Supporting multi data stores applications in cloud environments

Rami Sellami, Sami Bhiri, and Bruno Defude

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 store 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 paper 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 unifying data model used by applications developers to interact with heterogeneous relational and NoSQL data stores. Based on that, they 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. Finally, we present a declarative approach that enables to lighten the burden of the tedious and non-standard tasks of (1) discovering relevant cloud environment 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 is currently used to implement use cases from the OpenPaaS project.

Index Terms—REST-based API, NoSQL data stores, relational data stores, join queries, polyglot persistence, manifest based matching.

1 INTRODUCTION

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 [1]: 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 Sofwtare 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 Technology [2] (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 [3]. 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 people have also proposed to add more V 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 focus mainly on the variety property and more precisely on multiple data store based applications in the cloud.

In order to satisfy different storage requirements, cloud applications usually need to access and interact with different relational and NoSQL data stores having heterogeneous APIs. The heterogeneity of the data stores induces several problems when developing, deploying and migrating multiple data store applications. Below, we list the main four problems which we are tackling in this paper.

$Pb_1$ Heavy workload on the application developer: Nowadays data stores have different and heterogeneous APIs. Developers of multiple data store based applications need to be familiar with all these APIs when coding their applications.

$Pb_2$ No declarative way for executing complex queries: Due to the heterogeneity of the data models, there is currently no declarative way to define and execute complex queries over several data stores. This is mainly due to the absence of a global schema of heterogeneous data stores. In addition, NoSQL data stores are schemeless. That means developers have to cope themselves with the implementation of such queries.

$Pb_3$ Code adaptation: When migrating applications from one cloud environment to another, application devel-
opera have to re-adapt the application source code in order to interact with new data stores. Developers have potentially to learn and master new APIs.

**Pb4**

Tedious and non-standard processes of discovery and deployment: Once an application is developed or migrated, developers have to deploy it into a cloud provider. Discovering the most suitable cloud environment providing the required data stores and deploying the application on it are tedious and meticulous provider-specific process.

Consistency is also an important issue in multi-datastores applications. In fact, cloud data stores in general implement different consistency models (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 paper, focusing only on querying. The interested reader may read \[3\] which proposes a middleware service addressing the problem of client-centric consistency on top of eventually consistent distributed data stores (Amazon S3 for example).

In this paper we propose an integrated set of models, algorithms and tools aiming at alleviating developers’ tasks for developing, deploying and migrating multiple data stores based applications in cloud environment.

First, we define a unifying data model used by applications developers to interact with different data stores. This model tackles the problem of heterogeneity between data models and the absence of schemes in NoSQL data stores. Based on this model, developers may express and execute any type of queries using OPEN-PaaS-DataBase API (ODBAPI). This API is a streamlined and a unified REST-based API [4] for executing queries over relational and NoSQL data stores (see section 5). 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.

Second, we propose virtual data stores (VDS) to evaluate and optimize the execution of queries - especially complex ones- over different data stores (see section 6). In order to support the definition and the execution of queries over heterogeneous data models, we use the unifying data model that we accomplish with correspondence rules. Our solution is based on algebraic trees composed of data sources and algebraic operators and algebraic trees annotation.

Third, we present a declarative approach for discovering appropriate cloud environments and deploying applications on them while letting developers simply focus on specifying their storage and computing requirements (see section 7). A prototype of our approach has been developed and is currently used to implement use cases from an ongoing OpenPaas project called OPEN-PaaS (see section 8).

The remainder of the paper is organized as follows. In Section 2 we introduce the OPEN-PaaS Project and present a motivating example. In Section 3 we give an overview of our approach which we detail in the following four Sections: 4, 5, 6 and 7. In Section 8 we present the implementation and validation of our approach. In Section 9 we discuss the related work. Section 10 concludes our paper and outlines directions of future work.

2 **USE CASES AND MOTIVATION**

The OpenPaaS project [2] 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.

One of the objectives of this project is to provide support for developing and deploying multiple data store based applications in a cloud environment. The OpenPaaS project focuses in particular on relational, key/value and document data stores. In the following, we describe an example of polyglot persistence showing the problems induced by developing, migrating and deploying multiple data store based applications. We use this example in section 6 to illustrate how we evaluate and optimize the execution of complex queries involving heterogeneous data stores.

Polyglot persistence refers to situations where an application interacts with multiple data stores. For instance we can give the example where an application A needs to interact with a document data store, a relational data store and a key/value data store at the same time. Indeed, imagine that we have a document data store called dblpDB, a relational data store called personDB, and a key/value data store called RankDB. The dblpDB data store contains a collection of JSON documents called dblp. A document in this collection contains four elements: (1) the title element to denote publication titles, (2) the year element to define the publications year, (3) the author element to represent the authors name, and (4) the conference element to denote the conference acronym in which the paper has been published. In this kind of documents, we can find one or multiple values of the element author. Whereas the personDB data store contains one table called person. This table has four columns: (1) personId to denote the identifier of a person, (2) personName to define the name of a person, (3) personCountry to represent the country of a person and (4) the personAffiliation to denote the affiliation of persons. Finally, we have the data store RankDB that contains a set of key values called ConferenceRanking. In this latter, we found the elements conference and Rank to denote respectively the acronym of a conference and its rank.

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”. Answering such query is, nowadays, challenging. Indeed, since dblpDB, ConferenceRanking and personDB use different data models, it is not possible to execute such query in a declarative way anymore. The developers have to identify the sub-queries by hand, interact with each data store separately and implement the join operation by themselves. The developers need, obviously, to be familiar with APIs of the three data stores.
Imagine now that developers want to migrate their application A in order to use a CouchDB data store instead of MongoDB and that the current cloud environment does not support CouchDB data stores. In order to migrate their application, developers need (1) to re-adapt the code so that application A can interact with CouchDB API, (2) to discover another environment that can support new storage requirements, and (3) to deploy their application on the new environment. The processes of discovering environment offers and deploying applications on them are tedious and provider-specific and developers have to deal with them each time they want to deploy their applications.

3 Overview of our approach

In this section, we briefly introduce the main constituents of our approach which we detail in next sections. We show in particular how these elements enable overcoming the problems \( Pb_1 \) - \( Pb_4 \) listed above. Figure 1 depicts how these constituents intervene during the development, discovery, deployment and execution steps. Our approach relies on the following 4 elements:

Unifying data model. We define a data model which abstracts from the underlying (explicit/implicit) integrated data store models, and provide a common and unified view so that developers can define their queries over heterogeneous data stores. During the development step, the developers dispose of a global data model expressed according to our unifying model and which integrates local data store models. Our unifying data model decouples query definitions from the data stores specific languages. (contributing to resolving thereafter \( Pb_1 \) and \( Pb_2 \)).

REST API/services. Based on our unifying data model, we define a resource model upon which we develop a REST API, called ODBAPI, enabling to interact with involved data stores in a unique and uniform way. Each data store will be then wrapped behind a REST service implementing ODBAPI. Our API decouples the interactions with data stores from their specific drivers. By using our unifying data model to express the queries and ODBAPI to interact with the data stores, developers do not have to deal with various languages and APIs and do not have to adapt their code when migrating their applications (resolving thereafter \( Pb_1 \) and \( Pb_3 \)).

Virtual data stores. Wrapper REST services enable executing simple queries over the involved data stores. However, they are not meant to execute complex queries (such as join, union, etc.). In our approach, we consider virtual data store (VDS for short) a specific component responsible for executing queries submitted by a multiple data store application. A VDS (1) holds the global data model integrating the different data stores and which is specified according to our unifying data model and a set of correspondence rules, (2) is accessible as a REST service complying to the ODBAPI, and (3) 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 join queries over multiple data stores in a declarative way and take in charge the burden of their executions (resolving thereafter \( Pb_2 \)).

Dedicated components for discovery and deployment. In our approach, we consider two components, the discovery and deployment modules, responsible of finding appropriate cloud environments and deploying multiple data store applications on them respectively. As depicted in Fig. 1, developers express first 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 selects 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 (resolving thereafter \( Pb_4 \)).

In the following, we detail each of these constituents. Section 4 introduces our unifying data model. Section 5 presents our REST interface, ODBAPI. Section 6 details key steps of evaluating and optimizing queries execution by
VDS with a special focus on join queries. Section 4.1 describes the discovery and the deployment steps as well as the used manifests.

4 Unifying data model

In the following, we introduce our integrative and unifying data model used by application programmers to express their access to the different data stores they use. In order to abstract the different data models of the data stores, we have defined a unifying data model capable to express all data constructions. We present the different concepts needed to define the unifying data model (see Section 4.1) and we define them formally (see Section 4.2). The unifying 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 correspondence rules (see Section 4.3). Finally, a query algebra is presented, allowing to express complex queries (see Section 4.4).

4.1 Informal presentation of the unifying data model

Our proposed data model unifies the different concepts coming from existing data stores. For that, we have done a comparison (see TABLE 1) between the different concepts used in three categories of data stores which are commonly used in cloud environments: MySQL a relational data store, Riak a key/value data store and MongoDB a document data store. For instance, a table in MySQL is equivalent to a collection in MongoDB and to a database in Riak. We propose to refer to this concept by Entity Set in our unifying data model. In addition, a row in MySQL, a document in MongoDB 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 which includes all resources.

<table>
<thead>
<tr>
<th>Relational concepts</th>
<th>MongoDB concepts</th>
<th>Riak concepts</th>
<th>Unifying data model concepts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Database</td>
<td>Database</td>
<td>Environment</td>
<td>Database</td>
</tr>
<tr>
<td>Table</td>
<td>Collection</td>
<td>Database</td>
<td>Entity Set</td>
</tr>
<tr>
<td>Row</td>
<td>Document</td>
<td>Key/value</td>
<td>Entity</td>
</tr>
<tr>
<td>Column</td>
<td>Field</td>
<td>Value</td>
<td>Attribute</td>
</tr>
</tbody>
</table>

TABLE 1: Comparison chart of concepts used in different data stores

Based on TABLE 1, we define the unifying data model (see Fig. 2) based on five concepts:

- The Environment concept: It represents an attribute in a data store. In Fig. 3(a), Fig. 3(b), Fig. 3(c), we give an example of an EntitySet of type relational database.
- The Entity concept: An Entity is a set of one or multiple Attributes. In Fig. 3(c), we show Entities of type Rank identified by a Conference.
- The Attribute concept: A concept a of type Attribute is characterized by a set of two 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 \( A \) the set of the Attribute.
- The Entity concept: We define a concept \( e \) of type Entity as follow: \( e = \{ a_1, \ldots, a_n \} \) where \( a_i \in \{ 1..n \} \in A \). This set 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 \( E \) the set of the Entity.
- The Entity set concept: We define a concept \( E \) of type EntitySet as follow: \( E = \{ e_1, \ldots, e_n \} \) where \( e_i \in \{ 1..n \} \in E \). We denote by \( E \) the set of the EntitySet.

4.2 Formal description of the unifying data model

In this section, we propose a formal definition of the unifying data model concepts:

- Attribute concept: A concept \( a \) of type Attribute is characterized by a set of two 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 \( A \) the set of the Attribute.
- Entity concept: We define a concept \( e \) of type Entity as follow: \( e = \{ a_1, \ldots, a_n \} \) where \( a_i \in \{ 1..n \} \in A \). This set 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 \( E \) the set of the Entity.
- Entity set concept: We define a concept \( E \) of type EntitySet as follow: \( E = \{ e_1, \ldots, e_n \} \) where \( e_i \in \{ 1..n \} \in E \). We denote by \( E \) the set of the EntitySet.

![Fig. 3: Examples of EntitySets](image-url)
• **Database** concept: We define a concept \( d \) of type Database as follow: \( d = \{ e_1, ..., e_n \} \) where \( e_i \in \{ 1..n \} \in ES \). We note by \( D \) the set of Database.

• **Environment** concept: We define a concept \( env \) of type Environment as follow: \( env = \{ d_1, ..., d_n \} \) where \( d_i \in \{ 1..n \} \in D \). We denote by \( ENV \) the set Environment.

Let us now give an example of a formal presentation of the document entity set \( dblp \). Suppose that this entity set is in a database named \( dblpDB \) in the environment \( OurEnvironment \). Below, we present the example:

- \( OurEnvironment \in ENV, OurEnvironment = \{ dblpDB \} \) where \( dblpDB \in D \).
- \( dblpDB \in D, dblpDB = \{ dblp \} \) where \( dblp \in ES \).
- \( e \in E, e = \{ author, year, title, Conference \} \) where \( author, year, title, Conference \in A \).
- \( author, year, title, Conference \in A, author = \{ setofString, OurEnvironment.dblpDB.dblp.author \}; year = \{ String, OurEnvironment.dblpDB.dblp.year \}; title = \{ String, OurEnvironment.dblpDB.dblp.title \}; Conference = \{ String, OurEnvironment.dblpDB.dblp.Conference \} \).

### 4.3 Global schema

Even if NoSQL data stores do not support schema, some kind of global schema is needed to allow programmers to express their queries. Our global schema describes all the entity sets accessible by the application, but does not make any kind of schema integration like in classical mediation systems. To do so, we use our unifying data model. The possible joins between entity sets are expressed using correspondence rules.

In Fig. 4, we illustrate the corresponding data model to the \( dblp, person, \) and \( conferenceRanking \) entity sets. In this schema, we precise the database name and the environment where we can find these entity sets.

![Fig. 4: The corresponding data model to the \( dblp, person, \) and \( conferenceRanking \) entity sets](image)

The correspondence rules help applications’ programmers to correctly express the logic expression in a join query and to remove semantic ambiguities between attributes. In the definition below, we introduce this kind of rules.

### Correspondence rules

A correspondence rule is defined as follow: Let us consider \( A_1 < t_1, q_1 >, A_2 < t_2, q_2 > \in \mathcal{A} \). \( A_1 \) is equivalent to \( A_2 \) if and only if \( t_1 \) and \( t_2 \) are compatible and \( A_1 \), and \( A_2 \) denotes 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 \( CR \) the set of the correspondence rules.

In Fig. 4 we showcase two correspondence rules using dashed lines. The first is between the Attributes \( personName \) and \( author \) in the \( person \) and \( dblp \) entity sets respectively. Whereas the second rule is between the two Attributes \( conference \) in both \( conferenceRanking \) and \( dblp \) entity sets.

### 4.4 Query algebra

The query language that we propose is based on a query algebra defined on entity sets. This query algebra is simple and is composed by the classical unary operations (e.g. selection and projection) and the binary operations (e.g. Cartesian product, join, union, etc.). For manipulating complex objects like those defined in our unifying model, more complex algebra can be used, notably N1NF algebra which will be considered for a future work. For lack of space in this paper, we define only the projection, the selection, the Cartesian product and the join.

#### 4.4.1 Projection operation

A projection is a unary and mathematical operation that takes as input an entity set and returns as output an entity set. The resulted entity set 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. This problem can be encountered in the case of an entity set of type NoSQL because this kind of data stores is schema-less. Formally, we define the projection operation as follow:

**Projection operation** \( \pi_{\text{attributes}} : ES \rightarrow ES \)

Let \( E = \{ e_1, ..., e_n \} \in ES \) and \( R = \{ r_1, ..., r_l \} \in ES \).

The projection from \( E \) is defined as follow:

\[
R = \pi_{\text{attributes}}(E) = \{ r | r \text{ formed only by attributes of the projection} \}
\]

#### 4.4.2 Selection operation

A selection is a unary and mathematical operation that takes as input an entity set and returns as output an entity set. The resulted entity set contains the set of entities satisfying a given prepositional logic expression 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 follow:

**Selection operation** \( \sigma_{\text{predicates}} : ES \rightarrow ES \)

Let \( E = \{e_1, ..., e_n\} \subset ES \) and \( R = \{r_1, ..., r_l\} \subset E \).
The selection from \( E \) is defined as follow:

\[
R = \sigma_{\text{predicates}}(E) = \{r| r \in E \land \text{predicates}(r)\}
\]

4.4.3 Cartesian product

A Cartesian product is a binary and mathematical operation that
takes as input two entity sets and returns as output
an entity set. The resulted entity set contains the set of all
unions combining two entities belonging to the each
input entity set. Formally, we define the Cartesian product
function as follow:

**Cartesian product of two entity sets** \( \times : ES \times ES \rightarrow ES \)

Let \( E = \{e_1, ..., e_n\} \subset ES, F = \{f_1, ..., f_m\} \subset ES \) and
\( R = \{r_1, ..., r_l\} \subset ES \).
The Cartesian product of \( E \) and \( F \) is defined as follow:

\[
R = E \times F = \{(e_1 \cup f_1), (e_1 \cup f_2), ..., (e_1 \cup f_m), ...
\}

4.4.4 Join operation

A join operation can be seen as a conditional Cartesian product
where en element of the join result has to satisfy
the join condition. The entity set result of a join operation
contains the set of all combinations of entities in the
input entity sets that are equal on a given common attributes
denoted by a logical expression \( \text{cond} \).

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

**Join operation** \( \times_{\text{cond}} : ES \times ES \rightarrow ES \)

Let \( E = \{e_1, ..., e_n\} \subset ES, F = \{f_1, ..., f_m\} \subset ES \) and
\( R = \{r_1, ..., r_l\} \subset ES \), the join operation of \( E \) and \( F \) is
defined as follow:

\[
R = E \times_{\text{cond}} F = \{r| r \in E \land F \land \text{cond}(r)\}
\]

Based on this definition, we give an example allowing to join the two entity sets \( dblp \) and \( person \)
(see Fig. 3[a] and Fig. 3[b]) in order to get for each
title the affiliation of its authors. The logic expression
in our case is \( \text{person.personName} = \text{dblp.author} \).
So, the join operation may be written as follow:

result = \( \text{person} \times_{\text{person.personName} = \text{dblp.author}} \text{dblp} \).

As an example of entity in the entity set result
we give: \{"Leslie Valiant","1984","A theory of
the learnable","1","Leslie Valiant","United
Kingdom","Harvard University"\}.

5 OPENPAAS DATABASE API: ODBAPI

In this section, we introduce ODBAPI which is a REST API
enabling the execution of CRUD operations on different
types of data stores supporting our unifying data model.
This API is designed to provide an abstraction layer and
seamless interaction with data stores deployed in a cloud
environment. Developers can execute queries in a uniform
way regardless of the type of the data store (relational or
NoSQL). An overview of the API is given in Fig. 5. The
figure is divided in four parts that we introduce in the
following starting from the right to the left side. First, we
have the deployed data stores (e.g. relational DBMS, Couch\DB, etc.) that a developer may interact with. Second, we
find the proprietary API and driver of each data store
supported by ODBAPI. For instance, we use in our API
implementation the JDBC API and MySQL driver to interact
with a relational DBMS. The third part of Fig. 5 refers to the
ODBAPI implementation. In fact, this part is shared between
all the integrated data stores. In addition, it contains specific
implementation of each data store. To integrate a new data
store and define interactions with it, one has simply to add
the specific implementation of that data store. Finally, Fig. 5
shows the different operations that ODBAPI offers.

![Fig. 5: An overview of ODBAPI](image)

In our specification, we consider two kinds of operations
which inputs and outputs are JSON-based data. The first
operations family is dedicated to get meta-information about
the resources using the GET REST method. Indeed, ODBAPI
offers four operations:

- Get information about the user’s access right: This
  operation is provided by \( \text{getAccessRight} \) and allows
  a user to discover his access rights concerning the
  deployed data stores in a cloud environment. To do
  so, the user must append the keyword \( \text{accessright} \)
to his request.

- Get information about an Environment: This
  operation is ensured by \( \text{getEnvMetaData} \) and lists the
  information about an Environment. To execute this
  kind of operation, a user must provide the keyword
  \( \text{metadata} \) in his request. This keyword should be also
  present in the following two operations.

- Get information about a Database: A user can retrieve the
  information about a Database by executing the
  operation \( \text{getDbMetaData} \) and providing the name
  \( \text{dbName} \) of the target Database. This operation outputs
  information about a Database (e.g. duplication,
  replication, etc) and the \( \text{entity sets} \) that it contains.

- Get information about an EntitySet: This operation is provided by \( \text{getESMetaData} \) and enables to discover
the information about an 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 operations executed on resources of type either EntitySet or 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 entity sets 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 getEntityByld, 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 allIE. 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. Indeed, 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.

6 Query Evaluation and Optimization

In this section, we present in more details our approach to evaluate and optimize queries execution based on the VDS. In Section 6.1, we introduce the principles of this process. In Section 6.2, we apply these principles to answer a join query over three heterogeneous data stores.
In the following, we use two scenarios involving different data stores and annotations for our join query above.

- Scenario 1 - Join between two entity sets in the same data store: Let us consider that the two entity sets dblp and conference ranking are in the same data store of type MySQL.
- Scenario 2 - 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 key-value data store Riak respectively. The size of the relational data store is very large.

After the algebraic query optimization, the VDS annotates the algebraic tree with information got from a catalog storing metadata about the target DBMS and data (see Fig. 8 for the scenario 1 and Fig. 9 for scenario 2). These information include the type and the name of the data stores, their location, their capabilities (support of join queries for example), the size of the entity sets and additional statistics (if possible). Statistics are either exported by data stores or estimated using probing queries. 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. 8 we depict that the join between the entity sets ConferenceRanking and dblp can be done on a MySQL RDBMS located on host1 and has an estimated output size of 4 Mbytes. Whereas the algebraic tree of scenario 2 in Fig. 9 we can see that the same join can not be done on the same
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 routes automatically 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 this plan (the VDS or the specific node storing a very large entity set). In addition to the algebraic operations, a query execution plan can use two new other operations, ship and convert that enable respectively 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. Choosing the optimal plan is a classical of optimization based on a cost model.

In Fig. 10, 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 recombing 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).

### 7 Automatic Deployment of ODBAPI-based Application in a Cloud Provider

In this section, we describe how the discovery and deployment components operate as well as the structure of the different manifests used and produced by these two components.

#### 7.1 Abstract application manifest

Using a dedicated manifest, developers express what kind of data stores and execution environment their application requires. Fig. 11 depicts the structure of the abstract application manifest (AAM for short). Basically, the AAM contains two categories of information: the first category is expressed under the application element and the second one is defined under the environment element.

The application element (part of the AAM) contains information on the to-be-deployed application itself. Developers may specify several versions of the same application. And for each version, they need to precise on one hand information related to the deployable artefacts and on the other hand information related to the to-be-run instances. Thus each version element contains a set of deployable and instance elements. For each deployable artefact, developers specify the artefact type, the URL where such artefact can be used.
found and multi-tenacy level attribute indicating the application tenancy degree. An instance element specifies information on a specific instance to be created.

The environment element in the AAM 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. For each node, developers may specify the required type (for instance a document or key-value data store), the name of a specific component (for instance Mongo DB or CouchDB), any specific version, and the required size.

7.2 Discovery and deployment steps

The environment element (part of the AAM) is matched with cloud environment capabilities in order to find the most appropriate environment to deploy the application on it. The output of the matching process is an offer manifest that describes the selected cloud environment with the set of the candidate nodes.

Once an appropriate cloud environment is found, we proceed in three steps for deploying a multiple data stores application. First, we deploy the REST services giving access to the data stores. This first deployment step returns back the end-points of these services. Second, we deploy the VDS while specifying for it the services’ endpoints. We deploy one VDS per application. In the last step, we deploy the application itself by passing to it the VDS endpoint returned back in the previous step. Doing so, the application holds the endpoint of the VDS it needs to interact with and the VDS has itself the endpoints of the wrapper REST services.

For each of these three steps, we can use any deployment API (e.g. COAPS API, roboconf API, etc.). In our work, we are building on the COAPS API that is proposed in our team SIMBAD (see Section 7 for the definition of COAPS). We provide today a version of ODBAPI including four data stores: MySQL, Riak, MongoDB and CouchDB. The current version is developed in Java and is provided as a runnable RESTful web application (e.g. jar file). Now we run a tool ensuring the discovery of cloud providers and the automatic deployment of an ODBAPI-based application. Indeed, the application programmer describes his requirements in the abstract application manifest and he uploads it through the interface that we illustrate in Fig. 12. Once this manifest is uploaded, we implement a simple matching algorithm. This algorithm takes as input the abstract application manifest to elect the appropriate cloud provider that supports the ODBAPI client requirements and returns to the user the deployment manifest. In Fig. 12, we represent a part of this manifest. The application will be deployed in the cloud provider Cloud Foundry and it will use one service of type container and two services of type database: MongoDB and MySQL. We provide a video demonstration at http://www-inf.int-evry.fr/~sellam_r/Tools/ODBAPI/index.html. To deploy the application, we are based on COAPS API that is proposed in our team SIMBAD (see Section 7 for the definition of COAPS). This API allows to deploy applications using just one data store. To cover this gap, we propose to extend it in order to support multiple data stores applications deployment.

8 IMPLEMENTATION AND VALIDATION

In this section, we present the current state of the implementation of the different components of our approach that we have already presented in the previous sections. First of all, we present a tool allowing the discovery of data stores based on the abstract application manifest (see Section 8.1). Second, we present a state of progress about the implementation of ODBAPI and the data stores that we take into account (see Section 8.2). Added to that, we present some ODBAPI-based applications that illustrate the utility of our API. Finally, we evaluate the overhead related to ODBAPI compared to proprietary APIs (see Section 8.3).

8.1 Selecting data stores and deploying ODBAPI clients

We programmed a tool ensuring the discovery of cloud providers and the automatic deployment of an ODBAPI-based application. Indeed, the application programmer describes his requirements in the abstract application manifest and he uploads it through the interface that we illustrate in Fig. 12. Once this manifest is uploaded, we implement a simple matching algorithm. This algorithm takes as input the abstract application manifest to elect the appropriate cloud provider that supports the ODBAPI client requirements and returns to the user the deployment manifest. In Fig. 12, we represent a part of this manifest. The application will be deployed in the cloud provider Cloud Foundry and it will use one service of type container and two services of type database: MongoDB and MySQL. We provide a video demonstration at http://www-inf.int-evry.fr/~sellam_r/Tools/ODBAPI/index.html. To deploy the application, we are based on COAPS API that is proposed in our team SIMBAD (see Section 7 for the definition of COAPS). This API allows to deploy applications using just one data store. To cover this gap, we propose to extend it in order to support multiple data stores applications deployment.

![Select an abstract application model](http://www-inf.int-evry.fr/~sellam_r/Tools/ODBAPI/index.html)

Fig. 12: Screenshot of the interface allowing to select the abstract application in order to get the deployment manifest.

8.2 Current state of ODBAPI

We provide today a version of ODBAPI including four data stores: MySQL, Riak, MongoDB and CouchDB. The current version is developed in Java and is provided as a runnable RESTful web application (e.g. jar file). Now we are working diligently on testing ODBAPI using various use cases in the OpenPaaS project so that we identify possible discrepancies and make this version more stable to 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 allows accessing the ODBAPI specification, (2) the second allows downloading a jar file of ODBAPI, (3) and the third provides a user guide. We point out that we used the Restlet framework in order to implement ODBAPI.

3Roboconf home page: [http://roboconf.net/fr/index.html](http://roboconf.net/fr/index.html)

In order to show the feasibility and the utility of our API, we provide a client that we called ODBAPIClient. This latter allows a developer to use ODBAPI operations through JAVA methods. Hence, it is easy for him to program his application. We developed also an other ODBAPI-based client intended to handle the administration of relational and NoSQL data stores in a cloud provider. This client is a PHPMyadmin-like. In Fig. 13, 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 show also an overview of the entities of the city entity set. Finally, we implemented in the OpenPaaS project an ODBAPI-based module enabling the management of to-do tasks in a project. In this module, we interact with a document and a relational data stores by executing multi-sources queries.

Fig. 13: Screenshot of all databases overview

### 8.3 Evaluation of the overhead of ODBAPI

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 (see Section 5). In this section, we propose to evaluate the overhead related to ODBAPI. 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 $\Delta$ is obtained by calculating the ratio between the difference between 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:

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

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 2, we showcase the response time of these operations and the overhead. The average of this overhead is about 6.71%.

<table>
<thead>
<tr>
<th>Entities number</th>
<th>10</th>
<th>50</th>
<th>100</th>
<th>500</th>
</tr>
</thead>
<tbody>
<tr>
<td>ODBAPI (ms)</td>
<td>715</td>
<td>2399</td>
<td>4219</td>
<td>19115</td>
</tr>
<tr>
<td>JDBC (ms)</td>
<td>700</td>
<td>2250</td>
<td>3883</td>
<td>17463</td>
</tr>
<tr>
<td>$\Delta$ (%)</td>
<td>2.14</td>
<td>6.62</td>
<td>8.65</td>
<td>9.46</td>
</tr>
</tbody>
</table>

Table 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 3, we represent the obtained results and the overhead. The average of this overhead is about 4.35%.

<table>
<thead>
<tr>
<th>Entities number</th>
<th>10</th>
<th>50</th>
<th>100</th>
<th>500</th>
</tr>
</thead>
<tbody>
<tr>
<td>ODBAPI (ms)</td>
<td>677</td>
<td>2309</td>
<td>4164</td>
<td>18238</td>
</tr>
<tr>
<td>JDBC (ms)</td>
<td>662</td>
<td>2205</td>
<td>3878</td>
<td>17696</td>
</tr>
<tr>
<td>$\Delta$ (%)</td>
<td>2.26</td>
<td>4.71</td>
<td>7.37</td>
<td>3.06</td>
</tr>
</tbody>
</table>

Table 3: Response time of the operation delete entities with ODBAPI and JDBC

We calculate also the overhead of retrieving all entities in one query with ODBAPI and JDBC. For this, we illustrate in Table 4 the evolution of the 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.

<table>
<thead>
<tr>
<th>Entities number</th>
<th>100</th>
<th>500</th>
<th>1000</th>
<th>2000</th>
<th>3000</th>
<th>4000</th>
</tr>
</thead>
<tbody>
<tr>
<td>ODBAPI (ms)</td>
<td>215</td>
<td>259</td>
<td>305</td>
<td>349</td>
<td>416</td>
<td>543</td>
</tr>
<tr>
<td>JDBC (ms)</td>
<td>213</td>
<td>263</td>
<td>288</td>
<td>347</td>
<td>408</td>
<td>422</td>
</tr>
<tr>
<td>$\Delta$ (%)</td>
<td>0.93</td>
<td>2.28</td>
<td>5.90</td>
<td>0.57</td>
<td>1.96</td>
<td>28.67</td>
</tr>
</tbody>
</table>

Table 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. 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.

### 9 Related work

In our previous work [8], we focused on existing solutions of the state-of-the-art supporting multiple data stores based applications in the cloud environment. More precisely, (i) we described different scenarios related to the way applications use data stores, (ii) we defined the data requirements of applications in cloud environment, and (iii) we analyzed and classified existing works on cloud data management, focusing on multiple data stores requirements. As a result, we pointed out six requirements of using multiple data stores in a cloud environment. Based on these requirements,
we propose our end-to-end solution to support multiple data stores applications in cloud environments. To the best of our knowledge, there are no another global solution addressing all the problems we focus on. In this section, we discuss the related works to each major contribution in this paper.

We start by presenting works aiming at proposing unique API to enable CRUD queries execution across relational and NoSQL data stores. There are some proposals like the Spring Data Framework [9], SOS [10], and ONDM [11] that allow an application to query data from different NoSQL data stores. The Spring Data Framework provides some generic abstractions to handle different types of NoSQL DBMSs and relational DBMSs. Nevertheless, the addition of a new data store is not so easy and the solution is strongly linked to the Java programming model. SOS provides CRUD operations at the level of individual data store. These operations are provided via GET, PUT, and DELETE methods. 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.

Whereas, ONDM provides ORM like API based on popular Java Persistence API (JPA) to application developers to interact with NoSQL data stores. However, ONDM does not take into account relational data stores. There is also a REST API called RSenter [12], [13] that enables the data mining and search of unstructured data stored in NoSQL data stores. Although this work is interesting, it does not enable the CRUD operations and complex queries execution. In addition, it does not take into account the relational data store. In an other previous work [4], we proposed a generic resources model defining the different concept used in each type of data store. These resources are managed by ODBAPI a streamlined and unified REST API enabling to execute CRUD operations on different NoSQL and relational databases. ODBAPI decouples cloud applications from data stores alleviating therefore their migration. Moreover it relieves developers task by removing the burden of managing different APIs.

In addition to the CRUD operations execution across relational and NoSQL data stores, we find some substantial works [14], [15], [16], [17], [18] proposing mediation based solutions for join queries execution. In the context of relational integration, Lenzerini [14] formalizes the principles of relational query rewriting on multiple sources. In addition, Manolescu [15] proposed an XML based mediation by extending these proposals to handle semi-structured data. These works suppose that an integrated schema (or global schema) exists which is not possible in our case since we are dealing with schemaless data stores. Hence, they can not support NoSQL data stores. Our proposal for expressing and executing join queries is closed to these works and we can reuse their optimization techniques. Curé et al. [15] represent the global schema by a relational data model to query NoSQL and relational DBMSs. They define also a mapping language to map attributes of the data sources to the global schema and a bridge query language to rewrite queries. In a second step, Curé et al. [17] extend their solution by using an Ontology Based Data Access (OBDA). Additionally, they replace BQL with SPARQL. Although their proposal is promising, there are some lacking functionalities. Indeed, they do not propose a cost model at the global level and they do not support the complex query execution. Roijackers et al. [18] propose a hybrid mediation approach in which they integrate relational and document data stores. More precisely, documents are logically embedded inside relational data stores to form hybrid data stores queried via the proposed extended-SQL language called NoSQL query pattern (NQP).

The last part of the related work is dedicated to the data stores capabilities discovery and the matching between these capabilities and the application requirements. Indeed, we find several works [19], [20], [21], [22], [23], [24] enabling an application to negotiate its Data Management Contract (DMC), often referred to as data agreement or data license, with various clouds and to bind to the specific DBMSs according to its DMC. Truong et al. [19], [20], [21] 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 [24], 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 DEscription MOdel for DaaS (DEMODS) [21]. 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 application. In [22], 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. In [23], Ruiz-Alvarez et al. propose an automated approach to selecting the PaaS storage service according 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.

## 10 Conclusions and future work

In this paper, we proposed a generic approach to facilitate the developer task and enable the development of applications using multiple data stores while remaining agnostic to these latter. We introduced three solutions:

- **ODBAPI for CRUD operations:** We defined a generic resources model to represent the different elements of heterogeneous data stores in a Cloud environment.
Based on this, we define 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 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: We proposed virtual data stores to execute complex queries (including joins) across NoSQL and relational data stores. For this purpose, we defined a unifying data model able to describe the heterogeneous data models of data stores. It is used by the user to express his complex query and by the virtual data store to process it. Once a virtual data store receives a complex query, it constructs an optimal query execution plan, composed by sub-queries at the level of target data sources, conversion and shipping operations and a final query recombining partial results.

- Manifest for data stores discovery and automatic application deployment: Once the developer has completed the development of his application, we provided him the possibility to express his application requirements in terms of data stores in the abstract application manifest. Then, he sends it to the matching module that interacts with the cloud providers discovery module to elect the appropriate cloud provider to the application requirements. Indeed, the cloud providers discovery module discovers the capabilities of data stores of each cloud provider and returns these capabilities in the offer manifest. Based on that, the matching module implements the matching algorithm in order to elect the adequate cloud provider to the application requirements and generates the deployment manifest of the application. Once it is done, we deploy the application using the COAPS API that takes as input the deployment manifest.

Currently, we are working on applying ODBAPI and the virtual data store query optimization and execution approach to other qualitatively and quantitatively various scenarios in the OpenPaaS project. This allows us to identify possible discrepancies and make our work more reliable for public use. In addition, we aim to study an implementation for Hive allowing access to Hadoop data stores. Our second perspective consists in providing another matching algorithm supporting approximate matching. Hence we enable more flexibility in data stores discovery and applications deployment. Our third perspective is an extension to virtual data stores, allowing to support a larger class of complex queries across NoSQL and relational data stores (union, intersection, aggregates, group by like operations) and introducing more elaborate query processing optimization techniques, including asynchronous evaluation.

ACKNOWLEDGMENT
This work has been partly funded by the French FSN OpenPaaS grant( https://www.openpaas.org/display/openpaas/Open+PAAS+Overview).

REFERENCES


Rami Sellami He obtained his MS degree in 2012 and his engineer’s degree in 2011 from the Faculty of Sciences of Tunis, Tunis, Tunisia. He is currently a PhD student under the supervision of Professor Bruno Defude at Telecom SudParis, Evry, France. His research topic is about big data, polyglot persistence and cloud computing management (Platform-as-a-Service). More information can be found at http://www-inf.int-evry.fr/ sellam_r

Sami Bhiri Sami Bhiri is an associate professor at Telecom SudParis. He was the leader of the SOA unit at DERI, and adjunct lecturer at the National University of Ireland, Galway. Before joining DERI, he was a research and teaching assistant in the University of Nancy 1 and in the ECOO team of the LORIA-INRIA research laboratory from where he holds an M.S. (2001) and a Ph.D. (2005). His research interests are on semantically-enabled Service Oriented Architecture and semantic Business Process Management.

Bruno Defude received the Ph.D. degree in 1986 and Habilitation degree in 2005 in computer science, respectively, from Grenoble INP and Paris VI University, Paris, France. He is currently a Professor in the Computer Science Department at TELECOM SudParis, France. His research interests are distributed data management, cloud data management, and semantics for B2B integration.

Prof. Defude is a member of IEEE and ACM SIGMOD. He was the Chair of the IEEE WETICE Conference in 2011. He has been participating in several national and European research projects. He has published more than 60 research papers in international conferences and journals and has served as program committee member in many conferences and workshops.