



Supporting multiple data stores based applications in cloud environments

Rami Sellami

► To cite this version:

Rami Sellami. Supporting multiple data stores based applications in cloud environments. Modeling and Simulation. Université Paris Saclay (COmUE), 2016. English. NNT : 2016SACLL002 . tel-01280236

HAL Id: tel-01280236

<https://theses.hal.science/tel-01280236>

Submitted on 29 Feb 2016

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

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

NNT : 2016SACLL002

THESE DE DOCTORAT
DE L'UNIVERSITE PARIS-SACLAY,
préparée à TELECOM SudParis

ÉCOLE DOCTORALE N°580
Sciences et technologies de l'information et de la communication
Spécialité de doctorat : Informatique

Par

M Rami Sellami

SUPPORTING MULTIPLE DATA STORES BASED APPLICATIONS IN CLOUD
ENVIRONMENTS

Thèse présentée et soutenue à Evry, le 05/02/2016 :

Composition du Jury :

Mme Daniella GRIGORI	Professeur, Université Paris Dauphine	Président du jury
M. Bernd AMANN	Professeur, Université Pierre-et-Marie-Curie	Rapporteur
Mme. Genoveva VARGAS-SOLAR	Chargé de recherche (CNRS), Laboratoire d'Informatique de Grenoble	Rapporteur
M. Ladjel BELLATRECHE	Professeur, ISAE-ENSMA	Examineur
M. Sami BHIRI	Maître de conférences, Université de Monastir	Examineur
M. Talel ABDESSALEM	Professeur, Telecom ParisTech	Examineur
M. Bruno DEFUDE	Professeur, Telecom SudParis	Directeur de thèse



Acknowledgements

I thank God for making all things possible for me and giving me strength to go. A thesis journey is not always easy, but with the supervision, support and encouragement of many people, I have never regretted that I started it.

I would like to thank all members of the jury. I thank Professor Bernd Amann and Professor Genoveva Vargas-Solar for accepting being my thesis reviewers and for their attention and thoughtful comments. I also thank Professor Ladjene Belatreche, Professor Sami Bhiri, Professor Daniella Grigori and Professor Talel Abdesslem for accepting being my thesis examiners.

I would like to express my sincere appreciation and gratitude to my supervisor Bruno Defude. His valuable advice, enthusiasm and continuous support during this thesis allowed me to acquire new understandings and extend my experiences. He was not only an advisor but also a good listener and a friend who showed me just what I was able to achieve even when I did not see it myself. Thank you for taking me on this journey and for your guidance, it has been a true pleasure and I hope that we can continue our collaboration.

I owe my deepest gratitude and warmest affection to the members of the computer science department of Telecom SudParis. They contributed to making these last years a wonderful experience. I would like to especially thank Brigitte Houassine for her kind help and assistance. Here it goes an incomplete list: Emna, Nour, Mourad, Souha, Nabila, Nathan, Fethi, Mehdi, Walid, Samir, Elisabeth, Olivier, François, Amel, Alda, Amel, and Mohamed. Thanks for opening your hearts and letting me be part of your lives.

I am forever thankful to my family: my parents, Zouhir and Amel, my parents in law, Abdelmottaleb and Alia, my brothers, Mohamed and Ramzi, my sister in law Manel and my brother in law, Ilyess who were always there for me with encouraging words whenever I started doubting myself. Your encouragement made me go forward and made me want to succeed. I express my deepest gratitude to my loving wife Sirine. Her love, encouragement, understanding and support helped me to get through many difficult times. Thank you so much for having faith in me! I cannot forget, my beautiful son Aziz. You supported me without even you know it. I love you so much.

For my parents
For my parents in law
For my wife and my son
For my brothers
For my sister and my brother in law
For my friends

Résumé

Avec l'avènement du *Cloud Computing* et des *Big Data*, de nouveaux systèmes de gestion de bases de données sont apparus, connus en général sous le vocable systèmes NoSQL. Par rapport aux systèmes relationnels, ces systèmes se distinguent par leur absence de schéma, une spécialisation pour des types de données particuliers (documents, graphes, clé/valeur et colonne) et l'absence de langages de requêtes déclaratifs. L'offre est assez pléthorique et il n'y a pas de standard aujourd'hui comme peut l'être SQL pour les systèmes relationnels. De nombreuses applications peuvent avoir besoin de manipuler en même temps des données stockées dans des systèmes relationnels et dans des systèmes NoSQL. Le programmeur doit alors gérer deux (au moins) modèles de données différents et deux (au moins) langages de requêtes différents pour pouvoir écrire son application. De plus, il doit gérer explicitement tout son cycle de vie. En effet, il a à (1) coder son application, (2) découvrir les services de base de données déployés dans chaque environnement *Cloud* et choisir son environnement de déploiement, (3) déployer son application, (4) exécuter des requêtes multi-sources en les programmant explicitement dans son application, et enfin le cas échéant (5) migrer son application d'un environnement *Cloud* à un autre. Toutes ces tâches sont lourdes et fastidieuses et le programmeur risque d'être perdu dans ce haut niveau d'hétérogénéité.

Afin de pallier ces problèmes et aider le programmeur tout au long du cycle de vie des applications utilisant des bases de données multiples, nous proposons un ensemble cohérent de modèles, d'algorithmes et d'outils. En effet, notre travail dans ce manuscrit de thèse se présente sous forme de quatre contributions. Tout d'abord, nous proposons un modèle de données unifié pour couvrir l'hétérogénéité entre les modèles de données relationnelles et NoSQL. Ce modèle de données est enrichi avec un ensemble de règles de raffinement. En se basant sur ce modèle, nous avons défini notre algèbre de requêtes. Ensuite, nous proposons une interface de programmation appelée ODBAPI basée sur notre modèle de données unifié, qui nous permet de manipuler de manière uniforme n'importe quelle source de données qu'elle soit relationnelle ou NoSQL. ODBAPI permet de programmer des applications indépendamment des bases de données utilisées et d'exprimer des requêtes simples et complexes multi-sources. Puis, nous définissons la notion de bases de données virtuelles qui interviennent comme des médiateurs et interagissent avec les bases de données intégrées via ODBAPI. Ce dernier joue alors le rôle d'adaptateur. Les bases de données virtuelles assurent l'exécution des requêtes d'une façon optimale grâce à un modèle de coût et un algorithme de génération de plan d'exécution optimal que nous définis. Enfin, nous proposons une approche automatique de découverte de bases de données dans des environnements *Cloud*. En effet, les programmeurs peuvent décrire leurs exigences en termes de bases de données dans des manifestes, et grâce à notre algorithme d'appariement, nous sélectionnons l'environnement le plus adéquat à notre application pour la

déployer. Ainsi, nous déployons l'application en utilisant une API générique de déploiement appelée COAPS. Nous avons étendue cette dernière pour pouvoir déployer les applications utilisant plusieurs sources de données. Un prototype de la solution proposée a été développé et mis en œuvre dans des cas d'utilisation du projet OpenPaaS. Nous avons également effectué diverses expériences pour tester l'efficacité et la précision de nos contributions.

Mots-clés: Nuages des données, Données volumineuses, Bases de données NoSQL, Bases de données relationnelles, Optimisation et évaluation des requêtes de jointure, Persistance polyglotte, Découverte à base de manifestes

Abstract

The production of huge amount of data and the emergence of Cloud computing have introduced new requirements for data management. Many applications need to interact with several heterogeneous data stores depending on the type of data they have to manage: traditional data types, documents, graph data from social networks, simple key-value data, etc. Interacting with heterogeneous data models via different APIs, and multiple data stores based applications imposes challenging tasks to their developers. Indeed, programmers have to be familiar with different APIs. In addition, the execution of complex queries over heterogeneous data models cannot, currently, be achieved in a declarative way as it is used to be with mono-data store application, and therefore requires extra implementation efforts. Moreover, developers need to master and deal with the complex processes of Cloud discovery, and application deployment and execution.

In this manuscript, we propose an integrated set of models, algorithms and tools aiming at alleviating developers task for developing, deploying and migrating multiple data stores applications in cloud environments. Our approach focuses mainly on three points. First, we provide a unified data model used by applications developers to interact with heterogeneous relational and NoSQL data stores. This model is enriched by a set of refinement rules. Based on that, we define our query algebra. Developers express queries using OPEN-PaaS-DataBase API (ODBAPI), a unique REST API allowing programmers to write their applications code independently of the target data stores. Second, we propose virtual data stores, which act as a mediator and interact with integrated data stores wrapped by ODBAPI. This run-time component supports the execution of single and complex queries over heterogeneous data stores. It implements a cost model to optimally execute queries and a dynamic programming based algorithm to generate an optimal query execution plan. Finally, we present a declarative approach that enables to lighten the burden of the tedious and non-standard tasks of (1) discovering relevant Cloud environments and (2) deploying applications on them while letting developers to simply focus on specifying their storage and computing requirements. A prototype of the proposed solution has been developed and implemented use cases from the OpenPaaS project. We also performed different experiments to test the efficiency and accuracy of our proposals.

Keywords: Cloud Computing, Big Data, REST-based API, NoSQL data stores, relational data stores, join queries optimization and evaluation, polyglot persistence, manifest based matching

List of Publications

- [Publication 1] Rami Sellami and Bruno Defude. Using multiple data stores in the cloud: Challenges and solutions. In The 6th International Conference on Data Management in Cloud, Grid and P2P Systems, Globe'13, Prague, Czech Republic, August 28-29.
- [Publication 2] Rami Sellami, Sami Bhiri, and Bruno Defude. ODBAPI: a unified REST API for relational and NoSQL data stores. In The IEEE 3rd International Congress on Big Data (BigData'14), Anchorage, Alaska, USA, June 27 - July 2.
- [Publication 3] Rami Sellami, Michel Vedrigne, Sami Bhiri, and Bruno Defude. Automating resources discovery for multiple data stores cloud applications. In Cloud Computing and Services Science - Fifth International Conference, CLOSER'15, Lisbon, Portugal, May 20-22.
- [Publication 4] Rami Sellami, Sami Bhiri, and Bruno Defude. Supporting multi-data stores applications in cloud environments. IEEE Transactions on Services Computing, Special Issue on Big Data Analytics, Infrastructure, and Applications, Vol.PP, No.99, ISSN 1939-1374, June 2015.

Contents

1	General Introduction	1
1.1	Research Context	1
1.2	Research Problem	3
1.3	Thesis Principles And Objectives	5
1.4	Thesis Contributions	5
1.5	Thesis Outline	6
2	State Of The Art	
2.1	Introduction	9
2.2	Background	10
2.2.1	Cloud Computing	10
2.2.2	Big Data	11
2.2.3	Using Multiple Data Stores In Cloud Environment	13
2.2.3.1	Relational Data Stores	13
2.2.3.2	NoSQL Data Stores	14
2.2.3.3	An Illustrative Example	16
2.2.3.4	NoSQL vs RDBMS	18
2.2.3.5	Polyglot Persistence	19
2.2.3.6	Data Stores Integration	20
2.3	Related Work	22
2.3.1	Unifying Data Models	22
2.3.2	Easy Access To Multiple Data Stores	25
2.3.3	Transparent Access To Integrated Data Stores	28
2.3.4	Capturing Data Requirements And Cloud Data Store Capabilities	32
2.4	Conclusion	37
3	Unifying The Access To Relational And NoSQL Data Stores In Cloud Environments	
3.1	Introduction	40
3.2	Overview Of Our Approach	40
3.3	Unifying Data Models	42
3.3.1	Informal Description Of The Unified Data Model	42

3.3.2	Formal Description Of The Unified Data Model	44
3.3.3	Global Schema	45
3.3.4	Query Algebra	47
3.3.4.1	Projection Operation	47
3.3.4.2	Selection Operation	48
3.3.4.3	Cartesian Product	48
3.3.4.4	Join Operation	50
3.3.4.5	Union Operation	51
3.3.4.6	Intersection Operation	51
3.3.4.7	Nest and Unnest Operations	52
3.3.4.8	Algebraic Tree	54
3.4	ODBAPI: a unified REST API for relational and NoSQL data stores	55
3.4.1	Use Cases And Motivation	56
3.4.1.1	First scenario: Application migration from one data store to another	56
3.4.1.2	Second scenario: Polyglot persistence	57
3.4.2	ODBAPI: OpenPaaS DataBase API	58
3.4.2.1	Panorama Of ODBAPI	58
3.4.2.2	Operations of ODBAPI	59
3.4.2.3	Examples Of Queries	62
3.5	Conclusion	66
4	Virtual Data Stores For Query Evaluation and Optimization	
4.1	Introduction	67
4.2	Use Cases And Motivation	68
4.3	Query Evaluation	69
4.3.1	Query Evaluation Principles	69
4.3.2	Query Execution Plan	70
4.3.3	Queries Parsing and Algebraic Optimization	72
4.3.4	Queries Annotation	72
4.4	Query Optimisation	74
4.4.1	Catalog	76
4.4.2	Cost Model	78
4.4.3	Optimal Execution Plan Generation	80
4.5	Conclusion	86

5 Automating Resources Discovery And Deployment For Multiple Data Stores Applications In Cloud Environment

5.1	Introduction	87
5.2	Use Cases And Motivation	88
5.3	Automatic Discovery Of Cloud Data Stores	88
5.3.1	Automatic Discovery Principles	89
5.3.2	Abstract Application Manifest	89
5.3.3	Offer Manifest	91
5.3.4	Matching Algorithm	91
5.4	Automatic Deployment Of Multiple Data Stores Based Applications	94
5.4.1	An Extension Of The Compatible One Application and Plat- form Service API	95
5.4.2	Deployment Manifest	97
5.5	XML-Based Manifest Examples	97
5.6	Conclusion	101

6 Evaluation & Validation

6.1	Introduction	103
6.2	Proofs Of Concept	103
6.2.1	The OpenPaaS Project	104
6.2.2	Current State Of ODBAPI	104
6.2.3	Selecting Multiple Data Stores And Deploying ODBAPI Clients	108
6.3	ODBAPI Overhead Evaluation	109
6.4	Ease Of ODBAPI Use	110
6.5	Complex Queries Optimization	113
6.6	Conclusion	119

7 Conclusion & Future Works

7.1	Contributions	123
7.2	Perspectives	125

Bibliography

List of Figures

1.1	Three possible scenarios of applications interactions with data stores in Cloud environments	2
2.1	Cloud Computing Conceptual Reference Model defined by NIST	10
2.2	IBM characterizes Big Data by its volume, velocity, variety - or simply V3 [1]	12
2.3	Examples of illustrating the same data model with different types of data stores	17
2.4	Example of polyglot persistence implemented in an e-commerce platform [2]	20
3.1	An end-to-end overview of our solution	40
3.2	Unified data model	42
3.3	Exemples of <i>EntitySet</i> concepts of type <i>MongoDB</i> , <i>Riak</i> , and <i>MySQL</i> from right to left	43
3.4	An application interacting with three data stores in a Cloud environment	44
3.5	The corresponding data model to the <i>dblp</i> , <i>person</i> , and <i>conferenceRanking</i> entity sets	46
3.6	The graphic notation of the projection operation	48
3.7	The graphic notation of the selection operation	48
3.8	The graphic notation of the Cartesian product operation	49
3.9	The graphic notation of the join operation	50
3.10	The graphic notation of the union operation	51
3.11	The graphic notation of the intersection operation	52
3.12	Exeple of an algebraic tree	55
3.13	Application migration from one cloud environment to another scenario	56
3.14	Using multiple data stores in Cloud environment	57
3.15	An overview of ODBAPI	59
3.16	ODBAPI operations	60
4.1	Mediation architecture	69
4.2	Examples of algebraic trees	72
4.3	Annotated algebraic tree according the Scenario 1	73
4.4	Annotated algebraic tree according the Scenario 2	73
4.5	Execution plan for the Scenario 2	75
4.6	Example of computing an operation cost	81
4.7	The initialization step	84
4.8	The first iteration	85
4.9	The second iteration	85
5.1	Zoom-in on the discovery component	89
5.2	Abstract application manifest model	90

5.3	Offer manifest model	92
5.4	Generating the offer manifest	94
5.5	COAPS API overview [3]	96
5.6	An application deployment scenario [3]	96
5.7	Deployment manifest model	97
5.8	The abstract graph	100
6.1	OpenPaaS overview	104
6.2	Package diagram of ODBAPI	106
6.3	Screenshot of all databases overview	107
6.4	Screenshot of the interface allowing to import the abstract applica- tion manifest and generate the deployment manifest	108
6.5	Example of a join queries	114
6.6	Total cost evaluation according to the scenarios 1 to 4 and the joins number- Case of linear join queries (Logarithmic Scale)	117
6.7	Total cost evaluation according to the scenarios 1 to 4 and the joins number- Case of join queries as a star (Logarithmic Scale)	118
6.8	Example of an execution plan - Case of a linear join query	119
6.9	Total cost evaluation according to the scenarios 1 to 4 and the joins number- Case of linear join queries in a big data context (Logarith- mic Scale)	120
6.10	Total cost evaluation according to the scenarios 1 to 4 and the joins number- Case of join queries as a star in a big data context (Loga- rithmic Scale)	120

List of Tables

2.1	Synthesis of the related works unifying data models	26
2.2	Synthesis of the related works ensuring easy access to multiple data stores	29
2.3	Synthesis of the related works ensuring transparent access to integrated data stores	32
2.4	Synthesis of the related works capturing Data Requirements And Cloud Data Store Capabilities	36
3.1	Comparison chart of concepts used in different data stores	43
3.2	Sample of three <i>Entities</i> from the person <i>EntitySet</i>	47
3.3	Sample of two <i>Entities</i> from the dblp <i>EntitySet</i>	47
3.4	Result of the projection of the person <i>EntitySet</i> using the attributes personId and personName	48
3.5	Result of the selection of the person <i>EntitySet</i> taking into account the predicate <i>personId</i> = 1	48
3.6	Result of the Cartesian product of the dblp and person <i>EntitySets</i>	49
3.7	Result of the join between the dblp and person <i>EntitySets</i>	50
3.8	Sample of two <i>Entities</i> from the P1 <i>EntitySet</i>	51
3.9	Sample of two <i>Entities</i> from the P2 <i>EntitySet</i>	52
3.10	Sample of two <i>Entities</i> from the P1 <i>EntitySet</i>	52
3.11	Sample of two <i>Entities</i> from the P2 <i>EntitySet</i>	53
3.12	Sample of complex attributes from the dblp <i>EntitySet</i>	53
3.13	Result of the unnest operation	53
3.14	Sample of the dblp <i>EntitySet</i>	54
3.15	Result of the nest operation	54
4.1	Specification of our catalog's parameters	78
5.1	Summary of the computed distances between the offer manifest and the abstract application manifest	100
6.1	Lines number in each driver in ODBAPI	106
6.2	Response time of the operation create entities with ODBAPI and JDBC	109
6.3	Response time of the operation delete entities with ODBAPI and JDBC	110
6.4	Response time of the operation retrieve all entities with ODBAPI and JDBC	110
6.5	Evaluation of ODBAPI in terms of the implementation time and source code lines number	111
6.6	Results obtained for the case of linear queries	117
6.7	Results obtained for the case of queries as star	117
6.8	Gain between scenarios	118

6.9	Results obtained for the case of linear queries - Big data context .	119
6.10	Results obtained for the case of queries as star - Big data context .	120
6.11	Gain between scenarios - Big data context	121

General Introduction

Contents

1.1	Research Context	1
1.2	Research Problem	3
1.3	Thesis Principles And Objectives	5
1.4	Thesis Contributions	5
1.5	Thesis Outline	6

1.1 Research Context

Cloud computing has recently emerged as a new computing paradigm enabling on-demand and scalable provision of resources, platforms and software as services. Cloud computing is often presented at three levels [4]: the Infrastructure as a Service (IaaS) giving access to abstracted view on the hardware, the Platform as a Service (PaaS) providing programming and execution environments to the developers, and the Software as a Service (SaaS) enabling software applications to be used by Cloud's end users.

Due to its elasticity property, Cloud computing provides interesting execution environments for several emerging applications such as big data management. According to the National Institute of Standards and Technology¹ (NIST), Big Data is *data which exceed the capacity or capability of current or conventional methods and systems*. It is mainly based on the 3-Vs model where the three Vs refer to volume, velocity and variety properties [5]. Volume means the processing of large amounts of information. Velocity signifies the increasing rate at which data flows. Finally, variety refers to the diversity of data sources. Several researchers have also proposed to add more Vs to this definition. Veracity is widely proposed and represents the quality of data (accuracy, freshness, consistency etc.). Against this background, the challenges of big data management result from the expansion of the 3Vs properties. In our work, we mainly focus on the variety property and more precisely on multiple data store based applications in the Cloud.

Different interaction forms between applications and data stores are possible in a Cloud environment. In order to define the application requirements in term of data storage, we analyze three possible scenarios (see Fig.1.1). Regardless the

¹<http://www.nist.gov/>

scenario, applications are deployed in the Cloud because they require scalability and elasticity. These properties will interact with other requirements and mainly with the requirements related to the consistency property. The first requirement is:

- R_0 : Ensuring scalability and elasticity.

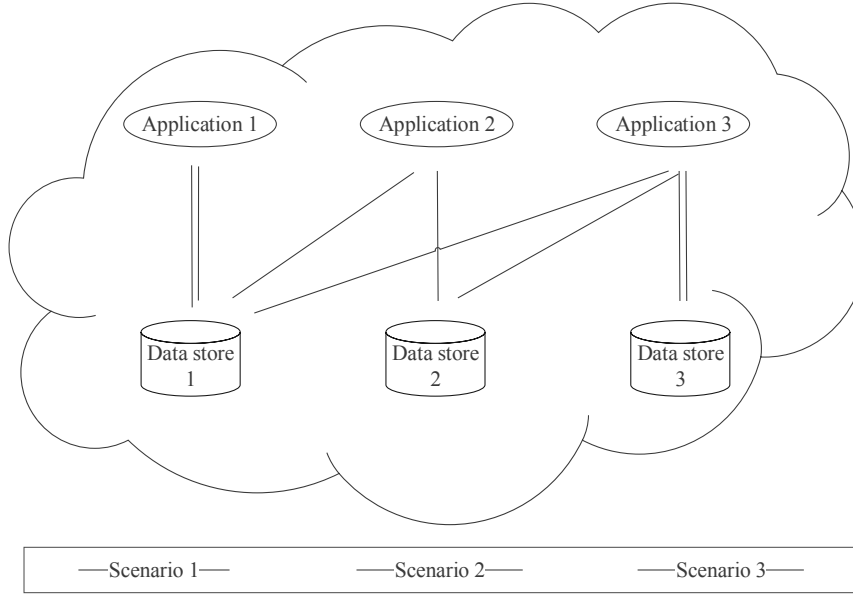


Figure 1.1: Three possible scenarios of applications interactions with data stores in Cloud environments

In the first scenario S_1 one or more applications use the same Cloud data store. Indeed, the Cloud provider supplies just one data store for applications. In Fig.1.1, this scenario is depicted by continuous lines. So, implementing an application in such case will be simple and convenient since we will find a unique API to manage the data store. However, it is difficult to a single data store in one Cloud to support the needs of all applications. Furthermore, some clients will be obliged to migrate their applications in another Cloud in order to meet their data requirements.

The second scenario S_2 corresponds to applications that use Cloud environments providing multiple data stores. Each client can choose its appropriate data store according to his/her application needs. This scenario is depicted in Fig.1.1 by the dashed lines. In addition, due to these multiple data stores, clients are not obliged to migrate to another Cloud. However, a single application can only use one data store to store all its data. So in some cases, an application may want to migrate its data from one data store (e.g. a relational data store) to another (e.g. a NoSQL data store). The problem here is to decide if a data store is convenient for an application and if not how to efficiently migrate data from one data store to another. Added to the requirement R_0 , we identify two new data requirements:

- R_1 : Choosing a data store based on data requirements. In this context, we present three sub-requirements: (R_{11}) defining application needs and requirements towards data, (R_{12}) defining data store capabilities, and (R_{13}) defining a matching between application needs and data stores capabilities.
- R_2 : Migrating application from a data store to another (i.e. migrating data and adapting the application source code to the new API).

In the third scenario S_3 , an application can use multiple data stores. For instance, an application can use a relational data store and a NoSQL data store at the same time or partition its data into multiple data stores. This scenario is illustrated by the dotted lines in Fig.1.1. Moreover, these data stores may belong to different Clouds (e.g. a public and private Clouds). This ensures a high level of efficiency and modularity. This case corresponds to what is popularly referred to as the polyglot persistence [2]. Nevertheless, it represents some limits and difficulties. Linking an application with multiple data stores is very complex due to the different APIs, data models, query languages or consistency models. If the application needs to query data coming from different data sources (e.g. joining data, aggregating data, etc.), it can not do it declaratively unless some kinds of mediation have been done before. Finally, the different data stores may use different transaction and consistency models (e.g. classical ACID and eventual consistency). It is not easy for programmers to understand these models and to properly code their applications to ensure desired properties. For this scenario, three new requirements are added:

- R_3 : Easy access to multiple data stores.
- R_4 : Transparent access to integrated data stores.
- R_5 : Easy programming of consistency guaranties with different data stores.

1.2 Research Problem

Spurred by the Cloud Computing popularity, researches are increasingly abundant and are focusing on various axes in this area. In this context, data management in the Cloud is an inherent research topic that recently received particular attention. A plethora of modern applications and researches, such as Bigtable [6], PNUTS [7], Dynamo [8], take into account the data management in the Cloud (especially scenarios S_1 and S_2 presented in Section 1.1). Nevertheless, they are not sufficient to address the requirements R_4 , R_5 , and R_6 of the scenario S_3 which is the case of multiple data stores based applications in Cloud environment. This lack of solutions implies that developers need to address these requirements by themselves and so they have to be aware of many technical details. In order to satisfy the different storage requirements, Cloud applications usually need to access and interact with different relational and NoSQL data stores having heterogeneous APIs, query languages, data models, consistency models, etc. The heterogeneity of the

data stores induces several problems when developing, deploying and migrating multiple data store applications. Developers are central to this high heterogeneity and they have to cope with the arising problems and related difficulties. Below, we list the main four problems that a developer may encounter that are tackled in this thesis:

- Pb₁* **Heavy workload on the applications developers:** Nowadays data stores have different and heterogeneous proprietary APIs. Developers of multiple data store based applications need to be familiar with all these APIs when coding their applications. As developers are central to this plethora of APIs, they should understand data models and APIs of each data store to properly code a polyglot persistence based application. In this context, the developers task is cumbersome and requires a high adaptation time to the APIs that can be judged as a wasted time.
- Pb₂* **No declarative way for executing complex queries:** Due to the heterogeneity of data models and the absence of schemas in NoSQL data stores, there is currently no declarative way to express and execute complex queries over several data stores. This is mainly due to the absence of a global schema unifying the access to relational and NoSQL data stores. In addition, NoSQL data stores are schemeless; That means developers have to cope themselves with the implementation of such queries in their application source code. Doing so, the execution of complex queries will be naive and non optimal. It is worth noting that a complex query may be a join, a union, etc.
- Pb₃* **Code adaptation:** When migrating applications from one Cloud environment to another, application developers have to re-adapt their application source code in order to interact with new data stores and migrate their data from the old data stores to the new ones. However, this is a tedious task since developers have potentially to learn and master new proprietary APIs. This may cause an important delay during the migration process and hamper the application lifecycle.
- Pb₄* **Tedious and non-standard discovery and deployment processes:** Once an application is developed or before is migrated, developers have to deploy it into a Cloud provider with respect to its requirements. To do so, developers must discover the most suitable Cloud environment providing the required data stores and data characteristics. However, this step is tedious and meticulous because it lacks of automation and developers have to do it manually. Indeed, developers must discover data stores services of each Cloud provider and try to match between their requirements and the Cloud providers capabilities. This would be less complex if the application interacts with a single data store.

Consistency is also an important issue in multiple data stores based applications. In fact, Cloud data stores in general implement different consistency models (e.g. strong consistency model for RDBMS and weak consistency models for

NoSQL DBMS). This implies that the consistency model at the application level is not really defined. We do not address this issue in this thesis, focusing only on querying. The interested reader may read [9] which proposes a middleware service addressing the problem of client-centric consistency on top of eventually consistent distributed data stores (Amazon S3 for example).

1.3 Thesis Principles And Objectives

In this thesis, we aim to propose solutions for supporting multiple based application developers in Cloud environments. Our objective is fourfold: (1) unifying relational and NoSQL data models, (2) easing the programming of polyglot persistence based applications, (3) automating multiple data stores discovery and applications deployment, and (4) evaluating and optimizing complex queries execution.

In our approach, we consider the following principles:

- **Cloud Computing and Big Data:** This thesis is developed within the field of Cloud Computing and Big Data. Indeed, we should take into account the different characteristics of each area namely the big volume of data and the variety of data stores regarding the Big Data area, and the dynamicity (i.e. application migration) and the abundance of data stores services with regard to the field of Cloud Computing.
- **NoSQL data stores support:** The different proposed solutions should take into account relational and NoSQL data stores.
- **Multiple data stores based applications:** The different solutions should support multiple data stores based applications since an application may meet its requirements in different data stores services in a Cloud environment.
- **Easy access to heterogeneous data stores:** In this thesis, solutions should take into account the importance of facilitating the access to relational data stores in order to easily express queries and correctly evaluate and optimize their execution.
- **Automation:** The different solutions should be automatic in order to relieve utmost developers task by removing the burden of managing the high heterogeneity of data stores in Cloud environments.

1.4 Thesis Contributions

In this thesis, we propose an integrated set of models, algorithms and tools aiming at alleviating developers' tasks for developing multiple data stores based applications, discovering Clouds data stores capabilities, executing complex queries, deploying and migrating applications in Cloud environments. It is worthy to say that we take into account relational and NoSQL data stores.

First, we define a unified data model used by applications developers to interact with different data stores. This model tackles the heterogeneity problem between data models and the schemes absence in NoSQL data stores. It is used to express CRUD and complex queries. In addition, we used the proposed unified data model to evaluate and optimize complex queries execution (tackling therefore problems Pb_1 and Pb_2).

Second, we propose OPEN-PaaS-DataBase API (ODBAPI) that developers may use to express and execute any type of queries. This API is a streamlined and a unified REST-based API for executing queries over relational and NoSQL data stores. The highlights of ODBAPI are twofold: (i) decoupling Cloud applications from data stores in order to facilitate the migration process, and (ii) easing the developers task by lightening the burden of managing different APIs (tackling therefore problems Pb_1 , Pb_2 , and Pb_3).

Third, we propose virtual data stores (VDS) to evaluate and optimize queries execution - especially complex ones- over different data stores. In order to support queries definition and execution over heterogeneous data models, we use the unified data model that we extend with correspondence rules. The VDS implements a dynamic programming based algorithm to generate optimal execution plans using a cost model that we have defined. Our solution is based on algebraic trees composed of data sources and algebraic operators and algebraic trees annotation (tackling therefore problem Pb_2).

Fourth, we present a declarative and an automatic approach for discovering appropriate Cloud environments and deploying applications on them. The proposed approach lets developers simply focus on specifying their storage and computing requirements. The key ingredients of this solution are (1) the use of manifests to expose applications requirements and data stores capabilities, and (2) the matching algorithm that elects the most appropriate Cloud provider to an application. For the deployment step, we propose to extend and use the COAPS API which is a deployment API proposed in our team and in the CompatibleOne project (tackling therefore problems Pb_1 and Pb_4).

A prototype of our approach has been developed and has been used to implement use cases from a french PaaS project called OPEN-PaaS. We will detail the different aspects of the realized implementations as a proof of concept. In addition, we will describe the realized scenarios and perform some experiments to evaluate our proposals.

1.5 Thesis Outline

The remainder of this manuscript is organized as follows. Chapter 2 contains two main parts. On the one hand, we provide a description of the different concepts needed for the understanding of our work. In fact, we give the definitions of Cloud Computing and Big Data areas. We also define the different variants that we aim to use in this work resumed in relational and NoSQL data stores, polyglot persistence, and data integration. On the other hand, we give an overview of the state of the

art related to unifying data models, easing access to multiple data stores, ensuring transparent access to integrated data stores as well as capturing data requirements and Cloud data stores capabilities. Therein, we will cite the different work in each part with a discussion to show the advantages that we would like to have in our solution and the drawbacks that we would like to escape.

Chapters 3, 4 and 5 are the core of our thesis which elaborate our end-to-end solution to support multiple data stores based applications developers in a Cloud environment.

In Chapter 3, we present first an overview of the different contributions proposed in this thesis and show how they are linked between each others. Afterwards, we introduce our first two contributions. First, we present our unified data model to unify the view over relational and NoSQL data stores. Herein, we present a formal and non formal representation of our unified data model. Based on that, we introduce our global schema that developers use to express simple and complex queries and a mediator layer used to evaluate and optimize queries execution. At the end of this part, we propose a query algebra that is used by the VDS to support complex queries execution.

Second, we introduce some use cases in order to motivate our second contribution which is ODBAPI. It is noteworthy that these motivating examples will be used also to motivate the other contributions presented in the two next chapters. Then, we give a overview of ODBAPI and we present the different operations ensured by our API. Finally, we present three examples illustrating how we unify the execution of queries and we express filtering and join queries.

In Chapter 4, we present our approach to evaluate and optimize query execution using the VDS. First, we introduce our query evaluation principles and we define our query execution plan. We finish this part by giving an example of join query evaluation. Second, we present our query optimization principles. In this part, we define the different parameters of our catalog and our cost model. Based on that, we introduce our algorithm to generate an optimal execution plan.

In Chapter 5, we present our last contribution which is the automation of resources discovery and multiple data stores based application deployment in Cloud environments. First, we introduce the principles of the discovery step and the structure of the abstract application manifest and the offer manifest. Based on that, we introduce our matching algorithm. Second, we present the principles of the deployment step and we show how we extend the COAPS API to support multiple data stores based application deployment. We finish this part by defining the structure of the deployment manifest. Finally, we give an example of multiple data stores discovery.

In Chapter 6, we present the different aspects of evaluation of our proposals. It details the proof of concept, the evaluation environment and scenarios as well as the results and findings.

Finally, we sum up our contributions in Chapter 7. We conclude this last chapter with some perspectives that we aim to realize at short, medium and long term.

State Of The Art

Contents

2.1	Introduction	9
2.2	Background	10
2.2.1	Cloud Computing	10
2.2.2	Big Data	11
2.2.3	Using Multiple Data Stores In Cloud Environment	13
2.2.3.1	Relational Data Stores	13
2.2.3.2	NoSQL Data Stores	14
2.2.3.3	An Illustrative Example	16
2.2.3.4	NoSQL vs RDBMS	18
2.2.3.5	Polyglot Persistence	19
2.2.3.6	Data Stores Integration	20
2.3	Related Work	22
2.3.1	Unifying Data Models	22
2.3.2	Easy Access To Multiple Data Stores	25
2.3.3	Transparent Access To Integrated Data Stores	28
2.3.4	Capturing Data Requirements And Cloud Data Store Capabilities	32
2.4	Conclusion	37

2.1 Introduction

In this work, we aim to provide an end-to-end solution for supporting developers of multiple data stores based applications in Cloud environments. For ease of understanding the remainder of this manuscript, one should have a basic knowledge on different paradigms and concepts. In addition, it is mandatory to provide an overview of the current stat of the art.

Hence, we dedicate this chapter to briefly introduce these basics to the reader in Section 2.2. Then, we present an overview and a synthesis on the related work in Section 2.3.

2.2 Background

In this section, we present the basic concepts related to our work. We start by introducing in Section 2.2.1 and Section 2.2.2 the environment of our work which involves Cloud Computing as well as the Big Data areas. Afterwards, we define in Section 2.2.3 the different variants that we aim to use in this work resumed in relational and NoSQL data stores, polyglot persistence, and data integration.

2.2.1 Cloud Computing

Cloud Computing is an emerging paradigm in information technology. It is defined by the National Institute of Standards and Technology [10] as a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. Figure 2.1 shows an overview of Cloud Computing reference architecture viewed by NIST. It identifies the major actors, their activities and functions in the Cloud. This figure aims to facilitate the understanding of the requirements, uses, characteristics and standards of Cloud Computing.

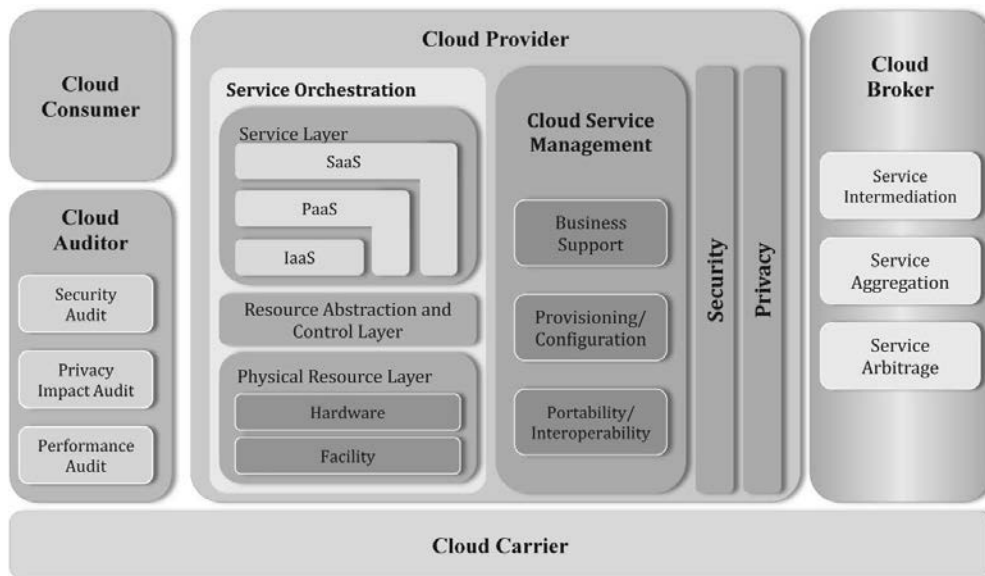


Figure 2.1: Cloud Computing Conceptual Reference Model defined by NIST

Cloud Computing is characterized by its economic model based on "pay-as-you-go" model. This model allows consumers to consume computing resources as needed. Moreover, resources in the Cloud are accessible over the network through standard mechanisms that promote their use by different platforms. The resources are offered to consumers using a multi-tenant model with heterogeneous resources assigned to consumer on demand. These resources are provisioned in an elastic manner that allows to rapidly scale up or down in line with demand. Furthermore,

Cloud resources usage can be monitored and controlled in order to respect the "pay-as-you-go" model.

Services in the Cloud are basically delivered under three well discussed layers namely the Infrastructure as a Service (IaaS), the Platform as a Service (PaaS) and the Software as a Service (SaaS). We present these layers below:

- **IaaS:** Consumers are able to access Cloud resources on demand. These resources can be virtual machines, storage resources and networks. The provider takes the responsibility of installing, transparently managing and maintaining these resources.
- **PaaS:** Consumers are able to develop, deploy and manage their applications onto the Cloud using the libraries, editors and services offered by the provider. The provider takes the responsibility to provision, manage and maintain the Infrastructure resources.
- **SaaS:** Consumers are able to use running applications on an IaaS or a PaaS through an interface. They are not responsible of managing or maintaining the used Cloud resources.

Nowadays, more models appeared and are generally referred to as XaaS targeting a specific area. For example there is the DaaS for Database as a Service, NaaS for Network as a Service, etc.

At the same time, Clouds can be provisioned following different models according to the users needs. If the Cloud is used by a single organization, we talk about Private Cloud. In this case, this organization owns the Cloud and is responsible of its management and maintenance. However, if the Cloud is owned by different organizations, we are talking about community or federation Cloud. Whenever the Cloud is exposed to public use, we are talking about Public Cloud. In this case, an organization owns the Cloud and manages it while it is used by other organizations. Finally, there exists another model in which the Cloud is composed of two or more Clouds. In these Clouds, Public or Private Clouds are glued together.

2.2.2 Big Data

According to the NIST Big Data Public Working Group ² (NIST), Big Data is data which exceed the capacity or capability of current or conventional methods and systems. In [1], IBM defines the term Big Data as information that can not be processed or analyzed using traditional processes or tools. It is mainly based on the 3-Vs model where the three Vs refer to volume, velocity and variety properties (as shown in Figure 2.2). We define these properties as follows:

- **Volume:** It means the processing of large amounts of information. Indeed, over the past decades, several high technologies appear and they are accompanying people in their everyday life. If they can track and record something,

²http://bigdatawg.nist.gov/_uploadfiles/M0392_v1_3022325181.pdf

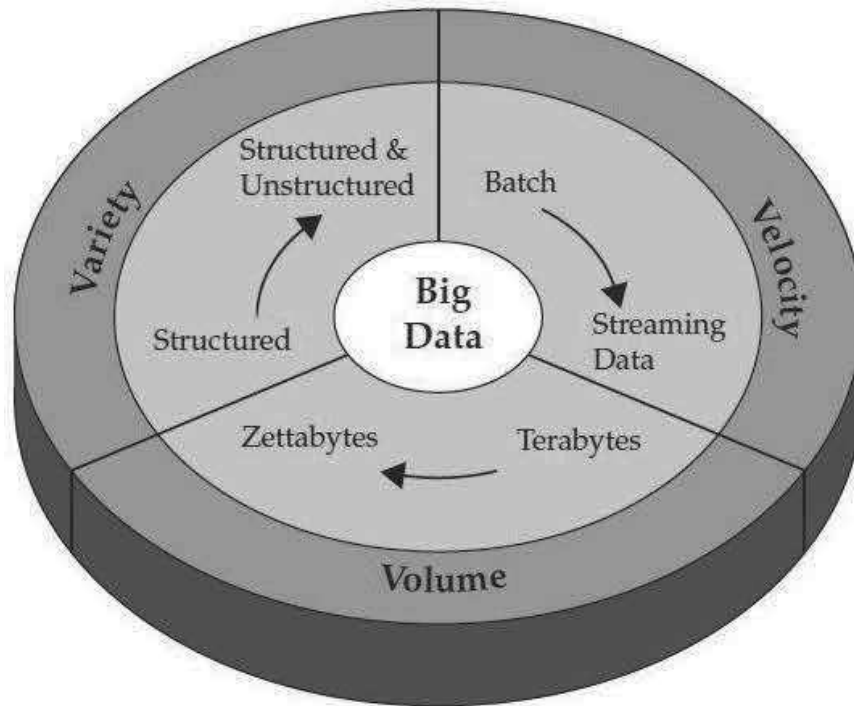


Figure 2.2: IBM characterizes Big Data by its volume, velocity, variety - or simply V3 [1]

they typically do it. For instance, taking your smartphone out of your holster, checking-in for a plane, passing your badge into work, buying a song on iTunes, etc. everyone of these simple actions generates events and data. Managing this big volume of data may be overwhelming big data organization and arises several challenges nowadays. To deal with this, we can find a plethora of modern solutions to analyze data in order to gain a better understanding of our data and to efficiently use it.

- Velocity:** It signifies the increasing rate at which data flows. IBM considers that this property means how quickly the data is collected and stored, and its associated rates of retrieval [1]. It is noteworthy that velocity follows the evolution of the volume characteristic and when it is about a fast moving data, we can call this streaming data or complex event processing. Whether we can handle the data velocity, it will help researchers and business experts in making valuable decisions and efficient data use. To deal with data velocity issues, some researches suggest to do data sampling and data streaming analysis.
- Variety:** It refers to the diversity of data sources and data structures. With the emergence of new technologies (e.g. Cloud Computing, sensors, smart devices, social networks, etc.) resulted data has been complex since it is

a combination of structured, semi-structured and unstructured data. To deal with this characteristic, we find today several kind of data stores (e.g. relational data stores, NoSQL data stores, etc.) allowing to store those heterogeneous data. In addition, it exists a plenty of approaches and solutions allowing data integration based on either a mediator/ wrapper or a unique API, and data analysis without the heterogeneity burden.

In addition to these three characteristics, several people have also proposed to add more Vs to the basic definition. For example, we can cite the veracity which is widely proposed and represents the quality of data (accuracy, freshness, consistency etc.). There is also the data volatility representing how long is data valid and how long should it be stored. In our work, we mainly focus on the variety property and more precisely on multiple data stores based applications in a Cloud environment.

2.2.3 Using Multiple Data Stores In Cloud Environment

In our work, we are interested in multiple data stores based applications in a Cloud environment. For ease of understanding of our approach, we propose to introduce the basic concepts. Indeed, we give a brief introduction of relational and NoSQL data stores which are the main data stores deployed in a Cloud environment (see Section 2.2.3.1 and Section 2.2.3.2) and we present a comparison between them (see Section 2.2.3.4). Then, we present the polyglot persistence concept (see Section 2.2.3.5). We finish this section by talking about the mediation approach (see Section 2.2.3.6).

2.2.3.1 Relational Data Stores

The relational model for databases had been invented by Edgar Frank Codd in 1969 [11]. In this model, data is represented in terms of tuples (or rows), and grouped into relations (or tables). This model enables users to specify and query data based on a declarative way. A constructed database with respect to a relational model is called a relational database. In [12], özsu et al. define a relational database as a table based structured collection of data related to some real-life that we are trying to model. Users directly define what information the database contains and what information they desire from it, and let the database management system software take care of describing data structures for storing the data and retrieval procedures for answering queries. To do so, relational data stores use Structured Query Language (SQL), which is a standard user application that provides an easy programming interface for database interaction.

Operations across databases are grouped and managed based on transactions. A transaction is a unit of work performed by a database management system against a database and treated in a coherent and reliable way independent of other transactions. For example, a transfer of funds from one bank account to another, even involving multiple changes such as debiting one account and crediting another, is a single transaction. In a relational database management system, transactions

are processed with respect to the Atomicity, Consistency, Isolation and Durability (ACID) properties in order to ensure accuracy, completeness and data integrity. These properties are defined by Jim gray [13] as follows:

- **Atomicity:** It means that a transaction must be treated as an atomic unit of operations. Indeed, either all of its operations are executed or none. Hence, we can say that there is no state where a transaction is left partially completed. States should be defined either before the execution of the transaction or after its execution (it can be an abortion or a failure).
- **Consistency:** This property signify that the transaction preserves consistency of stored data in the database and none of the integrity constraints will ever be violated. In other words, if the database was in a consistent state before the execution of a transaction, it must remain consistent after the execution of the transaction as well.
- **Isolation:** Isolation means that one transaction cannot read data from another transaction that is not yet completed. If two transactions are concurrently executing, each one will see the world as if they were sequentially executing, and if one needs to read data that is written by another, it will have to wait until the other is finished.
- **Durability:** After committing a transaction, the changes it has made survive any failure in the database management system. Indeed, the database should be durable enough to hold all its latest updates even if the system fails or restarts.

Over the past decades, relational database management systems had been widely used and dominated the commercial markets with systems such as Oracle, MySQL, Sql Server, etc. They had also been amply deployed in a Cloud environment.

2.2.3.2 NoSQL Data Stores

NoSQL that stands for Not only SQL refers to a new generation of data stores. This kind of data stores do not use either the relational model or the SQL language. They are mostly open source, schema-less, largely distributed database systems that enable rapid, ad-hoc organization and analysis of extremely high-volume, disparate data types [14]. They mainly come from web companies developing very high intensive web applications such as Facebook, Amazon, etc. Unlike the relational data stores, they are not standardized. Indeed, each NoSQL data store has its data model, its query language, its API, etc. NoSQL data stores are sometimes referred to as Cloud databases, non-relational databases, Big Data databases and a myriad of other terms. To query and store data, each NoSQL data store has its own querying mechanism. We can find some NoSQL systems that provide a simple interface/API. Indeed, NoSQL users can access data using for example a text-based protocol or a HTTP REST API with JSON inside. Some other NoSQL

data stores use declarative query languages (i.e. SQL-like, SPARQL-like, etc). This kind of querying mechanism is quite common in graph data stores and is more user-friendly than the other mechanisms.

Relational database management systems respect the ACID properties in order to try to exhibit a strong consistency and availability of their data. Nevertheless, once we start looking at the NoSQL data stores, these properties are not available and new properties appear which are Basically Available, Soft-State and Eventually Consistent. These properties are from the CAP theorem stating that a distributed system can not ensure consistency, availability and partition tolerance at the same time [15–17]. In general, they are referred to as BASE properties or eventual consistency and they enable to ensure a weak consistency and a high Availability and Partition Tolerance. NoSQL data stores are classified in four families of data stores depending of their underlying data model that we present below:

- **Key-value data stores:** Data are stored as a couple of an alpha numeric identifiers called keys and an associated values. These data are grouped in a simple and standalone table referred to as hash table or dictionary. In key-value data stores, data are stored, retrieved, and managed by only keys. We can find nowadays dozens of key-value data stores as Redis, Riak, Berkeley DB, etc. Regarding data querying, it may be done only through the keys since values do not have known structures. Indeed, we can execute CRUD operations using keys and list a set of values by defining some restriction on the keys. These operations may be executed by using either a proprietary API (e.g. Redis) or a HTTP API (e.g. Riak). These data stores work differently compared to the relational data stores. In fact, this latter pre-define the data structure in the database as a series of tables containing fields with well defined data types. However, key-value systems treat the data as a single nontransparent collection which may have different fields for every record. This offers considerable flexibility mainly for dynamic data structures.
- **Document data stores:** Data are stored in documents and grouped together in collections. While each document data store implementation differs on the details of this definition, they all store data in some standard formats or encodings such as XML, JSON, BSON, etc. Practical usability of these data stores can be guessed by the fact that there are a plethora of document data stores available and widely used such as MongoDB, CouchDB, etc. As we previously said the querying method differs from one NoSQL data store to another. For instance, in CouchDB data are accessed using their unique identifier through a HTTP REST API using a browser (i.e. REST client) or the *curl* method. In addition, it is possible to query data by anything else than an identifier using views. A view is defined based on a map-reduce function. Whereas, in MongoDB data are accessed using a set of non HTTP operations provided by a MongoDB Client. The operation allowing the data querying is *find* and it can be combined with some functions in order to

make it more efficient. Compared to relational data stores, collections correspond to tables and documents to records or rows. However, there is one major difference, every record in a table has the same number of fields, while documents in a collection have different numbers of fields.

- **Graph data stores:** A graph data store is structured as a network containing nodes and edges relating these nodes to represent relationships among them. The nodes and the edges may also contain properties that describe some data contained within each object. A relationship connecting two nodes, is identified by its name, and can be traversed in both directions. Many graph data stores are available Neo4j is the most popular one among them. In general, querying mechanisms in graph data stores are declarative. Indeed, they are either SQL-like or SPARQL-like query language. For instance the query language of Neo4j is called Cypher. This latter provides a familiar way to create and query data. This language borrows from SQL the clause-based structure of its queries and some keywords such as "where", "group by", etc. Whereas the AllegroGraph data store uses directly SPARQL query language.
- **Columnar data stores:** This kind of data stores is a database management system that stores data tables as sections of columns of data rather than as rows of data. This storage model is very efficient because of its capability to achieve high level of compression. Consequently columnar data stores offer very high performance and a highly scalable architecture. Nowadays, we find a variety of columnar data stores like HBase, BigTable and HyperTable, Vertica, etc. Most of parallel massively relational systems are columns based and are used in data warehouse environments. Columnar NoSQL data stores are basically key/value stores (family of specific mechanisms to columns) which improve access to data.

2.2.3.3 An Illustrative Example

In this section, we propose an example of the same data model represented with a relational data store, a graph data store and a document data store. Through this example, we will bring out the high level of heterogeneity between relational and NoSQL data stores. Doing so, we can present how the same data store can be represented according to the data model of each data store. Then we concretely show how querying mechanisms differ from one data store to another.

For this purpose, we choose a very simple example which is the person data store. In this latter, we assume that a person is identified by unique identifier, a name (i.e. a first name and a last name), and a country where he/she lives. In addition, we propose to describe a relationship between two persons. For instance, a relationship may be of type co-authorship between two persons, a person that knows another person, etc. In Figure 2.3, we give three examples of representing the person data store. Indeed, we present it by two relational tables. The first table is called *person* and has three columns *personId*, *personName*, and *personCountry*

(see Figure 2.3a). The second table is called *relationship* and has three columns also *initiator*, *receptor*, and *relationship*. These columns denote the initiator of a relation, the receptor of a relation and the name of the relation respectively. Then we showcase how we represent the person data store with a document data store (see Figure 2.3b). In fact, we present each person by a JSON file. Reader should notice that the information about a person is not represented in a unique way. For instance, the first name and the last name of a person is denoted by using either one attribute which is *personName* or two attributes that are *personFirstName* and *personLastName*. In addition, we use the name of a relationship to denote a new attribute as a JSON array element (i.e. this element may have multiple values). Finally, we propose to represent the person data store with a graph data store (see Figure 2.3c). Indeed, each node in this data store represents a person and each relationship denotes either co-authorship between two persons or a person that knows another person.

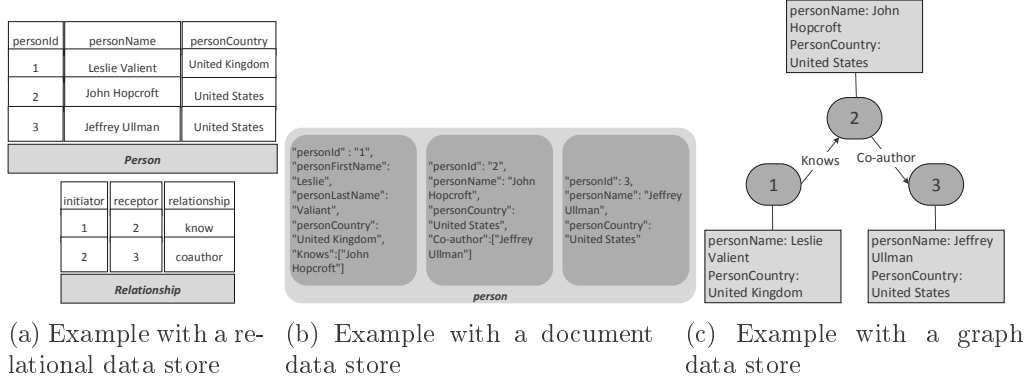


Figure 2.3: Examples of illustrating the same data model with different types of data stores

In the rest of this section, we propose to show how the querying mechanism differs from one data store to another by giving examples of the same query processed across the three data stores:

- **Query 1:** The first query is **get the person having id equals 1**. It is expressed over a relational data store using the declarative query language SQL as follows: `SELECT * FROM person WHERE id=1`. In the graph data store, let us suppose that we are using the Neo4j data store, the query will be written using the Cypher query language as follows: `MATCH n WHERE n.id=1 RETURN n`. Finally, in the document data store, if we suppose that we are using the CouchDB data store, the query will be written as follows: `curl -X GET http://127.0.0.1: 5984/person/1`.
- **Query 2:** The second query is **get the list of persons having the relationship of authorship**. Using SQL syntax, the query can be expressed as follows: `SELECT P1.personName, P2.personName FROM Person P1, Person P2, Relationship R WHERE R.relationship="coauthor" AND P1.personId=R.`

initiator AND P2.personId=R.receptor. Based on Cypher query language, the query is written as follows: `MATCH (n)-[:co-author]->(m) RETURN n, m`. Finally, we express the same query with the CouchDB syntax using the view notion since this query requires filtering. It can be written as follows:

```
function(doc) {
  if(doc.personName && doc.co-author.length > 0) {
    for(var idx in doc.co-author) {
      emit(doc.personName, doc.co-author[idx]);
    }
  }
}
```

2.2.3.4 NoSQL vs RDBMS

NoSQL systems are gaining popularity as a new generation of database management systems against RDBMS. Nevertheless, we do not deny that each system stresses strengths and limitations. In the remainder of this section, we try to show that the NoSQL systems are not meant as the silver bullet to overtake RDBMS problems since they also showcase some notable limitations:

- **Consistency:** Consistency has been widely studied in the context of transactions (see [12] for a general presentation of distributed transaction processing). A strong consistency model has emerged for relational DBMSs named ACID. This model is difficult to scale that is the reason why large scale web applications have popularized a new kind of consistency model named the BASE model. BASE ensures scalability but at the cost of a more complex writing of applications, because programmers have to manually ensure part of the consistency. For example, applications have to access only data in a single node, avoiding the use of costly distributed protocols to synchronize nodes.
- **Scalability:** On the one hand, relational data stores are scalable and we can find a lot of products designed to ensure this property (e.g. OracleRAC). This kind of products work efficiently during data querying. However, it may fail when it comes to function with distributed data, to execute join queries across distributed tables, to update distributed data, etc [18]. This will lead to a problem of consistency. In addition, it is very expensive since they are not free. On the other hand, we can scale NoSQL data stores by replicating and distributing data over many servers. As NoSQL data stores support the execution of simple operations (i.e. read and write operations), this will efficiently work and it will increase the number of queries per second.
- **Data model:** Relational data stores are based on the relational data model which is a generic model for all data stores that we find nowadays (e.g.

MySQL, Oracle, etc.). This is a very interesting point since it allows users to model their databases whatever the complexity of their problems. In addition it is very useful in several situation such as integrating several relational data stores that requires only the definition of a global schema grouping the local schemes of the integrated data stores. Unlike NoSQL data stores that provide a specific data model for each data store. This data model is well tailored to a specific use case. Indeed, each NoSQL data store has been defined for specific tasks and goals and it has been tailored one or few specific usage patterns. In addition, even if NoSQL data stores expose a data model, this latter remains poor in information compared to relational one.

- **Queries type:** Thanks to the consistency of the relational data model, users can efficiently execute complex queries over their data. Hence, they maximize the exploitation of their data and its analysis. Querying relational data is ensured through the standardized and declarative query language SQL. Although this kind of queries is important, their execution may fail in a context of Big Data as relational data stores are not really scalable. Regarding NoSQL data stores, this kind of data stores supports only CRUD operations. This can be seen as one of the shortcomings of NoSQL data stores. Indeed, the fact that NoSQL data stores do not enable complex execution minimize the use of data. To deal with this, such queries can be processed in a programmatic way by coding each operation in source code of the application. This alternative is not optimal for users and remains a troubleshooting solution.

2.2.3.5 Polyglot Persistence

At first glance, reader can think that either types of DBMS can support the application needs and ensure data storage and retrieval providing higher scalability and availability; however, we can not deny that it is more powerful whether NoSQL-DBMS is coupled to a RDBMS and vice versa. This phenomenon is widely called polyglot persistence [2]. This latter consists of the use of multiple data stores and several approaches to data persistence in order to support the whole application needs. For ease of understanding the polyglot persistence, we present the classical example of an e-commerce platform depicted in Fig.2.4. In this example, we illustrate five data stores. In fact, the users' information (e.g. his/her name, his/her address, his/her credit card, etc.) are stored in a *key/value store* and the products' catalog are stored in a *document store*. The financial records (e.g. products' prices, amount of an order, etc.) are saved in a *RDMS*. Finally, binary data (e.g. images of products, videos, etc.) and customer social graph are stored respectively in a *blob store* and a *graph store*.

Polyglot persistence introduces several advantages. For instance, it allows an application to coexist within multiple data stores in order to cover all its needs. Consequently, application programming will be more flexible. Furthermore, it is the solution for data storage when it consists on scaling an application. In fact, data quantities and types will increase; thus the complexity of data managing will

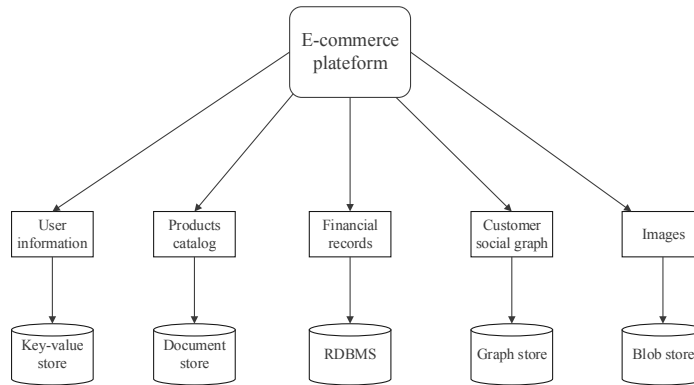


Figure 2.4: Example of polyglot persistence implemented in an e-commerce platform [2]

increase. Hence, this complexity will be weighed against the right data storage. Nonetheless, we can not deny that polyglot persistence represents some limits. Due to the various data stores, the data interaction and manipulation are complex. Indeed, joining, querying, aggregating, etc. data across heterogeneous data stores are not a walk in the park. These operations must be implemented in the application source code by the developer. Moreover, whether we scale an application that is based on shards, partitions and joins, all these elements will accompany the application. This would cause some problems such as managing the partitions or querying with joins. In addition, we must not overlook the strong heterogeneity between the proprietary API. This APIs heterogeneity induces two main problems. First, it ties Cloud applications to specific data stores hampering therefore their migration. Second, it requires developers to be familiar with different APIs.

2.2.3.6 Data Stores Integration

Data integration [19] refers to a set of techniques and approaches used (1) to combine data residing in different data stores and (2) to provide users with a unified view of these data. This unified view is called in general global schema. There is a plenty of techniques and approaches allowing data integration. There are some substantial work proposing schema mappings based solution to integrate heterogeneous data. This solution has four key ingredients: (1) local sources that are data stores to be integrated, (2) local schemas that are schema of sources, (3) a global schema and (4) a set of schema mappings that specifies how local sources and the global schema are related. The first category of solutions to integrate data is the federated database systems [20,21]. In fact, they are database management systems which transparently maps multiple autonomous database management systems into a single one. The three key ingredients of such system are: the autonomy, the heterogeneity and the distribution. This kind of systems is able to decompose the query into sub-queries and then to constitute the final result set.

Afterwards, there is a conventional wisdom about the use of mediation systems.

Indeed, the mediation is a classical approach for querying heterogeneous sources and a mediation-based architecture is usually based on two main parts: the mediator and the adapter (also called wrapper) [22]. Indeed, the mediator allows an application to interact with heterogeneous data stores by seamlessly answering their queries. A mediator requires a global schema, a common query language and a pivot model. Consequently, the mediation approach needs a common data model representing the integrated data sources, a unique query language allowing an application to query in a uniform way, and a pivot model enabling query rewriting. Moreover, the adapter provides access to the integrated data stores by converting application queries into the native format of a target data store. Then, the adapter converts the result set from the native format to a common format understandable by the mediator. It is worth noting that in such architecture, each data store exposes its local data model (also called local schema) and has a unique adapter.

In the mediation approach, the sources contain the real data, while the global schema provides an integrated and virtual view of the sources. Modeling the relation between the sources and the global schema represents an important step in this approach and can be ensured by two methods: global as view (GAV) and local as view (LAV). By modeling the relation, we mean the mapping between local schemas and the global schema and translating the queries to the proprietary query language to interact with the integrated sources. In the GAV method, we process in bottom up by expressing the global schema in terms of data stores. The major advantage of this method is the easiness of queries rewriting and its main shortcoming is the lack of dynamicity. Indeed, if we add a new data store in the system, we are obliged to redefine the global schema. Regarding the LAV method, it is a top-down method and we define the global schema independently from the sources. Then, we define the relationships between the global schema and the sources as views. The advantage of this method is its dynamicity since adding a new source does not imply the redefinition of the global schema. Its main shortcoming is the fact that the queries re-writing algorithm is very expensive.

Recently, new techniques appear to integrate heterogeneous data. As an example, we can cite the work of Paton et al. [23]. In their solution authors propose to integrate data using schema mappings that are validated against user requirements in a pay as you go fashion. To do so, they collect the users' feedback and compute some metrics based on that. Then, they annotate schema mappings to decrease the error percentage. In this section, we mainly focus on the mediation systems since we this kind of systems in our work in order to integrate relational and NoSQL data stores.

In this first main part of this chapter, we introduced the basic concepts related to our work. We defined the Cloud Computing and Big Data areas. Afterwards, we presented basic elements to well understand multiple data stores use in a Cloud environments. Indeed, we defined relational and NoSQL data stores and showcase a comparison between these two families of data stores. Then, we introduced the polyglot persistence and outline its advantages and its limits. Finally, we gave an overview of data stores integration.

In the next main part, we will go further in our investigations. We will give a detailed study of the state of the art related to our proposals in this manuscript.

2.3 Related Work

Supporting the lifecycle of multiple data stores based applications in Cloud environments is not deeply explored. Indeed, it involves developing applications interacting with relational and NoSQL data stores, discovering the suitable Clouds data stores to the applications, deploying applications in Cloud environments, migrating applications from one Cloud environment to another, and running the applications. It is noteworthy that running applications includes complex queries execution over relational and NoSQL execution. Meanwhile, there is a plethora of work dealing with these aspects long before the apparition of Cloud paradigm and NoSQL data stores especially.

In this section, we present different work that focused on ensuring a transparent access to heterogeneous data stores as well as discovering multiple data stores. We start by presenting the work which are interested in unifying data models in Section 2.3.1. Afterwards, we present state of the art of approaches easing access to multiple data stores in Section 2.3.2. In Section 2.3.3, we present some data integration based solutions. Finally, we introduce some solutions interested in capturing application requirements in terms of data and cloud data stores capabilities in Section 2.3.4. We conclude each section by a synthesis of the presented solutions comparing them on the basis of the criteria that we dressed in our research objectives.

2.3.1 Unifying Data Models

The cornerstone of a solution supporting multiple data stores based application is a unified data model. This latter is invoked during the lifecycle of an application. Indeed, it is used by developers to express their queries in their applications source codes. Then, it is used to define the resource model of REST-based APIs enabling the interaction with relational and NoSQL data stores in Cloud environments. Afterward, it helps developers to have a global view on data stores to well describe their requirements and deploy their applications in Cloud environments. Finally, it is used to evaluate and optimize the execution of complex queries over relational and NoSQL data stores. For this purpose, we present in this section some substantial work proposing different unified data models to manage data heterogeneity [24–32].

Unifying relational and object-oriented database systems [24, 25] Won Kim proposes a unified data model enabling the interaction with relational and object-oriented data stores. This data model contains the same components of a relational schema defined with an object-oriented syntax. Indeed, a table is a class, a row is an instance, a column is an attribute, a procedure is a method, a table hierarchy is a class hierarchy, a child table is a subclass and a parent table

is a superclass. Based on this data model, the author proposes the UNISQL/X which is a commercial unified relational and object-oriented database system and implements SQL/X as a query language [24]. Although this model is interesting, it still lacks of generalization as it does not represent data stores and the environment where they are deployed. In addition, it does not take into account NoSQL data stores. In this context, we want to stress that Won Kim tackles the importance of having a unified data model to interact with NoSQL data stores. We refer the interested reader to [33].

GARLIC data model [26] The GARLIC data model is an extension of the ODMG-93 object model. The main building blocks of this model are objects and values. Each object is uniquely identified and has a type expressed in the data model through an object interface. This latter is composed of a set of attributes, relationships and methods. Whereas values can be either base values (e.g. integers, strings, object references, etc.) or structured values (i.e. interfaces without identity). The extension of the ODMG-93 data model enables to support the management of integrity constraints and object references. However, this data model unifies only the access to relational and object oriented data. In addition, it does not take into account the modeling of data stores and the environment where they are deployed.

OEM: Object Exchange Model [27, 28] The Lightweight Object REpository (LORE) is a database management system for semi-structured data. Its query language is referred to as LOREL which has a syntax closer to a select-from-where syntax and its data model is called Object Exchange Model (OEM). This latter is a simple nested object model that is represented using a labeled and directed graph. It is also a self-describing model since it is based on labeling in order to describe objects' meaning. This avoids to define a fixed schema in advance and enables to ensure more flexibility during data modeling. In other word, we can say that each object represents its own schema. Each object in the OEM model is structured by four fields that are a label, a type, a value and an object Id. They denote a variable-length character string describing an object, a data type of an object's value (atomic e.g. integer, string, etc. or complex e.g. set, list, etc.), an object's value, and a unique identifier of an object respectively. Although authors propose a very interesting data modeling in order to ensure a unique view over heterogeneous data sources, the OEM is still lack of some abstractions. In fact, in their modeling they take into account only collections of data and their values. They do not consider data stores representation and environments in which these data stores are deployed. This may be necessary especially when it comes to evaluate and optimize complex queries execution across relational and NoSQL data stores. In addition, representing values of data in a unified data model is not always useful and it is costly in a Big Data context.

The generalized data model of Cloudy [29] Cloudy is a modular Cloud storage system based on a generic data model. It can be customized to meet ap-

plication requirements. Cloudy offers to an application several external interfaces (key/value, SQL, XQuery, etc.) while supporting different storage engines (such as BerkeleyDB or in-memory hash-map). Each Cloudy node in the system is an autonomous running process executing the entire Cloudy stack. Its generic data model is based on the DPI data structure. This latter is a set of fields that allows to identify, transfer, and query data. It contains (1) a key to uniquely identify data, (2) information about the data structure, (3) the data itself, and (4) some metadata. Data in this data structure are handled using three kind of operations to read, write and delete. Even though Cloudy's data model is generic, it is not enough. Indeed, it does not support other types of NoSQL data stores and it does not enable complex queries execution. In addition, this data model does not take into account the modeling of data stores where data are stored and the environment where they are deployed. This is very important in order to evaluate and optimize queries execution in multiple data stores based applications.

SQL++ data model [30] SQL++ is a semi-structured query language and enables the interaction with relational and JSON native data stores. It is based on a simple data model which is a super-set of JSON format and SQL data model (especially relational tables). Indeed, a tuple in the relational model corresponds to a JSON object. A relational column (i.e. string, integer, boolean, etc.) is similar to the respective JSON scalar. Finally, a JSON array is equivalent to a relational table with order. Nevertheless this data model does not represent all kind of NoSQL data stores (namely key/value, column, and graph data stores). In addition, it does not take into account the representation of databases and the environment where they are deployed. This proves that their model lacks of abstraction and dynamicity.

Heterogeneous data resource collaborative management model [31] Tao Sun et al. propose a heterogeneous data resource collaborative management model in Cloud environment. This model represents massive resources storage and massive storage networks generation. It enables the update and balancing of workloads, the security management and some monitoring methods. This model includes four main components: (1) The physical storage layer that stores the heterogeneous data in relational, NoSQL and all type files based data stores. (2) The data resource network layer which allows the management and the use of Cloud storage services by abstracting all the physical nodes (i.e. data stores) into logical nodes and interconnecting these nodes between them. It is noteworthy that each logical node may store various types of data. (3) The data conversion layer that unifies the data resource formats by transforming them into GroupDB database center's format. (4) The GroupDB data management layer which enables data management (e.g. data integration, data fusion, etc.). Nevertheless, authors remain superficial in their proposed model and abstract their data in a high level way. Indeed, they ignore the representation of data collections (e.g. tables, document collections, etc) and data (tuples, documents, attributes, etc.). These omissions can prevent users from controlling their data as well as to express multi-sources queries.

xR2RML: a non-relational databases to RDF mapping language [32]

xR2RML is a language enabling the description of mappings of various types of data stores to RDF. It takes into account relational, XML, object-oriented and some NoSQL data stores. It is an extension of R2RML that belongs to the W3C Recommendation and is a language for expressing mappings from relational data stores to RDF datasets. Once the RDF data set is created, user can seamlessly query it using a declarative query language. Nevertheless, using their approach, users can not express complex queries especially when it comes to interact with NoSQL data stores. In addition, converting result sets (especially SQL result sets) into RDF is likely to be quite inefficient. This can be justified by the absence of semantics during the construction of mappings.

Synthesis

In this section, we present an overview of the related work to unifying relational and NoSQL data models. Hereinafter, we emphasize the shortcomings of these works based on the synthesis showcased in Table 2.1. In this synthesis we consider six criteria. Indeed, we take into account if the data model has a dynamic aspect or not. We characterize a data model by dynamicity if it represents the data, the data stores where they are stored, and the environment where these data stores are deployed. In addition, we verify if the studied solutions enable to unify relational and NoSQL data models. Then, we check if they are understandable by the users and allow to express complex and multi sources queries. Finally, we verify if these data models are defined to represent Cloud data stores or not. In Table 2.1, we show for each work, whether it responds or not to our criteria. In this table, the + is used to say that the proposed work treats a given criteria. While, the – is used to say that the related work does not propose a solution for a given criteria. It is worth noting that we use this notation throughout this the related work section to synthesis the studied works.

Based on this synthesis, we conclude that the majority of the studied works do not support NoSQL data stores. In addition, the studied unified data models lack of understandability and they are not closer to the user. This prevents users from expressing complex queries across heterogeneous data stores. Finally, authors in their modeling do not take into account Cloud environments.

2.3.2 Easy Access To Multiple Data Stores

In some cases, applications want to store and manipulate explicitly their data in multiple data stores. Applications know in advance the set of data stores to use and how to distribute their data on these stores. However, in order to simplify the development process, application developers do not want to manipulate different proprietary APIs especially when interacting with multiple relational and NoSQL data stores. Furthermore, it is impossible to write queries manipulating data coming from different data stores (such as a join between two data stores). Two classes of solutions can be used in this case. The first one is based on the definition

Criteria	Dynamicty	Relational data stores	NoSQL data stores	Queries expression	Understandability	Cloud environment
Studied Solutions						
Won Kim [24,25]	–	+	–	+	+	–
GARLIC data model [26]	–	+	–	+	–	–
OEM [27,28]	–	+	–	+	+	–
Cloudy data model [29]	–	+	–	–	+	+
SQL++ data model [30]	–	+	–	+	–	–
Tao Sun et al. [31]	+	+	+	–	–	+
xR2RML [32]	+	+	+/–	–	–	–

Table 2.1: Synthesis of the related works unifying data models

of a neutral API capable to support access to the different data stores. The second class is based on the model-driven architecture and engineering methodology [34].

In this section, we focus exclusively on the first class of solutions to ease access to multiple data stores (especially NoSQL and relational data stores). Stonebraker [35] exposes the problems and the limits that a user may encounter while using NoSQL data stores especially. These problems derive from the lack of standardization and the absence of a unique query language and API, not only between NoSQL data stores but also between relational data stores and NoSQL data stores. To rule out these problems, there are nowadays some substantial research works proposing solutions to provide transparent access to heterogeneous data stores [30, 36–40]. In this context, some of them are based on the definition of a neutral API; others are based on a framework capable to support access to different data stores.

JDBC: Java DataBase Connectivity [36] Developers used to use JDBC for java application in order to interact with different types of relational data stores (i.e. Oracle, MySQL, etc.). However, interacting with different types of data stores is more complex in the Cloud’s context because there is a large number of possible data stores, which are quite heterogeneous in all dimensions: data models, query languages, transaction models, etc. In particular, NoSQL data stores [41] are widely used in Cloud environment and are not standardized at all: their data models are different (key/value, document, graph or column-oriented), their query languages are proprietary and they implement consistency models based on eventual consistency.

A REST-based API for Database-as-a-Service systems [37] Haselmann et al. present a universal REST-based API concept. This API allows to interact with different Database-as-a-Services (DaaS) whether they are based on relational or NoSQL data stores. They propose a new terminology of different concepts in either types of data stores. In fact, they introduce the terms entity, container, and attribute to represent respectively (i) an information object similar to a tuple in a relational data store, (ii) set of information objects equivalent to a table, and (iii) the content of an information object. These terms represent the resources targeted by their API. This API enables either CRUD operations or complex queries execution. However, authors remain only to the conceptual model and do not give any details about the implementation level of their API. In addition, their resource model is not generic to each category of data store. Haselmann et al. do not deny that proposing such an API is an awkward task. Indeed, they point out in their paper a list of problems that encounter their API.

OData: an Open Data Protocol [30] OData is a REST-based web protocol allowing data querying and updating by building and consuming RESTful APIs. Operations may be either CRUD operations or some complex queries expressed using the OData-defined query language. OData enables to publish and edit resources via web clients within corporate network and across the internet using simple HTTP messages. Resources are identified using URIs and defined based on an abstract data model. Data are represented using a JSON-based format or a XML-based format. Even if this approach seems quite promising, it is more a specification than an implementation especially for the management of complex queries. The query language is also not well very defined for the moment.

Spring Data Framework [38] The Spring Data Framework provides some generic abstractions to handle different types of NoSQL and relational data stores. These abstractions are refined for each data store. In addition, they are based on a consistent programming model using a set of patterns and abstractions defined by the Spring Framework. Nevertheless, adding a new data store is not so easy and the solution is strongly linked to the Java programming model.

SOS: A uniform access to non-relational data stores [39] Atzeni et al. propose a common programming interface to seamlessly access to NoSQL and relational data stores referred to as Save Our Systems (SOS). SOS is a database access layer between an application and the different data stores. To do so, authors define a common interface to access different NoSQL data stores and a common data model to map application requests to the target data store. They argue that SOS can be extended to integrate relational data store; meanwhile, there is no proof of the efficiency and the extensibility of their system.

ONDM: an object NoSQL Datastore Mapper [40] Object-NoSQL Datastore Mapper (ONDM) is a framework aiming at facilitating persistent objects

storage and retrieval in NoSQL data stores. In fact, it offers to NoSQL-based applications developers an Object Relational Mapping-like (ORM-like) API (e.g. JPA API). However, ONDM does not take into account relational data stores.

Synthesis

In this section, we introduced an overview on existing works related to accessing multiple data stores in a unique way. In the following, we will show how the presented works failed to cover some of the objectives that we dressed in Chapter 1 especially when it consists on accessing NoSQL data stores. Nowadays, NoSQL data stores enter the stage of providing more scalability and flexibility in terms of databases in Cloud environment. Nevertheless, there is a gap to fill in terms of developer support. Indeed, each type of NoSQL data stores exposes different traits, drivers, APIs, and data models. In most of the time, application developers are lost in this plethora of data stores and they have to manage that by hook or by crook. All that will degrade the developer productivity.

We propose a synthesis of the studied works based on criteria derived directly from our main objective which is supporting multiple data stores based applications developers in Cloud environment (see Table 2.2). This can be provided by ensuring a unique access for relational and NoSQL data stores. For this purpose we fix a set of six requirements. Indeed, we check if the proposed works are unique API and based on REST architecture. Then, we verify whether these works take into account relational and NoSQL data stores or not. Afterward, we make sure that these works allow the portability of the source code of the application or not. Portability means the possibility of using different programming languages. Finally, we check the extendability of the proposed APIs. By extendability we mean the possibility of adding a new data store in the API.

2.3.3 Transparent Access To Integrated Data Stores

Ensuring transparent access to heterogeneous data stores is a classical problem which has been widely addressed by the mediation community [42]. Two components are used to allow different data sources to be integrated and manipulated in a transparent way: a mediator and adapters. The mediator hides the sources to applications. It implements a global schema described in a neutral data model and including all the information to share. Applications issue queries on the global schema using the query language associated to the neutral data model and the mediator rewrites these queries as sub-queries and sends them to the target data sources. Each data source is encapsulated by an adapter transforming a neutral query into a (set of) data source query.

In this section, we present some substantial works proposing different mediation-based approaches to ensure transparent access heterogeneous data stores [22, 43–50].

Criteria	Unique API	REST based API	NoSQL data stores	Relational data stores	Portability	Extendability
	Studied Solutions					
JDBC [36]	+	–	–	+	–	+
Haselmann et al. [37]	+	+	+	+	–	–
OData [30]	+	+	+	+	+	–
Spring Data Framework [38]	+	+	+	+	–	–
SOS [39]	+	–	+	–	–	–
ONDM [40]	+	–	+	–	–	–

Table 2.2: Synthesis of the related works ensuring easy access to multiple data stores

TSIMMIS: a mediation-based system [22] The Stanford-IBM Manager of Multiple Information sources (TSIMMIS) is a mediation-based system to integrate data stores. The key ingredients of this system are: (1) a unified data model called OEM (see Section 2.3.1), (2) a mediator, (3) wrappers, and (4) the LOREL query language. Although this work represents an important reference in the field of data integration, it is not available today to integrate NoSQL data stores and manage data with big size.

GARLIC: integrating relational and non-relational data stores into a federation [43] GARLIC is a solution to build a federated database system that effectively exploits the query capabilities of the underlying data stores of type IBM DB2, while using a query processor component to develop optimized execution plans and compensate for any functionality that the data sources may lack. IBM DB2 are database management systems that involve relational data, object-oriented data, and non-relational data like JSON and XML. In GARLIC, applications express their queries using SQL and send them to the federated server. This latter cooperates with wrappers of the integrated data stores to construct an optimal execution plan. This execution plan contains the decomposition of the query into sub-queries and the optimal sequencing of the execution of the sub-queries in the integrated data stores. Result sets of these queries are returned to the federated server in order to construct the final result set and return it to the application. Despite its widespreadness, GARLIC presents some shortcomings. For instance, it does not support NoSQL data stores integration which is necessary

nowadays.

Data integration over NoSQL stores using access path based mappings [44, 45] Curé et al. [44] propose a data integration system to enable querying NoSQL and relational data stores. Their approach considers the following assumptions: (1) the global schema is a standard relational data model since the majority of end-users are familiar with this model and its query language, and (2) the sources can be of type relational or NoSQL. To support the execution queries over NoSQL data stores, they define a mapping language to map attributes of the data stores to the global schema and a Bridge Query Language (BQL) to rewrite queries. In a second step, Curé et al. [45] extend their solution by using an Ontology Based Data Access (OBDA). Additionally, they replace BQL with SPARQL³. Although their proposal is promising, there are some lacking functionalities (i.e., no query optimization at the global level, no complex query execution, etc.). Indeed, authors do not define a cost model to optimize the execution of the queries. In addition, they do not support all NoSQL data stores (only document one).

BigIntegrator: a mediation based relational and Clouds' data stores integration [46] The BigIntegrator is a mediation based system enabling the integration of relational and Cloud-based (i.e. Bigtable) data stores. In this system, authors propose to use SQL as a query language in order to join data stored in Bigtable and relational data stores and they define a RDBMS wrapper and a Bigtable wrapper for each kind of the integrated data store. In addition, they propose the BigIntegrator query processor layer that plays the role of the mediator. This layer enables queries re-writing into trees and Datalog queries, queries optimization and algebra operations based execution plan generation. The key ingredients of this layer are (1) the absorber manager that takes the Datalog query and, for each source predicate referenced in the query, calls the corresponding absorber of its wrapper, and (2) the finalizer manager that takes the algebra expression and, for each access filter operator referenced in the algebra expression, calls the corresponding finalizer of its wrapper. Despite its importance, the BigIntegrator system stresses some limitations. Indeed, it suffers from the lack of genericity since it supports just one particular NoSQL data store (i.e. Bigtable). Besides, if a user wants to integrate a new data store, he/she has to define a well tailored wrapper to this data store, an absorber manager, and a finalizer manger.

Bridging relational and document-centric data stores [47] Roijackers et al. propose a hybrid mediation approach in which they integrate relational and document data stores. To do so, they define a logical representation to store document data in relational data stores, and they extend the SQL query language in order to seamlessly query data. The proposed extended-SQL language is called NoSQL query pattern (NQP). Based on that, they develop a basic approach for

³<http://www.w3.org/TR/rdf-sparql-query/>

processing such extended queries, and present a range of practical optimizations. Nevertheless, this approach does not take into account other NoSQL data stores.

CloudMdsQL: querying heterogeneous Cloud data stores with a common language [48] Valduriez et al. propose the Cloud multidatastore query language (CloudMdsQL) which is a common language for relational and NoSQL data stores querying. Indeed, it enables to query multiple heterogeneous data stores using a single query containing nested sub-queries. Each sub-query targets a particular data store and may contain embedded invocations to the data store's native query interface. Due to the absence of schemes in NoSQL data stores, authors propose to use table expressions which is generally an expression that returns a table. This notion allows to represent a nested query and especially targets schemeless data stores. These table expressions are saved in data stores; hence this yields important optimization opportunities (e.g. minimizing data transfers between nodes, reusing queries decomposition and optimization techniques, etc.). They also propose a query engine that follows a distributed mediator/wrapper based architecture where they refer to the mediator as master and to the wrappers as workers. Although their proposal is promising, authors do not propose a cost model to optimally execute a query. In addition, they do not specify whether CloudMdsQL may work in a Big Data context or not.

Ontop: an OBDA based relational data integration [49, 50] Ontop is an Ontology Based Data Access (OBDA) for relational data integration. It exposes relational data stores as virtual RDF graphs by linking the term in the ontology to the data stores through mappings got using the mapping language R2RML. Then the resulted RDF graph is queried using SPARQL. The ontop framework is composed of four inputs that are: (1) an ontology to uniquely represent data stores, (2) a set of mappings to link data stores to the ontology, (3) the integrated data stores, and (4) the queries written in SPARQL. Although the ontop represents a new flavor to integrate data, it showcases some limits. Indeed, it does not enable NoSQL data stores integration. In addition, it does not support complex queries execution such join queries over relational and NoSQL data stores. Finally, it is not meant to be deployed in a Cloud environment.

Synthesis

In this section, we present a synthesis of the works that we have analyzed above in Table 2.3. Indeed, we investigate whether these solutions enable the integration of relational and NoSQL data stores in Cloud environment. In addition, we take into account application using a big size of data. During our study, we check also if these approaches enable the execution of complex queries across relational and NoSQL data stores or not. Finally, we are interested in the works that invoke cost models to optimize queries execution.

In this context, the source of all these challenges is the appearance of NoSQL data stores. This new kind of data stores represents a plethora of categories and

Criteria	Big data support	Relational data stores	NoSQL data stores	Complex queries execution	Cost Model based optimization	Cloud environment
Studied Solutions						
TSIMMIS [22]	–	+	–	+	+	–
GARLIC [43]	–	+	–	+	+	–
Curé et al. [44, 45]	–	+	+ / –	–	–	–
BigIntegrator [46]	–	+	+ / –	–	–	+
Roijackers et al. [47]	–	+	+ / –	–	–	–
CloudMdsQL [48]	–	+	+	+	–	+
Ontop [49, 50]	+	+	–	–	+ / –	–

Table 2.3: Synthesis of the related works ensuring transparent access to integrated data stores

variants causing a high level of heterogeneity. Additionally, NoSQL data stores are schemaless i.e., there is no local schemas to represent the integrated data stores. Subsequently, it will be complicated to construct a global schema in the mediator. Added to that, the studied works do not involve Cloud computing and Big Data areas. Finally, few of them propose a cost model to optimize the execution of queries.

2.3.4 Capturing Data Requirements And Cloud Data Store Capabilities

Choosing one or multiple data stores based on data requirements is a very important step before deploying and running applications in Cloud environments. In this context, we have to (1) define application needs and requirements towards data, (2) expose data stores capabilities, and (3) define a matching algorithm between application needs and data stores capabilities.

For this purpose, we present and discuss some previous works that provide the requirements cited above. We introduce first the CAMP standard in order to discover the application needs and the data stores requirements [51]. Then, we give a short overview of the CDMI standard that allows to describe and discover data storage capabilities [52]. We want to emphasize that it is also possible to obtain

data stores capabilities using the data stores proprietary API. Finally, we discuss some works that enable an application to negotiate its requirements with Cloud data stores and to express these requirements in data contracts (or manifests) [53–58]. This latter is an agreement between the application and the Cloud data stores.

Cloud Application Management for Platforms [51] Cloud application management for platforms (CAMP) is a specification defined for applications management, including packaging and deployment, in the PaaS. Indeed, CAMP provides to the application developers a set of interfaces and artifacts based on the REST architecture in order to manage the application deployment and their use of the PaaS resources. An application may be thereafter deployed, run, stopped, suspended, snapshotted, and patched in the PaaS. Concerning the relationship data storage/application, CAMP allows an application to define its needs especially in terms of data storage.

CAMP proposes a PaaS resource model. This model allows the developer to control the whole life cycle of his/her application and its deployment environment. This model contains the four following resources:

- *Platform*: This resource provides an overview on the PaaS and allows to discover which application is running. Indeed, this resource references the deployed applications, the running applications and the PaaS capabilities which are called respectively *Assembly Templates*, *Assemblies*, and *Components*. It also enables the PaaS *capabilities and requirements* discovery. In our case it allows to get a primary view on the application and the data storage platform.
- *Assemblies*: This resource exists under two possible forms. The first one is the *Assembly Template* and it defines a deployed application and its dependencies. Whereas, the second is the *Assembly* and it represents an application instance.
- *Components*: This resource may exist in two kinds that are *Application Component* and *Platform Component*. Each kind may be under two forms also. On the one hand, the *Application Component* and *Application Component Template* define respectively an instantiated instance of an application component and a discrete configuration of a deployed application component. On the other hand, we have *Platform Component* and *Platform Component Template* that represent an instantiated instance of a platform component and a discrete configuration of a platform component.
- *Capabilities and Requirements*: A capability defines the configuration of *Application Components* or *Platform Components*. Whereas a requirement expresses the dependency of an application on an *Application Component* or *Platform Component* and it is created by the application developer or administrator.

As we see through the PaaS resource model of CAMP, there are various resources that are focusing on defining the capabilities of either application or data storage platform. These resources are *Platform*, *Assembly Template*, *Application Component Template*, *Platform Component Template*, and *Capabilities and Requirements*. Hence, it enables the discovery and the publication of application needs in terms of data storage. Moreover, it allows to define the data stores capabilities.

Cloud Data Management Interface [52] The Storage Networking Industry Association (SNIA), an association of producers and consumers of storage networking products, has defined a standard for Infrastructure as a Service (IaaS) in the cloud. This standard is referred to as Cloud Data Management Interface (CDMI). Based on the REST architecture, CDMI allows to create, retrieve, update, and delete data in the cloud. It provides an object model in which the root element is called *Root Container*. This latter represents the main container that will contain all the needed data by an application. This element is related to the following elements:

- *Container*: The *Root Container* element may contain zero or more sub-containers. Each container is characterized by a set of capabilities that will be inherited by the data objects that it contains.
- *Data object*: Each *container* may store zero or more *data objects*. These are used to store data based on the *container* capabilities.
- *Queue object*: The *Queue objects* stores zero or more values. These values are created and fetched in a first-in first-out manner. The queue mechanism organizes the data access by allowing one or more writers to send data to a single reader in a reliable way.
- *Domain object*: This element allows to associate the client's ownership with stored objects.
- *Capability object*: The *capability objects* describe the *container's* capabilities in order to discover the cloud capabilities in terms of data storage.

This international standard supports several features. Indeed, it enables the available capabilities discovery in the cloud storage offering. In addition, it supports the containers and their contents management. It defines also meta-data to be associated with containers and the objects they contain. So, CDMI allows to define the data stores requirements but at a lower level (it is more infrastructure oriented than platform oriented).

Data contracts for Cloud-based data marketplaces [53–55] Truong et al. propose to model and specify data concerns in data contracts to support concern-aware data selection and utilization. For this purpose, they define an abstract

model to specify a data contract and the main data contract terms. Moreover, they propose some algorithms and techniques in order to enforce the data contract usage. In fact, they present a data contracts compatibility evaluation algorithm and they define how to construct, compose and exchange a data contract. In [54], they introduce their model for exchanging data agreements in the Data-as-a-Service (DaaS) based on a new type of services which is called Data Agreement Exchange as a Service (DAES). This model is called DDescription MOdel for DaaS (DEMODS) [55]. However, Truong et al. propose this data contract for data and not to store data or to help the developer to choose the appropriate data stores for his/her application.

An approach to identify and monitor SLA parameters for storage-as-a-service cloud delivery model [56] Ghosh et al. identify non-trivial parameters of the Service Level Agreement (SLA) for Storage-as-a-Service Cloud which are not offered by the present day Cloud vendors. Moreover, they propose a novel SLA monitoring framework to facilitate compliance checking of Service Level Objectives by a trusted third part. Although Ghosh et al. try to enrich the SLA parameters to support the Storage-as-a-Service, this is still inadequate for our purpose in this paper which is discovering data stores based on application requirements and data stores capabilities.

An automated approach to Cloud storage service selection [57,58] Ruiz-Alvarez et al. propose an automated approach to select a PaaS storage service according to an application requirements. For this purpose, they define a XML schema based on a machine readable description of the capabilities of each storage system. The goal of this XML schema is twofold: (i) expressing the storage needs of consumers using high-level concepts, and (ii) enabling the matching between consumers requirements and data storage systems offerings. Nevertheless, they consider in their work that an application may interact with only one data store and they did not invoke the polyglot persistence aspect.

An ontology-based system for Cloud infrastructure services discovery [59] Zhang et al. propose the Cloud Computing Ontology (CoCoOn) that enables to denote functional and non functional concepts, attributes and relations of infrastructures services in Cloud environments. Using the CoCoOn ontology, Cloud providers expose a description of their services. In addition, they propose CloudRecommender which is a system implementing their ontology in a relational model. This system enables the selection of infrastructure services using SQL queries to match user requests to the services descriptions. However, this approach is intended to work in the IaaS and not in the PaaS. In addition, users can not discover multiple resources in one query.

Cloud service selection based on variability modeling [60] Wittern et al. propose to use Cloud feature modeling based on variability modeling in order

to present users requirements and Cloud services capabilities. Based on that, they define also a Cloud service selection process as a methodology for decision making. Although this approach is automatic and dynamic, it does not support the discovery and the selection of multiple services.

Synthesis

In this section, we present a synthesis about the presented works above. To do so, we fix a set of six criteria on which we rely to evaluate these works. We check whether these works propose a solution to (1) describe multiple data stores based application requirements, (2) expose data stores capabilities and (3) apply matching techniques to elect the most suitable Cloud provider to an application. We also verify if these works use manifests to describe requirements and capabilities in terms of data stores. Finally, we point out that we target applications in PaaS. This analysis is showcased in Table 2.4.

Criteria	Multiple data stores	PaaS	Manifest-based modeling	Capturing users requirements	Data store capabilities	Matching techniques
Studied Solutions						
CAMP [51]	–	+	–	+	+	–
CDMI [52]	–	–	–	+	–	–
Truong et al. [53–55]	–	–	+	+	–	+ / –
Ghosh et al. [56]	–	–	–	–	–	–
Ruiz-Alvarez et al. [57,58]	–	+	+	+	+	+
Zhang et al. [59]	–	–	–	+	+	–
Wittern et al. [60]	–	–	–	+	+	+

Table 2.4: Synthesis of the related works capturing Data Requirements And Cloud Data Store Capabilities

Against this analysis, we conclude that all the studied works do not support multiple data stores based applications discovery. In addition, few of them are intended to support data stores in the PaaS layer of a Cloud environment. Finally, these works do not propose to describe applications requirements and data stores capabilities in manifest even if this modeling ensures more automaticity in data stores discovery.

2.4 Conclusion

Approaches about unifying data models, easing access to heterogeneous data stores, integrating data, and Cloud services discovery exist way before the appearance of NoSQL data stores and the notion of polyglot persistence. So, there is a plethora of works dealing with them. In this chapter, we first introduced the basic concepts that are straightforwardly related to our work. Then, we provide an overview of the existing work in each area. Tables 2.1, 2.2, 2.3, and 2.4 present illustrative pictures to show how each work responds to the objectives listed in Chapter 1. Despite all these proposals, challenges introduced by Cloud environments and NoSQL data stores are not fully addressed. In general, these proposals do not support NoSQL data stores and are not well integrated in Cloud applications lifecycle.

In our work, we will propose four solutions that respond to all the dressed objectives. Indeed, these solutions concern the support of multiple data stores based applications in Cloud environments. We propose an integrated set of models, algorithms and tools aiming at alleviating developers task for developing, deploying, executing complex queries and migrating multiple data stores based applications in Cloud environments. It is worthy to say that we take into account relational and NoSQL data stores.

In the next chapter, we will introduce our first two contributions. Indeed, we will present our unified data model covering the heterogeneity between relational and NoSQL data stores. Afterwards, we will present ODBAPI which is a REST based API that allows the execution of queries across relational and NoSQL data stores in a unique way. This API has been defined based on a resource model mapping the different concepts in our unified data model.

Unifying The Access To Relational And NoSQL Data Stores In Cloud Environments

Contents

3.1	Introduction	40
3.2	Overview Of Our Approach	40
3.3	Unifying Data Models	42
3.3.1	Informal Description Of The Unified Data Model	42
3.3.2	Formal Description Of The Unified Data Model	44
3.3.3	Global Schema	45
3.3.4	Query Algebra	47
3.3.4.1	Projection Operation	47
3.3.4.2	Selection Operation	48
3.3.4.3	Cartesian Product	48
3.3.4.4	Join Operation	50
3.3.4.5	Union Operation	51
3.3.4.6	Intersection Operation	51
3.3.4.7	Nest and Unnest Operations	52
3.3.4.8	Algebraic Tree	54
3.4	ODBAPI: a unified REST API for relational and NoSQL data stores	55
3.4.1	Use Cases And Motivation	56
3.4.1.1	First scenario: Application migration from one data store to another	56
3.4.1.2	Second scenario: Polyglot persistence	57
3.4.2	ODBAPI: OpenPaaS DataBase API	58
3.4.2.1	Panorama Of ODBAPI	58
3.4.2.2	Operations of ODBAPI	59
3.4.2.3	Examples Of Queries	62
3.5	Conclusion	66

3.1 Introduction

Storing and manipulating data from multiple data stores in Cloud environments is gaining momentum. However, applications developers are central to that. Indeed, they supposed to develop this kind of applications, to deploy it, to execute their queries, to manage its migration from one Cloud environment to another, etc. To cope with this, developers must be familiar with data models and APIs of each data stores including new data stores in case of migration. Our main purpose in this work is to rid developers from these heavy and cumbersome tasks. Indeed, we provide them a set of tools, techniques, and algorithms aiming at developing, deploying and migrating multiple data stores based applications in Cloud environments.

In this chapter, we start by giving an overview of our contributions that we will detail in next chapters (see Section 3.2). Then, we present our unified data model that we use to cover the heterogeneity between relational and NoSQL data stores by ensuring a unique view on these data stores (see Section 3.3). Finally, we introduce our REST based API, ODBAPI that enable to uniformly execute queries across relational and NoSQL data stores (see Section 3.4).

3.2 Overview Of Our Approach

For ease of presentation of our work, we propose to introduce the main constituents of our approach that we detail in next chapters. We show in particular how these elements enable overcoming the problems ($Pb_1 - Pb_4$) listed in the first chapter of this manuscript (see Chapter 1). Figure 3.1 depicts how these constituents intervene during the development, discovery, deployment and execution steps. Our approach relies on the following 4 elements:

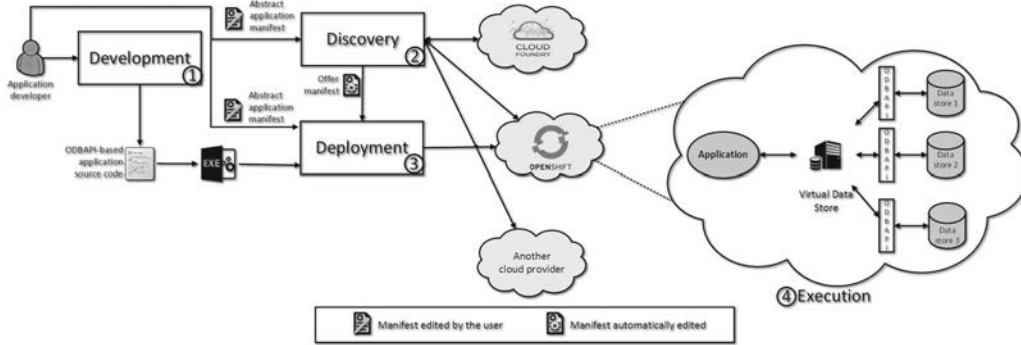


Figure 3.1: An end-to-end overview of our solution

- **Unifying data models:** We define a data model which abstracts the underlying (explicit/implicit) integrated data stores models, and we provide a common and unified view so that developers can define and execute their queries over heterogeneous data stores. We propose to refine this data model

by the refinement rules in order to provide more information to the developers. During the development step, the developers dispose of a global data model expressed according to our unified model and which integrates local data store models. Our unified data model decouples query definitions from the data stores specific languages. During the execution step, the unified data model is used by the virtual data stores in order to process complex queries and to optimally execute it (contributing to resolving thereafter).

- ODBAPI, a REST-based API:** Based on our unified data model, we define a resource model upon which we develop a streamlined and a unified REST API. We called it ODBAPI that stands for OpenPaas DataBase API. This API enables to interact with relational and NoSQL data stores in a unique and uniform way. Our API decouples the interactions with data stores from their specific drivers. The proposed unified data model enables to express queries and to interact with heterogeneous data stores using ODBAPI. Hence, developers do not have to deal with various query languages and APIs. In addition, they do not have to adapt their code when migrating their applications. ODBAPI is also used in our solution integrating NoSQL and relational data stores to execute complex queries. Indeed, it plays the role of wrappers in order to convert queries and result sets from the sources format to the virtual data stores (VDS for short) format and vice versa.
- Virtual data stores:** Wrapper REST services (i.e. ODBAPI services) enable executing simple queries over the involved data stores. However, they are not meant to execute complex queries (such as join, union, etc.) on multiple data stores or entity sets. In our approach, we consider VDS a specific component responsible for executing queries submitted by a multiple data store application. A VDS holds the global data model integrating the different data stores and which is specified according to our unified data model and a set of the refinement rules. Besides, it is accessible as a REST service complying to the ODBAPI and maintains the end-points of the wrapper REST services (in other word the integrated data stores). A multiple data store application submits CRUD and complex queries to the VDS which is responsible of their execution by interacting with appropriate data stores via their REST services. VDSs enable developers to express their complex queries over multiple data stores in a declarative way and take in charge the burden of their executions.
- Dedicated components for discovery and deployment:** In our approach, we consider two components: the discovery and deployment modules. They are responsible of finding appropriate cloud environments and deploying multiple data store applications on them respectively. As depicted in Fig. 3.1, developers first express their requirements about the used data stores as well as the computation environment via an abstract application manifest. Based on that manifest, the discovery component finds and elects

the appropriate cloud environment and produces an offer manifest. This manifest will be in turn used by the deployment component to deploy the application on that selected environment. The discovery and deployment modules relieves the application developers from the burden of dealing with different APIs and discovery/deployment procedures.

3.3 Unifying Data Models

In this section, we present our integrative and unified data model used by application programmers to express their access to the different data stores they use and by virtual data stores to execute complex queries. In order to abstract the different data models of the data stores, we define a unified data model capable to express all data constructions. We present the different concepts needed to define the unified data model (see Section 3.3.1) and we formally define them (see Section 3.3.2). The unified data model is used to express the "global" schema of an application, that is the description of all entity sets used in the application together with their the refinement rules (see Section 3.3.3). Finally, a query algebra is presented, allowing to express complex queries as algebraic trees in order to optimize and evaluate their execution (see Section 3.3.4).

3.3.1 Informal Description Of The Unified Data Model

Our proposed data model unifies the different concepts coming from existing data stores nowadays. As a first step of this work, we propose to do a comparison between the different data stores and their concepts. In Table 3.1, we represent a comparison chart between the different concepts used in four categories of data stores which are commonly used in Cloud environments: *MySQL* a relational data store, *Riak* a key/value data store, *Cassandra* a column data store, and *MongoDB* a document data store. For instance, a table in *MySQL* is equivalent to a collection in *MongoDB*, to a Column family in *Cassandra* and to a database in *Riak*. We propose to refer to this concept by *Entity Set* in our unified data model. In addition, a row in *MySQL*, a document in *MongoDB*, a Column key/Column value pair in *Cassandra* and a key/value pair in *Riak* are represented by the *Entity* concept. In order to organize elements of type *databases* belonging to one Cloud environment, we define a new concept that we call *Environment* that includes all resources.

Based on Table 3.1, we define the unified data model (see Figure. 3.2) based on five concepts. For ease of understanding our data model, we based ourselves on Figure 3.3 in which we depict three *EntitySets* of type *MongoDB*, *Riak*, and *MySQL* from right to left.



Figure 3.2: Unified data model

Relational concepts	MongoDB concepts	Riak concepts	Cassandra concepts	Unified data model concepts
Database	Database	Environment	Keyspace	Database
Table	Collection	Database	Column family	Entity Set
Row	Document	Key/value	Column key/ Column value	Entity
Column	Field	key	Column key	Attribute

Table 3.1: Comparison chart of concepts used in different data stores

- The *Attribute* concept: It represents an attribute in a data store. In Figure 3.3, the elements *personName*, *Rank*, and *year* are concepts of type *Attributes*.

Conference	Rank
STOC	A*
WAW	C
COLT	A*
DEXA	B
ESA	A
PODC	A*

personId	personName	personCountry	personAffiliation
1	Leslie Valient	United Kingdom	Harvard University
2	John Hopcroft	United States	Cornell University
3	Jeffrey Ullman	United States	Stanford University
4	Dana Angluin	United States	Yale University
5	Martins Krikis	United States	Yale University
6	Serge Abiteboul	France	INRIA Saclay
7	Christos Papadimitriou	United States	University of California at Berkeley
8	Martha Sideri	United States	University of Pennsylvania
...

author	title	year
author: ["Leslie Valiant"] year: "1999" title: "Robust Logics" Conference: "STOC"	title: "Manipulation-Resistant Reputations Using Hitting Time" Year: "2007" author: ["John Hopcroft", "Jeffrey Ullman"] Conference: "WAW"	year: "1996" title: "Computational Aspects of Organization Theory (Extended Abstract)" author: ["Christos Papadimitriou"] Conference: "ESA"
author: ["Dana Angluin", "Martins Krikis"] title: "Teachers, Learners and Black Boxes" year: "1997" Conference: "COLT"	author: ["Serge Abiteboul"] title: "Issues in Monitoring Web Data" Year: "2002" Conference: "DEXA"	year: "1991" title: "Optimal Coteries" author: ["Christos Papadimitriou", "Martha Sideri"] Conference: "PODC"

Figure 3.3: Examples of *EntitySet* concepts of type *MongoDB*, *Riak*, and *MySQL* from right to left

- The *Entity* concept: An *Entity* is a set of one or multiple *Attributes*. In Figure 3.3, we show *Entities* in the *EntitySet* called Conference ranking identified by the *Attributes* Conference.
- The *EntitySet* concept: An *EntitySet* is a set of one or multiple concepts of type *Entity*. In Figure 3.3, dblp, Conference ranking and Person represent *EntitySets* of type respectively document collection, Key/Value database, and relational table.
- The *Database* concept: A *Database* contains one or multiple concepts of type *EntitySet*. In Figure 3.4, we showcase an application A interacting with three *Databases* called dblpDB of type MongoDB data store, personDB of type relational data store, and RankDB of type Riak data store. These *Databases* contain respectively the *EntitySets* dblp, Person, and Conference ranking illustrated in Figure 3.3.
- The *Environment* concept: The root concept in our model is *Environment*. This concept represents a pool of data stores and an application can choose some of them to interact with. As a concrete example, we can give the example of the Cloud environment in which the application A interacts with the three data stores (see Figure 3.4).

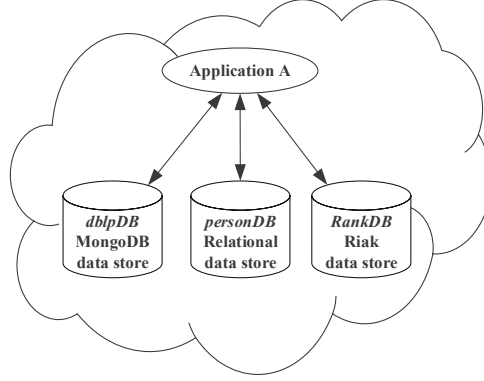


Figure 3.4: An application interacting with three data stores in a Cloud environment

3.3.2 Formal Description Of The Unified Data Model

For a good modeling of data stored in the integrated data stores and an efficient definition of the queries and their execution, we propose to define our query algebra with respect to the unified data model. For this purpose, we present in this section a formal definition of the unified data model concepts:

- *Attribute* concept: A concept a of type *Attribute* is characterized by a pair of the elements $\{t, q\}$: (1) an *Attribute* type t , and (2) a qualified name q allowing to identify the path to the appropriate *Attribute*. A type *Attribute* may be either atomic (i.e. it is a predefined type t such as integer, String, etc.) or recursively composed by applying type constructors (record and set) on existing types (either atomic or composed). We note by \mathbb{A} the set of the *Attribute*.
- *Entity* concept: We define a concept e of type *Entity* as follows: $e = \{a_1, \dots, a_n\}$ where $a_i | i \in \{1..n\} \in \mathbb{A}$. It represents the set of all possible attributes of an entity, that is a specific data may not used part of these attributes. We denote by \mathbb{E} the set of the *Entity*.
- *Entity set* concept: We define a concept es of type *EntitySet* as follows: $es = \{e_1, \dots, e_n\}$ where $e_i | i \in \{1..n\} \in \mathbb{E}$. We denote by \mathbb{ES} the set of *EntitySet*.
- *Database* concept: We define a concept d of type *Database* as follows: $d = \{es_1, \dots, es_n\}$ where $es_i | i \in \{1..n\} \in \mathbb{ES}$. We note by \mathbb{D} the set of *Database*.
- *Environment* concept: We define a concept env of type *Environment* as follows: $env = \{d_1, \dots, d_n\}$ where $d_i | i \in \{1..n\} \in \mathbb{D}$. We denote by \mathbb{ENV} the set *Environment*.

Let us now give an example of a formal presentation of the document *EntitySet* *dblp* that belongs to a database named *dblpDB* deployed in the Cloud environment *OurEnvironment*:

- $OurEnvironment \in \mathbb{ENV}$, $OurEnvironment = \{dblpDB\}$ where $dblpDB \in \mathbb{D}$,
- $dblpDB \in \mathbb{D}$, $dblpDB = \{dblp\}$ where $dblp \in \mathbb{ES}$,
- $dblp \in \mathbb{ES}$, $dblp = \{e\}$ where $e \in \mathbb{E}$,
- $e \in \mathbb{E}$, $e = \{author, year, title, Conference\}$ where $author, year, title, Conference \in \mathbb{A}$,
- $author, year, title, Conference \in \mathbb{A}$,
 $author = \{set\ of\ String, OurEnvironment.dblpDB.dblp.author\}$,
 $year = \{String, OurEnvironment.dblpDB.dblp.year\}$,
 $title = \{String, OurEnvironment.dblpDB.dblp.title\}$,
 $Conference = \{String, OurEnvironment.dblpDB.dblp.Conference\}$.

3.3.3 Global Schema

Integrating relational and NoSQL data stores is very important since it allows combining data residing in different sources and providing users with a unified view of these data. This is indeed unrealizable since NoSQL data stores are schema-less, but some kind of global schema is needed to allow users to express and execute their queries. For this sake, we propose in this section to define a simple and minimalist global schema describing all the *EntitySets* accessible by an application. It is noteworthy that our global schema provides users by the minimum information required to express their queries but it does not define any kind of schema integration like in classical mediation systems.

The global schema that we propose is our unified data model enriched by a set of refinement rules. The refinement rules help applications programmers to correctly express their queries. For instance, we define the Correspondence rules to express the logic expression in a join query and to remove semantic ambiguities between attributes. In the following, we propose to define some kinds of the refinement rules.

Definition 3.3.1 (Correspondence rules) *A correspondence rule is defined as follows: Let us consider $A_1 < t_1, q_1 >, A_2 < t_2, q_2 > \in \mathbb{A}$. A_1 is equivalent to A_2 if and only if t_1 and t_2 are compatible, and A_1 and A_2 denote the same information (they have the same semantic). To represent such relation, we use the binary operator \equiv to denote $q_1 \equiv q_2$.*

We denote by \mathbb{CR} the set of the correspondence rules.

In Fig. 3.5, we illustrate the corresponding data model to the *dblp*, *person*, and *conferenceRanking* *EntitySets*. In this schema, we precise the database name and the environment where we can find these *EntitySets*. In addition, we showcase two correspondence rules using dashed lines. The first is between the *Attributes* *personName* and *author* in the *person* and *dblp* *EntitySets* respectively. Whereas

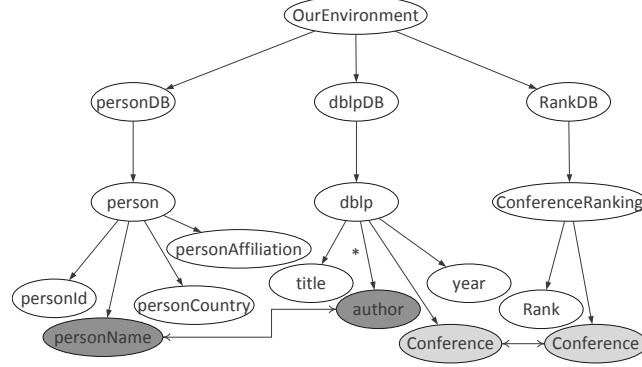


Figure 3.5: The corresponding data model to the *dblp*, *person*, and *conferenceRanking* entity sets

the second rule is between the two *Attributes conference* in both *conferenceRanking* and *dblp EntitySets*. Based on this, we can write (For ease of presentation, we do not note the whole qualified name of each *Attribute*. We remove the name of an environment and a database and we start from the name of an *EntitySet*):

cr_1 *conferenceRanking.conference* \equiv *dblp.conference*.

cr_2 *person.personName* \equiv *elementOf(dblp.author)*. We use the function *elementOf* in order to express the equivalence between a *person.personName* and one element of the multi-valuated attribute *dblp.author*.

Definition 3.3.2 (Concatenation rules) A concatenation rule is defined as follows: Let us consider $A_1 < t_1, q_1 >, A_2 < t_2, q_2 >, A_3 < t_3, q_3 > \in \mathbb{A}$. the concatenation of A_1 to A_2 is equivalent to A_3 if and only if t_1, t_2 and t_3 are compatible, and (A_1, A_2) and A_3 denote the same information (they have the same semantic). To represent such relation, we use the binary operator \equiv and the concatenation symbol \bullet to denote $q_1 \bullet q_2 \equiv q_3$.

We denote by COR the set of the concatenation rules.

Based on the Definition 3.3.2, we give the following example: Let us suppose that we have two data stores with the address domain. In the first one, there is only one attribute called *fullAddress*. Whereas the second one contains four attributes that are *Address*, *City*, *Country*, and *ZIP*. According to this, we can write the following concatenation rule:

cor_1 *Address* \bullet *City* \bullet *Country* \bullet *ZIP* \equiv *fullAddress*.

With the same idea, it is possible to add new types of annotations on the global schema. For instance, it would be interesting to add views (represented as virtual entity sets) and rules defining views as operations on actual entity sets. In this way, our global schema may be closer to an integrated schema. We would investigate this as a future work.

3.3.4 Query Algebra

Codd [61] defined the query algebra as a family of algebra with a well-founded semantics used for modeling the data stored in relational databases, and defining queries on it. Indeed, it is a theory in which we define a set of operations that can be executed on relations containing a set of tuples. The query language that we propose is based on a query algebra defined on *EntitySets/Entities* and not on relations/tuples. This query algebra is simple and is composed by the classical unary operations (e.g. selection and projection) and binary operations (e.g. Cartesian product, join, union, etc.). For manipulating complex attributes like those defined in our unifying model, more complex algebra can be used, notably N1NF algebra [62]. In our work, we define two kind of operations coming from the N1NF algebra that are the nest and unnest operations. In the rest of this section, we define the projection, the selection, the Cartesian product, the join, the union, and the intersection. Besides, we present the nest and unnest operations.

Readers must notice that for ease of presentation of the examples in this section, we propose to denote *EntitySets* by tables as the relational model. In addition, we will use the person and dblp *EntitySets* to illustrate the examples (see Table 3.2 and Table 3.3).

personId	personName	personCountry	personAffiliation
1	Leslie Valient	United Kingdom	Harvard University
2	John Hopcroft	United States	Cornell University
3	Serge Abiteboul	France	INRIA Saclay

Table 3.2: Sample of three *Entities* from the person *EntitySet*

author	title	year	Conference
Leslie Valient	Robust Logics	1999	STOC
Serge Abiteboul	Issues in Monitoring Web Data	2002	DEXA

Table 3.3: Sample of two *Entities* from the dblp *EntitySet*

3.3.4.1 Projection Operation

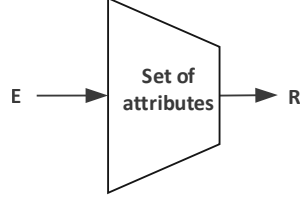
A projection is a unary and mathematical operation that takes as input an *EntitySet* and returns as output an *EntitySet*. The resulted *EntitySet* contains *Entities* restricted to a set of *Attributes*. During the projection, whether an *Entity* does not contain one or multiple target *Attributes*, this *Entity* will be discarded of the output. This problem can be encountered in the case of an *EntitySet* of type NoSQL because this kind of data stores is schema-less. Formally, we define the projection operation as follows:

Definition 3.3.3 (Projection operation) $\pi_{attributes} : \mathbb{ES} \rightarrow \mathbb{ES}$

Let $E = \{e_1, \dots, e_n\} \subset \mathbb{ES}$ and $R = \{r_1, \dots, r_l\} \subset \mathbb{ES}$.

The projection from E is defined as follows: $R = \pi_{attributes}(E) = \{r | r \text{ formed only by attributes of the projection}\}$

We graphically represent the projection as depicted in Figure 3.6. For instance, if we apply the projection on the person *EntitySet* taking into account the attributes `personId` and `personName`. We can denote the projection as $Result = \pi_{personId, personName}(person)$ and the *EntitySet* Result is illustrated in Table 3.4.



personId	personName
1	Leslie Valient
2	John Hopcroft
3	Serge Abiteboul

Figure 3.6: The graphic notation of the projection operation

Table 3.4: Result of the projection of the person *EntitySet* using the attributes `personId` and `personName`

3.3.4.2 Selection Operation

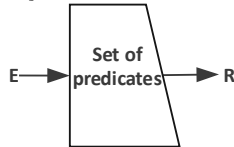
A selection is a unary and mathematical operation that takes as input an *EntitySet* and returns as output an *EntitySet*. The resulted *EntitySet* contains the set of *Entities* satisfying a given prepositional logic expression that we name *predicates*. During the selection, if a predicate contains an *Attribute* that is missing, the appropriate *Entity* will be ignored. Formally, we define the selection operation as follows:

Definition 3.3.4 (Selection operation) $\sigma_{predicates} : ES \rightarrow ES$

Let $E = \{e_1, \dots, e_n\} \subset ES$ and $R = \{r_1, \dots, r_l\} \subset E$.

The selection from E is defined as follows: $R = \sigma_{predicates}(E) = \{r | r \in E \wedge predicates(r)\}$

We graphically represent the selection as depicted in Figure 3.7. For instance, if we apply the selection on the person *EntitySet* taking into account the predicate `personId = 1`, we can denote the selection as $Result = \sigma_{personId=1}(person)$ and the *EntitySet* Result is illustrated in Table 3.5.



personId	personName	personCountry	personAffiliation
1	Leslie Valient	United Kingdom	Harvard University

Figure 3.7: The graphic notation of the selection operation

Table 3.5: Result of the selection of the person *EntitySet* taking into account the predicate `personId = 1`

3.3.4.3 Cartesian Product

A Cartesian product is a binary and mathematical operation that takes as input two *EntitySets* and returns as output an *EntitySet*. The resulted *EntitySet* contains

the set of all unions combining two *Entities* belonging to the each input *EntitySet*. Formally, we define the Cartesian product function as follows:

Definition 3.3.5 (Cartesian product operation) $\times : \mathbb{ES} * \mathbb{ES} \rightarrow \mathbb{ES}$

Let $E = \{e_1, \dots, e_n\} \subset \mathbb{ES}$, $F = \{f_1, \dots, f_m\} \subset \mathbb{ES}$ and $R = \{r_1, \dots, r_l\} \subset \mathbb{ES}$.

The Cartesian product of E and F is defined as follows: $R = E \times F = \{(e_1 \bullet f_1), (e_1 \bullet f_2), \dots, (e_1 \bullet f_m), \dots, (e_i \bullet f_j), \dots, (e_n \bullet f_m)\}$

We graphically represent the Cartesian product as depicted in Figure 3.8. For instance, if we apply the Cartesian product of *dblp* and *person* *EntitySets*, we can denote $Result = dblp \times person$ and we obtain the resulted *EntitySet* represented in Table 3.6.

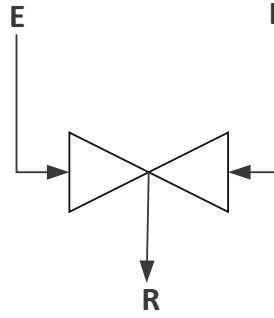


Figure 3.8: The graphic notation of the Cartesian product operation

author	title	year	Conference	personId	personName	personCountry	personAffiliation
Leslie Valient	Robust Logics	1999	STOC	1	Leslie Valient	United Kingdom	Harvard University
Leslie Valient	Robust Logics	1999	STOC	2	John Hopcroft	United States	Cornell University
Leslie Valient	Robust Logics	1999	STOC	3	Serge Abiteboul	France	INRIA Saclay
Serge Abiteboul	Issues in Monitoring Web Data	2002	DEXA	1	Leslie Valient	United Kingdom	Harvard University
Serge Abiteboul	Issues in Monitoring Web Data	2002	DEXA	2	John Hopcroft	United States	Cornell University
Serge Abiteboul	Issues in Monitoring Web Data	2002	DEXA	3	Serge Abiteboul	France	INRIA Saclay

Table 3.6: Result of the Cartesian product of the *dblp* and *person* *EntitySets*

3.3.4.4 Join Operation

A join operation can be seen as a conditional Cartesian product where an element of the join result has to satisfy the join condition. The *EntitySet* resulted from a join operation contains the set of all combinations of *Entities* in the input *EntitySets* that are equal on a given common attributes denoted by a logical expression *cond*.

Formally, we define the join operation as the following function:

Definition 3.3.6 (Join operation) $\times_{cond} : \mathbb{ES} * \mathbb{ES} \rightarrow \mathbb{ES}$

Let $E = \{e_1, \dots, e_n\} \subset \mathbb{ES}$, $F = \{f_1, \dots, f_m\} \subset \mathbb{ES}$ and $R = \{r_1, \dots, r_l\} \subset \mathbb{ES}$, the join operation of E and F is defined as follows: $R = E \times_{cond} F = \{r | r \in E \times F \wedge cond(r)\}$

We graphically represent the join operation as depicted in Figure 3.12. For instance, if we apply the join between *dblp* and *person* *EntitySets* based on the logical expression *person.personName in dblp.author*, we can denote $Result = person \times_{person.personName \text{ in } dblp.author} dblp$ and we obtain the resulted *EntitySet* represented in Table 3.7. It is worth noting that the logical expression is defined based on the global schema thanks to the correspondence rules (see Section 3.3.3).

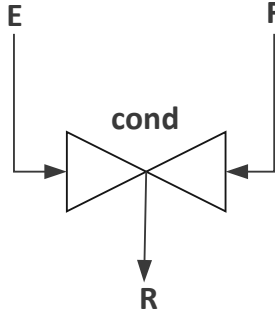


Figure 3.9: The graphic notation of the join operation

author	title	year	Conference	personId	personName	personCountry	personAffiliation
Leslie Valient	Robust Logics	1999	STOC	1	Leslie Valient	United Kingdom	Harvard University
Serge Abiteboul	Issues in Monitoring Web Data	2002	DEXA	3	Serge Abiteboul	France	INRIA Saclay

Table 3.7: Result of the join between the *dblp* and *person* *EntitySets*

3.3.4.5 Union Operation

A union is a binary and mathematical set operation that takes as input two *EntitySets* and returns as output an *EntitySet*. The resulted *EntitySet* contains the set of all distinct *Entities* belonging to the one of the *EntitySets* in entry. It is noteworthy that the involved *EntitySets* must be union-compatible (i.e. the two *EntitySets* must have the same set of attributes). Formally, we define the union function as follows:

Definition 3.3.7 (Union operation) $\cup : \mathbb{ES} * \mathbb{ES} \rightarrow \mathbb{ES}$

Let $E = \{e_1, \dots, e_n\} \subset \mathbb{ES}$, $F = \{f_1, \dots, f_m\} \subset \mathbb{ES}$ and $R = \{r_1, \dots, r_l\} \subset \mathbb{ES}$.

The union of E and F is defined as follows: $R = E \cup F = \{r | r \in E \vee r \in F\}$

We graphically represent the union as depicted in Figure 3.10. To give an example of the union operation between two *EntitySets*, let us consider that the person *EntitySet* is constructed from two *EntitySets* called P1 and P2 respectively (see Table 3.8 and Table 3.9). For instance, if we apply the union of P1 and P2 *EntitySets*, we can denote $person = P1 \cup P2$ and we obtain the resulted *EntitySet* represented in Table 3.2. It is worth noting that the entity having personId equals to 2 appears in both *EntitySets* P1 and P2 and appears in the *EntitySet* person just one time since the resulted *EntitySet* represents the set of all distinct *Entities* belonging to the one of the *EntitySets* in entry.

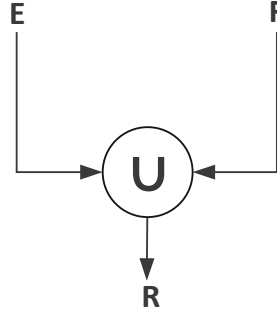


Figure 3.10: The graphic notation of the union operation

personId	personName	personCountry	personAffiliation
1	Leslie Valient	United Kingdom	Harvard University
2	John Hopcroft	United States	Cornell University

Table 3.8: Sample of two *Entities* from the P1 *EntitySet*

3.3.4.6 Intersection Operation

An intersection is a binary and mathematical set operation that takes as input two *EntitySets* and returns as output an *EntitySet*. The resulted *EntitySet* contains all *Entities* belonging to both *EntitySets* in entry. It is noteworthy that the involved

personId	personName	personCountry	personAffiliation
2	John Hopcroft	United States	Cornell University
3	Serge Abiteboul	France	INRIA Saclay

Table 3.9: Sample of two *Entities* from the P2 *EntitySet*

EntitySets must be intersection-compatible (i.e. the two *EntitySets* must have the same set of attributes). Formally, we define the intersection function as follows:

Definition 3.3.8 (Intersection operation) $\cap : \mathbb{ES} * \mathbb{ES} \rightarrow \mathbb{ES}$

Let $E = \{e_1, \dots, e_n\} \subset \mathbb{ES}$, $F = \{f_1, \dots, f_m\} \subset \mathbb{ES}$ and $R = \{r_1, \dots, r_l\} \subset \mathbb{ES}$.

The intersection of E and F is defined as follows: $R = E \cap F = \{r | r \in E \wedge r \in F\}$

We graphically represent the intersection as depicted in Figure 3.11. To give an example of the intersection operation between two *EntitySets*, let us consider that the person *EntitySet* is constructed from two *EntitySets* called P3 and P4 respectively (see Table 3.10 and Table 3.11). For instance, if we apply the intersection of P3 and P4 *EntitySets*, we can denote $person = P1 \cap P2$ and we obtain the resulted *EntitySet* represented in Table 3.2.

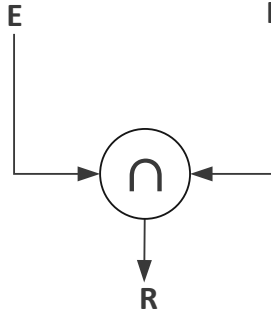


Figure 3.11: The graphic notation of the intersection operation

personId	personName	personCountry	personAffiliation
1	Leslie Valient	United Kingdom	Harvard University
2	John Hopcroft	United States	Cornell University
3	Serge Abiteboul	France	INRIA Saclay
5	Martins Krikis	United States	Yale University

Table 3.10: Sample of two *Entities* from the P1 *EntitySet*

3.3.4.7 Nest and Unnest Operations

The operations we have introduced so far are used to handle simple attributes. In this section, we give more importance to the complex attributes (e.g. an attribute

personId	personName	personCountry	personAffiliation
1	Leslie Valient	United Kingdom	Harvard University
2	John Hopcroft	United States	Cornell University
3	Serge Abiteboul	France	INRIA Saclay
4	Dana Engluin	United States	Yale University

Table 3.11: Sample of two *Entities* from the P2 *EntitySet*

composed by multiple attributes, a multivalued attribute, etc.). In particular, we focus on the nest and unnest operations that allow to create a complex structure and to flatten it respectively.

The unnest operation takes as input an *EntitySet* E having at least two attributes including one complex attribute called a (i.e. a is composed of a set of attributes) and F the set of the other attributes. This operation outputs the *EntitySet* E with the attribute a flatten into a set of sub-attributes. Below we define the unnest operation (see Definition 3.3.9).

Definition 3.3.9 (Unnest operation) $unnest : \mathbb{ES} * \mathbb{A} \rightarrow \mathbb{ES}$

Let $E = \{e_1, \dots, e_n, a\} \subset \mathbb{ES}$, $a = \{a_1, \dots, a_m\} \subset \mathbb{A} | \exists a_k | k \in \{1..m\}, a_k = \{v_{k1}..v_{kp}\}$ a complex attribute (i.e. $E = \{e_1, \dots, e_n, \{a_1, \dots, a_m\}\}$) and $R = \{r_1, \dots, r_l\} \subset \mathbb{ES}$.

The unnest operation is defined as follows:

$$R = unnest(E, a_k) = \{r_j | j \in \{1..l\} | \exists t \in \{1..p\}, r_j = \{a_i | i \in \{1..m\} \setminus \{k\}\} \bullet \{v_{kt}\}\}$$

In Table 3.12, we showcase an example of a complex attribute which is the *author* attribute. This latter is multivalued and has two values: *John Hopcroft* and *Jeffrey Ullman*. By applying the unnest operation, we denote it as follows: $unnest(dblp, author)$ and we obtain two new entities and each one contains mono-valued attribute called *author* (see Table 3.13).

author	title	year	Conference
John Hopcroft	Manipulation resistant reputations	2007	WAW
Jeffrey Ullman	using hitting time		

Table 3.12: Sample of complex attributes from the dblp *EntitySet*

author	title	year	Conference
John Hopcroft	Manipulation resistant reputations	2007	WAW
	using hitting time		
Jeffrey Ullman	Manipulation resistant reputations	2007	WAW
	using hitting time		

Table 3.13: Result of the unnest operation

The nest operation takes as input an *EntitySet* E and a non-empty set of attributes a and returns as output an *EntitySet* R containing a complex attribute

composed from the attributes of a . This operation is defined as follows in Definition 3.3.10.

Definition 3.3.10 (Nest operation) $Nest : \mathbb{ES} * \mathbb{A} \rightarrow \mathbb{ES}$

Let $E = \{e_1, \dots, e_n\} \subset \mathbb{ES}$, $a = \{a_1, \dots, a_m\} \subset \mathbb{A} | \exists a_k | k \in \{1..m\}, a_k \in \{v_{k1}..v_{kp}\}$ an attribute to nest and $R = \{r_1, \dots, r_l\} \subset \mathbb{ES}$.

The nest operation is defined as follows:

$$R = nest(E, a_k) = \{r_j | j \in \{1..l\} | r_j = \{a_i | i \in \{1..m\} \setminus \{k\}\} \bullet \{v_{k1}..v_{kp}\}\}$$

In Table 3.14, we showcase an example of an entity set containing simple attributes. The two last entities contain the same values of the attributes *title*, *year*, and *Conference*. However, the value of the *author* attribute is different. Hence, we propose to apply the nest operation based on the *author* attribute. We can denote the following operation: $nest(dblp, author)$. We showcase the result of this operation in the Table 3.15.

author	title	year	Conference
Leslie Valient	Robust Logics	1999	STOC
John Hopcroft	Manipulation resistant reputations using hitting time	2007	WAW
Jeffrey Ullman	Manipulation resistant reputations using hitting time	2007	WAW

Table 3.14: Sample of the dblp *EntitySet*

author	title	year	Conference
Leslie Valient	Robust Logics	1999	STOC
John Hopcroft	Manipulation resistant reputations using hitting time	2007	WAW
Jeffrey Ullman			

Table 3.15: Result of the nest operation

3.3.4.8 Algebraic Tree

All these operations that we defined in the previous sections will be used by the virtual data stores in order to evaluate and optimize query execution (see Chapter 4). Indeed, once the virtual data store receives a query, it rewrites it as an algebraic tree in order to reduce the size of data and the time of the query execution.

An algebraic tree contains two kinds of nodes that are either roots and a leaf representing *EntitySets* or internal nodes representing the elementary operations. These nodes are linked between each others using incoming edges representing the operands and outgoing edges representing resulted *EntitySet* of an operation. In Figure 3.12, we give an example of rewriting a query in an algebraic tree including all the operations defined above. For instance, let us consider the query allowing to join *dblp* and *person EntitySets* in order to get the affiliation of authors of

the paper entitled "Robust Logics". To answer this query, we have to do (1) a selection based on "dblp.title=Robust Logics", (2) a join using the logic expression *person.personName in dblp.author*, and (3) a projection to get the affiliation of authors and their names.

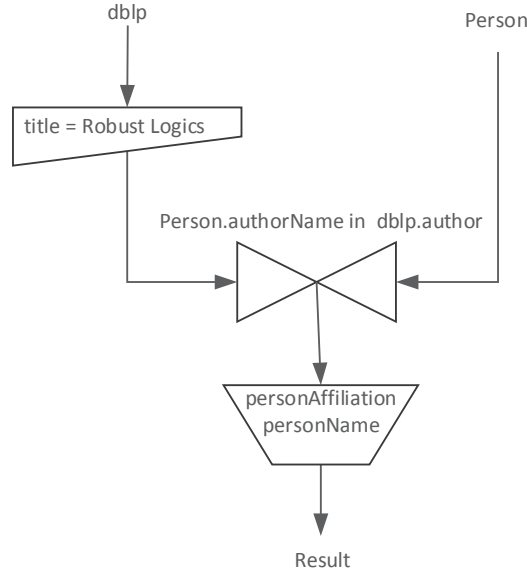


Figure 3.12: Exemple of an algebraic tree

In the next section, we will present our second contribution in this manuscript which is ODBAPI. This latter is a REST based API that allows the execution of queries across relational and NoSQL data stores in a unique way. This API has been defined based on a resource model mapping the different concepts in our unified data model.

3.4 ODBAPI: a unified REST API for relational and NoSQL data stores

In order to satisfy different storage requirements, cloud applications usually need to access and interact with different relational and NoSQL data stores having heterogeneous proprietary APIs. This APIs heterogeneity induces two main problems. First it ties cloud applications to specific data stores hampering therefore their migration. Second, it requires developers to be familiar with different APIs.

In this section, we introduce the OpenPaaS Project in which we propose ODBAPI and two possible use cases (see Section 3.4.1). Then, we present a generic resource model defining the different concept used in each type of data store (see Section 3.4.2). This resources model is defined based on the unified data model presented in the previous Section 3.3. These resources are managed by ODBAPI a streamlined and a unified REST API enabling to execute CRUD and complex operations on different NoSQL and relational databases (see Section

3.4.2). ODBAPI decouples Cloud applications from data stores alleviating therefore their migration. Moreover it relieves developers task by removing the burden of managing different APIs.

3.4.1 Use Cases And Motivation

In this section, we propose to present two use cases in order to motivate the utility of our API. Indeed, we introduce two possible scenarios of the use of our API: application migration from one data store to another (see Section 3.4.1.1), and multiple data store use in one cloud environment (see Section 3.4.1.2).

3.4.1.1 First scenario: Application migration from one data store to another

Cloud environment usually provides one data store for the deployed applications. However, in some situations, this data store model does not support the whole applications requirements. Subsequently, an application needs to migrate from one data store to another in order to find a more convenient data store to its requirements (in the same Cloud provider or another one). It is worth noting that an application may migrate to another cloud environment to find the most suitable data store according to some new requirements.

In Figure 3.13, we exemplify a migration scenario where the *Application A* needs to migrate from Cloud provider 1 where it interacts with the document data store *CouchDB* to Cloud provider 2 in order to meet new data requirements. In the new Cloud environment, the application connects to another document data store *Mongo DB*. In this case, developers need (1) to discover another environment that can support new storage requirements, (2) to re-adapt the code so that application A can interact with MongoDB API, (3) to migrate data from the old data store to the new one, and (4) to deploy their application on the new environment.

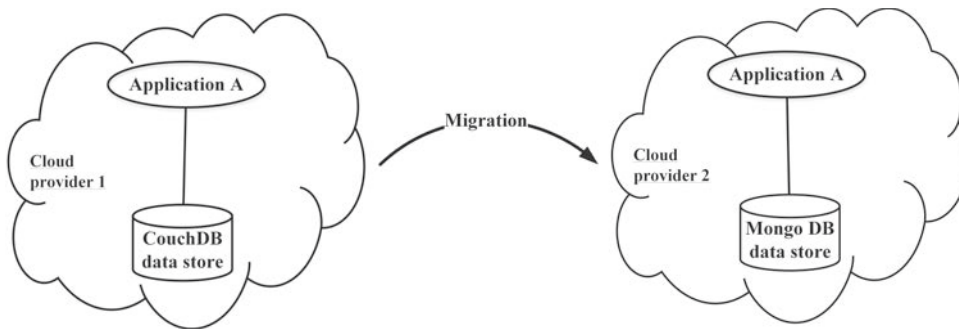


Figure 3.13: Application migration from one cloud environment to another scenario

At first glance, application migration seems simple and automatic; but behind this scenario there are onerous responsibilities to be ensured by applications' developers. In the rest of this section, we will only focus on the first task of the developer which is the re-adaption of the application source code in order to in-

interact with the new data store. To deal with this, developers must first of all be familiar with the old (resp. new) data model and API of the old (resp. new) data store. Based on that, he/she will adapt the source code of the application by looking on each instruction and modifying it using the new API. This work is costly in terms of time and manpower. Indeed, developers have to discover the new proprietary API (resp. APIs) and to be familiar with it (resp. them). Then, he/she must update the source code.

3.4.1.2 Second scenario: Polyglot persistence

In a Cloud environment, an application can use multiple data stores that corresponds to what is popularly referred to as the polyglot persistence. In Figure 3.14, we show an example of this situation. *Application A* interacts with three heterogeneous data stores: a relational data store, a document data store that is *CouchDB*, and a key value data store which is *Riak*. The main advantage is to use specialized data stores, well adapted to special requirements.

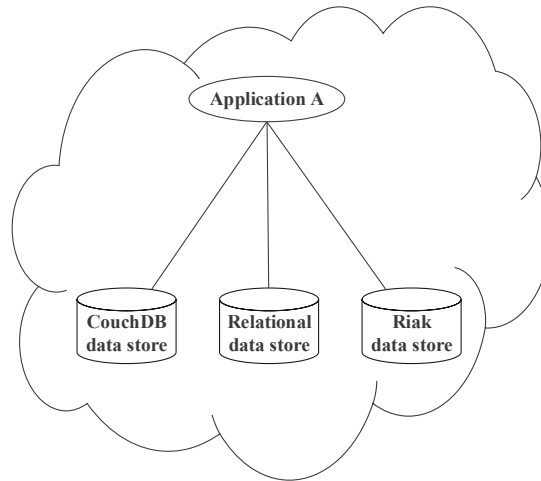


Figure 3.14: Using multiple data stores in Cloud environment

Nevertheless, this scenario presents some limits. Linking an application with multiple data stores is very complex due to the different APIs, data models, query languages and consistency models. If the application needs to query data coming from different data sources (e.g joining data, aggregating data, etc.), it can not do it declaratively. Finally, the different data stores may use different transaction and consistency models (for example classical ACID and eventual consistency). It is not easy for programmers to understand these models and to properly code their application to ensure desired properties. In either scenarios, we see how the developer's task is cumbersome in order to develop and manage the source code of an application since this application interacts with heterogeneous data stores with different APIs. Hence, the developer has to be familiar with a plethora of APIs.

To deal with this problem, we propose in this section a streamlined and unified REST API enabling to execute operations on different NoSQL and relational

databases. This API is called ODBAPI and its goal is twofold: First, it simplifies the developer's task during the application migration from one data store to another. By using a unique API, the adaptation of the source code of an application becomes easier. Second, it alleviates the burden of interacting with various data stores at the same time by using just ODBAPI.

3.4.2 ODBAPI: OpenPaaS DataBase API

Based on our unified data model, we define a generic resource model defining the different concepts used in each category of data store. These resources are managed by ODBAPI. This API enables the execution of CRUD operations on different types of data stores. Doing so, we give an overview of ODBAPI (see Section 3.4.2.1). Then, we introduce the different operations ensured by ODBAPI (see Section 3.4.2.2).

3.4.2.1 Panorama Of ODBAPI

In this section, we turn the focus on ODBAPI. This latter is designed to provide an abstraction layer and seamless interaction with data stores deployed in Cloud environments. Developers can execute CRUD and complex queries in a uniform way regardless of the type of the data store whether it is relational or NoSQL. We want to support three types of queries within ODBAPI:

- **Simple CRUD queries on a single data store:** ODBAPI allows to express these queries in a uniform way from the target data store.
- **Complex queries on a single data store:** Some data stores may support a highly expressive query language. We want to let developers used this language even if it is not compatible with our query algebra (see Section 3.3). In this case, ODBAPI encapsulates queries in the body of the query using JSON format.
- **Complex queries on multiple data stores:** Using virtual data stores, several data stores can be abstracted in a uniform way. In this case, complex queries can be expressed using our query algebra (we use SQL-like statement for that).

An overview of ODBAPI is given in Figure 3.15. The figure is divided in four parts that we introduce in the following starting from the right to the left side:

- **Data stores:** We have first of all the deployed data stores that a developer may interact with during his/her application coding. In the figure, we showcase that a developer may use a relational data store, a document data store that is CouchDB and a key/value data store that is Riak.
- **Proprietary APIs and drivers:** Second, we find the proprietary API and driver of each data store implemented by ODBAPI. For instance, we use in

our API implementation the JDBC API and MySQL driver to interact with a relational DBMS.

- **ODBAPI interface:** The third part of Figure 3.15 represents the ODBAPI interface and the different implementations of each data stores. In fact, it represents the shared part between all the integrated data stores and it provides a unique view to the application side. It contains specific implementations of each data store. The current version of ODBAPI implementation includes four data stores: (1) relational DBMS, (2) CouchDB, (3) MongoDB and (4) Riak. It is note worthy that if a user wants to integrate a new data store and define interactions with it in ODBAPI, he/she has simply to add the specific implementation of that data store by including its driver and its API.
- **ODBAPI operations:** Finally, we show the different operations that ODBAPI offers to the user. These operations are introduced in the next section.

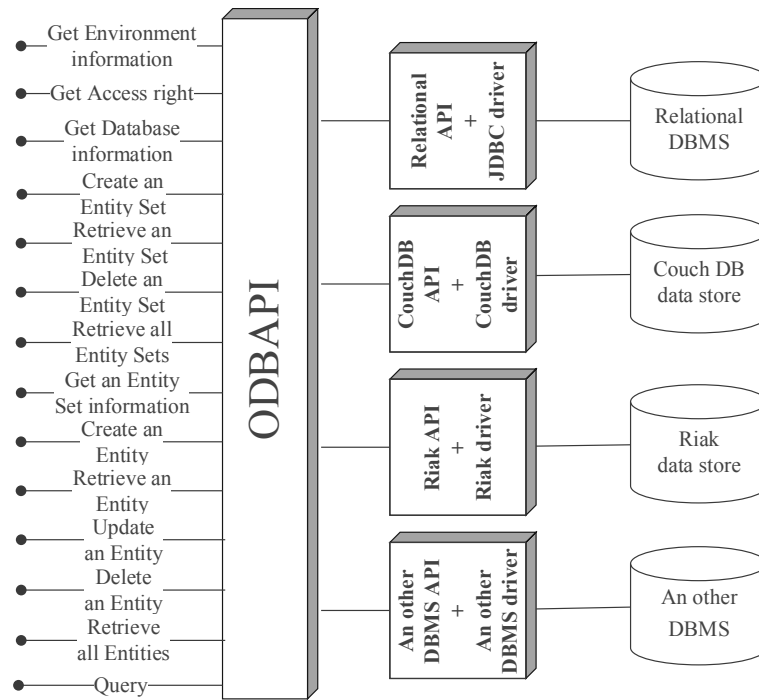


Figure 3.15: An overview of ODBAPI

3.4.2.2 Operations of ODBAPI

In this section, we introduce the list of operations provided by ODBAPI. In Figure 3.16, we propose a box based representation of the different operations ensured by ODBAPI. Each box contains the name of a resource (e.g. /odbapi/entityset/esName, /odbapi/entityset/esName/entity/entityID, etc) and the different opera-

tions that are intended to this resource. Each operation is ensured by a REST method (e.g. GET, PUT, etc).

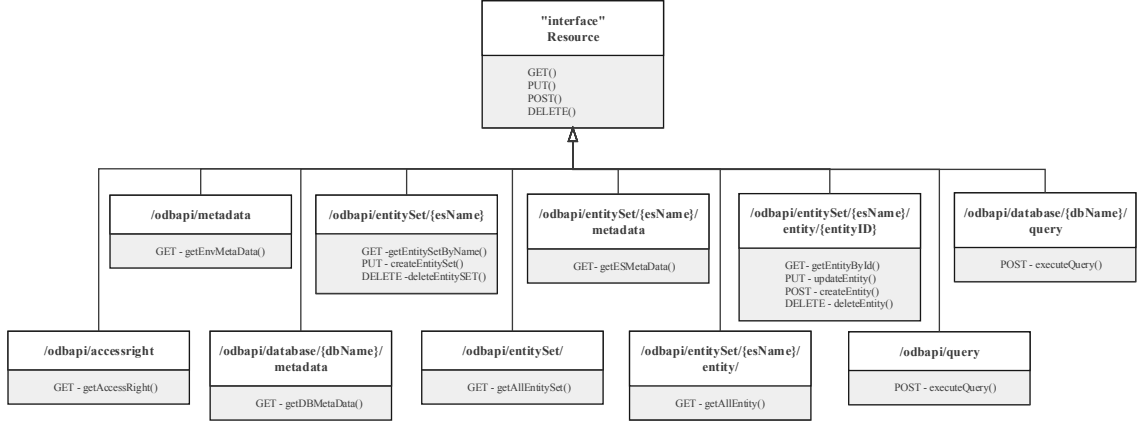


Figure 3.16: ODBAPI operations

In our specification, we consider two families of operations which inputs and outputs are JSON-based data. In the one hand, we have the family of operations that is dedicated to get meta-information about the resources using the *GET* REST method. In the other hand, we have the family of operations that allows to execute queries on resources of type *Database*, *EntitySet* or *Entity*. We start first by introducing four operations to manage meta-data about ODBAPI resources:

- **Get information about the user's access right:** This operation is provided by *getAccessRight* and allows a user to discover his/her access rights concerning the deployed data stores in a Cloud environment. To do so, the user must append the keyword *accessright* to his/her request.
- **Get information about a resource of type *Environment*:** This operation is ensured by *getEnvMetaData* and lists the information about a resource of type *Environment*. To execute this kind of operation, a user must provide the keyword *metadata* in his/her request. This keyword should be also present in the following two operations.
- **Get information about a resource of type *Database*:** A user can retrieve the information about a resource of type *Database* by executing the operation *getDBMetaData* and providing the name *dbName* of the target *Database*. This operation outputs information about a *Database* (e.g. duplication, replication, etc) and the *EntitySets* that it contains.
- **Get information about a resource of type *EntitySet*:** This operation is provided by *getESMetaData* and enables to discover the information about a resource of type *EntitySet* by giving its name *esName*. For instance, it helps the user to know the number of *Entities* that an *EntitySet* contains.

The second operations family represents the CRUD and complex queries executed on resources of type *Database*, *EntitySet* and *Entity*. In this context, ODBAPI provides the following operations:

- **Get an *EntitySet* by its *esName*:** By executing the operation *getEntitySetByName*, a user can retrieve an *EntitySet* by giving its name *esName*. It is ensured by the *GET* method.
- **Create an *EntitySet*:** The operation *createEntitySet* allows a user to create an *EntitySet* by giving its name *esName*. This operation is provided by the REST method *PUT*.
- **Delete an *EntitySet*:** An *EntitySet* can be deleted by using the operation *deleteEntitySet* and giving as input its name *esName*. It is ensured by the *DELETE* REST method.
- **Get list of all *EntitySet*:** A user can retrieve the list of all *EntitySet* by executing the operation *getAllEntitySet* and using the keyword *allES*. It returns the names of the *EntitySets* and several information (e.g. number of entities in each entity set, the type of database containing it, etc.).
- **Get an *Entity* by its *entityID*:** By executing the operation *getEntityById*, a user can retrieve an *Entity* by giving its identifier *entityID*. It is ensured by the *GET* method.
- **Update an *Entity*:** An *Entity* can be updated by using the operation *updateEntity* and its identifier *entityID*. It is ensured by the *PUT* method.
- **Create an *Entity*:** The operation *createEntitySet* allows a user to create an *Entity* by giving its identifier *entityID*. This operation is provided by the REST method *POST*.
- **Delete an *Entity*:** An *Entity* can be deleted by using the operation *deleteEntity* and giving as input its identifier *entityID*. It is ensured by the *DELETE* method.
- **Get list of all *Entities*:** A user can retrieve the list of all *Entities* of an *EntitySet* by executing the operation *getAllEntity* and using the keyword *allE*. It outputs the identifiers of the *Entities* and their contents.
- **Query one or multiple *EntitySets*:** A user can run a query across one or multiple heterogeneous *EntitySets* by executing the operation *POST* and using the keyword *query*. It outputs a new *EntitySet*. When executing this kind of query, one has to write his/her query in a JSON format using a SQL-like syntax. Indeed, the input JSON-based file contains three elements that are *select*, *from* and *where* and these elements are JSON Arrays of type String. A user can execute filtering queries across one *EntitySet* and complex queries across one or multiple *EntitySets*. A complex query can be a join,

union, etc. It is noteworthy that we consider this kind of queries as specific retrieve queries.

- **Query a specific *Database*:** A user can run a native query on a specific database in order to benefit from its expressive query language (e.g. a graph query). In this case, the query is just serialized in JSON and executed via the operation *POST* and using the keyword *query*. One can notice that in this case ODBAPI does not standardize the expression of the query but just the way it is issued.

3.4.2.3 Examples Of Queries

In this section, we give some examples of queries expressed using ODBAPI syntax. Indeed, we give first an example of unifying query execution. Then we present an example of a filtering query and an example of a join query.

Unifying query execution We present in this section three examples of the same operation targeting the three data stores of type CouchDB, MySQL and Riak named respectively *dblp*, *Person*, and *ConferenceRanking* and defined in the previous section (see Figure 3.3). This operation consists in retrieving an *Entity* by its *entityID*. Hence, we show how ODBAPI unifies the access to heterogeneous data stores to execute CRUD operations.

We give the first example which is a HTTP request and response of retrieving an *Entity* of type document illustrated below. User should specify the type of the HTTP method that is **GET** followed by the target resource `/odbapi/entityset/dblp/entity/111`. Added to that, he/she should precise the type of the target data store `database/couchDB` and the content type `application/json` that is an acceptable for the response. In our case, it is the JSON format. As a response to this request, we have the status code **200 OK** accompanied by the asked resource written in the JSON format. Reader should note that each output is written in JSON and starts with the element *data*.

```
> GET /odbapi/entityset/dblp/entity/111
> Database-Type: database/couchDB
> Accept: application/json

< HTTP/1.1 200 OK
< Content-Type: application/json
< {
<   "data":
<     [
<       {
<         "entityID": "111",
<         "author": ["Leslie Valiant"],
<         "year": "1999",
<         "title": "Robust Logics",
```

```
<           "conference": "STOC"
<         }
<       ]
<    }
```

We illustrate below the example that is a HTTP request and response of retrieving an *Entity* of type value. In this request, the user should specify the key `/odbapi/entityset/ConferenceRanking/entity/1` of the target value and the type of the target data store `database/Riak` that is the key value DBMS Riak. This request should return the status code 200 OK and the asked resource written in the JSON format.

```
> GET /odbapi/entityset/ConferenceRanking/entity/STOC
> Database-Type: database/Riak
> Accept: application/json

< HTTP/1.1 200 OK
< Content-Type: application/json
< {
<   "data":
<     [
<       {
<         "conference": "STOC",
<         "Rank": "A*"
<       }
<     ]
<   }
```

The last example is presented below and consists in a HTTP request and response of retrieving an *Entity* of type relational tuple. In fact, user should precise the HTTP method `GET` followed by the resource `/odbapi/entityset/person/entity/1` that he/she wants to retrieve. In addition, he/she should specify the type of the target data store `Database-Type: database/MySQL` that is the relational DBMS MySQL and the content type `application/json`. In this example, the response of this request is the status code 200 OK and the requested resource written in the JSON format.

```
> GET /odbapi/entityset/person/entity/100
> Database-Type: database/MySQL
> Accept: application/json

< HTTP/1.1 200 OK
< Content-Type: application/json
```



```
< {
<   "data":
<     [
<       {
<         "personId": "1",
<         "personName": "Leslie Valiant",
<         "personCountry": "United Kingdom",
<         "personAffiliation": "Havard University"
<       }
<     ]
<   }
< }
```

Example of filtering query In this section we give the example of a filtering query. In this kind of query we invoke the selection and projection operations in order to filter *Attributes* in an *EntitySet*. In the example below user precises the HTTP method **POST** followed by the URL `/odbapi/query`. In addition, he/she should specify the type of the target data store **Database-Type: database/VirtualDataStore** that is the VDS and the content type **application/json**. In the body of his/her query, the user specifies (1) in the element *select* the attributes *personName* and *personCountry* that he/she wants to project, (2) in the element *from* the name of the target *EntitySet* *person* and (3) in the element *where* the predicate of the selection that is *personId = 1*. The response of this request is the status code 200 OK and the answer is written in the JSON format.

```
> POST /odbapi/query
> Database-Type: database/VirtualDataStore
> Accept: application/json
> {
>   "select": ["personName", "personCountry"],
>   "from": ["person"],
>   "where": ["personId = 1"]
> }

< HTTP/1.1 200 OK
< Content-Type: application/json
< {
<   "data":
<     [
<       {
<         "personName": "Leslie Valiant",
<         "personCountry": "United Kingdom"
<       }
<     ]
<   }
< }
```

Example of a join query In this section we give the example of a join query between *person* and *dblp EntitySets*. The user precises the HTTP method **POST**, the URL **/odbapi/query** to inform the VDS about the type of query, and the type of the target data store **Database-Type: database/VirtualDataStore** that is the VDS. In the body of his/her query, the user expresses his/her query. Indeed, he/she specifies in the element *select* the attributes *personName*, *personCountry*, and *title* that he/she wants to project as a result to his/her query. Then, he/she expresses in the element *from* the name of the *person* and *dblp EntitySets* to join. Finally, he/she defines in the element *where* the predicate of the selection that is *personId < 3* and the logical expression of join *personName in author*. The response of this request is the status code 200 OK and the answer is written in the JSON format below. Answering this kind of queries will be tackled in the next chapter.

```
> POST /odbapi/query
> Database-Type: database/VirtualDataStore
> Accept: application/json
> {
>   "select": ["personName", "personCountry", "title"],
>   "from": ["person", "dblp"],
>   "where": ["personId <3", "personName in author"]
> }
```

```
< HTTP/1.1 200 OK
< Content-Type: application/json
< {
<   "data":
<     [
<       {
<         "personName": "Leslie Valiant",
<         "personCountry": "United Kingdom",
<         "title": "Robust logics"
<       },
<       {
<         "personName": "John Hopcroft",
<         "personCountry": "United States"
<         "title": "Manipulation-Resistant
<           Reputations Using Hitting Time"
<       }
<     ]
<   }
```

3.5 Conclusion

In this chapter, we presented our first two contributions. The first one represents the unified data model that allows to provide a unique view on relational and NoSQL data stores. Then we presented a global schema that is the unified data model enriched with a set of the refinement rules to allow complex queries expression and execution. Finally, we presented our query algebra defined based on the formal definition of the unified data model.

The second one is the unique REST API that enables the management of the described resources in a uniform manner based on our unified data model. This API is called ODBAPI and allows the expression and execution of CRUD and complex queries on relational and NoSQL data stores. It is designed to provide utmost control for the developer against heterogeneous data stores. ODBAPI eases the interaction with data stores at the same time by replacing an abundance of APIs. Moreover, it decouples cloud applications from data stores alleviating therefore their migration.

The proposed API is not meant as the silver bullet to solve all heterogeneity problems between relational and NoSQL data stores. However, we proved that we are aware of this problem and we are trying to alleviate this burden by unifying the execution of CRUD operations. We presented also how we propose to express complex queries such as join. But, we do not deny that it still remains the problem of executing this kind of queries that we try to tackle in the next chapter. Indeed, we will present the VDS component that we propose to support the optimization and evaluation of complex queries across relational and NoSQL data stores.

Virtual Data Stores For Query Evaluation and Optimization

Contents

4.1	Introduction	67
4.2	Use Cases And Motivation	68
4.3	Query Evaluation	69
4.3.1	Query Evaluation Principles	69
4.3.2	Query Execution Plan	70
4.3.3	Queries Parsing and Algebraic Optimization	72
4.3.4	Queries Annotation	72
4.4	Query Optimisation	74
4.4.1	Catalog	76
4.4.2	Cost Model	78
4.4.3	Optimal Execution Plan Generation	80
4.5	Conclusion	86

4.1 Introduction

Although ODBAPI solves the problem of heterogeneity between relational and NoSQL data stores by proposing a unique layer to execute different operations, it is unable to support complex queries execution (i.e. multi-data store queries). As a remedy to this, we propose in this chapter the virtual data store (VDS) component which supports the evaluation and optimization of complex queries execution across NoSQL and relational data stores. The VDS implements a dynamic programming based algorithm to generate optimal execution plan using a cost model.

In the upcoming sections, we introduce first a motivating example of a join query across three heterogeneous data store (see Section 4.2). Then, we present the principles of our solution to evaluate queries execution (see Section 4.3). Finally, we describe our solution to optimize complex queries execution using a cost model and the algorithm to generate an optimal execution plan (see Section 4.4).

4.2 Use Cases And Motivation

In this section, we motivate our solution to evaluate and optimize queries execution across relational and NoSQL data stores. Doing so, we propose to use the second use case presented in the previous chapter in Section 3.4.1.2 which is the case of the polyglot persistence. Indeed, we gave the example of the application A that interacts with a document data store called dblpDB, a relational data store called personDB, and a key/value data store called RankDB (see Figure 3.3 and Figure 3.5). Suppose now that application A needs at some point to retrieve the affiliation and the name of authors having at least a paper published in a conference ranked "A". This query may be expressed using ODBAPI syntax as follows:

```
> POST /odbapi/joinquery
> Database-Type: database/VirtualDataStore
> Accept: application/json
> {
>   "select": [
>     "person.personName",
>     "person.personAffiliation"
>   ],
>   "from": [
>     "person",
>     "dblp",
>     "ConferenceRanking"
>   ],
>   "where": [
>     "person.personName in dblp.author",
>     "dblp.conference = ConferenceRanking.conference",
>     "ConferenceRanking.Rank=A"
>   ]
> }
```

Answering such query is, nowadays, challenging since it involves a relational and two NoSQL data stores. In fact, since dblpDB, ConferenceRanking, and personDB use different data models, the developers have to identify the sub-queries by hand, interact with each data store separately and implement the join operation by themselves. This remains (1) purely programmatic, (2) not optimal and naif, and (3) time consuming. For this sake, we propose in this chapter a solution allowing the execution of complex queries expressed using ODBAPI syntax. The main component of this solution is the virtual data store (VDS). The key ingredients of this latter are the global schema (i.e. the unified data model and the correspondence rules), a catalog and a cost model. All these components are well introduced in the upcoming sections and represent the basis of our algorithm to generate the optimal execution plan of a complex query.

4.3 Query Evaluation

In this section, we introduce our approach to execute queries. Indeed, we present in Section 4.3.1 the principles of this solution. Afterward, we define in Section 4.3.2 the structure of the execution plan and the different operations constituting it. We end this section by giving an example of queries evaluation in Section 4.3.3.

4.3.1 Query Evaluation Principles

Compared to classical mediation architectures, our solution differs in that (1) we do not have a real global schema but just a collection of entity sets (i.e. the unified data model) and some refinement rules, (2) some data stores have poor query capabilities (no join support for example) and (3) some entity sets may be very large (several Gigabytes or more). Against this background, we propose a mediation based approach handling the execution of queries sent by multiple data stores based applications. We showcase in Figure 4.1 an overview of the architecture of this solution. Each input query is expressed using ODBAPI syntax and is processed by a single component which is the VDS. This latter acts as a mediator in a classical mediation architecture. It involves a catalog to evaluate and optimize queries execution (see Section 4.4.1), a global schema to re-write the queries (see Section 3.3.3), and an execution manifest to interact with the integrated sources. Each integrated data store is encapsulated by an ODBAPI service (i.e. it plays the role of the wrapper) capable to execute ODBAPI calls against a specific data store and to transform results into JSON structures. Considering our unified data model and the associated query algebra, we need a NoSQL data store supporting complex queries to implement the VDS. In our prototype, we choose CouchDB which is a JSON-based document data store with join capabilities.

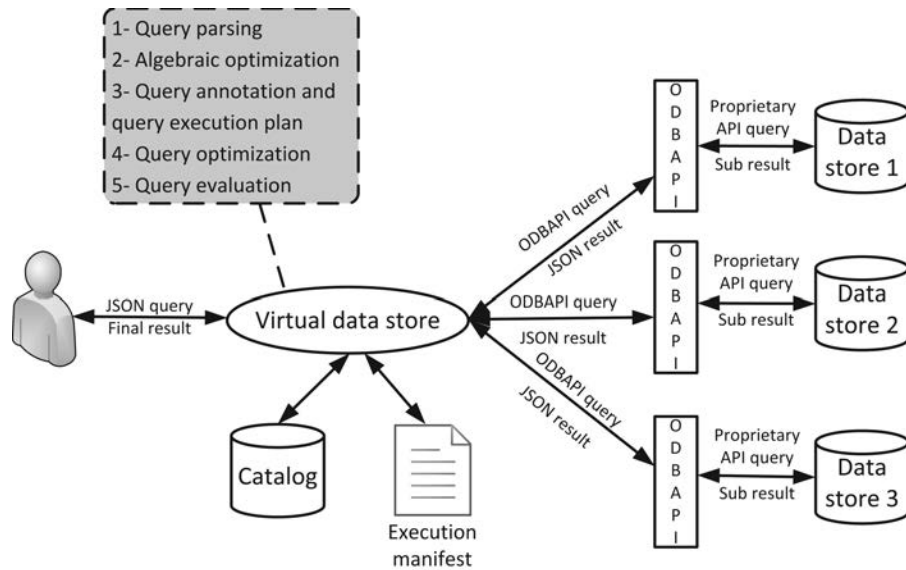


Figure 4.1: Mediation architecture

In our work, we consider that we can have three kinds of operations. Indeed, simple (i.e. CRUD) operations are directly executed by the target wrapper without any global optimization at the VDS level. In this case, the VDS just routes the query to the target wrapper which processes the query and transforms its result in JSON if needed before sending it back to the client application. Native complex queries are also directly executed by the target wrapper without the need of rewriting and optimization. Finally, complex queries are supported by the VDS that acts as a mediator and implements query optimization techniques. It analyzes the input query, splits it in sub-queries, sends them to the involved data stores through wrappers and combines/transforms the results before sending it back to the application.

The query optimization and evaluation process is composed of five steps (see the list of the steps in Figure 4.1). First, the query is parsed and represented by an algebraic tree composed of data stores and algebraic operators (see Section 3.3.4). Second, this tree is optimized using basic algebraic optimization by pushing unary operators towards data stores. Third, data stores in the tree are annotated by metadata extracted from the catalog (see Section 4.4.1). These annotations are used to transform the tree in a combination of several ODBAPI expressions on single data stores. An optimal combination is constructed using a cost function and dynamic programming (see Section 4.3.2). This latter is composed of several ODBAPI sub-queries expressed on a single data store and others sub-queries to recombine the partial results into the final one. Fourth, the optimization is done using a cost function [63] defined by a linear combination of the response time of the CPU, the time of the input/output, and the time of the communication or the data shipping ($Cost\ model = \alpha * t_{CPU} + \beta * t_{I/O} + \gamma * t_{COM}$). A calibration is needed to adjust the cost model to an actual infrastructure (see Section 4.4.2). Finally, the query is evaluated. To present in details our query evaluation and optimization process, we will focus especially on join queries but the same principles can be applied to other complex queries.

4.3.2 Query Execution Plan

The main step in evaluating queries in a mediation based solution is to generate a query execution plan. This latter represents a set of ordered operations that the VDS has to follow to interact with the integrated data stores in order to execute the sub-queries and return to the application the result set. An execution plan may be represented either graphically or textually. In our work, we propose to use the graphical representation through a tree where nodes denote the operations and edges represent the dependencies between two operations (see the example in Figure 6.8). It is noteworthy that leaves represent the first operations with which we should start and the root represents the last operation to execute. Hence, we can explain the notion of dependencies between operations in a query execution plan by the fact that an operation can be executed if and only if their two child operations finish their execution. An operation may be one of the following operations:

- **Push-down Operation:** This operation is a sub-query that can be eval-

uated on a single data store. In most cases, it is a simple combination of restriction and projection but it can involve more complex operations like joins if the data store is capable to compute them. It enables to reduce the volume of entity sets before executing other operations on another data store. In the query execution plan, this kind of operation represents the leaves.

- **Virtual Data Store Join (VDS Join) Operation:** This operation corresponds to a join operation executed in the VDS. It is a binary operation since it enables to join two entity sets.
- **External Join Operation:** This operation corresponds to a join operation delegated to one of the integrated data stores. It happens when a data store supporting join queries execution contains an entity set with a very large size. In this case, we prefer to migrate the other entity set to this data store and execute the join there because shipping big size of data is costly. It is worth noting that knowing the ability of a data store to execute a join query, the size of an entity set, etc. is provided through the annotations that we define below (see Section 4.4.1).
- **Projection and Selection Operations:** These operations are unary and denote a projection and selection respectively. They may be executed either in the VDS or in an integrated data store. These operations are very useful since they allow to reduce the size of data before transferring it from a site to another. In addition, they are used to construct the final result set.
- **Union and Intersection Operations:** This kind of operations represent the union and the intersection between two entity sets. They may be executed in either the VDS or an integrated data store.
- **Extract-transform-Ship-load (ETSL) Operation:** Unlike the other operations, this operation is particular since its role is to prepare an entity set to be the input of one of the operations introduced above. It represents the fact of (1) extracting an entity set from its location, (2) transforming the format of an entity set to adapt it to the format of the data store where the operation will be executed, (3) shipping an entity set from its location to the data store where the operation will be executed, and (4) loading an entity set in the data store where the operation will be executed.

For the moment, we do not define operations for computing nest and unnest operations. The addition of these operations would complexify the optimization process. It is considered as a future work.

In order to optimally execute a query, we have to define an optimal query execution plan which is not resource hungry and costly in time. For this purpose, we find several mechanisms for electing and constructing the optimal one. In section 4.4.3, we present our algorithm to generate the optimal query execution plan. In addition, we propose to enrich this latter by annotating each node (i.e. operation) using either a set of information from the VDS catalog (see Section

4.4.1) or some computed parameters based on the catalog also. These annotations are very useful since they enable to obtain more information about an operation and its output entity set. For instance, we can save the cost of an operation, the cost of generating the resulted EntitySet, etc. The purpose of these annotations is twofold. First, they maintain required information to execute a query with respect to the query execution plan. Second, they contain information essential for the proper functioning of our algorithm to generate the optimal query execution plan and especially to generate a subsequent operation.

4.3.3 Queries Parsing and Algebraic Optimization

In this section, we use the example of the join query between three entity sets presented in Section 4.2. Once the VDS receives the query, it constructs the corresponding algebraic tree (see Fig. 4.2a). This tree is naively constructed from the query without any optimization. Then, it is optimized using algebraic rewriting rules to privilege the execution of unary operations first in the integrated data stores in order to reduce the size of the transferred entity sets (see Fig. 4.2b).

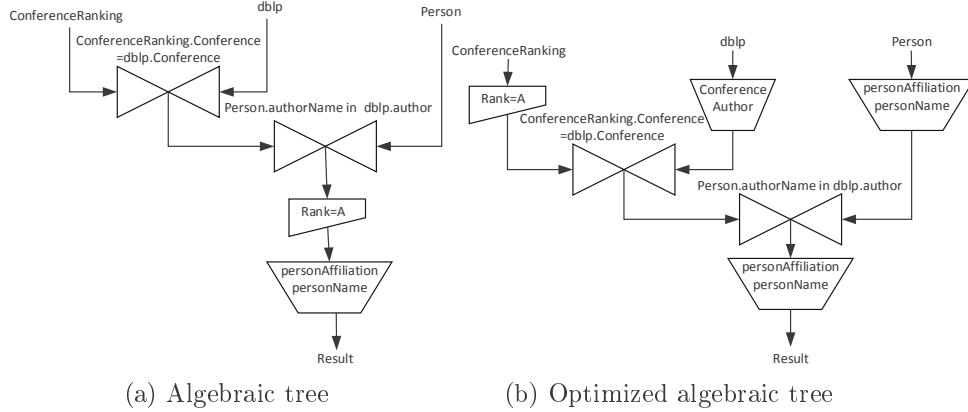


Figure 4.2: Examples of algebraic trees

4.3.4 Queries Annotation

In the following, we use two scenarios involving different data stores and annotations for our join query above:

- **Scenario 1 - A join between two entity sets in the same data store:** Let us consider just for this scenario that the two entity sets *dblp* and *conference ranking* are in the same data store of type *MySQL* and the *Person* entity set is a document data store of type *MongoDB*.
- **Scenario 2 - A very large size of the relational entity set:** Let us consider now that the entity sets *person*, *dblp*, and *conference ranking* belong to a relational data store *MySQL*, a document data store *MongoDB* and a

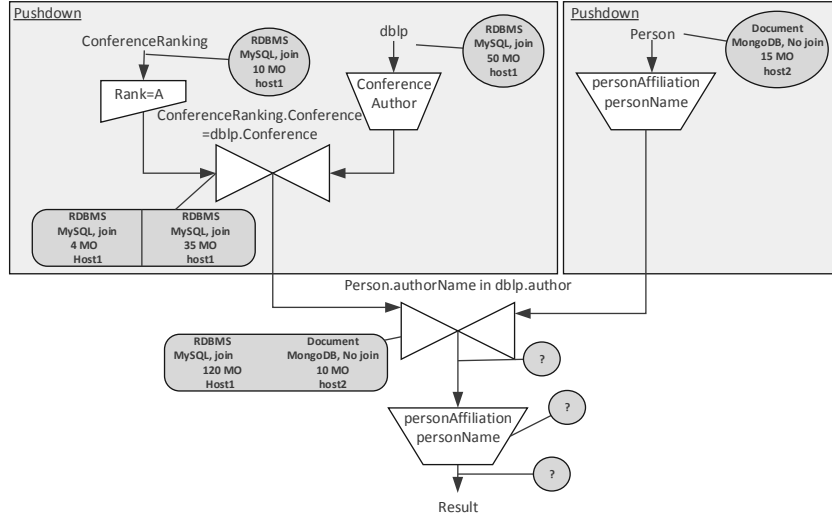


Figure 4.3: Annotated algebraic tree according the Scenario 1

key value data store *Riak* respectively. The size of the relational entity set is very large.

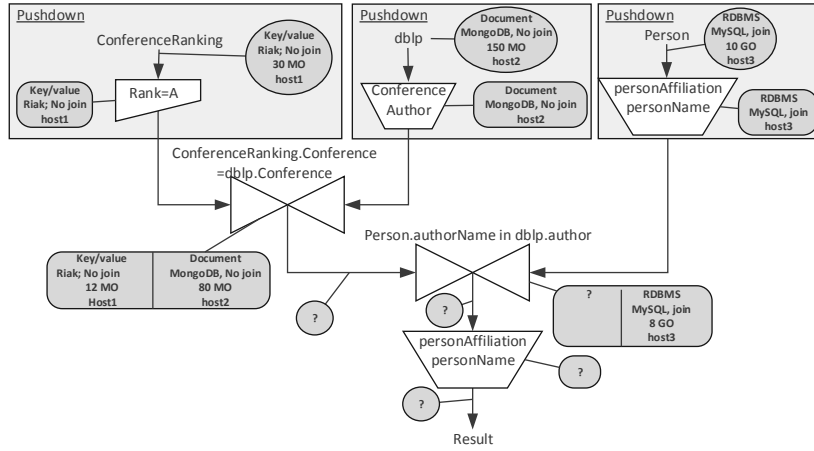


Figure 4.4: Annotated algebraic tree according the Scenario 2

After some preliminary optimization made on the algebraic query, the VDS annotates the algebraic tree with metadata about the target DBMS and data got from the catalog (see Fig. 4.3 for the scenario 1 and Fig. 4.4 for scenario 2). The annotations are put on the leaves of the tree (the entity sets that are on the top of the algebraic trees) and are propagated to the other nodes using estimation models for statistics (for example to estimate the output size of a join). For example, for the algebraic tree of scenario 1 in Fig. 4.3, we depict that the join between the entity sets *ConferenceRanking* and *dblp* can be done on a MySQL RDBMS located on *host1* (i.e. both operands are on the same data store and the same node) and has an estimated output size of 4 Mbytes. Whereas the algebraic tree of scenario

2 in Fig. 4.4, we can see that the same join can not be done on the same data store because these inputs came from two different data stores (a Riak DBMS and a MongoDB DBMS).

This annotated tree will be transformed into an optimized query execution plan in two steps. The first step will extract the sub-queries that can be evaluated on a single data store. The principle here is to maximize the work done by the integrated data stores. This extraction is done based on the annotations defining the location of the entity sets. At the end of this step, two situations are possible that we present below:

- **There is a single extracted sub-query.** It corresponds to the case of a query involving just one entity set, or a query involving two or multiple entity sets located in the same data store (if this data store supports the execution of complex operations i.e. join, union, etc.). The VDS automatically routes the query to the target wrapper and it will be executed by the target store itself. Then, the wrapper returns the final result to the VDS in order to answer the application. If the data store does not support complex operations, the complex operations will be executed by the VDS before sending back the results to the application.
- **There are several extracted sub-queries.** It corresponds to the case of a query involving two or multiple entity sets in two or multiple data stores. In this situation, we have to determine where and how to recombine the partial results. Both scenario 1 and 2 correspond to this situation.

The second step consists in constructing the optimal query execution plan able to recombine the partial results and to determine on which node evaluating each recombining operation (the VDS or the specific node storing a very large entity set). In addition to the algebraic operations, a query execution plan can use an other *ETSL* operation that enables the transfer of result set and the conversion of the result set. In fact, when we have binary operations with inputs coming from different data stores, we have to move and possibly convert at least one input set.

In Fig. 6.8, we showcase the optimal execution plan for the scenario 2. The size of the relational data store is very large (8 Gbytes). In this case, the VDS is not the best candidate to recombine partial results. It is better to maximize the work done by the relational data store, including recombining partial results using join operations, even if it will pay the cost of conversion. In fact, this cost is negligible compared to the cost of join query execution and large data transfer. Once the optimal query execution plan is constructed, it can be evaluated. For the moment we just consider synchronous evaluation of this plan, that is an operator can not be evaluated until all its inputs are present (we do not support on the fly evaluation). The evaluation consists in the query execution plan traversal.

4.4 Query Optimisation

In order to optimize queries execution, we fix the following list of principles:

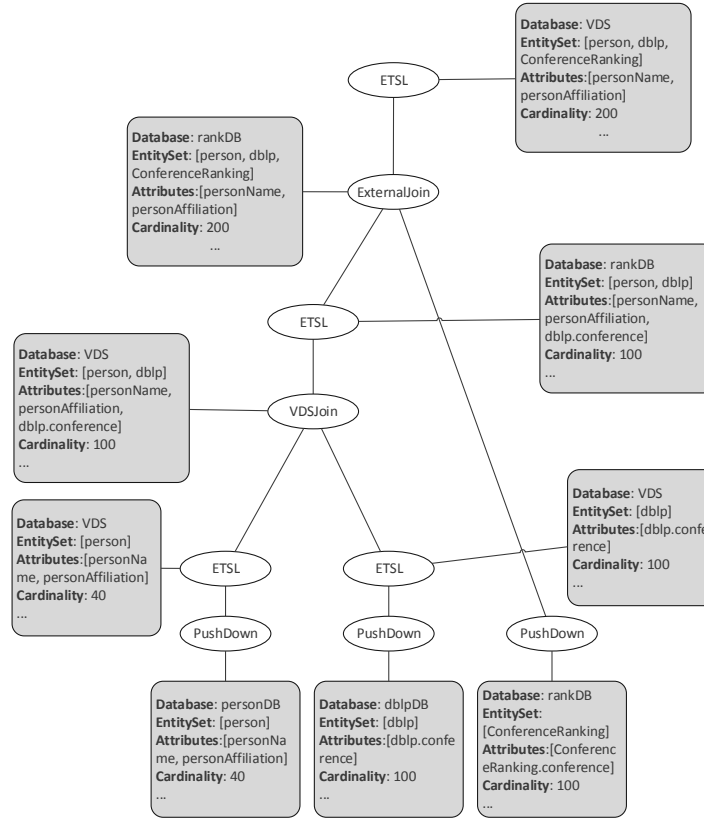


Figure 4.5: Execution plan for the Scenario 2

- Preserving the autonomy of the integrated data stores,
- Maximizing the work done by the integrated data stores,
- Optimizing a query execution plan following two possible strategies: (i) using the VDS only to evaluate sub-queries across multiple data stores, and (ii) using the VDS and powerful data stores (i.e. a data store is considered powerful if it is able to evaluate complex queries) in order to avoid shipping data having big size,
- Defining a cost function to capture the two possible strategies,
- Using dynamic programming approach based optimization,
- Applying no parallelism at an integrated data store (i.e. we suppose that a data store executes one query at the same time).
- Applying pipeline execution (i.e. an operation can start its execution when its operands are completely computed)

In the upcoming sections, we present our approach to optimize join queries execution across relational and NoSQL data stores. The key ingredients of this

solution are the catalog that we define in Section 4.4.1 and the cost model that we introduce in Section 4.4.2. Using these two components, we present in Section 4.4.3 our algorithm to generate the optimal execution plan.

4.4.1 Catalog

In this section, we introduce the parameters of our catalog. They include the type (i.e. relational, document, etc.) and the name of a data store, its location, its capabilities (support of join queries for example), the size of the entity sets, etc. Some of these information are either exported by the integrated data stores using their proprietary APIs or estimated using some probing queries. In addition, we can find some computed parameters that are calculated based on the collected information. In Table 4.1, we introduce the main parameters that we use to define our cost model. The showcased parameters represents in general the cost of an operation. It is noteworthy that a parameter depends at least on one of the following three costs that are the cost of the CPU, the cost of the input/output, and the cost of the communication. If it depends on two or more costs, it is represented by a vector; otherwise it is represented by a scalar.

Catalog's parameters	Specification
initProjection (resp. initSelection)	This parameter defines the initialization cost of the sources to execute a projection (resp. a selection). It is a scalar.
initJoin	This parameter defines the initialization cost of the sources to execute a join query. It is a scalar.
initUnion (resp. initIntersection)	This parameter defines the initialization cost of the sources to execute a union (resp. an intersection). It is a scalar.
initETSL	This parameter defines the initialization cost of the sources to execute an operation of type ETSL. It is a scalar.
canJoin	This parameter enables to denote if a data store supports the join queries execution or not. It is a boolean parameter. It is exclusively defined for the integrated sources since the VDSs are defined to take in charge join queries execution.
load	This parameter defines the cost of data loading. It differs from one data store to another since it depends on the data store's mechanism to load data. This parameter is a scalar since it only involves the cost of the Input/Output to the disc.
scan	This parameter denotes the cost to scan data in an entity set. This parameter depends on the type of the data store that contains the involved entity set in

Catalog's parameters	Specification
	an operation. It is a scalar since it only involves the cost of the Input/Output to the disc.
ship	This parameter represents the cost of data shipping. It concerns transferring data from a data store to the VDS, from the VDS to a data store, and from a data store to another. It is considered as a scalar since it only involves the communication cost.
convert	This parameter represents the cost of converting data. It consists in transforming data from JSON format to the proprietary format of data stores and vice versa. It is considered as a scalar since it only involves the CPU cost.
projection (resp. selection)	This parameter represents the cost of a projection (resp. selection). The projection (resp. selection) may inside/outside the VDS. It is a vector including the costs of the Input/Output and the CPU.
union (resp. intersection)	This parameter represents the cost of a union (resp. intersection). The union (resp. intersection) may inside/outside the VDS. It is a vector including the costs of the Input/Output and the CPU.
join	This parameter represents the cost. The join operation may be an external join or a VDS join. It is a vector including a cost of the Input/Output and the CPU.
cardinality	This parameter denotes the number of entities in an entity set. It is a scalar.
selectivity	This parameter defines the selectivity of attributes and it depends on an operation. It is bound between 0 and 1. When the operation is a selection, it denotes an attribute. However when it consists in a join operation, it involves two attributes that are defined by the correspondence rules. It is a scalar.
averageLengthAttribute	This parameter represents the average length of an attribute in all entities in an entity set. It is a scalar.
averageLengthEntity	This parameter represents the average length of an entity in an entity set. It is computed by summing the average length of each attribute constituting this entity. It is a scalar.
α, β, γ	These parameters represent the coefficients that we consider in our cost model to evaluate the cost of the CPU, the cost the input/output, and the communication

Catalog's parameters	Specification
	cost. These of parameters are separately defined for each data store.

Table 4.1: Specification of our catalog's parameters

4.4.2 Cost Model

Based on our catalog's parameters, we define a set of cost formulas constituting our cost model. These formulas are tailored for each operation defined in Section 4.3.2. Then, they are implemented by the algorithm of optimal execution plan generation that we present in the next section.

The cost of an execution plan is the sum of the cost of each operation (i.e. node in the execution plan) composing an execution plan. It is worth noting that a cost does not directly represent the time. Of course, a larger cost implies a larger time. It is used to compare two query execution plans but not to directly estimate a response time. To evaluate a cost formulas, we compute the matrix multiplication between a row vector containing the coefficients α , β , and γ , a column vector containing the values of the parameters defined in the catalog, and a constant variable called *const* which is a scalar and can be a cardinality, selectivity, etc. In addition, if a parameter does not depend on a given metric (CPU cost, Input/Output cost, or Communication cost), this latter will take a null value in the column vector. The matrix multiplication is computed as follows:

$$const \begin{pmatrix} \alpha & \beta & \gamma \end{pmatrix} \begin{pmatrix} t_{CPU} \\ t_{I/O} \\ t_{COM} \end{pmatrix} = const * (\alpha * t_{CPU} + \beta * t_{I/O} + \gamma * t_{COM})$$

Let us start by the cost formulas of the projection and selection operations that we denote by *costProjection* and *costRestriction* respectively (see Formula 4.1 and Formula 4.2). The projection (resp. selection) operation is the linear combination of the parameters *initProjection* (resp. *initSelection*), *scan*, and *projection* (resp. *selection*). The *costProjection* (resp. *costRestriction*) takes as input the parameter *S* to denote either an integrated data store or a VDS where the join will be executed, the variable *L* to denote the total size of the entity sets in entry, and an estimation function that we call *estimate*. This function enables to compute how many time the projection (resp. selection) will be executed elementary. It invokes the selectivity and the cardinality parameters.

S: a node, L: length of an entity set, att: projection attributes

$$\begin{aligned} costProjection(S, L, estimate(att, L)) = & initProjection(S) + scan(S) * L + \\ & projection(S) * estimate(att, L) \end{aligned} \tag{4.1}$$

S : a node, L : length of an entity set, $cond$: restriction condition

$$costRestriction(S, L, estimate(cond, L)) = initRestriction(S) + scan(S) * L + restriction(S) * estimate(cond, L) \quad (4.2)$$

Then we define a cost formula for the join operation that we refer to as *costJoin* (see Formula 4.3). It involves the external join or the VDS join. This operation is the linear combination of the parameters *initJoin*, *scan*, and *join*. The *costJoin* takes as input the parameter S to denote either an integrated data store or a VDS where the join will be executed, the variable L to denote the total size of the entity sets in entry, and an estimation function that we call *estimate*. It involves the join condition and the selectivity parameter. At best, the value of this estimation should be equal to the cardinality of the resulted entity set. At worst, the value of the estimation is equal to the cardinality of the Cartesian product of the two entity sets to join.

S : a node, L : length of an entity set, $cond$: join condition

$$costJoin(S, L, estimate(cond, L)) = initJoin(S) + scan(S) * L + join(S) * estimate(cond, L) \quad (4.3)$$

Afterward, we define a cost formula for the union operation that we call *costUnion* (see Formula 4.4). This operation is the linear combination of the parameters *initUnion*, *scan*, and *union*. It takes as input the parameter S to denote either an integrated data store or a VDS where the union will be executed and the variable L to denote the total size of the entity sets in entry.

S : a node, L : length of an entity set

$$costUnion(S, L) = initUnion(S) + L * (scan(S) + union(S)) \quad (4.4)$$

We define also a cost formula for the intersection operation that we call *costIntersection* (see Formula 4.5). It takes as input the parameter S to denote either an integrated data store or a VDS where the intersection will be executed, the variable L to denote the total size of the entity sets in entry, and an estimation function that enables as to compute how many time the intersection operation will be executed.

S : a node, L : length of an entity set

$$costIntersection(S, L, estimate(L)) = initIntersection(S) + scan(S) * L + intersection(S) * estimate(L) \quad (4.5)$$

Finally, we introduce the *ETSL* operation. This operation can be occurred in three situation depending on the type of the operation that precedes it, the forwarding data store and the addressee data store. It is worth noting that a data store may be either a VDS or an integrated data store. Indeed, these situations are: (1) in the case of a VDS join, the forwarding data store is an integrated one and the addressee data store is a VDS (see Formula 4.6), (2) in the case of an external join, the forwarding data store is a VDS and the addressee data store is an integrated one (see Formula 4.7), and (3) in the case of an external join, the forwarding data store and the addressee data store are integrated (see Formula 4.8). In the following, we introduce the three cost formulas for the *ETSL* operation with respect to each scenario. These formulas are a linear combination of the parameter *initETSL*, *scan*, *convert*, *ship* and *load* defined in the catalog. We use also the variable *N* which is the volume of data that we wish to transfer during an operation and it is computed as follows $N = cardinality * averageLengthEntity$. Finally, the variable *VDS* and *S* denote a VDS and an integrated data store respectively.

S: an integrated data store, *VDS*: a VDS, *N*: Volume of data

$$costETSL(S, VDS, N) = initETSL(S) + N * (scan(S) + convert(S) + ship(S) + load(VDS)) \quad (4.6)$$

VDS: a VDS, *S*: an integrated data store, *N*: Volume of data

$$costETSL(VDS, S, N) = initETSL(VDS) + N * (scan(VDS) + convert(S) + Ship(S) + load(VDS)) \quad (4.7)$$

$S_{forwarding, addressee}$: integrated data stores, *N*: Volume of data

$$costETSL(S_{forwarding}, S_{addressee}, N) = initETSL(S_{forwarding}) + N * (scan(S_{forwarding}) + convert(S_{forwarding}) + ship(S_{forwarding}) + convert(S_{addressee}) + ship(S_{addressee}) + load(S_{addressee})) \quad (4.8)$$

4.4.3 Optimal Execution Plan Generation

In this section, we introduce our algorithm to generate the optimal execution plan. The highlights of our algorithm are twofold. First, we propose to define our algorithm based on the dynamic programming method. This latter is proposed to solve complex optimization problems by dividing one problem into a set of sub-problems that are easier to solve. The optimal solution of the complex problem is constructed in a bottom-up way from the set of the optimal solutions of the sub-problems. In our work, we apply this method to elect the optimal execution plan which is represented as a tree (see Section 4.3.2). In fact, we start by constructing the leaves (which are the first optimal sub-plans) and every time we go one level

up in the execution plan by adding a new operation (which is a root in the new optimal sub-plan). We proceed like that until obtaining the optimal execution plan of a query. Second, we introduce parallelism between nodes but not inside a node. Two operations which are planned on distinct nodes can be parallelized but if they are planned on the same node they cannot. Indeed, the fact of parallelizing operations in one data store is very resource hungry and it may hamper the node functioning.

In order to evaluate the total cost of an operation without scanning the whole sub-plan, we define three parameters to calculate the cost of its creation. This enables us to avoid doing the same work every time we add a new operation. These parameter are:

- **NodeCost**: It represents the total time of allocating the a node to execute queries in a sub-plan. The value of the *NodeCost* of an operation is the sum of the *NodeCost* of the children operations in the execution plan and the cost of the operation itself. The *nodeCost* of a push down operation is null.
- **WaitBegin**: It denotes the waiting time of a VDS before executing the first operation in a sub-plan, that is at the level of the push down operations.
- **WaitLate**: It defines the total waiting time of a VDS between the first operation execution and the last one.

These parameters are implemented in our algorithm and they are also used to annotate each node in the execution plan. In Figure 4.6, we showcase an example of two children operations referred to as *Left operation* and *Right operation* and a parent node called *Parent operation*. Each operation is annotated with *NodeCost*, *WaitBegin*, *WaitLate*, and *TotalCost*. This latter contains the sum of the three other parameters. To compute the value of the *NodeCost* of an operation, we have two possible cases (i.e. parallelization of operations execution or not). Hence, we can note (Reader should notice that the *NodeCost* is null when the operation is of type push down):

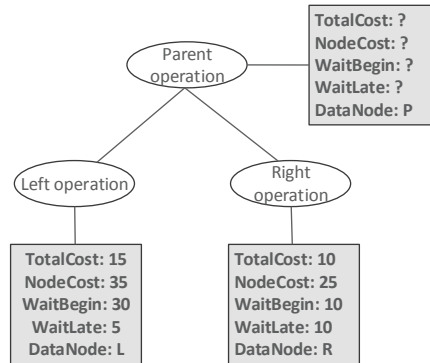


Figure 4.6: Example of computing an operation cost

$$NodeCost_{Parent\ operation} = \begin{cases} \begin{aligned} &cost_{Parent\ operation} \\ &+NodeCost_{Right\ operation} \\ &+NodeCost_{Left\ operation} \end{aligned} & \text{if } L = R \text{ (sequential execution)} \\ \begin{aligned} &cost_{Parent\ operation} \\ &+Max(NodeCost_{Right\ and\ Left\ operations}) \end{aligned} & \text{if } L \neq R \text{ (parallel execution)} \end{cases}$$

Then, we compute the *WaitBegin* parameter by summing the values of the *WaitBegin* parameters of the children nodes (namely *Left* and *Right operations*). It is computed as follows:

$$WaitBegin_{Parent\ operation} = WaitBegin_{Right\ operation} + WaitBegin_{Left\ operation}$$

Finally, we compute the *WaitLate* parameter according to the situation of the node where the *Parent operation* will be executed. In fact, we check if the source is available to the execution of another operation or not. Hence, we can define two possible way to calculate the *WaitLate* parameter (Reader should notice that the parameter *delay_{Parent operation}* represents the waiting time of the release of a working node):

$$WaitLate_{Parent\ operation} = \begin{cases} \begin{aligned} &WaitLate_{Right\ operation} \\ &+WaitLate_{Left\ operation} \end{aligned} & \text{if the node is available} \\ \begin{aligned} &WaitLate_{Right\ operation} \\ &+WaitLate_{Left\ operation} \\ &+delay_{Parent\ operation} \end{aligned} & \text{if not} \end{cases}$$

In Algorithm1, we introduce our algorithm to generate the optimal execution plan. This algorithm takes as input a query (in our case a join query) and it returns as output the optimal execution plan of this query. For ease of presentation, we propose to define two main steps in the algorithm that we introduce in the following:

- **Initialization step (lines 3-24):** In this step, we parse the query and we extract all the required information to generate the optimal execution plan. We start by focusing on the clause SELECT in order to extract each attribute is this clause and identify the entity set containing it (lines 3-6). Then, for each entity set in the clause FROM, we create a new node in the execution plan (lines 7-10). This node represents a leaf in the plan and it denotes an operation of type push-down. It is worth noting that we create a node of type push-down for each entity set even though there is no unary operation to execute on it. Finally, we focus on the clause WHERE (lines 11-23). For each condition in this clause, we check its kind and we decide about the work that

we will apply to it. Indeed, we have three possible kinds of conditions. First, the condition may be a join condition between two entity sets co-located in the same data store and this latter supports join queries execution, hence we merge the two push-down nodes of the appropriate entity sets in one push-down node and we add the join operation to it. Second, the condition may be a join query in general (i.e. a VDS join or an external join in the case of a big data scenario), so we save it as a join condition and we will use it afterward during the plan generation. Third, the condition may be a selection on an entity set, hence we modify the node push-down assigned to this entity set by delegating the execution of the selection operation to it.

- **Optimal execution plan generation (lines 24-45):** In this step, we construct the optimal execution plan following the dynamic programming method. We start by sub-plans constructed in the initialization step which are stored in the variable *initNodes* (line 24). Then, we construct the sub-plans containing *i* join conditions (line 25). To do so, for each join condition (line 26), we take two sub-plans *p1* and *p2* (lines 27-28) and we check if we can join them through the condition (line 29). If it is the case, we add this condition to the new sub-plan joining *p1* and *p2* (line 30) and we create the root node of this plan. Indeed, the root may represent either a VDS join operation (lines 31-32) or an external join operation (lines 33-40). The second case depends on whether the data store (i.e. a VDS or an integrated data store) containing the entity set supports join queries execution or not. It is worth noting that every time we add a new node in the plan, this node is automatically annotated with information from the catalog and the values of the parameters *NodeCost*, *WaitBegin*, *WaitLate*, and *TotalCost*. In addition, a node of type ETSL is added after each operation of type VDS join, external join and push-down. Based on these annotations, we prune non optimal sub-plans, and sub-plans co-existing in the same data store that have the same cost or are similar in term of execution scenario. A plan is considered optimal, if it has the smallest cost when we apply our cost model.

In the rest of this section, we propose to run the Algorithm1 on the join query introduced in Section 4.2. The first step in our algorithm is the initialization. Indeed, we create three nodes of type push-down (see Figure 4.7). The first one is for the entity set *person* which is stored in the database *personDB*. We precise also the attributes that we need to answer the query that are *personName* and *personAffiliation*. The second one corresponds to the entity set *dblp*. The third one denotes the entity set *conferenceRanking*. It is noteworthy that in this node we fill the parameter *where* with a selection condition which is *Rank = A*. This enables us to maximize the work done by the integrated data stores before executing VDS joins operation.

As we have two join conditions in our join query, we will construct the sub-plans in two times. In Figure 4.8, we introduce the first iteration of the second step of our algorithm which is the generation of the optimal execution plan. Indeed, we construct three sub-plans. The first one allows to execute a VDS join operation

Algorithm 1 Generation of the optimal execution plan

```

1: input  $q$ : A query
2: output plan: The optimal execution plan of the query  $q$ 
3:  $selectAttributes \leftarrow \emptyset$ 
4: while (exist(attribute  $A$  in the clause SELECT)) do
5:    $selectAttributes.add(entitySetOf(A), A)$ 
6: end while
7:  $initNodes \leftarrow \emptyset$ 
8: while (exist(entitySet  $ES$  in the clause FROM)) do
9:    $initNodes.createPushdown(ES, selectAttributes.get(ES))$ 
10: end while
11:  $JoinConditions \leftarrow \emptyset$ 
12: while (exist(condition  $C$  in the clause WHERE)) do
13:    $entitySetLeft \leftarrow siteOfLeftEntitySet(C)$ 
14:    $site \leftarrow siteOf(entitySetLeft)$ 
15:   if  $isJoinCondition(C)$  then
16:     if  $site == siteOf(RightEntitySet(C))$  and  $canJoin(site)$  then
17:        $initNodes.merge(entitySetLeft, RightEntitySet(C))$ 
18:     else
19:        $JoinConditions.add(C)$ 
20:     end if
21:   else
22:      $initNodes.addRestrictionCondition(entitySetLeft, C)$ 
23:   end if
24: end while
25:  $plans \leftarrow initNodes$ 
26: for  $i = 1$  to number of join conditions do
27:   for each condition  $C$  in  $JoinConditions$  do
28:     for each plan  $p1$  in  $plans$  do
29:       for each plan  $p2$  in  $plans$  do
30:         if  $match(p1, p2, C)$  then
31:            $newConditions.add(conditionsOf(p1), conditionsOf(p2), C)$ 
32:            $node \leftarrow createVDSJoin(p1, p2, C)$ 
33:            $plans.add(node)$ 
34:           if  $canJoin(siteOf(p1))$  then
35:              $node \leftarrow createExternalJoin(p1, p2, C)$ 
36:              $plans.add(node)$ 
37:           end if
38:           if  $canJoin(siteOf(p2))$  then
39:              $node \leftarrow createExternalJoin(p2, p1, C)$ 
40:              $plans.add(node)$ 
41:           end if
42:         end if
43:       end for
44:     end for
45:   end for
46:    $prune(plans)$ 
47: end for

```

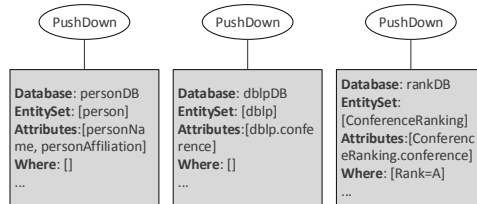


Figure 4.7: The initialization step

between the entity sets *person* and *dblp* using the join condition *personName in authors* (see the arrows labeled with the number 1). The two other sub-plans enables the execution of join query between the entity sets *dblp* and *conferenceRanking* using the condition *dblp.conference = ConferenceRanking.conference* (see the arrows labeled with the number 2). In fact, we can execute either an external join in the data store *rankDB* or a VDS join.

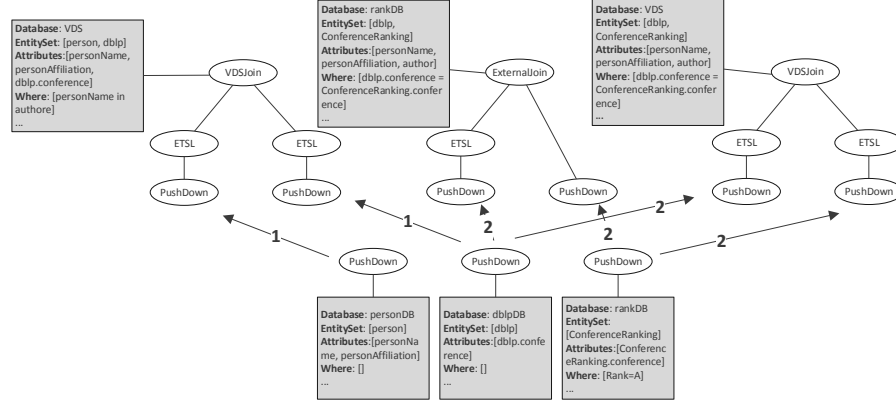


Figure 4.8: The first iteration

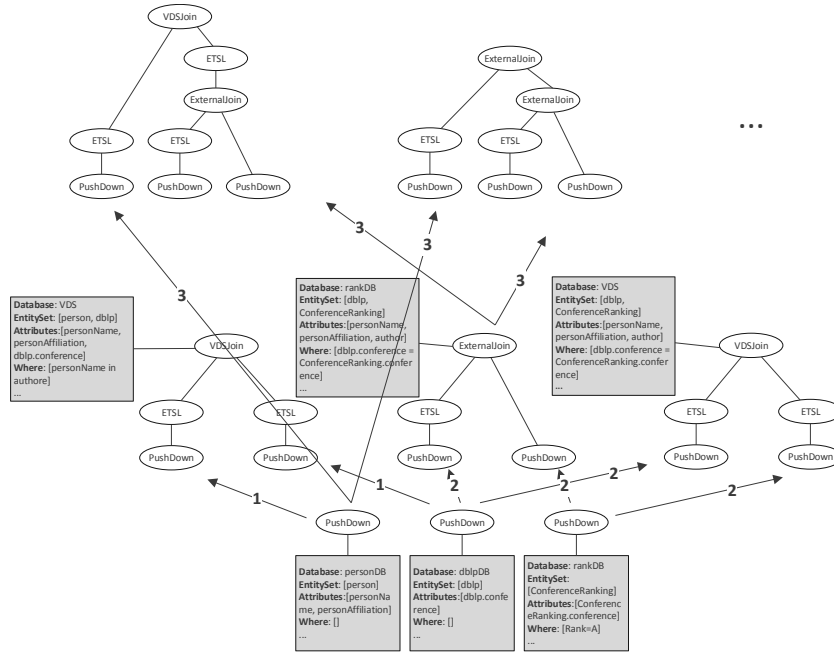


Figure 4.9: The second iteration

Once we have covered all the join conditions, we move to the second and last iteration to finalize and elect the optimal execution plan. In Figure 4.9, we showcase this step using the arrows labeled with the number 3. Indeed, we give just

the example of two plans in which we illustrate the execution of a join operation using the join condition *personName in authors* after having executed the join between the entity sets *dblp* and *conferenceRanking*. We present that we can execute it using either an external join in the *rankDB* or a VDS join.

4.5 Conclusion

In this chapter, we proposed the VDS component to execute complex queries (especially joins) across NoSQL and relational data stores. For this purpose, we defined a unified data model able to describe the heterogeneous data models of data stores. It is used by the user to express his/her complex query and by the VDS to process it. Once a VDS receives a complex query, it evaluates the queries based on a set of strategies and it optimizes the queries execution using a cost model. This latter is implemented in the optimal execution plan generation algorithm.

In the next chapter, we will introduce our last contribution. In this work, we define a manifest based algorithm to discover Clouds' data stores and deploy a multiple data stores-based application in an elected Cloud environment.

Automating Resources Discovery And Deployment For Multiple Data Stores Applications In Cloud Environment

Contents

5.1	Introduction	87
5.2	Use Cases And Motivation	88
5.3	Automatic Discovery Of Cloud Data Stores	88
5.3.1	Automatic Discovery Principles	89
5.3.2	Abstract Application Manifest	89
5.3.3	Offer Manifest	91
5.3.4	Matching Algorithm	91
5.4	Automatic Deployment Of Multiple Data Stores Based Applications	94
5.4.1	An Extension Of The Compatible One Application and Platform Service API	95
5.4.2	Deployment Manifest	97
5.5	XML-Based Manifest Examples	97
5.6	Conclusion	101

5.1 Introduction

In this contribution, we consider two components, the discovery and deployment modules, responsible of finding appropriate Cloud environments and deploying multiple data stores based applications on them respectively. Developers first express their requirements about the used data stores as well as the computation environment via an abstract application manifest. Based on this manifest, the discovery component implements our matching algorithm to produce an offer manifest, and select the appropriate cloud environment. A deployment manifest is constructed as a result to that. This manifest will be in turn used by the

deployment component to deploy the application on that selected environment. The deployment process is ensured by COAPS API that we extended to support multiple data stores based applications. The discovery and deployment modules relieves the application developers from the burden of dealing with different APIs and discovery/deployment procedures.

In this chapter, we motivate our contribution in Section 5.2. Then, we present in Section 5.3 our solution to automate Clouds data stores discovery. Afterward, we introduce in Section 5.4 our approach to automate multiple data stores-based applications in cloud environment. Finally, we give in Section 5.5 two examples of manifests.

5.2 Use Cases And Motivation

In this section, we rely on the previous motivating examples presented in Section 3.4.1 of Chapter 3 to motivate our contribution in this chapter. Indeed, in both cases (in the case of either the application migration or the polyglot persistence), the developer should discover all data stores capabilities of the available Cloud providers and should choose the most suitable cloud provider to his/her application's requirements. Then, the developer deploys his/her application in the elected Cloud provider.

At first glance, this seems trivial and easy; however the developer must support the discovery and the deployment of his/her application since it is manually done without any automation. Against a plethora of cloud providers, developer may fail or omit to take into account a cloud provider or one of its services during the discovery process. In addition, developers should first deploy the environment of their applications (i.e. the required data stores allocation, an application container allocation, etc.) prior to its application deployment. However this is actually a tedious task because even if we find some automated solutions for that, it is rather intended to deploy single data store based applications.

In this chapter, we tackle these problems and we propose a declarative approach for discovering appropriate cloud environments and deploying multiple data stores based applications on them while letting developers simply focus on specifying their storage and computing requirements. One can note that even if all these steps are automated, some of them can be done manually if unnecessary.

5.3 Automatic Discovery Of Cloud Data Stores

In this section, we present our logic to automatically discover the capabilities of Cloud providers' data stores while meeting the application requirements. The key ingredient of this solution is the use of manifests enabling the description of user requirements and data stores capabilities. In the remainder of this section, we define our discovery principles (see Section 5.3.1). Then we introduce the abstract application manifest and the offer manifest structure (see Section 5.3.2 and Section 5.3.3). Finally, we present our matching algorithm (see Section 5.3.4).

5.3.1 Automatic Discovery Principles

Once the developer finishes his/her application coding using ODBAPI, he/she should deploy it in a Cloud environment in order to run it. However, prior to that he/she must discover Clouds' resources. To do so, we propose in this work a manifest based discovery process. In Figure 5.1, we showcase an overview of our approach. First, the developer describes his/her application's requirements in the abstract application manifest in terms of storage and computing. Second, he/she gives it as an input to the *matching algorithm*. This algorithm interacts with the *data stores directory* in order to obtain the data stores capabilities of each cloud provider stored in the offer manifest. This manifest represents the second input of the *matching algorithm* which allows for obtaining the deployment manifest. This latter is used to deploy the application in the elected Cloud provider. It is worth noting that the *data stores directory* is automatically updated by interacting with the Cloud providers (e.g. Cloud Foundry, OpenShift, etc.) using their proprietary APIs.

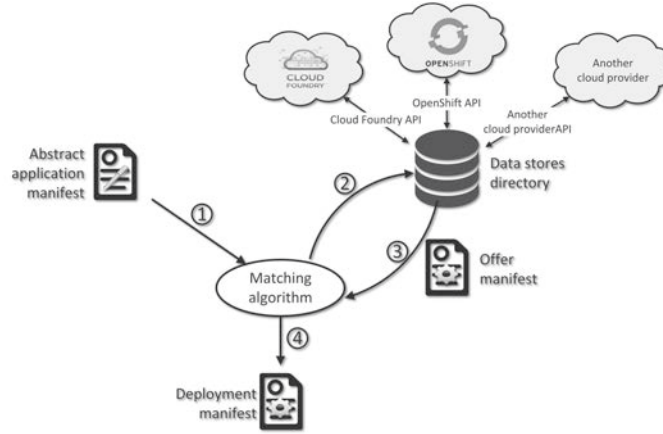


Figure 5.1: Zoom-in on the discovery component

5.3.2 Abstract Application Manifest

Using a dedicated manifest, developers express what kind of data stores and execution environment their application requires. Indeed, this manifest contains two categories of requirements. First, we have requirements in terms of data stores. The developer provides four information about the required data stores: its type, its name, its version, and its size. In addition, it is worth noting that when the developer fills this manifest, he/she has the freedom to specify one or multiple information. For each information, he/she gives a constraint expressed by a constant value, a joker (denoting any values) or some conditions (expressed as inequalities). Hence, we will ensure more flexibility in our model and in the discovery step. For instance, one developer precises that he/she wants a data store having the name

MongoDB and the type document and another developer precises that he/she wants a data store of type document without specifying its name (in this case any data store of type document fulfill the specification). Thus, the discovery step for the second developer will be easier and more flexible. Whereas the second category of requirements is dedicated to the application deployment. Indeed, the developer precises the number of the virtual machines needed to run his/her application. In addition, he/she describes his/her application executable by giving its name, its type and its location.

Figure 5.2 depicts the structure of the abstract application manifest. The root element of our model is the *abstract application manifest* element and it is identified by the attribute *name*. Basically, it contains three categories of information: the first category is expressed under the *application* element, the second one is defined under the *environment* element and the third one is defined under the *user information* element. In the following, we introduce these three elements:

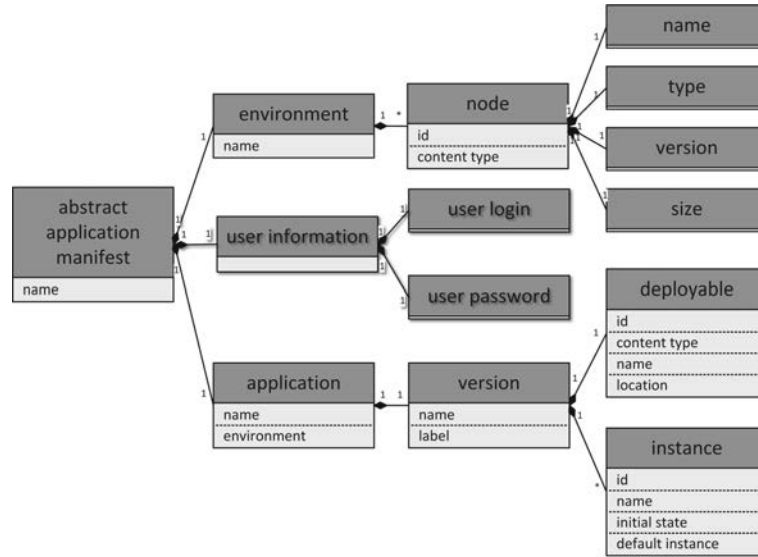


Figure 5.2: Abstract application manifest model

- **The user information element:** This element represents the required authentication information to access the discovery component and to interact with the data stores repository. It contains the *user login* and the *user password* elements representing the developer identifiers.
- **The environment element:** The *environment* element in the abstract application manifest specifies the requirements on the used data stores as well as on the application execution environment. These requirements are expressed over a set of *node* elements where each *node* refers to a data store or an execution environment component. A *node* element is identified by an *id* attribute and a *content_type* attribute. This latter can be a container or a database to respectively denote engine resources to host and run services

and storage resources. For each *node* and when a *content_type* is equal to a database, developers may specify the required *type* (e.g. a document or a key-value data store), the *name* of a specific component (e.g. *MongoDB* or *CouchDB*), any specific *version*, and the required *size*. Otherwise, if a *content_type* is equal to a container, users specify only the *name* and the *version* elements.

- **The application element:** This element contains information on the to-be-deployed application itself. It is characterized by a unique *name* attribute and the *environment* attribute where the application will be deployed. Developers may specify several versions of the same application. And for each version, they need to precise on the one hand information related to the deployable artefacts and on the other hand information related to the to-be-run instances. A *version* element is identified by a *name* and a *label* and contains a set of *deployable* and *instance* elements. The *deployable* element represents the application executable file. It is identified by a unique *id* attribute, a *content_type* attribute defining the executable file type, a *name* attribute denoting its name, a *location* attribute containing the URL where such element can be found and retrieved, and a *multitenancy_level* attribute indicating the application tenancy degree. Whereas the *instance* element represents the running application instances required by the user. This element is identified by a unique *id*, a *name*, *initial_state* defining the state of the application (e.g. running, stopped, etc.) and *default_instance* representing the running instances by default.

5.3.3 Offer Manifest

The offer manifest contains information about the capabilities of data stores of each discovered Cloud provider considering the abstract application manifest. Indeed, each Cloud provider may return one or multiple offers in terms of their data stores capabilities. These offers represent all possible combinations that an environment can propose to the application. In Figure 5.3, we present a modeling of the offer manifest based on a class diagram. The root element is *offer manifest* and it is identified by the *name* attribute. It contains one or multiple *cloud provider* element. This element represents a discovered cloud provider and it is identified by a unique *id*. It contains (1) the *name* element defining the Cloud provider name and (2) the *offers* element representing the capabilities that a Cloud provider exposes with respect to the application requirements described in the abstract application manifest. The *offers* element is composed by one or multiple *offer* elements that contains one or multiple *node* element. This latter is similar to the *node* element in the abstract application manifest.

5.3.4 Matching Algorithm

In this section, we introduce the matching algorithm enabling the election of the Cloud provider that best supports the application's requirements in terms of data

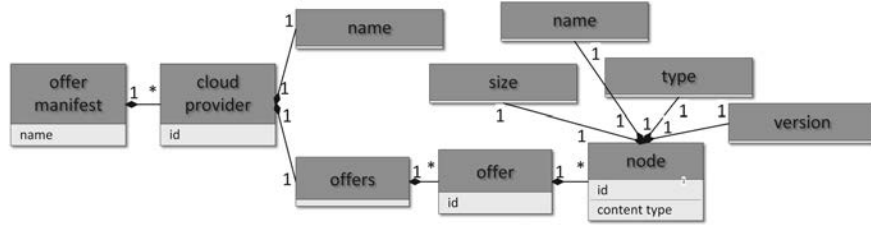


Figure 5.3: Offer manifest model

stores (see Algorithm 2). We propose a flexible matching allowing to elect a Cloud provider which does not always fulfill all the application's requirements but it is closed to them (which is the role of the computed distance in the algorithm). We use also a threshold fixed by a developer or an administrator in order to compute a bounded distance between zero and its value. Furthermore, we impose that the result is a single Cloud provider (we do not consider deployment in multiple Cloud providers). The algorithm takes as input the abstract application manifest and the threshold and returns as output the deployment manifest that we introduce in Section 5.4.2. In the following, we describe the different steps of the matching algorithm:

Algorithm 2 Matching algorithm

```

1: input AAM: the abstract application manifest
2: input threshold: the threshold to limit the number of differences
3: output DM: the deployment manifest
4: OM ← queryDataStoresRepository(AAM) # see Algorithm 3#
5: i ← 0
6: while (exist(Cloud Provider CP in OM)) do
7:   while (exist(Offer O in OM)) do
8:     distance[i] ← 0
9:     for each node N in AAM do
10:      for each property prop in N do
11:        if (!valid(prop, OM.CP.O.node.prop)) then
12:          distance[i] ← distance[i] + updateDistance(prop,
13:            OM.CP.O.node.prop)
14:        end if
15:      end for
16:    end for
17:    i ← i + 1
18:  end while
19: electedCP ← electCP(distance, threshold)
20: return createDM(AAM, OM, electedCP)

```

1. **Offer manifest construction (line 4)**: The algorithm constructs the offer

manifest using the operation *queryDataStoresRepository* by interacting with the data stores repository. We give more details about this operation in Algorithm 3.

2. **Distances' calculation (lines 6-18):** Algorithm 2 calculates for each Cloud provider the number of differences between its capabilities and the user requirements described in the abstract application manifest. We estimate the distance between the abstract application manifest and the offer manifest by the number of differences. These numbers are stored in the data structure *distance*. These values are calculated as follows: For each property in the two manifests, if they are not corresponding then we update the distance by adding the appropriate penalty to the property. The two properties correspond if the actual value of the offer manifest property fulfill the requirement expressed by the abstract application manifest property (which is either a constant, a joker or a condition). These penalties are customized according to the property importance. By default, all penalties are fixed at 1; however the user can configure these penalties according to the importance that he/she gives to the properties.
3. **Cloud provider election (lines 19-20):** Finally, we elect a Cloud provider and construct the deployment manifest. This is done through the operation *electCP* that takes as inputs the data structure *distance* and the *threshold* and returns the identifier of the elected Cloud provider if any. This identifier is the smallest value bounded between 0 and the *threshold*.

In algorithm 3, we present in more details the operation *queryDataStoresRepository* that returns a super-set of the result from the data stores repository. In fact, for each Cloud provider, we extract all data stores corresponding to the data types of the abstract application manifest. If there are no corresponding data stores, the Cloud provider is rejected (lines 10-18). For ease of presentation of this algorithm, we build ourselves on the Figure 5.4. Indeed, in the left side of this figure, we construct from the abstract application manifest a simple graph in which nodes represent the *type* elements in the *node* elements. This graph is a kind of a sample that will be used to construct the offer manifest. In the right side of Figure 5.4, we illustrate a data stores repository of a Cloud provider having four types of data stores. Based on this repository and the graph based sample, we extract the list of the Cloud provider's offers that we represent in the form of a graph. Once we check all Cloud providers, we group all the offers in the offer manifest (line 19).

In the case of the selection of a new data store by the discovery process, our approach does not support neither the creation of the target schema on the selected data stores, nor the migration of existing instances. This has to be done outside our process. For instance, we can use ExSchema which is a tool enabling the automatic discovery of external data schemes from the source code of multiple data stores based applications [64].

Algorithm 3 The *queryDataStoresRepository* algorithm

```

1: input AAM: the abstract application manifest
2: output OM: the offer manifest
3: length  $\leftarrow$  0
4: for each node N in AAM do
5:   if (content-type == "database") then
6:     T[length]  $\leftarrow$  getType(N)
7:     length  $\leftarrow$  length + 1
8:   end if
9: end for
10: for each Cloud Provider CP in the data stores directory do
11:   for i=0 to length do
12:     if (CP contains T[i]) then
13:       Add all names of this type to the tree of the current CP
14:     else
15:       Reject the current CP
16:     end if
17:   end for
18: end for
19: return the resulted trees as the OM

```

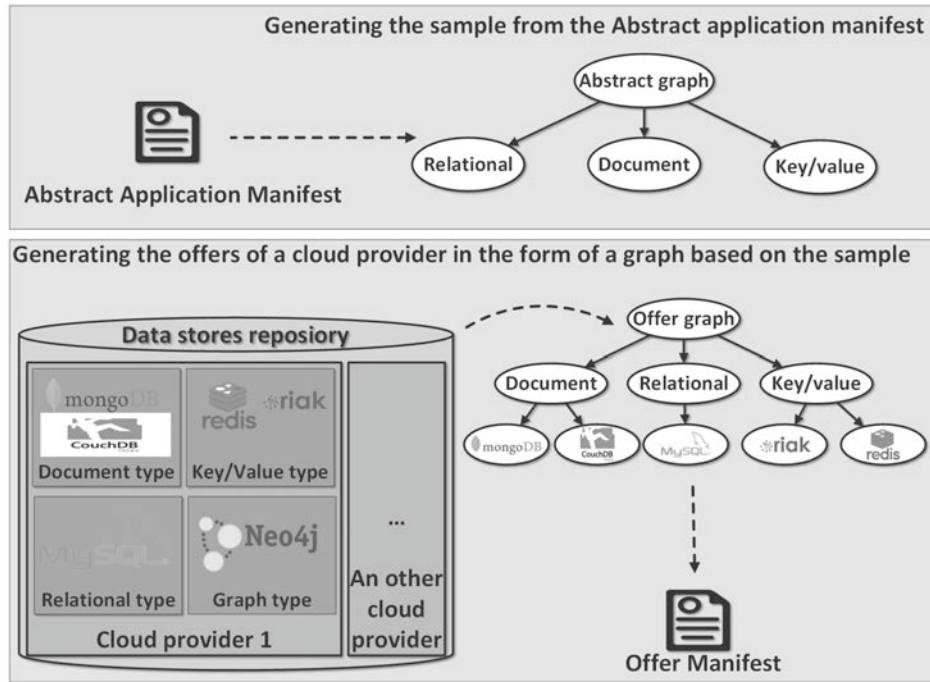


Figure 5.4: Generating the offer manifest

5.4 Automatic Deployment Of Multiple Data Stores Based Applications

In this section, we introduce our solution to deploy multiple data stores based applications in any Cloud provider. To do so, our starting point is COAPS API [65]

that is a manifest-based deployment API proposed in our team. Nevertheless this API supports only single data store based application. Hence, we propose to extend COAPS API in order to enable multiple data stores based application deployment in Cloud environments. Against this background, we define in Section 5.4.1 the COAPS API. Then, we present in Section 5.4.2 the structure of the deployment manifest.

5.4.1 An Extension Of The Compatible One Application and Platform Service API

The Compatible One Application and Platform Service API (COAPS API) [65] allows human and/or software agents to provision and manage PaaS applications. This API provides a unique layer to interact with any Cloud provider based on manifests. This REST API manages two kinds of resources. The first kind of resources represents the environment which is a set of settings required by developers to host and run their applications in Cloud environments. Indeed, these settings may be a runtime (java 7, java 6, ruby, etc.), frameworks/containers (spring, tomcat, ruby, etc.) and eventually services (databases, messaging, etc.). Whereas the second kind of resources is applications which are computer software or program deployable in a Cloud environment. Application source archives should be provided by the developer in a bundled format (i.e. war, ear, zip, etc.) or extracted format (i.e. a local folder with the different files and dependencies, distant URL, etc.).

In Figure 5.5, we illustrate an overview of the COAPS API that we introduce from the right side to the left side.

- **Integrated Cloud providers (Integrated PaaS):** We have first of all Cloud providers where a developer may deploy his/her application. In the figure, we showcase that the developer may deploy his/her application in two PaaS that are CloudFoundry and OpenShift.
- **COAPS API interface:** The second part of Figure 5.5 represents the COAPS API interface and different implementations of each PaaS. In fact, it represents the shared part between all the integrated PaaS and it provides a unique view to the application side. It contains specific implementations of each PaaS. The current version of the COAPS API implementation includes two Cloud providers: (1) CloudFoundry and (2) OpenShift.
- **COAPS API operations:** Finally, we show the different operations that COAPS API offers to the user. These operations are well introduced in the rest of this section.

For ease of presentation of the COAPS API operations, we propose to describe it through a scenario of an application deployment in a Cloud provider (see Figure 5.6). Indeed, the developer describes first his/her requirements in terms of application and environment in the deployment manifest. Then, he/she executes the

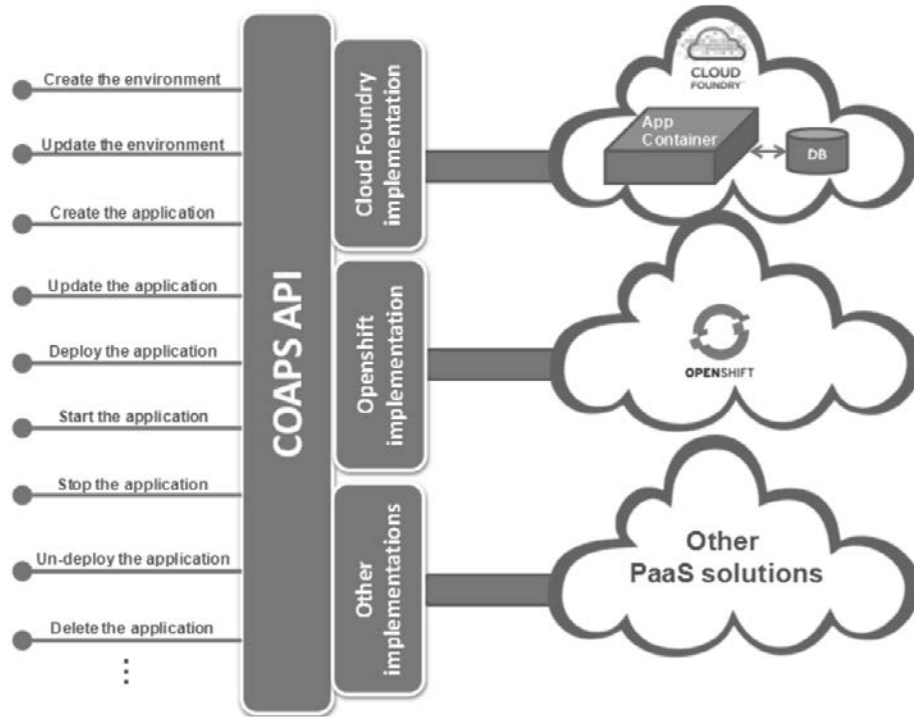


Figure 5.5: COAPS API overview [3]

operation *createEnvironment* to create an environment with respect to the manifest. Afterward, he/she should execute the operation *createApplication* to create a resource of type application and the operation *deployApplication* to deploy his/her application. Finally, he/she runs his/her application by executing the operation *startApplication*. Added to these operations, developers may update an environment or an application, list the deployed applications in an environment, stop and un-deploy an application.

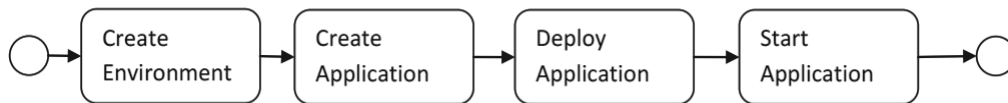


Figure 5.6: An application deployment scenario [3]

Although COAPS API is efficient and allows application deployment in a Cloud environment in a transparent way, it does not support multiple data stores applications deployment. For this sake, we propose to extend it in order to support this kind of applications. Indeed, we propose to enable the creation of environment resources containing multiple data stores. Then, we provide the possibility to deploy over it polyglot persistence based applications. In Section 5.4.2, we present how we extend COAPS API deployment manifest.

5.4.2 Deployment Manifest

The deployment manifest's structure is closest to the abstract application manifest (see Section 5.3.2) and it is defined based on the COAPS API's manifest [65] (see Figure 5.7). Hence, in order to avoid the repetition, we do not describe this manifest structure in details. However, we present how we extended the manifest compared to that of COAPS API. This extension is two-fold. In the one hand, we increase the number of *node* elements having a *content_type* equal to database from one to multiple. Thus, we can now support the multiple data stores based application deployment. In the other hand, we propose to enrich each *node* element having a *content_type* equal to database with additional information. In fact, we add new attributes about data stores services in the *node* element. These attributes are the *size* attribute, the *type* attribute and the *version* attribute to respectively denote the size of a data store, its type and its version.

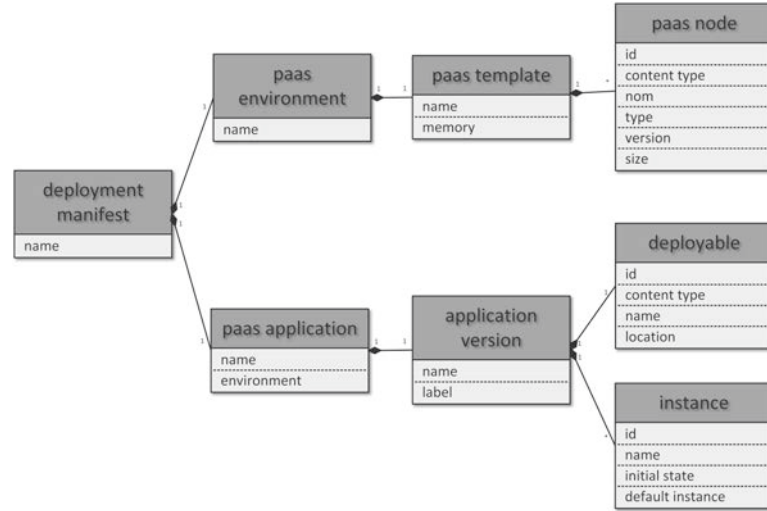


Figure 5.7: Deployment manifest model

5.5 XML-Based Manifest Examples

In this section, we present examples of manifests implemented in XML format. We give below an example of an abstract application manifest (see Listing 5.1). Indeed, the developer provides *user1* as a login and *pswd* as a password. Then, he/she describes the environment where he/she wants to deploy his/her ODBAPI based application. Indeed, he/she chooses as name for the environment *ODBAPI-Env* and for the environment template *ODBAPIEnvTemp*. In this template, the user requires *tomcat* as an application container, a document data store without precising any name by filling the element *name* with the character *** and MySQL as a relational data store. Regarding the application configuration, the user names his/her application *ODBAPIApplication*. He/She precises that the application is a

runnable file and he/she requires to have two application instances: one is running by default and the other is stopped.

```

1<abstract_application_manifest name="AAM">
2  <user_information>
3    <user_login>user1</user_login>
4    <user_password>pswd</user_password>
5  </user_information>
6  <environment name="ODBAPIEnv">
7    <template name="ODBAPIEnvTemp" memory="128">
8      <node id="1" content_type="container">
9        <name>tomcat</name>
10       <version>1</version>
11     </node>
12     <node id="2" content_type="database">
13       <name>*</name>
14       <version>1.0</version>
15       <type>document</type>
16       <size>large</size>
17     </node>
18     <node id="3" content_type="database">
19       <name>mysql</name>
20       <version>*</version>
21       <type>relational</type>
22       <size>small</size>
23     </node>
24   </template>
25 </environment>
26 <application name="ODBAPIApplication" environment=
27   "ODBAPIEnv">
28   <version name="version1.0" label="1.0">
29     <deployable id="1" content_type="artifact"
30       name="ODBAPIApplication.war" location="1444d7"
31       multitенancy_level="SharedInstance"/>
32     <instance id="1" name="Instance1"
33       initial_state="1" default_instance="true"/>
34     <instance id="2" name="Instance2"
35       initial_state="1" default_instance="false"/>
36   </version>
37 </application>
38</abstract_application_manifest>

```

Listing 5.1: XML based representation of the abstract application manifest

Based on this example of abstract application manifest, we give below an example of the offer manifest got during the matching algorithm execution. We illustrate this manifest by a XML-based presentation (see Listing 5.2). In fact, in this example we take into account two cloud providers named *Cloud provider 1* and *Cloud provider 2*. The first one proposes two offers with respect to abstract application manifest. Whereas the second one proposes just one offer.

```

1<offer_manifest name="OffersOfCloudProviders">
2  <cloud_provider id="1">
3    <name>Cloud provider 1</name>
4    <offers>
5      <offer id="1">
6        <node id="1" content_type="container">
7          <name>tomcat</name>
8          <version>1</version>
9        </node>
10       <node id="2" content_type="database">
11         <type>document</type>
12         <name>couchDB</name>
13         <version>1.0</version>
14         <size>small</size>
15       </node>
16       <node id="3" content_type="database">
17         <type>relational</type>
18         <name>mysql</name>
19         <version>2.0</version>
20         <size>small</size>
21       </node>
22     </offer>
23     <offer id="2">
24       <node id="1" content_type="container">
25         <name>tomcat</name>
26         <version>1</version>
27       </node>
28       <node id="2" content_type="database">

```

```

29         <type> document </type>
30         <name> mongoDB </name>
31         <version> 1.0 </version>
32         <size> large </size>
33     </node>
34     <node id="3" content_type = "database">
35         <type> relational </type>
36         <name> mysql </name>
37         <version> 2.0 </version>
38         <size> small </size>
39     </node>
40 </offer>
41 </offers>
42 </cloud_provider>
43 <cloud_provider id ="2">
44     <name> Cloud provider 2 </name>
45     <offers>
46         <offer id ="1">
47             <node id="1" content_type = "container">
48                 <name> tomcat </name>
49                 <version> 1 </version>
50             </node>
51             <node id="2" content_type = "database">
52                 <type> document </type>
53                 <name> couchDB </name>
54                 <version> 1.0 </version>
55                 <size> small </size>
56             </node>
57             <node id="3" content_type = "database">
58                 <type> relational </type>
59                 <name> oracle </name>
60                 <version> 2.0 </version>
61                 <size> small </size>
62             </node>
63         </offer>
64     </offers>
65 </cloud_provider>
66 </offer_manifest>

```

Listing 5.2: XML based representation of the offer manifest

Based on these two manifests, we present an execution scenario example of the matching algorithm. We suppose that the user's threshold is equal to 2 and the user's preferences regarding the penalties are: (1) $Penalty_{Type} = 4$, (2) $Penalty_{Name} = 2$, (3) $Penalty_{Size} = 1$, and (4) $Penalty_{Version} = 1$. We start by comparing the type of data stores. Indeed, the user requires a relational and document data store. In Figure 5.8, we illustrate the abstract graph according to the abstract application manifest showcased in Listing 5.1. Then, we compare each offer to the application's requirements and we gradually compute the distance between the offer and the abstract application manifest. By applying our matching algorithm, we obtain the distances illustrated in Table 5.1. In fact, we show that the *offer 2* of the *Cloud provide 1* has a null distance. Hence, we elect this offer since it meets exactly the application's requirements. We point out that the *Cloud provider 2* is discarded since it has a distance equals to 3 (it is computed as follows: $Penalty_{Size} + Penalty_{Name}$) which is greater than the threshold. This is due to the absence of a relational data store having the name *MySQL* and the size of the proposed document data store is not *large*. Against this, we construct the deployment manifest presented in Listing 5.3 using the *offer 2* of the *Cloud provider 1*.

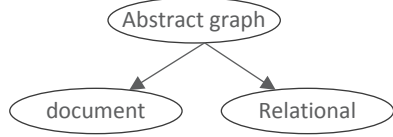


Figure 5.8: The abstract graph

Cloud provider id	Offer id	distance
1	1	1
1	2	0
2	1	3

Table 5.1: Summary of the computed distances between the offer manifest and the abstract application manifest

```

1<deployment_manifest name="DM">
2  <paas_environment name="ODBAPIEnv">
3    <paas_template name="ODBAPIEnvTemp" memory="128">
4      <paas_node id="1" content_type="container"
5        name="tomcat" version="1.7">
6        <paas_node id="2" content_type="database"
7          name="MongoDB" version="1.0" type="document"
8          size="large" />
9        <paas_node id="3" content_type="database"
10         name="mysql" version="2.0" type="relational"
11         size="small" />
12      </paas_template>
13    </paas_environment>
14    <paas_application name="ODBAPIApplication" environnement=
15      "ODBAPIEnv">
16      <application_version name="version1.0" label="1.0">
17        <deployable id="1" content_type="artifact"
18          name="ODBAPIApplication.war" location="1444d7"
19          multitennancy_level="SharedInstance"/>
20        <instance id="1" name="Instance1"
21          initial_state="1" default_instance="true"/>
22        <instance id="2" name="Instance2"
23          initial_state="1" default_instance="false"/>
24      </application_version>
25    </paas_application>
26</abstract_application_manifest>
  
```

Listing 5.3: XML based representation of the deployment manifest

Now, we propose to give an example of an abstract application manifest where the developer knows what he/she desires in advance and fills the abstract application manifest without using wildcards(see Listing 5.4). Indeed, he/she specifies that he/she wants a *MySQL* relational data store and a *MongoDB* document one. In this case, our discovery component will only focus on Cloud providers having the same data stores services with the required size and version. One possible deployment manifest is the manifest represented above in Listing 5.3.

```

1<abstract_application_manifest name="AAM">
2  <user_information>
3    <user_login>user1</user_login>
4    <user_password>pswd</user_password>
5  </user_information>
6  <environment name="ODBAPIEnv">
7    <template name="ODBAPIEnvTemp" memory="128">
8      <node id="1" content_type="container">
9        <name>tomcat</name>
10       <version>1</version>
11     </node>
12     <node id="2" content_type="database">
13       <name>MongoDB</name>
14       <version>1.0</version>
15       <type>document</type>
16       <size>large</size>
17     </node>
18     <node id="3" content_type="database">
19       <name>MySQL</name>
20       <version>2.0</version>
21       <type>relational</type>
22       <size>small</size>
23     </node>
24   </template>
25 </environment>
  
```

```

26 <application name="ODBAPIApplication" environnement=
27 "ODBAPIEnv">
28   <version name="version1.0" label="1.0">
29     <deployable id="1" content_type="artifact"
30       name="ODBAPIApplication.war" location="1444d7"
31       multitenancy_level="SharedInstance"/>
32     <instance id="1" name="Instance1"
33       initial_state="1" default_instance="true"/>
34     <instance id="2" name="Instance2"
35       initial_state="1" default_instance="false"/>
36   </version>
37 </application>
38</abstract_application_manifest>

```

Listing 5.4: XML based representation of an abstract application manifest containing only the type of data stores

5.6 Conclusion

In this chapter, we introduced our approach to discover data stores and to deploy multiple data stores based application in Cloud environments. Indeed, we presented first our matching algorithm that enable the discovery of the most suitable Cloud provider to an ODBAPI-based application requirements. It is based on the abstract application manifest and the offer manifest and it outputs a deployment manifest. This latter is used by COAPS API to deploy the application in the elected Cloud provider. We extended COAPS API in order to support multiple data stores based application deployment.

In the next chapter, we will validate through different use cases for our four contributions. We also performed different experiments to test the efficiency and accuracy our proposals.

Evaluation & Validation

Contents

6.1	Introduction	103
6.2	Proofs Of Concept	103
6.2.1	The OpenPaaS Project	104
6.2.2	Current State Of ODBAPI	104
6.2.3	Selecting Multiple Data Stores And Deploying ODBAPI Clients	108
6.3	ODBAPI Overhead Evaluation	109
6.4	Ease Of ODBAPI Use	110
6.5	Complex Queries Optimization	113
6.6	Conclusion	119

6.1 Introduction

All along this manuscript, we proposed an end-to-end approach enabling the support of developers during the lifecycle of multiple data stores based applications. This approach involves four main components that we try to validate and evaluate in this chapter. For this purpose, we present (i) the implementations we have done as a proof of concepts to realize our four components (see Section 6.2), and (ii) the experiments we have conducted to evaluate the efficiency of our solutions. Indeed, we evaluate the overhead of ODBAPI compared to proprietary API (see Section 6.3). Then, we assess the gain in terms of implementation time and the number of lines in a source code of an application implemented using ODBAPI and an other one implemented using proprietary APIs (see Section 6.4). Finally, we evaluate our cost model and the algorithm for generating optimal execution plan (see Section 6.5). Our goal is (i) to prove that our approach is feasible and accurate in real use-cases and (ii) to analyze the parameters that impact our results' quality.

6.2 Proofs Of Concept

In this section, we present the proofs of concept that we have developed to realize our approach. We firstly present the OpenPaaS project in which we have integrated the proofs of concept (see Section 6.2.1). Second, we present a state of progress about the implementation of ODBAPI and the data stores that we take

into account (see Section 6.2.2). Added to that, we present some ODBAPI-based applications that illustrate the utility of our API. Third, we present a tool allowing the discovery of multiple data stores based on the *abstract application manifest* (see Section 6.2.3).

6.2.1 The OpenPaaS Project

The OpenPaaS project ⁴ aims at developing a PaaS technology dedicated to enterprise collaborative applications deployed on hybrid Clouds (private / public). OpenPaaS is a platform that allows to design and deploy applications based on proven technologies provided by partners such as collaborative messaging system, integration and workflow technologies that will be extended in order to address Cloud Computing requirements (see Figure 6.1).

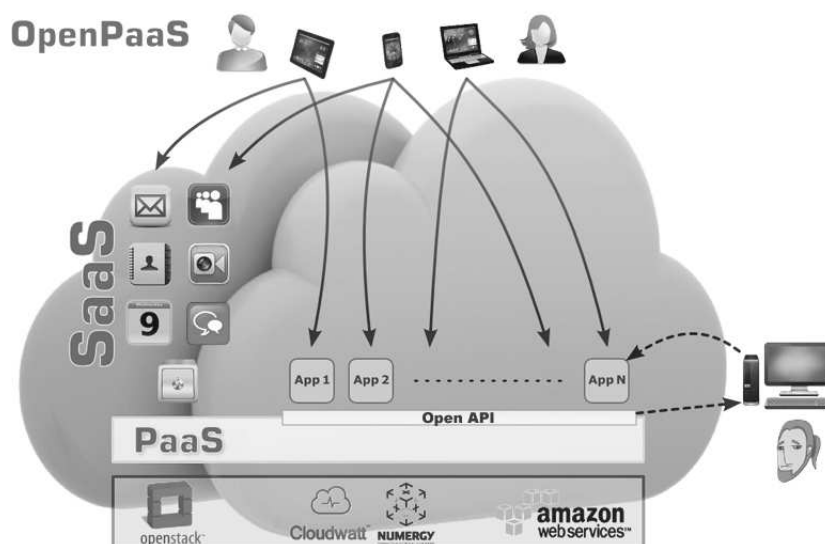


Figure 6.1: OpenPaaS overview

One of the main objective of this projet is to specify a unified REST-based API that allows an application deployed in the OpenPaaS platform to interact (i.e. express and execute simple and complex queries) with relational and NoSQL data stores. The OpenPaaS project focuses in particular on relational, key/value and document data stores. In addition, OpenPaaS aims at automatically deploying a multiple data stores based application in an appropriate Cloud provider.

6.2.2 Current State Of ODBAPI

According to the requirements of the OpenPaaS project, we provide today a version of ODBAPI including four data stores: MySQL, Riak, MongoDB and CouchDB.

⁴<https://research.linagora.com/display/openpaas/Open+PAAS+Overview>

The current version is developed within JAVA and is provided as a runnable RESTful web application. We have worked diligently on testing ODBAPI using various use cases in the OpenPaaS project so that we identified possible discrepancies and made this version more stable for use. A description of the realized work is available at http://www-inf.int-evry.fr/~sellam_r/Tools/ODBAPI/index.html. In this page, reader will find three links: (1) the first one allows accessing the full ODBAPI specification, (2) the second one allows downloading a jar file of ODBAPI, (3) and the third one provides a detailed user guide. We point out that we used the Restlet framework⁵ in order to implement ODBAPI. As it is a REST-based API, ODBAPI can be implemented by any programming language supporting the REST architecture (e.g. PHP, AngularJS, JAVA, etc.).

In order to present the different component constituting the ODBAPI, we propose to illustrate the implementation through a package diagram. This kind of diagram allows to gather classes that are logically linked into one module. Hence, we provide a global view on ODBAPI. In Figure 6.2, we showcase this diagram. In the left side of this diagram, we illustrate the main package. This package is called *server* and contains the main class of the project that receives a REST query and routes it to the target resources. To do so, this class interacts with classes from the packages in the right side of the diagram. These packages are described as follows:

- **The *all* package:** This package provides the required operations to list either all entities in an entity set or all entity sets in a data store.
- **The *entity* package:** This package supplies the list of operations to query resources of type entity. Indeed, we can create, retrieve, update and delete an entity.
- **The *entitySet* package:** This package provides the operations enabling to manage resources of type entity sets. In fact, we can create, retrieve and delete an entity set.
- **The *query* package:** This package is intended to handle the execution of complex queries. Currently, we support the execution of filtering (i.e. selection and projection operations) and join operations.

The addition of a new data store implies the implementation of a new ODBAPI driver. It is important to evaluate the cost of such development. For this purpose, we rely on the package diagram of ODBAPI (see Figure 6.2). In each package, we illustrate a set of JAVA interfaces that a developer should implement to add a new driver to ODBAPI. In all, we present ten interfaces. Hence, we conclude that a developer should implement ten JAVA classes for each new driver. In Table 6.1, we showcase the lines number coded in each driver. We have an average lines number equals to 633,5. In addition, we present the implementation duration of each driver in terms of days done by one developer. We observe that this duration

⁵<http://restlet.org/>

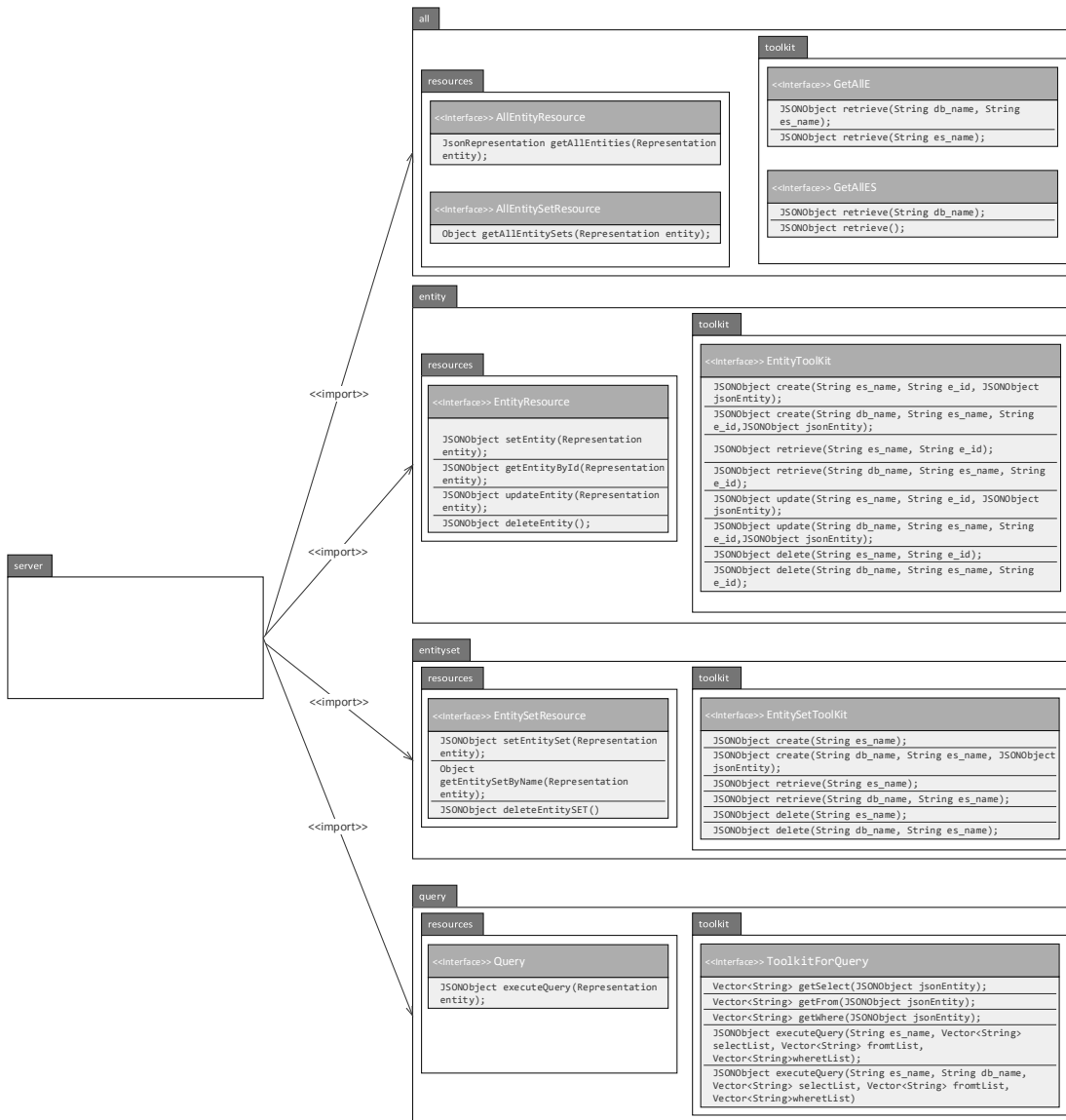


Figure 6.2: Package diagram of ODBAPI

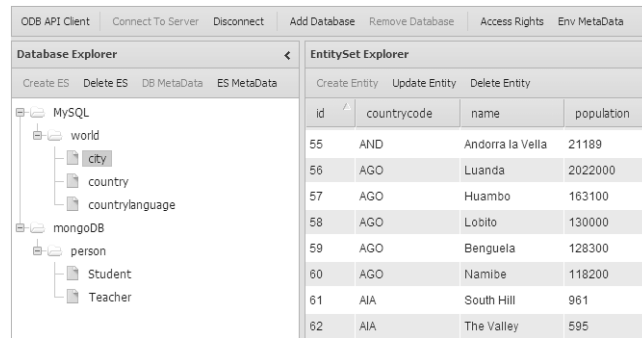
does not exceed two days. Based on the average lines number, the classes number as well as on the implementation duration, we conclude that adding a new driver is not costly in terms of time and manpower.

Data store name	MySQL	MongoDB	CouchDB	Riak
Lines number	739	637	608	550
Implementation time (day)	1	2	1,5	1,5

Table 6.1: Lines number in each driver in ODBAPI

In order to show the feasibility, the portability, and the utility of our API, we provide three ODBAPI-based applications:

- **Management of to-do tasks in a programming project:** We implemented in the Open PaaS project an ODBAPI-based module enabling the management of to-do tasks in a programming project. In this module, we interact with a document data store containing information about tasks and a relational data store containing information about developers involved in a project and their duties in a given task. Users of this module interact with the data stores by executing either CRUD operations or multi-sources queries (i.e. filtering and join). This module has been developed using AngularJS and it has been integrated in the final demonstration of the Open PaaS project.
- **Management of relational and NoSQL data stores:** We also developed an ODBAPI-based application intended to handle the administration of relational and NoSQL data stores in a Cloud provider. This application is a PHPMyadmin-like and it is developed using PHP, HTML and Javascript. In Fig. 6.3, we show a screenshot of the user interface of this client. In fact, it gives an overview of two heterogeneous data stores. There is a MySQL database called *world* and it contains three entity sets: *city*, *country*, and *countrylanguage*. Added to that, we have a MongoDB database that is named *person* and it is composed by two entity sets: *Student* and *Teacher*. We also show an overview of the entities of the *city* entity set.



id	countrycode	name	population
55	AND	Andorra la Vella	21189
56	AGO	Luanda	2022000
57	AGO	Huambo	163100
58	AGO	Lobito	130000
59	AGO	Benguela	128300
60	AGO	Namibe	118200
61	AJA	South Hill	961
62	AJA	The Valley	595

Figure 6.3: Screenshot of all databases overview

- **ODBAPI client for applications programming:** We propose a client that we called *ODBAPI client*. This latter allows a developer to use ODBAPI operations through JAVA methods as if he/she uses a proprietary API. This eases the implementation of ODBAPI-based applications. We provide a jar file that a developer should add to his/her local libraries or maven dependencies. This client is used to compute the experiments related to ODBAPI in Section 6.3 and Section 6.4.

Thanks to these three applications, we show the utility of ODBAPI. In addition,

we prove that it is portable and can be implemented by any REST-based programming language. In addition, we proved that adding a new driver to ODBAPI is not a tedious task.

6.2.3 Selecting Multiple Data Stores And Deploying ODBAPI Clients

In this section, we present a tool ensuring the discovery of Cloud providers and the automatic deployment of an ODBAPI-based application. This tool implements our matching algorithm to create the deployment manifest and the extension of the COAPS API to deploy a multiple data stores based application. In order to show the feasibility of this component in our work, we propose to discover data stores of Cloud Foundry and OpenShift as Cloud providers to deploy an ODBAPI-based application. This application is intended to interact with a relational data store of type MySQL and a document data store in a Cloud environment. Indeed, the application developer describes his/her requirements in the *abstract application manifest* and he/she uploads it through the interface that we illustrate in Fig. 6.4. Then, this tool executes the matching algorithm to elect the Cloud provider that supports the ODBAPI client requirements and returns the *deployment manifest*. Based on this latter, we deploy the ODBAPI client by the mean of COAPS API. We wish to emphasize that we provide this tools and some demonstration videos at http://www-inf.int-evry.fr/~sellam_r/Tools/ODBAPI/index.html for more details.



Figure 6.4: Screenshot of the interface allowing to import the abstract application manifest and generate the deployment manifest

6.3 ODBAPI Overhead Evaluation

Using ODBAPI facilitates the developer's task greatly; however, it comes with the cost of an overhead. In fact, ODBAPI is based on the proprietary APIs of relational and NoSQL data stores. In this section, we propose to evaluate the overhead related to ODBAPI. Doing so, we propose to answer the following question:

- **Question₁**: Does this overhead decrease ODBAPI performance?

For this purpose, we implemented two applications doing the same CRUD operations: one is using ODBAPI and the other is using JDBC. These two applications are deployed and run in the same environment. The data store, the ODBC driver and the application run on the same server. We are aware that we are doing extra works compared to these proprietary APIs. Indeed, for each query, our API rewrites it into the proprietary query language of the integrated data store. Then, it converts the result to JSON format before answering the application. In addition, since ODBAPI is a REST-based architecture API, this also may generate an overhead due to the REST protocol and the data shipping.

The overhead Δ is obtained by calculating the ratio between the difference between the response time of ODBAPI and the response time of the proprietary API, and the response time of the proprietary API. We use the following formula :

$$\Delta = \left(\frac{t_{ODBAPI} - t_{proprietaryAPI}}{t_{proprietaryAPI}} \right) * 100.$$

In the rest of this section, we limit ourselves to present only the overhead of ODBAPI when application interacts with relational data store using JDBC. We have started also evaluating the overhead of our API compared with MongoDB API and the first result obtained with this API are in line with a light overhead as well.

We start by calculating the evolution of the response time according to the number of the created entities using ODBAPI and JDBC. In TABLE 6.2, we showcase the response time of this operations and the overhead. The average of this overhead is about 6.71 %.

Entities number	10	50	100	500
ODBAPI (ms)	715	2399	4219	19115
JDBC (ms)	700	2250	3883	17463
Δ (%)	2.14	6.62	8.65	9.46

Table 6.2: Response time of the operation create entities with ODBAPI and JDBC

We use the same principle to evaluate the overhead of deletion of a relational entities. In TABLE 6.3, we present the obtained results and the overhead. The average of this overhead is about 4.35 %.

We calculate also the overhead of retrieving all entities in one query with ODBAPI and JDBC. For this, we illustrate in TABLE 6.4 the evolution of the

Entities number	10	50	100	500
ODBAPI (ms)	677	2309	4164	18238
JDBC (ms)	662	2205	3878	17696
Δ (%)	2.26	4.71	7.37	3.06

Table 6.3: Response time of the operation delete entities with ODBAPI and JDBC

response time according to the number of retrieved entities using ODBAPI and JDBC. The average of this overhead is about 8.06 %. The performance of ODBAPI degrades for 4000 entities which is probably due to a problem of memory management.

Entities number	100	500	1000	2000	3000	4000
ODBAPI (ms)	215	269	305	349	416	543
JDBC (ms)	213	263	288	347	408	422
Δ (%)	0.93	2.28	5.90	0.57	1.96	28.67

Table 6.4: Response time of the operation retrieve all entities with ODBAPI and JDBC

To sum up, the overhead that we obtained is quite acceptable for all type of operations (Answering therefore Question₁). However, we can enhance the response time of our API by decreasing the conversion time that is big especially when it comes to convert big volume of data.

6.4 Ease Of ODBAPI Use

ODBAPI is intended to ease the developers life and alleviate the burden on them while implementing multiple data stores based applications. However, it would be certainly worthwhile to concretely evaluate this and try to answer this question:

- **Question₂**: Does ODBAPI really alleviates the burden on developers?

To tackle Question₂, we evaluate the gain for developers of using ODBAPI compared to proprietary APIs. The gain is measured in terms of implementation time and of source code lines number. Doing so, we want to show how ODBAPI alleviates the burden on developers of multiple data stores based application.

For this purpose, we asked four developers (two trainees, a researcher, and an engineer) to implement two versions of an application interacting with MySQL as a relational data store and MongoDB as a document data store. In this application, developers should create an entity set, and create, retrieve and delete an entity in both data stores. The first version of the application is coded using ODBAPI to interact with both data stores and is referred to as *Version₁*. Whereas the second version is implemented using proprietary APIs of each data store and it is called

Version₂. Indeed, developers should use the JDBC API and the MongoDB API to interact with the relational and the MongoDB data stores respectively.

In Table 6.5, we illustrate a comparison chart between the implementation time and the lines number of code sources of the same application implemented with/without ODBAPI. It is worthy to say that the implementation time includes the time of the familiarization to the APIs. It is comparable in both cases because the four developers were already familiar with JDBC (just one API to learn for both scenarios). To count lines number, we start from the first line in a JAVA class (i.e. the name of a class) and we finish by the closing brace of the JAVA class. We do not take into account the lines containing comments and the imports of the libraries and the dependencies. To compare between the two versions, we propose to evaluate the gain in terms of implementation time and the lines of source code number. The gain in terms of implementation time (resp. lines number) is obtained by calculating the ratio between the difference between the implementation time (resp. lines number) of *Version₂* and the implementation time (resp. lines number) of *Version₁*, and the implementation time (resp. lines number) of *Version₂*. We define the following formulas:

$$gain_{implementation\ time} = \left(\frac{time_{version2} - time_{version1}}{time_{version2}} \right) * 100$$

$$gain_{lines\ number} = \left(\frac{linesNumber_{version2} - linesNumber_{version1}}{linesNumber_{version2}} \right) * 100$$

Using these two formulas, we obtain an average *gain_{implementation time}* equals to 49,5% and an average *gain_{lines number}* equals to 63,68%. These results are important since they prove that we really alleviate the burden on developers and we ease his/her task. Indeed, the gain in terms of time encourages the use of ODBAPI since (1) it improves the developers productivity and (2) it increase the adoption of ODBAPI. Whereas the gain in terms of lines number promotes developers to use ODBAPI. In fact, it shows the portability of applications. In addition, applications may migrate from one Cloud environment to another frequently. Hence the adaptation of the source code of the application to the new Cloud environment will be easier and lighter. Finally, it decreases the errors number in application and developers fix bugs in their codes sources easily (answering therefore Question₂).

	Developer ₁		Developer ₂		Developer ₃		Developer ₄	
	Version ₁	Version ₂	Version ₁	Version ₂	Version ₁	Version ₂	Version ₁	Version ₂
Implementation time (mn)	40	66	55	103	45	112	57	118
Lines of source code number	20	57	22	52	23	62	20	65
<i>gain_{implementation time}</i> (%)	39,94		46,60		59,82		51,69	
<i>gain_{lines number}</i> (%)	64,91		57,69		62,9		69,23	

Table 6.5: Evaluation of ODBAPI in terms of the implementation time and source code lines number

In Listing 6.1, we present an example of an application programmed using ODBAPI. This application interacts with a MongoDB and a MySQL data stores. In fact, it enables the creation of an entity set called *person* of type MongoDB (see lines 3-6). Then it creates a document having an *id* equals to 1 (see lines 9-12), it retrieves it (see lines 15-17) and it deletes it (see lines 20-22). In addition, this sample of code source allows the creation of an entity set of type MySQL (see lines 25-27). This entity set is referred to as *city*. It also enables to create (see lines 30-32), retrieve (see lines 35-36) and delete an entity (see lines 39-40). In Listing 6.2, we present the same application implemented using the proprietary APIs of MySQL (see lines 3-41) and MongoDB (see lines 43-52).

```

1  ...
2  /*****Create an entity set of type MongoDB*****/
3  JSONObject jsonEntitySet1 = getJsonQuery("file/EntitySet1.json");
4  CreateEntitySetImpl ces = new CreateEntitySetImpl();
5  ces.createEntitySet("http://localhost:8182/odbapi/entityset/person",
6  "database/MongoDB", "test", jsonEntitySet1);
7
8  /*****Insert a document in a MongoDB database*****/
9  JSONObject jsonEntity1 = getJsonQuery("file/Entity1.json");
10 CreateEntityImpl ce = new CreateEntityImpl();
11 ce.createEntity("http://localhost:8182/odbapi/entityset/person/entity/1",
12 "database/MongoDB", "test", jsonEntity1);
13
14 /*****Retrieve a document in MongoDB database*****/
15 RetrieveEntityImpl re = new RetrieveEntityImpl();
16 re.retrieveEntity("http://localhost:8182/odbapi/entityset/person/entity/1",
17 "database/MongoDB", "test");
18
19 /*****Delete a document in MongoDB database*****/
20 DeleteEntityImpl de = new RetrieveEntityImpl();
21 de.retrieveEntity("http://localhost:8182/odbapi/entityset/person/entity/1",
22 "database/MongoDB", "test");
23
24 /*****Create an entity set of type MySQL*****/
25 JSONObject jsonEntitySet2 = getJsonQuery("file/EntitySet2.json");
26 ces.createEntitySet("http://localhost:8182/odbapi/entityset/city",
27 "database/MySQL", "world", jsonEntitySet2);
28
29 /*****Insert a tuple in MySQL database*****/
30 JSONObject jsonEntity2 = getJsonQuery("file/EntitySet2.json");
31 ce.createEntitySet("http://localhost:8182/odbapi/entityset/city/entity/2_",
32 "database/MySQL", "world", jsonEntity2);
33
34 /*****Retrieve a tuple in MySQL database *****/
35 re.retrieveEntity("http://localhost:8182/odbapi/entityset/city/entity/2_",
36 "database/MySQL", "world");
37
38 /*****Delete a tuple in MySQL database *****/
39 de.retrieveEntity("http://localhost:8182/odbapi/entityset/city/entity/2_",
40 "database/MySQL", "world");
41 ...

```

Listing 6.1: Sample of a code source of an ODBAPI-based application interacting with a MySQL and MongoDB data stores

```

1  ...
2  /*****      JDBC part      *****/
3  Connection conn = null;
4  Properties connectionProps = new Properties();
5  connectionProps.put("user", "root");
6  connectionProps.put("password", "root");
7  conn = DriverManager.getConnection("jdbc:mysql://localhost:3306"+ " /world",
8  connectionProps);
9  System.out.println("Connected to database");
10 String query0="DROP_TABLE_city";
11 Statement stmt0 = conn.createStatement();
12 stmt0.executeUpdate(query0);
13 String query="CREATE_TABLE_city(id_INT_PRIMARY_KEY,nom_VARCHAR(100));";
14 Statement stmt = conn.createStatement();
15 stmt.executeUpdate(query);
16 Statement stmt2 = conn.createStatement();
17 String query2="INSERT INTO_city_VALUES(1,'France')";
18 stmt2.executeUpdate(query2);
19 Statement stmt3 = conn.createStatement();

```

```

20 String query3="SELECT_*_FROM_city_WHERE_id=1;";
21 ResultSet rs = stmt3.executeQuery(query3);
22 while (rs.next()) {
23     System.out.println(rs.getInt(1)+"_"+rs.getString(2));
24 }
25 Statement stmt4 = conn.createStatement();
26 String query4="DELETE_FROM_city_WHERE_id=1;";
27 stmt4.executeUpdate(query4);
28 Statement stmt5 = conn.createStatement();
29 String query5="SELECT_*_FROM_city;";
30 ResultSet rs5 = stmt5.executeQuery(query3);
31 while (rs5.next()) {
32     System.out.println(rs5.getInt(0)+"_"+rs5.getString(1));
33 }
34 stmt.close();
35 stmt2.close();
36 stmt3.close();
37 stmt4.close();
38 stmt5.close();
39 rs.close();
40 rs5.close();
41 conn.close();
42 /***** MongoDB API part *****/
43 MongoClient mongoClient = new MongoClient();
44 MongoDB database = mongoClient.getDatabase("test");
45 MongoCollection<Document> collection = database.getCollection("person");
46 Document doc = new Document("name", "Rami").append("type", "sellami");
47 collection.insertOne(doc);
48 Document myDoc = collection.find(eq("name", "Rami")).first();
49 System.out.println(myDoc.toJson());
50 collection.deleteOne(eq("name", "Rami"));
51 Document myDoc1 = collection.find(eq("name", "Rami")).first();
52 System.out.println(myDoc1.toJson());
53 ...

```

Listing 6.2: Sample of a code source of a proprietary APIs-based application interacting with a MySQL and MongoDB data stores

To sum up, in this section we answer the Question₂. Indeed, it is not costly in terms of time and manpower. Then, we conducted some experiments to demonstrate that ODBAPI really facilitates developers tasks. It increases their productivity. In addition, it encourages them to adopt it in their applications.

6.5 Complex Queries Optimization

In order to validate our approach for optimal execution plans generation and the proposed cost model, we first verified that their behaviors are similar to what we have intuitively expected. In fact, to check the reliability of Algorithm 1, we tested it with different scenarios that either are trivial or allow us to intuitively predict the optimal execution plan.

Afterwards, we propose to perform some experiments in order to evaluate the cost of an optimal execution plan. In parallel with this, we evaluate the running time our algorithm (i.e. the optimization time) and the size of the research area in terms of number of nodes in all execution plans. These experiments will enable us to answer the following questions:

- **Question₃**: Does the parallelism of operations execution reduces the total cost of a query?
- **Question₄**: Does the external joins reduce the total cost of a query?
- **Question₅**: Does the parallelism and the external joins are complementary and reduce the total cost of a query?

To perform these experiments, we propose to use two variants of join queries that we illustrate by a directed graph. In this latter, the nodes represent the entity sets and the edges represent the join between two given entity sets. The first variant has a linear form in which each entity set may join at least one entity set and at most two entity sets (see Figure 6.5a). This kind of queries is usually used in relational data stores. Whereas the second variant has a star form in which all entity sets invoked in a query join only the same entity set (see Figure 6.5b). This kind of queries is generally used to interact with data warehouses. These two variants of queries are executed in four different scenarios in which we parallelize subqueries execution (or not) and we enable external joins execution (or not). These scenarios are as follows :

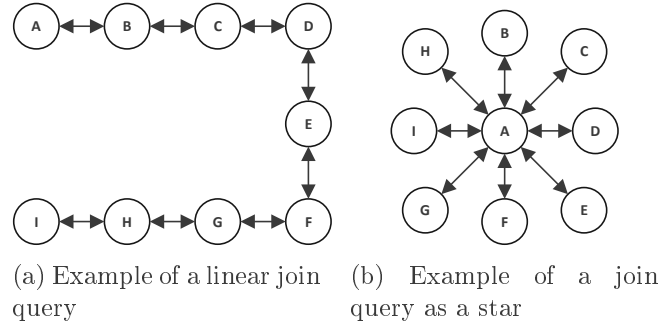


Figure 6.5: Example of a join queries

- **Scenario 1:** It consists in sequentially running a join query while only using the VDS and integrated data stores for the push-down operations. It is noteworthy that this scenario is the most naive one.
- **Scenario 2:** It consists in sequentially running a join query using the VDS and the integrated data stores for the external joins and the push-down operations.
- **Scenario 3:** It consists in running a join query in parallel while only using the VDS. It is worthy to say that the parallelism, in this case, is done at that integrated data stores to execute push-down operations.
- **Scenario 4:** It consists in running a join query in parallel using the VDS and the integrated data stores. We parallelize the execution of the external joins and the push-down operations.

To conduct these experiments, we propose to use four queries for each variant of join queries (i.e. linear or star) and to increment the number of joins by two starting by three joins. In the following, we present an example of a linear join query and a join query as a star. Both of these queries contain five joins and involve six entity sets (i.e. A , B , C , D , E and F). Each entity set contains at least a join attribute and may contain some other attributes over which we apply projection and selection operations.

LINEAR JOIN QUERY

```

> POST /odbapi/query
> Database-Type: database/VirtualDataStore
> Accept: application/json
> {
>   "select" : ["D.tag", "idA", "C.dataC", "F.tag"],
>   "from" : ["A", "B", "C", "D", "E", "F"],
>   "where" :
>   [
>     "rank=3",
>     "A.idA=B.fkeyBA",
>     "B.idB=C.fkeyCB",
>     "D.fkeyDA=A.idA",
>     "E.fkeyED=D.idD",
>     "F.fkeyFE=E.idE"
>   ]
> }

```

JOIN QUERY AS A STAR

```

> POST /odbapi/query
> Database-Type: database/VirtualDataStore
> Accept: application/json
> {
>   "select" : ["D.tag", "idA", "C.dataC", "F.tag"],
>   "from" : ["A", "B", "C", "D", "E", "F"],
>   "where" :
>   [
>     "rank=3",
>     "A.idA=B.fkeyBA",
>     "A.idA=C.fkeyCA",
>     "A.idA=D.fkeyDA",
>     "A.idA=E.fkeyEA",
>     "A.idA=F.fkeyFA",
>   ]
> }

```

During our experiments, we consider the following constraint on the catalog's parameters. Indeed, we set the parameters α , β , and γ to the same value since we assign the same importance to the cost of the CPU, the input/output, and the communication respectively. Then, we set the parameters *convert*, *ship*, *scan*, *load*, and *initETSL* to the same value at the VDS and the integrated data stores. We suppose that the integrated data stores are more efficient in executing join queries than the VDS (especially when it comes to handle a very large size of data). Hence, the values of *initJoin* and *join* parameters are larger at the VDS. Finally, we approximately set the same cardinality and the selectivity parameters

for the involved entity sets.

In Table 6.6 and Table 6.7, we showcase the results of executing a linear join query and a join query as a star while changing the number of joins and the scenarios. We also present in Figure 6.6 and Figure 6.7 the total cost evaluation according to the scenarios 1 to 4 and the joins number. Thanks to these results, we validate the key ingredients of our approach to generate an optimal execution plan (independently of the variant of the join query). Against this, we compute the average gain in the four scenarios. We provide an overview of the gains between the different scenarios in Table 6.8. The gain is obtained by calculating the ratio between the difference between the total cost of the worst scenario and the total cost of the best scenario, and the total cost of the worst scenario. We use the following formula :

$$gain = \left(\frac{totalCost_{naiveScenario} - totalCost_{bestScenario}}{totalCost_{naiveScenario}} \right) * 100$$

:

- **Importance of the external joins:** In both scenarios 2 and 4 we show that involving the integrated data stores in the execution of join queries is important. Indeed, we obtain a better total cost compared to scenarios that are exclusively using the VDS. We compute the average gain between the gain in scenarios 1 and 2, and in scenarios 3 and 4. We have an average gain equals to 89,23%.
- **Importance of the parallelism:** In both scenarios 3 and 4 we prove that parallelizing the execution of external joins and pushdown operations is beneficial and generates an important gain compared to the two other scenarios (in which the execution is sequential). The average gain of this two scenarios compared to the two others is equal to 83,61%.
- **Importance of combining different optimization strategies:** In the scenario 4, we maximize the work done by the integrated data stores and we execute queries in parallel. This scenario has the best total cost compared to the naive scenario which is the scenario 1. Indeed, the average gain is equal to 99,97%.
- **A small optimization time:** Although we propose a dynamic programming based algorithm, the optimization time is small and reasonable especially when we take into account the size of the research area.

At this stage, we answer the Question₃ and Question₄. Indeed, we prove the importance of the external joins and the parallelism to obtain an optimal total cost.

In Figure 6.8, we showcase an example of an optimal execution plan of a linear join query. This query involves eleven entity sets. First we remark that we only have nine nodes of type push-down. Indeed, we have defined in the catalog that the entity sets *B* and *C* belongs to the data store *S2* and this data store supports

		3 joins	5 joins	7 joins	9 joins
Scenario 1	Total cost (10^4)	16900,61	113605,9	874579,4	1092576,37
	Optimization time(ms)	6	14	31	116
	Research Area (nodes)	8	26	43	94
Scenario 2	Total cost (10^4)	1215,24	9088,11	37046,65	51682,33
	Optimization time (ms)	10	36	60	149
	Research Area (nodes)	36	285	605	2050
Scenario 3	Total cost (10^4)	27,73	249,62	3167,34	4204,53
	Optimization time (ms)	7	15	32	119
	Research Area (nodes)	8	26	43	94
Scenario 4	Total cost (10^4)	6,06	42,51	535,12	693,76
	Optimization time (ms)	11	44	72	167
	Research Area (nodes)	36	285	605	2005

Table 6.6: Results obtained for the case of linear queries

		3 joins	5 joins	7 joins	9 joins
Scenario 1	Total cost (10^4)	23180,71	46897,97	119899,13	305455,02
	Optimization time(ms)	8	33	126	1090
	Research Area (nodes)	16	86	456	2314
Scenario 2	Total cost (10^4)	1080,46	3039,93	4853,58	8158,75
	Optimization time (ms)	19	67	196	859
	Research Area (nodes)	110	821	5475	34487
Scenario 3	Total cost (10^4)	30,06	57,58	113,11	219,88
	Optimization time (ms)	9	40	136	1113
	Research Area (nodes)	16	86	456	2314
Scenario 4	Total cost (10^4)	5,85	10,23	16,43	24,85
	Optimization time (ms)	22	86	228	958
	Research Area (nodes)	110	821	5475	34487

Table 6.7: Results obtained for the case of queries as star

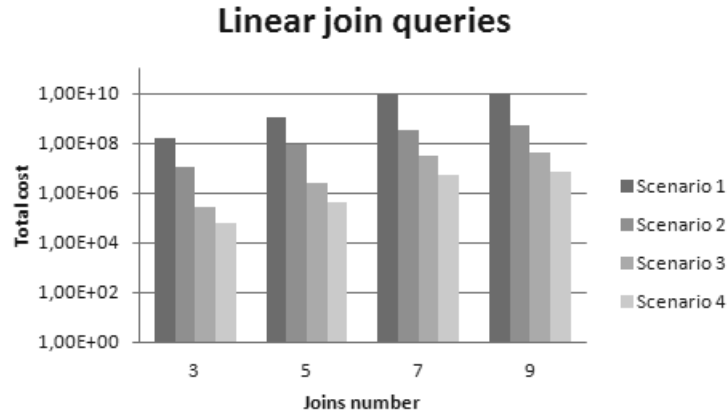


Figure 6.6: Total cost evaluation according to the scenarios 1 to 4 and the joins number- Case of linear join queries (Logarithmic Scale)

the execution of join queries. Hence, we merge these two nodes into a single node. Added to that, we delegate the execution of the join between the entity sets B and C to their proprietary data store. It is the same case of the entity sets G and

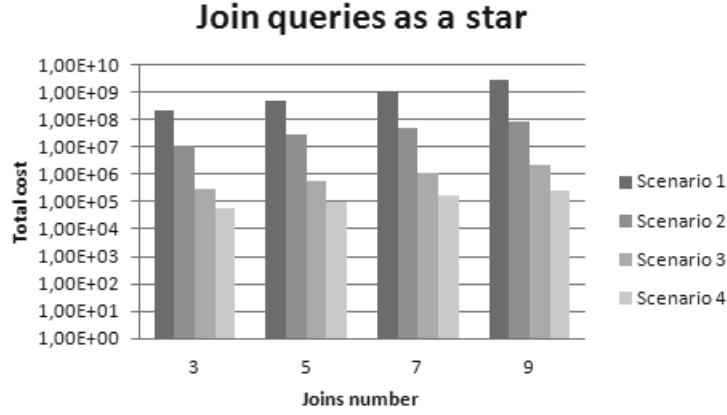


Figure 6.7: Total cost evaluation according to the scenarios 1 to 4 and the joins number- Case of join queries as a star (Logarithmic Scale)

	Linear join query				Join query as a star				Average gain
	3 joins	5 joins	7 joins	9 joins	3 joins	5 joins	7 joins	9 joins	
$gain_{Scenario2 \backslash Scenario1}$	92,8	92,8	95,8	95,3	95,3	93,5	96,01	97,3	94,85
$gain_{Scenario3 \backslash Scenario1}$	99,81	99,8	99,6	99,6	99,9	99,9	99,9	99,9	99,8
$gain_{Scenario4 \backslash Scenario1}$	100	100	99,9	99,9	100	100	100	100	99,97
$gain_{Scenario4 \backslash Scenario2}$	99,5	98,6	98,7	98,7	99,5	99,7	99,7	99,7	99,26
$gain_{Scenario4 \backslash Scenario3}$	82,96	82,96	83,1	83,05	80,5	82,22	85,46	88,69	83,61

Table 6.8: Gain between scenarios

H that belong to the data store $S5$. Second, there is a part of the work done by the integrated data stores. For instance, the external join between the entity sets J and K in the data store $S7$ and the external join between a resulting entity set from an external join and a resulting one from a VDS join. Third, we invoke in some other cases the VDS to execute join when it is costly if an integrated data store executes it.

In Table 6.9 and Table 6.10, we illustrate the results obtained after executing queries across a very large size of data (see the total cost evaluation according to the joins number in Figure 6.9 and Figure 6.10). It is noteworthy that values of the total cost in these figures are divided by 10^{11} in order to clearly show the gain between different scenarios. To conduct these experiments, we propose to use the same four queries used below and the same scenarios. We just modify the catalog by increasing the cardinality of entity sets. Indeed we multiply each cardinality by 100. These experiments show that we also obtain an optimal total cost in the case of scenario 4. We provide an overview of the gains between the different scenarios in Table 6.11. The average gain between the scenarios 1 and 4 is equals to 100%. The gain becomes more important compared to the previous results. Hence, we conclude that maximizing the work of integrated data stores is very important especially when it comes to query a very large amount of data.

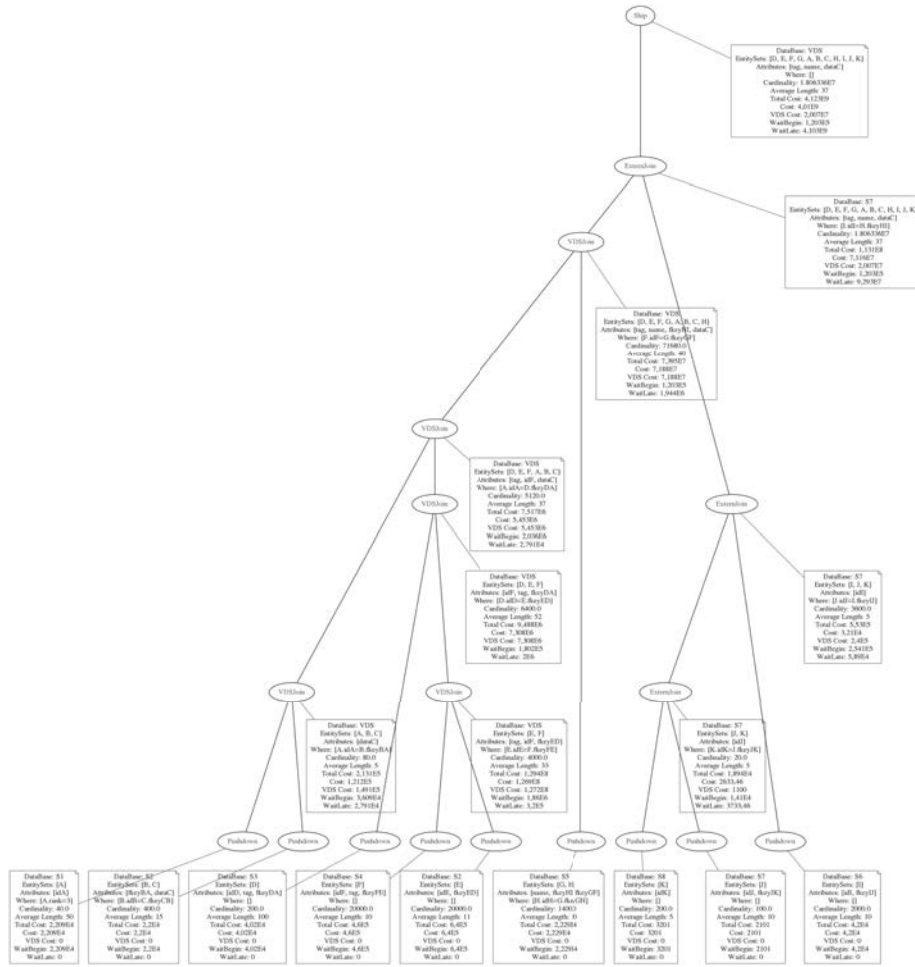


Figure 6.8: Example of an execution plan - Case of a linear join query

		3 joins	5 joins	7 joins	9 joins
Scenario 1	Total cost (10^{11})	2, 12.10 ⁵	3, 68.10 ¹⁰	1, 49.10 ¹¹	1, 56.10 ¹¹
	Optimization time(ms)	6	15	31	114
	Research Area (nodes)	8	26	43	94
Scenario 2	Total cost (10^{11})	2, 37.10 ³	4, 26.10 ⁸	4, 03.10 ⁹	4, 58.10 ⁹
	Optimization time (ms)	10	37	61	151
	Research Area (nodes)	36	285	605	2050
Scenario 3	Total cost (10^{11})	1, 61.10 ²	1, 81.10 ⁷	2, 53.10 ⁸	3, 19.10 ⁸
	Optimization time (ms)	6	15	32	119
	Research Area (nodes)	8	26	43	94
Scenario 4	Total cost (10^{11})	9, 35	2, 64.10 ⁶	4, 55.10 ⁷	5, 74.10 ⁷
	Optimization time (ms)	11	43	74	166
	Research Area (nodes)	36	285	605	2005

Table 6.9: Results obtained for the case of linear queries - Big data context

6.6 Conclusion

Although our solution is intended to support developers of multiple data stores based applications, consumers do not accept to adopt a solution that did not proved

		3 joins	5 joins	7 joins
Scenario 1	Total cost (10^{11})	$1,28.10^8$	$3,28.10^{16}$	$2,63.10^{17}$
	Optimization time(ms)	8	33	127
	Research Area (nodes)	16	86	456
Scenario 2	Total cost (10^{11})	$1,30.10^6$	$3,48.10^{14}$	$2,43.10^{15}$
	Optimization time (ms)	19	69	222
	Research Area (nodes)	110	821	5475
Scenario 3	Total cost (10^{11})	$1,61.10^2$	$2,57.10^6$	$5,15.10^6$
	Optimization time (ms)	9	36	138
	Research Area (nodes)	16	86	456
Scenario 4	Total cost (10^{11})	8,35	$1,34.10^5$	$3,29.10^5$
	Optimization time (ms)	22	84	228
	Research Area (nodes)	110	821	5475

Table 6.10: Results obtained for the case of queries as star - Big data context

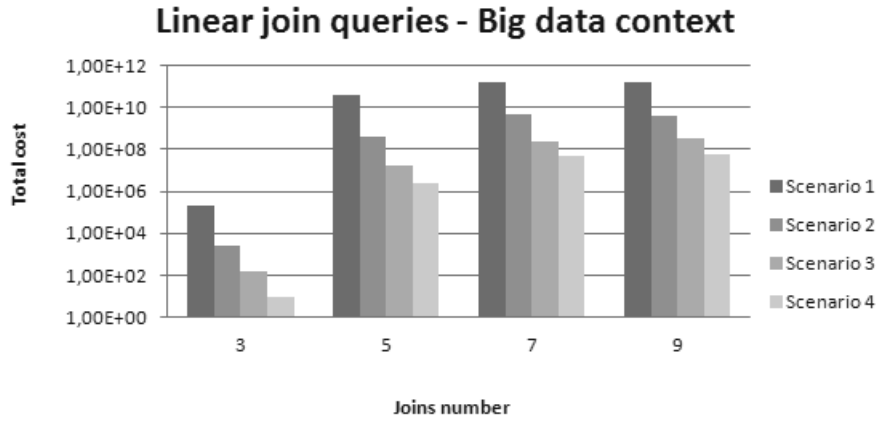


Figure 6.9: Total cost evaluation according to the scenarios 1 to 4 and the joins number- Case of linear join queries in a big data context (Logarithmic Scale)

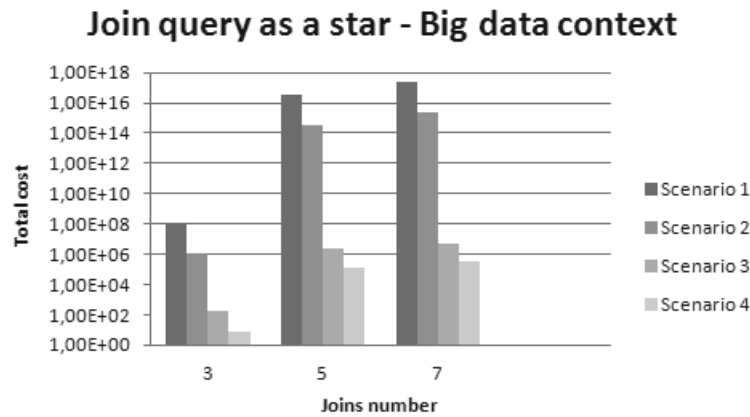


Figure 6.10: Total cost evaluation according to the scenarios 1 to 4 and the joins number- Case of join queries as a star in a big data context (Logarithmic Scale)

	Linear join query				Join query as a star			Average gain
	3 joins	5 joins	7 joins	9 joins	3 joins	5 joins	7 joins	
$gain_{Scenario2 \backslash Scenario1}$	98,9	98,8	97,1	97,4	99	99	99	98,45
$gain_{Scenario3 \backslash Scenario1}$	99,9	100	99,8	99,8	100	100	100	99,92
$gain_{Scenario4 \backslash Scenario1}$	100	100	100	100	100	100	100	100
$gain_{Scenario4 \backslash Scenario2}$	99,6	99,4	98,9	98,6	100	100	100	99,5
$gain_{Scenario4 \backslash Scenario3}$	94,2	85,4	82	82	94,8	94,8	93,6	89,57

Table 6.11: Gain between scenarios - Big data context

its efficiency in real environments. For this sake, we presented in this chapter some implementations and experiments to validate our approach. Indeed, we presented some proofs of concept proposed and integrated in the OpenPaaS project. We introduced the current state of ODBAPI as well as three applications implemented using it. Then we presented a tool allowing the automatic discovery of Clouds data stores. Afterwards, we presented the different experiments that we realized to prove the efficiency of our work. First, we evaluate the overhead of ODBAPI compared to the proprietary APIs. Indeed, we obtain a quiet acceptable overhead that it does not prevent developers to use our API. Second, we evaluate the gains in terms of implementation time and number of lines of source code in order to show the easiness of using ODBAPI. In fact, we proved that we really alleviate the burden on developers (e.g. during application implementation, application validation and testing, etc.). This will increase their productivity. In addition, we show that ODBAPI is portable and user-friendly. Third, we evaluate our algorithm to generate optimal execution plan. This enables us to prove the importance of parallelizing queries execution and the invoking integrated data stores to execute complex queries.

To sum up, the obtained results are encouraging. We showed the feasibility and efficiency of our approach of supporting multiple data stores applications in Cloud environments. At the same time, they opened new perspectives to our work that we will introduce in the next chapter.

Conclusion & Future Works

Contents

7.1 Contributions	123
7.2 Perspectives	125

7.1 Contributions

The production of a huge amount of data and the emergence of Cloud computing have introduced new requirements for data management. Many applications need to interact with several heterogeneous data stores depending on the type of data they have to manage: traditional data types, documents, graph data from social networks, simple key-value data, etc. Interacting with heterogeneous data models via different APIs, and multiple data stores based applications imposes challenging tasks to their developers. Indeed, programmers have to be familiar with different APIs. In addition, the execution of complex queries over heterogeneous data models cannot, currently, be achieved in a declarative way as it is used to be with mono-data store application, and therefore requires extra implementation efforts. Moreover, developers need to master and deal with the complex processes of Cloud discovery, and application deployment and execution.

In this manuscript, we proposed an integrated set of models, algorithms and tools aiming at alleviating developers task for developing, and deploying multiple data stores applications in Cloud environments. Indeed, we introduced four main contributions:

- **Unifying relational and NoSQL data models:** The main key ingredient of data stores integration is to ensure a unified view of the heterogeneous integrated data stores. In the light of this, we proposed a unified data model that allows to provide a unique view on relational and NoSQL data stores. Then we presented a global schema as the set of collections used in the application and expressed with the unified data model enriched with a set of the refinement rules. Finally, we proposed our query algebra defined based on a formal definition of the unified data model and allowing complex queries expression and execution.

- ODBAPI to uniquely interact with relational and NoSQL data stores:** Based on our unified data model, we defined a generic resource model to represent the different elements of heterogeneous data stores in a Cloud environment. Then, we proposed a unique REST API that enables the management of the described resources in a uniform manner. This API is called ODBAPI and allows the execution of CRUD and complex operations on relational and NoSQL data stores. The highlights of ODBAPI are twofold: (i) decoupling cloud applications from data stores in order to facilitate their development and their migration, and (ii) easing the developers task by lightening the burden of managing different APIs. It is noteworthy that in the current version of ODBAPI server, we took into account four data stores: MySQL, Riak, CouchDB, and MongoDB.
- Virtual data stores for complex queries execution:** Although ODBAPI enables the execution of queries across relational and NoSQL data stores, it does not support multi-sources queries execution. To cover this gap, we proposed the VDS that is a mediator based component integrating relational and NoSQL data stores. The role of the VDS is fivefold: (1) it parses the query in order to create the corresponding algebraic tree, (2) it performs algebraic optimization by pushing down unary operations, (3) it annotates the algebraic tree with information from the catalog, (4) it generates an optimal execution plan, and (5) it evaluates the query. In order to ensure this, we proposed a cost model composed of a set of cost formulas for each elementary operation. This cost model is implemented in our algorithm generating the optimal execution plan that is based on a dynamic programming approach.
- Data stores discovery and automatic application deployment:** Once the developer has completed the development of his/her application, we provided him the possibility to express his/her application requirements (mainly data stores) in the *abstract application manifest*. Then, he/she sends it to the *Cloud providers discovery* component to elect the appropriate Cloud provider to the application requirements. Indeed, this component discovers the capabilities of Clouds data stores and returns these capabilities in the *offer manifest*. Based on that, we run our *matching algorithm* to elect the most suitable Cloud provider to the application requirements and generate the *deployment manifest* of the application. Finally, we deploy the application using the COAPS API that takes as input the *deployment manifest*. We extended COAPS in order to support multiple data stores based applications deployment in Cloud environments.

The contributions that we presented in this manuscript have been validated by developing a prototype of the proposed solutions that is used to implement use cases from the OpenPaaS project. In addition, several experiments have been performed in order to evaluate the overhead of ODBAPI and our algorithm to optimize and evaluate complex queries.

To sum up, these contributions respect the research objectives that we dressed at the beginning of this thesis as follows (see Section 1.3):

- **Cloud Computing and Big Data:** The different components that we proposed in this manuscript are intended to work in a Cloud computing area. We support all the lifecycle of an application from the discovery of Cloud providers to the Cloud deployment, including the coding of the application using ODBAPI which is a REST-base API. In addition, these different contributions take into account the very large volume of data and the variety of data stores regarding the Big Data area. In fact, our VDS-based solution supports the optimization and evaluation of complex queries over voluminous data. Besides, we consider multiple data stores based applications in the different components.
- **NoSQL data stores support:** The different proposed solutions take into account relational and NoSQL data stores.
- **Multiple data stores based applications:** Thanks to our end-to-end solution, we support support multiple data stores based applications lifecycle in a Cloud environment.
- **Easy access to heterogeneous data stores:** In this thesis, we facilitate the access to relational data stores thanks to our unified data model. Using this latter, developers can easily express complex queries and the VDS is able to evaluate and optimize their execution.
- **Automation:** In order to relieve utmost developers task by removing the burden of managing the high heterogeneity of data stores in Cloud environments, the different components of our work automatically function.

7.2 Perspectives

During our work, we faced different complex problems. We solved various of them and we included others in our future work. The evaluation also opened new research perspectives to follow in short, medium and long term.

As short term work, we aim to go further in applying ODBAPI and the VDS to other qualitatively and quantitatively various scenarios and real use cases. This allows us to identify possible discrepancies and to make our work more reliable for public use. In addition, we are aware that the VDS component is still in early stage and we would like to enhance it and evaluate its efficiency and accuracy by adding new kind of operations (e.g. aggregates). Indeed, it only enables the optimization and evaluation of a restricted set of complex queries regarding the expressiveness of our query algebra. Moreover, our execution model needs to be enhanced including non blocking executions. Finally, we focus to enhance our Cloud data stores discovery component by supporting other deployment APIs (e.g. ROBOCONF ⁶). This will enable us to render our manifest-based approach more

⁶Home page of ROBOCONF: <http://roboconf.net/fr/index.html>

generic and to test it with new solutions. To do so, we have first to discover this API and try to express its required inputs in the deployment manifest.

As medium term future work, we will focus on two major related challenges. First, we aim to enhance our unified data model and associated query algebra. In fact, in the proposed unified data model, we enable the representation of complex attributes (e.g. an attribute composed by multiple attributes, a multivalued attribute, etc.). However this family of attributes is not optimally used. Even if, we defined the nest and unnest operations, we do not support them in the VDS component and especially in optimization process. For that, we would like to extend our cost model to optimally execute queries targeting complex objects in a data model in the VDS.

Second, our unified data model suffers from a lack of semantics and exposes some ambiguities. Even if we accomplish it by a set of refinement rules, these latter are not sufficient. For instance, we may encounter two attributes that are syntactically written in the same manner; but they do not semantically denote the same meaning (e.g. *name* may denote the full name of a person or his/her first name). In addition, one attribute may have different meaning (e.g. the attribute *conferenceRanking.conference* may denote a ranking of a conference or a journal). To cover this gap, we propose to define an ontology to semantically enrich our unified data model and remove these ambiguities (e.g. [45, 66]). This will enable us to more automate our approach and efficiently query data. In addition, we can enrich our unified data model using virtual entity sets like relational views. These views may be modeled as new virtual entity sets in the global schema defined as query expressions on actual entity sets.

Finally, as long term work, we will take into account graph data stores. Indeed, this type of data stores exposes a very specific data model since data belong either to the nodes or to the edges. Mapping such data stores models in our unified data model is not possible today. We believe that considering this kind of data stores in our unified data model is mandatory because it is a family of NoSQL data stores and it is widely used nowadays. To cope with this challenge, we may consider one intuitive solution that consists in representing the nodes set (resp. edges set) by an entity set. Hence, we illustrate a graph data store by two entity sets in our unified data model. Once we achieve that, we will propose a graph data store based implementation of ODBAPI.

Afterwards, we aim to have a better support for the execution of queries over very large volumes of data. Although we support some optimization and evaluation of queries across voluminous data, we are aware that we can further improve their execution. To do so, we plan to investigate the integration of parallel data execution models such as Hadoop MapReduce [67, 68] or Apache Spark [69] in our execution process. One possible solution for that is to encapsulate big data sources by ODBAPI wrappers implemented on top of Map/Reduce and/or Spark. These wrappers have to transform our queries in terms of HiveQL [70] or SQLSpark [71].

Bibliography

- [1] IBM, Paul Zikopoulos, and Chris Eaton. *Understanding Big Data: Analytics for Enterprise Class Hadoop and Streaming Data*. McGraw-Hill Osborne Media, 1st edition, 2011.
- [2] P.J. Sadalage and M.J. Fowler. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Always learning. Addison Wesley Professional, 2012.
- [3] Mohamed Sellami, Yangui Sami, Mohamed Mohamed, and Samir Tata. *The Compatible One Application and Platform Service (COAPS) API specification*. Telecom SudParis, Computer Science Department, 2013.
- [4] Christian Baun, Marcel Kunze, Jens Nimis, and Stefan Tai. *Cloud Computing - Web-Based Dynamic IT Services*. Springer, 2011.
- [5] Andrew McAfee and Erik Brynjolfsson. Big data: The management revolution. (cover story). *Harvard Business Review*, 90(10):60–68, 2012.
- [6] Chang Fay and et al. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2), 2008.
- [7] Brian F. Cooper and et al. Pnuts: Yahoo!’s hosted data serving platform. *PVLDB*, 1(2):1277–1288, 2008.
- [8] Giuseppe DeCandia and et al. Dynamo: amazon’s highly available key-value store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles, SOSP 2007, Stevenson, Washington, USA, October 14-17*, pages 205–220, 2007.
- [9] Tim Kraska, Martin Hentschel, Gustavo Alonso, and Donald Kossmann. Consistency rationing in the cloud: Pay only when it matters. *PVLDB*, 2(1):253–264, 2009.
- [10] NIST. Final version of nist cloud computing definition published. <http://www.nist.gov/itl/csd/cloud-102511.cfm>, 2011.
- [11] E. F. E, F. Codd. Derivability, redundancy and consistency of relations stored in large data banks. *SIGMOD Rec.*, 38(1):17–36, June 2009.
- [12] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems (2Nd Ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1999.
- [13] Jim Gray. The transaction concept: Virtues and limitations, 1981.
- [14] Pramod J. Sadalage and Martin Fowler. *NoSQL distilled : a brief guide to the emerging world of polyglot persistence*. Addison-Wesley, Upper Saddle River, NJ, 2013.

- [15] Eric A. Brewer. Towards robust distributed systems (abstract). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing, July 16-19, 2000, Portland, Oregon, USA.*, page 7, 2000.
- [16] E. Brewer. Cap twelve years later: How the "rules" have changed. *Computer*, 45(2):23–29, Feb 2012.
- [17] Daniel Abadi. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *IEEE Computer*, 45(2):37–42, 2012.
- [18] Neal Leavitt. Will nosql databases live up to their promise? *IEEE Computer*, 43(2):12–14, 2010.
- [19] Maurizio Lenzerini. Data integration: A theoretical perspective. In *Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 3-5, Madison, Wisconsin, USA*, pages 233–246, 2002.
- [20] Amit P. Sheth and James A. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Comput. Surv.*, 22(3):183–236, 1990.
- [21] Dennis Heimbigner and Dennis McLeod. A federated architecture for information management. *ACM Trans. Inf. Syst.*, 3(3):253–278, 1985.
- [22] Hector Garcia-Molina and et al. The TSIMMIS approach to mediation: Data models and languages. *J. Intell. Inf. Syst.*, 8(2):117–132, 1997.
- [23] Khalid Belhajjame, Norman W. Paton, Suzanne M. Embury, Alvaro A. A. Fernandes, and Cornelia Hedeler. Incrementally improving dataspaces based on user feedback. *Inf. Syst.*, 38(5):656–687, 2013.
- [24] Won Kim. Unisql/x unified relational and object-oriented database system. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, May 24-27, 1994.*, page 481, 1994.
- [25] Won Kim. On unifying relational and object-oriented database systems. In *TOOLS 1992: 6th International Conference on Technology of Object-Oriented Languages and Systems, Sydney, Australia.*, pages 5–17, 1992.
- [26] Michael J. Carey, Laura M. Haas, Peter M. Schwarz, Manish Arya, William F. Cody, Ronald Fagin, Myron Flickner, Allen Luniewski, Wayne Niblack, Dragutin Petkovic, Joachim Thomas II, John H. Williams, and Edward L. Wimmers. Towards heterogeneous multimedia information systems: The garlic approach. In *Proceedings RIDE-DOM '95, Fifth International Workshop on Research Issues in Data Engineering - Distributed Object Management, Taipei, Taiwan, March 6-7, 1995*, pages 124–131, 1995.

-
- [27] Yannis Papakonstantinou, Hector Garcia-Molina, and Jennifer Widom. Object exchange across heterogeneous information sources. In *Proceedings of the Eleventh International Conference on Data Engineering, March 6-10, 1995, Taipei, Taiwan*, pages 251–260, 1995.
 - [28] Jason McHugh, Serge Abiteboul, Roy Goldman, Dallon Quass, and Jennifer Widom. Lore: A database management system for semistructured data. *SIGMOD Record*, 26(3):54–66, 1997.
 - [29] Donald Kossmann, Tim Kraska, Simon Loesing, Stephan Merkli, Raman Mittal, and Flavio Pfaffhauser. Cloudy: A modular cloud storage system. *PVLDB*, 3(2):1533–1536, 2010.
 - [30] David Chappell. Introducing odata: Data access for the web, the cloud, mobile devices, and more. Technical report, Chappell and Associates, May 2011.
 - [31] Tao Sun and Xinjun Wang. Research on heterogeneous data resource management model in cloud environment. *International Journal of Database Theory and Application*, 6(5):141–152, 2013.
 - [32] Franck Michel, Loic Djimenou, Catherine Faron-Zucker, and Johan Montagnat. Translation of relational and non-relational databases into rdf with xr2rml. In *WEBIST 2015 - Proceedings of the 11th International Conference on Web Information Systems and Technologies, Lisbon, Portugal, 20-22 May, 2015*, pages 443–454, 2015.
 - [33] Won Kim. Web data stores (aka nosql databases): a data model and data management perspective. *IJWGS*, 10(1):100–110, 2014.
 - [34] John D. Poole. Model-driven architecture: Vision, standards and emerging technologies. In *In In ECOOP 2001, Workshop on Metamodeling and Adaptive Object Models*, 2001.
 - [35] Michael Stonebraker. Stonebraker on nosql and enterprises. *Commun. ACM*, 54(8):10–11, 2011.
 - [36] Maydene Fisher, Jon Ellis, and Jonathan C. Bruce. *JDBC API Tutorial and Reference*. Pearson Education, 3 edition, 2003.
 - [37] Till Haselmann, Gunnar Thies, and Gottfried Vossen. Looking into a rest-based universal api for database-as-a-service systems. In *12th IEEE Conference on Commerce and Enterprise Computing, CEC 2010, Shanghai, China, November 10-12, 2010*, pages 17–24, 2010.
 - [38] Mark Pollack, Oliver Gierke, Thomas Risberg, Jon Brisbin, and Michael Hunger, editors. *Spring Data*. O’Reilly Media, October 2012.

- [39] Paolo Atzeni, Francesca Bugiotti, and Luca Rossi. Uniform access to non-relational database systems: The sos platform. In *Advanced Information Systems Engineering - 24th International Conference, CAiSE 2012, Gdansk, Poland, June 25-29, 2012. Proceedings*, pages 160–174, 2012.
- [40] Luca Cabibbo. Ondm: an object-nosql datastore mapper. *Faculty of Engineering, Roma Tre University*. Retrieved June 15th, 2013.
- [41] A. B. M. Moniruzzaman and Syed Akhter Hossain. Nosql database: New era of databases for big data analytics - classification, characteristics and comparison. *CoRR*, abs/1307.0191, 2013.
- [42] AnHai Doan, Alon Y. Halevy, and Zachary G. Ives. *Principles of Data Integration*. Morgan Kaufmann, 2012.
- [43] Vanja Josifovski, Peter M. Schwarz, Laura M. Haas, and Eileen Tien Lin. Garlic: a new flavor of federated query processing for db2. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, June 3-6, 2002*, pages 524–532, 2002.
- [44] Olivier Curé and et al. Data integration over NoSQL stores using access path based mappings. In *Database and Expert Systems Applications - 22nd International Conference, DEXA 2011. Proceedings, Part I*, pages 481–495, 2011.
- [45] Olivier Curé and et al. On the potential integration of an ontology-based data access approach in NoSQL stores. *IJDST*, 4(3):17–30, 2013.
- [46] Minpeng Zhu and Tore Risch. Querying combined cloud-based and relational databases. In *2011 International Conference on Cloud and Service Computing, CSC 2011, Hong Kong, December 12-14, 2011*, pages 330–335, 2011.
- [47] John Roijackers and George H. L. Fletcher. On bridging relational and document-centric data stores. In *Big Data - 29th British National Conference on Databases, BNCOD'13*, pages 135–148, 2013.
- [48] Boyan Kolev, Patrick Valduriez, Carlyna Bondiombouy, Ricardo Jimenez-Peris, Raquel Pau, and José Orlando Pereira. CloudMdsQL: Querying Heterogeneous Cloud Data Stores with a Common Language. *Distributed and Parallel Databases*, page 41, 2015.
- [49] Martin Giese, Ahmet Soyly, Guillermo Vega-Gorgojo, Arild Waaler, Peter Haase, Ernesto Jiménez-Ruiz, Davide Lanti, Martín Rezk, Guohui Xiao, Özgür L. Özçep, and Riccardo Rosati. Optique: Zooming in on big data. *IEEE Computer*, 48(3):60–67, 2015.
- [50] Timea Bagosi, Diego Calvanese, Josef Hardi, Sarah Komla-Ebri, Davide Lanti, Martín Rezk, Mariano Rodriguez-Muro, Mindaugas Slusnys, and Guohui

-
- Xiao. The ontop framework for ontology based data access. In *The Semantic Web and Web Science - 8th Chinese Conference, (CSWS) 2014, Wuhan, China, August 8-12, 2014, Revised Selected Papers*, pages 67–77, 2014.
- [51] Carlson Mark and et al. Cloud application management for platforms, August 2012.
 - [52] SNIA. Cloud data management interface, June 2012.
 - [53] Hong Linh Truong and et al. Data contracts for cloud-based data marketplaces. *IJCSE*, 7(4):280–295, 2012.
 - [54] Hong Linh Truong and et al. Exchanging data agreements in the daas model. In *2011 IEEE Asia-Pacific Services Computing Conference, APSCC 2011, Jeju, Korea (South), December 12-15*, pages 153–160, 2011.
 - [55] Quang Hieu Vu and et al. Demods: A description model for data-as-a-service. In *IEEE 26th International Conference on Advanced Information Networking and Applications, AINA, 2012, Fukuoka, Japan, March 26-29*, pages 605–612, 2012.
 - [56] Nirnay Ghosh and Soumya K. Ghosh. An approach to identify and monitor sla parameters for storage-as-a-service cloud delivery model. In *Workshops Proceedings of the Global Communications Conference, GLOBECOM 2012, 3-7 December, Anaheim, California, USA*, pages 724–729, 2012.
 - [57] Arkaitz Ruiz-Alvarez and Marty Humphrey. An automated approach to cloud storage service selection. In *Proceedings of the 2Nd International Workshop on Scientific Cloud Computing*, ScienceCloud '11, pages 39–48, 2011.
 - [58] Arkaitz Ruiz-Alvarez and Marty Humphrey. A model and decision procedure for data storage in cloud computing. In *12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2012, Ottawa, Canada, May 13-16*, pages 572–579, 2012.
 - [59] Miranda Zhang, Rajiv Ranjan, Armin Haller, Dimitrios Georgakopoulos, Michael Menzel, and Surya Nepal. An ontology-based system for cloud infrastructure services' discovery. In *8th International Conference on Collaborative Computing: Networking, Applications and Worksharing, CollaborateCom 2012, Pittsburgh, PA, USA, October 14-17, 2012*, pages 524–530, 2012.
 - [60] Erik Wittern, Jörn Kühlenkamp, and Michael Menzel. Cloud service selection based on variability modeling. In *Service-Oriented Computing - 10th International Conference, ICSOC 2012, Shanghai, China, November 12-15, 2012. Proceedings*, pages 127–141, 2012.
 - [61] E. F. Codd. *The Relational Model for Database Management: Version 2*. Addison-Wesley Longman Publishing Co., Inc., 1990.

- [62] Serge Abiteboul and Nicole Bidoit. Non first normal form relations: An algebra allowing data restructuring. *J. Comput. Syst. Sci.*, 33(3):361–393, 1986.
- [63] Donald Kossmann. The state of the art in distributed query processing. *ACM Comput. Surv.*, 32(4):422–469, December 2000.
- [64] Juan Carlos Castrejón, Genoveva Vargas-Solar, Christine Collet, and Rafael Lozano. Exschema: Discovering and maintaining schemas from polyglot persistence applications. In *2013 IEEE International Conference on Software Maintenance, Eindhoven, The Netherlands, September 22-28, 2013*, pages 496–499, 2013.
- [65] Mohamed Sellami, Yangui Sami, Mohamed Mohamed, and Samir Tata. Paas-independent provisioning and management of applications in the cloud. In *2013 IEEE Sixth International Conference on Cloud Computing, Santa Clara, CA, USA, June 28 - July 3, 2013*, pages 693–700, 2013.
- [66] Djamel Benslimane, Mahmoud Barhamgi, Frédéric Cuppens, Franck Morvan, Bruno Defude, Ebrahim Nageba, François Paulus, Stephane Morucci, Michael Mrissa, Nora Cuppens-Boulahia, Chirine Ghedira, Riad Mokadem, Said Oulmakhzoune, and Jocelyne Fayn. PAIRSE: a privacy-preserving service-oriented data integration system. *SIGMOD Record*, 42(3):42–47, 2013.
- [67] Tom White. *Hadoop - The Definitive Guide: MapReduce for the Cloud*. O’Reilly, 2009.
- [68] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: A flexible data processing tool. *Commun. ACM*, 53(1):72–77, January 2010.
- [69] Cliff Engle, Antonio Lupher, Reynold Xin, Matei Zaharia, Michael J. Franklin, Scott Shenker, and Ion Stoica. Shark: fast data analysis using coarse-grained distributed memory. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 689–692, 2012.
- [70] Rakesh Kumar, Neha Gupta, Shilpi Charu, Somya Bansa, and Kusum Yadav. Comparison of sql with hiveql. *International Journal for Research in Technological Studies*, 1(9), 2014.
- [71] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. Spark SQL: relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 1383–1394, 2015.

Titre : Soutenir les applications utilisant des bases de données multiples dans un environnement Cloud Computing

Mots clés : Nuages des données, Données volumineuses, Bases de données NoSQL, Bases de données relationnelles, Optimisation et évaluation des requêtes de jointure, Persistance polyglotte, Découverte à base de manifestes

Résumé : Avec l'avènement du « *cloud computing* » et des « *big data* », de nouveaux systèmes de gestion de bases de données sont apparus, appelés les systèmes NoSQL. Par rapport aux systèmes relationnels, ils se distinguent par une variété des types de données et l'absence de schéma et de langages de requêtes. De nombreuses applications peuvent avoir besoin de manipuler en même temps des données stockées dans des systèmes relationnels et/ou NoSQL. Le développeur doit alors gérer au moins deux modèles de données / langages de requêtes différents pour coder son application. De plus, il doit gérer tout le cycle de vie de son application. Cette tâche est lourde et fastidieuse. Afin d'aider le programmeur tout au long du cycle de vie de son application, nous proposons

tout d'abord un modèle de données unifié pour couvrir l'hétérogénéité entre les modèles de données relationnelles et NoSQL. Ensuite, nous proposons une interface de programmation appelée ODBAPI, qui nous permet d'interagir de manière uniforme avec les bases de données relationnelles et/ou NoSQL. Puis, nous définissons la notion de bases de données virtuelles qui interviennent comme des médiateurs et interagissent avec les bases de données intégrées via ODBAPI. Elles assurent l'exécution des requêtes complexes d'une façon optimale grâce à notre modèle de coût. Enfin, nous proposons une approche automatique de découverte de bases de données et de déploiement des applications dans des environnements « *Cloud* ».

Title: Supporting multi data stores applications in cloud environments

Keywords: Cloud Computing, Big Data, REST-based API, NoSQL data stores, relational data stores, join queries optimization and evaluation, polyglot persistence, manifest based matching.

Abstract: The production of huge amount of data and the emergence of Cloud computing have introduced new requirements for data management. Many applications need to interact at the same time with several heterogeneous relational and NoSQL data stores depending. This kind of applications imposes challenging tasks to the developers. In addition, the execution of complex queries over heterogeneous data models cannot, currently, be declaratively, and therefore requires extra implementation efforts. In this thesis, we propose to alleviate developers' task during the lifecycle of multiple data stores applications in cloud environments. First, we provide a unified data model used by developers to interact with

relational and NoSQL data stores. Based on that, developers express queries using OPEN-PaaS-DataBase API (ODBAPI) which is a unique REST API allowing developers to code their applications independently of the target data stores. Second, we propose virtual data stores which act as a mediator and interact with integrated data stores wrapped by ODBAPI. This component supports the execution of single and complex queries over heterogeneous data stores. It implements a cost model to optimally execute queries and a dynamic programming based algorithm to generate an optimal query execution plan. Finally, we present a declarative and automatic approach that enables relevant Cloud providers' discovery and applications deployment.