisque cette opération se répète et n'influence donc pas le résultat, (ii) on prend les système suivant comme candidats de placement : Apache Spark , MongoDB , PostgreSQL et Apache Hive), (iii) le transfert des données entre les serveurs est fait en utilisant la technologie WiFi a qui a un taux de transfert égale à 54 Mbit/s et finalement (iv) on ne considère pas le cas d'interrogation de données stockés dans deux magasins de données distinct.) .

Dans le tableau des résultats (5.13) nous présentons un extrait de ce qu'on a obtenu expérimentalement. Vu que le nombre de solutions retournés par la méthode exhaustive est très grand (256 solutions après la simplification).

| Etape | Emplacement des ensemble de données | Performance de la requête 1 | Performance de la requête 2 | Performance de la requête 3 | Performance de la requête 4 |
|---|---|---|---|---|---|
| Initiale | Conso_inf36 dans hdfs , Conso_sup36 dans hdfs, données intermédiaires dans hdfs. Hdfs est configuré sur un seul serveur | Requête : Q1 Exécution = moteur d'exécution : Apache Hive en utilisant Tez , système de stockage : HDFS Résultat = 34 s. | Requête : Q2 Exécution = moteur d'exécution : Apache Hive en utilisant Tez , système de stockage : HDFS Résultat = 52s. | Requête : Q3 Exécution = moteur d'exécution : Apache Hive en utilisant Tez , système de stockage : HDFS Résultat = 1min 44 s. | Requête : Q4 Exécution = moteur d'exécution : Apache Hive en utilisant Tez , système de stockage : HDFS Résultat = échec d'exécution, condition de faisabilité de l'exécution des opérateurs sur la taille des données n'est pas vérifiée |
| Solution 1 | Conso_inf36 dans PostgreSQL , Conso_sup36 dans PostgreSQL, données intermédiaires dans PostgreSQL. L'installation de PostgreSQL est centralisée | Requête : Q1 Exécution = moteur d'exécution : PostgreSQL , système de stockage : PostgreSQL Résultat = 4 min. | Requête : Q2 Exécution = moteur d'exécution : PostgreSQL , système de stockage : PostgreSQL Résultat = échec d'exécution, condition de faisabilité de l'exécution des opérateurs sur la taille des données n'est pas vérifiée | _ | _ |
| Solution 2 | Conso_inf36 dans PostgreSQL , Conso_sup36 dans PostgreSQL, avec déplacement des données intermédiaires. L'installation de PostgreSQL est centralisée | Requête : Q1 Exécution = moteur d'exécution : PostgreSQL , système de stockage : PostgreSQL Résultat = 4 min. Transformation 1= déplacement vers Hive | Requête : Q2 Exécution = moteur d'exécution : Apache Hive en utilisant Tez , système de stockage : HDFS Résultat = 52s. Transformation 2= déplacement vers PostgreSQL | Requête : Q3 Exécution = moteur d'exécution : PostgreSQL , système de stockage : PostgreSQL Résultat = 117,52s. | Requête : Q4 Exécution = moteur d'exécution : PostgreSQL , système de stockage : PostgreSQL Résultat = échec d'exécution, condition de faisabilité de l'exécution des opérateurs sur la taille des données n'est pas vérifiée |
| Solution 3 | Conso_inf36 dans MongoDB , Conso_sup36 dans MongoDB, données intermédiaires dans MongoDB. | Requête : Q1 Exécution = moteur d'exécution : MongoDB, système de stockage : MongoDB Résultat = échec | _ | _ | _ |

| | | | | | |
|---|---|---|---|---|---|
| | L'installation de MongoDB est centralisée (version 3.4) | d'exécution, condition de faisabilité de l'exécution de l'opérateur union n'est pas vérifiée. | | | |
| Solution 4 | Conso_inf36 dans hdfs , Conso_sup36 dans hdfs, données intermédiaires dans MongoDB. L'installation de MongoDB est centralisée (version 3.4) | Requête : Q1 Exécution = moteur d'exécution : Apache Hive en utilisant Tez , système de stockage : HDFS Résultat = 34 s. Transformation 1= déplacement vers MongoDB | Requête : Q2 Exécution = moteur d'exécution : MongoDB, système de stockage : MongoDB Résultat = échec d'exécution, condition de faisabilité de l'exécution de l'opérateur join n'est pas vérifiée | _ | _ |
| Solution 5 | Conso_inf36 dans hdfs , Conso_sup36 dans hdfs, données intermédiaires dans MongoDB et HDFS. L'installation de MongoDB est centralisée (version 3.4) | Requête : Q1 Exécution = moteur d'exécution : Apache Hive en utilisant Tez , système de stockage : HDFS Résultat = 34 s. | Requête : Q2 Exécution = moteur d'exécution : Apache Hive en utilisant Tez , système de stockage : HDFS Résultat = 52s. Transformation 1= déplacement vers MongoDB | Requête : Q3 Exécution = moteur d'exécution : MongoDB, système de stockage : MongoDB Réécriture de requête : « db.hni_agg_inf_sup_1.aggregate( [//first stage { $group: { _id: { cd_prof: "$cd_prof" ,dh_cour_5mn: "$dh_cour_5mn" ,plag_puis: "$plag_puis",sect_act: "$sect_act"}, sum_tot_nrj_s : { $sum: "$tot_nrj_s"} } } , // Second Stage { $group: | _ |

| | | | | { _id: { cd_prof: "$cd_prof" ,dh_cour_5mn: "$dh_cour_5mn" ,plag_puis: "$plag_puis",sect_act: "$sect_act"}, puis_moy : { $avg: "$sum_tot_nrj_s "} } } ] , { allowDiskUse: true }) » Résultat = échec d'exécution, condition de faisabilité de l'exécution des opérateurs sur la taille des données n'est pas vérifiée | |
|---|---|---|---|---|---|
| Solution 6 | Conso_inf36 dans un système de fichier, Conso_sup36 dans un système de fichier, données intermédiaires dans un système de fichier. L'installation d'Apache Spark est centralisée (version 2.7) | Requête : Q1 Exécution = moteur d'exécution : Apache Spark, système de stockage : un système de fichier Résultat = 0, 218 s. | Requête : Q2 Exécution = moteur d'exécution : Apache Spark, système de stockage : un système de fichier Résultat = 0, 335s. | Requête : Q3 Exécution = moteur d'exécution : Apache Spark, système de stockage : un système de fichier Résultat = 22,95s. | Requête : Q4 Exécution = moteur d'exécution : Apache Spark, système de stockage : un système de fichier Résultat = 51,38s. |

*Tableau 5.13* Résultat du déroulement de l'algorithme de recommandation sur un exemple simple

La solution optimale de placement de données est la solution 6, elle consiste à placer les données dans un système de fichier et les interroger avec Apache Spark. Le temps de réponse du workload en utilisant cette solution est égale à 74.88s.

Dans la suite du classement on trouve en tete les deux solutions : [Q1 in 'hdfs hive', Q2 in 'FS spark', Q3 in 'FS spark', Q4 in 'FS spark'] et [Q1 in 'hdfs hive', Q2 in 'hdfs hive', Q3 in 'FS spark', Q4 in 'FS spark'] qui sont caractérisées respectivement par les temps de réponses : 157.2s et 210s.

## 5.3 Architecture logicielle détaillée

Dans ce qui suit, nous présentons l'architecture logicielle du prototype développé. Dans cet objectif, nous utilisons les diagrammes de classe de la spécification UML. Nous commençons par la hiérarchie

en paquets des classes du prototype, ensuite nous présentons le diagramme de classe pour chaque paquet.

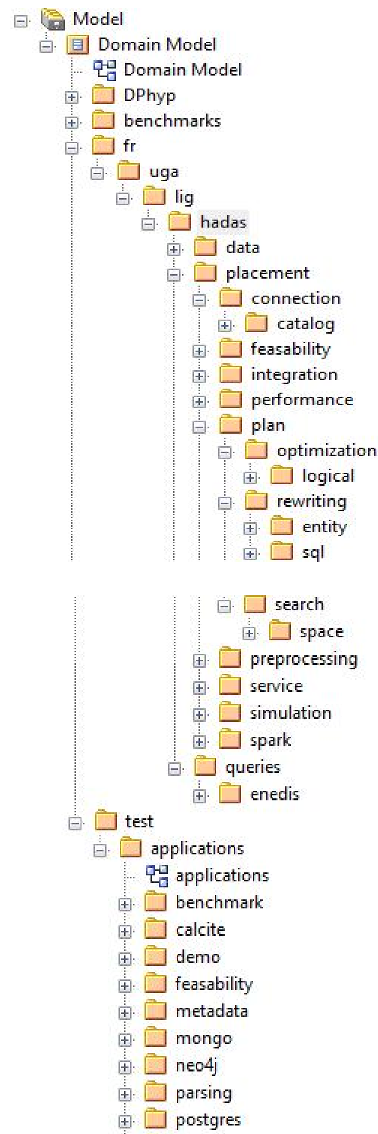



*Figure 5.16* Hiérarchie du prototype de DWS

Le prototype développé est composé de deux grandes parties : la partie principale (paquet *fr.uga.lig.hadas*) et la partie tests (paquet *test.applications*). La partie principale du prototype comporte à son tour des classes appartenant au modèle du domaine et des classes service.

#### *5.3.1.1 Modèle du domaine*

Les classes du domaine dans notre application sont représentées dans les parquets : *fr.uga.lig.hadas.placement.connection.catalog* et *fr.uga.lig.hadas.placement.feasability*.

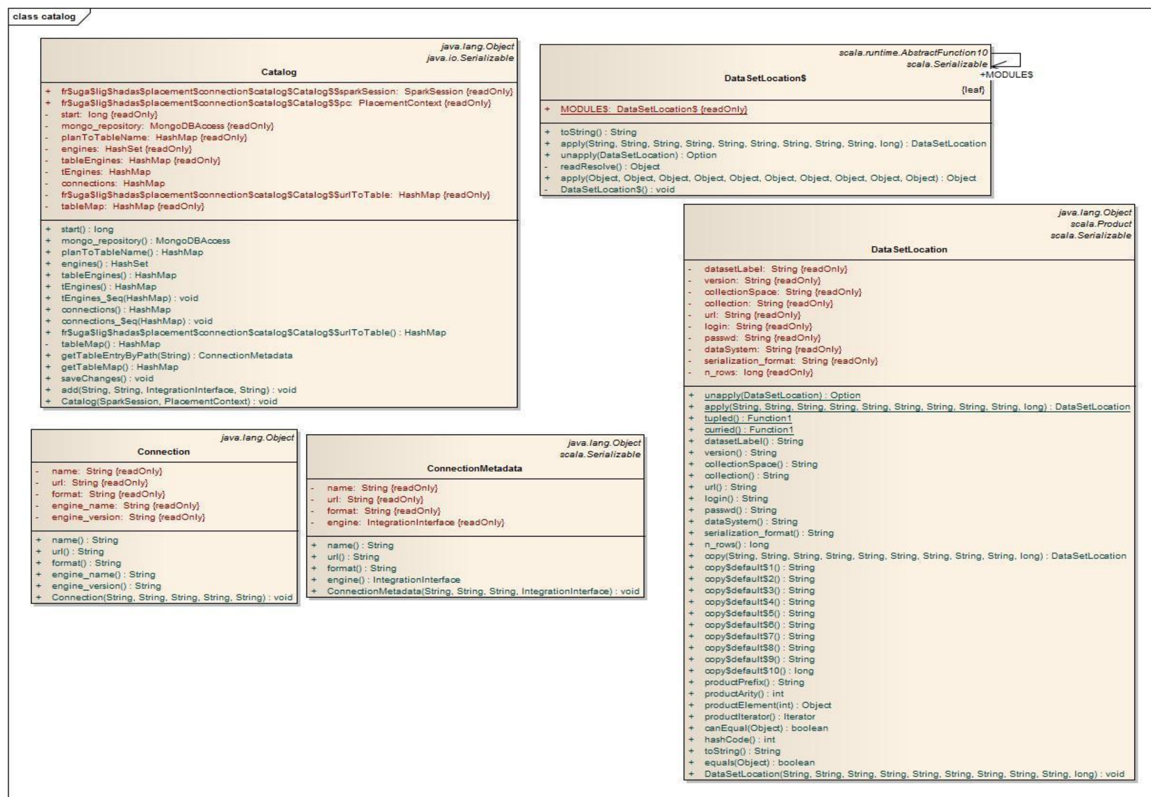



*Figure 5.17* Diagramme de classes du paquet : "fr.uga.lig.hadas.placement.connection.catalog"

La figure 5.17 présente le diagramme de classe du premier paquet. Il est composé de quatre classes qui permettent de créer le catalogue des ensembles des données se trouvant dans le lac de données, qui sert à rendre accessibles ces derniers au service de recommandation de placement de données.

- Classe *Catalog* : une classe qui représente un modèle pour un catalogue des ensembles des données dans l'écosystème.
- Classes *Connection*, *ConnectionMetadata* : deux classes qui permettent à l'application de se connecter aux différents magasins de données. La première est une classe de connexion à la base de métadonnées, et la deuxième permet de représenter les métadonnées de ce type d'entité dans l'application
- Classe *DatasetLocation* : cette classe permet de localiser un ensemble de données dans l'écosystème. Les objets de cette classe contiennent une référence sur le magasin de données qui gère les données.

Quant aux classes du deuxième paquet du modèle du domaine sont représentées dans le diagramme de la figure 5.18.

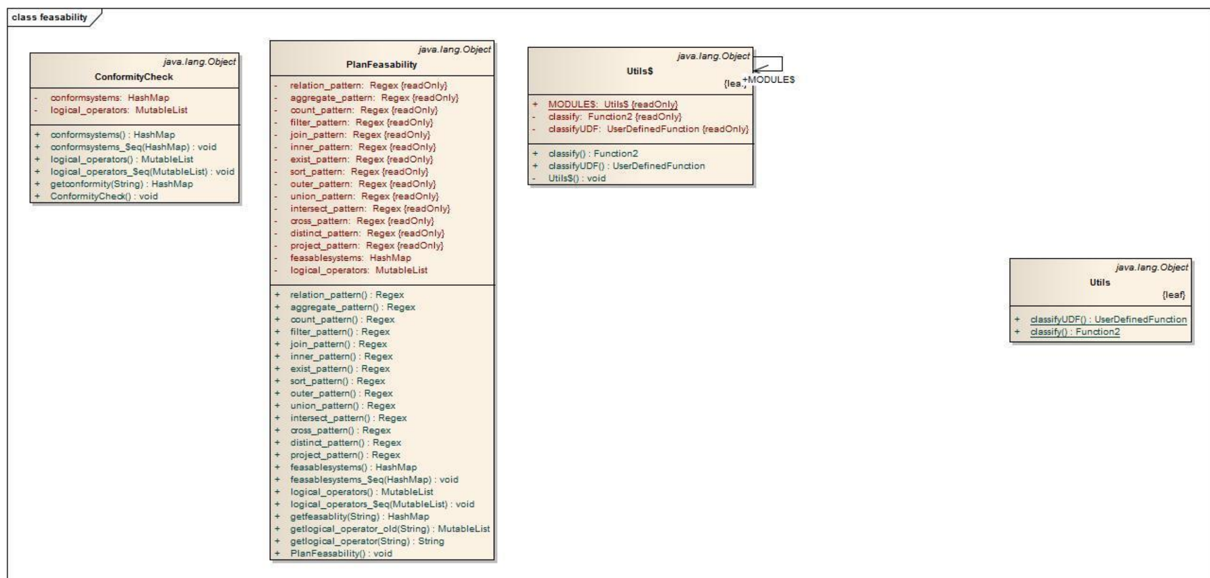



*Figure 5.18* Diagramme de classes du paquet : "fr.uga.lig.hadas.placement.feasability"

La classe *ConformityCheck* permet d'associer à chaque ensemble de données une restriction sur l'utilisation des systèmes de stockage. Elle est composée de deux attributs : *conform_systems* de type liste de chaîne de caractères, et *dataset* de type chaîne de caractères. La classe *PlanFeasability* permet d'associer à chaque opérateur logique les systèmes qui les supportent. Elle est composée des attributs suivants :

- *feasablesystems* : un ensemble de chaînes de caractères qui représentent les systèmes qui vérifient le critère de faisabilité
- *logical_operators* : une liste de chaînes de caractères qui représentent les opérateurs logiques de la requête qu'on veut valider par le critère de faisabilité
- *relation_pattern*, *aggregate_pattern*, ... : des attributs de type chaîne de caractère qui servent à détecter la présence d'un opérateur logique en utilisant des expressions régulières

Le dernier paquet (*fr.uga.lig.hadas.placement.integration.model.postgres*) de cette catégorie regroupe un ensemble de classes qui permettent de modéliser les structures de données de PostgreSQL et qui seront utilisées dans la fonctionnalité d'injection de statistiques. Le diagramme de la figure 5.19 représente ces classes.

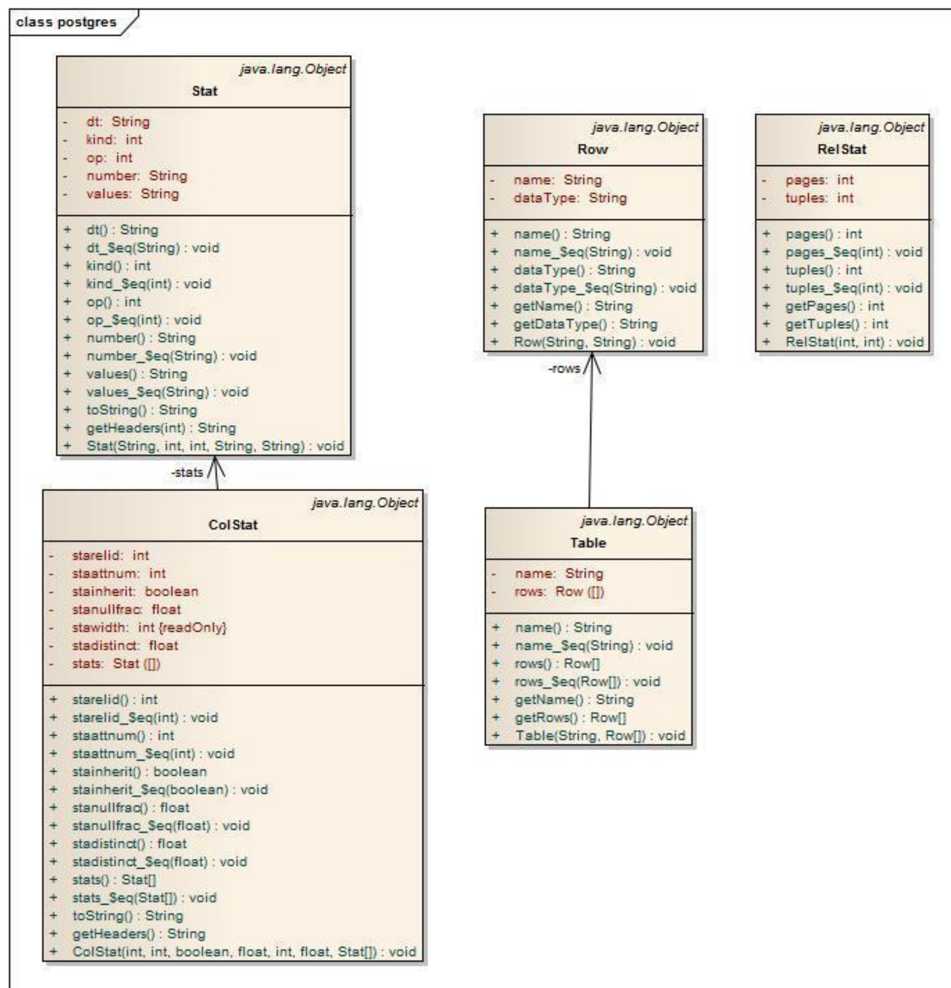



*Figure 5.19* Diagramme de classes du paquet : "fr.uga.lig.hadas.placement.integration.model.postgres "

La classe *Row* représente un attribut d'un enregistrement. Elle est composée de deux attributs : *name*, un attribut de type chaîne de caractères qui représente le nom d'un attribut, et *dataType*, un attribut de type chaîne de caractères qui représente le type de l'attribut. La classe *Table* représente une relation et elle a comme attributs : *name,* de type chaîne de caractères, et *rows,* de type tableau de *Row* et qui représente tous les attributs. Quant aux classes *Stat*, *ColStat* et *RowStat*, elles sont spécifiques aux statistiques de PostgreSQL.

Les deux premières classes contiennent les attributs de la relation de statistiques : *pg_statisti*c et la dernière classe contient deux attributs : *pages* qui représente le nombre de pages d'une relation, et *tuples* qui représente le nombre de tuples.

#### *5.3.1.2 Classes service*

Dans cette section nous présentons les diagrammes de classe de la couche métier de DWS. Nous commençons par la présentation du paquet *fr.uga.lig.hadas.integration*. Ce dernier est composé par l'interface *IntegrationInterface* qui présente un contrat à suivre pour ajouter un nouveau moteur de traitement à DWS et les classes qui l'implémente comme la classe *CalciteMongoDB*. Les méthodes les plus importantes de cette interface sont :

- *createCopy* : permet de créer une copie de l'ensemble de données dans le magasin de données cible.

- *supportsMove* : indique si le magasin de données supporte l'opération de déplacement vers le magasin de données indiqués en argument
- *getTransformationCost* : permet d'estimer le coût de la transformation d'un ensemble de données d'un magasin de données vers un autre
- *getRowsEstimation* : retourne le nombre d'enregistrements estimé du résultat intermédiaire. Prend en argument la requête exprimée en SQL.
- *getExecutionTime* : retourne le temps d'exécution d'une requête dans le magasin de données candidat
- *getDF* : retourne le résultat de la requête sous forme de *DataFrame*.
- *rename* : renomme un ensemble de données dans le magasin de données candidat
- *predict_exec* : prédit le temps d'exécution de la requête dans le magasin de données candidat

Les attributs de cette interface sont : (i) *store_label* et *engine_label*, attributs de type chaîne de caractères qui représentent respectivement les labels du magasin de données et du moteur de traitement. (ii) *exécution,* un attribut de type booléen qui indique si l'exécution du workload est supportée. (iii) Et finalement *transformer*, un attribut de type *PhysicalTransformation* qui permet l'exécution de transformations physiques comme l'indexation ou le partitionnement.

Cette interface a été implémentée pour les systèmes supportés dans ce prototype notamment : Neo4j, Apache Phoenix, pour manipuler les données dans Apache Hbase, PostgreSQL, Apache Spark et Apache Calcite, pour manipuler les données dans MongoDB. Dans ce qui suit, nous présentons une de ces implémentations.

*CalciteMongoDB* est une classe qui implémente l'interface décrite dans la section précédente. Elle étend sa définition par un ensemble d'attributs et de méthodes. Les attributs spécifiques à cette classe sont : (i) *MongoDBHost*, *MongoDBPort*, *MongoDBDB*, *MongoDBUsr* et *MongoDBPass*, qui sont les attributs permettant la connexion au serveur du magasin de données MongoDB, (ii) *calcitemodelURL*, un attribut de type chaîne de caractères qui représente l'URL permettant de se connecter au moteur d'interrogation de données Apache Calcite, (iii) *connection*, un attribut qui permet de se connecter à une connexion Apache Calcite, (iv) *ml_based*, un attribut de type booléen qui indique s'il existe un nœud de prédiction associé à ce magasin de données, et finalement *injector*, un attribut de type *MongoInjection* et qui permet l'injection de statistiques dans MongoDB en utilisant Apache Calcite. Quant aux méthodes spécifiques à cette classe elles sont listées ci-dessous :

- *estimate_rows* : une méthode qui permet d'extraire le nombre d'enregistrements estimé à partir du plan de requête. Elle accepte comme argument le plan sous forme textuelle.
- *estimate_cpu* : une méthode qui permet d'extraire l'estimation du coût de la requête à partir de son plan physique. Elle accepte comme argument le plan sous forme textuelle.
- *getCostMetrics* : une méthode qui permet de mesurer le temps d'exécution d'une requête et d'extraire son coût et le nombre d'enregistrements du résultat.

Le paquet suivant que nous présentons est celui qui regroupe les classes nécessaires pour la simulation de l'exécution du workload et de l'injection des statistiques. Le diagramme de classe de la figure 5.20 représente sa structure.

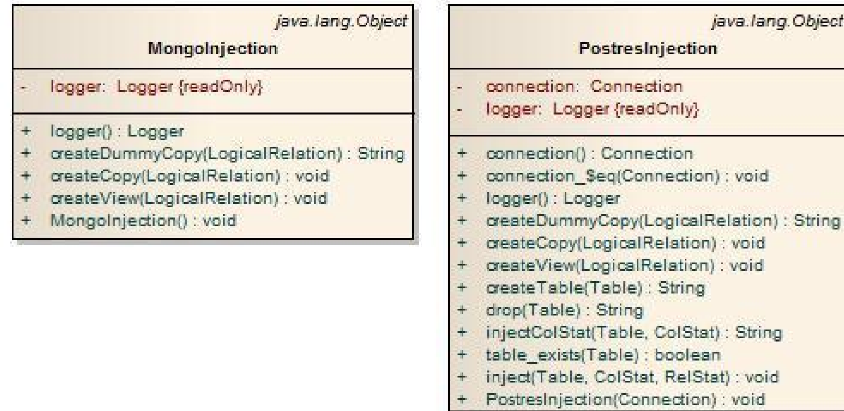



*Figure 5.20* Diagramme de classes du paquet : "fr.uga.lig.hadas.placement.simulation"

Ce paquet (*fr.uga.lig.hadas.placement.simulation*) est composé de deux classes : *MongoInjection* et *PostgresInjection*, vu que dans ce prototype on implémente cette fonctionnalité pour les deux systèmes MongoDB et PostreSQL. La première classe contient les méthodes suivantes :

- *createDummyCopy* : une méthode qui prend en argument un objet de type *LogicalRelation* (une classe dans l'API Spark SQL qui représente une relation et qui crée une copie de cette dernière contenant un seul enregistrement).
- *createCopy* : une méthode qui prend en argument un objet de type *LogicalRelation* et qui crée une copie intégrale des données de la relation référencée
- *createView* : une méthode qui prend en argument un objet de type *LogicalRelation* et qui crée une vue dans le magasin de données cible (dans ce cas MongoDB).

La classe *PostgresInjection* contient des méthodes similaires à celles présentées ci-dessus ainsi qu'un ensemble de méthodes spécifiques :

- *createTable*, *dropTable* et *table_exists* : des méthodes qui prennent en argument un objet de type *Table*. La première crée le schéma d'une relation dans PostgreSQL, la deuxième supprime la table référencée, et la dernière vérifie si une table existe.
- *InjectColStat* : prend en argument des objets de type *Table* et *ColStat* et injecte les statistiques de ces objets dans PostgreSQL
- *InjectStat* : prend en argument des objets de type *Table*, *RelStat* et *ColStat* et injecte les statistiques de ces objets dans PostgreSQL. Elle fait appel à la méthode précédente.

Les classes présentées précédemment sont utilisées dans le paquet *fr.uga.lig.hadas.placement.plan.search.space.exhaustive*. Dans ce prototype nous avons d'abord considéré l'optimisation du placement des données en utilisant la méthode de la programmation dynamique, ensuite la méthode exhaustive. Dans ce travail nous ne présentons pas les classes et les méthodes de la méthode d'optimisation basée sur la programmation dynamique vu que, comme expliqué au début de ce chapitre, nous avons choisi d'explorer exhaustivement toutes les solutions possibles de placement, et on laisse l'optimisation du temps d'exécution de l'algorithme de placement de données en perspective. Dans ce travail on a favorisé la qualité du résultat sur l'efficacité du traitement. Une autre partie de ce paquet concerne des classes spécifiques à l'API de Spark qui permettent de lancer l'algorithme au-dessus de ce moteur de traitement. Le diagramme 5.21 illustre les classes de ce paquet et leurs relations.

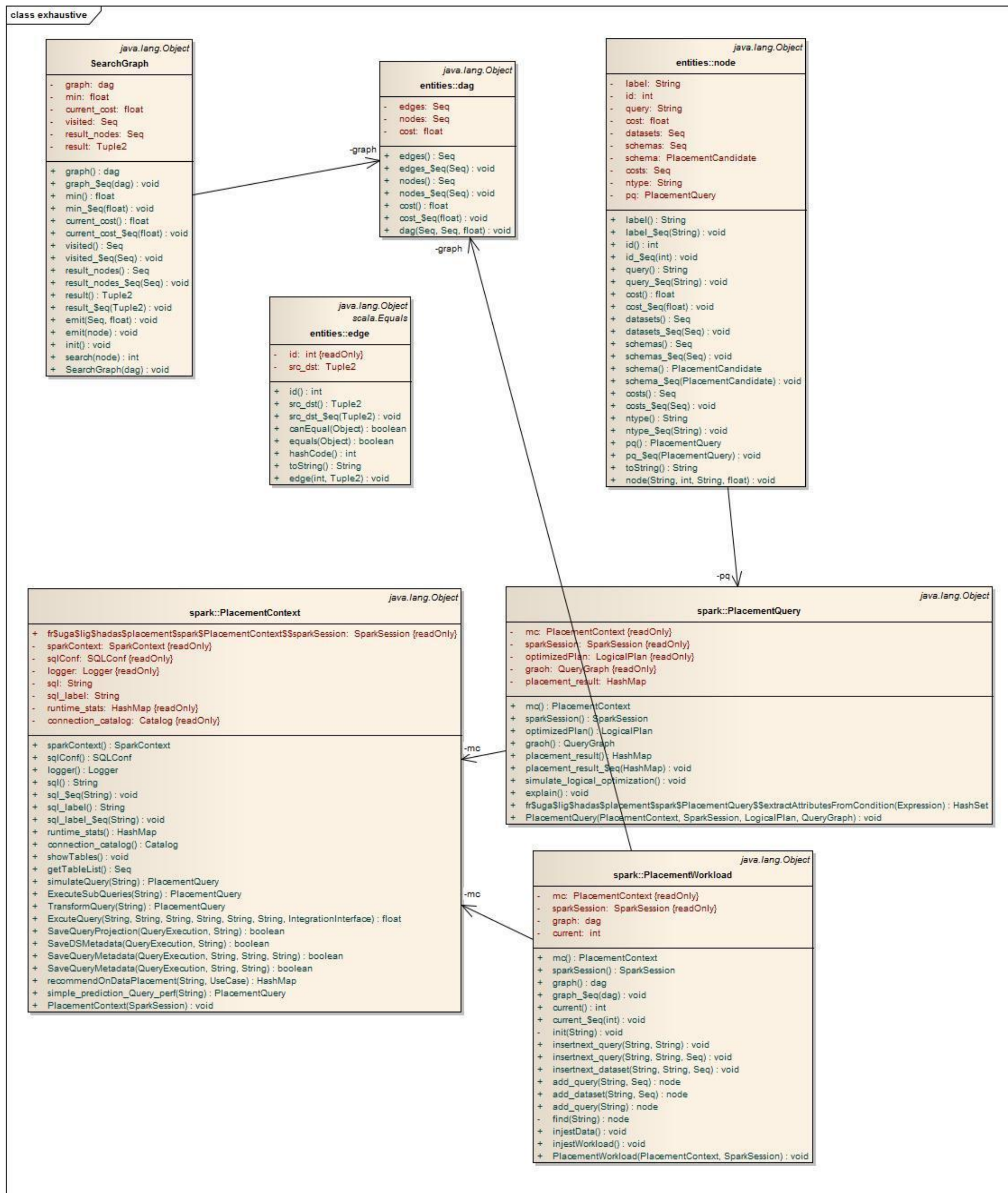



*Figure 5.21* Diagramme de classes du paquet : "fr.uga.lig.hadas.placement.plan.search.space.exhaustive"

En s'inspirant du moteur de traitement de données Apache Spark et du travail effectué dans le papier MuSQLe [GPTK16], nous désignons le point d'entrée au moteur d'exécution de notre module de recommandation et le point d'accès aux ressources de calculs (la grappe de calcul Spark) par la classe *PlacementContext*. Cette classe est composée de plusieurs attributs :

- *sparkContext*, *sparkSession* et *sqlConf* : des attributs de type *SparkContext*, *SparkSession* et *SQLConf*. Ces attributs permettent l'accès à une grappe de calcul Spark et à exécuter les traitements sur cette ressource.

- *runtime_stats* : un attribut de type *Hashap* qui permet de stocker les statistiques qui résultent de l'exécution des requêtes.
- *connection_catalog* : un attribut de type *Catalog* et qui permet de localiser tous les ensembles de données de l'écosystème.

Quant aux méthodes de cette classe, elles sont définies ci-dessous :

- *showTables* et *getTableList* : deux méthodes qui permettent de lister les ensembles de données du catalogue. La première les affiche à la console et la deuxième retourne leurs références dans une liste
- *simulateQuery* : une méthode qui prend en argument une requête SQL et simule son exécution dans les magasins de données où son exécution est supportée.
- *executeSubQueries* : une méthode qui prend en argument une requête, la décompose et exécute ses sous requêtes
- *transformQuery* : une méthode qui transforme une requête complexe en un ensemble de sous requêtes sans les exécuter.
- *excuteQuery* : une méthode qui exécute une requête. Cette méthode est utile en cas d'absence d'un nœud de prédiction du temps de réponse.
- *saveQueryProjection*, *saveDSMetadata*, *saveQueryMetadata* : des méthodes qui stockent des métadonnées sur les requêtes et les ensembles de données.
- *recommendOnDataPlacement* : une méthode qui permet de recommander le placement de données pour un script composé de plusieurs requêtes.
- *simple_prediction_Query_perf* : une méthode qui prédit le temps d'exécution d'une requête SQL dans un magasin de données cible.

Dans notre application la dernière classe est utilisée pour optimiser le placement d'une requête ou d'un workload complet, d'où les liens d'agrégation avec les classes *PlacementQuery* et *PlacementWorkload*.

*PlacementWorkload* utilise comme structure de données un graph orienté acyclique, d'où son association avec la classe *Dag*. *PlacementQuery* utilise quant à elle une classe *QueryGraph* pour la structure de son plan d'opérateurs et des fonctionnalités sur ce plan. Cette classe est définie dans la figure 5.22.

*Figure 5.22* Structure de la classe QueryGraph

Les méthodes les plus importantes de cette classe sont :

- *extract_tables* : une méthode qui permet d'extraire toutes les références des ensembles de données dans le plan logique
- *generateSearchGraph* : génère l'espace de recherche à partir du plan logique initial
- *predict_exec_time* : prédit le temps de réponse en exécutant des appels aux micro services qui conviennent
- *simulate_cost_estimation* : permet de simuler l'exécution de la requête si cette fonctionnalité est supportée.

Ses attributs principaux sont dont :

- *plan* : le plan logique initial. Le type de l'attribut est *LogicalPlan* (une classe de l'API SparkSQL.)
- *search_graph* : le graph de recherche de plan optimal.
- *code_generator* : une référence à la classe *CodeGenerator* qui permet de transformer un plan logique en requête.

Notre démarche nécessite la transformation des données et la réécriture de la requête. La structure du composant de transformation est décrite dans le diagramme de la figure 5.23.

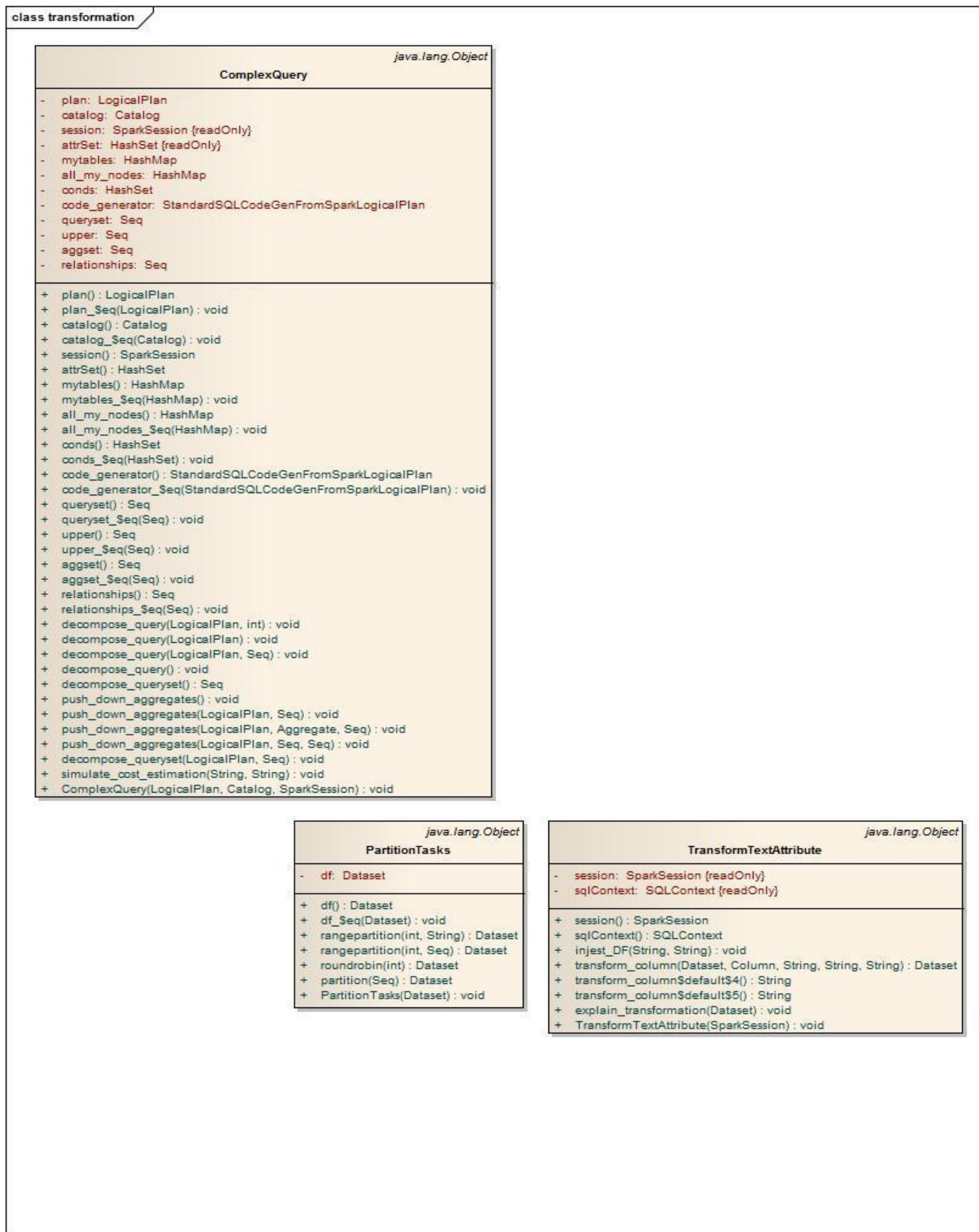



*Figure 5.23* Diagramme de classes qui implémentent la réécriture de requêtes et un exemple de transformation

Dans ce paquet deux types de transformations sont conçus : la transformation d'un ensemble de données et la transformation des requêtes en utilisant la technique de réécriture. Dans la première catégorie on trouve les classes décrites ci-dessous :

- La classe *TransformTextAttribute* : une classe qui utilise l'API de SparkSQL pour transformer un ensemble de données ayant des attributs complexes stockés sous forme textuelle, des attributs atomiques ou des agrégats. Les méthodes de cette classe sont : (i) *injest_DF*, qui permet de lire un ensemble de données depuis une source externe et de le persister en mémoire. (ii) *transform_column* prend en argument un ensemble de données matérialisée en mémoire de type *Dataset* (de l'API Spark), la référence de la colonne à transformer, l'expression régulière de la transformation et deux références pour les colonnes transformées. Cette méthode exécute la transformation requise. (iii) La troisième méthode de cette classe

est *explain_transformation*. Elle prend comme argument l'ensemble de données transformées et affiche les opérateurs de transformation utilisés.

- La classe *PartitionTasks* : une classe qui permet de partitionner les données horizontalement dans Spark. Elle est composée par trois méthodes : (i) *rangepartition* prend en argument la référence d'une colonne et le nombre de partitions, et partitionne un ensemble de données selon ces paramètres, (ii) *roundrobin* a comme argument des partitions. Elle divise l'ensemble de données en utilisant ce paramètre et l'algorithme *round robin*. (iii) *partition* prend en argument une liste de colonnes et exécute un algorithme de partitionnement par hachage sur un ensemble de données.
- La classe *ComplexQuery* : une classe qui transforme un plan de requête complexe en plans plus simples. Elle permet aussi d'utiliser des heuristiques pour transformer un plan de requête. Les méthodes de cette classe sont :
    - *decompose_query* :
        - ⇨ arguments : *logical* de type *LogicalPlan* et *position* de type entier
        - ⇨ Décompose un plan de requête en deux sous-plans selon la valeur de *position*.
    - *decompose_query*
        - ⇨ arguments : *logical* de type *LogicalPlan* et *projection* de type ensemble d'objets de type *Attribute*
        - ⇨ Elle permet de décomposer un plan initial en un ensemble de sous plans composé d'un seul opérateur
    - *decompose_queryset*
        - ⇨ Appel récursif à la méthode précédente
    - *push_down_aggregate*
        - ⇨ arguments : *logical* de type *LogicalPlan* et *projection* de type ensemble d'objets de type *Attribute*
        - ⇨ Une méthode qui transforme un plan logique en descendant les opérateurs d'agrégations vers les feuilles du plan
    - push_down_aggregates
        - ⇨ Appel récursif à la méthode précédente

Le diagramme de classe pour la réécriture de requête est dans le diagramme de la figure 5.24.

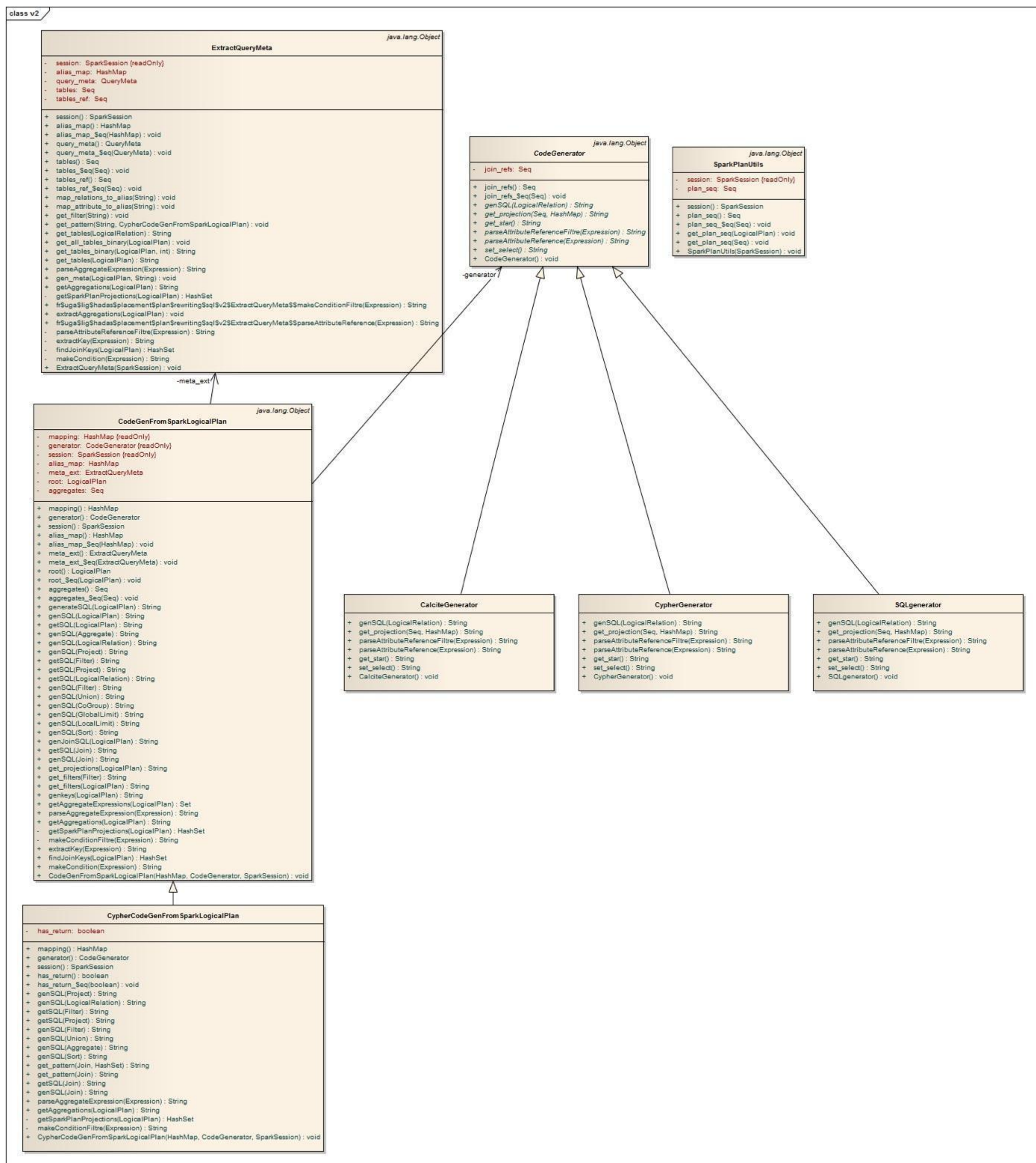



*Figure 5.24* Diagramme de classes qui implémentent la génération du code à partir d'un plan de requête

Dans ce prototype nous avons implémenté la réécriture de requêtes pour trois systèmes : Apache Calcite, Apache Spark et Neo4j. Les classes qui assurent cette fonctionnalité sont :

- *CodeGenerator* : classe abstraite de génération de requêtes SQL à partir d'un plan logique
- *CodeGenFromSparkLogicalPlan* : implémentation de la classe précédente pour les plans logiques en entrée qui suivent l'API de SparkSQL
- *CalciteGenerator* : implémentation de la classe précédente pour le dialecte du langage SQL supporté par Apache Calcite

- *CypherCodeGenFromSparkLogicalPlan* : implémentation de la classe précédente pour le dialecte du langage de requêtes Cypher de Neo4j et les plans logiques en entrée qui suivent l'API de SparkSQL

- *SQLgenerator* : pour le dialecte du langage SQL

D'autres classes sont utilisés dans ce paquet pour des fonctions utilitaires comme :

- *SparkPlanUtils* : pour présenter des fonctions utilitaires exploitées dans les étapes de transformation du plan de requête.

- *ExtractQueryMeta* : classe qui permet d'extraire des métadonnées des requêtes et de les stocker dans la base de métadonnées.

## 5.4 Résumé du chapitre

Nous avons présenté une architecture de recommandation sur le placement de données. Nous avons implémenté un prototype qui nous a permis de réaliser une preuve de concept. Ce prototype est basé sur un ensemble de modèles de prédictions construit à partir de mesures effectuées dans notre environnement expérimental. Ces derniers ont été effectués sur une installation centralisée des magasins de données vu que la distribution rend la prédiction du temps de réponse un problème difficile à cause du comportement aléatoire du temps de communication dans le réseau. Par contre, vu l'importance de la distribution de données dans ce domaine d'étude, on considère l'apprentissage automatique du temps de réponse sur des mesures effectuées sur des workloads distribués une perspective intéressante à ce travail.

Le résultat de placement est un ensemble de schéma de placements ordonnés par le temps de réponse. Il sera possible comme perspective d'ajouter un score qui impactera l'ordre du résultat. Ce score peut prendre en compte l'accès conçurent d'un ensemble de données par plusieurs applications.

Dans le contexte des workloads Big Data il est important aussi d'investiguer l'effet du passage à l'échelle sur la performance. Une piste de recherche intéressante sera d'étudier l'effet d'une augmentation de la taille des données sur le résultat de la recommandation. Une première étape consiste à faire varier les tailles des données et observer le temps de réponse ainsi que son comportement avec ce changement sur le paramètre taille des données.

Une dernière perspective à explorer pour ce travail de recherche sera d'explorer les paramètres des systèmes. On peut considérer des métriques comme : le type de CPU, la mémoire, le nombre de cœurs d'un processeur et la configuration d'un moteur de traitement qui peut être distribuée, locale, basée sur YARN, … En effet, nous avons travaillé principalement dans le cadre de ce projet sur les paramètres de données et de workload dans l'algorithme d'estimation de performance du placement et sur une partie des métadonnées systèmes « statiques ».

# Chapitre 6 Conclusion et perspectives

Dans ce chapitre, nous résumons notre travail, présentons et discutons nos contributions. Enfin, nous proposons quelques perspectives.

## 6.1 Résumé et critique

Au cours de cette thèse nous avons étudié le problème de recherche de solution optimale de placement de données. Ce dernier est un problème de recommandation basé sur des métadonnées collectées de l'écosystème et d'autres définies par les utilisateurs. Les métadonnées sont conçues sur trois niveaux de l'écosystème :

- Des métadonnées du niveau Systèmes qui caractérisent les propriétés statiques des systèmes par exemple leurs modèles de données, leurs opérateurs logiques et physiques et les modèles de distribution
- Des métadonnées sur les ensembles des données qui décrivent les identifiants des ensembles (e.g le chemin d'accès dans hdfs, les noms des ensembles de données...), leurs annotations, leurs statistiques descriptives (comme les valeurs médianes, min, max, la variance, covariance, moyenne), la liste des métadonnées sur leurs partitions (par exemple, le schéma de distribution des partitions qui peut être uniforme, ou bien qu'elle favorise certaines propriétés …) et sur leurs structures (notamment la liste des attributs, le nombre des enregistrements, la structure de données, le type de fichier) et finalement leurs métadonnées administratives et de provenance définis comme la version (de type «timestamps»), le propriétaire, la licence, la source des données, la date de création, la date de modification et la description.
- Les métadonnées sur les workloads correspondent aux informations sur les requêtes usuelles (disponibles en gardant un historique des requêtes) ainsi qu'aux métadonnées d'administration des jobs. On trouve ainsi : la description des patrons des requêtes ( «attribute oriented», «range oriented» …), la fréquence des requêtes («continuous stream», «days of the week», «hours» …), la géolocalisation des clients (les clients qui accèdent une partition particulière doivent être à proximité du serveur qui gère cette partition), le plan de la requête logique et physique et finalement des statistiques (particulièrement les mesures de «selectivity», «cardinality» et du «cost»).

Le module de recommandation prend en entrée la définition d'un workload structuré en tant que graph de requêtes, détermine la combinaison de systèmes qui vérifient trois critères que nous avons définis pour le problème de placement de données et les retourne à l'utilisateur ordonnés par le temps de réponse estimé du workload dans chaque système.

Le premier critère est la faisabilité d'une solution de placement de données. Ce critère permet d'inférer si un opérateur logique est supporté par l'api d'un système qui a été sélectionné comme système de placement candidat.

Le deuxième critère est la conformité. Il vérifie un ensemble de règles métiers qui permettent de valider une solution de placement par rapport au choix du propriétaire des données, du concepteur de l'architecture de l'écosystème ou bien du développeur d'une application.

Le dernier critère est la performance. On souhaite proposer à l'utilisateur uniquement des résultats performants qui minimisent le temps de réponse.

L'architecture qu'on propose est composée principalement d'un module d'optimisation, un module de transformation et d'écriture de requête et finalement d'un module d'estimation de temps de réponse

basé sur les techniques d'apprentissage automatique. Le dernier module présenté est composé de plusieurs nœuds de prédiction de temps de réponse qui utilise des modèles d'apprentissages construits à partir des expériences faites dans le cadre expérimental de notre environnement de recherche. Chaque nœud est propre à un système de stockage et de traitement de données et à une configuration du serveur (ou des serveurs) de déploiement du système. Un ensemble de perspectives peuvent être proposés pour améliorer ce travail.

## 6.2 Perspectives

En termes de perspective, on trouve important d'adapter notre proposition à d'autres environnements. Pour cette raison, on propose de remplacer les modules d'apprentissage actuel en d'autres qui utilisent des techniques dynamiques. Ces techniques permettent d'apprendre au fur et à mesure de l'exécution de la recommandation et de mettre à jour le modèle d'apprentissage afin d'adapter le résultat aux changements de l'environnement. Ainsi, nous proposons comme perspective l'exploration des techniques d'apprentissage automatique continu pour amélioration de ce module.

On appelle un modèle d'apprentissage automatique dynamique tout modèle d'apprentissage construit à partir de données acquises en continu par l'algorithme [Goo]. Cet algorithme fait l'entraînement en ligne et met à jour le modèle régulièrement. Un exemple d'utilisation de cette méthode est la prédiction de l'évolution des tendances de la finance. Les tendances pour ce type d'applications varient par an et il devient impossible de se baser sur les modèles de l'année précédente pour prédire l'évolution de l'année en cours. D'où l'importance d'utiliser les modèles dynamiques qui permettront d'adapter le modèle aux changements faits à travers les années. Dans notre contexte, ces modèles sont préconisés pour adapter un modèle construit à partir de données collectées dans un serveur ayant des caractéristiques particulières aux données qui seront produites par le serveur de production.

Une deuxième perspective concerne la transformation des données. Dans ce projet nous avons étudié la réécriture de requête et le changement du système de stockage, par contre nous ne nous sommes pas approfondi dans l'étude de la transformation du modèle de données et de l'influence du temps de déplacement d'un ensemble de donnée d'un système à un autre dans la performance totale du workload. La prochaine étape à faire sera de mesurer à l'aide de benchmark le temps que prendra une opération de chargement de données dans chaque système cible et ensuite utiliser la même technique expérimentale que nous avons proposé pour prédire le temps de chargement. Il faut aussi implémenter un module qui permet de transformer les statistiques qui permet de calculer toutes les statistiques nécessaires pour les résultats intermédiaires et les ensembles de données transformés. Ce calcul peut se baser sur les estimations des cardinalités du résultat et les statistiques des ensembles de données de départ.

Une troisième perspective peut être étudiée se rapporte à la sélection de variables de prédiction. En effet, dans notre approche on utilise exclusivement des métadonnées sur les ensembles de données et sur les workloads. Il est intéressant aussi de considérer des paramètres de configuration des systèmes comme le nombre de cœurs dans un processeur, le type de processeur et le système d'exploitation et la capacité du cpu utilisé.

Comme quatrième perspective, on peut considérer de dynamiser davantage l'algorithme de recommandation en remplaçant l'algorithme de recherche de solution optimale exhaustive par un algorithme qui utilise les techniques de programmation dynamique.

Finalement, une application intéressante à nos travaux dans un contexte de lac de données que nous pouvons considérer est l'auto adaptation du stockage. Un point de départ pour le développement de cet outil est l'algorithme de placement de données que nous avons proposé. Le changement à faire

pour obtenir ce module est d'utiliser un historique des requêtes collectées automatiquement de l'environnement et l'utiliser comme entrée à l'algorithme à la place du workload que l'utilisateur doit spécifier. L'auto-adaptation du stockage dans un datalake peut s'avérer très utile dans un monde ou la complexité des écosystèmes Big Data ne cessent d'augmenter. En exploitant les métadonnées sur les trois niveaux, l'écosystème peut gérer de manière plus rationnelle les ressources de calcul et de stockage, décider où stocker les données dans un environnement polystore, décider des transformations comme la politique de partitionnement de données et anticiper des prétraitements.

# - Annexe 1 : modèle de données relationnel

Le modèle relationnel est l'un des premiers modèles de données et reste le modèle de référence pour les systèmes de gestions de bases de données (DBMS). Il est basé sur l'algèbre relationnelle qui représente les ensembles de données sous forme de relations et offre un ensemble d'opérateurs de recherche de données qui permettent de calculer un nouveau sous ensemble de données (relation) à partir d'une ou plusieurs relations.

La notion de base dans ce modèle de données est la relation. La relation [Mak77] est définie mathématiquement comme un ensemble R de n-uplets $<d_0 \ldots d_n>$ de valeurs appartenant respectivement a n domaines $D_0..D_n$. Un domaine selon ce modèle est un ensemble de valeurs bien défini. Par exemple, le domaine des secteurs d'activités des clients industriels est défini comme suit : {« S1 : Agriculture », « S2 : Industrie », « S3 : Tertiaire », « S4 : Non affecté »}.

Les opérateurs de l'algèbre relationnelle peuvent être unaires ou binaires [Cod72] :

- Sélection de données (∂) : à partir d'une relation R de départ comprenant n tuples calcule un sous ensemble $R_1$ composé de i tuples avec $i<=n$. Tous les tuples de la relation $R_1$ doivent vérifier les prédicats de la sélection. Un prédicat est une condition booléenne exprimée en utilisant des opérateurs logiques et qui sert généralement à comparer les données avec une valeur bien déterminée.
- Projection (∏) : calcule à partir d'une relation R ayant p attributs un sous ensemble $R_1$ qui a $p_1$ attributs avec $p_1<=p$. Le nombre de tuples reste inchangé les attributs communs gardent les mêmes valeurs.
- Jointure (⊳⊲) : crée une relation $R_3$ à partir de deux autres relations $R_1$ et $R_2$. Cet opérateur concatène les tuples de $R_1$ et ceux de $R_2$ vérifiant une condition faisant intervenir des attributs de $R_1$ et de $R_2$ (la condition est souvent l'égalité d'un attribut de $R_1$ et d'un attribut de $R_2$, cet attribut commun est alors appelé la clé de la jointure).
- Opérateurs ensemblistes :
    - Union (⋃) : à partir de deux relations ayant les mêmes attributs, il calcule l'opération d'union ensembliste.
    - Différence (−) : à partir de deux relations ayant les mêmes attributs, il calcule l'opération de différence ensembliste.
    - Produit cartésien (×) : concatène tous les tuples de la première relation avec tous les tuples de la deuxième relation.

Les systèmes qui implémentent ce modèle théorique sont appelés des RDBMS acronyme pour Relational Database Management System et ils utilisent le langage SQL pour la manipulation des données. Ce langage introduit la notion de table, équivalente à la notion de relation, et de colonne qui correspond à la notion d'attribut. Il permet de spécifier des schémas qui sont composés d'un nombre défini au préalable de colonnes par table. Ces colonnes permettent de stocker des valeurs de types prédéfinis ou définis par l'utilisateur. Ces types peuvent être atomiques, comme le type entier, chaine de caractères, etc., ou structurés (type objet, collection, …). Toutefois, les relations normalisées doivent avoir des colonnes atomiques.

Le langage SQL est un langage déclaratif riche, complet pour l'interrogation, la définition, la manipulation et le contrôle des données relationnelles :

- Il est riche puisqu'il permet d'exprimer tout type de requêtes à partir de ses opérateurs de base et de nouveaux opérateurs introduits via des plugins spécialisés
- Il est complet à partir de la version SQL2 puisqu'il implémente l'algèbre relationnelle et qu'il a été standardisé
- Il est déclaratif : il permet à l'utilisateur d'exprimer les caractéristiques du résultat attendu dans décrire les étapes nécessaires pour le produire, le RDBMS ayant à charge de déterminer la meilleure façon de le calculer.

Il a été créé en 1970 et depuis plusieurs standards de ce langage ont vus le jour : SQL 2016 [MHK+18], SQL 2011, SQL 2008, SQL 2003, SQL3 (1999), SQL2 (1992) et SQL1 (1986). Ce langage est une implémentation de l'algèbre relationnelle dans les RDBMS. Le tableau de l'annexe 1 résume les différents standards existants et les différences entre eux.

## - Annexe 2 : comparaison des standards du langage SQL et son adaptation en Big Data

| Standard | Nouveauté | Base de données / moteur de traitement |
|---|---|---|
| SQL1 86 [Mel96] | - Opérateurs de descriptions et de manipulation de données de base. | - Ingres<br>- Oracle<br>- Sybase<br>- Informix |
| SQL2 92 [KC95] | - L'opérateur « union » dans la définition des vues<br>- Conversion des types caractères et numérique<br>- Utiliser l'opérateur * dans la projection<br>- Grouper les opérations<br>- Amélioration des clés primaires<br>- Les identifiant en minuscules<br>- Opération d'insertion<br>- Supporter plusieurs modules<br>- Définition de schéma<br>- Prédicat nul étendu<br>- Caractères nationaux<br>- Trait de soulignement<br>- Ordre des noms référentiels | - Apache calcite<br>- Oracle 8i<br>- DB2<br>- PostgreSQL<br>- SQL Server<br>- … |
| SQL3 99 | - curseurs<br>- nouveau type : Bit, Bit varying<br>- types définis par l'utilisateur<br>- le résultat (result set) peut être mis à jour | |

| | | |
|---|---|---|
| | - mise à jour du résultat en lots<br>- amélioration de la fonctionnalité de connexion et de transaction<br>- extensions objet, actives, OLAP, procédurales | |
| SQL:2003 [EMK+04] | - Nouveaux types : BIGINT, MULTISET, XML<br>- Nouveaux opérateurs pour ces nouveaux types comme : COLLECT, fusion<br>- Amélioration des routines invoqués par SQL<br>- Extension de l'expression « create table »<br>- Opérateur merge<br>- Objet pour les schémas : générateur de séquences<br>- Colonnes identité, colonne générée | |

*Tableau 7-1 Comparaison de quelques standards SQL*

Le tableau précédent présente une comparaison des différents standards. Le changement marquant dans ces standards est la conformité avec l'algèbre relationnelle dans la version SQL2 92 et le support du type objet dans la version SQL3 99.

*Remarque : Les dialectes du langage SQL en Big Data :*

L'adaptation des solutions NoSQL dans les projets industriels a été considérée comme un défi à cause de la complexité des APIs offerts et l'expertise en nouvelles technologies que celle-ci demande. Pour cette raison, en plus des solutions Big Data ayant un modèle de données relationnel, des systèmes de traitement de données ayant des interfaces et des langages d'interrogation de données proches du langage SQL (généralement dénommée comme langage de requête / API : « SQL like ») ont été proposées. Ces systèmes ont un moteur d'exécution de requête ayant une architecture similaire à celle des RDBMS. Mais ils imposent des limites sur certains opérateurs (par exemple, les versions anciennes de Spark SQL implémentent la spécification SQL 2011, mais elles ne permettent pas de définir l'opération Union all). Ces systèmes s'intègrent avec d'autre data stores et solutions NoSQL et permettent aux applications d'avoir une vue relationnelle sur des données qui sont stockés selon un modèle de données alternatif. Pour accéder et manipuler des données complexes (valeurs structurées …), ils ont proposé des variations de la syntaxe du langage SQL et des opérateurs de réécriture de

requête. Cette fonctionnalité est appelée dans quelques systèmes (Apache Calcite, par exemple) le SQL étendu et l'exemple de la figure suivante en illustre le principe.

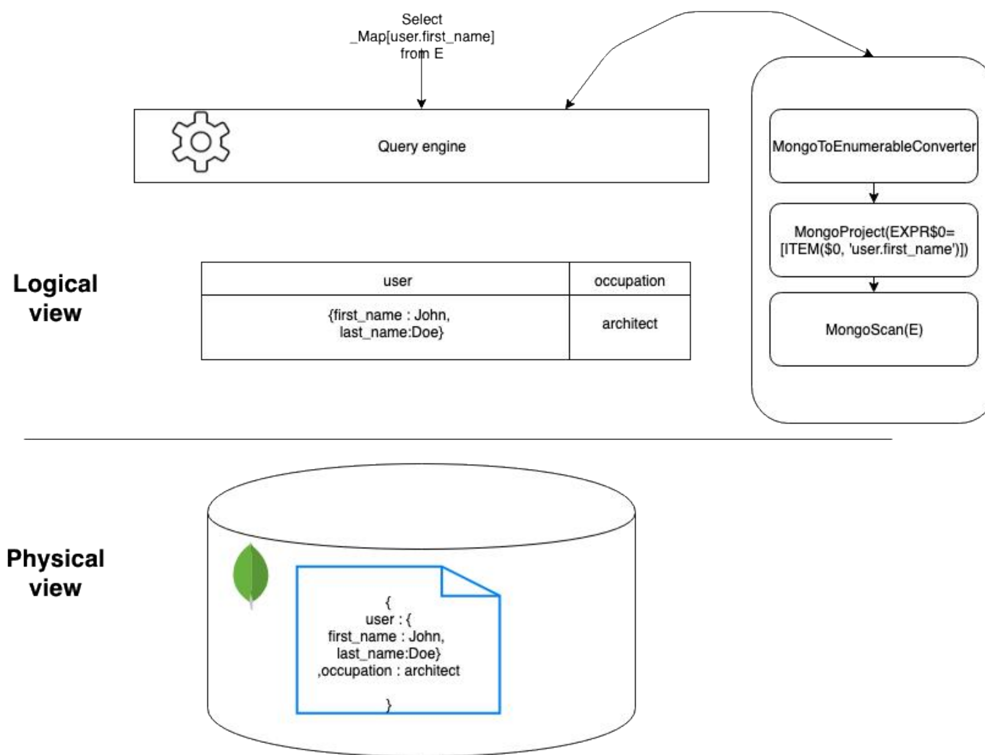



*Figure 7.1: Principe de manipulation de valeurs structuré en SQL dans les systèmes Big Data*

# Bibliographie


[ABM07] Salwani Abdullah, Edmund Burke, and Barry Mccollum. Using a Randomised Iterative Improvement Algorithm with Composite Neighbourhood Structures for the University Course Timetabling Problem, volume 39, pages 153–169. 01 2007.

[ACR+12] M. Akdere, U. Cetintemel, M. Riondato, E. Upfal, and S. B.Zdonik. Learning-based query performance modeling and pre-diction. In2012 IEEE 28th International Conference on Data Engineering, pages 390–401, 2012.3

[adlC] Les auteurs de la CRE. Accueil. Available at https://www.smartgrids-cre.fr/, Accessed: 2021-03-09.

[AG15] Assad Al-Ghrairi. Heap and heap sort, 01 2015.

[AGZ+15] Michael Armbrust, Ali Ghodsi, Matei Zaharia, Reynold Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph Bradley, Xiangrui Meng, Tomer Kaftan, and Michael Franklin. Spark sql. Pages 1383–1394, 05 2015.

[AHH+15] Fatima Binta Adamu, Adib Habbal, Suhaidi Hassan, R. Les Cottrell, Bebo White, and Ibrahim Abdullah. A Survey On Big Data Indexing Strategies. 2015.

[AKGC+13] Mohammed Al-Kateb, Ahmad Ghazal, Alain Crolotte, Ramesh Bhashyam, Jaiprakash Chimanchode, and Sai Pavan Pakala. Temporal query processing in Teradata. In Proceedings of the 16th International Conference on Extending Database Technology, EDBT'13, page 573–578, New York, NY, USA, 2013. Association for Computing Machinery.

[Ala97] Suad Alagic. The odmg object model: Does it make sense? SIG-PLAN Not., 32(10):253–270, October 1997.

[AM17] Shakil Akhtar and Ravi Magham. Using Phoenix, pages 15–35. 122017.[Amo11] Fabiano Amorim. Complete Showplan Operators. Cengage Learning, 2011.

[AMSVP10] Salman Abdul Moiz, P Sailaja, G Venkataswamy, and Supriya Pal. Database replication: A survey of open source and commercial tools. 13, 01 2010.

[AU11] Foto Afrati and Jeffrey Ullman. Optimizing multiway joins in a map-reduce environment. IEEE Trans. Knowl. Data Eng.,23:1282–1298, 09 2011.

[Aut21] MongoDB Authors. Welcome to the MongoDB Documentation — MongoDB Documentation. Available at https://docs.MongoDB.com/, Accessed : 2021-03-25, 03 2021.

[AWSa]AWS. Qu'est-ce que les données diffusées en continu ? – Amazon Web Services (AWS). Available at https://aws.amazon.com/fr/streaming-data/, Accessed: 2021-03-15.

[AWSb]Team AWS. Tables globales : Réplication multi-régions avec DynamoDB - Amazon DynamoDB. Available at https://docs.aws.amazon.com/frfr/amazondynamodb/latest/developerguide/GlobalTables.html, Accessed:2021−03−25.

[AXF18] Ahmad Askarian, Rupei Xu, and Andras Farago. Parallelizing Large-Scale Graph Algorithms Using the Apache Spark-Distributed Memory System, 2018.

[AXG+15] Rodrigo Aniceto, Rene Xavier, Valeria Guimaraes, Fernanda Hondo, Maristela Holanda, Maria Walter, and Sergio Lifschitz. Evaluating the Cassandra NoSQL database approach for genomic data persistency. International journal of genomics, 2015:502795,11 2015.4

[Bal12] MagdaBalazinska.Cse444: Database internals – query execution. Available at https://courses.cs.washington.edu/courses/cse444/12sp/lectures/lecture07-execution.pdf, Accessed: 2021-03-25, 04 2012.

[BCN17] Nikita Bobrov, George Chernishev, and Boris Novikov. Workload-Independent Data-Driven Vertical Partitioning, pages 275–284.Springer International Publishing, Cham, 2017.

[BCRH+18] Edmon Begoli, Jesus Camacho-Rodrıguez, Julian Hyde, Michael Mior, and Daniel Lemire. Apache calcite: A foundational frame-work for optimized query processing over heterogeneous data sources. pages 221–230, 05 2018.
[BCS16] Angela Bonifati, Radu Ciucanu, and S lawek Staworko. Learning Join Queries from User Examples. ACM Transactions on Database Systems, 40(4):24:1–24:38, January 2016.
[Bea15] Rahul Beakta. Big Data and Hadoop: A review paper. international journal of computer science information te, 2, 01 2015.
[BHG87] Philip A. Bernstein, Vassco Hadzilacos, and Nathan Goodman. Concurrency Control and Recovery in Database Systems. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.
[BN09] Philip A. Bernstein and Eric Newcomer. Principles of Transaction Processing. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2009.
[Bre09] Stéphane Bressan. Distributed Query Optimization, pages 908–912. Springer US, Boston, MA, 2009.
[Bru15] Rudi Bruchez. Les Bases De Données NoSQL : Comprendre Et Mettre En Œuvre. Eyrolles, 61, bd Saint-Germain 75240 Paris Cedex 05, 2nd Edition, 2015.
[Cas] Architecture d'un cluster [Cassandra : Partitionnement et distribution]. Available at https://stph.scenari-community.org/contribs/nos/Cassandra3/co/1-Les-Clusters2.html, Accessed: 2021−03−25. [Cat11] Rick Cattell. Scalable sql and NoSQL data stores. SIGMOD Rec.,39(4):12–27, May 2011.
[Caz]Tristan Cazenave. Overestimating the admissible heuristic of a *for multiple sequence alignment.
[CDG+08] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. ACM Trans. Comput. Syst., 26(2), June 2008.
[CH10] Prabhakar Chaganti and Rich Helms. Amazon SimpleDB Developer Guide. Packt Publishing, 1st edition, 2010.5
[Che18] Jianyu Chen. A high-bandwidth snappy decompressor in reconfigurable logic. 10 2018.
[CM17] Carlos Coronel and Steven Morris. Database systems: design, implementation, and management. Cengage Learning, Australia; United States, 12e edition, 2017. OCLC: ocn945782247.
[CML14] Min Chen, Shiwen Mao, and Yunhao Liu. Big Data: A survey. Mob. Netw. Appl., 19(2):171–209, April 2014.
[Cod72] E. F. Codd. Relational completeness of data base sublanguages. In Database Systems, pages 65–98. Prentice-Hall, 1972.
[Cor15] Corp. Guide du Big Data - l'annuaire de référence à destination des utilisateurs.Availableathttps://www.bigdataparis.com/guide/BD14-15GuideBD14136.pdf, Accessed : 2021−03−15,2014−2015.
[CSPG16] T. Chawla, G. Singh, E. S. Pilli, and M. C. Govil. Research issues in rdf management systems. In2016 International Conference on Emerging Trends in Communication Technologies (ETCT), pages1–5, 2016.
[CSSC17] William Croft, Wei Shi, Jorg-Rudiger Sack, and Jean-Pierre Corriveau. Comparison of approaches of geographic partitioning for data anonymization. Journal of Geographical Systems, 19, 07 2017.
[CZ17] Lianhua Chi and Xingquan Zhu. Hashing techniques: A survey and taxonomy. ACM Comput. Surv., 50(1):11:1–11:36, April 2017.
[DCM19] Daniel Delahaye, Supatcha Chaimatanan, and Marcel Mongeau. Simulated Annealing: From Basics to Applications. In Michel Gendreau and Jean-Yves Potvin, editors, Handbook of Metaheuristics, International Series in Operations Research & Management Science, pages 1–35. Springer International Publishing, Cham,2019.

[DG04] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. InOSDI'04: Sixth Symposium on Operating System Design and Implementation, pages 137–150, San Francisco, CA, 2004.
[DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. Commun. ACM, 51(1) :107–113, January 2008.
[drwd] équipe de rédaction web d'Oracle. Quelle est la différence entre ELT et ETL ? Available at https://www.oracle.com/fr/database/elt-vs-etl.html, Accessed:2021-03-09.
[DrwdD] Equipe de rédaction web de Drools. Drools - Business Rules Management System (Java™, Open Source). Available at https://www.drools.org/, Accessed : 2021-03-09.6
[DrwdT] Equipe de rédaction web de Talend. ETL : Qu'est-ce que l'Extract Transform Load ? Available at https://www.talend.com/fr/resources/guide-etl/,Accessed:2021-03-09.
[EEI+13] Amr Ebaid, Ahmed Elmagarmid, Ihab Ilyas, Mourad Ouzzani, Joge-Arnulfo Quiané-Ruiz, Nan Tang, and Si Yin. Nadeef: a generalized data cleaning system. Proceedings of the VLDB Endowment, 6:1218–1221, 08 2013.
[EM10] Orri Erling and Ivan Mikhailov. Virtuoso: Rdf support in a native rdbms. Semantic Web Information Management, -1:501, 12 2010.
[EMK+04] Andrew Eisenberg, Jim Melton, Krishna Kulkarni, Jan-Eike Michels, and Fred Zemke. Sql:2003 has been published. SIGMODRec., 33(1):119–126, March 2004.
[ESBES19] Tharwat El-Sayed, Mohammed Badawy, and Ayman El-Sayed. Impact of small files on Hadoop performance: Literature survey and open points. Menoufia Journal of Electronic Engineering Re-search, 28:109–120, 01 2019.
[FB18] Diogo Fernandes and Jorge Bernardino. Graph databases comparison: Allegrograph, arangodb, infinitegraph, neo4j, and orientdb. In Proceedings of the 7th International Conference on Data Science, Technology and Applications, DATA 2018, page 373–380, Setubal, PRT, 2018. SCITEPRESS - Science and Technology Publications, Lda.
[Feg98] Leonidas Fegaras. A new heuristic for optimizing large queries. In 9th International Conference, DEXA'98, pages 726–735.Springer-Verlag, 1998.
[FLG07] G. Fischer, J. Lusiardi, and J. Gudenberg. Abstract syntax trees -and their role in model driven software development. pages 38–38,09 2007.
[FPST11] Avrilia Floratou, Jignesh M. Patel, Eugene J. Shekita, and Sandeep Tata. Column-oriented storage techniques for mapreduce. Proc. VLDB Endow., 4(7):419–429, April 2011.
[GAKS14] Ivan Giangreco, Ihab Al Kabary, and Heiko Schuldt. Adam -a database and information retrieval system for big multimedia collections. pages 406–413, 06 2014.
[GB10] Petra Grd and Miroslav Baca. Analysis of b-tree data structure and its usage in computer forensics. 01 2010.
[Geo11] Lars George. HBase: The Definitive Guide. O'Reilly Media, 1edition, 2011.7
[GMPQ+04] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, V. Vassalos, and J. Widom. The tsimmis approach to mediation: Data models and languages. Journal of Intelligent Information Systems, 8:117–132, 2004.
[GMUW08] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. Database Systems: The Complete Book. Prentice Hall Press, USA,2 edition, 2008.
[GMUW14] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. Database systems: the complete book. Pearson, Harlow, 2. edition, new international edition edition, 2014. OCLC: 856868464.
[Goo] Team Google. Comparaison des modèles statique et dynamique. Available at https://developers.google.com/machine-learning/crash-course/static-vs-dynamic-training/video-lecture?hl=fr, Accessed: 2021-03-25.

[GPTK16] V. Giannakouris, N. Papailiou, D. Tsoumakos, and N. Koziris. Musqle: Distributed sql query execution over multiple engine environments. In2016 IEEE International Conference on Big Data (Big Data), pages 452–461, 2016.
[Gra]Team Spark Graphix. Partition Strategy (Spark1.3.1JavaDoc). Available at https://spark.Apache.org/docs/1.3.0/api/java/org/Apache/spark/graphx/PartitionStrategy.html, Accessed: 2021-03-25.
[Gro15] Malaska T. Seidman J. Shapira G. Grover, M. Hadoop application architectures. O'Reilly Media, Sebastopol, CA, 1st edition, 2015.
[GSD17] David Gembalczyk, Felix Martin Schuhknecht, and Jens Dittrich. An experimental analysis of different key-value stores and relational databases. In Bernhard Mitschang, Daniela Nicklas, Frank Leymann, Harald Schoning, Melanie Herschel, Jens Teubner, Theo Harder, Oliver Kopp, and Matthias Wieland, editors, Datenbank systeme fur Business, Technologie und Web (BTW 2017), 17. Fachtagung des GI-Fachbereichs, Datenbanken und Informations systeme" (DBIS), 6.-10. Marz 2017, Stuttgart, Germany, Proceedings, volume P-265 of LNI, pages 351–360. GI, 2017.
[GSSH16] Abdullah Gani, Aisha Siddiqa, Shahaboddin Shamshirb and, and Fariza Hanum. A survey on indexing techniques for Big Data: Taxonomy and performance evaluation. Knowl. Inf. Syst., 46(2):241–284, February 2016.
[GTS06] A. Goyal, Ashish Thakral, and G. Sharma. Improved a* algorithm for query optimization. 2006.
[Gup17] Anish Gupta. Hive-processing structured data in Hadoop. 03 2017.
[GZH+19] Yan Gao, Hongbo Zhao, Ruoshi Hao, Zhiyong Cao, and Mengran Liu. Research on architecture construction of information system of soil environment under the perspective of Big Data. IOP Conference Series: Earth and Environmental Science, 310:052041, 092019.8
[Hal]Giffen Mc Cormick Prentice Adam Smith Hall, David Ricardo. Chapitre 3 L'offre et la demande. - ppt. Available at https://slideplayer.fr/slide/3971080/, Accessed : 2021-03-15.
[HBB11] Herodotos Herodotou, Nedyalko Borisov, and Shivnath Babu. Query optimization techniques for partitioned tables. In Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data, SIGMOD '11, pages 49–60, New York, NY, USA, 2011. ACM.
[HK17] Rachel Warren Holden Karau.4. Joins (SQL and Core) - High Performance Spark [Book]. Available at https://www.oreilly.com/library/view/high-performance-spark/9781491943199/ch04.html, Accessed: 2021-03-25, 2017.
[HN13] Michael Hausenblas and Jacques Nadeau. Apache drill: Interactive ad-hoc analysis at scale. Big Data, 1:100–104, 06 2013.
[Ho12] RickyHo. MongoDB architecture. Available at http://horicky.blogspot.fr/2012/04/MongoDB-architecture.html, Accessed: 2017-12-04, April 2012.
[Hor] Hortonworks. Apache Hadoop yarn. Available at https://fr.hortonworks.com/Apache/yarn/, Accessed:2017-12-7.
[Hor17] Hortonworks. Yarn - what's the big deal. Available athttps://fr.hortonworks.com/blog/yarn-whats-the-big-deal/, Accessed: 2017-12-7, feb 2017.
[HQZ18] Rihan Hai, Christoph Quix, and Chen Zhou. Query Rewriting for Heterogeneous Data Lakes: 22nd European Conference, ADBIS 2018, Budapest, Hungary, September 2–5, 2018, Proceedings, pages 35–49. 07 2018.
[HSL16] Z. He, M. Shi, and C. Li. Research and application of path-finding algorithm based on unity 3d. In2016 IEEE/ACIS 15th International Conference on Computer and Information Science (ICIS), pages 1–4, 2016.
[Hub13] Thibault Hubert. Prévision de la demande et pilotage des flux en approvisionnement lointain. Theses, Ecole Centrale Paris, January2013.
[HYZ17] Long-Kai Huang, Qiang Yang, and Wei-Shi Zheng. Online hashing. CoRR, abs/1704.01897, 2017.

[IBM] IBM.Data replication – ibm analytics. Available athttps://www.ibm.com/analytics/us/en/technology/data-replication/, Accessed: 2017-10-27.
[IK90] Y. E. Ioannidis and Younkyung Kang. Randomized algorithms for optimizing large join queries. SIGMOD Rec., 19(2):312–321, May1990.
[ino15] Big Data MDX with Mondrian and Apache Kylin. Available at https://www.slideshare.net/inovex/big-data-mdx-with-mondrian-and-Apache-kylin, Accessed:2021-03-15, November2015.9
[IW87] Yannis Ioannidis and Eugene Wong. Query optimization by simulated annealing. volume 16, pages 9–22, 12 1987.
[JBK+20] Aman Jain, Ata F. Baarzi, George Kesidis, Bhuvan Urgaonkar, Nader Alfares, and Mahmut Kandemir. Splitserve: Efficiently splitting Apache spark jobs across faas and iaas. In Proceedings of the 21st International Middleware Conference, Middleware '20, page 236–250, New York, NY, USA, 2020. Association for Computing Machinery.
[JK84] Matthias Jarke and Jurgen Koch. Query optimization in database systems. ACM Comput. Surv., 16(2):111–152, June 1984.
[Ka99] Harald Kosch. The use of randomized search strategies for complex parallel relational query optimization. 10 1999.
[Kah11] Scott D. Kahn. On the future of genomic data. Science (New York, N.Y.), 331(6018):728–729, February 2011.
[KC95] Jerry Kiernan and Michael J. Carey. Extending sql-92 for oodb access: Design and implementation experience. SIGPLAN Not.,30(10):467–480, October 1995.
[KMJ13] Magzhan Kairanbay and Hajar Mat Jani. A review and evaluations of shortest path algorithms. International Journal of Scientific Technology Research, 2:99–104, 01 2013.
[KSMH17] S. Kalid, A. Syed, A. Mohammad, and M. N. Halgamuge. Big-data NoSQL databases: A comparison and analysis of "big-table", "dynamodb", and "cassandra". In2017 IEEE 2nd International Conference on Big Data Analysis (ICBDA), pages 89–93, 2017.
[KV13] Ridhi Kapoor and R Virk. Selectivity cost estimates in query optimization in distributed databases. International Journal of Enhanced Research in Management Computer Applications, 2:2319–7471, 07 2013.
[KY12] Jason Kane and Qing Yang. Compression speed enhancements tolzo for multi-core systems. pages 108–115, 10 2012.
[KYDF17] Dan Kangas, Weixu Yang, Ajay Dholakia, and Brian Finley. Lenovo Big Data reference architecture for hortonworks data platform. Lenovo press, 2017.
[LC15] Félix Lopez and Eulogio Cruz. Literature review about neo4jgraph database as a feasible alternative for replacing rdbms. Industrial Data, 18:135, 12 2015.
[Len09] Joe Lennon. Beginning CouchDB. Apress, USA, 1st edition, 2009.
[LG11] Xiang Liu and Daoxiong Gong. A comparative study of a-star algorithms for search and rescue in perfect maze. 04 2011.10
[LGM+15] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. How good are query optimizers, really? Proc. VLDB Endow., 9(3):204–215, November2015.
[Li14] Haifeng Li. Distributed NoSQL: MongoDB - Dataconomy. Available at https://dataconomy.com/2014/08/distributed-nosql-MongoDB/, Accessed: 2021-03-25, 08 2014.
[LSH+14] Jeff LeFevre, Jagan Sankaranarayanan, Hakan Hacigumus, Junichi Tatemura, Neoklis Polyzotis, and Michael Carey. Miso: Souping up Big Data query processing with a multistore system. Proceedings of the ACM SIGMOD International Conference on Management of Data, 06 2014.
[LV91] Rosana S. G. Lanzelotte and Patrick Valduriez. Extending the search strategy in a query optimizer. In Proceedings of the 17thInternational Conference on Very Large Data Bases, VLDB '91, page 363–373, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc.

[LW01] Andy Liaw and Matthew Wiener. Classification and regression by random forest. Forest, 23, 11 2001.

[Mak77] Akifumi Makinouchi. A consideration on normal form of not-necessarily-normalized relation in the relational data model. pages447–453, 1977.

[Mar13] MariaDB. Database master-slave replication in the cloud - mariadb. Available at https://mariadb.org/database-master-slave-replication-in-the-cloud/, Accessed: 2017-10-27, May 2013.

[Mar15] Adam Marcus. The architecture of open source applications: The NoSQL ecosystem. Available at http://www.aosabook.org/en/nosql.html, Accessed: 2017-11-28, March 2015.

[ME]Nayyer Masood and Barry Eaglestone. Component and federation concept models in a federated database system.

[ME92] Priti Mishra and Margaret H. Eich. Join processing in relational databases. ACM Comput. Surv., 24(1):63–113, March 1992. [ME16] Michael Malak and Robin East. Spark GraphX in Action. Manning Publications Co., USA, 1st edition, 2016.

[Mel96] Jim Melton. Sql language summary. ACM Comput. Surv.,28(1):141–143, March 1996.

[MHK+18] Jan Michels, Keith Hare, Krishna Kulkarni, Calisto Zuzarte, Zhen Hua Liu, Beda Hammerschmidt, and Fred Zemke. The new and improved sql: 2016 standard. SIGMOD Rec., 47(2):51–60, December 2018.11

[Mil13] J. J. Miller. Graph database applications and concepts with neo4j.2013. [MN08] Guido Moerkotte and Thomas Neumann. Dynamic programming strikes back. Shasha, Dennis; Wang, Jason T. L.: Proceedings of the ACM SIGMOD 2008 International Conference on Management of Data, ACM, 539-552 (2008), 01 2008.

[MNM+19] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. Neo. Proceedings of the VLDB Endowment,12(11):1705–1718, Jul 2019.

[Mon17a] MongoDB. MongoDB documentation. Available at https://docs.MongoDB.com, Accessed: 2017-11-22, 2017.

[Mon17b] MongoDB.3.4. MongoDB architecture guide. Available at MongoDB. Retrieved from https://saipraveenblog.files.wordpress.com/2016/12/MongoDBarchitectureguide.pdf, Accessed:2017−11−28, June2017.

[Mon17c] MongoDB.3.4. Performance best practices for MongoDB. Available at https://saipraveenblog.files.wordpress.com/2016/12/MongoDB-performance-best-practices.pdf, Accessed: 2017-11-27, June 2017.

[MOS+13] K. Morton, K. Osborne, R. Sands, R. Shamsudeen, and J. Still. Pro Oracle SQL. Expert's voice in Oracle. Apress, 2013.

[MP18] Ryan Marcus and Olga Papaemmanouil. Deep reinforcement learning for join order enumeration. In Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, aiDM'18, New York, NY, USA,2018. Association for Computing Machinery. [mw15] media wiki. Hbase query language features, Apr 2015.

[NB15] Ricardo Neves and Jorge Bernardino. Performance and scalability of voldemort NoSQL. 07 2015.

[Neo] Team Neo4J.Clustering - Operations Manual. Available athttps://neo4j.com/docs/operations-manual/4.2/clustering/, Accessed: 2021-03-25.

[noaa] AWS Data Pipeline - Service ETL géré - Amazon Web Services. Available at https://aws.amazon.com/fr/datapipeline/, Accessed : 2021-02-26.

[Noab] La CNIL publie un nouveau modèle de registre simplifié | CNIL. Available at https://www.cnil.fr/fr/la-cnil-publie-un-nouveau-modele-de-registre-simplifie, Accessed: 2021-02-26.

[noac] standard deviation. Available at https://www.oxfordreference.com/view/10.1093/oi/authority.20110803100527452, Accessed : 2021-02-26.12

[Noad] Talend Data Preparation - Libre-service dans le cloud ou on-premise. Available at https://www.talend.com/fr/products/data-preparation/, Accessed: 2021-02-26. [OBGK18] Jennifer Ortiz, Magdalena Balazinska, Johannes Gehrke, and S. Keerthi. Learning state representations for query optimization with deep reinforcement learning. pages 1–4, 06 2018.

[Oraa] Team Oracle. Database Concepts. Available at https://docs.oracle.com/cd/E1188201/server.112/e40540/logical.htmCN CP T1046, Accessed:2021−03−25.

[Orab] Team Oracle. Using the Rule-Based Optimizer. Available athttps://docs.oracle.com/cd/B1050001/server.920/a96533/rbo.htm, Accessed:2021−03−25.

[Ora17] Oracle®. Sql tuning guide. In Oracle ® Database - 12c Release 1(12.1), July 2017.

[OV11] M. Tamer Ozsu and Patrick Valduriez. Principles of distributed database systems. Springer Science+Business Media, New York,3rd ed edition, 2011. OCLC: ocn706920112.

[Pok11] Jaroslav Pokorny. NoSQL databases: A step to database scalability in web environment. In Proceedings of the 13th International Conference on Information Integration and Web-based Applications and Services, iiWAS '11, pages 278–283, New York, NY, USA, 2011. ACM.

[Posa] Team PostgreSQL. 12.9. Types d'index GiST et GIN. Available athttps://docs.postgresql.fr/8.3/textsearch-indexes.html, Accessed:2021-03-25.

[Posb] Team PostgreSQL.50.3. Etape d'analyse. Available athttps://docs.postgresql.fr/10/parser-stage.html, Accessed: 2021-03-25.

[Pos15] Team PostreSQL. hstore. Available at https://www.postgresql.org/docs/9.0/hstore.html,Accessed:2021-03-25, October 2015.

[Pos21] Team PostgreSQL. GIN Indexes. Available at https://www.postgresql.org/docs/9.5/gin.html,Accessed:2021-03-25, February 2021.

[QTCZ18] Jianzhong Qi, Yufei Tao, Yanchuan Chang, and Rui Zhang. Theoretically optimal and empirically efficient r-trees with strong parallelizability. Proc. VLDB Endow., 11(5):621–634, January 2018.

[RDD] Team SparkRDD. RDD Programming Guide-Spark3.1.1 Documentation. Available at https://spark.Apache.org/docs/latest/rdd-programming-guide.htmlresilient-distributed-datasets-rdds, Accessed: 2021-03-25.13

[Ros99] Jeffrey S. Rosenthal. Parallel computing and monte carlo algorithms, 1999.

[Rud17] Sebastian Ruder. An overview of gradient descent optimization algorithms, 2017.

[SA17] Cdsa Swa and Zahid Ansari. Apache pig - a data flow framework based on Hadoop map reduce. International Journal of Engineering Trends and Technology, 50:271–275, 08 2017.

[Sch19] Hans-Jurgen Schonig. Mastering PostgreSQL 12: Advanced techniques to build and administer scalable and reliable PostgreSQL database applications, 3rd Edition. Packt Publishing Ltd, November 2019. Google-Books-ID: g1jBDwAAQBAJ.

[SD21] Pegdwendé N. Sawadogo and Jérôme Darmont. On data lake architectures and metadata management. J. Intell. Inf. Syst.,56(1):97–120, 2021

[SDG09] B. Sun, J. Du, and T. Gao. Study on the improvement of k-nearest-neighbor algorithm. In2009 International Conference on Artificial Intelligence and Computational Intelligence, volume 4, pages 390–393, 2009.

[SDRL16] Martin Strohbach, Jorg Daubert, Herman Ravkin, and Mario Lischka. Big Data Storage, pages 119–141. 01 2016.

[Ser13] Lars George (Director EMEA Services). Hbase schema de-sign. Available at https://2013.nosql-matters.org/cgn/wp-content/uploads/2013/05/HBase-Schema-Design-NoSQL-Matters-April-2013.pdf, Accessed: 2017-12-04, April 2013.

[SF12] Pramod J. Sadalage and Martin Fowler. NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence. Addison-Wesley Professional, 1st edition, 2012.

[SF13] Pramod J. Sadalage and Martin Fowler. NoSQL distilled: a brief guide to the emerging world of polyglot persistence. Addison-Wesley, Upper Saddle River, NJ, 2013.

[SGMH18] K. Srinivasa, Siddesh G M, and Srinidhi Hiriyannaiah. Apache Hive, pages 55–72. 04 2018.

[Shr15] A. Shrivastava. Probabilistic Hashing Techniques for Big Data.2015.

[Shr17] Raju Shrestha. High availability and performance of database in the cloud - traditional master-slave replication versus modern cluster-based solutions. pages 413–420, 01 2017.

[Siv12] Swaminathan Sivasubramanian. Amazon dynamodb: A seamlessly scalable non-relational database service. In Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, SIGMOD '12, page 729–730, New York, NY, USA,2012. Association for Computing Machinery.14

[SJM+17] Anil Shanbhag, Alekh Jindal, Samuel Madden, Jorge Quiane, and Aaron J. Elmore. A robust partitioning scheme for ad-hoc query workloads. In Proceedings of the 2017 Symposium on Cloud Computing, SoCC '17, pages 229–241, New York, NY, USA, 2017.ACM.

[snh] snhegde. Clusters and Racks. Available at https://docs.vmware.com/en/VMware-Validated-Design/5.0/com.vmware.vvd.sddc-design.doc/GUID-49EAD8E1-3F0D-4EB4-9AFB-2EF8AA81B07A.html, Accessed: 2021-03-25.

[Spa] Team Spark. Cluster Mode Overview - Spark 3.1.1 Documentation. Available at https://spark.Apache.org/docs/latest/cluster-overview.html, Accessed: 2021-03-25.

[SS19] Ahmad Shah and Muhammad Sethi. The improvised gzip, a technique for real time lossless data compression. EAI Endorsed Trans-actions on Context-aware Systems and Applications, 6:160599, 062019.

[Tay] James Taylor. Apache phoenix: Transforming hbase into a sql database. Available at http://phoenix.Apache.org/presentations/HadoopSummit2014-16x9.pdf, Accessed: 2021-03-09,

[Tea15] Apache Hive Team. Language manual – Apache hive – Apache software foundation. Available at https://cwiki.Apache.org/confluence/display/Hive/LanguageManual, Accessed: 2017-12-11, 2015.

[Tea16] Apache HBase Team. Hbase™ reference guide. Available athttps://hbase.Apache.org/1.2/book.html, Accessed: 2017-12-11,2016.

[Tea17] Neo4J Team. The neo4j operations manual - v3.3. Available at https://neo4j.com/docs/pdf/neo4j-operations-manual-3.3.pdf, Accessed: 2017-12-11, 2017.

[Teca] Technet.microsoft.com. Partitionning. Available at https://technet.microsoft.com/en-us/library/ms178148(v=sql.105).aspx, Accessed: 2017-11-20.

[Tecb] Technet.microsoft.com. Query tuning - analyzing a query: Logical and physical operators reference. Available at https://technet.microsoft.com/en-us/library/ms191158(v=sql.105).aspx, Accessed: 2017-11-24.

[TLRG08] David Taniar, Clement H. C. Leung, Wenny Rahayu, and Sushant Goel. High-Performance Parallel Database Processing and Grid Databases. John Wiley & Sons, September 2008.

[VKKS17] Revathi Velusamy, Rakshitha K.R, Sruthi. K, and Guruprasaath.S. Big Data with hadoop – for data management, processing and storing. International Research Journal of Engineering and Tech-nology (IRJET), 4:1111–1116, 08 2017.15

[VMP17] Vandana, V. Monica, and N. K. Parambalath. Shuffle phase optimization in spark. In2017 International Conference on Advances in Computing, Communications and Informatics (ICACCI), pages1028–1034, 2017.

[Voh16a] Deepak Vohra. Apache Avro, pages 303–323. 09 2016. [Voh16b] Deepak Vohra. Apache HBase Primer. 01 2016.
[Voh16c] Deepak Vohra. Apache Parquet, pages 325–335. 09 2016. [VVDG+11] Nathalie Villa-Vialaneix, Taoufiq Dkaki, Sébastien Gadat, Jean-Michel Inglebert, and Quoc-Dinh Truong. Recherche et représentation de communautés dans un grand graphe. Une approche combinée. Document numérique, 14 :59–80, 05 2011.
[WCHN13] Wentao Wu, Yun Chi, Hakan Hacıgumu¸s, and Jeffrey Naughton. Towards predicting query execution time for concurrent and dynamic database workloads. Proceedings of the VLDB Endowment,6:925–936, 08 2013.
[WCZ+13] Wentao Wu, Yun Chi, Shenghuo Zhu, Junichi Tatemura, H. Hacigumus, and J.F. Naughton. Predicting query execution time: Are optimizer cost models really unusable? pages 1081–1092, 04 2013.
[Whi15] T White. Hadoop: The definitive guide. O'Reilly Media., Sebastopol, CA, USA, 4th edition, 2015.
[Wik17] Wikipedia. Hashtable. Available at https://en.wikipedia.org/wiki/Hash<sub>t</sub>able, Accessed:2017−11−07, October2017.
[WKW16] Karl Weiss, Taghi Khoshgoftaar, and Ding Ding Wang. A survey of transfer learning. Journal of Big Data, 3, 05 2016.
[WSSJ14] Jingdong Wang, Heng Tao Shen, Jingkuan Song, and Jianqiu Ji. Hashing for similarity search: A survey. CoRR, abs/1408.2927,2014.
[XE02] Li Xu and David Embley. Combining the best of global-as-view and local-as-view for data integration. ISTA, Volume 48 of LNI,12 2002.
[YHKS08] Mika Ylianttila, Erkki Harjula, Timo Koskela, and Jaakko Sauvola. Analytical model for mobile p2p data management systems. pages 1186 – 1190, 02 2008.
[ZCM20] Asma Zgolli, Christine Collet, and Cédrine Madera. Metadata in Data Lake Ecosystems, chapter 4, pages 57–96. John Wiley Sons, Ltd, 2020.
[ZCHC11] Wanli Zuo, Yongheng Chen, Fengling He, and Kerui Chen. Optimization strategy of top-down join enumeration on modern multi-core cpus.J. Comput., 6:2004–2012, 2011.
[ZCR15] Weijie Zhao, Yu Cheng, and Florin Rusu. Workload-driven vertical partitioning for effective query processing over raw data. CoRR, abs/1503.08946, 2015.16
[Zho09] Jingren Zhou. Sort-Merge Join, pages 2673–2674. Springer US, Boston, MA, 2009. [ZS17] N. Zhivchikova and Y. Shevchuk. Riak kv performance in sensor data storage application. Program Systems: Theory and Applications, 8:61–85, 01 2017.
[Zur]Thomas Zurek. Hash Joins. Available at http://www.dcs.ed.ac.uk/home/tz/phd/thesis/node21.htm, Accessed: 2021-03-25