Collège Doctoral
Université Sud de France
Montpellier - Perpignan

THÈSE

Pour obtenir le grade de

Docteur

Délivré par l'**Université de Montpellier**

Préparée au sein de l'école doctorale **I2S***
Et de l'unité de recherche **UMR 5506**

Spécialité: **Informatique**

Présentée par **Carlyna Bondiombouy**
carlyna.bondiombouy@inria.fr

Traitement de requêtes dans les systèmes multistores

Soutenue le 12/07/2017 devant le jury composé de :

Mme. Angela BONIFATI	Professeur	Université Claude Bernard Lyon 1	Rapporteur
M. Farouk TOUMANI	Professeur	Université Blaise Pascal	Rapporteur
Mme. Sarah COHEN-BOULAKIA	Maitre de Conférence (HDR)	Université Paris-Sud	Examinatrice
M. Pierre GENEVÈS	Chargé de Recherche	CNRS	Examinateur
Mme. Esther PACITTI	Professeur	Université de Montpellier	Examinatrice
M. Patrick VALDURIEZ	Directeur de recherche	Inria	Directeur de thèse

UNIVERSITÉ MONTPELLIER

* **I2S:** ÉCOLE DOCTORALE INFORMATION STRUCTURES SYSTÈMES

The Lord is gracious and compassionate, slow to anger and rich in love. The Lord is good to all; he has compassion on all he has made. All your works praise you, Lord; your faithful people extol you. They tell of the glory of your kingdom and speak of your might, so that all people may know of your mighty acts and the glorious splendor of your kingdom.

—Bible

Dedication

To my family.

To children worldwide.

Acknowledgments

Thank you my LORD, my victory Butterfly, my Redeemer for always being there for me, thank you for your superabundant graces unworthy. You are my all, the one in whom I find my worth, I am nothing without you my faithful friend. Victory belongs to JESUS, I can do all this through him who gives me strength. Your grace is enough for me, dear husband you are the reason that I live. I put my trust and hope in you, you are my provider. All of the glory, the honor and the praise belong to you.

I would like to express my profound gratitude to Patrick Valduriez my thesis supervisor, for his helpful assistance, great availability, patience, precious advice and encouragement. His competence, scientific rigor and clairvoyance have taught me a lot. They have been and will remain engines of my work.

I thank Boyan Kolev and Oleksandra Levchenko without whom this thesis would not be what it is, as much by the discussions, the suggestions, and the contributions. I thank Esther Pacitti for her availability, valuable advices on my research and my career planning. I am thankful to the collaborators and contributors during my thesis. To Ricardo Jiménez-Peris, Raquel Pau, José Pereira and Pavlos Kranas for the great collaboration that we had in the CoherentPaas project.

Thanks also go to the committee members Farouk Toumani, Angela Bonifati, Pierre Genevès, Sarah Cohen-Boulakia and Esther Pacitti, for spending their time to evaluate my thesis, for their profound and carefully aimed comments and suggestions during the oral defense. This work would not have been possible without the support of the Congolese State, which enabled me, thanks to an allocation of research, to devote myself serenely to the elaboration of my thesis.

I would also like to record my sincere thanks to all members of the Zenith team. Special thanks to Dennis Shasha, Djamel Yagoubi, Sakina Mahboubi, Miguel Liroz, Ji Liu, Medhi Zitouni and Maximilien Servajean who gave me helpful advice on my research work. Thanks also go to Florent Masseglia, Reza Akbarinia, Laurence Fontana, Khadidja Meguelati, Titouan Lorieul, Saber Salah, Reda Bouadjenek, Rim Moussa, Daniel Gaspar, Valentin Leveau and Sen Wang with whom, I often recall the happy time we spent together.

I would like to thank my parents for all the opportunities they have offered me to achieve the greatest success. Thank you for teaching me how to love. To my sisters, brothers, nieces, nephews, aunts, uncles, grandmothers and grandfathers thank you. Your love, prayers and guidance have helped me to face all the trials. To my angels: Samuel and Elijah, who taught me that life is a gift, a treasure, a grace to be received and to be

given away.

I thank Dovene Agogue, Nora Aouba, Josiane Mendy, Bettina Ibouanga, Arnaud Nzomambou, More Ogounde, Tresor Pemosso, Audrey Bouhouama Kassa, Polha Guimbi, Tidiane Cherif Fall, Bouanga Solila, Yoka Nida, Anna Ngouba, Styven Lankiang, Davina Makosso, Aurore Ngono, Marie Fernandez Boni, Florent Fernandez and Armelle Nzang. The times spent together, your advice, your suggestions and your words of encouragement have illuminated my life. I would like to acknowledge Frederic Kikadidi for his support, for reviewing my thesis and providing valuable feedback.

I express my gratitude to the professors Abdoul Aziz and Oumar Diankha for their support, for sharing their valuable research experience with me. I would like to thank the Musika team in particular: Agnès, Marie, and Céline. They made my life a truly memorable experience. I would like to thank Francois Baty Sorel and Marie Pluzanski for moral support and encouragement. I express my thanks to Brigitte Manckoundia, Gabrielle Attibayeba, Narcisse Nadjingar, Mamoudou Ibrahima, Alain Loufimpou, Yoka Noelle, Victor Gbenou, Braithe Mangombi, Amaelle Otandault, Grace Embolo, Armide Meboua, Julie Minsta, Chardel Gokini, Géraud Fokou, Amiel Balebana, Debora Issoibeka, Rebecca Yaca, Carelle Koutchanou, Bernadette Faye, Therese Nouga and her family for their helps and valuable advices.

I would like to express my heartfelt thanks to Lalou Yavoucko, Kilone Manta Mondeila, Gerrys Mboumba, Anais Gabiot, Sabrina Mathat, Amegbo Assogba, Narcisse Badiette, Marie Thiamane, Debora Batchi, Nailoth Betty, Ray Loubayi, Jose Kotshi, Hapsita Oriane Ady, Francis Ganga, Benny Boungou, Anne-Marie Manga, Djennie Doukaga, Judith Poaty, Bonheur Djatto, Sara Tchibinda, Cinthia Elenga, Yedisa Makosso, Zoé Elenga, Caprel Tchissambou, Group of adoration of sainte Bernadette, Olga Kouikani, Gerolg Sita, Marie Nkounkou, Emmanuel Ehounda, Boverly Mandiangou, Patric Mbah, Mariette Nkama, Marcelle Kombo, Elise Assala and Anaelle Kibelolo.

I am also very grateful to Ismael Mayaki, Bery Mbaiossoum, Guy Ngaha, Théophile Ngapa, Marie Ngapa, Moustapha Bikienga, Zakaria Sahraoui, Raoul Tiam and Henry Teguiak. I would like to thank everyone at UM, LIRMM, Inria and all the other people who had a direct or indirect contribution to this work and were not mentioned above. I appreciate their help and support.

Résumé

Le cloud computing a eu un impact majeur sur la gestion des données, conduisant à une prolifération de nouvelles solutions évolutives de gestion des données telles que le stockage distribué de fichiers et d'objets, les bases de données NoSQL et les frameworks de traitement de données. Cela a conduit également à une grande diversification des interfaces aux SGBD et à la perte d'un paradigme de programmation commun, ce qui rend très difficile l'intégration de données pour un utilisateur, lorsqu'elles se trouvent dans des sources de données spécialisées, par exemple, relationnelle, document et graphe.

Dans cette thèse, nous abordons le problème du traitement de requêtes avec plusieurs sources de données dans le cloud, où ces sources ont des modèles, des langages et des API différents. Cette thèse a été préparée dans le cadre du projet européen CoherentPaaS et, en particulier, du système multistore CloudMdsQL. CloudMdsQL est un langage de requête fonctionnel capable d'exploiter toute la puissance des sources de données locales, en permettant simplement à certaines requêtes natives portant sur les systèmes locaux d'être appelées comme des fonctions et en même temps optimisées, par exemple, en exploitant les prédicats de sélection, en utilisant le bindjoin, en réalisant l'ordonnancement des jointures ou en réduisant les transferts de données intermédiaires.

Dans cette thèse, nous proposons une extension de CloudMdsQL pour tirer pleinement parti des fonctionnalités des frameworks de traitement de données sous-jacents tels que Spark en permettant l'utilisation ad hoc des opérateurs de map/filter/reduce (MFR) définis par l'utilisateur en combinaison avec les ordres SQL traditionnels. Cela permet d'effectuer des jointures entre données relationnelles et HDFS. Notre solution permet l'optimisation en permettant la réécriture de sous-requêtes afin de réaliser des optimisations majeures comme le bindjoin ou le filtrage des données le plus tôt possible.

Nous avons validé notre solution en implémentant l'extension MFR dans le moteur de requête CloudMdsQL. Sur la base de ce prototype, nous proposons une validation expérimentale du traitement des requêtes multistore dans un cluster pour évaluer l'impact sur les performances de l'optimisation. Plus précisément, nous explorons les avantages de l'utilisation du bindjoin et du filtrage de données dans des conditions différentes. Dans l'ensemble, notre évaluation des performances illustre la capacité du moteur de requête CloudMdsQL à optimiser une requête et à choisir la stratégie d'exécution la plus efficace.

Titre en français

Traitement de requêtes dans les systèmes multistores

Mots-clés

- Systèmes de gestion de données dans le cloud
- Systèmes multistores
- Systèmes multi-bases de données
- Traitement de requêtes

Abstract

Cloud computing is having a major impact on data management, with a proliferation of new, scalable data management solutions such as distributed file and object storage, NoSQL databases and big data processing frameworks. This also leads to a wide diversification of DBMS interfaces and the loss of a common programming paradigm, making it very hard for a user to integrate its data sitting in specialized data stores, e.g. relational, documents and graph data stores.

In this thesis, we address the problem of query processing with multiple cloud data stores, where the data stores have different models, languages and APIs. This thesis has been prepared in the context of the CoherentPaaS European project [1] and, in particular, the CloudMdsQL multistore system. CloudMdsQL is a functional query language able to exploit the full power of local data stores, by simply allowing some local data store native queries to be called as functions, and at the same time be optimized, e.g. by pushing down select predicates, using bind join, performing join ordering, or planning intermediate data shipping.

In this thesis, we propose an extension of CloudMdsQL to take full advantage of the functionality of the underlying data processing frameworks such as Spark by allowing the ad-hoc usage of user defined map/filter/reduce (MFR) operators in combination with traditional SQL statements. This allows performing joins between relational and HDFS big data. Our solution allows for optimization by enabling subquery rewriting so that bind join can be used and filter conditions can be pushed down and applied by the data processing framework as early as possible.

We validated our solution by implementing the MFR extension as part of the CloudMdsQL query engine. Based on this prototype, we provide an experimental validation of multistore query processing in a cluster to evaluate the impact on performance of optimization. More specifically, we explore the performance benefit of using bind join and select pushdown under different conditions. Overall, our performance evaluation illustrates the CloudMdsQL query engine's ability to optimize a query and choose the most efficient execution strategy.

Title in English

Query Processing in Multistore Systems

Keywords

- Cloud data stores
- Multistore systems
- Multidatabase systems
- Query processing

Équipe de Recherche

Zenith Team, Inria & LIRMM

Laboratoire

LIRMM - Laboratoire d'Informatique, Robotique et Micro-électronique de Montpellier

Adresse

Université Montpellier
Bâtiment 5
CC 05 018
Campus St Priest - 860 rue St Priest
34095 Montpellier cedex 5

Résumé Étendu

Introduction

Le cloud a eu un impact majeur sur la gestion des données, conduisant à une prolifération de nouvelles solutions telles que le stockage de fichiers et d'objets distribués (ex. GFS, HDFS), les bases de données NoSQL (ex. Hbase, MongoDB, No4J) et les frameworks de traitement de données massives (ex. MapReduce, Spark). Ces solutions ont été la base d'une riche offre de services de cloud (IaaS, PaaS, SaaS, DaaS, etc.). Cependant, cela conduit à une grande diversification des interfaces des SGBD et à la perte d'un paradigme de programmation commun.

Pour la gestion des données dans le cloud, on peut s'appuyer sur les systèmes de gestion de données relationnels (SGBDR), qui disposent d'une version distribuée et parallèle. Cependant, les SGBDR sont critiqués depuis quelques années pour leur approche taille unique [54]. Bien qu'ils aient été en mesure d'intégrer le support pour toutes sortes de données (par exemple, des objets multimédias, des documents XML) et de nouvelles fonctions, cela a entraîné une perte de performances, de simplicité et de flexibilité pour les applications avec des exigences spécifiques. Cette critique a conduit au développement de SGBD plus spécialisés pour un type d'application. Ce qui rend difficile pour un utilisateur d'intégrer ses données qui résident dans des bases de données spécialisées, par exemple, relationnelles, documents et graphes. Considérons par exemple un utilisateur qui a une base de données relationnelle contenant des auteurs, une base de documents contenant des revues et une base de données graphe contenant les relations entre auteurs, et veut s'informer sur les conflits d'intérêts dans l'examen de certains documents. La solution principale aujourd'hui serait de fournir un programme (par exemple écrit en Java) qui accède aux trois bases de données via leurs API et intègre les données (en mémoire). Cette solution est très laborieuse et difficilement extensible (par exemple, pour traiter une nouvelle base de données).

Cette thèse a été préparée dans le cadre du projet européen CoherentPaaS [1]. Ce projet a dû faire face à deux problèmes majeurs de la gestion des données dans le cloud : la perte de cohérence des données due à l'absence de transactions et le fait que les requêtes dans les bases de données doivent être programmées et optimisées manuellement. CoherentPaaS s'attaque à ce problème en fournissant une plateforme PaaS riche avec une intégration cohérente, évolutive et efficace des technologies de gestion des données NoSQL, SQL et de traitement d'événements complexes (CEP). Pour ce faire, CoherentPaaS fournit

un modèle de programmation et un langage communs pour interroger différentes bases de données. La plateforme est conçue pour permettre à différents sous-ensembles des données d'un utilisateur de se matérialiser dans différents modèles de données, de sorte que chaque sous-ensemble est traité de la manière la plus efficace en fonction de ses modèles d'accès aux données les plus courants. D'autre part, une application peut toujours accéder à une base de données directement, sans utiliser notre moteur de requête. Cela constitue un système de stockage de données multiples avec des niveaux élevés d'hétérogénéité et d'autonomie locale.

Dans cette thèse, nous nous concentrons sur le problème du traitement efficace des requêtes de données hétérogènes avec un langage commun. Le problème peut être exprimé comme suit. Soit $Q(S1, S2, ...Sn)$ une requête sur n bases de données, chacune avec un modèle de données et un langage de requête différents, et dans certains cas (par exemple une base de documents, une base de données graphe) une API différente, le problème est de proposer une approche pour traduire Q dans un plan d'exécution de requêtes optimisé (QEP), avec une gestion efficace des résultats intermédiaires.

Afin de relever ces défis, CoherentPaaS propose le *système multistore* et son langage fonctionnel CloudMdsQL pour interroger plusieurs bases de données hétérogènes à l'aide de requêtes imbriquées. Une requête CloudMdsQL peut exploiter toute la puissance des bases de données locales, en permettant simplement à certaines requêtes natives sur des données locales d'être appelées en tant que fonctions et en même temps optimisées sur la base d'un modèle de coût simple. L'architecture du moteur de requête est entièrement distribuée, de sorte que les nœuds du moteur de requête peuvent communiquer directement entre eux, en échangeant du code (des plans de requêtes) et des données. Cette architecture distribuée offre des opportunités importantes d'optimisation, en particulier, de réduire au minimum l'expédition de données entre les nœuds, de sélectionner le plutôt possible de prédicats, l'utilisation du bindjoin [32] et l'ordonnancement de jointures tout en réduisant le temps d'exécution, le coût de communication et le trafic réseau. Ces possibilités d'optimisation sont exploitées par le compilateur CloudMdsQL.

Dans le contexte des systèmes multistores, une grande attention est accordée à l'intégration des données non structurées généralement stockées dans HDFS avec des données relationnelles. Une solution principale est d'utiliser un moteur de requête relationnel qui permet aux requêtes de type SQL de récupérer des données de HDFS. Cette solution est utilisée par exemple dans le système Polybase de Microsoft pour intégrer des données HDFS dans SQL Server Parallel Data Warehouse (PDW). Cependant, PDW doit fournir une vue relationnelle des données non structurées ce qui est difficile et pas toujours faisable.

Dans cette thèse, nous proposons une extension du système CloudMdsQL afin de tirer pleinement parti des fonctionnalités des frameworks de traitement de données HDFS en permettant l'utilisation des opérateurs map/filter/reduce (MFR) définis par l'utilisateur en combinaison avec les ordres SQL traditionnels. Notre solution permet l'optimisation grâce à la réécriture de sous-requêtes afin que le bindjoin puisse être utilisé et que les conditions de filtrage puissent être poussées et appliquées le plutôt possible par le framework.

Cette thèse contient 5 chapitres principaux : vue d'ensemble du traitement des requêtes dans les systèmes multistores ; conception de CloudMdsQL ; extension de CloudMdsQL avec MFR ; prototype et validation expérimentale. Nous décrivons ci-dessous ces chapitres, puis donnons une conclusion qui résume les contributions et propose des directions de recherche futures.

Vue d'ensemble du traitement des requêtes dans les systèmes multistores

Nous donnons un aperçu du traitement des requêtes dans les systèmes multistores. Nous commençons par introduire les solutions récentes de gestion des données dans le cloud et le traitement des requêtes dans les systèmes multi-bases de données.

Les systèmes multistores [40] (également appelés polystores [25]) fournissent un accès intégré à plusieurs base de données dans le cloud via un ou plusieurs langages de requête. Divers systèmes multistores ont été construits, avec des objectifs, des architectures et des approches de traitement de requêtes différents. Pour faciliter la comparaison, nous divisons ces systèmes en fonction du niveau de couplage avec les bases de données sous-jacentes, c'est-à-dire faiblement couplés, fortement couplés et hybrides.

- Les systèmes faiblement couplés s'inspirent des systèmes multi-bases de données en ce sens qu'ils peuvent traiter des bases de données autonomes, qui peuvent alors être accédés par le langage commun du système multistore ainsi que séparément par leur langage local. Ils suivent l'architecture médiateur-wrapper avec plusieurs bases de données (par exemple NoSQL et SGBDR). Chaque base de données est autonome, c'est-à-dire contrôlée localement, et peut être accédée par d'autres applications.

- Les systèmes fortement couplés visent à interroger efficacement les données structurées et non structurées. Ils peuvent également avoir un objectif spécifique, comme l'auto-optimisation ou l'intégration des données HDFS et SGBDR. Cependant, ils sacrifient l'autonomie des systèmes au profit des performances de sorte que les bases de données ne peuvent être accessibles que par le système multistore, directement à travers leur API locale. Comme les systèmes faiblement couplés, ils fournissent un langage unique pour interroger les données hétérogènes. Cependant, le processeur de requêtes utilise directement les interfaces locales de stockage de données, ou dans le cas de HDFS, il interface un framework de traitement de données telle que MapReduce ou Spark. Ainsi, lors de l'exécution de la requête, le processeur de requêtes accède directement aux bases de données, ce qui est efficace. Cependant, le nombre de bases de données qui peuvent être interfacés est généralement très limité.

- Les systèmes hybrides combinent les avantages des systèmes faiblement couplés, notamment l'accès à de nombreuses bases de données différentes, et des systèmes

fortement couplés, notamment l'accès efficace à certaines bases de données directement à travers leur interface locale. L'architecture suit l'architecture médiateur-wrapper, tandis que le processeur de requêtes peut également accéder directement à certaines bases de données, par exemple, HDFS via MapReduce ou Spark.

Nous examinons et analysons certains systèmes multistores représentatifs pour chaque catégorie : (1) BigIntegrator, Forward et QoX ; (2) Polybase, HadoopDB et Estocada ; (3) Spark SQL, BigDAWG et CloudMdsQL. Nos comparaisons révèlent plusieurs tendances importantes. La principale tendance qui domine est la capacité d'intégrer des données relationnelles (stockées dans desSGBDR) avec d'autres types de données dans différentes bases de données, tels que HDFS ou NoSQL. Cependant, une différence importante entre les systèmes multistores réside dans le type de bases de données pris en charge. Nous notons également l'importance croissante de l'accès à HDFS au sein de Hadoop, en particulier avec MapReduce ou Spark. Une autre tendance est l'émergence de systèmes multistores auto-ajustables, dont l'objectif est d'exploiter les bases de données disponibles pour les performances. En termes de modèle de données et de langage de requête, la plupart des systèmes fournissent une abstraction relationnelle de type SQL.

Conception du système multistore CloudMdsQL

CloudMdsQL est un langage SQL fonctionnel, capable d'interroger plusieurs bases de données hétérogènes (relationnelles et NoSQL) au sein d'une requête unique pouvant contenir des invocations intégrées dans l'interface de requête native de chaque base de données. La principale innovation est qu'une requête CloudMdsQL peut exploiter toute la puissance des bases de données locales, en permettant simplement à certaines requêtes natives sur données locales (par exemple, une requête de recherche breadth-first sur une base de données graphe) d'être appelées comme des fonctions et en même temps d'être optimisées avec un modèle de coût simple, par exemple en exploitant les prédicats de sélection ; en utilisant le bindjoin ; en exécutant l'ordonnancement de jointure ou en planifiant la transmission de données intermédiaires.

Les solutions employées dans les systèmes multi-bases de données [24, 46] ne s'appliquent pas directement aux systèmes multistores. Tout d'abord, notre langage commun ne s'utilise pas pour interroger les bases de données sur le Web, qui pourraient être en très grand nombre. Dans le cloud une requête porte sur quelques bases de données et l'utilisateur doit avoir des droits d'accès sur chaque base de données. Ensuite, les bases de données peuvent avoir des langages très différents, allant de l'interface très simple get /put dans les bases de données clé-valeur, à des langages SQL ou SPARQL complets. Et aucun langage unique ne peut capturer tous les autres efficacement, par ex. SQL ne peut pas exprimer directement la traversée de chemin dans un graphe (bien sûr, nous pouvons représenter un graphe en relations mais cela nécessite de traduire des traversées de chemins en jointures coûteuses). Enfin, les bases de données NoSQL peuvent être sans schéma, ce qui rend quasiment impossible de dériver un schéma global. Enfin, ce dont l'utilisateur a besoin, c'est la capacité d'exprimer des requêtes puissantes pour exploiter

toute la puissance des différentes langages de base de données, par exemple, exprimer directement une traversée de chemin dans une base de données de graphe. Pour cela, nous avons besoin d'un nouveau langage de requête.

Nous pouvons traduire ces observations en cinq besoins principaux pour notre langage commun :

1. Intégrer des requêtes entièrement fonctionnelles sur différentes bases de données NoSQL et SQL à l'aide du mécanisme de requête native de chaque base de données ;

2. Permettre aux requêtes imbriquées d'être arbitrairement chaînées en séquences, le résultat d'une requête (pour une base de données) peut être utilisé comme entrée d'une autre (pour une autre base de données) ;

3. Etre indépendant du schéma, de sorte que les bases de données sans ou avec des schémas différents peuvent être facilement intégrés ;

4. Autoriser des transformations de métadonnées de données, par exemple, convertir les attributs ou les relations en données et vice versa [61] ;

5. Etre facilement optimisable de sorte que l'optimisation efficace des requêtes, introduite dans les systèmes de multi-bases de données, puisse être réutilisée (par exemple, exploiter le bindjoin [32] ou transférer les plus petits résultats intermédiaires).

Le langage CloudMdsQL, et son moteur de requêtes répondent à ces besoins. Alors que les quatre dernièrs besoins ont bien été introduits dans les systèmes multi-bases de données, CloudMdsQL contribue à satisfaire aussi le premier. Le langage est capable d'interroger plusieurs bases de données hétérogènes au sein d'une seule requête contenant des sous-requêtes imbriquées, chacune renvoyant à une base de données particulière et pouvant contenir des invocations incorporées dans l'interface de requête native du système de données local.

La conception du moteur de requête profite du fait qu'il fonctionne dans une plateforme de cloud. Contrairement à l'architecture traditionnelle médiateur-wrapper où le médiateur et les wrappers sont centralisés, le système CLoudMdsQL a une architecture entièrement distribuée qui offre d'importantes possibilités d'optimisation, par exemple, en minimisant les transferts de données entre nœuds. Cela permet de réutiliser les techniques d'optimisation de requêtes qui sont à la base du traitement des requêtes distribuées [46].

Extension de CloudMdsQL avec MFR

CloudMdsQL est capable d'exploiter la pleine puissance des bases de données locales, en permettant simplement à certaines requêtes natives des bases de données locales d'être appelées comme des fonctions et en même temps d'être optimisées, par exemple, en utilisant

les prédicats sélectifs le plus tôt possible, en utilisant le bindjoin, en réalisant l'ordonnancement de jointure ou en planifiant la transmission de données intermédiaire.

Notre principale contribution dans cette thèse est d'étendre CloudMdsQL pour intégrer les données à partir de différentes bases de données, y compris les données non structurées (HDFS) accessibles à travers des frameworks de traitement des données. Cela permet d'effectuer des jointures entre des données relationnelles et HDFS accédé via le framework Spark. Nous définissons une notation simple (dans CloudMdsQL) pour spécifier de manière déclarative la séquence des opérateurs map/filter /reduce (MFR).

Nous supposons que chaque base de données est entièrement autonome, c'est-à-dire que le moteur de requête n'a aucun contrôle sur la structure et l'organisation des données dans les bases de données. Pour cette raison, l'architecture de notre moteur de requête est basée sur l'approche traditionnelle médiateur-wrapper. Toutefois, les utilisateurs doivent être conscients de la façon dont les données sont organisées dans les bases de données, de sorte qu'ils écrivent des requêtes valides. Une requête unique de notre langage peut demander que les données soient récupérées dans deux bases de données, puis une jointure à effectuer sur les jeux de données récupérés. La requête contient donc des invocations intégrées aux bases de données sous-jacentes, exprimées sous forme de sous-requêtes.

Comme notre langage de requête est fonctionnel, il permet un couplage strict entre les données et les fonctions. Une sous-requête, à destination du framework de traitement des données, est représentée par une séquence d'opérations MFR, exprimée dans une notation formelle. D'autre part, SQL est utilisé pour exprimer les sous-requêtes sur les bases de données relationnelles ainsi que la requête principale qui effectue l'intégration des données récupérées par toutes les sous-requêtes. Ainsi, une requête bénéficie à la fois d'une grande expressivité (en permettant l'utilisation ad hoc des opérateurs MFR définis par l'utilisateur en combinaison avec les ordres SQL traditionnels) et l'optimisation (en permettant la réécriture de sous-requête afin que les conditions de filtrage et le bindjoin puissent être poussées et exécutées par la base de données le plus tôt possible).

Prototype

Nous avons développé l'extension MFR comme extension du moteur de requête CloudMdsQL. Chaque nœud du moteur de requête se compose de deux parties - master et worker - et est colocalisée à chaque nœud de la base de données dans un cluster d'ordinateurs. Un nœud master prend comme entrée une requête et produit un plan de requête, qu'il envoie à un nœud du moteur de requête choisi pour son exécution. Il utilise un planificateur de requêtes qui effectue l'analyse et l'optimisation des requêtes et produit un plan de requête sérialisé qui peut être facilement transféré entre les nœuds du moteur de requête. Les workers collaborent pour exécuter un plan de requête, produit par un master, au travers des bases de données sous-jacentes impliquées dans la requête. Chaque nœud worker agit comme un processeur de base de données léger et se compose de plusieurs modules génériques (c'est-à-dire la même bibliothèque de code) : un contrôleur d'exécution de requêtes, un moteur d'opérateurs, un système de stockage de tables, et un module wrapper

spécifique à un système de données.

L'implémentation actuelle du moteur de requête utilise une version modifiée du SGBD Open Source Derby pour accepter les requêtes CloudMdsQL et transformer le plan d'exécution correspondant aux opérations SQL Derby. Pour étendre le moteur de requête CloudMdsQL avec MFR, nous avons développé un planificateur MFR pour être utilisé par le wrapper du framework de traitement de données. Le planificateur MFR trouve des opportunités d'optimisation et traduit la séquence résultante d'opérations MFR à une séquence de méthodes API du framework à exécuter.

Nous avons validé le moteur de requête CloudMdsQL avec des wrappers pour les bases de données suivantes : Sparksee, une base de données graphe avec une API Python ; Derby, une base de données relationnelle accessible via son pilote JDBC ; MongoDB, une base de données documents avec une API Java ; et Apache Spark un framework de traitement de données au-dessus de HDFS, accessibles par une API Apache Spark.

Validation Expérimentale

En nous servant du prototype de moteur de requête CloudMdsQL, nous avons effectué une validation expérimentale du traitement des requêtes multistores dans un cluster pour évaluer l'impact de l'optimisation sur les performances. Plus précisément, nous explorons les avantages de l'utilisation du bindjoin, une technique très efficace, dans des conditions différentes. Dans notre validation expérimentale, nous nous concentrons sur des requêtes qui peuvent exprimer une intégration de données dans plusieurs bases de données, en particulier NoSQL, SGBDR et HDFS accessibles via le framework Spark. D'abord, nous évaluons l'impact de la réécriture et de l'optimisation des requêtes sur le temps d'exécution dans un cluster en utilisant trois bases de données : relationnelle (Derby), document (MongoDB) et graphe (Sparksee).

Nous montrons les temps d'exécution pour 3 plans d'exécutions différents de 5 requêtes. Nous comparons les temps d'exécution, avec différents ordres de jointure, transferts intermédiaires de données et réécritures de sous-requêtes. Nous explorons également les avantages de l'utilisation du bindjoin dans différentes conditions. Les résultats des expériences montrent que le troisième QEP de chaque requête utilisant le bindjoin est bien meilleur que les deux premiers en termes de temps d'exécution.

Ensuite, nous évaluons notre approche MFR dans un cluster Grid5000 avec trois bases de données : PostgreSQL, MongoDB et HDFS. Nous avons validé notre approche en utilisant 3 requêtes différentes, en exécutant chacune avec 3 différentes configurations HDFS pour évaluer le passage à l'échelle. Nous comparons les performances entre les QEP sans bindjoin et les QEP avec bindjoin. Les résultats montrent que le bénéfice de l'optimisation bindjoin est plus élevé dans les configurations avec un nombre plus élevé de nœuds (16 nœuds) en termes de temps d'exécution. De plus, la quantité de données traitées est réduite pendant l'exécution de la séquence MFR en réordonnant les opérateurs MFR selon les règles déterminées.

Dans l'ensemble, notre évaluation des performances illustre la capacité du moteur de

requête CloudMdsQL à optimiser une requête et à bien choisir la stratégie d'exécution la plus efficace.

Conclusion

Dans cette thèse, nous avons abordé le problème du traitement des requêtes avec plusieurs bases de données dans le cloud, où les systèmes de données ont des modèles, des langages et des API différents.

Nous avons proposé une extension du système multistore CloudMdsQL pour tirer pleinement parti des fonctionnalités des frameworks de traitement de données sous-jacents tels que Spark. CloudMdsQL est un langage de requête fonctionnel capable d'exploiter toute la puissance des systèmes de données sous-jascents, en permettant simplement à certaines requêtes natives d'être appelées comme des fonctions et en même temps optimisées, par exemple, en traitant les prédicats sélectifs le plutôt possible, en utilisant le bindjoin, en réalisant l'ordonnancement de jointure ou en planifiant la transmission de données intermédiaires. Notre extension permet l'utilisation des opérateurs MFR définis par l'utilisateur en combinaison avec les ordres SQL traditionnels. Cela permet d'effectuer des jointures entre données relationnelles et HDFS. Notre solution permet l'optimisation en permettant la réécriture de la sous-requête afin que le bindjoin puisse être utilisé et que les conditions de filtrage puissent être poussées et appliquées le plutôt possible par le traitement de données.

Nous avons validé notre solution en implémentant l'extension MFR dans le moteur de requête CloudMdsQL. Sur la base de ce prototype, nous avons fait une validation expérimentale du traitement des requêtes multistore dans un cluster pour évaluer l'impact de l'optimisation sur les performances. Plus précisément, nous explorons les avantages de l'utilisation du bindjoin et de la sélection le plus tôt possible dans différentes conditions. Dans l'ensemble, notre évaluation des performances illustre la capacité du moteur de requête CloudMdsQL à optimiser une requête et à bien choisir la stratégie d'exécution la plus efficace.

La recherche sur les systèmes multistores est relativement récente, avec de nouveaux problèmes. Sur la base de nos contributions, nous pouvons identifier les orientations de recherche suivantes.

Prise en charge des vues multistores

Les vues ont été largement utilisées dans les multi-bases de données pour assurer la transparence à la distribution et l'hétérogénéité, cachant le fait que les données soient stockées dans différents SGBD. L'ajout de vues dans CloudMdsQL faciliterait l'écriture des requêtes sur plusieurs bases de données puisque les expressions de table nécessaires au mappage des données vers le modèle CloudMdsQL seraient capturées par des expressions de vue.

Cela nécessite la définition d'un langage de définition de vues pour CloudMdsQL et

l'adaptation de la réécriture de requêtes pour traiter les vues. Toutefois, comme CloudMdsQL prend en charge les fonctions natives, le traitement des requêtes à l'aide de vues devient plus difficile. Par exemple, une requête CloudMdsQL peut intégrer des données à partir d'une vue multistore MSV, peut-être définie par des fonctions définies par l'utilisateur, et une base de données DS par une fonction définie par l'utilisateur. Mais DS pourrait faire partie de MSV et une simple réécriture de la requête aurait accès à DS deux fois.

Une solution serait d'effectuer une analyse de requête pour découvrir que DS fait partie de MSV et réécrire la requête de sorte que DS soit accédée une seule fois, mais en produisant deux tables (une pour chaque fonction native), qui sont ensuite combinées. Une approche plus complexe consisterait à identifier le sous-ensemble des prédicats dans des fonctions natives qui pourraient être combinées, par ex. en faisant l'union des prédicats simples, et ainsi produire une table unique.

Support des vues matérialisées

Pour accélérer le traitement des requêtes analytiques, les vues multistores peuvent être matérialisées, comme dans les entrepôts de données. Certains systèmes multistores supportent des vues matérialisées, mais d'une manière simple. Par exemple, les vues matérialisées opportunistes d'Odyssey permettent de mettre en cache et de réutiliser les résultats de la requête. Toutefois, les données mises en cache ne sont pas actualisées à la suite des mises à jour des données de base.

Le support des vues matérialisées multistores nécessiterait de traiter le problème de la maintenance de vue, c'est-à-dire de maintenir les données matérialisées en cohérence avec les données de base qui peuvent être mises à jour. Ce problème est traité dans les entrepôts de données à l'aide des SGBDR et des outils Extract-Transform-Load (ETL). Les ETL s'interfacent avec les sources de données et extraient les données mises à jour de plusieurs façons, en fonction des capacités des sources de données, c'est-à-dire des notifications de mise à jour, de l'extraction incrémentale des données de mise à jour ou uniquement de l'extraction complète des données.

Dans le contexte des systèmes multistores, nous devrions étendre les modules wrapper avec la capacité d'extraction des ETL. Le problème principal est que les bases de données non relationnelles ne supportent généralement pas les notifications de mise à jour et que l'extraction complète peut constituer l'option courante et coûteuse, ce qui nécessite de comparer chaque extraction avec la précédente pour identifier les modifications.

Une solution au problème de la maintenance de vue matérialisée consiste à calculer la vue de façon incrémentalle, en utilisant des tables différentielles qui capturent les données mises à jour. Nous pourrions adapter cette solution dans le contexte des systèmes multistores, dans le cas simple des définitions de vue non récursives en adaptant des algorithmes efficaces comme le célèbre algorithme par comptage [31].

Traitement parallèle des requêtes multistores

Pour réussir les analyses de données massives, il est essentiel d'intégrer des données provenant de plusieurs bases de données. Cela peut être fait en étendant un moteur OLAP parallèle (OPE) avec les capacités de CloudMdsQL. Les OPE exploitent généralement le parallélisme intra-requête (à la fois inter-opération et intra-opération) pour obtenir de hautes performances.

étant donné un OPE avec des capacités CloudMdsQL, le problème est d'intégrer (par exemple joindre) de grandes masses de données à partir d'une ou plusieurs bases de données dans l'OPE, d'une manière qu'il exploite le parallélisme. Cependant, bien que toutes les bases de données que nous avons utilisées fournissent un support pour le partitionnement des données et le traitement parallèle, le moteur CloudMdsQL n'utilise pas le parallélisme. C'est parce que nous avons un wrapper, un moteur d'opérateurs et un client de base de données sur un serveur, donc une requête lourde à une base de données produira de grandes quantités de données qui seront centralisées sur ce serveur.

Une approche prometteuse consiste à introduire le parallélisme entre les requêtes dans le moteur CloudMdsQL. Cela nécessite la capacité des wrappers d'accéder directement aux partitions des bases de données (aux nœuds de données, et non aux nœuds principaux). Selon les bases de données, ceci peut être plus ou moins difficile. Par exemple, HDFS facilite la tâche en fournissant un accès direct aux chunks de données. Mais les SGBDR obligent généralement à accéder aux données partitionnées par l'intermédiaire d'un nœud central maître.

Gestion des flux de données

Les moteurs de traitement de flux (SPE) deviennent omniprésents pour le traitement et l'analyse des données en temps réel. Contrairement aux bases de données, de nouvelles données sont générées continuellement dans un ordre fixe (par exemple, par l'heure d'arrivée ou par un horodatage ajouté aux données) et traitées à la volée à l'aide de requêtes continues. Cependant, il est également nécessaire de combiner des données en continu avec des données stockées dans de multiples systèmes. Ainsi, une direction importante de la recherche dans BigDAWG, est de coupler une SPE avec un système multistore.

Ensuite, le problème majeur est de supporter le traitement des données en temps réel sur les données en temps réel et stockées.

L'approche utilisée dans BigDAWG est de fournir un îlot de flux de données qui est pris en charge par le SPE S-Store [18] qui est fortement couplé avec les autres bases de données. Les requêtes S-Store reposent sur des opérateurs pour la gestion de flux bien connus, comme le filtrage, la jointure, l'agrégat et la fenêtre glissante. Cette approche pourrait être adaptée pour étendre le système multistore CloudMdsQL avec des données en continu. Cela nécessite d'étendre le modèle de données CloudMdsQL avec des données ordonnées et des opérateurs de diffusion en continu. Pour supporter le traitement des données en temps réel sur les données stockées, nous pourrions alors exploiter des vues matérialisées stockées en mémoire et le parallélisme.

Benchmarking des systèmes multistores

A mesure que les systèmes multistores arrivent à maturité, le besoin de benchnmarks augmente nécessairement. Une première étape dans cette direction est le benchmark du système CloudMdsQL [39]. Ce benchmark utilise le standard benchmark TPC H (`www.tpc.org/tpch`) avec 8 jeux de données répartis sur 5 bases de données différentes, chacune ayant des interfaces et des modèles de données différents : relationnel, clé-valeur, document, et un moteur OLAP. Les requêtes TPC H et les cas de test sont divisés en deux groupes qui mettent l'accent sur les performances grâce au bindjoin et au support des requêtes natives.

Le benchmark CloudMdsQL se concentre principalement sur les opérateurs relationnels et leurs équivalents dans les bases de données non-SQL. Comme travail futur, il faudra l'étendre pour évaluer les avantages de performances dans le contexte d'opérateurs spécifiques de base de données, tels que les parcours de graphe ou les requêtes sur des documents. Ensuite, on pourrait généraliser pour comparer les performances de différents systèmes multistores.

Publications

Les contributions de cette thèse ont conduit aux publications suivantes :

- *Carlyna Bondiombouy*, Patrick Valduriez. Query Processing in Multistore Systems : an overview. International Journal of Cloud Computing, $5(4) : 309 - 346$, 2016.
- Boyan Kolev, *Carlyna Bondiombouy*, Patrick Valduriez, Ricardo Jiménez-Peris, Raquel Pau, José Pereira. Demonstration of the CloudMdsQL Multistore System. BDA'2016 : Gestion de données - principes, technologies et applications, 2016, 14.
- *Carlyna Bondiombouy*, Boyan Kolev, Patrick Valduriez, Oleksandra Levchenko. Multistore Big Data Integration with CloudMdsQL. BDA'2016 : Gestion de données - principes, technologies et applications, 2016, 5.
- Boyan Kolev, Patrick Valduriez, *Carlyna Bondiombouy*, Ricardo Jiménez-Peris, Raquel Pau, José Pereira. CloudMdsQL : querying heterogeneous cloud data stores with a common language. Distributed and Parallel Databases, $34(4) : 463 - 503$, 2016.
- *Carlyna Bondiombouy*, Boyan Kolev, Oleksandra Levchenko, Patrick Valduriez. Multistore Big Data Integration with CloudMdsQL. Transactions Large-Scale Data- and Knowledge-Centered Systems, $28 : 48 - 74$, 2016.
- Boyan Kolev, *Carlyna Bondiombouy*, Oleksandra Levchenko, Patrick Valduriez, Ricardo Jiménez-Peris, Raquel Pau, José Orlando Pereira. Design and Implementation of the CloudMdsQL Multistore System. International Conference on Cloud Computing and Services Science (CLOSER), volume $1 : 352 - -359$, 2016.

- Boyan Kolev, *Carlyna Bondiombouy*, Patrick Valduriez, Ricardo Jiménez-Peris, Raquel Pau, José Pereira. The CloudMdsQL Multistore System. ACM SIGMOD International Conference on Data Management, $2113 - -2116$, 2016.

- *Carlyna Bondiombouy*, Boyan Kolev, Oleksandra Levchenko, Patrick Valduriez. Integrating Big Data and Relational Data with a Functional SQL-like Query Language. Database and Expert Systems Applications (DEXA), $1 : 170 - -185$, 2015.

- *Carlyna Bondiombouy*. Query Processing in Cloud Multistore Systems. BDA' 2015 : Gestion de données - principes, technologies et applications, 2015, 2.

Contents

Acknowledgments **iii**

Résumé **v**

Abstract **vii**

Résumé Étendu **xi**

1 Introduction **1**
1.1 Thesis Context 1
1.2 Contributions 2
1.3 Organization of the Thesis 5

2 Overview of Query Processing in Multistore Systems **7**
2.1 Introduction 7
2.2 Cloud Data Management 8
2.2.1 Distributed Storage 10
2.2.1.1 Block-based Distributed File Systems 11
2.2.1.2 Object-based Distributed File Systems 12
2.2.1.3 Combining Block Storage and Object Storage 13
2.2.2 NoSQL Systems 14
2.2.2.1 Key-Value Stores 14
2.2.2.2 Wide Column Stores 15
2.2.2.3 Document Stores 16
2.2.2.4 Graph Data stores 17
2.2.3 Data Processing Frameworks 18
2.2.3.1 Concluding Remarks 20
2.3 Multidatabase Query Processing 21
2.3.1 Mediator-Wrapper Architecture 21
2.3.2 Multidatabase Query Processing Architecture 22
2.3.3 Multidatabase Query Processing Techniques 23
2.3.3.1 Heterogeneous Cost Modeling 23
2.3.3.2 Heterogeneous Query Optimization 24
2.3.3.3 Adaptive Query Processing 26

2.4 Multistore Systems . . . 27
2.4.1 Loosely-Coupled Multistore Systems . . . 27
2.4.2 Tightly-Coupled Multistore Systems . . . 31
2.4.3 Hybrid systems . . . 35
2.4.4 Comparative Analysis . . . 38
2.5 Conclusion . . . 40

3 Design of CloudMdsQL 41
3.1 Overview . . . 41
3.2 Related Work . . . 42
3.3 Basic Concepts . . . 44
3.3.1 Data Model . . . 44
3.3.2 Language Concepts . . . 45
3.4 Query Engine Architecture . . . 46
3.4.1 Overview . . . 46
3.4.2 Master . . . 48
3.4.3 Worker . . . 49
3.5 Query Language . . . 50
3.5.1 Named Table Expressions . . . 50
3.5.2 Nested Queries . . . 52
3.5.2.1 Within SQL Expressions . . . 52
3.5.2.2 Within Native Expressions . . . 53
3.5.3 CloudMdsQL SELECT Statement . . . 55
3.6 Query Processing . . . 57
3.6.1 Query Decomposition . . . 57
3.6.2 Query Optimization . . . 58
3.6.3 Query Execution . . . 60
3.6.4 Interfacing Data Stores . . . 61
3.6.4.1 Querying SQL Compatible NoSQL Data Stores . . . 62
3.6.4.2 SQL Capabilities . . . 62
3.6.4.3 Using Native Queries . . . 64
3.7 Use Case Example . . . 65
3.8 Conclusion . . . 72

4 Extending CloudMdsQL with MFR 73
4.1 Overview . . . 73
4.2 Query Language . . . 75
4.2.1 MFR Notation . . . 75
4.2.2 Combining SQL and MFR . . . 77
4.3 Query Engine Architecture . . . 78
4.4 Query Processing . . . 80
4.4.1 Query Optimization . . . 80
4.4.2 MFR Rewrite Rules . . . 81

4.4.3 Bind Join . . . 82
4.5 Use Case Example . . . 83
4.6 Conclusion . . . 89

5 Prototype **91**
5.1 Overview . . . 91
5.2 Query Planner . . . 92
5.3 Execution Engine . . . 94
5.4 Wrappers . . . 96
5.5 Conclusion . . . 96

6 Experimental Validation **99**
6.1 Experimental Setup . . . 99
6.2 CloudMdsQL Experimentation . . . 100
6.2.1 Datasets . . . 100
6.2.2 Experimental Results . . . 101
6.3 MFR Experimentation . . . 105
6.3.1 Datasets . . . 105
6.3.2 Experimental Results . . . 106
6.4 Conclusion . . . 109

7 Conclusion **111**
7.1 Contributions . . . 111
7.2 Directions for Future Work . . . 113

Bibliography **117**

Chapter 1

Introduction

Cloud computing is having a major impact on data management, with a proliferation of new scalable, solutions such as distributed file and object storage, NoSQL databases and big data processing frameworks. These solutions have been the basis for a rich offering of services (IaaS, PaaS, SaaS, DaaS, etc.) that can be used to build cloud data-intensive applications. However, this has also led to a wide diversification of DBMS interfaces and the loss of a common programming paradigm.

For cloud data management, one could rely on RDBMSs as most of them have a distributed and parallel version. However, RDBMSs have been lately criticized for their "one size fits all" approach [54]. Although they have been able to integrate support for all kinds of data (e.g. multimedia objects, XML documents) and new functions, this has resulted in a loss of performance, simplicity and flexibility for applications with specific, tight performance requirements. Therefore, it has been argued that more specialized DBMS engines are needed.

This makes it very hard for a user to integrate its data sitting in specialized data stores, e.g. relational, documents and graph data stores. For example, consider a user who gives a relational data store with authors, a document store with reviews, and a graph data store with author friendships, wants to find out about conflicts of interests in the reviewing of some papers. The main solution today would be to provide a program (e.g. written in Java) that accesses the three data stores through their APIs and integrates the data (in memory). This solution is obviously labor-intensive, complex and not easily extensible (e.g. to deal with a new data store).

1.1 Thesis Context

This thesis has been prepared in the context of the CoherentPaaS European project [1]. This project had to deal with two issues faced by current cloud data store management frameworks: the loss of data consistency due to the lack of transactions, and the fact that queries across data stores need be programmed and optimized manually. CoherentPaaS addresses this problem by providing a rich PaaS with a coherent, ultra-scalable, and efficient integration of NoSQL, SQL and Complex Event Processing (CEP) data management

technologies.

However, unlike in the current cloud landscape, CoherentPaaS provides a common programming model and language to query multiple data stores. The platform is designed to allow different subsets of enterprise data to be materialized within different data models, so that each subset is handled in the most efficient way according to its most common data access patterns. On the other hand, an application can still access a data store directly, without using our query engine. This constitutes a multiple data store systems with high levels of heterogeneity and local autonomy.

In this thesis, we focus on the problem of efficient query processing of heterogeneous data stores with a common language. The problem can be expressed as follows: Let $Q(S1, S2, \ldots Sn)$ be an query, over n data stores, each with a different data model and query language, and in some cases (e.g. a document store, a graph data store) a different API, propose an approach to translate Q into an optimized query execution plan (QEP), with efficient management of intermediate results.

In order to address these challenges, CoherentPaaS proposes a multistore system with a functional language (CloudMdsQL), for querying multiple heterogeneous data stores using nested queries. A CloudMdsQL query can exploit the full power of the local data stores, by simply allowing some local data store native queries to be called as functions, and at the same time be optimized based on a simple cost model. The architecture of the query engine is fully distributed, so that query engine nodes can directly communicate with each other, by exchanging code (query plans) and data. This fully distributed architecture provides important opportunities for optimization, in particular, minimizing data shipping between nodes, select predicate pushdown, bind join, and join ordering while reducing execution time, communication cost and network traffic. These optimization opportunities are exploited by the CloudMdsQL compiler.

In the context of multistore systems, much attention is being paid on the integration of unstructured big data typically stored in HDFS with relational data. One main solution is to use a relational query engine that allows SQL-like queries to retrieve data from HDFS. This solution is used for instance in Polybase from Microsoft to integrate HDFS data within SQL Server Data Warehouse. However, it requires the system to provide a relational view of the unstructured data and hence is not always feasible. In this thesis, we propose an extension of CloudMdsQL to take full advantage of the functionality of the underlying data processing frameworks by allowing the ad-hoc usage of user defined map/filter/reduce (MFR) operators in combination with traditional SQL statements. Our solution allows for optimization by enabling subquery rewriting so that bind join can be used and filter conditions can be pushed down and applied by the data processing framework as early as possible.

1.2 Contributions

The main contributions of this thesis are:

- **An overview of query processing in multistore systems.** We give an overview of

query processing in multistore systems. We start by introducing the recent cloud data management solutions and query processing in multidatabase systems. Then, we describe and analyze some representative multistore systems, based on their architecture, data model, query languages and query processing techniques. To ease comparison, we divide multistore systems based on the level of coupling with the underlying data stores, i.e., loosely-coupled, tightly-coupled and hybrid. Our analysis reveals some important trends, which we discuss.

- **The extension of CloudMdsQL with MFR notation.** CloudMdsQL is a functional query language able to exploit the full power of local data stores, by simply allowing some local data store native queries to be called as functions, and at the same time be optimized, e.g. by pushing down select predicates, using bind join, performing join ordering, or planning intermediate data shipping. Our main contribution is to extend CloudMdsQL to integrate data retrieved from different data stores, including unstructured (HDFS) data accessed through a data processing frameworks. It allows performing joins between relational and HDFS data, and extend the query engine with operators for the Spark framework. We define a simple notation (in CloudMdsQL) to specify in a declarative way the sequence of map/filter/reduce (MFR) operators. We exploit the full power of the frameworks , yet avoiding the use of SQL engines on top of them. Furthermore, we allow for optimization by enabling subquery rewriting so that bind join can be used and filter conditions can be pushed down and applied by the data processing framework as early as possible.

- **Prototype.** We have developed the MFR extension as part of the CloudMdsQL query engine. Each query engine node consists of two parts – master and worker – and is collocated at each data store node in a computer cluster. A master node takes as input a query and produces a query plan, which it sends to one chosen query engine node for execution. It uses a query planner that performs query analysis and optimization, and produces a query plan serialized that can be easily transferred across query engine nodes. Workers collaborate to execute a query plan, produced by a master, against the underlying data stores involved in the query. Each worker node acts as a lightweight runtime database processor atop a data store and is composed of three generic modules (i.e. same code library) - query execution controller, operator engine, and table storage - and one wrapper module that is specific to a data store. The current implementation of the query engine uses a modified version of the open source Derby database to accept CloudMdsQL queries and transform the corresponding execution plan into Derby SQL operations. To extend the CloudMdsQL query engine with MFR, we developed an MFR planner to be used by the wrapper of the data processing framework (DPF). The MFR planner finds optimization opportunities and translates the resulting sequence of MFR operations to a sequence of DPF's API methods to be executed.

- **Experimental validation.** Based on the CloudMdsQL query engine prototype, we

give an experimental validation of multistore query processing in a cluster to evaluate the impact on performance of optimization. More specifically, we explore the performance benefit of using bind join, a very efficient technique, under different conditions. In our experimental validation, we focus on queries that can express an integration of data across several data stores, in particular, NoSQL (graph and document data stores), RDBMS and HDFS accessed through the Spark framework. Overall, the experimental results confirm the importance of using simple optimization techniques to reduce query execution times.

These contributions have led to the following publications:

- *Carlyna Bondiombouy*, Patrick Valduriez. Query Processing in Multistore Systems: an overview. International Journal of Cloud Computing, $5(4) : 309 - 346$, 2016.
- Boyan Kolev, *Carlyna Bondiombouy*, Patrick Valduriez, Ricardo Jiménez-Peris, Raquel Pau, José Pereira. Demonstration of the CloudMdsQL Multistore System. BDA'2016: Gestion de données - principes, technologies et applications, 2016, 14.
- *Carlyna Bondiombouy*, Boyan Kolev, Patrick Valduriez, Oleksandra Levchenko. Multistore Big Data Integration with CloudMdsQL. BDA'2016: Gestion de données - principes, technologies et applications, 2016, 5.
- Boyan Kolev, Patrick Valduriez, *Carlyna Bondiombouy*, Ricardo Jiménez-Peris, Raquel Pau, José Pereira. CloudMdsQL: querying heterogeneous cloud data stores with a common language. Distributed and Parallel Databases, $34(4) : 463 - 503$, 2016.
- *Carlyna Bondiombouy*, Boyan Kolev, Oleksandra Levchenko, Patrick Valduriez. Multistore Big Data Integration with CloudMdsQL. Transactions Large-Scale Data- and Knowledge-Centered Systems, $28 : 48 - 74$, 2016.
- Boyan Kolev, *Carlyna Bondiombouy*, Oleksandra Levchenko, Patrick Valduriez, Ricardo Jiménez-Peris, Raquel Pau, José Orlando Pereira. Design and Implementation of the CloudMdsQL Multistore System. International Conference on Cloud Computing and Services Science (CLOSER), $1 : 352 - -359$, 2016.
- Boyan Kolev, *Carlyna Bondiombouy*, Patrick Valduriez, Ricardo Jiménez-Peris, Raquel Pau, José Pereira. The CloudMdsQL Multistore System. ACM SIGMOD International Conference on Data Management, $2113 - -2116$, 2016.
- *Carlyna Bondiombouy*, Boyan Kolev, Oleksandra Levchenko, Patrick Valduriez. Integrating Big Data and Relational Data with a Functional SQL-like Query Language. Database and Expert Systems Applications (DEXA), $1 : 170 - -185$, 2015.
- *Carlyna Bondiombouy*. Query Processing in Cloud Multistore Systems. BDA'-2015: Gestion de donnees - principes, technologies et applications, 2015, 2.

1.3 Organization of the Thesis

The remaining chapters of the thesis are organized as follows.

Chapter 2: State of The Art

In this chapter, we review the state of the art of query processing in multistore systems. First, we present different cloud data management solutions, including distributed file systems, NoSQL systems and data processing systems. Then, we review the main query processing techniques from multidatabase systems since they also deal with heterogeneous data stores. Finally, we introduce the three kinds of multistore systems (Loosely-coupled, Tightly-coupled and Hybrid systems), compare the functionality and implementations techniques of each system.

Chapter 3: Design of CloudMdsQL

In this chapter, we describe the CloudMdsQL language and its query engine. First, we discuss the state of the art solutions for multidatabase systems, and we analyze the limitations of the existing approaches. Then, we describe the main concepts of CloudMdsQL, including the data model and the language. Afterward, we present the architecture of the query engine and the principal components. We present in detail the different query processing steps of CloudMdsQL according to the query engine architecture. Finally, we illustrate the query processing steps with a use case.

Chapter 4: Extending CloudMdsQL with MFR

In this chapter, we present our extension of CloudMdsQL to express subqueries that can take full advantage of the functionality of a data processing framework when accessing unstructured data stores such as HDFS. First, we present our approach. Next, we introduce the language and the MFR notation. Then, we describe the query engine architecture of our system. Afterward, we present the query processing steps and show the MFR rewrite rules. Finally, we illustrate our solution with a use case example.

Chapter 5: Prototype

In this chapter, we describe the implementation of the CloudMdsQL query engine and our MFR extension. First, we describe the query planner component, which compiles a CloudMdsQL query and generates a query execution plan to be processed by the query engine. Then, we present the execution engine with its modules (query execution controller, operator engine, table storage and wrappers). Finally, we describe the specific wrappers that have been implemented in the CoherentPaaS project to interface with data stores.

Chapter 6: Experimental Validation

This chapter present our experimental validation of CloudMdsQL and our MFR extension. First, we describe our experimental setup. Then, we give an experimental validation of CloudMdsQL. Finally, we present the experimental validation of the MFR extension.

Chapter 7: Conclusion

In this chapter, we summarize and discuss the main contributions of this thesis. We also give some directions for future work.

Chapter 2

Overview of Query Processing in Multistore Systems

Building cloud data-intensive applications often requires using multiple data stores (HDFS, NoSQL, RDBMS, etc.), each optimized for one kind of data and tasks. However, the wide diversification of data store interfaces makes it difficult to access and integrate data from multiple data stores. This important problem has motivated the design of a new generation of systems, called multistore systems, which provide integrated or transparent access to a number of cloud data stores through one or more query languages. In this chapter, we give an overview of query processing in multistore systems. The objective is not to give an exhaustive survey of all systems and techniques, but to focus on the main solutions and trends, based on the study of nine representative systems (3 for each class namely: loosely-coupled, tightly-coupled, ans hybrid systems). This chapter is based on [15].

This chapter is organized as follows. Section 2.1 introduces the chapter and a classification of multistore systems. In Section 2.2, we introduce the recent cloud data management, including distributed file systems, NoSQL systems and data processing frameworks. In Section 2.3, we review the main query processing techniques for multidatabase systems, based on the mediator-wrapper architecture. Finally, in Section 2.4, we analyze the three kinds of multistore systems, based on their architecture, data model, query languages and query processing techniques. Section 2.5 concludes and discusses open issues.

2.1 Introduction

The problem of accessing heterogeneous data stores, i.e. managed by different data management systems such as RDBMS and XML DBMS, has long been studied in the context of multidatabase systems [46] (also called federated database systems, or more recently data integration systems [24]). Most of the work on multidatabase query processing has been done in the context of the mediator-wrapper architecture, using a declarative, SQL-like language. The mediator-wrapper architecture allows dealing with three major proper-

ties of the data stores: distribution (i.e. located at different sites), heterogeneity (i.e. with different data models and languages) and autonomy (i.e. under local control) [46].

The state-of-the-art solutions for multidatabase query processing can be useful to transparently access multiple data stores in the cloud. However, operating in the cloud makes it quite different from accessing data stores on a wide-area network or the Internet. First, the kinds of queries are different. For instance, a web data integration query, e.g. from a price comparator, could access lots of similar web data stores, whereas a cloud query should be on a few but quite different cloud data stores and the user needs to have access rights to each data store. Second, both mediator and data source wrappers can only be installed at one or more servers that communicate with the data stores through the network. However, operating in a cloud, where data stores are typically distributed over the nodes of a computer cluster, provides more control over where the system components can be installed and thus, more opportunities to design an efficient architecture.

These differences have motivated the design of more specialized *multistore systems* [40] (also called polystores [25]) that provide integrated access to a number of cloud data stores through one or more query languages. Several multistore systems are being built, with different objectives, architectures and query processing approaches, which makes it hard to compare them. To ease comparison, we divide multistore systems based on the level of coupling with the underlying data stores, i.e. loosely-coupled, tightly-coupled and hybrid.

Loosely-coupled systems are reminiscent of multidatabase systems in that they can deal with autonomous data stores, which can then be accessed through the multistore system common language as well as separately through their local language.

Tightly-coupled systems trade autonomy for performance, typically in a shared-nothing cluster, so that data stores can only be accessed through the multistore system, directly through their local language.

Hybrid systems tightly-couple some data stores, typically an RDBMS, and loosely-couple some others, typically HDFS through a data processing framework like MapReduce or Spark.

2.2 Cloud Data Management

Some important characteristics of cloud data have been considered for designing data management solutions. Cloud data can be very large, unstructured or semi-structured, and typically append-only (with rare updates). And cloud users and application developers may be in high numbers, but not DBMS experts. Therefore, current cloud data management solutions have traded ACID (Atomicity, Consistency, Isolation and Durability) transactional properties for scalability, performance, simplicity and flexibility [49].

The preferred approach of cloud providers is to exploit a shared-nothing cluster [46], i.e. a set of loosely connected computer servers with a very fast, extensible interconnect (e.g. Infiniband). When using commodity servers with internal direct-attached storage, this approach provides scalability with excellent performance-cost ratio. Compared to

traditional DBMSs, cloud data management uses a different software stack with the following layers: distributed storage, database management and distributed processing. In the rest of this section, we introduce this software stack and present the different layers in more details.

Cloud data management (see Figure 2.1) relies on a distributed storage layer, whereby data is typically stored in files or objects distributed over the nodes of a shared-nothing cluster. This is one major difference with the software stack of current DBMSs that relies on block storage. Interestingly, the software stack of the first DBMSs was not very different from that used now in the cloud. The history of DBMSs is interesting to understand the evolution of this software stack. The very first DBMSs, based on the hierarchical or network models, were built as extensions of a file system, such as COBOL, with inter-file links. And the first RDBMSs too were built on top of a file system. For instance, the famous Ingres RDBMS [55] was implemented atop the Unix file system. But using a general-purpose file system was making data access quite inefficient, as the DBMS could have no control over data clustering on disk or cache management in main memory. The main criticism for this file-based approach was the lack of operating system support for database management (at that time) [53]. As a result, the architecture of RDBMSs evolved from file-based to block-based, using a raw disk interface provided by the operating system. A block-based interface provides direct, efficient access to disk blocks (the unit of storage allocation on disks). Today all RDBMSs are block-based, and thus have full control over disk management.

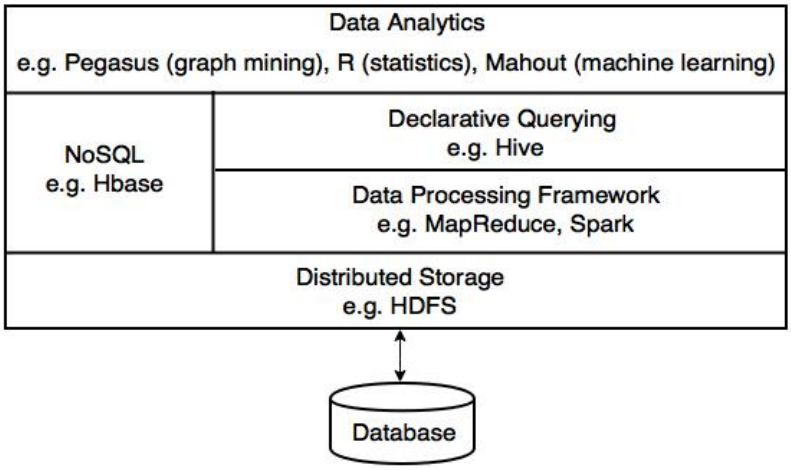


Figure 2.1 – Cloud data management software stack

The evolution towards parallel DBMSs kept the same approach, in particular, to ease the transition from centralized systems. Parallel DBMSs use either a shared-nothing or shared-disk architecture. With shared-nothing, each node (e.g. a server in a cluster) has exclusive access to its local disk through internal direct-attached storage. Thus, big relational tables need be partitioned across multiple disks to favor parallel processing. With shared-disk, the disks are shared among all nodes through a storage area network, which eases parallel processing. However, since the same disk block can be accessed in

concurrent mode by multiple cluster nodes, a distributed lock manager [46] is necessary to avoid write conflicts and provide cache coherency. In either architecture, a node can access blocks either directly through direct-attached storage (shared-nothing) or via the storage area network (shared-disk).

In the context of cloud data management, we can identify two main reasons why the old DBMS software stack strikes back. First, distributed storage can be made fault-tolerant and scalable (e.g. HDFS), which makes it easier to build the upper data management layers atop (see Figure 2.1). Second, in addition to the NoSQL layer (e.g. Hbase over HDFS), data stored in distributed files can be accessed directly by a data processing framework (e.g. MapReduce or Spark), which makes it easier for programmers to express parallel processing code. The distributed processing layer can then be used for declarative (SQL-like) querying, e.g. with a framework like Hive over MapReduce. Finally, at the top layer, tools such as Pegasus (graph mining), R (statistics) and Mahout (machine learning) can be used to build more complex big data analytics.

2.2.1 Distributed Storage

The distributed storage layer of a cloud typically provides two solutions to store data, files or objects, distributed over cluster nodes. These two solutions are complementary, as they have different purposes and they can be combined.

File storage manages data within unstructured files (i.e. sequences of bytes) on top of which data can be organized as fixed-length or variable-length record. A file system organizes files in a directory hierarchy, and maintains for each file its metadata (file name, folder position, owner, length of the content, creation time, last update time, access permissions, etc.), separate from the content of the file. Thus, the file metadata must first be read for locating the file's content. Because of such metadata management, file storage is appropriate for sharing files locally within a cloud data center and when the number of files are limited (e.g. in the hundreds of thousands). To deal with big files that may contain high numbers of records, files need be partitioned and distributed, which requires a distributed file system.

Object storage manages data as objects. An object includes its data along with a variable amount of metadata, and a unique identifier in in a flat object space. Thus, an object can be represented as a triple (oid, data, metadata), and once created, it can be directly accessed by its oid. The fact that data and metadata are bundled within objects makes it easy to move objects between distributed locations. Unlike in file systems where the type of metadata is the same for all files, objects can have variable amounts of metadata. This allows much user flexibility to express how objects are protected, how they can be replicated, when they can be deleted, etc. Using a flat object space allows managing massive amounts e.g. billions or trillions) of unstructured data. Finally, objects can be easily accessed with a simple REST-based API with put and get commands easy to use on Internet protocols. Object stores are particularly useful to store a very high number of relatively small data objects, such as photos, mail attachments, etc. Therefore, most cloud providers leverage an object storage architecture, e.g. Amazon Web Services S3, Rackspace Files,

Microsoft Azure Vault Storage and Google Cloud Storage.

Distributed file systems in the cloud can then be divided between block-based, extending a traditional file system, and object-based, leveraging an object store. Since these are complementary, there are also systems that combine both. In the rest of this section, we illustrate these three categories with representative systems.

2.2.1.1 Block-based Distributed File Systems

One of the most influential systems in this category is Google File System (GFS). GFS [29] has been developed by Google (in C++ on top of Linux) for its internal use. It is used by many Google applications and systems, such as Bigtable and MapReduce, which we discuss next.

Similar to other distributed file systems, GFS aims at providing performance, scalability, fault-tolerance and availability. However, the targeted systems, shared-nothing clusters, are challenging as they are made of many (e.g. thousands of) servers built from inexpensive hardware. Thus, the probability that any server fails at a given time is high, which makes fault-tolerance difficult. GFS addresses this problem. It is also optimized for Google data-intensive applications, such as search engine or data analysis. These applications have the following characteristics. Firstly, their files are very large, typically several gigabytes, containing many objects such as web documents. Secondly, workloads consist mainly of read and append operations, while random updates are rare. Read operations consist of large reads of bulk data (e.g. 1 MB) and small random reads (e.g. a few KBs). The append operations are also large and there may be many concurrent clients that append the same file. Thirdly, because workloads consist mainly of large read and append operations, high throughput is more important than low latency.

GFS organizes files as a tree of directories and identifies them by pathnames. It provides a file system interface with traditional file operations (create, open, read, write, close, and delete file) and two additional operations: snapshot and record append. Snapshot allows creating a copy of a file or of a directory tree. Record append allows appending data (the "record") to a file by concurrent clients in an efficient way. A record is appended atomically, i.e. as a continuous byte string, at a byte location determined by GFS. This avoids the need for distributed lock management that would be necessary with the traditional write operation (which could be used to append data).

The architecture of GFS is illustrated in Figure 2.2. Files are divided into fixed-size partitions, called *chunks*, of large size, i.e. 64 MB. The cluster nodes consist of GFS clients that provide the GFS interface to applications, chunk servers that store chunks and a single GFS master that maintains file metadata such as namespace, access control information, and chunk placement information. Each chunk has a unique id assigned by the master at creation time and, for reliability reasons, is replicated on at least three chunk servers (in Linux files). To access chunk data, a client must first ask the master for the chunk locations, needed to answer the application file access. Then, using the information returned by the master, the client can request the chunk data to one of the replicas.

This architecture using single master is simple. And since the master is mostly used

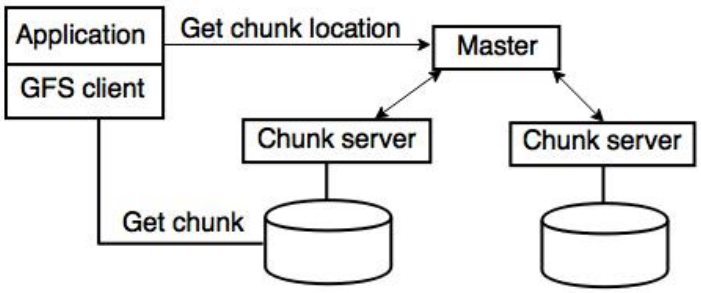


Figure 2.2 – GFS architecture

for locating chunks and does not hold chunk data, it is not a bottleneck. Furthermore, there is no data caching at either clients or chunk servers, since it would not benefit large reads. Another simplification is a relaxed consistency model for concurrent writes and record appends. Thus, the applications must deal with relaxed consistency using techniques such as checkpointing and writing self-validating records. Finally, to keep the system highly available in the face of frequent node failures, GFS relies on fast recovery and replication strategies.

There are open source implementations of GFS, such as Hadoop Distributed File System (HDFS), a popular Java product. HDFS has been initially developed by Yahoo and is now the basis for the successful Apache Hadoop project, which together with other products (MapReduce, Hbase) has become a standard for big data processing. There are other important open source block-based distributed file systems for cluster systems, such as GlusterFS for shared-nothing and Global File System 2 (GFS2) for shared-disk, both being now developed by Red Hat for Linux.

2.2.1.2 Object-based Distributed File Systems

One of the first systems in this category is Lustre, an open source file system [59]. Lustre was initially developed (in C) at Carnegie Mellon University in the late 1990s, and has become very popular in High Performance Computing (HPC) and scientific applications in the cloud, e.g. Intel Cloud Edition for Lustre. The architecture of the Lustre file system has three main components:

- One or more metadata servers that store namespace metadata, such as filenames, directories, access permissions, etc. Unlike block-based distributed file systems, such as GFS and HDFS, where the metadata server controls all block allocations, the Lustre metadata server is only involved when opening a file and is not involved in any file I/O operations, thus avoiding scalability bottlenecks.

- One or more object storage servers that store file data on one or more object storage targets (OSTs). An object storage server typically serves between two and eight OSTs, with each OST managing a single local disk file system.

- Clients that access and use the data. Lustre presents all clients with a unified namespace for all of the files and data, using the standard file system interface, and allows concurrent and coherent read and write access to the files in the file system.

These three components can be located at different server nodes in a shared-disk cluster, with disk storage connected to the servers using storage area network. Clients and servers are connected with the Lustre file system using a specific communication infrastructure called Lustre Networking (LNET). Lustre provides cache consistency of files' data and metadata by a distributed lock manager. Files can be partitioned using *data striping*, a technique that segments logically sequential data so that consecutive segments are stored on different disks. This is done by distributing objects across a number of object storage servers and OSTs. To provide data reliability, objects in OSTs are replicated using primary-copy replication and RAID6 disk storage technology.

When a client accesses a file, it completes a filename lookup on the metadata server and gets back the layout of the file. Then, to perform read or write operations on the file, the client interprets the layout to map the operation to one or more objects, each residing on a separate OST. The client then locks the file range being operated on and executes one or more parallel read or write operations directly to the OSTs. Thus, after the initial lookup of the file layout, unlike with block-based distributed file systems, the metadata server is not involved in file accesses, so the total bandwidth available for the clients to read and write data scales almost linearly with the number of OSTs in the file system.

Another popular open source object-based distributed file system is XtreemFS [36]. XtreemFS is highly fault-tolerant, handling all failure modes including network splits, and highly-scalable, allowing objects to be partitioned or replicated across shared-nothing clusters and data centers.

2.2.1.3 Combining Block Storage and Object Storage

An important trend for data management in the cloud is to combine block storage and object storage in a single system, in order to support both large files and high numbers of objects. The first system that combined block and object storage is Ceph [58]. Ceph is an open source software storage platform, now developed by Red Hat, which combines object, block, and file storage in a shared-nothing cluster at exabyte scale. Ceph decouples data and metadata operations by eliminating file allocation tables and replacing them with data distribution functions designed for heterogeneous and dynamic clusters of unreliable object storage devices (OSDs). This allows Ceph to leverage the intelligence present in OSDs to distribute the complexity surrounding data access, update serialization, replication and reliability, failure detection, and recovery. Ceph and GlusterFS are now the two major storage platforms offered by Red Hat for shared-nothing clusters.

HDFS, on the other hand, has become the De facto standard for scalable and reliable file system management for big data. Thus, there is much incentive to add object storage capabilities to HDFS, in order to make data storage easier for cloud providers and users. In Azure HDInsight, Microsoft's Hadoop-based solution for big data management in the cloud, HDFS is integrated with Azure Blob storage, the object storage manager, to operate

directly on structured or unstructured data. Blob storage containers store data as key-value pairs, and there is no directory hierarchy.

Hortonworks, a distributor of Hadoop software for big data, has recently started a new initiative called Ozone, an object store that extends HDFS beyond a file system, toward a more complete storage layer. Similar to GFS, HDFS separates metadata management from a block storage layer. Ozone uses the HDFS block storage layer to store objects identified by keys and adds a specific metadata management layer on top of block storage.

2.2.2 NoSQL Systems

NoSQL systems are specialized DBMSs that address the requirements of web and cloud data management. As an alternative to relational data stores, they support different data models and different languages than standard SQL. They emphasize scalability, fault-tolerance and availability, sometimes at the expense of consistency. NoSQL (Not Only SQL) is an overloaded term, which leaves much room for interpretation and definition. In this chapter, we consider the four main categories of NoSQL systems that are used in the cloud: key-value, wide column, document and graph. In the rest of this section, we introduce each category and illustrate with representative systems.

2.2.2.1 Key-Value Stores

In the key-value data model, all data is represented as key-value pairs, where the key unlikely identifies the value. Object stores, which we discussed above, can be viewed as a simple form of key-value store. However, the keys in key-value stores can be sequences of bytes of arbitrary length, not just positive integers, and the values can be text data, not just Blobs. Key-values stores are schemaless, which yields great flexibility and scalability. They typically provide a simple interface such as put (key, value), value = get (key), delete (key).

A popular key-value store is Dynamo [22], which is used by some of Amazon's core services that need high availability. To achieve scalability and availability, Dynamo sacrifices consistency under some failure scenarios and uses a synthesis of well known peer-to-peer techniques [47]. Data is partitioned and replicated across multiple cluster nodes in several data centers, which allows to handle entire data center failures without a data outage. The consistency among replicas during updates is maintained by a quorum-like technique and an asynchronous update propagation protocol. Dynamo employs a gossip based distributed failure detection and membership protocol. To facilitate replica consistency, it makes extensive use of object versioning and application-assisted conflict resolution in a manner that provides a novel interface for developers to use. Other popular key-value stores are Memcached, Riak and Redis.

An extended form of key-value store is able to store records, as sets of key-value pairs. One key, called major key or primary key, e.g. a social security number, uniquely identifies the record among a collection of records, e.g. people. The keys are usually sorted, which enables range queries as well as ordered processing of keys. Amazon SimpleDB

and Oracle NoSQL data store are examples of advanced key-value stores. Many systems provide further extensions so that we can see a smooth transition to wide column store and document stores, which we discuss next.

2.2.2.2 Wide Column Stores

Wide column stores are advanced key-value stores, where key-value pairs can be grouped together in columns within tables. They combine some of the nice properties of relational data stores, i.e. representing data as tables, with the flexibility of key-value stores, i.e. schemaless data.

Each row in a table is uniquely identified by a *row key*, which is like a mono-attribute key in a relational table. But unlike in a relational table, where columns can only contain atomic values, tables contain wide columns, called *column families*. A column family is a set of columns, each of which has a name, a value, and a timestamp (used for versioning) and within a column family, we can have different columns in each row. Thus, a column family is like a nested table within a column. Figure 2.3 shows a simple example of wide column table with two rows. The first column is the row key. The two other columns are column families.

Row key	Name	Email
100	Prefix : "Dr" Last : "Dobb"	email : gmail.com : " dobb@gmail.com"
101	First : "Alice" Last : "Martin"	email : gmail.com : " amartin@gmail.com" email : free.fr : " amartin@free.fr"

Figure 2.3 – A wide column table with two rows

Wide column stores extend the key-value store interface with more declarative constructs that allow scans, exact-match and range queries over column families. They typically provide an API for these constructs to be used in a programming language. Some systems also provide an SQL-like query language, e.g. Cassandra Query Language (CQL).

At the origin of wide column stores is Google Bigtable [19], a database storage system for shared-nothing clusters. Bigtable uses GFS for storing structured data in distributed files, which provides fault-tolerance and availability. It also uses a form of dynamic data partitioning for scalability. And like GFS, it is used by popular Google applications, such as Google Earth, Google Analytics and Google+.

In a Bigtable row, a row key is an arbitrary string (of up to 64KB in the original system). A column family is a unit of access control and compression. A column family is defined as a set of columns, each identified by a *column key*. The syntax for naming column keys is **family:qualifier**, e.g. "email:gmail.com" in Figure 2.3. The qualifier, e.g. "gmail.com", is like a relational attribute value, but used as a name as part of the column key to represent a single data item. This allows the equivalent of multi-valued

attributes within a relational table, but with the capability of naming attribute values. In addition, the data identified by a column key within a row can have multiple versions, each identified by a timestamp (a 64 bit integer).

Bigtable provides a basic API for defining and manipulating tables, within a programming language such as C++. The API offers various operators to write and update values, and to iterate over subsets of data, produced by a scan operator. There are various ways to restrict the rows, columns and timestamps produced by a scan, as in a relational select operator. However, there are no complex operators such as join or union, which need to be programmed using the scan operator. Transactional atomicity is supported for single row updates only.

To store a table in GFS, Bigtable uses range partitioning on the row key. Each table is divided into partitions, each corresponding to a row range. Partitioning is dynamic, starting with one partition (the entire table range) that is subsequently split into multiple partitions as the table grows. To locate the (user) partitions in GFS, Bigtable uses a metadata table, which is itself partitioned in metadata tablets, with a single root tablet stored at a master server, similar to GFS's master. In addition to exploiting GFS for scalability and availability, Bigtable uses various techniques to optimize data access and minimize the number of disk accesses, such as compression of column families as in column stores, grouping of column families with high locality of access and aggressive caching of metadata information by clients.

Bigtable builds on other Google technologies such as GFS and Chubby Lock Service. In May 2015, a public version of Bigtable was launched as Google Cloud Bigtable. There are popular open source implementations of Bigtable, such as: Hadoop Hbase that runs on top of HDFS; Cassandra that combines ideas from Bigtable and DynamoDB; and Accumulo.

2.2.2.3 Document Stores

Document stores are advanced key-value stores, where keys are mapped into values of document type, such as JSON, YAML or XML. Documents are typically grouped into collections, which play a role similar to relational tables. However, documents are different than relational tuples. Documents are self-describing, storing data and metadata (e.g. markups in XML) altogether and can be different from one another within a collection. Furthermore, the document structures are hierarchical, using nested constructs, e.g. nested objects and arrays in JSON. In addition to the simple key-value interface to retrieve documents, document stores offer an API or query language that retrieve documents based on their contents. Document stores make it easier to deal with change and optional values, and to map into program objects. This makes them attractive for modern web applications, which are subject to continual change, and where speed of deployment is important.

The most popular NoSQL document store is MongoDB [48], an open source software written in C++. MongoDB provides schema flexibility, high availability, fault-tolerance and scalability in shared-nothing cluster. It stores data as documents in BSON (Binary

JSON), an extension of JSON to include additional types such as int, long, and floating point. BSON documents contain one or more fields, and each field contains a value of a specific data type, including arrays, binary data and sub-documents.
MongoDB provides a rich query language to update and retrieve BSON data as functions expressed in JSON. The query language can be used with APIs in various programming languages. It allows key-value queries, range queries, geospatial queries, text search queries, and aggregation queries. Queries can also include user-defined JavaScript functions.

To provide efficient access to data, MongoDB includes support for many types of secondary indexes that can be declared on any field in the document, including fields within arrays. These indexes are used by the query optimizer. To scale out in shared-nothing clusters of commodity servers, MongoDB supports different kinds of data partitioning: range-based (as in Bigtable), hash-based and location-aware (whereby the user specifies key-ranges and associated nodes). High-availability is provided through primary-copy replication, with asynchronous update propagation. Applications can optionally read from secondary replicas, where data is eventually consistent [1]. MongoDB supports ACID transactions at the document level. One or more fields in a document may be written in a single transaction, including updates to multiple sub-documents and elements of an array. MongoDB makes extensive use of main memory to speed up data store operations and native compression, using its storage engine (WiredTiger). It also supports pluggable storage engines, e.g. HDFS, or in-memory, for dealing with unique application demands.

Other popular document stores are CouchDB, Couchbase, RavenDB and Elasticsearch. High level query languages can also be used on top of document stores. For instance, the Zorba query processor supports two different query languages, the standard XQuery for XML and JSONiq for JSON, which can be used to seamlessly process data stored in different data stores such as: Couchbase, Oracle NoSQL Data store and SQLite.

2.2.2.4 Graph Data stores

Graph data stores represent and store data directly as graphs which allows easy expression and fast processing of graph-like queries, e.g. computing the shortest path between two nodes in the graph. This is much more efficient than with a relational data store where graph data need be stored as separated tables and graph-like queries require repeated, expensive join operations. Graph data stores have become popular with data-intensive web-based applications such as social networks and recommender systems.

Graph data stores represent data as nodes, edges and properties. Nodes represent entities such as people or cities. Edges are lines that connect any two nodes and represent the relationship between the two. Edges can be undirected, in which case the relationship is symmetric, or directed, in which case the relationship is asymmetric. Properties provide information to nodes, e.g. a person's name and address, or edges, e.g. the name of the relationship such as "friend". The data graph is typically stored using a specific storage

[1] Eventual consistency is a form of consistency, weaker than strong consistency, which says that if we stop having replica updates, then all replicas reach the same state.

manager that places data on disk so that the time needed for graph-specific access patterns is minimized. This is typically accomplished by storing nodes as close as possible to their edges and their neighbor nodes, in the same or adjacent disk pages.

Graph data stores can provide a flexible schema, as in object data stores where objects are defined by classes, by specifying node and edge types with their properties. This facilitates the definition of indexes to provide fast access to nodes, based on some property value, e.g. a city's name. Graph queries can be expressed using graph operators through a specific API or a declarative query language, e.g. the Pixy language that works on any graph data store compatible with its API.

A popular graph data store is Neo4j [16], a commercially supported open source software. It is a robust, scalable and high-performance graph data store, with full ACID transactions. It supports directed graphs, where everything is stored in the form of either a directed edge, a node or an attribute. Each node and edge can have any number of attributes. Neo4j enforces that all operations that modify data occur within a transaction, guaranteeing data consistency. This robustness extends from single server embedded graphs to shared-nothing clusters. A single server instance can handle a graph of billions of nodes and relationships. When data throughput is insufficient, the graph data store can be distributed and replicated among multiple servers in a high availability configuration. However, graph partitioning among multiple servers is not supported (although there are some projects working on it). Neo4j supports a declarative query language called Cypher, which aims at avoiding the need to write traversals in code. It also provides REST protocols and a Java API. As of version 2.0, indexing was added to Cypher with the introduction of schemas.

Other popular graph data stores are InfiniteGraph [4], Titan [7], GraphBase [2], Trinity [50, 8] and Sparksee [6].

2.2.3 Data Processing Frameworks

Most unstructured data in the cloud gets stored in distributed files such as HDFS and needs to be analyzed using user programs. However, to make application programs scalable and efficient requires exploiting parallel processing. But parallel programming of complex applications is hard. In the context of HPC, parallel programming libraries such as OpenMP for shared-memory or Message Passing Interface (MPI) for shared-nothing are used extensively to develop scientific applications. However, these libraries are relatively low-level and require careful programming. In the context of the cloud, data processing frameworks have become quite popular to make it easier for programmers to express parallel processing code. They typically support the simple key-value data model and support operators that are automatically parallelized. All the programmer has to do is to provide code for these operators. The most popular data processing frameworks, MapReduce, Spark and now Flink, differ in the functionality they offer in terms of operators, as well as in terms of implementation, for instance, disk-based versus in-memory. However, they all target scalability and fault-tolerance in shared-nothing clusters.

MapReduce [21] is a popular framework for processing and generating large datasets.

It was initially developed by Google in C++ as a proprietary product to process large amounts of unstructured or semi-structured data, such as web documents and logs of web page requests, on large shared-nothing clusters of commodity nodes and produce various kinds of data such as inverted indices or URL access frequencies. Different implementations of MapReduce are now available such as Amazon MapReduce (as a cloud service) or Hadoop MapReduce (as a Java open source software).

MapReduce enables programmers to express in a simple, functional style their computations on large data sets and hides the details of parallel data processing, load balancing and fault-tolerance. The programming model includes only two operations, *map* and *reduce*, which we can find in many functional programming languages such as Lisp and ML. The map operation is applied to each record in the input data set to compute one or more intermediate (key, value) pairs. The reduce operation is applied to all the values that share the same unique key in order to compute a combined result. Since they work on independent inputs, map and reduce can be automatically processed in parallel, on different data partitions using many cluster nodes.

Figure 2.4 gives an overview of MapReduce execution in a cluster. There is one master node (not shown in the figure) in the cluster that assigns map and reduce tasks to cluster nodes, i.e. map and reduce nodes. The input data set is first automatically split into a number of partitions, each being processed by a different map node that applies the map operation to each input record to compute intermediate (key,value) pairs. The intermediate result is divided into n partitions, using a partitioning function applied to the key (e.g. hash(key) mod n).

Map nodes periodically write to disk their intermediate data into n regions by applying the partitioning function and indicate the region locations to the master. Reduce nodes are assigned by the master to work on one or more partitions. Each reduce node first reads the partitions from the corresponding regions on the map nodes, disks, and groups the values by intermediate key, using sorting. Then, for each unique key and group of values, it calls the user reduce operation to compute a final result that is written in the output data set.

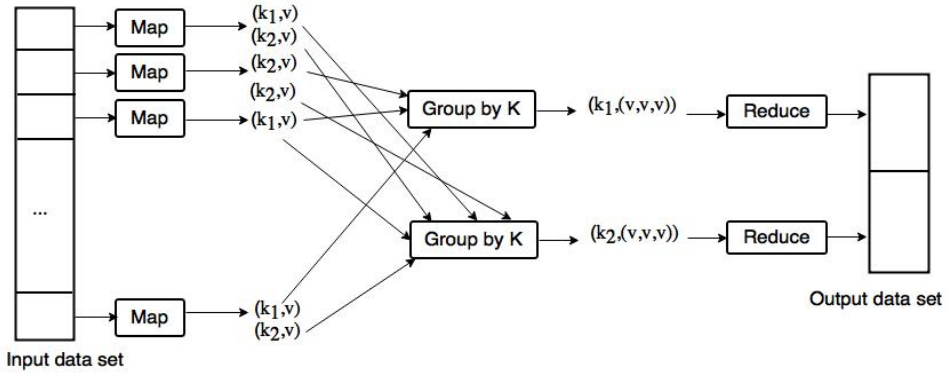


Figure 2.4 – Overview of MapReduce execution

Fault-tolerance is important as there may be many nodes executing map and reduce

operations. Input and output data are stored in GFS that already provides high fault-tolerance. Furthermore, all intermediate data are written to disk, which helps checkpointing map operations and thus provides tolerance to soft failures. However, if one map node or reduce node fails during execution (hard failure), the task can be scheduled by the master onto other nodes. It may also be necessary to re-execute completed map tasks, since the input data on the failed node disk is inaccessible. Overall, fault-tolerance is fine-grained and well-suited for large jobs.

The often mentioned advantages of MapReduce are its abilities to express various (even complicated) map and reduce functions, and its extreme scalability and fault-tolerance. However, it has been criticized for its relatively low-performance due to the extensive use of disk accesses, in particular compared with parallel DBMSs [54]. Furthermore, the two functions map and reduce are well-suited for OLAP-like queries with data selection and aggregation but not appropriate for interactive analysis or graph processing.

Spark is an Apache open source data processing framework in Java originally developed at UC Berkeley [63]. It extends the MapReduce model for two important classes of analytics applications: iterative processing (machine learning, graph processing) and interactive data mining (with R, Excel or Python). Compared with MapReduce, it improves the ease of use with the Scala language (a functional extension of Java) and a rich set of operators (map, reduce, filter, join, sortByKey, aggregateByKey, etc.). Spark provides an important abstraction, called Resilient Distributed Dataset (RDD), which is a collection of elements partitioned across cluster nodes. RDDs can be created from disk-based resident data in files or intermediate data produced by transformations with Scala programs. They can also be made memory-resident for efficient reuse across parallel operations.

Flink is the latest Apache open source data processing framework. Based on the Stratosphere prototype [26], it differs from Spark by its in-memory runtime engine which can be used for real time data streams as well as batch data processing. It runs on HDFS and supports APIs for Java and Scala.

2.2.3.1 Concluding Remarks

The software stack for data management in the cloud, with three main layers (distributed storage, database management and distributed processing) has led to a rich ecosystem with many different solutions and technologies, which are still evolving. Although HDFS has established itself as the standard solution for storing unstructured data, we should expect evolutions of distributed file systems that combine block storage and object storage in a single system. For data management, most NoSQL data stores, except graph data stores, rely on (or extend) the key-value data model, which remains the best option for data whose structure needs to be flexible. There is also a rapid evolution of data processing frameworks on top of distributed file systems. For example, the popular MapReduce framework is now challenged by more recent systems such as Spark and Flink. Multistore systems should be able to cope with this evolution.

2.3 Multidatabase Query Processing

A multidatabase system provides transparent access to a collection of multiple, heterogeneous data stores distributed over a computer network [46]. In addition to be heterogeneous and distributed, the data stores can be autonomous, i.e. controlled and managed independently (e.g. by a different data store administrator) of the multidatabase system.

Since the data stores already exist, one is faced with the problem of providing integrated access to heterogeneous data. This requires data integration, which consists in defining a global schema for the multidatabase over the existing data and mappings between the global schema and the local data store schemas. Once data integration is done, the global schema can be used to express queries over multiple data stores as if it were a single (global) data store.

Most of the work on multidatabase query processing has been done in the context of the mediator-wrapper architecture [56]. This architecture and related techniques can be used for loosely-coupled multistore systems, which is why we introduce them. In the rest of this section, we describe the mediator-wrapper and multidatabase query processing architectures, and the query processing techniques.

2.3.1 Mediator-Wrapper Architecture

In this architecture (see Figure 2.5), there is a clear separation of concerns: the mediator deals with data store distribution while the wrappers deal with data store heterogeneity and autonomy. This is achieved by using a common language between mediator and wrappers, and the translation to the data store language is done by the wrappers.

Each data store has an associated wrapper that exports information about the source schema, data and query processing capabilities. To deal with the heterogeneous nature of data stores, wrappers transform queries received from the mediator, expressed in a common query language, to the particular query language of the source. A wrapper supports the functionality of translating queries appropriate to the particular server, and reformatting answers (data) appropriate to the mediator. One of the major practical uses of wrappers has been to allow an SQL-based DBMS to access non SQL data stores.

The mediator centralizes the information provided by the the wrappers in a unified view of all available data (stored in a global catalog). This unified view can be of two fundamental types [42]: local-as-view (LAV) and global-as-view (GAV). In LAV, the global schema definition exists, and each data store schema is treated as a view definition over it. In GAV on the other hand, the global schema is defined as a set of views over the data store schemas. These views indicate how the elements of the global schema can be derived, when needed, from the elements of the data store schemas. The main functionality of the mediator is to provide uniform access to multiple data stores and perform query decomposition and processing using the wrappers to access the data stores.

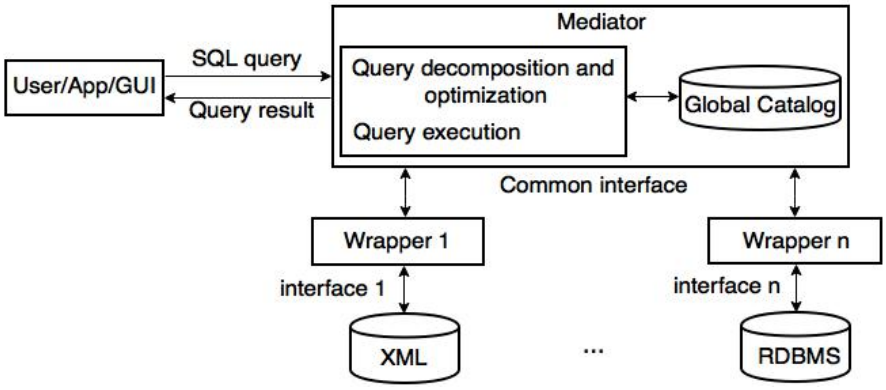


Figure 2.5 – Mediator-Wrapper architecture

2.3.2 Multidatabase Query Processing Architecture

We assume the input is a query on relations expressed on a global schema in a declarative language, e.g. SQL. This query is posed on global relations, meaning that data distribution and heterogeneity are hidden. Three main layers are involved in multidatabase query processing.

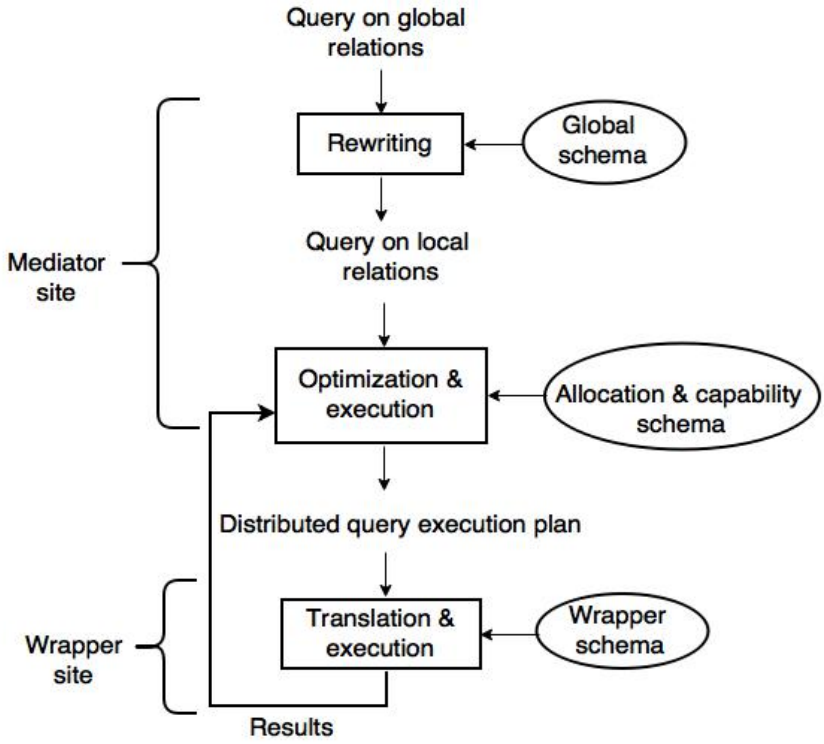


Figure 2.6 – Generic layering scheme for multidatabase query processing (modified after [46]).

The first two layers map the input query into an optimized query execution plan (QEP). They perform the functions of query rewriting, query optimization and some query execution. The first two layers are performed by the mediator and use meta-information stored in the global catalog (global schema, data store location, cost information, etc.). Query rewriting rewrites the input query into a query on local relations, using the global schema. Thus, the global schema provides the view definitions (i.e. GAV or LAV mappings between the global relations and the local relations stored in the data stores) and the query is rewritten using the views.

The second layer performs distributed query optimization and (some) execution by considering the location of the relations and the different query processing capabilities of the data stores exported by the wrappers. The distributed QEP produced by this layer groups within subqueries the operations that can be performed by the data stores and wrappers. As in centralized DBMSs, query optimization can be static or dynamic. However, the lack of homogeneity in multidatabase systems (e.g. some data stores may have unexpected long delays in answering) make dynamic query optimization important. In the case of dynamic optimization, there may be subsequent calls to this layer after execution by the next layer. This is illustrated by the arrow showing results coming from the next layer. Finally, this layer integrates the results coming from the different wrappers to provide a unified answer to the users query. This requires the capability of executing some operations on data coming from the wrappers. Since the wrappers may provide very limited execution capabilities, e.g. in the case of very simple data stores, the mediator must provide the full execution capabilities to support the mediator interface.

The third layer performs query translation and execution using the wrappers. Then it returns the results to the mediator which can perform result integration from different wrappers and subsequent execution. Each wrapper maintains a wrapper schema that includes the local schema and mapping information to facilitate the translation of the input subquery (a subset of the QEP) expressed in a common language into the language of the data store. After the subquery is translated, it is executed by the data store and the local result is translated back in the common format.

2.3.3 Multidatabase Query Processing Techniques

The three main problems of query processing in multidatabase systems are: heterogeneous cost modeling, heterogeneous query optimization, to deal with different capabilities of data stores' DBMSs and adaptive query processing, to deal with strong variations in the environment (failures, unpredictable delays, etc.).

2.3.3.1 Heterogeneous Cost Modeling

Heterogeneous cost modeling refers to cost function definition, and the associated problem of obtaining cost-related information from the data stores. Such information is important to estimate the costs of executing subqueries at the data stores, which in turn are used to estimate the costs of alternative QEPs generated by the multidatabase query optimizer.

There are three alternative approaches for determining the cost of executing queries in a multidatabase system: black-box, customized and dynamic.

The black-box approach treats the data stores as a black box, running some test queries on them, and from these determines the necessary cost information. It is based on running probing queries on data stores to determine cost information. Probing queries can, in fact, be used to gather a number of cost information factors. For example, probing queries can be issued to retrieve data from data stores to build up and update the multidatabase catalog. Statistical probing queries can be issued that, for example, count the number of tuples of a relation. Finally, performance measuring probing queries can be issued to measure the elapsed time for determining cost function coefficients.

The customized approach uses previous knowledge about the data stores, as well as their external characteristics, to subjectively determine the cost information. The basis for this approach is that the query processors of the data stores are too different to be represented by a unique cost model. It also assumes that the ability to accurately estimate the cost of local subqueries will improve global query optimization. The approach provides a framework to integrate the data stores cost model into the mediator query optimizer. The solution is to extend the wrapper interface such that the mediator gets some specific cost information from each wrapper. The wrapper developer is free to provide a cost model, partially or entirely.

The above approaches assume that the execution environment is stable over time. However, on the Internet for instance, the execution environment factors are frequently changing. The dynamic approach consists in monitoring the runtime behavior of data stores and dynamically collecting the cost information Three classes of environmental factors can be identified based on their dynamicity. The first class for frequently changing factors (every second to every minute) includes CPU load, I/O throughput, and available memory. The second class for slowly changing factors (every hour to every day) includes DBMS configuration parameters, physical data organization on disks, and data store schema. The third class for almost stable factors (every month to every year) includes DBMS type, data store location, and CPU speed. To face dynamic environments where network contention, data storage or available memory change over time, a solution is to extend the sampling method and consider user queries as new samples.

2.3.3.2 Heterogeneous Query Optimization

In addition to heterogeneous cost modeling, multidatabase query optimization must deal with the issue of the heterogeneous computing capabilities of data stores. For instance, one data store may support only simple select operations while another may support complex queries involving join and aggregate. Thus, depending on how the wrappers export such capabilities, query processing at the mediator level can be more or less complex. There are two main approaches to deal with this issue depending on the kind of interface between mediator and wrapper: query-based and operator-based.

Query-based Approach

In the query-based approach, the wrappers support the same query capability, e.g. a subset of SQL, which is translated to the capability of the data store. This approach typically relies on a standard DBMS interface such as Open Database Connectivity (ODBC) or its many variations (e.g. JDBC). Thus, since the data stores appear homogeneous to the mediator, query processing techniques designed for homogeneous distributed DBMS can be reused. However, if the data stores have limited capabilities, the additional capabilities must be implemented in the wrappers, e.g. join queries may need to be handled at the mediator, if the data store does not support join.

Since the data stores appear homogeneous to the mediator, a solution is to use a traditional distributed query optimization algorithm with a heterogeneous cost model. However, extensions are needed to convert the distributed execution plan into subqueries to be executed by the data stores and subqueries to be executed by the mediator. The hybrid two-step optimization technique is useful in this case: in a first step, a static plan is produced by a centralized cost-based query optimizer; in a second step, at startup time, an execution plan is produced by carrying out site selection and allocating the subqueries to the sites.

Operator-based Approach

In the operator-based approach, the wrappers export the capabilities of the data stores through compositions of relational operators. Thus, there is more flexibility in defining the level of functionality between the mediator and the wrapper. In particular, the different capabilities of the data stores can be made available to the mediator.

Expressing the capabilities of the data stores through relational operators allows tighter integration of query processing between mediator and wrappers. In particular, the mediator-wrapper communication can be in terms of sub plans. We illustrate the operator-based approach with planning functions proposed in the Garlic project [32]. In this approach, the capabilities of the data stores are expressed by the wrappers as planning functions that can be directly called by a centralized query optimizer. It extends a traditional query optimizer with operators to create temporary relations and retrieve locally stored data. It also creates the PushDown operator that pushes a portion of the work to the data stores where it will be executed.

The execution plans are represented, as usual, with operator trees, but the operator nodes are annotated with additional information that specifies the source(s) of the operand(s), whether the results are materialized, and so on. The Garlic operator trees are then translated into operators that can be directly executed by the execution engine. Planning functions are considered by the optimizer as enumeration rules. They are called by the optimizer to construct sub plans using two main functions: accessPlan to access a relation, and joinPlan to join two relations using access plans. There is also a join rule for *bind join*. A bind join is a nested loop join in which intermediate results (e.g. values for the join predicate) are passed from the outer relation to the wrapper for the inner relation, which uses these results to filter the data it returns. If the intermediate results are small

and indexes are available at data stores, bindings can significantly reduce the amount of work done by a data store. Furthermore, bindings can reduce communication cost.

Using planning functions for heterogeneous query optimization has several advantages. First, planning functions provide a flexible way to express precisely the capabilities of data stores. In particular, they can be used to model non relational data stores such as web sites. Second, since these rules are declarative, they make wrapper development easier. Finally, this approach can be easily incorporated in an existing, centralized query optimizer.

The operator-based approach has also been used in DISCO, a multidatabase system designed to access data stores over the web [56]. DISCO uses the GAV approach and an object data model to represent both mediator and data store schemas and data types. This allows easy introduction of new data stores with no type mismatch or simple type mismatch. The data store capabilities are defined as a subset of an algebraic machine (with the usual operators such as scan, join and union) that can be partially or entirely supported by the wrappers or the mediator. This gives much flexibility for the wrapper implementers to decide where to support data store capabilities (in the wrapper or in the mediator).

2.3.3.3 Adaptive Query Processing

Multidatabase query processing, as discussed so far, follows essentially the principles of traditional query processing whereby an optimal QEP is produced for a query based on a cost model, and then this QEP is executed. The underlying assumption is that the multidatabase query optimizer has sufficient knowledge about query runtime conditions in order to produce an efficient QEP and the runtime conditions remain stable during execution. This is a fair assumption for multidatabase queries with few data stores running in a controlled environment. However, this assumption is inappropriate for changing environments with large numbers of data stores and unpredictable runtime conditions as on the Web.

Adaptive query processing is a form of dynamic query processing, with a feedback loop between the execution environment and the query optimizer in order to react to unforeseen variations of runtime conditions. A query processing system is defined as adaptive if it receives information from the execution environment and determines its behavior according to that information in an iterative manner [11]. In the context of multidatabase systems, the execution environment includes the mediator, wrappers and data stores. In particular, wrappers should be able to collect information regarding execution within the data stores.

Adaptive query processing adds to the traditional query processing process the following activities: monitoring, assessing and reacting. These activities are logically implemented in the query processing system by sensors, assessment components, and reaction components, respectively. These components may be embedded into control operators of the QEP, e.g. an Exchange operator. Monitoring involves measuring some environment parameters within a time window, and reporting them to the assessment component.

The latter analyzes the reports and considers thresholds to arrive at an adaptive reaction plan. Finally, the reaction plan is communicated to the reaction component that applies the reactions to query execution.

2.4 Multistore Systems

Multistore systems provide integrated access to a number of cloud data stores such as NoSQL, RDBMS or HDFS, sometimes through a data processing framework such as Spark. They typically support only read-only queries, as supporting distributed transactions across data stores is a hard problem. We can divide multistore systems based on the level of coupling with the underlying data stores: loosely-coupled, tightly-coupled and hybrid. In this section, we introduce for each class a set of representative systems, with their architecture and query processing. We end the section with a comparative analysis.

In presenting these systems, we strive to use the same terminology we used so far in this chapter. However, it is not easy as we often need to map the specific terminology used in the original papers and ours. When necessary, to help the reader familiar with some systems, we make precise this terminology mapping.

2.4.1 Loosely-Coupled Multistore Systems

Loosely-coupled multistore systems are reminiscent of multidatabase systems in that they can deal with autonomous data stores, which can be accessed through the multistore system common interface as well as separately through their local API. They follow the mediator-wrapper architecture with several data stores (e.g. NoSQL and RDBMS) as depicted in Figure 2.7. Each data store is autonomous, i.e. locally controlled, and can be accessed by other applications. Like web data integration systems that use the mediator-wrapper architecture, the number of data stores can be very high.

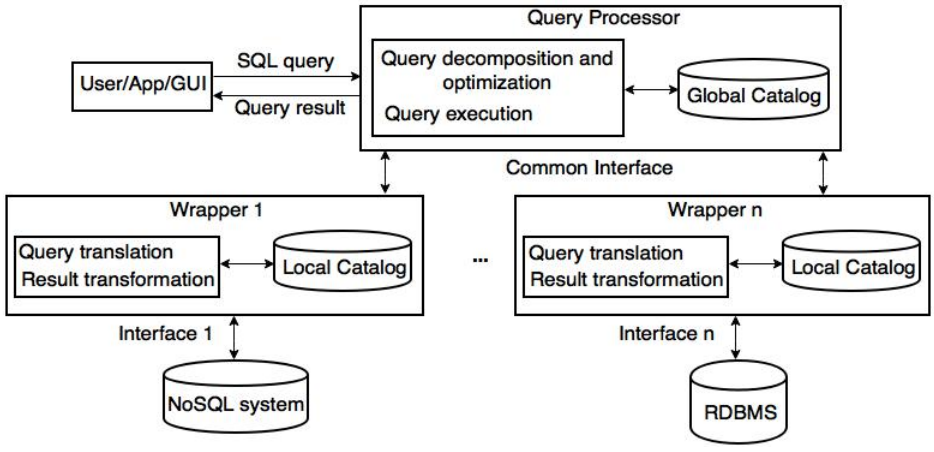


Figure 2.7 – Loosely-coupled multistore systems

There are two main modules: one query processor and one wrapper per data store. The query processor has a catalog of data stores, and each wrapper has a local catalog of its data store. After the catalogs and wrappers have been built, the query processor can start processing input queries from the users, by interacting with wrappers. The typical query processing is as follows:

1. Analyze the input query and translate it into subqueries (one per data store), each expressed in a common language, and an integration subquery.

2. Send the subqueries to the relevant wrappers, which trigger execution at the corresponding data stores and translate the results into the common language format.

3. Integrate the results from the wrappers (which may involve executing operators such union and join), and return the results to the user. We describe below three loosely-coupled multistore systems: BigIntegrator, Forward and Qox.

BigIntegrator

BigIntegrator [64] supports SQL-like queries that combines data in Bigtable data stores in the cloud and data in relational data stores. Bigtable is accessed through the Google Query Language (GQL), which has very limited query expressions, e.g. no join and only basic select predicates. To capture GQL's limited capabilities, BigIntegrator provides a novel query processing mechanism based on plugins, called absorber and finalizer, which enable to pre and post-process those operations that cannot be processed by Bigtable. For instance, a "LIKE" select predicate on a Bigtable or a join of two Bigtables will be processed through operations in BigIntegrator's query processor.

BigIntegrator uses the LAV approach for defining the global schema of the Bigtable and relational data stores as flat relational tables. Each Bigtable or relational data store can contain several collections, each represented as a source table of the form "table-name_source-name", where table-name is the name of the table in the global schema and source-name is the name of the data source. For instance, "Employees_A" represents an Employees table at source A, i.e. a local view of Employees. The source tables are referenced as tables in the SQL queries.

Figure 2.8 illustrates the architecture of BigIntegrator with two data stores, one relational data store and one Bigtable data store. Each wrapper has an importer module and absorber and finalizer plug-ins. The importer creates the source tables and stores them in the local catalog. The absorber extracts a subquery, called access filter, from a user query that selects data from a particular source table, based on the capabilities of the source. The finalizer translates each access filter (produced by the absorber) into an operator called interface function, specific for each kind of source. The interface function is used to send a query to the data store (i.e. a GQL or SQL query).

Query processing is performed in three steps, using an absorber manager, a query optimizer and a finalizer manager. The absorber manager takes the (parsed) user query and, for each source table referenced in the query, calls the corresponding absorber of its

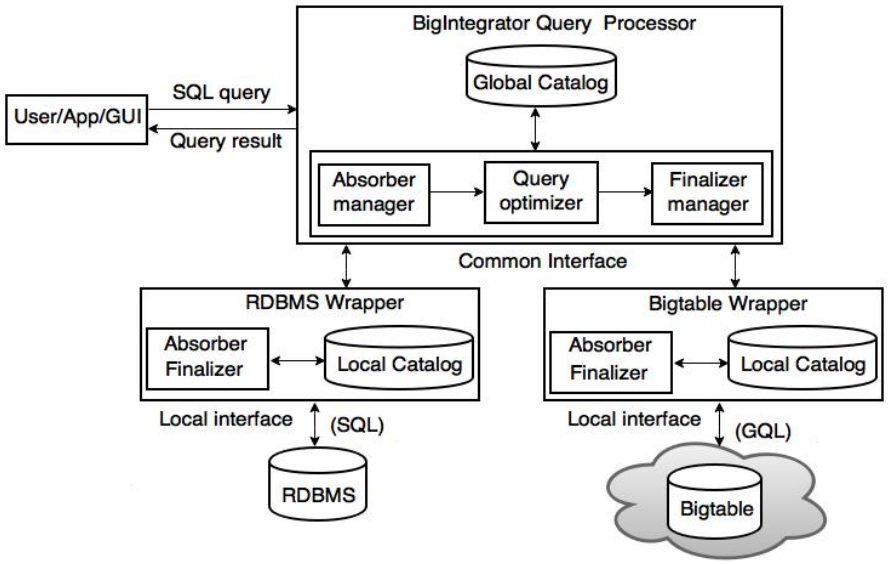


Figure 2.8 – BigIntegrator

wrapper. In order to replace the source table with an access filter, the absorber collects from the query the source tables and the possible other predicates, based on the capabilities of the data store. The query optimizer reorders the access filters and other predicates to produce an algebra expression that contains calls to both access filters and other relational operators. It also performs traditional transformations such as select push down and bind join. The finalizer manager takes the algebra expression and, for each access filter operator in the algebra expression, calls the corresponding finalizer of its wrapper. The finalizer transforms the access filters into interface function calls.

Finally, query execution is performed by the query processor that interprets the algebra expression, by calling the interface functions to access the different data stores and executing the subsequent relational operations, using in-memory techniques.

Forward

The Forward multistore system, or so-called Forward middleware in [45], supports SQL++ , an SQL-like language designed to unify the data model and query language capabilities of NoSQL and relational data stores. SQL++ has a powerful, semi-structured data model that extends both the JSON and relational data models. FORWARD also provides a rich web development framework [27], which exploits its JSON compatibility to integrate visualization components (e.g. Google Maps).

The design of SQL++ is based on the observation that the concepts are similar across both data models, e.g. a JSON array is similar to an SQL table with order, and an SQL tuple to a JSON object literal. Thus, an SQL++ collection is an array or a bag, which may contain duplicate elements. An array is ordered (similar to a JSON array) and each element is accessible by its ordinal position while a bag is unordered (similar to a SQL table).

Furthermore, SQL++ extends the relational model with arbitrary composition of complex values and element heterogeneity. As in nested data models, a complex value can be either a tuple or collection. Nested collections can be accessed by nesting SELECT expressions in the SQL FROM clause or composed using the GROUP BY operator. They can also be unnested using the FLATTEN operator. And unlike an SQL table that requires all tuples to have the same attributes, an SQL++ collection may also contain heterogeneous elements comprising a mix of tuples, scalars, and nested collections.

Forward uses the GAV approach, where each data store (SQL or NoSQL) appears to the user as an SQL++ virtual view, defined over SQL++ collections. Thus, the user can issue SQL++ queries involving multiple virtual views. The Forward architecture is that of Figure 2.7, with a query processor and one wrapper per data store. The query processor performs SQL++ query decomposition, by exploiting the underlying data store capabilities as much as possible. However, given an SQL++ query that is not directly supported by the underlying data store, Forward will decompose it into one or more native queries that are supported and combine the native query results in order to compensate for the semantics or capabilities gap between SQL++ and the underlying data store. Although not described in the original paper [45] cost-based optimization of SQL++ queries is possible, by reusing techniques from multidatabase systems when dealing with flat collections. However, it would be much harder considering the nesting and element heterogeneity capabilities of SQL++.

QoX

QoX [52] is a special kind of loosely-coupled multistore system, where queries are analytical data-driven workflows (or data flows) that integrate data from relational data stores, and various execution engines such as MapReduce or Extract-Transform-Load (ETL) tools. A typical data flow may combine unstructured data (e.g. tweets) with structured data and use both generic data flow operations like filtering, join, aggregation and user-defined functions like sentiment analysis and product identification. In a previous work [51], the authors proposed a novel approach to ETL design that incorporates a suite of quality metrics, termed QoX, at all stages of the design process. The QoX Optimizer deals with the QoX performance metrics, with the objective of optimizing the execution of data flows that integrate both the back-end ETL integration pipeline and the front-end query operations into a single analytics pipeline.

The QoX Optimizer uses xLM, a proprietary XML-based language to represent data flows, typically created with some ETL tool. xLM allows capturing the flow structure, with nodes showing operations and data stores and edges interconnecting these nodes, and important operation properties such as operation type, schema, statistics, and parameters. Using appropriate wrappers to translate xLM to a tool-specific XML format and vice versa, the QoX Optimizer may connect to external ETL engines and import or export data flows to and from these engines.

Given a data flow for multiple data stores and execution engines, the QoX Optimizer evaluates alternative execution plans, estimates their costs, and generates a physical plan

(executable code). The search space of equivalent execution plans is defined by flow transformations that model data shipping (moving the data to where the operation will be executed), function shipping (moving the operation to where the data is), and operation decomposition (into smaller operations). The cost of each operation is estimated based on statistics (e.g. cardinalities, selectivities). Finally, the QoX Optimizer produces SQL code for relational data store engines, Pig and Hive code for MapReduce engines, and creates Unix shell scripts as the necessary glue code for orchestrating different subflows running on different engines. This approach could be extended to access NoSQL engines as well, provided SQL-like interfaces and wrappers.

2.4.2 Tightly-Coupled Multistore Systems

Tightly-coupled multistore systems aim at efficient querying of structured and unstructured data for (big) data analytics. They may also have a specific objective, such as self-tuning or integration of HDFS and RDBMS data. However, they all trade autonomy for performance, typically in a shared-nothing cluster, so that data stores can only be accessed through the multistore system, directly through their local API.

Like loosely-coupled systems, they provide a single language for querying of structured and unstructured data. However, the query processor directly uses the data store local interfaces (see Figure 2.9), or in the case of HDFS, it interfaces a data processing framework such as MapReduce or Spark. Thus, during query execution, the query processor directly accesses the data stores. This allows efficient data movement across data stores. However, the number of data stores that can be interfaced is typically very limited.

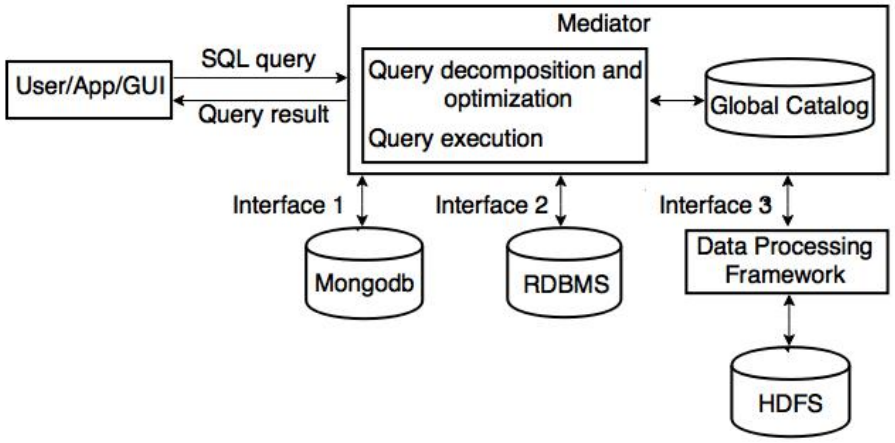


Figure 2.9 – Tightly-coupled multistore systems

In the rest of this section, we describe three representative tightly-coupled multistore systems: Polybase, HadoopDB and Estocada. Two other interesting systems are Odyssey and JEN. Odyssey [34] is a multistore system that can work with different analytic engines, such as parallel OLAP system or Hadoop. It enables storing and querying data within HDFS and RDBMS, using opportunistic materialized views, based on MISO [41].

MISO is a method for tuning the physical design of a multistore system (Hive/HDFS and RDBMS), i.e. deciding in which data store the data should reside, in order to improve the performance of big data query processing. The intermediate results of query execution are treated as opportunistic materialized views, which can then be placed in the underlying stores to optimize the evaluation of subsequent queries. JEN [62] is a component on top of HDFS to provide tight-coupling with a parallel RDBMS. It allows joining data from two data stores, HDFS and RDBMS, with parallel join algorithms, in particular, an efficient zigzag join algorithm, and techniques to minimize data movement. As the data size grows, executing the join on the HDFS side appears to be more efficient.

Polybase

Polybase [23] is a feature of the SQL Server Parallel Data Warehouse (PDW) product, which allows users to query unstructured (HDFS) data stored in a Hadoop cluster using SQL and integrate them with relational data in PDW. The HDFS data can be referenced in Polybase as external tables, which make the correspondence with the HDFS file on the Hadoop cluster, and thus be manipulated together with PDW native tables using SQL queries Polybase leverages the capabilities of PDW, a shared-nothing parallel DBMS. Using the PDW query optimizer, SQL operators on HDFS data are translated into MapReduce jobs to be executed directly on the Hadoop cluster. Furthermore, the HDFS data can be imported/exported to/from PDW in parallel, using the same PDW service that allows shuffling PDW data among compute nodes.

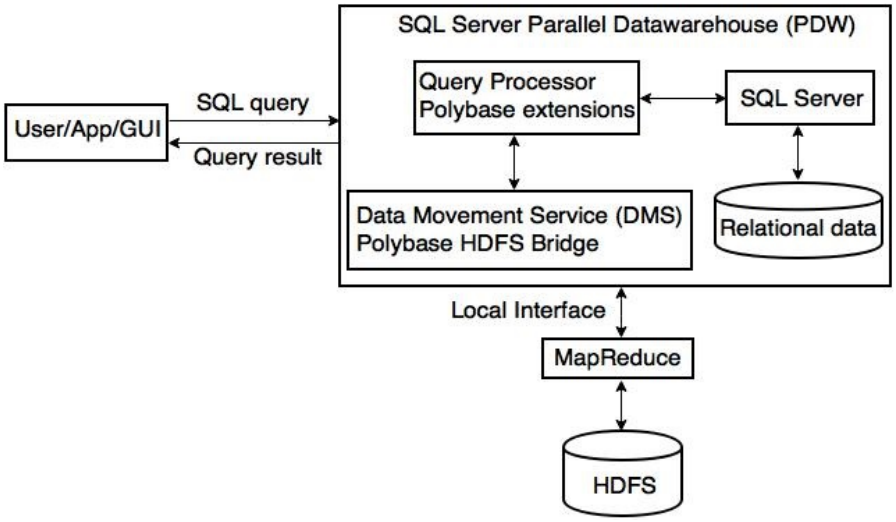


Figure 2.10 – Polybase architecture

The architecture of Polybase, which is integrated within PDW, is shown in Figure 2.10. Polybase takes advantage of PDW's Data Movement Service (DMS), which is responsible for shuffling intermediate data across PDW nodes, e.g. to repartition tuples, so that any matching tuples of an equi-join be collocated at the same computing node that performs the join. DMS is extended with an HDFS Bridge component, which is responsible for all

communications with HDFS. The HDFS Bridge enables DMS instances to also exchange data with HDFS in parallel (by directly accessing HDFS splits).

Polybase relies on the PDW cost-based query optimizer to determine when it is advantageous to push SQL operations on HDFS data to the Hadoop cluster for execution. Thus, it requires detailed statistics on external tables, which are obtained by exploring statistically significant samples of HDFS tables. The query optimizer enumerates the equivalent QEPs and select the one with least cost. The search space is obtained by considering the different decompositions of the query into two parts: one to be executed as MapReduce jobs at the Hadoop cluster and the other as regular relational operators at the PDW side. MapReduce jobs can be used to perform select and project operations on external tables, as well as joins of two external tables. However, no bind join optimization is supported. The data produced by the MapReduce jobs can then be exported to PDW to be joined with relational data, using parallel hash-based join algorithms.

One strong limitation of pushing operations on HDFS data as MapReduce jobs is that even simple lookup queries have long latencies. A solution proposed for Polybase [28] is to exploit an index built on the external HDFS data using a B+-tree that is stored inside PDW. This method leverages the robust and efficient indexing code in PDW without forcing a dramatic increase in the space that is required to store or cache the entire (large) HDFS data inside PDW. Thus, the index can be used as a pre-filter by the query optimizer to reduce the amount of work that is carried out as MapReduce jobs. To keep the index synchronized with the data that is stored in HDFS, an incremental approach is used which records that the index is out-of-date, and lazily rebuilds it. Queries posed against the index before the rebuild process is completed can be answered using a method that carefully executes parts of the query using the index in PDW, and the remaining part of the query is executed as a MapReduce job on just the changed data in HDFS.

HadoopDB

The objective of HadoopDB [9] is to provide the best of both parallel DBMS (high-performance data analysis over structured data) and MapReduce-based systems (scalability, fault-tolerance, and flexibility to handle unstructured data) with an SQL-like language (HiveQL). To do so, HadoopDB tightly couples the Hadoop framework, including MapReduce and HDFS, with multiple single-node RDBMS (e.g. PostgreSQL or MySQL) deployed across a cluster, as in a shared-nothing parallel DBMS.

HadoopDB extends the Hadoop architecture with four components: database connector, catalog, data loader, and SQL-MapReduce-SQL (SMS) planner. The database connector provides the wrappers to the underlying RDBMS, using JDBC drivers. The catalog maintains information about the data stores as an XML file in HDFS, and is used for query processing. The data loader is responsible for (re)partitioning (key, value) data collections using hashing on a key and loading the single-node databases with the partitions (or chunks). The SMS planner extends Hive, a Hadoop component that transforms HiveQL into MapReduce jobs that connect to tables stored as files in HDFS. This architecture yields a cost-effective parallel RDBMS, where data is partitioned both in RDBMS

tables and in HDFS files, and the partitions can collocated at cluster nodes for efficient parallel processing.

Query processing is simple, relying on the SMS planner for translation and optimization, and MapReduce for execution. The optimization consists in pushing as much work as possible into the single node databases, and repartitioning data collections whenever needed. The SMS planner decomposes a HiveQL query to a QEP of relational operators. Then the operators are translated to MapReduce jobs, while the leaf nodes are again transformed into SQL to query the underlying RDBMS instances. In MapReduce, repartitioning should take place before the reduce phase. However, if the optimizer detects that an input table is partitioned on a column used as aggregation key for Reduce, it will simplify the QEP by turning it to a single Map-only job, leaving all the aggregation to be done by the RDBMS nodes. Similarly, repartitioning is avoided for equi-joins as well, if both sides of the join are partitioned on the join key.

Estocada

Estocada [17] is a self-tuning multistore system with the goal of optimizing the performance of applications that must deal with data in multiple data models, including relational, key-value, document and graph. To obtain the best possible performance from the available data stores, Estocada automatically distributes and partitions the data across the different data stores, which are entirely under its control and hence not autonomous. Hence, it is a tightly-coupled multistore system.

Data distribution is dynamic and decided based on a combination of heuristics and cost-based decisions, taking into account data access patterns as they become available. Each data collection is stored as a set of partitions, whose content may overlap, and each partition may be stored in any of the underlying data stores. Thus, it may happen that a partition is stored in a data store that has a different data model than its native one. To make Estocada applications independent of the data stores, each data partition is internally described as a materialized view over one or several data collections. Thus, query processing involves view-based query rewriting.

Estocada support two kinds of requests, for storing data and querying, with three main modules: storage advisor, catalog, query processor and execution engine. These components can directly access the data stores through their local interface. The query processor deals with single model queries only, each expressed in the query language of the corresponding data store. However, to integrate various data stores, one would need a common data model and language on top of Estocada. The storage advisor is responsible for partitioning data collections and delegating the storage of partitions to the data stores. For self-tuning the applications, it may also recommend repartitioning or moving data from one data store to the other, based on access patterns. Each partition is defined as a materialized view expressed as a query over the collection in its native language. The catalog keeps track of information about partitions, including some cost information about data access operations by means of binding patterns which are specific to the data stores.

Using the catalog, the query processor transforms a query on a data collection into

a logical QEP on possibly multiple data stores (if there are partitions of the collection in different stores). This is done by rewriting the initial query using the materialized views associated with the data collection, and selecting the best rewriting, based on the estimated execution costs. The execution engine translates the logical QEP into a physical QEP which can be directly executed by dividing the work between the data stores and Estocada's runtime engine, which provides its own operators (select, join, aggregate, etc.).

2.4.3 Hybrid systems

Hybrid systems try to combine the advantages of loosely-coupled systems, e.g. accessing many different data stores, and tightly-coupled systems, e.g. accessing some data stores directly through their local interface for efficient access. Therefore, the architecture (see Figure 2.11) follows the mediator-wrapper architecture, while the query processor can also directly access some data stores, e.g. HDFS through MapReduce or Spark.

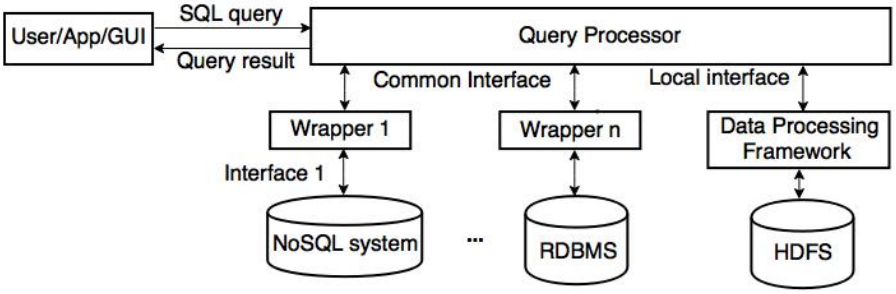


Figure 2.11 – Hybrid architecture

We describe below the three hybrid multistore systems: SparkSQL, CloudMdsQL and BigDAWG.

SparkSQL

SparkSQL [10] is a recent module in Apache Spark that integrates relational data processing with Spark's functional programming API. It supports SQL-like queries that can integrate HDFS data accessed through Spark and external data stores (e.g. relational data stores) accessed through a wrapper. Thus, it is a hybrid multistore system with tight-coupling of Spark/HDFS and loose-coupling of external data stores.

SparkSQL uses a nested data model that includes tables and DataFrames. It supports all major SQL data types, as well as user-defined types and complex data types (structs, arrays, maps and unions), which can be nested together. A DataFrame is a distributed collection of rows with the same schema, like a relational table. It can be constructed from a table in an external data store or from an existing Spark RDD of native Java or Python objects. Once constructed, DataFrames can be manipulated with various relational operators, such as WHERE and GROUPBY, which take expressions in procedural Spark code.

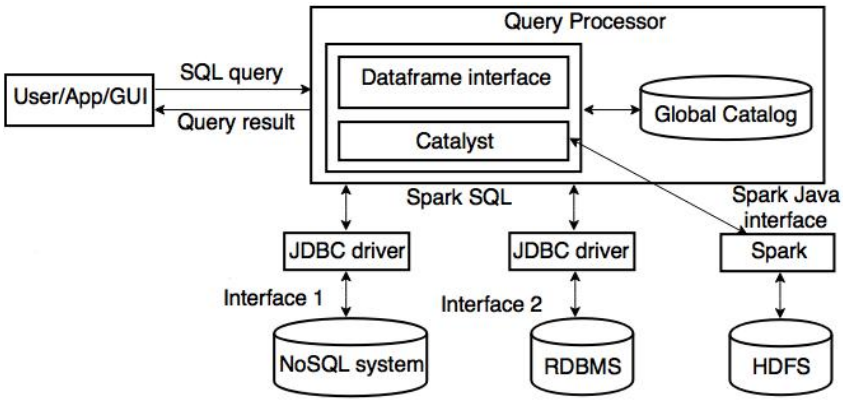


Figure 2.12 – SparkSQL architecture

Figure 2.12 shows the architecture of SparkSQL, which runs as a library on top of Spark, The query processor directly accesses the Spark engine through the Spark Java interface, while it accesses external data stores (e.g. an RDBMS or a key-value store) through the SparkSQL common interface supported by wrappers (JDBC drivers). The query processor includes two main components: the DataFrame API and the Catalyst query optimizer. The DataFrame API offers tight integration between relational and procedural processing, allowing relational operations to be performed on both external data stores and Spark's RDDs. It is integrated into Spark's supported programming languages (Java, Scala, and Python) and supports easy inline definition of user-defined functions, without the complicated registration process typically found in other data store systems. Thus, the DataFrame API lets developers seamlessly mix relational and procedural programming, e.g. to perform advanced analytics (which is cumbersome to express in SQL) on large data collections (accessed through relational operations).

Catalyst is an extensible query optimizer that supports both rule-based and cost-based optimization. The motivation for an extensible design is to make it easy to add new optimization techniques, e.g. to support new features of SparkSQL, as well as to enable developers to extend the optimizer to deal with external data stores, e.g. by adding data store specific rules to push down select predicates. Although extensible query optimizers have been proposed in the past, they have typically required a complex language to specify rules, and a specific compiler to translate the rules into executable code. In contrast, Catalyst uses standard features of the Scala functional programming language, such as pattern-matching, to make it easy for developers to specify rules, which can be complied with Java code.

Catalyst provides a general transformation framework for representing query trees and applying rules to manipulate them. This framework is used in four phases: (1) query analysis, (2) logical optimization, (3) physical optimization, and (4) code generation. Query analysis resolves name references using a catalog (with schema information) and produces a logical plan. Logical optimization applies standard rule-based optimizations to the logical plan, such as predicate pushdown, null propagation, and Boolean expression

simplification. Physical optimization takes a logical plan and enumerates a search space of equivalent physical plans, using physical operators implemented in the Spark execution engine or in the external data stores. It then selects a plan using a simple cost model, in particular, to select the join algorithms. Code generation relies on the Scala language, in particular, to ease the construction of abstract syntax trees (ASTs) in the Scala language. ASTs can then be fed to the Scala compiler at runtime to generate Java bytecode to be directly executed by compute nodes.

To speed up query execution, SparkSQL exploits in-memory caching of hot data using a columnar storage (i.e. storing data collections as sections of columns of data rather than as rows of data). Compared with Spark's native cache, which simply stores data as Java native objects, this columnar cache can reduce memory footprint by an order of magnitude by applying columnar compression schemes (e.g. dictionary encoding and run-length encoding). Caching is particularly useful for interactive queries and for the iterative algorithms common in machine learning.

BigDAWG

Like multidatabase systems, all the multistore systems we have seen so far provide transparent access across multiple data stores with the same data model and language. The BigDAWG (Big Data Analytics Working Group) multistore system (called polystore) [25] takes a different path, with the goal of unifying querying over a variety of data models and languages, Thus, there is no common data model and language. A key user abstraction in BigDAWG is an island of information, which is a collection of data stores accessed with a single query language. And there can be a variety of islands, including relational (RDBMS), Array DBMS, NoSQL and Data Stream Management System (DSMS). Within an island, there is loose-coupling of the data stores, which need to provide a wrapper (called shim) to map the island language to their native one. When a query accesses more than one data store, objects may have to be copied between local data stores, using a CAST operation, which provides a form of tight-coupling. This is why BigDAWG can be viewed as a hybrid multistore system.

The architecture of BigDAWG is highly distributed, with a thin layer that interfaces the tools (e.g. visualization) and applications, with the islands of information. Since there is no common data model and language, there is no common query processor either. Instead, each island has its specific query processor. Query processing within an island is similar to that in multidatabase systems: most of the processing is pushed to the data stores and the query processor only integrates the results. The query optimizer does not use a cost model, but heuristics and some knowledge of the high performance of some data stores. For simple queries, e.g. select-project-join, the optimizer will use function shipping, in order to minimize data movement and network traffic among data stores. For complex queries, e.g. analytics, the optimizer may consider data shipping, to to move the data to a data store that provides a high-performance implementation.

A query submitted to an island may involve multiple islands. In this case, the query must be expressed as multiple subqueries, each in a specific island language. To specify

the island for which a subquery is intended, the user encloses the subquery in a SCOPE specification. Thus, a multi-island query will have multiple scopes to indicate the expected behavior of its subqueries. Furthermore, the user may insert CAST operations to move intermediate datasets between islands in an efficient way. Thus, the multi-island query processing is dictated by the way the subqueries, SCOPE and CAST operations are specified by the user.

2.4.4 Comparative Analysis

The multistore systems we presented above share some similarities, but do have important differences. The objective of this section is to compare these systems along important dimensions and identify the major trends. Although we have not yet presented it, we include CloudMdsQL, which is the focus of this thesis. We divide the dimensions between functionality and implementation techniques.

Multistore system	*Objective*	*Data model*	*Query language*	*Data stores*
Loosely-coupled				
BigIntegrator	Querying relational and cloud data	Relational	SQL-like	BigTable, RDBMS
Forward	Unifying relational and NoSQL	JSON-based	SQL++	RDBMS, NoSQL
QoX	Analytic data flows	Graph	XML-based	RDBMS, MapReduce, ETL
Tightly-coupled				
Polybase	Querying Hadoop from RDBMS	Relational	SQL	HDFS, RDBMS
HadoopDB	Querying RDBMS from Hadoop	Relational	SQL-like (HiveQL)	HDFS, RDBMS
Estocada	Self-tuning	No common model	Native query languages	RDBMS, NoSQL
Hybrid				
SparkSQL	SQL on top of Spark	Nested	SQL-like	HDFS, RDBMS
BigDAWG	Unifying relational and NoSQL	No common model	Island query languages, with CAST and SCOPE operators	RDBMS, NoSQL, Array DBMS, DSMSs
CloudMdsQL	Querying relational and NoSQL	JSON-based	SQL-like with native subqueries	RDBMS, NoSQL HDFS

Table 2.1 – Functionality of multistore systems.

Table 2.1 compares the functionality of multistore systems along four dimensions: objective, data model, query language, and data stores that are supported. Although all multistore systems share the same overall goal of querying multiple data stores, there are many different paths toward this goal, depending on the functional objective to be achieved. And this objective has important impact on the design choices. The major trend

that dominates is the ability to integrate relational data (stored in RDBMS) with other kinds of data in different data stores, such as HDFS (Polybase, HadoopDB, SparkSQL, JEN) or NoSQL (BigTable only for BigIntegrator, document stores for Forward). Thus, an important difference lies in the kind of data stores that are supported. For instance, Estocada, BigDAWG and CLoudMdsQL can support a wide variety of data stores while Polybase and JEN target the integration of RDBMS with HDFS only. We can also note the growing importance of accessing HDFS within Hadoop, in particular, with MapReduce or Spark, which corresponds to major use cases in structured/unstructured data integration.

Another trend is the emergence of self-tuning multistore systems, such as Estocada and Odyssey, with the objective of leveraging the available data stores for performance. In terms of data model and query language, most systems provide a relational/ SQL-like abstraction. However, QoX has a more general graph abstraction to capture analytic data flows. And both Estocada and BigDAWG allow the data capture analytic data flows. And both Estocada and BigDAWG allow the data stores to be directly accessed with their native (or island) languages. CloudMdsQL also allows native queries, but as subqueries within an SQL-like language.

Multistore system	*Special modules*	*Schemas*	*Query processing*	*Query optimization*
Loosely-coupled				
BigIntegrator	Importer, absorber, finalizer	LAV	Access filters	Heuristics
Forward	Query processor	GAV	Data store capabilities	Cost-based
QoX	Dataflow engine	No	Data/function shipping, operation decomposition	Cost-based
Tightly-coupled				
Polybase	HDFS bridge	GAV	Query splitting	Cost-based
HadoopDB	SMS planer, db connector	GAV	Query splitting	Heuristics
Estocada	Storage advisor	Materialized views	View-based query rewriting	Cost-based
Hybrid				
SparkSQL	Catalyst extensible optimizer	Dataframes	In-memory caching using columnar storage	Cost-based
BigDAWG	Island query processors	GAV within islands	Function/data shipping	Heuristics
CloudMdsQL	Query planner	No	Bind join	Cost-based

Table 2.2 – Implementation techniques of multistore systems.

Table 2.2 compares the implementation techniques of multistore systems along four dimensions: special modules, schema management, query processing, and query optimization. The first dimension captures the system modules that either refine those of the generic architecture (e.g. importer, absorber and finalizer, which refine the wrapper module, Catalyst extensible optimizer or QoX's data flow engine, which replace the

query processor) or bring new functionality (e.g. Estocada's storage advisor). Most multistore systems provide some support for managing a global schema, using the GAV or LAV approaches, with some variations (e.g. BigDAWG uses GAV within (single model) islands of information). However, QoX, Estocada, SparkSQL and CloudMdsQL do not support global schemas, although they provide some way to deal with the data stores local schemas.

The query processing techniques are extensions of known techniques from distributed database systems, e.g. data/function shipping, query decomposition (based on the data stores's capabilities, bind join, select pushdown). Query optimization is also supported, with either a (simple) cost model or heuristics.

2.5 Conclusion

In this chapter, we gave an overview of query processing in multistore systems, focusing on the main solutions and trends. We started by introducing cloud data management, including distributed file systems such as HDFS, NoSQL systems and data processing frameworks (such as MapReduce and Spark) and query processing in multidatabase systems. Then, we described and analyzed representative multistore systems, based on their architecture, data model, query languages and query processing techniques. To ease comparison, we divided multistore systems based on the level of coupling with the underlying data stores, i.e. loosely-coupled, tightly-coupled and hybrid.

We analyzed three multistore systems for each class: BigIntegrator, Forward and QoX (loosely-coupled); Polybase, HadoopDB and Estocada (tightly-coupled); CloudMdsQL, SparkSQL and BigDAWG (hybrid). Our comparisons reveal several important trends. The major trend that dominates is the ability to integrate relational data (stored in RDBMS) with other kinds of data in different data stores, such as HDFS or NoSQL. However, an important difference between multistore systems lies in the kind of data stores that are supported. We also note the growing importance of accessing HDFS within Hadoop, in particular, with MapReduce or Spark. Another trend is the emergence of self-tuning multistore systems, with the objective of leveraging the available data stores for performance. In terms of data model and query language, most systems provide a relational/ SQL-like abstraction. However, QoX has a more general graph abstraction to capture analytic data flows. And both Estocada and BigDAWG allow the data stores to be directly accessed with their native (or island) languages.

The query processing techniques are extensions of known techniques from distributed database systems, e.g. data/function shipping, query decomposition (based on the data stores capabilities, bind join, select pushdown). And query optimization is supported, with either a (simple) cost model or heuristics.

Chapter 3

Design of CloudMdsQL

In this chapter, we present the design of the Cloud Multidatastore Query Language (CloudMdsQL), and its query engine. CloudMdsQL is a functional SQL-like language, capable of querying multiple heterogeneous data stores (relational and NoSQL) within a single query that may contain embedded invocations to each data store's native query interface. The major innovation is that a CloudMdsQL query can exploit the full power of local data stores, by simply allowing some local data store native queries (e.g. a breadth-first search query against a graph data store) to be called as functions, and at the same time be optimized based on a simple cost model, e.g. by pushing down select predicates, using bind join, performing join ordering, or planning intermediate data shipping. This chapter is based on [40].

This chapter is organized as follows. Section 3.1 gives an overview of the chapter. Section 3.2 discusses related work in more details. Section 3.3 introduces CloudMdsQL's basic concepts, including its data model and language constructs. Section 3.4 presents the architecture of the query engine and its main components. Section 3.5presents the language in more details. Section 3.6 reveals the query processing steps. Section 3.7 gives an example walkthrough. Section 3.8 concludes.

3.1 Overview

The state of the art solutions for multidatabase systems [24, 46] (see Section 3.2) do not directly apply to multistore systems. First, our common language is not for querying data stores on the web, which could be in very high numbers. A query should be on a few cloud data stores (perhaps less than 10) and the user needs to have access rights to each data store. Second, the data stores may have very different languages, ranging from very simple get/put in key-value stores, to full SQL or SPARQL languages. And no single language can capture all the others efficiently, e.g. SQL cannot express graph path traversal (of course, we can represent a graph with two relations Edges and Nodes, but this requires translating path traversals into expensive joins). Even a graph query language, which is very general, cannot capture an array data model easily. Third, NoSQL data stores can be without schema, which makes it (almost) impossible to derive a global

schema. Finally, and very important, what the user needs is the ability to express powerful queries to exploit the full power of the different data store languages, e.g. directly express a path traversal in a graph data store. For this, we need a new query language.

We can translate these observations into five main requirements for our common language:

1. To integrate fully-functional queries against different NoSQL and SQL data stores using each data store's native query mechanism;

2. To allow nested queries to be arbitrarily chained together in sequences, so the result of one query (for one data store) may be used as the input of another (for another data strore);

3. To be schema independent, so that data stores without or with different schemas can be easily integrated;

4. To allow data-metadata transformations, e.g. to convert attributes or relations into data and vice versa [61];

5. To be easily optimizable so that efficient query optimization, introduced in state of the art multidatabase systems, can be reused (e.g. exploiting bind joins [32] or shipping the smallest intermediate results).

The CloudMdsQL, and its query engine, addresses these requirements. While the latter four have already been identified as requirements and introduced in multidatabase mediator-wrapper architectures, CloudMdsQL contributes to satisfying also the first one. The language is capable of querying multiple heterogeneous data stores (e.g. relational and NoSQL) within a single query containing nested subqueries, each of which refers to a particular data store and may contain embedded invocations to the data store's native query interface.

The design of the query engine takes advantage of the fact that it operates in a cloud platform. Unlike the traditional mediator-wrapper architectural model where mediator and wrappers are centralized, we propose a fully distributed architecture that yields important optimization opportunities, e.g. minimizing data transfers between nodes. This allows us to reuse query decomposition and optimization techniques from distributed query processing [46].

3.2 Related Work

The common data model and query language used by a mediator (see Figure 2.5) have a major impact on the effectiveness of data store integration. The two dominant solutions today, with major product offerings, are relational/SQL and XML Xquery, each having its own advantages. The relational model provides a simple data representation (tables) for mapping the data stores, but with rigid schema support. The major advantage of a

relational solution is that SQL is familiar to users and developers, with SQL APIs used by many tools, e.g. in business intelligence. Furthermore, recent extensions of SQL such as SQL/XML include support for XML data types. On the other hand, the XML model provides a tree-based representation that is appropriate for Web data, which are typically semi-structured, and flexible schema capabilities. As a particular case, XML can represent relational tables, but at the expense of more complex parsing. XQuery is now a complete query language for XML, including update capabilities, but more complex than SQL. As a generalization for Web linked data, there is also current work based on RDF/SPARQL [33]. There is still much debate on relational versus XML, but in the cloud, relational-like data stores, e.g. NoSQL key-value stores such as Google Bigtable and Hadoop Hbase, are becoming very popular, thus making a relational-like model attractive.

The main requirements for a common query language (and data model) are support for nested queries, schema independence and data-metadata transformation [61]. Nested queries allow queries to be arbitrarily chained together in sequences, so the result of one query (for one data store) may be used as the input of another (for another data store). Schema independence allows the user to formulate queries that are robust in front of schema evolution. Data-metadata transformation is important to deal with heterogeneous schemas by transforming data into metadata and conversely, e.g. data into attribute or relation names, attribute names into relation names, relation names into data. These requirements are not supported by query languages designed for centralized data stores, e.g. SQL and XQuery. Therefore, federated query languages need major extensions of their centralized counterpart.

We now discuss briefly two kinds of such extensions of major interest: relational languages and functional SQL-like languages. In [61], the authors propose an extended relational model for data and metadata integration, the Federated Relational Data Model, with a relational algebra, Federated Interoperable Relational Algebra (FIRA) and an SQL-like query language that is equivalent to FIRA, Federated Interoperable Structured Query Language (FISQL). FIRA and FISQL support the requirements discussed above, and the equivalence between FISQL and FIRA provides the basis for distributed query optimization. FISQL and FIRA appear as the best extensions of SQL-like languages for data and metadata integration. In particular, it allows nested queries. But as with SQL, it is not possible to express some complex control on how queries are nested, e.g. using programming language statements such as IF THEN ELSE, or WHILE. Note that, to express control over multiple SQL statements, SQL developers typically rely on an imperative language such as Java in the client layer or a stored procedure dialect such as PLSQL in the data store layer. Another major limitation of the relational language approach is that it does not allow exploiting the full power of the local data store repositories. For instance, mapping an SQL-like query to a graph data store query will not exploit the graph DBMS capabilities, e.g. generating a breadth-first search query.

Database programming languages (DBPLs) have been proposed to solve the infamous impedance mismatch between programming language and query language. A functional language has several advantages for accessing heterogeneous data stores. First, nested queries and complex control can be easily supported. Second and more important, the

full power of the local data store repositories could be exploited, by simply allowing local data store queries, e.g. a breadth-first search query, to be called as native functions. In particular, functional DBPLs such as FAD [20] can represent all query building blocks as functions and function results can be used as input to subsequent functions, thus making it easy to deal with nested queries with complex control. The first SQL-like functional DBPL is Functional SQL [57]. However, DBPLs are also full-fledge programming languages, aimed to develop complex data-intensive applications. This generality makes them hard to optimize [35]. But for accessing heterogeneous data stores in the cloud, we do not need a full-fledge DBPL. More recently, FunSQL [12] has been proposed for the cloud, to allow shipping the code of an application to its data. Another popular functional DBPL is LINQ [44], whose goal is to reconcile object-oriented programming, with relations and XML. LINQ allows any .NET programming language to manipulate general query operators (as functions) with two domain-specific APIs that work over XML (XLinq) and relational data (DLinq) respectively. The operators over relational data provide a simple object-relational mapping that makes it easy to specify wrappers to the underlying RDBMS.

3.3 Basic Concepts

The common querying model targets integration of data from several stores based on a diverse set of data models, mediating the data through a common data model. Consequently, the common query language and its data model are designed to achieve such data integration accordingly.

3.3.1 Data Model

CloudMdsQL sticks to the relational data model, because of its intuitive data representation, wide acceptance and ability to integrate datasets by applying joins, unions and other relational algebra operations. To be robust against schema evolution and driven by the fact that NoSQL data stores can be schema-less, CloudMdsQL keeps its common data model schema-less, while at the same time it is designed to ensure that all the datasets retrieved from the data stores match the common data model.

Operators. The common data model supports basic relational operators (projection, selection, joins, aggregation, sorting and set operations). In addition, in order to satisfy the common language requirements, the data model includes another two operators as follows. First, to support data and data-metadata transformations, CloudMdsQL introduces a Python operator that can perform user-defined operations over intermediate relations and/or generate synthetic data by executing embedded code of the programming language Python. Second, the requirement for running optimal nested queries from heterogeneous data stores implies the usage of the bind join method [32], which uses the data retrieved from one data store to rewrite the query to another data store, so that from the latter are retrieved only the tuples that match the join criteria.

Data Types. The CloudMdsQL data model supports a minimal set of data types, enough to capture data types supported by the data models of most data stores: scalar data types – integer, float, string, binary, timestamp; composite data types – array, dictionary (associative array); null values. Standard operations over the above data types are also available: arithmetic operations, concatenation and substring, as well as operations for addressing elements of composite types (e.g. array[index] and dictionary['key']). Type constructors for the composite data types follow the well-known JSON-style constructors: an array is expressed as a comma separated list of values, surrounded by brackets; a dictionary is expressed as a comma separated list of key:value pairs, surrounded by curly braces.

3.3.2 Language Concepts

The CloudMdsQL language itself is SQL-based with the extended capabilities for embedding native queries to data stores and embedding procedural language constructs. The involvement of the latter is necessitated by the requirement for performing data and data-metadata transformations and conversions of arbitrary datasets to relations in order to comply with the common data model. To support such procedural programmability, CloudMdsQL queries can contain embedded constructs of the programming language Python, the choice of which is justified by its richness of data types, native support for iteration with generator functions, ease of use, richness in standard libraries and wide usage. An important concept in CloudMdsQL is the notion of "table expression", inspired from XML table views [30, 43], which is generally an expression that returns a table (i.e. a structure, compliant with the common data model). A table expression is used to represent a nested query and usually addresses a particular data store. Three kinds of table expressions are distinguished:

- Native table expressions, using a data store's native query mechanism;
- SQL table expressions, which are regular nested SELECT statements;
- Embedded blocks of Python statements that produce relations.

A table expression is usually assigned a name and a signature, thus turning it into a "named table expression", which can be used in the FROM clause of the query like a regular relation. Named table expression's signature defines the names and types of the columns of the returned relation. Thus, each CloudMdsQL query is executed in the context of an ad-hoc schema, formed by all named table expressions within the query. This approach fills the gap produced by the lack of a global schema and allows the query compiler to perform semantic analysis of the query. A named table expression is usually defined as a query against a particular data store and contains references to the data store's data structures. However, the expression can also instantiate other named table expressions, defined against other data stores, thus chaining data as per the requirement for nesting queries.

For example, the following simple CloudMdsQL query contains two subqueries, defined by the named table expressions T1 and T2, and addressed respectively against the data stores DB1 (an SQL data store) and DB2 (a MongoDB data store):

```
T1(x int, y int)@DB1 = ( SELECT x, y FROM A )
T2(x int, z string)@DB2 = {*
   db.B.find( {$lt:  {x, 10}}, {x:1, z:1, _id:0} ) *}
SELECT T1.x, T2.z
FROM T1, T2
WHERE T1.x = T2.x AND T1.y <= 3
```

The purpose of this query is to perform relational algebra operations (expressed in the main SELECT statement) on two datasets retrieved from a relational and a document data store. The two subqueries are sent independently for execution against their data stores in order the retrieved relations to be joined by the common query engine. The SQL table expression `T1` is defined by an SQL subquery, while `T2` is a native expression using a MongoDB query that retrieves from the document collection B the attributes x and z of those documents for which `x < 10`. The subquery of expression `T1` is subject to rewriting by pushing into it the filter condition `y <= 3`, specified in the main SELECT statement, thus reducing the amount of the retrieved data by increasing the subquery selectivity. The so retrieved datasets are then converted to relations following their corresponding signatures, so that the main CloudMdsQL SELECT statement can be processed with semantic correctness.

3.4 Query Engine Architecture

Although the focus of this chapter is on the design of the CloudMdsQL language, we still need to show how queries can be optimized and processed in a cloud environment. Thus, in this section, we introduce the architecture of the query engine, with its main components, and briefly introduce query processing, which will be more detailed in Sections 3.6 and 3.7. We ignore fault-tolerance, which is out of the scope of this chapter.

3.4.1 Overview

The design of the query engine takes advantage of the fact that it operates in a cloud platform, with full control over where the system components can be installed. This is quite different from web data integration systems for instance, where both mediator and data store wrappers can only be installed at one or more servers that communicate with the data stores through the network. In our context, the query engine is part of a more general platform (CoherentPaaS) that allows deployment over one or more data centers. For simplicity in this chapter, we consider the case of a single data center, i.e. a computer

cluster.

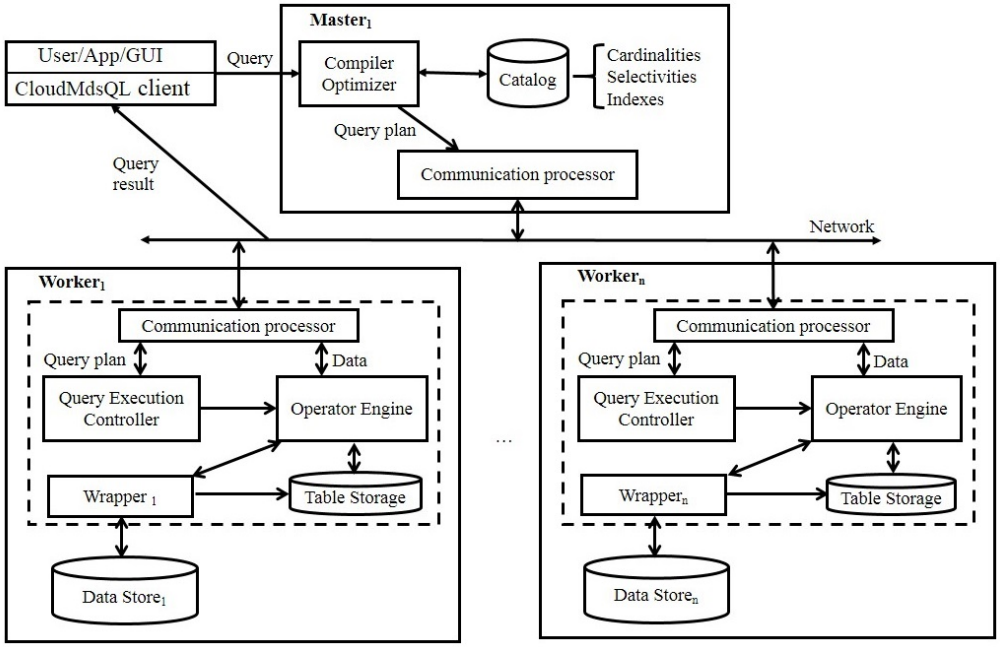


Figure 3.1 – **Architecture of the query engine**

The architecture of the query engine is fully distributed (see Figure 3.1), so that query engine nodes can directly communicate with each other, by exchanging code (query plans) and data. Thus, the query engine does not follow the traditional mediator-wrapper architectural model where mediator and wrappers are centralized. This distributed architecture yields important optimization opportunities, e.g. minimizing data transfers by moving the smallest intermediate data for subsequent processing by one particular node.

Each query engine node consists of two parts – master and worker – and is collocated at each data store node in a computer cluster. Each master or worker has a communication processor that supports send and receive operators to exchange data and commands between nodes. To ease readability in Figure 3.1, we separate master and worker, which makes it clear that for a given query, there will be one master in charge of query planning and one or more workers in charge of query execution. To illustrate query processing with a simple example, let us consider a query Q on two data stores in a cluster with two nodes (e.g. the query introduced in Section 3.3.2). Then a possible scenario for processing Q, where the node id is written in subscript, is the following:

- At client, send Q to Master_1.
- At Master_1, produce a query plan P (see Figure 3.2) for Q and send it to Worker_2, which will control the execution of P.

- At $Worker_2$, send part of P, say P_1, to $Worker_1$, and start executing the other part of P, say P_2, by querying $Data\ Store_2$.
- At $Worker_1$, execute P_1 by querying $Data\ Store_1$, and send result to $Worker_2$.
- At $Worker_2$, complete the execution of P_2 (by integrating local data with data received from $Worker_1$), and send the final result to the client.

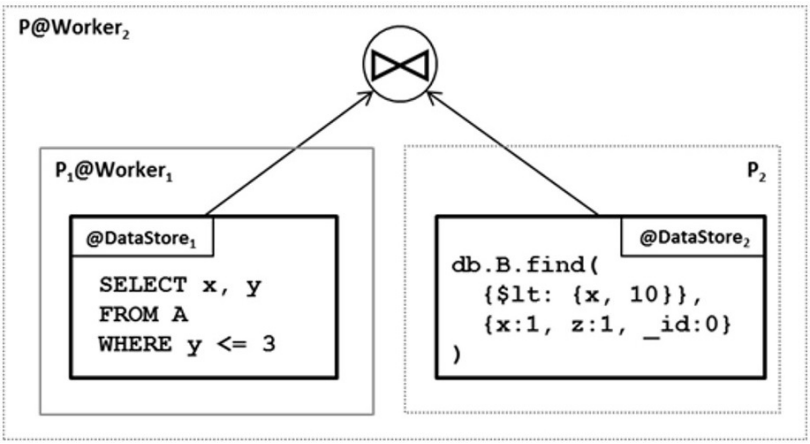


Figure 3.2 – **A simple query plan**

This simple example shows that query execution can be fully distributed among the two nodes and the result sent from where it is produced directly to the client, without the need for an intermediate node.

3.4.2 Master

Since there are multiple masters (one at each cluster node), the client chooses one of them to send a query to. Although load balancing is not crucial as masters do not carry heavy loads, we can still do it using a random pick or a simple round robin process at the client side to distribute queries across masters.

A master takes as input a query and produces a query plan, which it sends to one chosen query engine node for execution. The query planner performs query analysis and optimization, and produces a query plan serialized in a JSON-based intermediate format that can be easily transferred across query engine nodes. Each operation in the plan carries the identifier of the query engine node that is in charge of performing it. Thus, the topmost operation determines the first worker, to which the master should send the query plan. As for declarative query languages (e.g. SQL), a query plan can be abstracted as a tree of CloudMdsQL operators and communication (send/receive) operators to exchange data and commands between query engine nodes. This allows us to reuse query decomposition and optimization techniques from distributed query processing [46], which we adapt to our fully distributed architecture. In particular, we strive to:

- Minimize local execution time in the data stores, by pushing down select operations in the data store subqueries and exploiting bind join by query rewriting;
- Minimize communication cost and network traffic by reducing data transfers between workers.

To compare alternative rewritings of a query, the query planner uses a simple catalog, which is replicated at all nodes in primary copy mode. The catalog provides basic information about data store collections such as cardinalities, attribute selectivities and indexes, and a simple cost model. Such information can be given with the help of the data store administrators. The query language provides a possibility for the user to define cost and selectivity functions whenever they cannot be derived from the catalog, mostly in the case of using native subqueries. The search space explored for optimization is the set of all possible rewritings of the initial query, by pushing down select operations, expressing bind joins, join ordering, and intermediate data shipping. Unlike in traditional query optimization where many different permutations are possible, this search space is not very large, so we can use a simple exhaustive search strategy.

3.4.3 Worker

Workers collaborate to execute a query plan, produced by a master, against the underlying data stores involved in the query. As illustrated in Section 3.4.2, there is a particular worker, selected by the query planner, which becomes in charge of controlling the execution of the query plan. This worker can subcontract parts of the query plan to other workers and integrate the intermediate results to produce the final result.

Each worker node acts as a lightweight runtime data store processor atop a data store and is composed of three generic modules (i.e. same code library) - query execution controller, operator engine, and table storage - and one wrapper module that is specific to a data store. These modules provide the following capabilities:

- Query execution controller: initiates and controls the execution of a query plan (received from a master or worker) by interacting with the operator engine for local execution or with one or more workers (through communication processors) in case part of the query plan needs to be subcontracted. In the latter case, the query execution controller will synchronize the execution of the operator(s) that require the intermediate results produced by the distant workers, once they are received back.
- Operator engine: executes the query plan operators on data retrieved from the wrapper, from another worker, or from the table storage. These operators include CloudMdsQL operators to execute table expressions in the query and communication (send/receive) operators to exchange data with other workers. Some operators are simply passed on to the wrapper for producing intermediate results from the data store. The operator engine may write intermediate relations to the table storage.

- Table Storage: provides efficient, uniform storage (main memory and disk) for intermediate and result data in the form of tables. Storage of intermediate data is necessary in particular cases, e.g. when an intermediate relation needs to be consumed by more than one operator or when it participates in a blocking operation such as aggregation, sorting or nested-loop join. In other cases, intermediate relations are directly pulled by the consuming operator.

- Wrapper: interacts with its data store through its native API to retrieve data, transforms the result in the form of table, and writes the result in table storage or delivers it to the operator engine. To query its data store, each wrapper is invoked by the operator engine through generic interface methods, which it maps to data store specific API calls. Wrappers are discussed in more detail in Section 3.6.4.

3.5 Query Language

The major innovation of CloudMdsQL refers to the involvement of native subqueries and the way both SQL and native subqueries interoperate with each other to provide the desired coherence across all data stores. In this section we provide details about how multiple diverse data stores can be queried through CloudMdsQL by means of nested table expressions.

Named table expressions are definitions of temporary (at query level) tables representing nested subqueries against data stores and their signatures define the names and types of the attributes of the returned relations. Within a single CloudMdsQL query, all named table expressions form an ad-hoc schema, in the context of which the main SELECT statement of the query is processed and its semantic correctness is verified. Embedded Python constructs that can be used to define Python named table expressions necessitate the involvement of special conventions, the usage of which provides the required query expressivity and ability for nesting subqueries.

3.5.1 Named Table Expressions

Named table expressions are defined in the header of a CloudMdsQL query, preceding the main SELECT statement, and are instantiated in the FROM clause and/or from the definitions of other named expressions. The basic syntax of a named table expression is the following:

```
<expr-name>(<colname> <type>, ...)[@<datastore>]=<expr-def>
```

The declaration consists of the name of the expression, followed by its signature, which specifies the names and types of the attributes of the result relation, reference to the underlying data store, which the subquery is addressed to, and expression definition. An SQL expression definition should be surrounded by parentheses, which implies that the compiler processes it and transforms it to a subquery plan, part of the global execution plan, and therefore is subject to analysis, optimization and/or rewriting. A native/Python

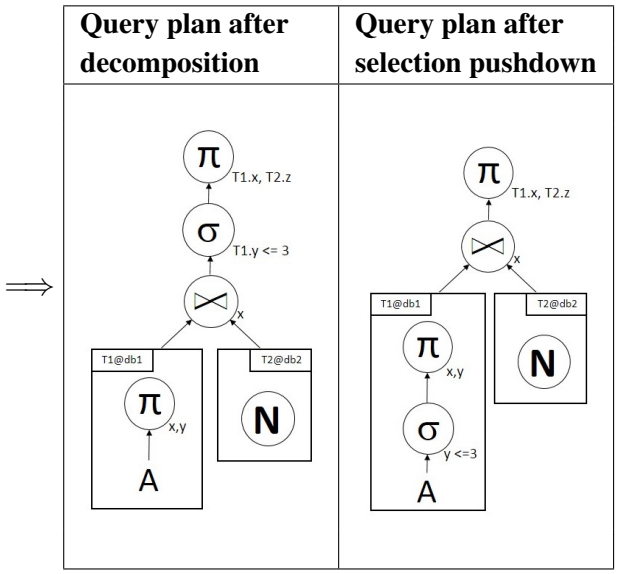
expression definition must be surrounded by native expression brackets, which is the following pair of opening / closing bracket symbols: {*...*}. Named table expressions are classified according to the way they interface the underlying data stores and/or intermediate relations, as follows.

Native named table expressions represent subqueries to data stores using their native query mechanism. They are executed in the context of a particular connection to a data store. The expression definition is a native query or code that should contain invocations to the native interface of the data store and produce a relation with the declared signature. The code is surrounded by native expression brackets, which gives information to the query engine not to process the contained expression but pass it as a black box to the corresponding wrapper. However, a native expression can still use as input intermediate data retrieved by other named table expressions, thus providing full capability for nesting queries. The query engine allows this by exposing the query execution context to the wrappers, like it does for Python expressions (see below).

SQL named table expressions are expressed as regular SELECT statements. They are quite different from native expressions, since they are compiled and analyzed by the query planner, as opposed to native expressions which are considered as black boxes and are not subject to analysis. An SQL expression against a data store contains in its FROM clause references to the data store tables. However, to provide support for nested querying, each SQL expression can also instantiate other named table expressions from the context of the current CloudMdsQL query (nested SQL queries are more detailed in Section 3.5.2.1). Furthermore, each data store subquery, expressed as an SQL expression, is subject to rewriting, whenever selection pushdowns or bind joins take place.

CloudMdsQL query

```
T1(x int, y int)@DB1={
SELECT x, y, z FROM A}
T2 (x int, z string)@DB2= {*
db.B.find({$lt:{x,10}},
{x:1, z:1, _id:0}) *}
SELECT T1.x, T2.z
FROM T1, T2
WHERE T1.x = T2.x
AND T1.y <= 3
```



To illustrate the usage of native and SQL table expressions and give a basic notion of how they are handled by the query planner, let us come back to the example, introduced in Section 3.3.2. The CloudMdsQL query below contains an SQL named table expression

T1 and a native named table expression T2. The query plan after decomposition shows that the SQL expression T1 is decomposed to a sub-plan assigned to the data store db1, while the sub-plan for db2 contains only a single node, corresponding to the definition of the native expression T2. Thus, the sub-plan for db1 may be modified by the planner, e.g. by pushing operations into it, as it is shown with the plan after selection pushdown. All the query processing steps are detailed in Section 3.6.

3.5.2 Nested Queries

As stated in the language requirements, CloudMdsQL must provide a mechanism for nesting queries – i.e. a named table expression must be able to instantiate other named table expressions, available in the execution context of the same query, and use their result sets as input. This is achievable in all types of expressions: in native/Python expressions by invoking the CloudMdsQL object, and in SQL expressions by simply referencing named table instantiations directly in the FROM clause, often in combination with references to the data store's tables.

3.5.2.1 Within SQL Expressions

An SQL expression against a data store contains references to data store tables, but may also refer to named table expressions in its FROM clause. If the SQL expression contains such mixed references, its corresponding subquery plan is split by the query compiler into two sub-plans. The first one contains only references to data store tables and is identified as a subquery plan that will be passed to the wrapper. The other one references only the root node of the first sub-tree and instantiations of other named table expressions from the context of the CloudMdsQL query and will be executed by the query engine as part of the common execution plan. This is illustrated with the following example:

Original query	**Rewritten equivalent query**
`T1(x int, y int)@DB2={*`	`T1(x int, y int)@DB2={*`
`db.B.find({$lt:{x,10}},`	`db.B.find({$lt:{x,10}},`
`{x:1,y:1,_id:0}) *}`	`{x:1,y:1,_id:0}) *}`
`T2 (x int, y int, z string)@DB1=(`	`T2(x int, y int, z string)@DB1=(`
`SELECT A.x, T1.y, B.z`	`SELECT A.x, B.z`
`FROM A JOIN B ON A.id=B.id`	`FROM A JOIN B ON A.id=B.id)`
`JOIN T1 ON A.x=T1.x)`	`SELECT T2.x, T1.y, T2.z`
`SELECT x, y, z FROM T2`	`FROM T2` **`JOIN T1 ON T2.x=T1.x`**

Here the query planner, upon parsing the original query and building the subquery plan for T2, detects the usage of the named table T1, plans the join with T1 as the outermost operation within the sub-plan, and pulls it into the common plan, thus transforming the whole query plan to correspond to the rewritten equivalent query above. In some more complex cases, the planner may not be able to place as outermost all the operations that

involve named table expressions; in such cases, the planner will split the sub-plan in order to be able to pull such operations from the sub-plan, which may result in building more than one sub-plans that originate from a single subquery.

Another nested query scenario is when a named table is referred in a sub-select statement within the subquery, thus making the result set of the named table an input to the subquery, as in the following example:

```
T1(x int, y int)@DB2 ={* db.B.find({$lt:{x,10}},{x:1,y:1,_id:0}) *}
T2(x int, z string)@DB1 = (
   SELECT A.x, B.z FROM A JOIN B ON A.id = B.id
   WHERE A.x IN (SELECT x FROM T1 WHERE y > 0) )
SELECT x, z FROM T2
```

To process the subquery `T2`, the query engine must first retrieve the table `T1`, evaluate the sub-select `SELECT x FROM T1 WHERE y > 0`, and then transform it to a list of the distinct values of `T1.x` to replace the sub-select with that list of values. This is similar to the processing of bind joins, which is explained in detail in Section 3.6.2.

3.5.2.2 Within Native Expressions

This section focuses on the capability of nesting subqueries within native/Python expressions. CloudMd sQL introduces two approaches that allow the programmer to write expression definitions that iterate through data retrieved by other subqueries – **table iteration** and **join iteration.**

With **table iteration**, the Python code of a table expression can iterate through the result set of another table expression by requesting a forward iterator through the `CloudMdsQL` object, instantiating the iterated table by its name. Because of the forward iteration pattern and due to the pipelining fashion of the query execution, the Python expression will start consuming tuples once a few tuples of the iterated table are available, without having to wait for the entire table to be retrieved. To build a relevant and adequate QEP, the query compiler needs to identify all dependencies between table expressions, i.e. for each named expression, the engine needs to know which other named table expressions it iterates through. For native/ Python expressions, since a black-box approach is used, the query engine does not perform any processing of the code; therefore the referenced inside the expression tables must be explicitly specified in the expression's signature. CloudMdsQL provides an additional `REFERENCING` clause, by which the programmer specifies that the expression definition performs iterations through a named table instantiation.

For example, let us consider the following query, assuming that DB1 is a relational data store with a table `person`, containing names and addresses of persons, and DB2 is a graph data store with Python API providing the function `GetShortest Distance`, which finds the shortest distance between two cities. Now we want to query both data stores to retrieve persons who work in department Herault, the cities where they live and work and what is the distance between their home and work cities.

```
person_herault(name string, h_city string, w_city string)@DB1 = (
   SELECT name, home_city, work_city
   FROM person p
   WHERE work_dept = 'Herault' )
person_distance(name string,h_city string,w_city string,distance int
   REFERENCING person_herault)@DB2 =
{* for (n, hc, wc) in CloudMdsQL.person_herault:
        yield ( n, hc, wc, GetShortestDistance(hc, wc) ) *}
SELECT name, h_city, w_city, distance FROM person_distance;
```

The execution flow of the above query is quite straightforward. It contains specialized subqueries which are chained in a strict way – first the table `person_herault` is retrieved for persons who work in Herault; then its dataset is used as input to the other subquery, the result of which is the table `person_distance` that contains one more column retrieved from the graph data store by calling its function `GetShortestDistance`; and finally a projection in the main SELECT statement defines the format of the result table. This approach provides good functionality because it allows arbitrary chaining of data across subqueries. But it tends to involve less flexible queries, because it does not allow selection pushdown, and hence requires specialized subqueries like `person_herault`, where the filter condition must be specified in the subquery.

The **join iteration** approach is applicable for any native/Python table expression that is one of the sides of an equi-join. The query execution requires that the other side of the join (we will call it "the outer relation") is evaluated first, so that the native/Python expression can generate its tuples by iterating through the values of the join attribute(s) of the outer relation. Thus, only tuples that match the join criteria are generated. This approach also allows for a native/Python subquery to use as input the result set of another subquery, but in a different way – in combination with a join operation. For example, the results from the above query can be retrieved using join iteration by the following query:

```
person(name string,h_city string,w_city string,w_dept string)@DB1 = (
   SELECT name, home_city, work_city, work_dept
   FROM person p )
distance(city_1 string, city_2 string, distance int
   JOINED ON city_1, city_2)@DB2 =
{* for (c1, c2) in CloudMdsQL.Outer:
        yield ( c1, c2, GetShortestDistance(c1, c2) ) *}
SELECT p.name, p.h_city, p.w_city, d.distance
FROM person p JOIN distance d
   ON p.h_city = d.city_1 AND p.w_city = d.city_2
WHERE p.w_dept = 'Herault';
```

The first thing to notice here is that the subqueries are more generic – the table ex-

pression `person` represents a projection over relational data without filters; the table expression `distance` defines a relation where each tuple consists of a pair of cities and the distance between them. And the whole query is more manipulable, because the filter condition `w_dept = 'Herault'` is specified in the main SELECT statement, but it can be pushed down into the subquery. Thus, if the two named table expressions are stored in the global catalog, they can be reused in a wider range of queries.

The JOINED ON clause in the signature of the Python expression declares that whenever the table `distance` participates in an equi-join with another relation on the attributes specified in the clause, the expression will generate its tuples by iterating through the values of the join attributes of the outer relation. The query is processed as follows. First, the subquery against DB1 is rewritten by adding the condition `work_dept = 'Herault'` and removing `work_dept` from the projection (it is not needed for the execution of the common query plan). Then, the subquery is executed and the query engine starts retrieving from DB1 tuples that form the result set of the outer relation. Then, the wrapper of DB2 starts the execution of the Python code that queries the graph data store. It consumes a projection on the attributes `h_city` and `w_city` of the outer relation, iterating through it via the special iterator object `CloudMdsQL.Outer`, and generates the tuples of its own result set.

To handle join iteration, the operator engine pipelines the join attribute values of the outer relation to the iterator object, which allows for the native/Python expression to start immediately iterating through them as soon as a few tuples are available, without having to wait for the entire outer relation to be retrieved. Once a tuple is generated by the native/Python expression, the operator engine immediately joins it with its corresponding tuple from the outer relation, thus performing the join on-the-fly with minimal cost. During the join execution, a hash map is maintained, where each already iterated join attribute value is mapped to zero or more tuples generated by the native/Python expression. Thus, the iteration is performed over a set of distinct values of the join attribute(s) of the outer relation, which saves from duplicate invocations of native API functions that can be expensive (e.g. `GetShortestDistance`).

3.5.3 CloudMdsQL SELECT Statement

SELECT queries in CloudMdsQL retrieve data from several data stores using embedded subqueries (for each data store) and integrate the data to build the result dataset. The CloudMdsQL SELECT statement looks like a typical SQL SELECT statement but supplements it with a header containing definitions of named table expressions:

```
[<named-table-expr> ...]
SELECT <column_list>
<from_clause>
[<where_clause>]
[<group_clause>]
[<having_clause>]
```

```
[<order_clause>]
[<limit_clause>]
```

Some of the clauses have CloudMdsQL specifics. `[<named-table-expr>...]` is an optional list of named table expressions as per the corresponding syntaxes described above. Names of table expressions must be unique within both the local (in the same query) and global (stored named expressions) context. The generic syntax of a named table expression definition is presented below.

```
<expr-name>(<colname> <type>, ...
       [WITHPARAMS <paramname> <type>, ...]
       [REFERENCING <tablename>, ...]
       [JOINED ON <colname>, ...]
       [CARDINALITY = <cardinality_function>]
       [SELECTIVITY(<colname>, ...)  = <selectivity_function>]
)[@<datastore>] = <expr-def>
```

Its signature may contain certain optional clauses, as follows. The `WITHPARAMS` clause specifies the names and types of the parameters, if any. The `REFERENCING` clause specifies the names of other named table expressions that are used within a native named table expression. The `JOINED ON` clause specifies the names of the columns of the table expression on which a join iteration method will be performed. The `CARDINALITY` clause specifies a user-defined cost function that can be used by the optimizer to estimate the expected cardinality of the named expression's result set. The function is expressed as an arithmetic expression that may refer to the cardinalities of the referenced named tables, e.g. `card(T1)`, and/or any of the named table expression's parameters. Similarly, a `SELECTIVITY` function may also be defined, which can give an estimate of the expected selectivity of an equality condition on the specified columns.

`<from_clause>` is a regular SQL FROM clause containing references to named table expressions – global or ad-hoc, parameterized or not. If a table refers to a parameterized expression, parameter values should be specified in parentheses. The FROM clause may contain JOIN expressions, specifying explicit join ordering and conditions. The JOIN keyword may be followed or preceded by execution directive in parentheses, which will override optimizer's decisions and will explicitly make the query engine perform a concrete method (e.g. bind join, hash, merge or nested-loop).

In the `<where_clause>` there can be specified a filter predicate expression. The query compiler will transform it to normal conjunctive form, thus determining the exact selection operations to be performed as part of the execution plan. The optimizer will then find the most appropriate place of each selection operation and push it down as much as possible in the execution plan tree. This optimization can finally result in rewriting subqueries to data stores by adding filter conditions, if the optimizer finds an opportunity to increase the selectivity of the subquery. However, only subqueries defined with SQL named table expressions can benefit from such an optimization.

3.6 Query Processing

In this section, we briefly describe in more detail the different steps of CloudMdsQL query processing, according to the query engine architecture (see Section 3.4), i.e. query decomposition, optimization and execution. We also discuss the details of interfacing data stores through wrappers. We end with a use case example showing the different query processing steps.

3.6.1 Query Decomposition

During query decomposition, the query planner compiles the query and builds a preliminary query execution plan (QEP). A query plan in its simplest form is a tree structure, representing relational operations, where the leaf nodes are references to tables, results from the execution of the subqueries against data stores. At this step, the planner also prepares a set of native queries which will be passed to the corresponding wrappers and hence to the underlying data stores (this process will be explained later). Each node of the query plan represents a relational operation and an intermediate relation, resulting from the operation. For more complex queries, since the language allows a single named table expression to be used as operand to several operations (e.g. referenced in other named table expressions and also in the main SELECT statement), it is possible for an intermediate relation to be the input of more than one operator, therefore the query plan appears to be a directed acyclic graph rather than a tree structure. If cyclic references exist, they will be discovered by the query engine at decomposition time and the query will be rejected.

While building the execution strategy, the planner identifies a forest of sub-trees within the query plan, each of which is associated to a certain data store. Each of these sub-plans is meant to be delivered to the corresponding wrapper, which has to translate it to a native query and execute it against its data store (for SQL subqueries, this process is more detailed in Section 3.6.4.1). The rest of the QEP is the part that will be handled by the query engine. So now we outline two main subsets of the global execution plan: (1) a forest of sub-trees that will be executed locally by each data store and (2) a common query plan that will be executed by the query engine with leaf nodes consuming the relations returned by each wrapper as result of sub-plan execution. At query decomposition step, the boundary between the two subsets is preliminary and may be modified during the query optimization step by pushing operations from the common plan to sub-plans to improve the overall execution efficiency or by pulling operations from sub-plans to the common plan in case a data store is not capable of handling them.

The next step of the decomposition is the semantic analysis of the query. Within only the common query plan, all table and column names are verified against the query's ad-hoc schema. On the other hand, since the query engine is agnostic to the underlying data stores' schemas, it does not perform semantic analysis of sub-plans, presuming that this will be done by the data stores upon handling each subquery's native equivalent. In fact, all the sub-plans are kept as abstract syntax trees and are never transformed into execution plans. Thus, the query engine is exempt from gathering full metadata from data stores,

except those metadata needed by the optimizer, e.g. the availability of indexes and some statistics.

3.6.2 Query Optimization

At query optimization step, the query planner generates different alternatives to the preliminary query plan and compares their costs using the cost model and the cost and metadata information, provided by the catalog or by the user. The cost information includes cardinalities and attribute selectivities of either a whole subquery or a particular data store table. To provide as much cost information as possible, each wrapper implementation should consider the cost-estimating capability of its data store and expose cost functions following one or more of the methods below:

- For a relational data store, if the data store can efficiently estimate the cost of a subquery and the size of its result set (like EXPLAIN on prepared statements), the query planner may benefit from this to directly ask a data store through its wrapper to estimate the cost of a subquery.
- If the data store is not capable of estimating the cost of a subquery, but keeps data store statistics (such as cardinalities, number of distinct values per column, etc.), the wrapper implementation should make use of all available data store statistics to provide implementations of the desired cost functions.
- If none of the above methods are applicable, but the data store supports aggregate queries like COUNT(*), MIN and MAX, the wrapper should contribute to the catalog information by periodically running in background probing queries, thus synthesizing and keeping statistics such as the number of tuples in a table, the number of distinct values of an attribute, and the min/max values of an attribute.
- However, because of the lack of cost models in some NoSQL data stores and the limited (or lack of) capability to build data store statistics, the CloudMdsQL query engine gives its data store administrator the possibility to define cost functions that give default cost information in case it cannot be retrieved using the above methods.
- Finally, the user may also provide user-defined cost functions, which is particularly useful in the case of native queries. For example, the native named table expression below defines a simple cardinality function, which gives information that the estimated cardinality of the returned table will be equal to the cardinality of the Outer relation, over which the native expression performs iteration.

```
distance(city_1 string, city_2 string, distance int
   JOINED ON city_1, city_2
   CARDINALITY = card(Outer) )@DB2 =
{* for (c1, c2) in CloudMdsQL.Outer:
```

```
    yield ( c1, c2, GetShortestDistance(c1, c2) ) *}
```

With this cost information, the query optimizer executes its search strategy to transform the preliminary execution plan into an optimized one. Notice that, when building its search space, the optimizer considers all sub-plans that are assigned to data stores just as atomic leaf nodes, meaning that the operations within the sub-plans are not subject to reordering. The search space explored for optimization is the set of all possible rewritings of the initial query, by pushing down select operations, expressing bind joins, join ordering, and intermediate data shipping. The result from the optimization step is an optimized QEP, where, besides the possibly modified order of common plan operations, additional information may be assigned to each operation as follows. Each binary operation (join or union) carries the identifier of the query engine node that is in charge of performing it, thus determining which intermediate relation will be shipped. Each equi-join operation carries also the join method to be performed – hash, nested-loop, merge, or bind join

Bind join [32] is an efficient method for performing semi-joins across heterogeneous data stores that uses subquery rewriting to push the join conditions. The approach to perform a bind join is the following: the left-hand side relation is retrieved, during which the tuples are stored in an intermediate storage and the distinct values of the join attribute(s) are kept in a list of values, which will be passed as a filter to the right-hand side subquery. For example, let us consider the following CloudMdsQL query:

```
A(id int, x int)@DB1 = (SELECT a.id, a.x FROM a)
   B(id int, y int)@DB2 = (SELECT b.id, b.y FROM b)
   SELECT a.x, b.y FROM b JOIN a ON b.id = a.id
```

Let us assume that the query planner has decided to use the bind join method and that the join condition will be bound to the right-hand side of the join operation. First, the relation `B` is retrieved from the corresponding data store using its query mechanism. Then, the distinct values of `B.id` are used as a filter condition in the query that retrieves the relation `A` from its data store. Assuming that the distinct values of `B.id` are $b_1 ... b_n$, the query to retrieve the right-hand side relation of the bind join uses the following SQL approach (or its equivalent according to the data store's query language):

```
SELECT a.id, a.x FROM a WHERE a.id IN (b_1, ..., b_n)
```

Thus, only the rows from A that match the join criteria are retrieved. In order to perform this operation, the final subquery to retrieve relation A must be composed by the query engine during runtime. Therefore, for each right-hand side of a bind join, the query compiler prepares an "almost ready" native query sentence, with placeholders for including the bind join condition, which will be added later by the query engine during runtime.

In order to estimate the expected performance gain of a bind join, the query optimizer takes into account the availability and type of indexes on the join attributes of the right-

hand side relation in the data store. Whenever such information is available from the data store, the wrapper should be able to provide it. Failing to do so will prevent the planner from planning bind join, as bind joins are beneficial only in case the join attributes are indexed.

Subquery rewriting can be planned by the optimizer in several occasions: (a) selection pushdowns, which result in pushing filter conditions from the common plan to sub-trees; (b) usage of bind joins which implies adding filter conditions to the subquery in order to allow the retrieval of only those tuples that match the join criteria; (c) taking advantage of sort-merge joins which requires adding sorting operations to subqueries in order to guarantee that the retrieved relations are sorted by their join attributes. The first rewriting approach is considered always efficient, i.e. whenever the data store is capable of handling it, the optimizer will plan selection pushdown. However, bind joins or merge joins will be planned either if explicitly specified by CloudMdsQL directives or as a result of optimization decision, of course taking into account data store's capabilities as well.

3.6.3 Query Execution

The QEP is passed to the first worker node for execution. The query execution controller is responsible for interpreting it and controlling its execution by passing native queries to the corresponding wrappers and instructing them to deliver the retrieved datasets to the operator engine and providing the operator engine the sequence of operators it must apply to the retrieved datasets.

The execution plan is received by the query execution controller in the form of a JSON document that contains sufficient information to configure and run efficiently each of the CloudMdsQL operations. The first step of the query execution controller is to identify the sub-plans within the plan that are associated to the collocated with the worker data store. Each sub-plan is sent to its corresponding data store wrapper to be translated into a native query. Then, the query execution controller identifies the parts of the common query plan associated with other worker nodes and sends them to their corresponding query execution controllers. For the rest of the query plan, the query execution controller looks for all named tables and temporary results involved in the execution plan, identifies the dependencies and configures their behavior. Finally, the query operator must be aware of parameterized operations that can return distinct results depending on the different input parameters.

Whenever possible, relations are just pipelined as a stream of volatile tuples from one operator to another, while in other cases the results are cached inside the table storage for later use. The table storage is used to store an intermediate relation, anytime the relation cannot be directly pipelined to its consuming operator, which happens in particular cases:

- If a named table expression is used more than once within the query and thus appears an operand to more than one operator;
- If the intermediate relation is an operand to a blocking operation, such as sorting or grouping;

- If the intermediate relation is the inner side of a nested-loop or hash join.

The table storage strives to use resources efficiently – it keeps an intermediate relation in-memory unless its size becomes so big that it must be spilled to disk. The query planner takes care not to plan for storing large tables, e.g. whenever an intermediate relation with big expected cardinality is involved in a hash or nested-loop join, the query planner will assign it to the outer side of the join, thus trying to keep large tables in the pipeline stream rather than storing them, in order to avoid table storage overflows.

The operator engine is then responsible for executing the operators in the order specified by the query execution controller. When a native call is required, the operator invokes the wrapper and opens a stream of external data that is ingested into the pipeline and, optionally, cached into the table storage. When the operator requires an existing named table, it is retrieved from the table storage and pipelined into the query execution flow. Data is never directly provided from operators to the wrapper: when necessary, the operator informs which named tables are required to solve the operation inside the data store. There is also a specific operator for CloudMdsQL that executes a Python program and pipelines the result in the form of tuples.

This iterator approach obtains tuples as soon as they are generated unless there exists a blocking operation. The resulting tuples are stored as a temporary named table into the table storage. This final table can be retrieved by the application with a forward sequential tuple iterator that supports rewinding and repositioning into marked rows. When the named table is no longer required it is automatically removed from the table storage.

3.6.4 Interfacing Data Stores

Wrappers are implemented as plugins to the query engine. In order for a data store to be accessed through the query engine, the wrapper developer must implement the corresponding wrapper following a common interface that is used by the query processor to interact with all wrappers. Whenever a CloudMdsQL query is processed, the query engine prepares a set of native subqueries (or subquery plans) that need to be executed against the data stores. The engine then passes each subquery to the corresponding wrapper, which is responsible for the following:

- The execution of native subqueries against the data store, for which there are two possibilities: (1) Server-side execution: The wrapper passes the query to the data store for remote execution (e.g. SQL); (2) Client-side execution: The wrapper executes the query locally, accessing the data store through a client library and API (e.g. Sparksee and its Python API);

- To guarantee that the retrieved data matches the number and types of columns, specified in the signature of the expected dataset in the CloudMdsQL query;

- To deliver the tuples of the retrieved datasets to the operator engine;

- To be able to instantiate other named table expressions, hence to access intermediate relations from the operator engine (table storage) during execution.

To add support for a new data store to the query engine, the data store administrator must implement a new wrapper. Whether the new data store will be subqueried through CloudMdsQL with SQL or native expressions depends on the data store's native query mechanism. If the new data store is an RDBMS or a mapping between the data store's query interface and SQL statements exist, the data store can be queried with SQL expressions and its wrapper should be implemented to handle subquery plans by translating them to native queries (see Sections 3.6.4.1 and 3.6.4.2). Otherwise, the data store must be queried with native expressions and the wrapper implementation should handle only native queries (see Section 3.6.4.3).

3.6.4.1 Querying SQL Compatible NoSQL Data Stores

Since the data model of some NoSQL data stores (e.g. key-value or document data stores) can be considered as a subset of the relational model, in most cases it is possible to map simple SQL commands to native queries, without compromising the functionality. In fact, SQL-like languages are already commonly used with data stores based on the BigTable data model, e.g. CQL for Cassandra. For such data stores, the recommended approach for subquerying within CloudMdsQL is to use SQL table expressions against the data store, even though the data store does not natively support SQL.

Whenever an SQL table expression is used as a nested query against a data store, it is considered as a sub-select statement and hence is transformed into a sub-tree in the QEP. Thus, each SQL table expression can be subject to further transformations and may be possibly rewritten by the optimizer before submitted for execution to the data store. This allows the CloudMdsQL engine to perform optimizations of the global QEP (like pushing selections, projections, aggregations, and joins down the tree into sub-plans) or take advantage of bind joins, etc. However, besides pushing down operations, the query optimizer does not perform further optimization (such as operation reordering) on a sub-plan, because it will only be used for building the corresponding native query, which normally is supposed to be optimized by the data store's optimizer. Each sub-plan is then delivered to the corresponding wrapper, which interprets and transforms it to a native query in order to execute it against the data store using its native query mechanism.

3.6.4.2 SQL Capabilities

In order to build executable subquery plans, the query planner must be aware of the capabilities of the corresponding data store to perform operations supported by the common data model. Therefore, the wrapper implementer must identify the subset of the common algebra that is supported by the data store. Thus the query planner can take the decision which parts of the global query plan can be handled locally by the data stores and which part should remain in the common query plan (see Section 3.6.1). For example, a MongoDB data store can perform selection operations – analogous to the document collection

method find() – but is not able to perform joins. Being aware of that, the query planner can push selection operations down to the subquery plan, but will assign any join operation between MongoDB document collections to the common query plan.

The method to handle data store capabilities, proposed in [56] requires that the query engine serializes the subquery plan (or single operations from it) to a sentence of a specific language that should be matched against a pattern, provided by the corresponding wrapper – if the validation succeeds, then the data store is capable of executing the subquery. Thus the query planner can determine the boundary between the common query plan the subplan that will be handled by the data store.

In CloudMdsQL, a similar approach is proposed which makes use of JSON schemas [5] as an instrument for the wrapper to express its data store's capabilities. To test the executability of a sub-plan (or a single operation) against a data store, the query planner serializes it to a JSON document that has to be validated against the JSON schema exposed by the wrapper. Below is an example of a capability JSON schema for a key-value data store that is capable only of performing selection operations involving comparisons on the 'key' attribute (only certain elements of the schema object are shown):

```
{ "properties": {
     "op": { "type": "string", "pattern": "SELECT" },
     "tableref": { "type": "string" },
     "filter": { "$ref": "#/definitions/expression" } },
    "definitions": {
     "expression": { "oneOf": [
      { "$ref": "#/definitions/comparison" },
      { "$ref": "#/definitions/function" } ] },
    "comparison": { "properties": {
     "comp": { "type": "string", "pattern": "=|<|>|<=|>=|<>" },
     "lhs": { "properties": {
      "colref": { "type": "string", "pattern": "key" }, },
     "rhs": { "type": "string" } } },
    "function":{ "properties": {
     "func": { "type": "string", "pattern": "AND|OR" },
     "lhs": { "$ref": "#/definitions/expression" },
     "rhs": { "$ref": "#/definitions/expression" } } }
} }
```

Now let us consider the following subquery that is composed of two conjunctive selection conditions, each of which is tested against the capability specification. The result shows that condition #1 can be handled by a selection operation in the key-value data store and therefore it will be left in the subquery, while condition #2 doesn't pass the validation, and therefore will be pulled up in the common plan to be processed by to common query engine.

```
SELECT key, value FROM tbl WHERE key BETWEEN 10 AND 20 AND value > key
```

Condition#1:key BETWEEN 10 AND 20 **Validation: success**	**Condition#2:value>key** **Validation: failure**

{"op": "SELECT",
"tableref": "tbl",
"filter": {
"func": "AND",
"lhs": { "comp": ">=",
"lhs": {"colref": "key"},
"rhs": "10" },
"rhs": { "comp": "<=",
"lhs": {"colref": "key"},
"rhs": "20" } } }
	{"op": "SELECT",
	"tableref": "tbl",
	"filter": {
	"comp": ">",
	"lhs": {"colref": "value"},
	"rhs": {"colref": "key"}
	}
	}

3.6.4.3 Using Native Queries

In a CloudMdsQL query, to write native named table expression subqueries against SQL incompatible data stores, embedded blocks of native query invocations are used. In such occasions, the wrapper is thin – it just takes the subquery as is and executes it against the data store without having to analyze the subquery or synthesize it from a query plan. Thus the wrapper provides transparency allowing CloudMdsQL queries to take the most of each data store's native query mechanism. When the data store does not have a text-based native query language but offers only an API, the wrapper is expected to expose such API through an embedded scripting language. This language must fulfill the following two requirements:

- Each query must produce a relation according to the common data model; the corresponding wrapper is then responsible to convert the data set to match the declared signature, if needed.
- In order to fulfill the requirement for nested tables support, the language should provide a mechanism to instantiate and use data retrieved by other named table expressions.

In this chapter we use Python as an example of embedded language used by a wrapper. The requirements above are satisfied by the yield keyword and CloudMdsQL object, similarly to what happens in Python named table expressions.

3.7 Use Case Example

To illustrate the details of CloudMdsQL query processing, we consider three data stores (briefly referred to as DB1, DB2 and DB3) as follows:

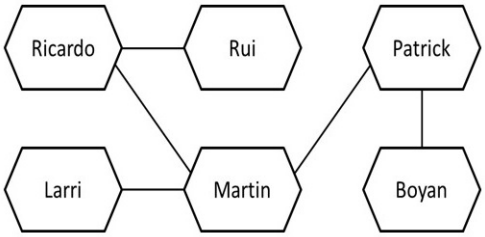
DB1 is a relational (e.g. Derby) data store storing information about scientists in the following table:
`Scientists:`

Name	Affiliation	Country
Ricardo	UPM	Spain
Martin	CWI	Netherlands
Patrick	INRIA	France
Boyan	INRIA	France
Larri	UPC	Spain
Rui	INESC	Portugal

DB2 is a document (e.g. MongoDB) data store containing the following collections of publications and reviews:

```
Publications(
{id:1, title:'Snapshot Isolation', author:'Ricardo', date:'2012-11-10'},
{id:5, title:'Principles of DDBS', author:'Patrick', date:'2011-02-18'},
{id:8, title:'Fuzzy DBs', author:'Boyan', date:'2012-06-29'},
{id:9, title:'Graph DBs', author:'Larri', date:'2013-01-06'} )
Reviews (
{pub_id:1, reviewer:'Martin', date:'2012-11-18', review:'...text...'},
{pub_id:5, reviewer:'Rui', date:'2013-02-28', review:'...text...'},
{pub_id:5, reviewer:'Ricardo', date:'2013-02-24', review:'...text...'},
{pub_id:8, reviewer:'Rui', date:'2012-12-02', review:'...text...'},
{pub_id:9, reviewer:'Patrick', date:'2013-01-19', review:'...text...'} )
```

DB3 is a graph data store (e.g. Sparksee) representing a social network with nodes representing persons and 'friend-of' links between them:
We now reveal step by step how the following CloudMdsQL query is processed by the engine. The query involves all the three data stores and aims to discover 'conflicts of interest in publications from Inria reviewed in 2013' (a conflict of interest about a publication is assumed to exist if the author and reviewer are friends or friends-of-friends in the social network). The subquery against DB3 uses the Sparksee Python API and user-defined functions and in particular, a function `FindShort estPathByName` defined over a graph object, which seeks the shortest path between two nodes by performing breadth-first search, referring the nodes by their 'name' attributes and limited to a maximal length



of the sought path.

```
scientists( name string, affiliation string )@DB1 = (
   SELECT name, affiliation
   FROM scientists )
pubs_revs( id int, title string, author string, reviewer string,
      review_date timestamp )@DB2 =
   ( SELECT p.id, p.title, p.author, r.reviewer, r.date
   FROM publications p, reviews r
   WHERE p.id = r.pub_id )
friendships( person1 string, person2 string, level int
     JOINED ON person1, person2
     WITHPARAMS maxlevel int
     CARDINALITY = card(Outer)/2 )@DB3 =
{* for (p1, p2) in CloudMdsQL.Outer:
     sp = graph.FindShortestPathByName( p1, p2, $maxlevel )
     if sp.exists():
      yield (p1, p2, sp.get_cost()) *}
friendship_levels( level int, friendship string
     WITHPARAMS maxlevel int
     CARDINALITY = maxlevel ) =
{* for i in range(0, $maxlevel):
      yield (i + 1, 'friend' + '-of-friend' * i) *}
SELECT pr.id, pr.title, pr.author, pr.reviewer, l.friendship
FROM scientists s, pubs_revs pr,
     friendships(2) f, friendship_levels(2) l
WHERE s.name = pr.author
    AND pr.author = f.person1 AND pr.reviewer = f.person2
    AND f.level = l.level
    AND pr.review_date BETWEEN '2013-01-01' AND '2013-12-31'
    AND s.affiliation = 'INRIA';
```

This query contains two SQL subqueries – one against a relational data store and the other against a document data store. The parameterized native named table expression

`friendships` against the graph data store defines a relation that represents the level of friendship between two persons (expressed by the length of the shortest path between them). The parameter `maxlevel` indicates a maximal value for the length of the sought path; the expression is invoked with actual value of the parameter `maxlevel=2`, meaning that only relationships of type direct-friend or friend-of-friend will be found. The parameterized Python named table expression `friendship_levels` generates synthetic data containing textual representations of friendship levels between `1` and `maxlevel`. Both the native and Python expressions provide cardinality functions that will be used by the query planner to compare different QEP. The main select statement specifies the join operations to integrate data retrieved from the three data stores. Upon query decomposition the query planner prepares the preliminary execution plan shown on Figure 3.3.

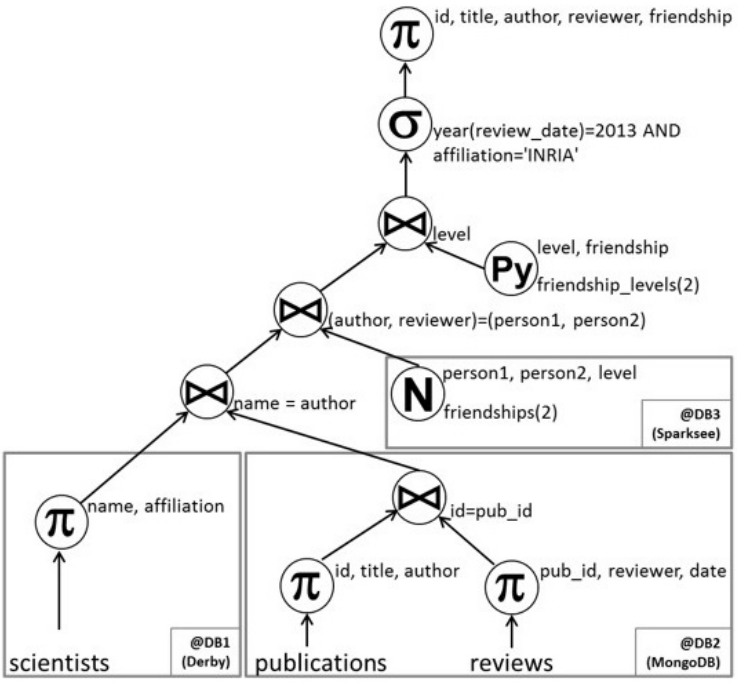


Figure 3.3 – **Preliminary execution plan**

In this notation the rectangles denote the boundary between the common QEP and the sub-plans that are delivered to the wrappers for execution against data stores. Each operator is denoted by a circle with the operator symbol inside. The operator symbols **N** and **Py** correspond to the native expression and Python operator respectively. In subscript to each operation, additional information is specified, such as the name of the expression for native/Python operations, and the filter/join condition for selection/ join operations. In superscript, the columns of the corresponding intermediate relation are specified.

In the next step, the query planner verifies the executability of sub-plans against the capability specifications provided by each wrapper. First, it finds out that the MongoDB

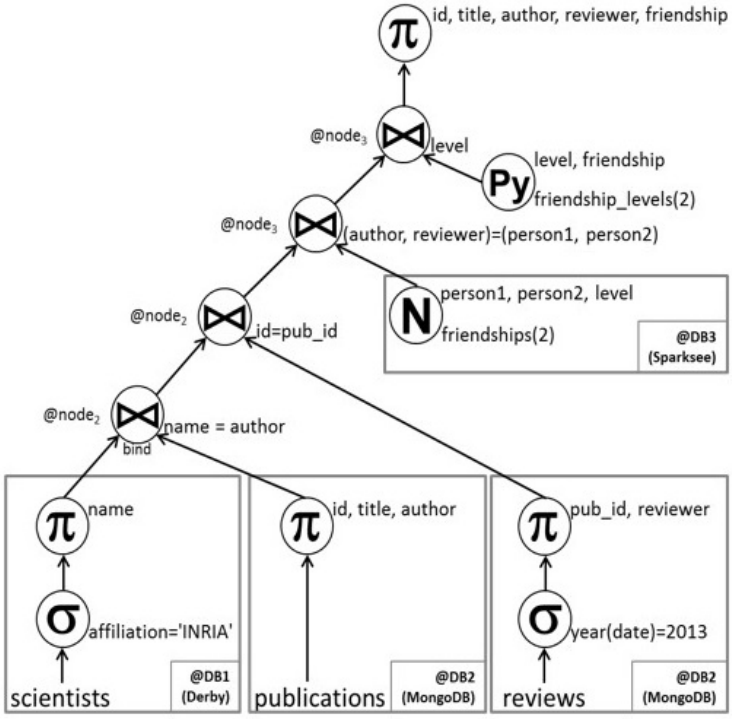


Figure 3.4 – **Optimized execution plan**

data store DB2 is not capable of performing the join between `publications` and `reviews`, therefore, it splits the sub-tree against DB2 into two sub-trees, aiming at independent retrieval of the two relations, and pulls the join operation in the common execution plan to be executed by the common query engine. Next, the optimizer seeks for opportunities for selection pushdowns, coordinating them as well with data store's capabilities. Thus, the selection `s.affiliation = 'INRIA'` is pushed into the sub-tree for DB1 and the selection `pr.review_date BETWEEN '2013-01-01' AND '2013-12-31'` is pushed into the sub-tree for DB2 that has to retrieve data from `reviews`. Doing this, the optimizer determines that the columns `s.affiliation` and `pr.review_date` are no longer referenced in the common execution plan, so they are simply removed from the corresponding projections on `scientists` and `reviews` from DB1 and DB2.

We assume that the Derby and MongoDB wrappers export the needed by the query optimizer metadata to the query engine's catalog. Taking also into account the cardinalities estimated by the user-defined cost functions of the native and Python expressions, the query planner searches for an optimal QEP, considering the usage of bind joins, join ordering, and the worker nodes in charge of each operation (which defines the way of shipping intermediate data). At the end of the optimization step, the preliminary plan is transformed into the plan on Figure 3.4 that is passed to the query execution controller of $node_3$.

Each join operation in the QEP is supplemented with the identifier of the node that is in charge of executing it. The enumeration of the nodes is according to the indexes of the collocated data stores as we named them, i.e. $node_1$ is collocated with DB1, etc. The join between `scientists` and `publications` is marked with the label `bind`, which means that a bind join method will be performed.

The QEP is executed by performing the following steps, including the sequence of queries executed against the three data stores:

1. The Derby wrapper at $node_1$ sends the following SQL statement to retrieve data from the `scientists` table in the Derby data store DB1, retrieves the corresponding intermediate relation, and transfers it to the operator engine of $node_2$:

   ```
   SELECT name FROM scientists WHERE affiliation = 'INRIA'
   ```

Name
Patrick
Boyan

   While retrieving the above tuples to the operator engine, the latter stores them in its temporary table storage and builds a set of distinct values of the column `name`, necessary for the next step.

2. The MongoDB wrapper at $node_2$ prepares a native query to send to the MongoDB data store DB2 to retrieve those tuples from `publications` that match the bind join criteria. It takes into account the bind join condition derived from the already retrieved data from DB1 and generates a MongoDB query whose SQL equivalent would be the following:

   ```
   SELECT id, title, author FROM publications
   WHERE author IN ('Patrick', 'Boyan')
   ```

   However, the wrapper for DB2 does not generate an SQL statement, instead it generates directly the corresponding MongoDB native query:

   ```
   db.publications.find(
    { author:  {$in:['Patrick', 'Boyan']} },
    { id:  1, title:  1, author:  1, _id:  0 } )
   ```

   Upon receiving the result dataset (a MongoDB document collection), the wrapper converts each document to a tuple, according to the signature of the named table expression pubs_revs, and then pipelines the tuples to the operator engine, which performs the actual join operation using the already retrieved result set from step 1.

The result of the bind join is the contents of the following intermediate relation:

id	Title	Author
5	Principles of DDBS	Patrick
8	Fuzzy DBs	Boyan

Since this relation will be consumed by only one operator, the operator engine does not need to store it in the table storage; therefore the tuples are simply pipelined as input to the operation described in step 4.

3. Independently from steps 1 and 2, the wrapper prepares another MongoDB query for DB2 that, taking into account the pushed down selection, retrieves reviews made in 2013. The generated native query (preceded by its SQL equivalent) and the result intermediate relation are as follows:

```
SELECT pub_id, reviewer FROM reviews
 WHERE date BETWEEN '2013-01-01' AND '2013-12-31'

db.reviews.find(
  { date:  {$gte:'2013-01-01', $lte:'2013-12-31'} },
  { pub_id:  1, reviewer:  1, _id:  0 } )
```

Pub_id	Reviewer
5	Rui
5	Ricardo
9	Patrick

4. The intermediate relations from steps 2 and 3 are joined by the operator engine at $node_2$ to result in another intermediate relation, which is transferred to the operator engine of $node_3$ to be pipelined to the next join operator:

id	Title	Author	Reviewer
5	Principles of DDBS	Patrick	Rui
5	Principles of DDBS	Patrick	Ricardo

5. The query engine sends to the wrapper of DB3 the Python code to be executed against the graph data store. It also provides an entry point to the intermediate data, represented by the special Python object `CloudMdsQL`. The wrapper of DB3 has preliminarily initialized the object `graph`, needed to reference the data store's

graph data. The Python code of the named table expression `friendships` iterates through a projection on the join attribute columns of tuples pipelined from the intermediate relation of step 4. For each tuple it tests if there exists a path with maximal length `maxlevel=2` between the author and reviewer in the graph data store. The produced tuples are as follows:

Person1	**Person**2	**Level**
Patrick	Ricardo	2

As the above tuples are generated by the Python expression `friendships`, they are immediately joined with their corresponding tuples of the relation from step 4 to produce the next intermediate relation:

id	**Title**	**Author**	**Reviewer**	**Level**
5	Principles of DDBS	Patrick	Ricardo	2

6. Independently from all of the above steps, the operator engine at $node_3$ executes the Python code of the expression `friendship_levels,` instantiated with parameter value `maxlevel=2` to produce the relation:

Level	**Friendship**
1	friend
2	friend-of-friend

Essentially, the involvement of this Python operator is not needed for the purpose of the query, because the textual representation of a level of friendship can be generated directly within the code of the native expression `friend ships`. However, we include it in the example in order to demonstrate a wider range of CloudMdsQL operators.

7. Finally, the root join operation is performed, taking as input the pipelined tuples of the intermediate relation from step 5 and matching them to the one from step 6, to produce the final result:

id	**Title**	**Author**	**Reviewer**	**Friendship**
5	Principles of DDBS	Patrick	Ricardo	friend-of-friend

This use case example demonstrates that the proposed query engine achieves its objectives by fulfilling the five requirements as follows:

(a) It preserves the expressivity of the local query languages by embedding native queries, as it was demonstrated with the named table expression `friendships`.

(b) It allows nested queries to be chained and nesting is allowed in both SQL and native expressions, as it was demonstrated in two scenarios. First, the subquery against the MongoDB data store DB2 uses as input the result from the subquery to the relational data store DB1. Second, the subquery against the Sparksee graph data store DB3 iterates through data retrieved from the other two data stores.

(c) The proper functioning of the query engine does not depend on the data stores' schemas; it simply converts the data retrieved from data stores to match the ad-hoc schema defined by the named table expressions' signatures.

(d) It allows data-metadata transformations as it was demonstrated with the named table expression `friendships`: metadata (the length of a path in the graph data store) is converted to data (the level of friendship). It also allows data to be synthesized as with the Python table expression `friendship_levels`.

(e) It allows for optimizing the query execution by rewriting subqueries according to the bind join condition and the pushed down selections and planning for optimal join execution order and intermediate data transfer.

3.8 Conclusion

In this chapter, we described CloudMdsQL, a common language for querying and integrating data from heterogeneous cloud data stores and its query engine. By combining the expressivity of functional languages and the manipulability of declarative relational languages, it stands in "the golden mean" between the two major categories of query languages with respect to the problem of unifying a diverse set of data management systems. CloudMdsQL satisfies all the legacy requirements for a common query language, namely: support of nested queries across data stores, data-metadata transformations, schema independence, and optimizability. In addition, it allows embedded invocations to each data store's native query interface, in order to exploit the full power of data stores' query mechanism.

The architecture of the CloudMdsQL query engine is fully distributed, so that query engine nodes can directly communicate with each other, by exchanging code (query plans) and data. Thus, the query engine does not follow the traditional mediator-wrapper architectural model where mediator and wrappers are centralized. This distributed architecture yields important optimization opportunities, e.g. minimizing data transfers by moving the smallest intermediate data for subsequent processing by one particular node. The wrappers are designed to be transparent, making the heterogeneity explicit in the query in favor of preserving the expressivity of local data stores' query languages. CloudMdsQL sticks to the relational data model, because of its intuitive data representation, wide acceptance and ability to integrate datasets by applying joins, unions and other relational algebra operations.

Chapter 4

Extending CloudMdsQL with MFR

With the advent of multistore systems, integration of unstructured big data typically stored using HDFS with relational data becomes a requirement. One main solution is to use a relational query engine that allows SQL-like queries to retrieve data from HDFS, which requires the system to provide a relational view of the unstructured data and hence is not always feasible. This chapter is based on [13, 14].

In this chapter, we extend CloudMdsQL to take full advantage of the functionality of the underlying data processing frameworks by allowing the ad-hoc usage of user-defined map/filter/reduce (MFR) operators in combination with traditional SQL statements. Furthermore, our solution allows for optimization by enabling subquery rewriting so that bind join can be used and filter conditions can be pushed down and applied by the data processing framework as early as possible.

This chapter is organized as follows. The decision made to use our extending language is done in Section 4.1. Section 4.2 introduces the language and its notation to express map/filter/reduce subqueries. Section 4.3 presents the architecture of the query engine. Section 4.4 elaborates more on the query processing and presents the properties of MFR operators that constitute rewrite rules to perform query optimization. Section 4.5 gives a use case example walkthrough. Section 4.6 concludes.

4.1 Overview

Multistore systems provide integrated access to multiple, heterogeneous data stores - through a single query engine. Much attention is being paid on the integration of unstructured big data (e.g. produced by web applications) typically stored in HDFS with relational data, e.g. in a data warehouse. One main solution is to use a relational query engine (e.g. Apache Hive) on top of a data processing framework (e.g. Hadoop MapReduce), which allows SQL-like queries to retrieve data from HDFS. However, this requires the system to provide a relational view of the unstructured data, which is not always feasible. In case the data store is managed independently from the relational query processing system, complex data transformations may need to take place (e.g. by applying specific map-reduce jobs) before the data can be processed by means of relational operators. Let

us illustrate the problem, which will be the focus of this chapter, with the following scenario.

Example scenario. An editorial office needs to find appropriate reporters for a list of publications based on given keywords. For the purpose, the editors need an analysis of the logs from a scientific forum stored in a Hadoop cluster in the cloud to find experts in a certain research field, considering the users who have mentioned particular keywords most frequently; and these results must be joined to the relational data in an RDBMS containing author and publication information. However, the forum application keeps log data about its posts in a non-tabular structure (the left side of the example below), namely in text files where a single record corresponds to one post and contains a fixed number of fields about the post itself (timestamp and username in the example) followed by a variable number of fields storing the keywords mentioned in the post.

		KW	expert	freq
2014-12-13, alice, storage, cloud		cloud	alice	2
2014-12-22, bob, cloud, virtual, app	→	storage	alice	1
2014-12-24, alice, cloud		virtual	bob	1
		app	bob	1

The unstructured log data needs to be transformed into a tabular dataset containing for each keyword the expert who mentioned it most frequently (the right side of the example above). Such transformation requires the use of programming techniques like chaining map/reduce operations that should take place before the data is involved in relational operators. Then the result dataset will be ready to be joined with the publication data retrieved from the RDBMS in order to suggest an appropriate reviewer for each publication. Being able to request such data processing with a single query is the scenario that motivates our work. However, the challenge in front of the query processor is optimization, i.e. it should be able of analyzing the operator execution flow of a query and performing operation reordering to take advantage of well-known optimization techniques (e.g. selection pushdowns and use of semi-joins) in order to yield efficient query execution.

Existing solutions to integrate such unstructured and structured data do not directly apply to solve our problem, as they rely on having a relational view of the unstructured data, and hence require complex transformations. SQL engines, such as Hive, on top of distributed data processing frameworks are not always capable of querying unstructured HDFS data, thereby forcing the user to query the data by defining map/reduce functions.

Our approach is different as we propose a query language that can directly express subqueries that can take full advantage of the functionality of the underlying data processing frameworks. Furthermore, the language should allow for query optimization, so that the query operator execution sequence specified by the user may be reordered by taking into account the properties of map/filter/reduce operators together with the properties of relational operators. This is especially useful for applying efficient query optimization by exploiting bind joins [32]; and we pay special attention to this throughout our experi-

mental evaluation. Finally, we want to respect the autonomy of the data stores, e.g. HDFS and RDBMS, so that they can be accessible and controlled from outside our query engine with their own interface.

We extend CloudMdsQL to retrieve data from three different kinds of data stores - an RDBMS and a distributed data processing framework such as Apache Spark or Hadoop MapReduce on top of HDFS - and combine them by applying data integration operators (mostly joins). We assume that each data store is fully autonomous, i.e. the query engine has no control over the structure and organization of data in the data stores. For this reason, the architecture of our query engine is based on the traditional mediator-architectural approach [60] that abstracts the query engine from the specifics of each of the underlying data stores. However, users need to be aware of how data are organized across the data stores, so that they write valid queries. A single query of our language can request data to be retrieved from both stores and then a join to be performed over the retrieved datasets. The query therefore contains embedded invocations to the underlying data stores, expressed as subqueries. As our query language is functional, it introduces a tight coupling between data and functions. A subquery, addressing the data processing framework, is represented by a sequence of map/filter/reduce operations, expressed in a formal notation. On the other hand, SQL is used to express subqueries that address the relational data store as well as the main statement that performs the integration of data retrieved by all subqueries. Thus, a query benefits from both high expressivity (by allowing the ad-hoc usage of user-defined map/filter/reduce operators in combination with traditional SQL statements) and optimizability (by enabling subquery rewriting so that bind join and filter conditions can be pushed inside and executed at the data store as early as possible).

4.2 Query Language

In this section, we introduce a formal notation to define Map/Filter/Reduce (MFR) subqueries in CloudMdsQL that request data processing in an underlying big data processing framework (DPF). Then we give an overview of how MFR statements are combined with SQL statements to express integration queries against a relational data store and a DPF. Notice that the data processing defined in an MFR statement is not executed by the query engine, but is meant to be translated to a sequence of invocations to API functions of the DPF. We use Apache Spark as an example of DPF, but the concept can be generalized to a wider range of frameworks that support the MapReduce programming model (such as Hadoop MapReduce, CouchDB, etc.).

4.2.1 MFR Notation

An MFR statement represents a sequence of MFR operations on datasets. A dataset is considered simply as an abstraction for a set of tuples, where a tuple is a list of values, each of which can be a scalar value or another tuple. Although tuples can generally have any number of elements, mostly datasets that consist of key-value tuples are being

processed by MFR operations. In terms of Apache Spark, a dataset corresponds to an RDD (Resilient Distributed Dataset - the basic programming unit of Spark). Each of the three major MFR operations (MAP, FILTER and REDUCE) takes as input a dataset and produces another dataset by performing the corresponding transformation. Therefore, for each operation there should be specified the transformation that needs to be applied on tuples from the input dataset to produce the output tuples. Normally, a transformation is expressed with an SQL-like expression that involves special variables; however, more specific transformations may be defined through the use of lambda functions.

Core operators. The MAP operator produces key-value tuples by performing a specified transformation on the input tuples. The transformation is defined as an SQL-like expression that will be evaluated for each tuple of the input data set and should return a pair of values. The special variable `TUPLE` refers to the input tuple and its elements are addressed using a bracket notation. Moreover, the variables `KEY` and `VALUE` may be used as aliases to `TUPLE[0]` and `TUPLE[1]` respectively. The FILTER operator selects from the input tuples only those, for which a specified condition is evaluated to *true*. The filter condition is defined as a boolean expression using the same special variables `TUPLE`, `KEY` and `VALUE`. The REDUCE operator performs aggregation on values associated with the same key and produces a key-value dataset where each key is unique. The reduce transformation may be specified as an aggregate function (`SUM, AVG, MIN, MAX or COUNT`). Similarly to MAP, two other mapping operators are introduced: FLAT _MAP may produce numerous output tuples for a single input tuple; and MAP_VALUES defines a transformation that preserves the keys, i.e. applicable only to the values.

Let us consider the following simple example inspired by the popular MapReduce tutorial application "word count". We assume that the input dataset for the MFR statement is a list of words. To count the words that contain the string "cloud", we write the following composition of MFR operations:

```
MAP(KEY, 1).FILTER( KEY LIKE '%cloud%' ).REDUCE( SUM )
```

The first operation transforms each tuple (which has a single word as its only element) of the input dataset into a key-value pair where the word is mapped to a value of 1. The second operation selects only those key-value pairs for which the key contains the string "cloud". And the third one groups all tuples by key and performs a sum aggregate on the values for each key.

To process this statement, the query engine first looks for opportunities to optimize the execution by operator reordering. By applying MFR rewrite rules (explained in detail in Section 4.4.2), it finds out that the FILTER and MAP operations may be swapped so that the filtering is applied at an earlier stage. Further, it translates the sequence of operations into invocations of the underlying DPF's API. Notice that whenever a REDUCE transformation function has the associative property (like the SUM function), an additional combiner function call may be generated that precedes the actual reducer, so that as much data as possible will be reduced locally; e.g., this would be valid in the case of Hadoop MapReduce as the DPF, because it does not automatically perform local reduce.

In the case of Apache Spark as the DPF, the query engine generates the following Python fragment to be included in a script that will be executed in Spark's Python environment:

```
dataset.filter( lambda k:  'cloud' in k ) \
     .map( lambda k:  (k, 1) ) \
     .reduceByKey( lambda a, b:  a + b )
```

In this example, all the MFR operations are translated to their corresponding Spark functions and all transformation expressions are translated to Python anonymous functions. In fact, to increase its expressivity, the MFR notation allows direct usage of anonymous functions to specify transformation expressions. This allows user-defined mapping functions, filter predicates, or aggregates to be used in an MFR statement. The user, however, needs to be aware of how the query engine is configured to interface the DPF, in order to know which language to use for the definition of inline anonymous functions (e.g. Spark may be used with Python or Scala, CouchDB - with JavaScript, etc.).

Input/output operators are normally used for transformation of data before and after the core map/filter/reduce execution chain. The SCAN operator loads data from its storage and transforms it to a dataset ready to be consumed by a core MFR operator. The PROJECT operator converts a key-value dataset to a tabular dataset ready to be involved in relational operations.

4.2.2 Combining SQL and MFR

Queries that integrate data from both a relational data store and a DPF usually consist of two subqueries (one expressed in SQL that addresses the relational data store and another expressed in MFR that addresses the DPF) and an integration SELECT statement. The syntax follows the CloudMdsQL grammar introduced in 3. A subquery is defined as a named table expression, i.e. an expression that returns a table and has a name and signature. The signature defines the names and types of the columns of the returned relation. Thus, each query, although agnostic to the underlying data stores' schemas, is executed in the context of an ad-hoc schema, formed by all named table expressions within the query. A named table expression can be defined by means of either an SQL SELECT statement (that the query compiler is able to analyze and possibly rewrite) or a native expression (that the query compiler considers as a black box and passes to the wrapper as is, thus delegating it the processing of the subquery).

We extend the usability of CloudMdsQL by adding the capability of handling MFR subqueries against DPFs and combining them with subqueries against other data stores. This is done in full compliance with CloudMdsQL properties, such as the ability to express nested subqueries (so that the output of one subquery, e.g. against an RDBMS, can be used as input to another subquery, e.g. MFR) which we further illustrate by the usage of bind joins. MFR subqueries are expressed as native named table expressions; this means that they are passed to their corresponding wrappers to process them (explained in more detail in Section 4.3).

In general, a single query can address a number of data stores by containing several named table expressions. We will now illustrate with a simple example how SQL and MFR statements can be combined, and in Section 4.5 will focus on a more sophisticated example involving 3 data stores. The following sample query contains two subqueries, defined by the named table expressions T1 and T2, and addressed respectively against the data stores aliased with identifiers `rdb` (for the SQL data store) and `hdfs` (for the DPF):

```
T1 (title string, kw string)@rdb = ( SELECT title, kw FROM tbl )
T2 (word string, count int)@hdfs = {*
     SCAN(TEXT,'words.txt')
      .MAP(KEY,1).REDUCE(SUM).PROJECT(KEY,VALUE) *}
SELECT title, kw, count FROM T1 JOIN T2 ON T1.kw = T2.word
WHERE T1.kw LIKE '%cloud%'
```

The purpose of this query is to perform relational algebra operations (expressed in the main SELECT statement) on two datasets retrieved from a relational data store and a DPF. The two subqueries are sent independently for execution against their data stores in order the retrieved relations to be joined by the query engine. The SQL table expression `T1` is defined by an SQL subquery. `T2` is an MFR expression that requests data retrieval from a text source and data processing by the specified map/reduce operations. Both subqueries are subject to rewriting by pushing into it the filter condition `kw LIKE '%cloud%'`, specified in the main SELECT statement, thus reducing the amount of the retrieved data by increasing the subquery selectivity and the overall efficiency. The so retrieved datasets are then converted to relations following their corresponding signatures, so that the main SELECT statement can be processed with semantic correctness. The PROJECT operator in the MFR statement provides a mapping between the dataset fields and the named table expression columns.

4.3 Query Engine Architecture

In this section, we briefly describe the architecture of our system with an overview of the required steps to process a query. The query language presented hereby assumes a query engine that follows the traditional mediator-wrapper architectural approach. By explicitly naming a data store identifier in a named table expression's signature, the query addresses the specific wrapper that is preliminarily configured and responsible for handling subqueries against the corresponding data store. Thus, a query can express an integration of data across several data stores, and in particular, integration of structured (relational DB), semi-structured (document DB), and unstructured (distributed storage, based on HDFS) data.

Figure 4.1 depicts the corresponding system architecture, containing a CloudMdsQL compiler, a common query processor (the mediator), three wrappers, and the three data stores a distributed data processing framework (DPF), an RDBMS, and a document data

store. The DPF is in charge of performing parallel data processing over a distributed data store. In this architecture, each data store has an associated wrapper that is responsible for executing subqueries against the data store and converting the retrieved datasets to tables matching the requested number and types of columns, so that they are ready to be consumed by relational operators at the query processor. The query processor consumes the query execution plan generated by the compiler and interacts with the wrappers through a common interface to: request handling of subqueries, centralize the information provided by the wrappers, and integrate the subqueries' results. The wrappers transform subqueries provided via the common interface into queries for the data stores. This generic architecture gives us the possibility to use a specific implementation of the query processor and DPF wrapper, while reusing the CloudMdsQL query compiler and wrappers for relational and document data stores. Although we can also reuse the CloudMdsQL query engine that has a distributed architecture.

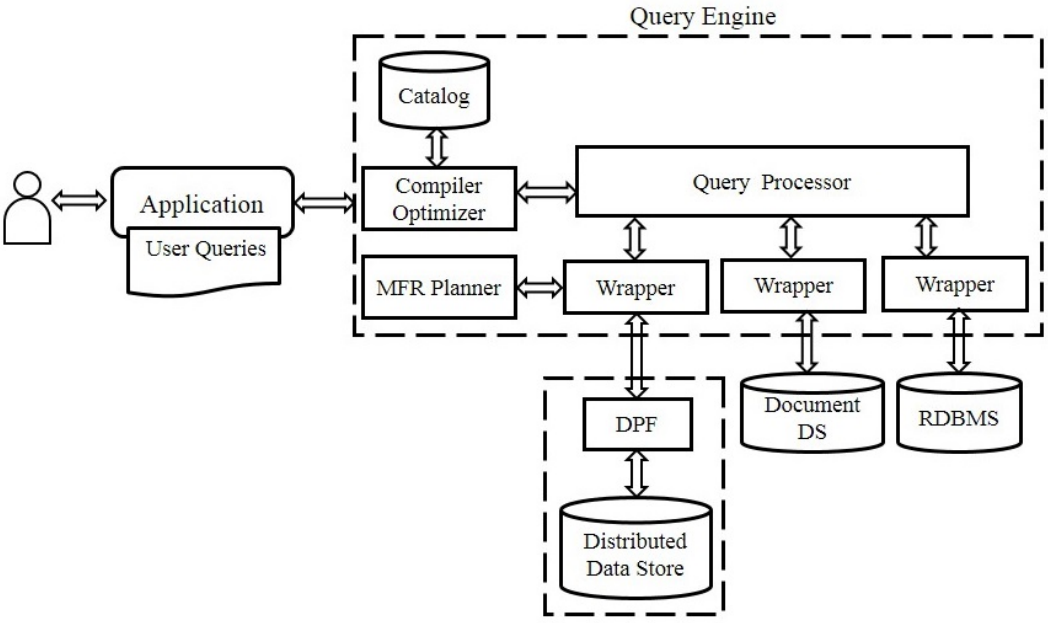


Figure 4.1 – Basic architecture of the query engine with MFR

Each of the wrappers is responsible for completing the execution of subqueries and retrieving the results. Upon initialization, each wrapper may provide to the query compiler the capability of its data store to process pushed down operations [40]. In our setup, all the three wrappers can accept pushdowns of filter predicates. Both the relational and document data store wrappers accept requests from the query processor in the form of query execution sub-plans represented as trees of relational algebra operators, resulting from the compilation of the SELECT statements expressed in the corresponding SQL named table expressions. The sub-plans may include selection operations resulting from pushed down predicates. The wrapper of the relational data store has to build a SELECT statement out of a query sub-plan and to run it against its data store; then it retrieves the

datasets and delivers them to the query processor in the corresponding format. The wrapper of the document data store (in our case, MongoDB) has to translate the sequence of relational operators from a query sub-plan to the corresponding sequence of MongoDB API calls; then it converts the resulting documents to tuples that match the signature of the corresponding named table expression [40].

The wrapper of the distributed data processing framework has a slightly different behavior as it processes MFR expressions wrapped in native subqueries. First it parses and interprets a subquery written in MFR notation; then uses the MFR planner to find optimization opportunities; and finally translates the resulting sequence of MFR operations to a sequence of DPF's API methods to be executed. Once a dataset is retrieved as a result of the subquery execution, the wrapper provides it to the query processor in the format requested by the corresponding named table expression signature. The MFR planner decides where to position pushed down operations; e.g. it applies rules for MFR operator reordering to find the optimal place of a filter operation in order to apply it as early as possible and thus to reduce the query execution cost. To search for alternative operation orderings, the planner takes into account MFR rewrite rules, introduced in next section.

4.4 Query Processing

The query compiler first decomposes the query into a preliminary query execution plan (QEP), which, in its simplest form, is a tree structure representing relational operations. At this step, the compiler also identifies sub-trees within the query plan, each of which is associated to a certain data store. Each of these sub-plans is meant to be delivered to the corresponding wrapper, which has to translate it to a native query and execute it against its data. The rest of the QEP is the common plan that will be handled by the query engine.

4.4.1 Query Optimization

Before its actual execution, a QEP may be rewritten by the query optimizer. To compare alternative rewritings of a query, the optimizer uses a simple catalog, which provides basic information about data store collections such as cardinalities, attribute selectivities and indexes, and a simple cost model. Because of the autonomy of the underlying data stores, in order to derive local cost models, various classical black-box approaches for heterogeneous cost modeling, such as probing [66] and sampling [65, 67], have been adopted by the query optimizer. Thus, cost information can be collected by the wrappers and exposed to the optimizer in the form of cost functions or data store statistics. Furthermore, the query language allows for user-defined cost and selectivity functions. And in case of lack of any cost information, heuristic rules are applied.

In our concrete example scenario with PostgreSQL, MongoDB, and MFR subqueries, we use the following strategy. The query optimizer executes an EXPLAIN request to PostgreSQL to directly estimate the cost of a subquery. The MongoDB wrapper runs in background probing queries to collect cardinalities of document collections, index avail-

abilities, and index value distributions (to compute selectivities) and caches them in the query engine's catalog. As for an MFR subquery, if there is no user-provided cost information, the optimizer assumes that it is more expensive than SQL subqueries and plans it at the end of the join order, which would also potentially benefit from the execution of bind joins.

The search space explored for optimization is the set of all possible rewritings of the initial query, by pushing down select operations, expressing bind joins, and join ordering. Unlike in traditional query optimization where many different permutations are possible, this search space is not very large, so we use a simple exhaustive search strategy.

Subquery rewriting takes place in order to request early execution of some operators and thus to increase its overall efficiency. Although several operations are subject to pushdowns across subqueries, we concentrate on the inclusion of only filter operations inside an MFR subquery. Generally, this is done in two stages: first, the query processor determines which operations can be pushed down for remote execution at the data stores; and second, the MFR planner may further determine the optimal place for inclusion of pushed down operations within the MFR operator chain by applying MFR rewrite rules (explained later in this section). Pushing a selection operation inside a subquery, either in SQL query or MFR operation chain, is usually considered beneficial, because it delegates the selection directly to the data store, which allows for early reducing of the size of data processed and retrieved from the data stores.

4.4.2 MFR Rewrite Rules

In this section, we introduce and enumerate some rules for reordering of MFR operators, based on their algebraic properties. These rules are used by the MFR planner to optimize an MFR subquery after a selection pushdown takes place.

Rule #1 (name substitution): upon pushdown, the filter is included just before the PRO JECT operator and the filter predicate expression is rewritten by substituting column names with references to dataset fields as per the mapping defined by the `PROJECT` expressions. After this initial inclusion, other rules apply to determine whether it can be moved even farther. Example:

```
T1(a int, b int)@db1 = {* ...  .PROJECT(KEY, VALUE[0]) *}
SELECT a, b FROM T1 WHERE a > b
```

is rewritten to:

```
T1(a int, b int)@db1 ={* ...  .FILTER(KEY>VALUE[0])
.PROJECT(KEY,VALUE[0])*}
SELECT a, b FROM T1
```

Rule #2: `REDUCE(<transformation>).FILTER(<predicate>)` is equivalent to `FILTER(<predicate>).REDUCE(<transformation>)`, if `predicate` condition is a function only of the `KEY`, because thus, applying the `FILTER` before the

REDUCE will preserve the values associated to those keys that satisfy the filter condition as they would be if the FILTER was applied after the REDUCE. Analogously, under the same conditions, MAP_VALUES(<transformation>) .FILTER(<predicate>) is equivalent to FILTER(<predicate>).MAP_VALUES(<transformation>).

Rule #3: MAP(<expr_list>).FILTER(<predicate1>) is equivalent to FILTER(<predicate2 >).MAP(<expr_list>), where predicate1 is rewritten to predicate2 by substituting KEY and VALUE as per the mapping defined in expr_list. Example:

```
MAP(VALUE[0], KEY).FILTER(KEY > VALUE) →
    FILTER(VALUE[0] > KEY).MAP(VALUE[0], KEY)
```

Since planning a filter as early as possible always increases the efficiency, the planner always takes advantage of moving a filter by applying rules #2 and #3 whenever they are applicable.

4.4.3 Bind Join

Bind join [32] is an efficient method for implementing semi-joins across heterogeneous data stores that uses subquery rewriting to push the join conditions. We adapt the bind join approach for MFR subqueries and we focus on it in our experimental evaluation, as it brings a significant performance gain in certain occasions.

Using bind join between relational data (expressed in an SQL named table expression) and big data (expressed in an MFR named table expression) allows for reducing the computation cost at the DPF and the communication cost between the DPF and the query engine. This approach implicates that the list of distinct values of the join attribute(s) from the relation, preliminarily retrieved from the relational data store, and is passed as a filter to the MFR subquery. To illustrate the approach, let us consider the following SELECT statement performing a join between an SQL named table R and an MFR named table H:

```
SELECT H.x, R.y FROM R JOIN H ON R.id = H.id WHERE R.z='abc'
```

To process this query using the bind join method, first, the table R is retrieved from the relational data store; then, assuming that the distinct values of R.id are $r_1 ... r_n$, the condition id IN $(r_1, ..., r_n)$ is passed as a FILTER to the MFR subquery that retrieves the dataset H from HDFS data store. Thus, only the tuples from H that match the join criteria are retrieved. Moreover, if the filter condition can be pushed even further in the MFR chain (according to the MFR rewrite rules) and thus to overcome at least one REDUCE operation, this may lead to a significant performance boost, as data will be filtered before at least one shuffle phase.

To estimate the expected performance gain of a bind join, the query optimizer takes into account the overhead a bind join may produce. First, when using bind join, the

query engine must wait for the SQL named table to be fully retrieved before initiating the execution of the MFR subquery. Second, if the number of distinct values of the join attribute is large, using a bind join may slower the performance as it requires data to be pushed into the MFR subquery. In the example above, the query engine first asks the RDBMS (e.g. by running an `EXPLAIN` statement) for an estimation of the cardinality of data retrieved from `R`, after rewriting the SQL subquery by including the selection condition `R.z='abc'`. If the estimated cardinality does not exceed a certain threshold, the optimizer plans for performing a bind join that can significantly increase the MFR subquery selectivity and affect the volume of transferred data.

4.5 Use Case Example

In this section, we reveal the steps the query engine takes to process a query using selection pushdown and especially bind join as optimization techniques. We also focus on the way the query engine dynamically rewrites the MFR subquery to perform a bind join. We consider three distinct data stores: PostgreSQL as the relational data store (referred to as `rdb`), MongoDB as the document data store (referred to as mongo) which will be subqueried by SQL expressions that are mapped by the wrapper to MongoDB calls, and an HDFS cluster (referred to as `hdfs`) processed using the Apache Spark framework.

Datasets. For the use case walkthrough we consider small sample datasets in the context of the multistore query example described in Section 4.1.
The `rdb` data store stores structured data about scientists and their affiliations in the following table:

`Scientists:`

Name	Affiliation	Country
Ricardo	UPM	Spain
Martin	CWI	Netherlands
Patrick	INRIA	France
Boyan	INRIA	France
Larri	UPC	Spain
Rui	INESC	Portugal

The `mongo` data store contains a document collection about publications including their keywords as follows:

```
Publications(
{ title:'Snapshot Isolation in Cloud DBs', author:'Ricardo',
     keywords:  ['transaction','cloud'] },
{ title:'Principles of Distributed Cloud DBs', author:'Patrick',
```

```
     keywords:  ['cloud', 'storage'] },
{ title:'Graph Data stores', author:'Larri', keywords:['graph', 'NoSQL']})
```

HDFS stores unstructured log data from a scientific forum in text files where a single record corresponds to one post and contains a timestamp and username followed by a variable number of fields storing the keywords mentioned in the post:

Posts (date, author, kw_1, kw_2,...,kw_n)

2014-11-10, alice, storage, cloud
2014-11-10, bob, cloud, virtual, app
2014-11-10, alice, cloud

Query 1. This query aims at finding appropriate reviewers for publications of authors with a certain affiliation. It considers each publication keywords and the experts who have mentioned them most frequently on the scientific forum. The query combines data from the three data stores and can be expressed as follows.

```
scientists( name string, affiliation string )@rdb = (
   SELECT name, affiliation
   FROM scientists )

publications(autor string, title string, keywords array)@mongo = (
    SELECT author, title, keywords
   FROM publications )

experts(kw string, expert string)@hdfs = {*
   SCAN(TEXT, 'posts.txt', ',')                                  (op1)
    .FLAT_MAP( lambda data:  product (data[2:], [data[1]]) )     (op2)
    .MAP( TUPLE, 1 )                                             (op3)
    .REDUCE( SUM )                                               (op4)
    .MAP( KEY[0], (KEY[1], VALUE) )                              (op5)
    .REDUCE( lambda a, b:  b if b[1] > a[1] else a )             (op6)
   .PROJECT(KEY, VALUE[0])                                       (op7) *}

SELECT p.author, p.title, e.kw, e.expert
FROM scientists s, publications p, experts e
WHERE s.affiliation = 'INRIA'
   AND p.author = s.name
   AND e.kw IN p.keywords
```

Query 1 contains three subqueries. The first two subqueries is a typical SQL statement to get data about respectively scientists (from PostgreSQL) and scientific publications (from

MongoDB). The third subquery is an MFR operation chain that transforms the unstructured log data from the forum posts and represents the result of text analytics as a relation that maps each keyword to the person who has most frequently mentioned it. To achieve the result dataset, the MFR operations request transformations over the stored data, each of which is expressed either in a declarative way or with anonymous (lambda) Python functions.

The `SCAN` operation op1 reads data from the specified text source and splits each line to an array of values. Let us recall that the produced array contains the author of the post in its second element and the mentioned keywords in the subarray starting from the third element. The following `FLAT_MAP` operation op2 consumes each emitted array as a tuple and transforms each tuple using the defined Python lambda function, which performs a Cartesian product between the keywords subarray and the author, thus emitting a number of keyword-author pairs. Each of these pairs is passed to the `MAP` operation op3, which produces a new dataset, where each keyword-author pair is mapped to a value of 1. Then the `REDUCE` operation op4 aggregates the number of occurrences for each keyword-author pair. The next `MAP` operation op5 transforms the dataset by mapping each keyword to a pair of author-occurrences. The `REDUCE` op6 finds for each keyword the author with the maximum number of occurrences, thus finding the expert who has mostly used the keyword. Finally, the `PROJECT` defines the mapping between the dataset fields and the columns of the returned relation.

Query Processing. First, Query 1 is compiled into the preliminary execution plan, depicted in Figure 4.2. Then, the query optimizer finds the opportunity for pushing down the condition `affiliation = 'INRIA'` into the relational data store. Thus, the selection condition is included in the WHERE clause of the subquery for `rdb`. Doing this, the compiler determines that the column `s.affiliation` is no longer referenced in the common execution plan, so it is simply removed from the corresponding projection on `scientists` from `rdb`. This pushdown implies increasing the selectivity of the subquery, which is identified by the optimizer as an opportunity for performing a bind join. To further verify this opportunity, the query optimizer asks `rdb` to estimate the cardinality for the rewritten SQL subquery and, considering also the availability of an index on the field `author` in the MongoDB collection `publications`, the optimizer plans for bind join by pushing into the sub-plan for MongoDB the selection condition `author IN <authors>`, where `<authors>` refers to the list of distinct values of the `s.name` column, which will be determined at runtime.

Analogously, by using the catalog information provided by the MongoDB wrapper to estimate the cardinality of the join between scientists and publications, the optimizer plans to also involve the MFR subquery into a bind join and thus pushes the bind join condition `kw IN (<keywords>)`. Here, `<keywords>` is a placeholder for the list of distinct keywords retrieved from the column `p.keywords`. Recall that each value `in p.keywords` is an array, so the query processor will have to first flatten the intermediate relation by transforming the array-type column `p.keywords` to a scalar-type column named _keywords. Since `p.keywords` participates in the join condition `kw IN`

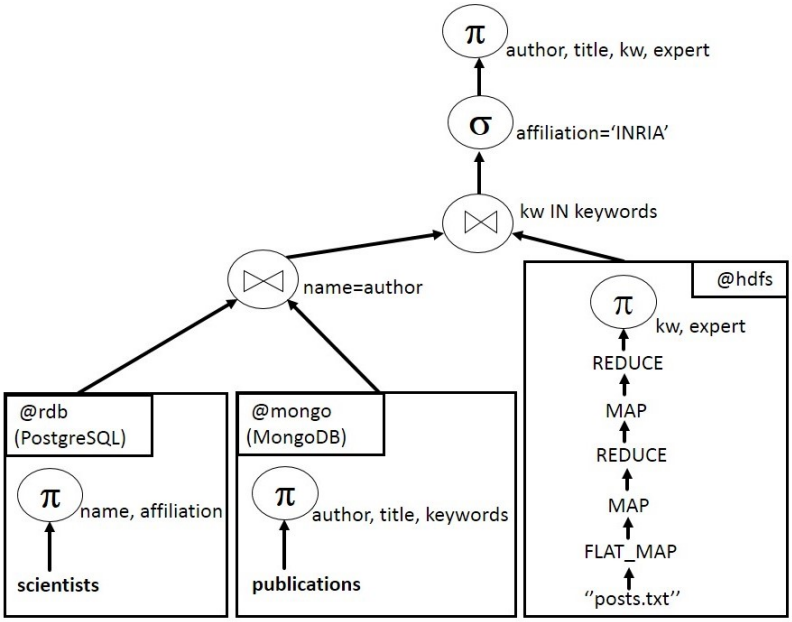


Figure 4.2 – **Preliminary query plan for Query 1**

`keywords`, its flattening leads to transforming the join to an equi-join which allows for the query engine to utilize efficient methods for equi-joins.

Furthermore, the MFR planner seeks for opportunities to move the bind join filter condition `kw IN (<keywords>)` earlier in the MFR operation chain by applying the MFR rewrite rules, explained below. At this stage, although `<keywords>` is not known, the planner has all the information needed to apply the rules. After these transformations, the optimized query plan (Figure 4.3) is executed by the query processor. In this notation, we use the symbol **F** to denote the flattening operator.

To execute the query plan, the query engine takes the following steps:

1. The query processor delivers to the wrapper of `rdb` the following SQL statement, rewritten by taking into account the pushed selection condition, for execution against the PostgreSQL data store, and waits for the corresponding result set to be retrieved in order to compose the bind join condition for the next step.

```
SELECT name
FROM scientists
WHERE affiliation = 'INRIA'
```

2. The MongoDB wrapper prepares a native query to send to the MongoDB data store to retrieve those tuples from `publications` that match the bind join criteria. It takes into account the bind join condition derived from the already retrieved data

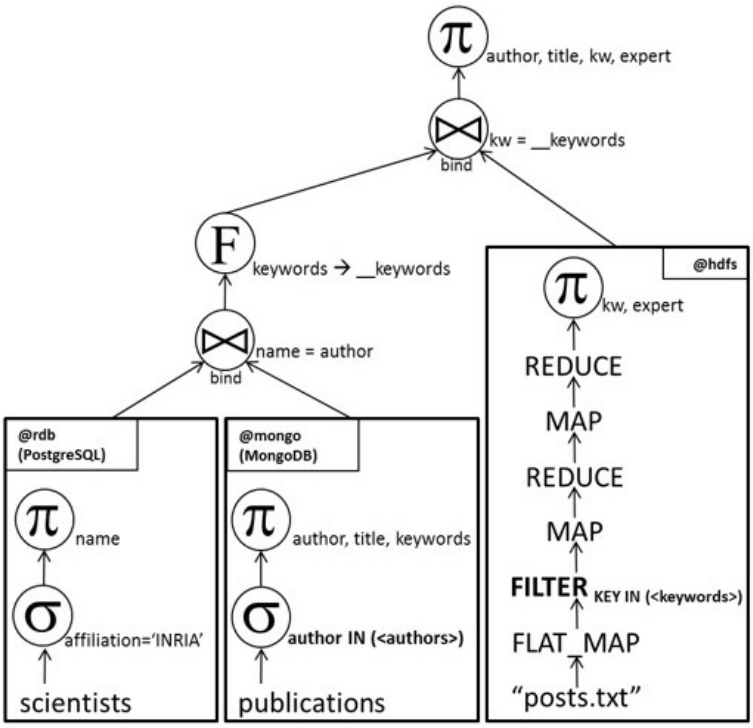


Figure 4.3 – **Optimized query plan for Query 1**

Name
Patrick
Boyan

from `rdb` and generates a MongoDB query whose SQL equivalent would be the following:

```
SELECT title, author, keywords FROM publications
WHERE author IN ('Patrick', 'Boyan')
```

However, the wrapper does not generate an SQL statement; instead it generates directly the corresponding MongoDB native query:

```
db.publications.find(
{ author: {$in:['Patrick', 'Boyan']} },
{ title: 1, author: 1, keywords: 1, _id: 0 } )
```

Upon receiving the result dataset (a MongoDB document collection), the wrapper converts it to a table, according to the signature of the named table expression

publications, ready to be joined with the already retrieved result set from step 1. The result of the bind join is the contents of the following intermediate relation:

author	title	keywords
Patrick	Principles of DDBS	['cloud', 'storage']

3. The flattening operator transforms the intermediate relation from step 2 to the following one:

author	title	_keywords
Patrick	Principles of DDBS	cloud
Patrick	Principles of DDBS	storage

4. The query processor identifies a list of the distinct values of the join attribute _keywords and derives from it the bind join condition kw IN ('cloud', 'storage') to push inside the subquery against hdfs.

5. The MFR planner for the wrapper of hdfs decides at which stage of the MFR sequence to insert the filter, by applying a number of rewrite rules. According to rule #1, the planner initially inserts the filter just before the PROJECT op7 by rewriting the condition expression as follows:

```
.FILTER( KEY IN ('cloud', 'storage') )
```

Next, by applying consecutively rules #2 and #3, the planner moves the FILTER before the MAP op5 by rewriting its condition expression according to rule #3:

```
.FILTER( KEY[0] IN ('cloud', 'storage') )
```

Analogously, rules #2 and #3 are applied again, moving the FILTER before op3, rewriting the expression once again, and thus settling it to its final position. After all transformations the MFR subquery is converted to the final MFR expression below.

```
SCAN( TEXT, 'posts.txt', ',' )
.FLAT_MAP( lambda data:  product(data[2:], [data[1]]) )
.FILTER( TUPLE[0] IN ('cloud', 'storage') )
.MAP( TUPLE, 1 )
.REDUCE( SUM )
.MAP( KEY[0], (KEY[1], VALUE) )
.REDUCE( lambda a, b:  b if b[1] > a[1] else a )
```

6. The wrapper interprets the reordered MFR sequence, translates it to the Python script below as per the Python API methods of Spark, and executes it within the Spark framework.

```
sc.textFile('posts.txt').map( lambda line:line.split(',') ) \
.flatMap( lambda data:  product(data[2:], [data[1]]) ) \
.filter( lambda tup:  tup[0] in ['cloud','storage'] ) \
.map( lambda tup:  (tup, 1) ) \
.reduceByKey( lambda a, b:  a + b ) \
.map( lambda tup:  (tup[0][0], (tup[0][1], tup[1])) ) \
.reduceByKey( lambda a, b:  b if b[1] > a[1] else a )
```

   The result of MFR query reordering and interpreting on Spark is another intermediate relation:

kw	expert
cloud	alice
storage	alice

7. The intermediate relations from steps 3 and 6 are joined to produce the final result that lists the suggested experts for each publication regarding the given keywords:

author	title	kw	expert
Patrick	Principles of DDBS	cloud	alice
Patrick	Principles of DDBS	storage	alice

4.6 Conclusion

In this chapter, we extended the CloudMdsQL query language with a simple notation to specify the sequence of MFR operators for the data processing frameworks , and we propose a query engine based on CloudMdsQL query engine in order to integrate data from relational, NoSQL, and big data stores (such as HDFS).

Our query language can directly express subqueries that can take full advantage of the functionality of the underlying data stores and processing frameworks. Furthermore, it allows for query optimization, so that the query operator execution sequence specified by the user may be reordered by taking into account the properties of map/filter/reduce operators together with the properties of relational operators.

Chapter 5

Prototype

To validate our proposed MFR extension, we have developed a prototype as part of the CloudMdsQL query engine. Remember form Chapter 3 that each query engine node consists of two parts (master and worker) and is collocated at each data store node in a computer cluster. A master node takes as input a query and produces a query plan, which it sends to one chosen query engine node for execution. It uses a query planner that performs query analysis and optimization, and produces a query plan serialized that can be easily transferred across query engine nodes. Workers collaborate to execute a query plan, produced by a master, against the underlying data stores involved in the query. Each worker node acts as a lightweight runtime database processor atop a data store and is composed of three generic modules (i.e. same code library) - query execution controller, operator engine, and table storage - and one wrapper module that is specific to a data store.

In this chapter, we describe the implementation of the CloudMdsQL query engine and our MFR extension. It is based on [38, 37].

This chapter is organized as follows. Section 5.1 gives an overview of the chapter. Section 5.2 describes the query planner component, which compiles a CloudMdsQL query and generates a query execution plan to be processed by the query engine. Section 5.3 presents the execution engine with its modules (query execution controller, operator engine, table storage and wrappers). Section 5.4 introduces the specific wrappers that have been implemented in the CoherentPaaS project to interface with data stores. Section 5.5 concludes.

5.1 Overview

The current implementation of the query engine uses a modified version of the open source Derby DBMS to accept CloudMdsQL queries and transform the corresponding execution plan into Derby SQL operations. To extend the CloudMdsQL query engine with MFR, we developed an MFR planner to be used by the wrapper of the data processing framework (DPF). The MFR planner finds optimization opportunities and translates

the resulting sequence of MFR operations to a sequence of DPF's API methods to be executed. Figure 5.1 shows the various components of the query engine prototype.

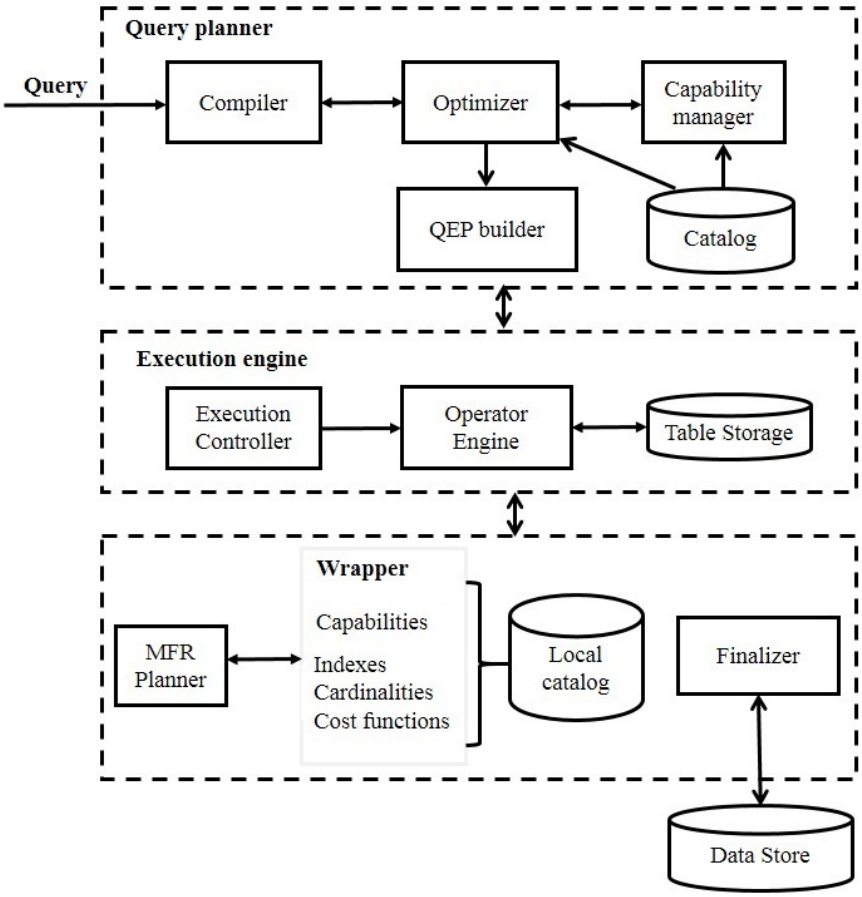


Figure 5.1 – **Query engine implementation components**

5.2 Query Planner

The query planner is implemented in C++; it compiles a CloudMdsQL query and generates a query execution plan (QEP) to be processed by the query execution engine. The result of the query planning is the JSON serialization of the generated QEP, which is represented as a directed acyclic graph, where leaf nodes are references to named tables and all other nodes represent relational algebra operations. The query planning process goes through several phases, which we briefly focus on below.

The query **compiler** uses the Boost.Spirit framework for parsing context-free grammars, following the recursive descent approach. Boost.Spirit allows grammar rules to be defined by means of C++ template metaprogramming techniques. Each grammar rule has

an associated semantic action, which is a C++ function that should return an object, corresponding to the grammar rule.
The compiler first performs lexical and syntax analyzes of a CloudMdsQL query to decompose it into an abstract syntax tree (AST) that corresponds to the syntax clauses of the query.

At this stage the compiler identifies a forest of sub-trees within the AST, each of which is associated to a certain data store (labeled by a named table) and meant to be delivered to the corresponding wrapper to translate it to a native query and execute it against the data store. The rest of the AST is the part that will be handled by the common query engine (the common query AST).

Furthermore, the compiler performs a semantic analysis of the AST by first resolving the names of tables and columns according to the ad-hoc schema of the query following named table signatures. Datatype analysis takes place to check for datatype compatibilities between operands in expressions and to infer the return datatype of each operation in the expression tree, which may be further verified against a named table signature, thus identifying implicit typecasts or type mismatches. WHERE clause analysis is performed to discover implicit equi-join conditions and opportunities for moving predicates earlier in the common plan. The crossreference analysis aims at building a graph of dependencies across named tables. Thus the compiler identifies named tables that participate in more than one operation, which helps the execution controller to plan for storing such intermediate data in the table storage. In addition, the optimizer avoids pushing down operations in the sub-trees of such named tables. To make sure that the dependency graph has no cycles, hence the generated QEP will be a directed acyclic graph, the compiler implements a depth-first search algorithm to detect and reject any query that has circular references.

Error handling is performed at the compilation phase. Errors are reported as soon as they are identified (terminating the compilation execution), together with the type of the error and the context, within which they were found (e.g. unknown table or column, ambiguous column references, incompatible types in expression, etc.).

The query **optimizer** uses the cost information in the **catalog** and implements a simple exhaustive search strategy to explore all possible rewritings of the initial query, by pushing down select operations, expressing bind joins, join ordering, and intermediate data shipping. Furthermore, it uses the **capability manager** to validate each rewritten subquery against its data store capability specification, which is exported by the wrapper into the catalog in the form of a JSON schema. Thus, the capability manager simply serializes each sub-tree of the AST into a JSON object and attempts to validate it against the corresponding JSON schema. This allows a rewritten sub-plan to a data store to be validated by the query planner before it is actually delivered to the wrapper for execution; and in case the validation fails, the rewriting action (e.g. selection pushdown) is reverted.

The **QEP builder** is responsible for the generation of the final QEP, ready to be handled by the query execution engine, which also includes: resolving attribute names to column ordinal positions considering the named table expression signatures, removing columns from intermediate projections in case they are no longer used by the operations

above (e.g. as a result of operation pushdown), and serializing the QEP to JSON.

5.3 Execution Engine

The main reasons to choose Derby data store to implement the operator engine are because Derby:

- Allows extending the set of SQL operations by means of `CREATE FUNCTION` statements. This type of statements creates an alias, which an optional set of parameters, to invoke a specific Java component as part of an execution plan.
- Has all the relational algebra operations fully implemented and tested.
- Has a complete implementation of the JDBC API.
- Allows extending the set of SQL types by means of `CREATE TYPE` statements. It allows working with dictionaries and arrays.

  Having a way to extend the available Derby SQL operations allows designing the resolution of the named table expressions. In fact, the query engine requires three different components to resolve the result sets retrieved from the named table expressions:

  - `WrapperFunction`: To send the partial execution plan to a specific data store using the wrappers interfaces and retrieve the results.
  - `PythonFunction`: To process intermediate result sets using Python code.
  - `NestedFunction`: To process nested CloudMdsQL queries.

    Named table expressions admit parameters using the keyword `WITHPARAMS`. However, the current implementation of the `CREATE FUNCTION` statement is designed to bind each parameter declared in the statement with a specific Java method parameter. In fact, it is not designed to work with Java methods that can be called with a variable number of parameters, which is a feature introduced since Java 6. To solve this gap, we have modified the internal validation of the `CREATE FUNCTION` statement and how to invoke Java methods with a variable number of parameters during the evaluation of the execution plan. For example, imagine that the user declares a named table expression `T1` that returns 2 columns (x and y) and has a parameter called a as follows:

```
T1(x int, y string
  WITHPARAMS a string)@db1 =
( SELECT x, y FROM tbl WHERE id = $a )
```

    The query execution controller will produce dynamically the following `CREATE FUNCTION` statement:

```
CREATE FUNCTION T1 ( a VARCHAR( 50 ) )
RETURNS TABLE ( x INT, y VARCHAR( 50 ))
LANGUAGE JAVA
PARAMETER STYLE DERBY_JDBC_RESULT_SET
READS SQL DATA
EXTERNAL NAME 'WrapperFunction.execute'
```

It is linked to the following Java component, which will use the wrapper interfaces to establish a communication with the data store `db1`:

```
public class WrapperFunction {
 public static ResultSet execute(
  String namedExprName,
  Long queryId,
  Object...  args
  /*dynamic args*/
 ) throws Exception {
 //Code to invoke the wrappers } }
```

Therefore, after accepting the execution plan in JSON format, the query execution controller parses it, identifies the sub-plans within the plan that are associated to a named table expression and dynamically executes as many CREATE FUNCTION statements as named table expressions exist with a unique name. As a second step, the execution engine evaluates which named expressions are queried more than once and must be cached into the temporary table storage, which will be always queried and updated from the specified Java functions to reduce the query execution time. Finally, the last step consists of translating all operation nodes that appear in the execution plan into a Derby specific SQL execution plan. Once the SQL execution plan is valid, the Derby core (which acts as the operator engine) produces a dynamic byte code that resolves the query that can be executed as many times as the application needs.

Derby implements the JDBC interface and an application can send queries though the Statement class. So, when the user has processed the query result and closed the statement, the query execution controller drops the previously created functions and cleans the temporary table storage.

To process data in distributed data stores we used a specific implementation of the Operator Engine and the MFR wrapper, adapting the parallel SQL engine Spark SQL ([10]) to serve as the operator engine, thus taking full advantage of massive parallelism when joining HDFS with relational data. To do this, each execution (sub-)plan is translated to a flow of invocations of Spark SQL's DataFrame API methods.

5.4 Wrappers

The wrappers are Java classes implementing a common interface used by the operator engine to interact with them. A wrapper may store locally catalog information and capabilities, which it provides to the query planner periodically or on demand. Each wrapper also implements a finalizer, which translates a CloudMdsQL sub-plan to a native data store query.

We have validated the query engine using four data stores – Sparksee (a graph data store with Python API), Derby (a relational data store accessed through its Java Database Connectivity (JDBC) driver), MongoDB (a document data store with a Java API), and unstructured data stored in an HDFS cluster and processed using Apache Spark as big data processing framework (DPF). To be able to embed subqueries against these data stores, we developed wrappers for each of them as follows.

The wrapper for Sparksee accepts as raw text the Python code that needs to be executed against the graph data store using its Python client API in the environment of a Python interpreter embedded within the wrapper.

The wrapper for Derby executes SQL statements against the relational data store using its JDBC driver. It exports an `explain()` function that the query planner invokes to get an estimation of the cost of a subquery. It can also be queried by the query planner about the existence of certain indexes on table columns and their types. The query planner may then cache this metadata information in the catalog.

The wrapper for MongoDB is implemented as a wrapper to an SQL compatible data store, i.e. it performs native MongoDB query invocations according to their SQL equivalent. The wrapper maintains the catalog information by running probing queries such as `db.collection.count()` to keep actual data store statistics, e.g. cardinalities of document collections. Similarly to the Derby wrapper, it also provides information about available indexes on document attributes.

The MFR wrapper implements an MFR planner to optimize MFR expressions in accordance with any pushed down selections. The wrapper uses Spark's Python API, and thus translates each transformation to Python lambda functions. Besides, it also accepts raw Python lambda functions as transformation definitions. The wrapper executes the dynamically built Python code using the reflection capabilities of Python by means of the `eval()` function. Then, it transforms the resulting RDD into a Spark DataFrame.

5.5 Conclusion

In this chapter, we described the prototype of the CloudMdsQL query engine and our MFR extension. The query engine includes a query planner that performs query analysis and optimization, and produces a query plan, and a lightweight runtime database processor atop each data store that is composed of three generic modules (i.e. same code library) - query execution controller, operator engine, and table storage - and one wrapper module that is specific to a data store. The query planner is implemented in C++, the operator

engine is based on the Derby DBMS; and the wrappers are implemented in Java.

The current implementation of the query engine uses a modified version of the open source Derby DBMS to accept CloudMdsQL queries and transform the corresponding execution plan into Derby SQL operations. To extend the CloudMdsQL query engine with MFR, we developed an MFR planner in Java to be used by the wrapper of the data processing framework (DPF). The MFR planner finds optimization opportunities and translates the resulting sequence of MFR operations to a sequence of DPF's API methods to be executed.

We validated the CloudMdsQL query engine with wrappers for four data stores: Sparksee, a graph data store with Python API; Derby, a relational data store accessed through its JDBC driver; MongoDB, a document data store with a Java API; and Apache Spark a data processing framework on top of HDFS, accessed by Apache Spark API.

Chapter 6

Experimental Validation

Based on the CloudMdsQL query engine prototype (see Chapter 5), we give an experimental validation of multistore query processing in a cluster to evaluate the impact on performance of optimization. More specifically, we explore the performance benefit of using bind join, a very efficient technique, under different conditions. In our experimental validation, we focus on queries that can express an integration of data across several data stores, in particular, NoSQL (graph and document data stores), RDBMS and HDFS accessed through the Spark framework. This chapter is based on [40, 14].

This chapter is organized as follows. Section 6.1 describes our experimental setup. Section 6.2 shows the CloudMdsQL experimental validation. Section 6.3 presents the MFR experimentation. Section 6.4 concludes.

6.1 Experimental Setup

We described below both setups used for experiments.

CloudMdsQL Experimental Setup

We loaded the generated datasets in 4 data stores, each running on a separate node in a cluster, as follows: Apache Derby at node_1 stores the `scientists` table, MongoDB at node_2 and node_3 stores respectively the `publications` and `reviews` document collections, and the Sparksee graph data store at node_4. The data store identifiers that we use within our queries are respectively `DB1`, `DB2`, `DB3`, and `DB4`. Each node in the cluster runs on a quad-core CPU at 2.4GHz, 32 GB main memory, 1.5Gbps HDD throughput, and the network bandwidth is 1Gbps.

MFR Experimental Setup

To evaluate the impact of optimization on query execution, we use a cluster of the `GRID 5000` platform ([3]), with one node for PostgreSQL and MongoDB and 4 to 16 nodes for the HDFS cluster. The Spark cluster, used as both the DPF and the query processor,

is collocated with the HDFS cluster. Each node in the cluster runs on 16 CPU cores at 2.4GHz, 64GB main memory, and the network bandwidth is 10Gbps.

6.2 CloudMdsQL Experimentation

The experimental validation shows the ability of the query engine to optimize CloudMdsQL queries, as optimizability is one of the objectives of the query language. Our experiment illustrate the impact of each optimization technique on the overall efficiency of the query execution.

In this section, we introduce the datasets, and we present our experimental results.

6.2.1 Datasets

We performed our experimental evaluation in the context of the use case example, presented in Section 3.7. For this purpose, we generated data to populate the Derby table scientists, the MongoDB document collections publications and reviews, and the Sparksee graph data store with scientists and friendship relationships between them. The datasets have the following characteristics:

- Table `scientists` contains 10k rows, distributed over 1000 distinct affiliations, thus setting to 0.1% the selectivity of an arbitrary equality condition on the `affiliation` attribute.
- Collection `publications` contains 1M documents, with uniform distribution of values of the `author` attribute, making 100 publications per scientist. The total size of the collection is 1GB.
- Collection `reviews` contains 4M documents, making 4 reviews per publication. The `date` attribute contains values between 2012-01-01 and 2014-12-31. This sets to 33% the selectivity of the predicate `year(date) = 2013`. The review attribute contains long string values. The total size of the collection is 20GB.
- The graph data store contains one node per scientist and 500k edges between them. This data is generated to assure that for each publication, 2 out of 4 reviewers are friends or friend-of-friends to the author.
- The catalog contains sufficient information, collected through the Derby and MongoDB wrappers, about the above specified cardinalities and selectivities. It also contains information about the presence of indexes on the attributes `scientists.affiliation`, `publications.id`, `publications.author`, `reviews.date`, `reviews.pub_id`, and `reviews.reviewer`,

6.2.2 Experimental Results

We prepared 5 different queries. For each of them we chose 3 alternative QEPs to run and compare their execution times, with different join orders, intermediate data transfer, and subquery rewritings. The execution times for the different QEPs are illustrated in each query's corresponding graphical chart.

All the queries use the following common named table expressions, which we created as stored expressions:

```
CREATE NAMED EXPRESSION
scient( name string, affiliation string )@DB1 = (
    SELECT name, affiliation FROM scientists );
CREATE NAMED EXPRESSION
pubs( id int, title string, author string )@DB2 = (
    SELECT id, title, author FROM publications );
CREATE NAMED EXPRESSION
revs( pub_id int, reviewer string, date timestamp, review string )@DB3 =
    ( SELECT pub_id, reviewer, date, review FROM reviews );
CREATE NAMED EXPRESSION
friends( name string, friend string JOINED ON name
     CARDINALITY = 100*card(Outer) )@DB4 =
    {* for n in CloudMdsQL.Outer:
     for f in graph.GetNeighboursByName( n ):
      yield ( n, f.getName() ) *};
   CREATE NAMED EXPRESSION
friendships( person1 string, person2 string, friendship string
     JOINED ON person1, person2 WITHPARAMS maxlevel int
     CARDINALITY = card(Outer) )@DB4 =
    {* for (p1, p2) in CloudMdsQL.Outer:
     sp = graph.FindShortestPathByName( p1, p2, $maxlevel )
     if sp.exists():
      yield ( p1, p2, 'friend' + '-of-friend' * (sp.get_cost()-1) ) *};
```

Thus, each of the queries is expressed as a single SELECT statement that uses the above named table expressions. For each of the queries we describe the alternative QEPs with a text notation, using the special symbols $\ltimes$ for joins, $\bowtie$ for bind joins (where the join condition is bound to the right side of the join), $\sigma()$ for selections, and @ in subscript to denote the node at which the operation is performed. If a selection is marked with **@QE** in subscript, then it is performed by the query engine, otherwise it is pushed down to be executed by the data store. The operation order is specified explicitly using parentheses. The relations within the QEP are referred with their first letter in capital, e.g. `R` stands for `reviews`.

Query 1 involves 2 tables and focuses on selection pushdowns and bind joins. The selectivity of the WHERE clause predicate is approximately 0.1%, which explains the benefit of the pushed down selection in $\texttt{QEP}_{12}$ that reduces significantly the data retrieved from the reviews document collection in DB3. Using a bind join in $\texttt{QEP}_{13}$ reduces to 0. % the data retrieved from the `publications` collection.

```
SELECT p.id, p.title, p.author,
     r.reviewer, r.review
FROM pubs p JOIN revs r ON p.id = r.pub_id
WHERE r.date = '2013-05-01'
```

The alternative query plans are:

$\texttt{QEP}_{11}$: σ_{QE} **(R)** $\bowtie_{@3}$ **P**
$\texttt{QEP}_{12}$: σ **(R)** $\bowtie_{@3}$ **P**
$\texttt{QEP}_{13}$: σ **(R)** $\bowtie_{@3}$ **P**

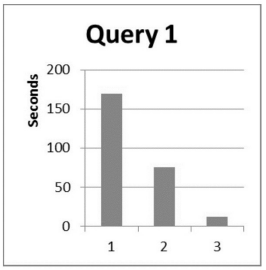


Figure 6.1

Query 2 involves 3 tables and focuses on the importance of choosing the optimal data shipping direction. All the plans involve the retrieval and transfer of a selection (6GB) on the `reviews` collection and the entire `publications` collection (1GB). $\texttt{QEP}_{21}$ retrieves both tables remotely. $\texttt{QEP}_{22}$ retrieves `P` locally and `R` remotely. $\texttt{QEP}_{23}$ retrieves `R` locally and $\sigma(S) \bowtie$ `P` (only 1MB) remotely. Although bind joins are applicable in all QEPs, we do not use them in order to focus on shipping of unfiltered data.

```
SELECT p.id, p.title, p.author, r.reviewer, r.review
FROM pubs p JOIN revs r ON p.id = r.pub_id
    JOIN scient s ON s.name = p.author
WHERE r.date BETWEEN '2013-01-01' AND '2013-12-31'
    AND s.affiliation = 'affiliation1'
```

The alternative query plans are:

QEP_{21}: **(σ (S) ⋈ @1 P) ⋈ @1 σ (R)**
QEP_{22}: **(σ (S) ⋈ @2 P) ⋈ @2 σ (R)**
QEP_{23}: **(σ (S) ⋈ @2 P) ⋈ @3 σ (R)**

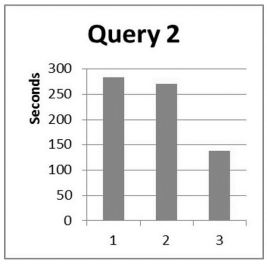


Figure 6.2

Query 3 involves 3 tables, of which the table `scientists` is used twice. To distinguish them, in the description of QEPs we use the symbols **Sa** and **Sr.** Because of the use of bind joins, this query handles much less data and executes much faster compared to the previous queries. The query focuses on different join orders, the effect of which comes mostly from the different selectivities of the bind join conditions.

```
SELECT p.id, p.title, p.author, r.reviewer, r.review, sr.affiliation
FROM pubs p JOIN revs r ON p.id = r.pub_id
    JOIN scient sa ON sa.name = p.author
    JOIN scient sr ON sr.name = r.reviewer
WHERE sa.affiliation = 'affiliation1' AND
    sr.affiliation IN ('affiliation2', 'affiliation3'
```

The alternative query plans are:

QEP_{31}: **((σ (Sr) ⋈@3 R) ⋈@3 P) ⋈@3 σ (Sa)**
QEP_{32}: **((σ (Sa) ⋈@2 P) ⋈@3 R) ⋈@3 σ (Sr)**
QEP_{33}: **(σ (Sa) ⋈@2 P) ⋈@3 (σ (Sr) ⋈@3 R)**

Query 4 includes the `friendships` subquery against the graph data store and focuses on the involvement of native named table expressions, using join iteration, and the usage of expensive native operations, such as breadth-first search. As the QEPs correspond to the ones for Query 3, the execution times depend on the join orders, but also on the number of distinct values of the relation to be joined with the `friendships` expression, which determines how many times breadth-first search is invoked.

```
SELECT p.id, p.title, p.author, r.reviewer,
```

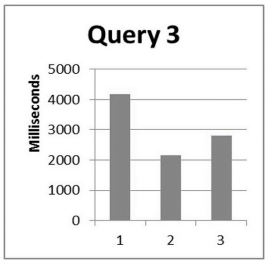


Figure 6.3

```
     r.review, f.friendship
FROM pubs p JOIN revs r ON p.id = r.pub_id
    JOIN scient sa ON sa.name = p.author
    JOIN scient sr ON sr.name = r.reviewer
    JOIN friendships(2) f ON p.author = f.person1
      AND r.reviewer = f.person2
WHERE sa.affiliation = 'affiliation1' AND
    sr.affiliation IN ('affiliation2', 'affiliation3')
```

The alternative query plans are:

QEP_{41}: **(((σ (Sr)** $\bowtie_{@3}$ **R)** $\bowtie_{@3}$ **P)** $\bowtie_{@3}$ **F)** $\bowtie_{@3}$ σ **(Sa)**
QEP_{42}: **(((σ (Sa)** $\bowtie_{@2}$ **P)** $\bowtie_{@3}$ **R)** $\bowtie_{@3}$ **F)** $\bowtie_{@3}$ σ **(Sr)**
QEP_{43}: **((σ (Sa)** $\bowtie_{@2}$ **P)** $\bowtie_{@3}$ **(σ (Sr)** $\bowtie_{@3}$ **R))** $\bowtie_{@3}$ **F**

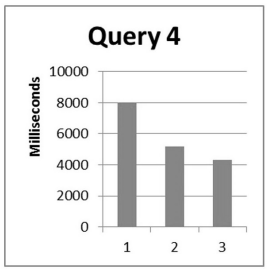


Figure 6.4

Query 5 resembles Query 4, but uses the `friends` native subquery that invokes another native operation that yields many output tuples for a single input tuple. Like for Query 4, the join order determines when the native expression is invoked and the number of its input tuples.

```
SELECT p.id, p.title, p.author, r.reviewer,
```

```
     r.review, f.friend
FROM pubs p JOIN revs r ON p.id = r.pub_id
    JOIN scient sa ON sa.name = p.author
    JOIN scient sr ON sr.name = r.reviewer
    JOIN friends f ON r.reviewer = f.name
WHERE sa.affiliation = 'affiliation1' AND

sr.affiliation IN ('affiliation2', 'affiliation3')
```

The alternative query plans are:

QEP_{51}: **(((**σ **(Sr)** $\bowtie_{@3}$ **R)** $\bowtie_{@3}$ **P)** $\bowtie_{@3}$ **F)** $\bowtie_{@3}$ σ **(Sa)**
QEP_{52}: **(((**σ **(Sa)** $\bowtie_{@2}$ **P)** $\bowtie_{@3}$ **R)** $\bowtie_{@3}$ **F)** $\bowtie_{@3}$ σ **(Sr)**
QEP_{53}: **((**σ **(Sa)** $\bowtie_{@2}$ **P)** $\bowtie_{@3}$ **(**σ **(Sr)** $\bowtie_{@3}$ **R))** $\bowtie_{@3}$ **F**

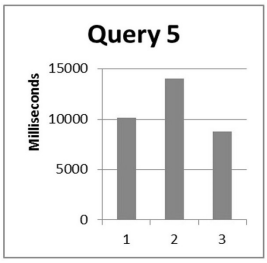


Figure 6.5

6.3 MFR Experimentation

This section, reports on experiments designed to test the effectiveness of our approach (extending CloudMdsQL with MFR) through experimental validation with three data stores and representative queries.

In this section, we first introduce the datasets, based on the use case example in chapter 4. Finally, we present our experimental results.

6.3.1 Datasets

Our experimental evaluation is based on the use case example presented in 4.5. For this purpose, we generated data to populate the PostgreSQL table `scientists`, the MongoDB document collection `publications`, and text files with unstructured log data stored in HDFS. The datasets have the following characteristics:

- Table `scientists` contains 10K rows, distributed over 1000 distinct affiliations, making 10 authors per affiliation.
- Collection `publications` contains 10M documents, with uniform distribution of values of the author attribute, making 1K publications per scientist. Each publication is randomly assigned a set of 6 to 10 keywords out of 10K distinct `keyword` values. Also, there is an association between authors and keywords, so that all the publications of a single author reference only 1% of all the keywords. This means that a join involving the publications of a single author will have a selectivity factor of 1%; hence 100 distinct values for the bind join condition. The total size of the collection is 10GB.
- HDFS contains 16K files distributed between the nodes, with 100K tuples per file making 1.6 billion tuples, corresponding to posts from 10K forum users with 10K distinct keywords mentioned by them. The first field of each tuple is a timestamp and does not have an impact on the experimental results. The second field contains the author of the post as a string value. The remainder of the tuple line contains 1 to 10 `keyword` string values, randomly chosen out of the same set of 10K distinct keywords. The total size of the data is 124GB.

6.3.2 Experimental Results

We prepared 3 different queries. We execute each of them in three different HDFS cluster setups - with 4, 8, and 16 nodes. We compare the execution times without and with bind join to the MFR subquery, which are illustrated in each query's corresponding graphical chart. We do not focus on evaluating the bind join between PostgreSQL and MongoDB, as its benefit is less significant when compared to the benefit of doing bind join to the MFR subquery, because of the big difference in data sizes.

All the queries use the following common named table expressions, which we created as stored expressions:

```
CREATE NAMED EXPRESSION
scientists( name string, affiliation string )@rdb = (
   SELECT name, affiliation
   FROM scientists );

CREATE NAMED EXPRESSION
publications(author string, title string, keywords array)@mongo = (
   SELECT author, title, keywords
   FROM publications );

CREATE NAMED EXPRESSION
experts(kw string, expert string)@hdfs = {*
   SCAN(TEXT, 'posts.txt', ',')
```

```
     .FLAT_MAP( lambda data:  product(data[2:], [data[1]]) )
     .MAP( TUPLE, 1 )
     .REDUCE( SUM )
     .MAP( KEY[0], (KEY[1], VALUE) )
     .REDUCE( lambda a, b:  b if b[1] > a[1] else a )
     .PROJECT(KEY, VALUE[0]) *};

CREATE NAMED EXPRESSION
experts_alt(kw string, expert string)@hdfs = {*
   SCAN(TEXT, 'posts.txt', ',')
     .FLAT_MAP( lambda data:  product(data[2:], [data[1]]) )
     .MAP_VALUES(lambda v:  Counter([v]))
     .REDUCE(lambda C1, C2:  C1 + C2)
     .MAP_VALUES( lambda C: \
      reduce(lambda a,b:  b if b[1] > a[1] else a, C.items()) )
     .PROJECT(KEY, VALUE[0]) *};
```

Each of the queries is expressed as a single SELECT statement that uses the above named table expressions. The named tables `scientists`, `publications`, and `experts` have exactly the same definition as in the use case example from Section 4.5.

The named table `experts_alt` does the same as `experts`, but its MFR sequence contains only one `REDUCE` (respectively, it does only one shuffle) and more complex map functions. It uses Python's Counter dictionary collection, with the additive property to sum up numeric values grouped by the key. The first `MAP_VALUES` maps a keyword to a Counter object, initialized with a single author key. Then the `REDUCE` sums all Counter objects associated to a single keyword, so that the result from it is an aggregated Counter dictionary, where an author is mapped to a number of occurrences of the keyword. The final `MAP_VALUES` uses Python's `reduce()` function (note that this is not Spark's reduce operator) to choose from all items in a Counter the author with the highest number of occurrences for a keyword.

Query 0 involves only the MongoDB data store and the DPF to find experts for the publications of only one author. Thus, the selectivity factor of the bind join is 1%, as the number of keywords used by a single author is 1% of the total number of keywords. As we experimented with different number of nodes, we observe that the query execution efficiency and the benefit of the bind join scale well when the number of nodes increases. This is also observed in the rest of the queries.

–Query0

```
SELECT p.author, p.title,
e.kw, e.expert
FROM publications p, experts e
WHERE p.author = 'author1'
```

```
   AND e.kw IN p.keywords
```

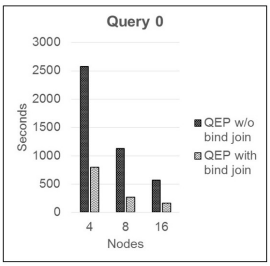


Figure 6.6

Query 1, as already introduced in Section 4.5, involves all the data stores and aims at finding experts for publications of authors with a certain affiliation. This makes a selectivity factor of 10% for the bind join, as there are 10 authors per affiliation. In addition, we explore another variant of the query, filtered to three affiliations, or 30% selectivity factor of the bind join. We enumerate the two variants as Query1.1 and Query 1.2.

–Query 1.1: selectivity factor 10%

```
SELECT p.author, p.title, e.kw, e.expert
FROM scientists s, publications p, experts e
WHERE s.affiliation = 'affiliation1'
   AND p.author = s.name AND e.kw IN p.keywords
```

–Query 1.2: selectivity factor 30%

```
SELECT p.author, p.title, e.kw, e.expert
FROM scientists s, publications p, experts e
WHERE s.affiliation IN ('affiliation1','affiliation2','affiliation3')
   AND p.author = s.name AND e.kw IN p.keywords
```

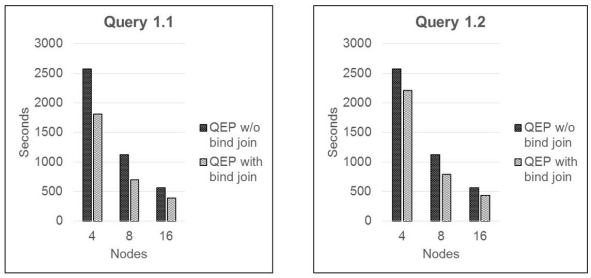


Figure 6.7

Query 2 does the same as Query 1, but uses the MFR subquery `experts_alt`, which uses more sophisticated map functions, but makes only one shuffle, where the key is a keyword. For comparison, the MFR expression `experts` makes two shuffles, of which the first one uses a bigger key, composed of a keyword-author pair. Therefore, the corresponding Spark computation of Query 2 involves much smaller size of data to be shuffled compared to Query 1, which explains its better overall efficiency and higher relative benefit of using bind join. Like with Query 1, we explore two variants with different selectivity factors of the bind join condition.

–Query 2.1: selectivity factor 10%

```
SELECT p.author, p.title, e.kw, e.expert
FROM scientists s, publications p, experts_alt e
WHERE s.affiliation = 'affiliation1'
   AND p.author = s.name AND e.kw IN p.keywords
```

–Query 2.2: selectivity factor 30%

```
SELECT p.author, p.title, e.kw, e.expert
FROM scientists s, publications p, experts_alt e
WHERE s.affiliation IN ('affiliation1',
'affiliation2', 'affiliation3')
   AND p.author = s.name AND e.kw IN p.keywords
```

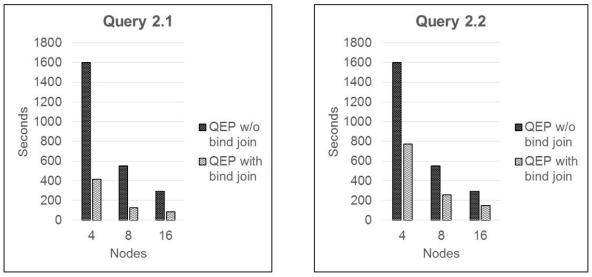


Figure 6.8

The results show the significant benefit of performing bind join in our experimental scenario, despite the overhead it produces.

6.4 Conclusion

In this chapter, we gave our experimental validation of CloudMdsQL and our MFR extension. More specifically, we explored the performance benefit of using bind join, a very efficient technique, under different conditions. In our experimental validation, we focused

on queries that can express an integration of data across several data stores, in particular, NoSQL (graph and document data stores), RDBMS and HDFS accessed through the Spark framework.

First, we evaluated the impact of CloudMdsQL query rewriting and optimization on execution time on a cluster using three data stores: relational (Derby), document (MongoDB), and graph (Sparksee). We showed the execution times for 3 different executions plans of 5 queries. We compared the execution times, with different join orders, intermediate data transfer, and subquery rewritings. We also explored the performance benefit of using bind join under different conditions. The results experiments show that the third QEP of each query using bind join is much better than the first two QEP, in terms of execution time, and we handle less data.

Second, we evaluated our MFR approach in a Grid5000 cluster with three data stores : PostgreSQL, MongoDB and HDFS. We validated our approach using 3 different queries, by executing each one with 3 different HDFS configurations to assess scalability. We compared the performance between the costs of the QEPs without bind join and the QEPs with bind join. The results show that the benefit of the bind join optimization is higher in configurations with higher number of nodes (16 nodes) in terms of execution time. In addition, the amount of processed data is reduced during the execution of the MFR sequence by reordering MFR operators according to the determined rules.

Overall, our performance evaluation illustrates the CloudMdsQL query engine's ability to optimize a query and choose the most efficient execution strategy.

Chapter 7

Conclusion

In this thesis, we addressed the problem of query processing with multiple cloud data stores, where the data stores have different models, languages and APIs. This thesis has been prepared in the context of the CoherentPaaS European project [1] and, in particular, the CloudMdsQL multistore system. In this context, a major need is the integration of relational data and unstructured big data, typically stored in HDFS and accessed through a data processing framework such as Spark. In this thesis, we proposed an extension of CloudMdsQL to take full advantage of the functionality of the underlying data processing frameworks by allowing the ad-hoc usage of user defined map/filter/reduce (MFR) operators in combination with traditional SQL statements. Our solution allows for optimization by enabling subquery rewriting so that bind join can be used and filter conditions can be pushed down and applied by the data processing framework as early as possible.

In the rest of the chapter, we summarize and discuss our contributions and propose some directions for future work.

7.1 Contributions

An overview of query processing in multistore systems

We reviewed the state-of-the-art in query processing in multistore systems. First, we introduced cloud data management solutions and technologies, including distributed file systems, NoSQL systems and data processing frameworks. Then, we introduced query processing in multidatabase systems, which be resued and adapted to multistore systems. To ease comparison, we divided multistore systems based on the level of coupling with the underlying data stores, i.e., (1) loosely-coupled, (2) tightly-coupled and (3) hybrid.

We surveyed and analyzed some representative multistore systems for each category: (1) BigIntegrator, Forward and QoX; (2) Polybase, HadoopDB and Estocada; (3) SparkSQL, BigDAWG and CloudMdsQL. We compared their functionality along several dimensions: objective, data model, query language, and supported data stores. We also compared their implementation techniques along special modules, e.g. dataflow engine (QoX), HDFS bridge (Polybase), island query processors (BigDAWG) and query plan-

ner (CloudMdsQL), schema management, and query processing. The comparisons reveal several trends: the ability to integrate relational data (stored in RDBMS) with other kinds of data stores; the growing importance of accessing HDFS within Hadoop and the fact that most systems provide a relational/ SQL-like abstraction.

The extension of CloudMdsQL with MFR notation

We extended the CloudMdsQL language to integrate data retrieved from different data stores, including unstructured (HDFS) data accessed through a data processing framework. This allows performing joins between relational and HDFS data.

We defined a simple notation (in CloudMdsQL) to specify in a declarative way the sequence of map/filter/reduce (MFR) operators. We exploit the full power of the frameworks , yet avoiding the use of SQL engines on top of them. Furthermore, we allow for optimization by enabling subquery rewriting so that bind join can be used and filter conditions can be pushed down and applied by the data processing framework as early as possible. The query operator execution sequence specified by the user may be reordered by taking into account the properties of MFR operators together with the properties of relational operators, yet allowing for optimization through the use of bind join and operator reordering. We illustrated our approach with a use case that reveals how the query engine dynamically rewrites the MFR subquery to perform bind join optimization.

Prototype

We developed the MFR extension as part of the CloudMdsQL query engine. The query engine includes a query planner that performs query analysis and optimization, and produces a query plan, and a lightweight runtime database processor atop each data store that is composed of three generic modules (i.e. same code library) - query execution controller, operator engine, and table storage - and one wrapper module that is specific to a data store.

The current implementation of the query engine uses a modified version of the open source Derby database to accept CloudMdsQL queries and transform the corresponding execution plan into Derby SQL operations. To extend the CloudMdsQL query engine with MFR, we developed an MFR planner to be used by the wrapper of the data processing framework (DPF). The MFR planner finds optimization opportunities and translates the resulting sequence of MFR operations to a sequence of DPF's API methods to be executed.

We validated the CloudMdsQL query engine with wrappers for four data stores: Sparksee, a graph data store with Python API; Derby, a relational data store accessed through its JDBC driver; MongoDB, a document data store with a Java API; and Apache Spark a data processing framework on top of HDFS, accessed by Apache Spark API.

Experimental validation

Based on the CloudMdsQL query engine prototype, we performed an experimental validation of multistore query processing in a cluster to evaluate the impact on performance of optimization. More specifically, we explored the performance benefit of using bind join, a very efficient technique, under different conditions. In our experimental validation, we focused on queries that can express an integration of data across several data stores, in particular, NoSQL (graph and document data stores), RDBMS and HDFS accessed through the Spark framework.

First, we evaluated the impact of query rewriting and optimization on execution time on a cluster using three data stores: relational (Derby), document (MongoDB), and graph (Sparksee). We showed the execution times for 3 different executions plans of 5 queries. We compared the execution times, with different join orders, intermediate data transfer, and subquery rewritings. We also explored the performance benefit of using bind join under different conditions. The results experiments show that the third QEP of each query using bind join is much better than the first two QEP, in terms of execution time, and we handle less data.

Second, we evaluated our MFR approach in a Grid5000 cluster with three data stores : PostgreSQL, MongoDB and HDFS. We validated our approach using 3 different queries, by executing each one with 3 different HDFS configurations to assess scalability. We compared the performance between the costs of the QEPs without bind join and the QEPs with bind join. The results show that the benefit of the bind join optimization is higher in configurations with higher number of nodes (16 nodes) in terms of execution time. In addition, the amount of processed data is reduced during the execution of the MFR sequence by reordering MFR operators according to the determined rules.

Overall, our performance evaluation illustrates the CloudMdsQL query engine's ability to optimize a query and choose the most efficient execution strategy.

7.2 Directions for Future Work

Research on multistore systems is fairly recent and there are many new issues. Based on our contributions, we can identify the following research directions.

Support of multistore views

Views have been widely used in multidatabases to provide distribution and heterogeneity transparency, hiding that data is stored in different DBMSs. Adding views in CloudMdsQL would make it easier for users to write queries over multiple data stores since the table expressions that are needed for mapping data to the CloudMdsQL model would be captured by view expressions.

This would require the definition of a view definition language for CloudMdsQL, through either LAV or GAV approaches, and adapting query rewriting to deal with views. However, since CloudMdsQL supports native functions, query processing using views

becomes more difficult. For instance, a CloudMdsQL query could integrate data from a multistore view MSV, perhaps defined through user-defined functions, and a data store DS also through a user-defined function. But DS could be part of MSV and a simple rewrite of the query would access DS twice.

One solution would be to perform query analysis to discover that DS is part of MSV and rewrite the query so that DS is accessed only once, but producing two tables (one for each native function), which are later combined. A more complex approach would be to identify the subset of the predicates in native functions that could be combined, e.g. by "ANDing" simple predicates, and thus produce a single table.

Support of materialized views

To speed up the processing of analytical queries, multistore views could be materialized, as in data warehouse. Some multistore systems support materialized views, yet in a simple way. For instance, Odyssey's opportunistic materialized views allow to cache and reuse query results. However, the cached data are not refreshed as a result of updates to the base data.

Supporting materialized multistore views would require to deal with the problem of view maintenance, i.e. keeping the materialized data consistent with the base data which may be updated. This problem is addressed in data warehouse using RDBMSs and Extract-Transform-Load (ETL) tools. ETLs interface with the data sources and extract the updated data in several ways, depending on the capabilities of the data sources, i.e. update notifications, incremental extract of the update data or only full extract of the data. In the context of multistore systems, we would need to extend the wrapper modules with the extract capability of ETLs. Then, the major problem is that non relational data stores typically do not support update notifications and that full extract may the common, expensive option, requiring to compare each new extract with the previous one to identify changes.

A solution to the problem of materialized view maintenance is to compute the view incrementally, using differential tables that capture the updated data. We could adapt this solution in the context of multistore systems, in the simple case of non recursive view definitions by adapting efficient algorithms like the popular counting algorithm [31].

Parallel processing of multistore queries

To make big data analytics successful, it is critical to be able to integrate data from multiple data stores. This can be done by extending an OLAP parallel engine (OPE) with the capabilities of CloudMdsQL. OPEs typically exploit intra-query (both inter-operation and intra-operation) parallelism to scale up and yield high performance.

Given an OPE with CloudMdsQL capabilities, the problem is to integrate (e.g. join) big data from one or more data stores in the OPE, in a way that exploits parallelism. However, although all the data stores we have used provide support for data partitioning and parallel processing, the CloudMdsQL engine does not exploit parallelism. This is

because we have a wrapper, operator engine and data store client at one server, so a big data query to that data store will produce big data that gets centralized at that server.

One promising approach is to introduce intra-query parallelism within the CloudMdsQL engine. This requires the ability of the wrappers to directly access the data stores' partitions (at data nodes, not master nodes). Depending on the data stores, it may be more or less difficult. For instance, HDFS makes it easy by providing direct access to data chunks. But RDBMSs typically force to access partitioned data through a central master node.

Dealing with data streams

Stream processing engines (SPEs) are becoming ubiquitous for real-time data processing and data analysis. Unlike with data stores, new data are generated continually in fixed order (e.g. by arrival time or by a timestamp appended to the data) and processed on-the-fly using continuous queries. However, it is also necessary to combine streaming data with data stored in multiple systems. Thus, an important direction of research, pioneered in BigDAWG, is to couple an SPE with a multistore system.

Then, the major problem is to support real-time data processing on both real-time and stored data. The approach used in BigDAWG is to provide a data stream island that is supported by a new SPE (S-Store [18]) that is tighlty-coupled with the other data stores. S-Store queries rely on well-known streaming primitives, including filter, join, aggregate, and window. This approach could be adapted to extend the CloudMdsQL multistore system with streaming data. This requires extending the CloudMdsQL data model with ordered data and streaming operators. To support real-time data processing on the stored data, we could exploit materialized views stored in memory and parallelism.

Benchmarking multistore systems

As multistore systems get mature, the need for benchnmarks will necessarily increase. A first step in this direction is the benchmarking of the CloudMdsQL system [39]. The CloudMdsQL benchmark uses the standard TPC H benchmark (`www.tpc.org/tpch`) with 8 datasets over 5 different data stores, each having different interfaces and data models: relational, key-value, document, graph databases, and an OLAP engine. The TPC H queries and test cases are divided in two groups that focus on the performance benefits thanks to bind joins and the support of native queries.

The CloudMdsQL benchmark focuses mostly on relational operators and their equivalents in non-SQL data stores. As a further work, it will need to be extended to evaluate the performance benefits in the context of data store specific operators, such as graph traversals or queries on nested documents. Then, it could be used to compare the performance of various multistore systems.

Bibliography

[1] Coherentpaas project. coherentpaas.eu.

[2] Graphbase. https://jena.apache.org/documentation/javadoc/jena/org/apache/jena/graph/impl/GraphBase.html.

[3] Grid5000. http://www.grid5000.fr.

[4] Infinitegraph. http://www.objectivity.com/products/infinitegraph/.

[5] Json schema and hyper-schema. http://json-schema.org.

[6] Sparksee. http://www.sparsity-technologies.com/.

[7] Titan. https://github.com/thinkaurelius/titan/wiki.

[8] Trinity. https://www.microsoft.com/en-us/research/project/trinity/.

[9] ABOUZEID, A., BAJDA-PAWLIKOWSKI, K., ABADI, D., RASIN, A., AND SILBERSCHATZ, A. Hadoopdb: An architectural hybrid of mapreduce and DBMS technologies for analytical workloads. *Proceedings of the VLDB Endowment (PVLDB) 2*, 1 (2009), 922–933.

[10] ARMBRUST, M., XIN, R., LIAN, C., HUAI, Y., LIU, D., BRADLEY, J., MENG, X., KAFTAN, T., FRANKLIN, M., GHODSI, A., AND ZAHARIA, M. Spark SQL: relational data processing in spark. In *ACM SIGMOD Int. Conf. on Management of Data* (2015), pp. 1383–1394.

[11] AVNUR, R., AND HELLERSTEIN, J. Eddies: Continuously adaptive query processing. In *ACM SIGMOD Int. Conf. on Management of Data* (2000), pp. 261–272.

[12] BINNIG, C., REHRMANN, R., FAERBER, F., AND RIEWE, R. Funsql: it is time to make SQL functional. In *Int. Conf. on Extending Database Technology (EDBT)* (2012), pp. 41–46.

[13] BONDIOMBOUY, C., KOLEV, B., LEVCHENKO, O., AND VALDURIEZ, P. Integrating big data and relational data with a functional sql-like query language. In *Int. Conf. on Database and Expert Systems Applications (DEXA)* (2015), pp. 170–185.

[14] BONDIOMBOUY, C., KOLEV, B., LEVCHENKO, O., AND VALDURIEZ, P. Multistore big data integration with cloudmdsql. *Trans. on Large-Scale Data- and Knowledge-Centered Systems (TLDKS) 28* (2016), 48–74.

[15] BONDIOMBOUY, C., AND VALDURIEZ, P. Query processing in multistore systems: an overview. *Int. Journal of Cloud Computing (IJCC) 5*, 4 (2016), 309–346.

[16] BRUGGEN, R. V. *Learning Neo4j*. Packt Publishing Limited, 2014.

[17] BUGIOTTI, F., BURSZTYN, D., D., A., ILEANA, I., AND MANOLESCU, I. Invisible glue: Scalable self-tuning multi-stores. In *Int. Conf. on Innovative Data Systems Research (CIDR)* (2015), p. 7.

[18] ÇETINTEMEL ET AL., U. S-store: A streaming newsql system for big velocity applications. *Proceedings of the VLDB Endowment (PVLDB) 7*, 13 (2014), 1633–1636.

[19] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W., WALLACH, D., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. Bigtable: A distributed storage system for structured data. *ACM Trans. on Computer Systems 26*, 2 (2008), 4:1–4:26.

[20] DANFORTH, S., AND VALDURIEZ, P. A FAD for data intensive applications. *IEEE Trans. Knowl. Data Eng. 4*, 1 (1992), 34–51.

[21] DEAN, J., AND GHEMAWAT, S. Mapreduce: Simplified data processing on large clusters. In *Symposium on Operating Systems Design and Implementation (OSDI)* (2004), pp. 137–150.

[22] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: amazon's highly available key-value store. In *ACM Symposium on Operating Systems Principles (SOSP)* (2007), pp. 205–220.

[23] DEWITT, D., HALVERSON, A., NEHME, R., SHANKAR, S., AGUILAR-SABORIT, J., AVANES, A., FLASZA, M., AND GRAMLING, J. Split query processing in polybase. In *ACM SIGMOD Int. Conf. on Management of Data* (2013), pp. 1255–1266.

[24] DOAN, A., HALEVY, A. Y., AND IVES, Z. G. *Principles of Data Integration*. Morgan Kaufmann, 2012.

[25] DUGGAN, J., ELMORE, A., STONEBRAKER, M., BALAZINSKA, M., HOWE, B., KEPNER, J., MADDEN, S., MAIER, D., MATTSON, T., AND ZDONIK, S. The bigdawg polystore system. *SIGMOD Record 44*, 2 (2015), 11–16.

[26] EWEN, S., SCHELTER, S., TZOUMAS, K., WARNEKE, D., AND MARKL, V. Iterative parallel data processing with stratosphere: an inside look. In *ACM SIGMOD Int. Conf. on Management of Data* (2013), pp. 1053–1056.

[27] FU, Y., ONG, K. W., PAPAKONSTANTINOU, Y., AND ZAMORA, E. FORWARD: data-centric uis using declarative templates that efficiently wrap third-party javascript components. *Proceedings of the VLDB Endowment (PVLDB) 7*, 13 (2014), 1649–1652.

[28] GANKIDI, V., TELETIA, N., PATEL, J., HALVERSON, A., AND DEWITT, D. Indexing HDFS data in PDW: splitting the data from the index. *Proceedings of the VLDB Endowment (PVLDB) 7*, 13 (2014), 1520–1528.

[29] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S. The google file system. In *ACM Symposium on Operating Systems Principles (SOSP)* (2003), pp. 29–43.

[30] GODFREY, P., GRYZ, J., HOPPE, A., MA, W., AND ZUZARTE, C. Query rewrites with views for XML in DB2. In *Int. Conf. on Data Engineering (ICDE)* (2009), pp. 1339–1350.

[31] GUPTA, A., MUMICK, I. S., AND SUBRAHMANIAN, V. S. Maintaining views incrementally. In *ACM SIGMOD Int. Conf. on Management of Data* (1993), pp. 157–166.

[32] HAAS, L., KOSSMANN, D., WIMMERS, E., AND YANG, J. Optimizing queries across diverse data sources. In *Int. Conf. on Very Large Databases (VLDB)* (1997), pp. 276–285.

[33] HAASE, P., MATHÄSS, T., AND ZILLER, M. An evaluation of approaches to federated query processing over linked data. In *Int. Conf. on Semantic Systems, (I-SEMANTICS)* (2010).

[34] HACIGÜMÜS, H., SANKARANARAYANAN, J., TATEMURA, J., LEFEVRE, J., AND POLYZOTIS, N. Odyssey: A multi-store system for evolutionary analytics. *Proceedings of the VLDB Endowment (PVLDB) 6*, 11 (2013), 1180–1181.

[35] HART, B. E., VALDURIEZ, P., AND DANFORTH, S. Parallelizing FAD using compile-time analysis techniques. *IEEE Data Engineering Bulletin 12*, 1 (1989), 9–15.

[36] HUPFELD, F., CORTES, T., KOLBECK, B., STENDER, J., FOCHT, E., HESS, M., MALO, J., MARTÍ, J., AND CESARIO, E. The xtreemfs architecture - a case for

object-based file systems in grids. *Concurrency and Computation: Practice and Experience 20*, 17 (2008), 2049–2060.

[37] KOLEV, B., BONDIOMBOUY, C., LEVCHENKO, O., VALDURIEZ, P., JIMÉNEZ-PERIS, R., PAU, R., AND PEREIRA, J. O. Design and implementation of the cloudmdsql multistore system. In *Int. Conf. on Cloud Computing and Services Science (CLOSER)* (2016), pp. 352–359.

[38] KOLEV, B., BONDIOMBOUY, C., VALDURIEZ, P., JIMÉNEZ-PERIS, R., PAU, R., AND PEREIRA, J. The cloudmdsql multistore system. In *ACM SIGMOD Int. Conf. on Management of Data* (2016), pp. 2113–2116.

[39] KOLEV, B., PAU, R., LEVCHENKO, O., VALDURIEZ, P., JIMÉNEZ-PERIS, R., AND PEREIRA, J. O. Benchmarking polystores: The cloudmdsql experience. In *IEEE Int. Conf. on Big Data* (2016), pp. 2574–2579.

[40] KOLEV, B., VALDURIEZ, P., BONDIOMBOUY, C., JIMÉNEZ-PERIS, R., PAU, R., AND PEREIRA, J. Cloudmdsql: querying heterogeneous cloud data stores with a common language. *Distributed and Parallel Databases 34*, 4 (2016), 463–503.

[41] LEFEVRE, J., SANKARANARAYANAN, J., HACIGÜMÜS, H., TATEMURA, J., POLYZOTIS, N., AND CAREY, J. MISO: souping up big data query processing with a multistore system. In *ACM SIGMOD Int. Conf. on Management of Data* (2014), pp. 1591–1602.

[42] LENZERINI, M. Data integration: A theoretical perspective. In *ACM SIGMOD/PODS (Principles of Database Systems) Conf.* (2002), pp. 233–246.

[43] LIU, Z. H., CHANG, H. J., AND STHANIKAM, B. Efficient support of xquery update facility in XML enabled RDBMS. In *Int. Conf. on Data Engineering (ICDE)* (2012), pp. 1394–1404.

[44] MEIJER, E., BECKMAN, B., AND BIERMAN, G. M. LINQ: reconciling object, relations and XML in the .net framework. In *ACM SIGMOD Int. Conf. on Management of Data* (2006), p. 706.

[45] ONG, K. W., PAPAKONSTANTINOU, Y., AND VERNOUX, R. The SQL++ semi-structured data model and query language: A capabilities survey of sql-on-hadoop, nosql and newsql databases. *ACM Computing Research Repository (CoRR) abs/1405.3631* (2014).

[46] ÖZSU, M. T., AND VALDURIEZ, P. *Principles of Distributed Database Systems, Third Edition*. Springer, 2011.

[47] PACITTI, E., AKBARINIA, R., AND DICK, M. E. *P2P Techniques for Decentralized Applications*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2012.

[48] PLUGGE, E., HAWKINS, T., AND MEMBREY, P. *The Definitive Guide to MongoDB:The NoSQL Database for Cloud and Desktop Computing*. Apress, 2010.

[49] RAMAKRISHNAN, R. Data management in the cloud. In *Int. Conf. on Data Engineering (ICDE)* (2009), p. 5.

[50] SHAO, B., WANG, H., AND LI, Y. Trinity: a distributed graph engine on a memory cloud. In *ACM SIGMOD Int. Conf. on Management of Data* (2013), pp. 505–516.

[51] SIMITSIS, A., WILKINSON, K., CASTELLANOS, M., AND DAYAL, U. Qox-driven ETL design: reducing the cost of ETL consulting engagements. In *ACM SIGMOD Int. Conf. on Management of Data* (2009), pp. 953–960.

[52] SIMITSIS, A., WILKINSON, K., CASTELLANOS, M., AND DAYAL, U. Optimizing analytic data flows for multiple execution engines. In *ACM SIGMOD Int. Conf. on Management of Data* (2012), pp. 829–840.

[53] STONEBRAKER, M. Operating system support for database management. *Communications of the ACM 24*, 7 (1981), 412–418.

[54] STONEBRAKER, M., ABADI, D., DEWITT, D., MADDEN, S., PAULSON, E., PAVLO, A., AND RASIN, A. Mapreduce and parallel dbmss: friends or foes? *Communications of the ACM 53*, 1 (2010), 64–71.

[55] STONEBRAKER, M., WONG, E., KREPS, P., AND HELD, G. The design and implementation of ingres. *ACM Trans. on Database Systems 1*, 3 (1976), 198–222.

[56] TOMASIC, A., RASCHID, L., AND VALDURIEZ, P. Scaling access to heterogeneous data sources with DISCO. *IEEE Trans. Knowl. Data Eng. 10*, 5 (1998), 808–823.

[57] VALDURIEZ, P., AND DANFORTH, S. Functional SOL (fsol), an SQL upward-compatible database programming language. *Information Sciences 62*, 3 (1992), 183–203.

[58] WEIL, S., BRANDT, S., MILLER, E., LONG, D., AND MALTZAHN, C. Ceph: A scalable, high-performance distributed file system. In *Symposium on Operating Systems Design and Implementation (OSDI)* (2006), pp. 307–320.

[59] WHITE, T. *Hadoop - The Definitive Guide: Storage and Analysis at Internet Scale*. O'Reilly, 2012.

[60] WIEDERHOLD, G. Mediators in the architecture of future information systems. *IEEE Computer 25*, 3 (1992), 38–49.

[61] WYSS, C. M., AND ROBERTSON, E. L. Relational languages for metadata integration. *ACM Trans. on Database Systems 30*, 2 (2005), 624–660.

[62] YUANYUAN, T., ZOU, T., OZCAN, F., GONSCALVES, R., AND PIRAHESH, H. Joins for hybrid warehouses: Exploiting massive parallelism and enterprise data warehouses. In *Int. Conf. on Extending Database Technology (EDBT)* (2015), pp. 373–384.

[63] ZAHARIA, M., CHOWDHURY, M., FRANKLIN, M., SHENKER, S., AND STOICA, I. Spark: Cluster computing with working sets. In *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)* (2010), pp. 10–10.

[64] ZHU, M., AND RISCH, T. Querying combined cloud-based and relational databases. In *Int. Conf. on Cloud and Service Computing (CSC)* (2011), pp. 330–335.

[65] ZHU, Q., AND LARSON, P. A query sampling method of estimating local cost parameters in a multidatabase system. In *Int. Conf. on Data Engineering (ICDE)* (1994), pp. 144–153.

[66] ZHU, Q., AND LARSON, P. Global query processing and optimization in the cords multidatabase system. In *Int. Conf. on Parallel and Distributed Computing Systems* (1996), pp. 640–647.

[67] ZHU, Q., SUN, Y., AND MOTHERAMGARI, S. Developing cost models with qualitative variables for dynamic multidatabase environments. In *Int. Conf. on Data Engineering (ICDE)* (2000), pp. 413–424.