Transparent Interoperability Middleware between Data and Service Cloud Layers

Elivaldo Lozer Fracalossi Ribeiro, Marcelo Aires Vieira, Daniela Barreiro Claro and Nathale Silva
FORMAS (Research Group on Formalisms and Semantic Applications) - LASID - DCC - PGCOMP,
Federal University of Bahia, s/n Adhemar de Barros Ave., Ondina, 40.170-110, Salvador, Bahia, Brazil

Keywords: Cloud Computing, Interoperability, Middleware, DaaS, DbaaS.

Abstract: Over the years, many organizations have been using cloud computing services to persist, consume and provide data. Models such as Software as a Service (SaaS), Data as a Service (DaaS), and Database as a Service (DBaaS) are consumed on demand to serve a specific purpose. In summary, SaaS is a delivery model for applications, while DaaS and DBaaS are models to provide data and database management systems on demand, respectively. SaaS applications require additional efforts to access those data due to their heterogeneity: Non-structured (e.g. text), semi-structured (e.g. XML, JSON), and structured format (e.g. Relational Database). Consequently, the lack of standardization from DaaS and DBaaS generates a lack of interoperability among cloud layers. In this paper, we propose a middleware MIDAS (Middleware for DaaS and SaaS) to provide transparent interoperability between Services (SaaS) and Data layers (DaaS and DBaaS). Our current version of MIDAS concerns two important issues: (i) a formal description of our middleware and (ii) a joining data from different DaaS and DBaaS. To evaluate our middleware, we provide a set of experiments to handle functional, execution time, overhead, and interoperability issues. Our results demonstrate the effectiveness of our approach to addressing interoperability concerns in cloud computing environments.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

1 INTRODUCTION

The volume of digital data grows exponentially, with an estimated total of 40 trillion gigabytes in 2020 (Gantz and Reinsel, 2012). Because these data need to be stored and available both to consumers and to organizations, data management have been facing some challenges to handle the variety and amount data. The cloud computing paradigm has emerged to fill some of these requirements, once it provides services with high availability and data distribution (Mell et al., 2011). By 2020, nearly 40% of the available data will be managed and stored by a cloud computing provider (Gantz and Reinsel, 2012).

Authors in (Armbrust et al., 2010) define cloud computing as a model that enables a ubiquitous and on-demand network of applications, platforms, and hardware, both provided as services. These services are organized into levels and consumed on demand by users in a scheme of pay-per-use. Software as a Service (SaaS), Data as a Service (DaaS), and Database as a Service (DBaaS) are instances of service types organized in cloud levels. SaaS is cloud applications made available to end users via the Internet. DaaS provides data on demand through application programming interfaces (APIs). DBaaS provides database management systems (DBMS) with mechanisms for organizations to store, access and manipulate their databases (Hacigumus et al., 2002). Although confusing, DaaS and DBaaS are different concepts.

The emergence of Internet of Things (IoT), social networks and the use of web-enabled devices such as smartphones, laptops, and notebooks generate a huge volume and variety of data (Armbrust et al., 2010). Data are stored in non-structured, semi-structured or structured databases. Governments, Institutions, and Companies most use DaaS as a way to make their data (expenses, budgets, economic or census data) available to public or private users across the Internet (Barouti et al., 2013).

The access to DaaS and DBaaS in different cloud providers by SaaS applications needs, in most of the cases, substantial efforts. This kind of situation handles a lock-in problem due to the lack of interoperability among cloud levels (Loutas et al., 2011; Silva et al., 2013). For instance, if demographic researchers need to make studies about census data provided by

Lozer Fracalossi Ribeiro, E., Aires Vieira, M., Barreiro Claro, D. and Silva, N.
Transparent Interoperability Middleware between Data and Service Cloud Layers.
DOI: 10.5220/0006704101480157
In *Proceedings of the 8th International Conference on Cloud Computing and Services Science (CLOSER 2018)*, pages 148-157
ISBN: 978-989-758-295-0
Copyright © 2019 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

governments in different DaaS (and/or DBaaS), they will face the difficult to process these data due to the lack of standards and consequently no interoperability between SaaS and DaaS (and/or DBaaS). To accomplish this interoperability issue, we propose our middleware called MIDAS (Middleware for DaaS and SaaS).

Our current version of MIDAS (MIDAS 1.8) is responsible for mediating the communication between different SaaS, DaaS, and DBaaS. MIDAS makes possible that SaaS applications retrieve data seamlessly on various cloud data sources since our MIDAS mediates all communication between SaaS and DaaS/DBaaS. Our version guarantees access to DaaS regardless of modifications made to its API.

We propose in this paper a new enhanced version of our middleware MIDAS to provide a transparent interoperability among different cloud layers. The current version of MIDAS (MIDAS 1.8) handles two important issues: (i) a formal description of our approach and (ii) a join clause to manipulate different data (DaaS and DBaaS) into a single query. Some minor improvements were made in order to adjust our MIDAS, such as (i) recognization of different data query structures sent by SaaS, such as SQL and NoSQL queries; (ii) manipulate different DaaS and DBaaS from statements such as join (SQL) and lockup (MongoDB); (iii) manipulate different data models returned by DaaS and DBaaS, such as JSON, XML, CSV and tables; and (iv) return the result into the required format by SaaS, such as JSON, XML, and CSV.

We performed some experiments to evaluate our novel approach, considering four important issues: Functional, execution time, overhead, and interoperability. Our results demonstrated that our middleware is effective, thus providing the desired results.

The remainder of this paper is organized as follows: Section 2 presents the most relevant related works; Section 3 describes our current version of MIDAS; Section 4 formalizes our middleware; Section 5 provides a set of experiments to validate our approach; Section 6 presents some results; and Section 7 concludes with some envisioning work.

2 RELATED WORKS

Some close works were proposed to solve the lack of interoperability. In medical field, authors in (Park and Moon, 2015) propose a solution for heterogeneous DBaaS that share medical data between different institutions. However, this approach handles data that follows the Health Level Seven (HL7) standards, thus minimizing efforts regarding heterogeneity.

The authors in (Igamberdiev et al., 2016) present a framework to solve problems in Big Data systems in the area of oil and gas. The goal is to automate the transfer of information between projects, identifying similarities and differences. Their framework handles only one data source per query, not allowing to merge data from more than one source.

Considering a non-domain-specific interoperability solution, there are two related work: (Sellami et al., 2014) and (Xu et al., 2016). These proposals do not deal with different types of NoSQL, nor envision to handle NewSQL approaches. Besides, they manipulate data sources without joining, and they do not work with data provided by DaaS. It is noteworthy that manipulating both DaaS and DBaaS is one of the main advantages of our proposal.

The cloud Interoperability Broker (CIB) is a solution to interoperate different SaaS (Ali et al., 2016). This work was evaluated in a dataset through an actual application, but unlike our proposal, they do not consider the interoperability between SaaS and DaaS.

Despite the fact that our prior approach (MIDAS 1.6) (Vieira et al., 2017), it had some limitations: (i) Each DaaS must be manually inserted and updated; (ii) DBaaS is not provided; and (iii) data were returned to SaaS only in JSON format.

Thus, to the best of our knowledge, this is the first middleware that interoperates SaaS with DaaS and/or DbaaS in cloud environments.

3 THE CURRENT MIDAS

The current MIDAS architecture is depicted in Fig. 1. This novel approach is composed of six components. The *Query Decomposer* which receives a query from SaaS and maps the statement into an internal structure. *Query Builder* which receives the query decomposed and builds a query to DaaS and/or DBaaS. The *Data Mapping* component which identifies and obtains data from different DBaaS. *Dataset Information Storage* (DIS), that sets the information about DaaS APIs. A *Crawler* which maintains DIS up-to-date. Finally, the *Result Formatter*, which formats, associates, and selects data before returning to SaaS.

The following components were included or modified to meet the goals of our current version: *Data Mapping*, *Query Builder*, *Result Formatter*, and *Crawler*. The other components, *Query Decomposer* and DIS, both works similarly to our previous version (Vieira et al., 2017).

The *Data Mapping* generates a DaaS from a DBaaS based on a manually maintained data dicti-

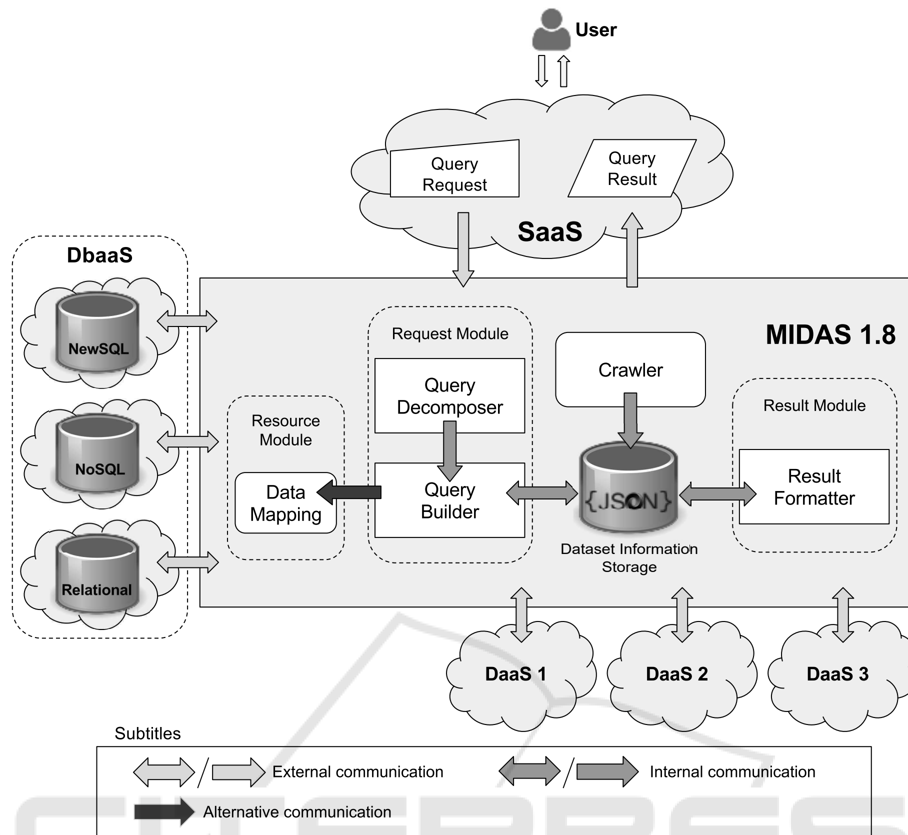

Figure 1: Our current MIDAS architecture.

onary. It identifies the DBaaS in which the data is stored and it obtains the requested data. DBaaS can be tables, columns, graphs, key-values or documents. The *Query Builder* accesses multiple DaaS in a single query if the query has a join statement (such as SQL join or MongoDB aggregation). In our current version, the *Result Formatter* receives either data from DaaS and DBaaS and performs the merge of such data, regardless the model. Finally, our *Crawler* maintains the DIS information up-to-date, considering that DaaS providers can change the parameters to conduct a query. Besides, the SaaS provider can now indicate the desired format to return its result.

Our *Crawler* has a challenging role in keeping DIS information up-to-date because of the DaaS. DaaS is not standardized thus it can change frequently. Moreover, they are usually distributed. Our *Crawler* searches for every DaaS API information from its web page, ensuring that the information does not cause any harm to the applications, in the case of updating. It was developed to run repeatedly toward search of different information from those persisted in DIS. When this information is found to be uneven, it is recorded in DIS.

Fig. 2 illustrates the MIDAS execution sequence for a SQL query with the join statement that accesses one DaaS and two DBaaS. In this example, SaaS sends a SQL query to MIDAS, which performs the decomposition (by *Query Decomposer*) and forwards to the *Query Builder*. *Query Builder* accesses the DIS and identifies that the data is in one DaaS (daas1) and two DBaaS (dbaas1 and dbaas2). Query Builder builds the request to DaaS and asks the *Data Mapping* to connect to both DBaaS to get the rest of the data. Each provider executes the request and returns the result to the *Result Formatter* (daas1, dbaas1, and dbaas2 returned in CSV, table, and document formats, respectively). The *Result Formatter* receives the data, performs the join, formats the requested return (JSON), and forwards to the SaaS.

4 FORMAL MODEL OF MIDAS

The formal model of MIDAS aims to explain the communication among its modules. The formalization of MIDAS is based on canonical models (Schreiner et al., 2015) with trees and sets of keys and values.

Definition 1 (MIDAS internal structure)**.** *The structure used internally by MIDAS (MIDASql) is a tuple*

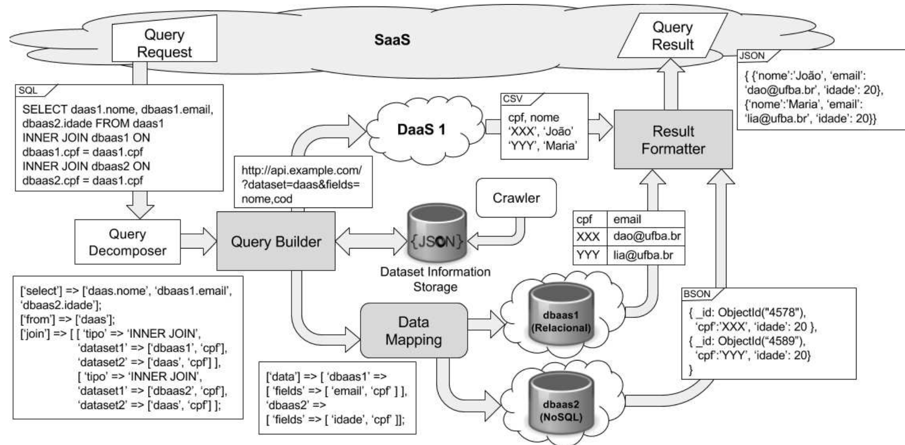

Figure 2: MIDAS execution sequence among one DaaS and two DBaaS through a join statement.

MIDASql $= (mDIS, mSaaS, mDaaS)$*, where: mDIS is the canonical model of DaaS presented in DIS; mSaaS is the canonical model that maps the query (sent by SaaS); and mDaaS is the canonical model that maps DaaS return(s).*

In the following sections, each canonical model is detailed.

4.1 Canonical Model mDIS

Definition 2 (mDIS). *The canonical model that stores DIS information (mDIS) is a tuple* $mDIS = (N_{root}, DAAS)$*, where:* N_{root} *is the name of the model; and DAAS is a set of DaaS models (daas set).*

Definition 3 (daas). *The canonical model for a specific DaaS (daas* $\in$ *DAAS) is a tuple* $daas = (N_{root_{daas}}, K)$*, where:* $N_{root_{daas}}$ *is the name of DaaS; and K is a predefined set of keys (k) for each DaaS, where K = {domain, search_path, query, sort, limit, dataset, records, fields, format}.*

Definition 4 (k). *A key* $k \in K$ *is an information about daas and it is defined as* $k = (N_{root_k}, i)$*, where:* N_{root_k} *is the name used to characterize a specific information about daas,* $k.N_{root_k} \in K$*; and i is the information about k, it can be empty, atomic, or multivalued.*

Considering a hypothetical DIS with two DaaS (NYC and v8), part of the canonical model *mDIS* can be seen in Fig. 3: The main node stores the beginning of subtrees, where each subtree stores the information about a particular DaaS. Each node of level *i* stores information on the *k* level, immediately above.

4.2 Canonical Model mSaaS

Definition 5 (mSaaS). *The canonical model mSaaS converts the query (submitted by SaaS) in a set with n queries (to sent to DaaS), where n indicates the number of relations in the query (e.g.:* $n = 2$ *indicates join with 2 tables),* $n \geq 1$*. The model mSaaS is a tuple* $mSaaS = (N_{root}, C_1)$*, where:* N_{root} *is the value of n; and* C_1 *is a set of first-level clauses (*c_1*) used in the mapping to identify queries and operations.*

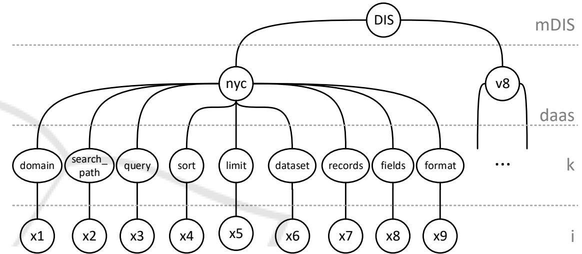

Figure 3: Example of *mDIS* for two DaaS.

Definition 6 (c_1). *A first-level clause* $c_1 \in C_1$ *stores specific information about a query OR about an operation, and it is a tuple* $c_1 = (N_{root_{c_1}}, C_2)$*, where:* $N_{root_{c_1}}$ *is the name that identifies the query OR the operation; and* C_2 *is a set of second-level clauses (*c_2*) used in the mapping to identify the query attributes and operations. Some important observations: (i)* $N_{root_{c_1}} \in \{q_1, q_2, \ldots, q_n, param\}$*, where* q_i *is an i-th relation and param is a node for storing data about join, order by and limit; and (ii) given n, there are* $n+1$ *clauses* c_1*.*

Definition 7 (c_2). *A second-level clause* $c_2 \in C_2$ *stores information about clauses of a query OR clauses of an operation, and it is a tuple* $c_2 = (N_{root_{c_2}}, V)$*, where:* $N_{root_{c_2}}$ *is the name that identifies the clause of a query OR the clause of an operation; e V is a set of values (v) for each* c_2*. Some important observations: (i) if* c_1 *represents a query* q_i*, then* $N_{root_{c_2}}$ *indicates j attributes (*$j \geq 0$*) of* q_i *stored, where* $N_{root_{c_2}} \in \{Projection, Selection, Dataset\}$*; (ii)*

if c_1 represents param, then $N_{root_{c_2}}$ indicates j attributes ($j \geq 0$) of n relations, where $N_{root_{c_2}} \in \{OrderBy, Limit, TypeJoin, CondJoin, Return\}$; and (iii) given n, there are $3n+5$ clauses c_2.

Definition 8 (v)**.** *A value $v \in V$ is an element representing information about c_2. Depending on the c_2, V may be empty, atomic, or multivalued. Thus, $V = \varnothing$ or $V = \{v_1, v_2, \ldots, v_w\}$, where: v_i is the i-th value v for c_2; and w is the number of values v in the set V of the key c_2, i.e., $v \in V$.*

For instance, the query of Table 1 (presented in SQL and NoSQL) generates the canonical model presented in Fig. 4; while the query in Table 2 generates the canonical model presented in Fig. 5.

Table 1: Example of a query in SQL and in NoSQL (MongoDB) without join/aggregation.

SQL	NoSQL (MongoDB)	
SELECT name, age	w7.find(	(from)
	{'age=10'},	(where)
FROM w7	{'name':1,	(select)
WHERE age=10	'age':1})	
ORDER BY name	.limit(10)	(limit)
LIMIT 10	.sort(	(order by)
	{'name':1});	

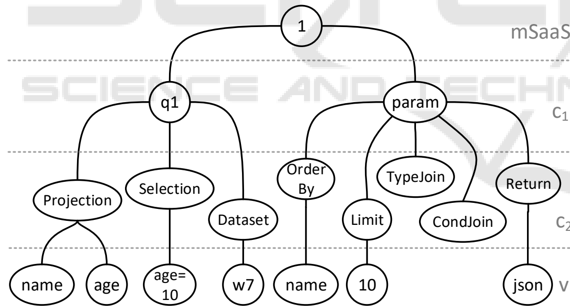

Figure 4: Example of *mSaaS* for query in Table 1.

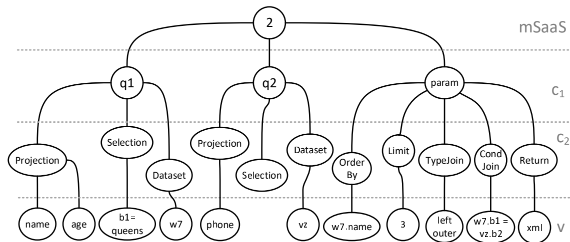

Figure 5: Example of *mSaaS* for query in Table 2.

After *mDIS* and *mSaaS* are generated, it is necessary to transform both canonical models into a set of URLs to submit to DaaS. The data from DaaS is received through a Uniform Resource Locator (URL), *MIDASql* provides a mechanism to convert *mDIS* and *mSaaS* into a set of URLs, the function *generateURLs()*. Our function has the following prototype: "URLs generateURLs(*mDIS*, *mSaaS*)". This means that, given a *mDIS* and a *mSaaS*, *generateURLs()* must returns a set of URLs, where: each URL is a concatenation sequence of *mDIS* and *mSaaS* elements; and the number of URLs is equal to the number of query relations (n, $n \geq 1$), i.e., each q_i (in *mSaaS*) generates URL$_i$. For this, we assume that: "$+$" is an operator that concatenates two strings (literals or variables); and $ch(p)$ is a function that returns the contents of the child(ren) of p node.

Thus, considering *DSname* = ch(q_i.dataset), URL$_i$ is generated according to Fig. 6.

```
URLi =       ch(DIS.DSname.domain)       +
             ch(DIS.DSname.search_path)  + '?' +
             ch(DIS.DSname.dataset)      + '=' + ch(qi.Dataset)
   + '&' +   ch(DIS.DSname.records)      + '=' + ch(qi.Projection)
   + '&' +   ch(DIS.DSname.query)        + '=' + ch(qi.Selection)
   + '&' +   ch(DIS.DSName.sort)         + '=' + ch(param.OrderBy)
   + '&' +   ch(DIS.DSname.limit)        + '=' + ch(param.Limit)
```

Figure 6: Concatenations that the generateURL() function uses to generate URL$_i$.

Considering the function *generateURLs()*, some observations are important: (i) when $ch(p)$ does not return any element, the corresponding line p in URL$_i$ must be disregarded; (ii) multivalued result of $ch(p)$ is separated by commas; (iii) the last two lines occur only for $n = 1$; and (iv) for $n \geq 2$, $ch(q_i.Projection)$ must initially include the corresponding $ch(param.CondJoin)$ if the junction attribute is not part of the projection attribute set (i.e., if $ch(param.CondJoin) \notin ch(q_i.Projection)$).

Given the *mDIS* of Fig. 7 and the *mSaaS* shown in Fig. 4, the *generateURLs()* generates the following URL: URL$_1$ = <http://w7.com/api/w/?dsw=w7&rcw=name,age&q=age=10& sort=name&rows=10>.

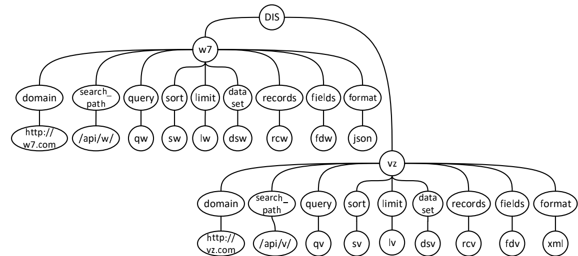

Figure 7: Example of *mSaaS* for query in Table 2.

On the other hand, given the same *mDIS* from the previous example (Fig. 7) and the *mSaaS* shown in Fig. 5, the *generateURLs()* generates the following URLs: URL$_1$ = <http://w7.com/api/w/?dsw=x7&rcw=b1,name,age&

Table 2: Example of a query in SQL and in NoSQL (MongoDB) with join/aggregation.

SQL	NoSQL (MongoDB)	
`SELECT w7.name,`	`db.w7.aggregate.([`	(from)
`w7.age, vz.phone`	`{$lookup:{from:'vz', localField:'b1',`	(join)
`FROM w7`	`foreignField:'b2'}},`	
`LEFT OUTER JOIN vz`	`{$match:{w7.b1='queens'}},`	(where)
`ON w7.b1=vz.b2`	`{$project:{w7.name:1, w7.age, vz.phone:1}}`	(select)
`WHERE w7.b1='queens'`	`{$sort:{w7.name:1}},`	(order by)
`ORDER BY w7.name`	`{$limit:3}`	(limit)
`LIMIT 3`	`]);`	

qw=b1='queens'>; and URL$_2$ = <http://vz.com/api/v/?dsv=vz&rcv=b2,phone>.

4.3 Canonical Model mDaaS

For each generated URL, the corresponding DaaS returns the request dataset. Before sending the results to SaaS, MIDAS performs some operations to make the data "presentable", such as join, order by, and limit, if applicable. This treatment is carried out employing the canonical model *mDaaS*.

Definition 9 (mDaaS). *The canonical model mDaaS maps the output of n DaaS. DaaS sends a return (in the format described in the mDIS) for each URL. If $n = 1$, then mDaaS just converts $ch(DIS.ch(q_1.dataset).format)$ (format returned by DaaS) into $ch(param.Return)$ (format desired by SaaS), and the process is finalized. On the other hand, when $n \geq 2$ (i.e., if there is a join), then the relations are mapped two-by-two, so that mDaaS generates n canonical mappings. In this case ($n \geq 2$), mDaaS is a tuple $mDaaS = (N_{root}, CJ)$, where: N_{root} is the name of the DaaS model (q_iD, i is the i-th relation); and CJ is a distinct set of $ch(param.CondJoin)$ (cj) values in the corresponding relation.*

Definition 10 (cj). *An information $cj \in CJ$ is a value that the condition of join $ch(param.CondJoin)$ assumes in the corresponding relation, being cj a tuple $cj = (N_{root_{cj}}, L)$, where: $N_{root_{cj}}$ is the name that identifies the value cj; and L is a set of lists (l) with all attributes that contain cj.*

Definition 11 (l). *A list $l \in L$ contains all elements of the same tuple in which cj is part, in the same order of occurrence of the relation (considering from left to right). The amount of $l \in L$ is equal to the amount of occurrences of cj in the relation, thus $l = \{a_1, a_2, \ldots, a_m\}$, where: a_i is the i-th attribute a for each l in cj; and m is the number of attributes $a \in l$.*

Considering that the query in Table 2 (with join) returns the two sets of data presented in Table 3, the canonical models (*mDaaS*) are shown in Fig. 8.

Table 3: DaaS returns for query presented in Table 2.

w7			vz	
b1	name	age	b2	phone
1	Andrew	14	1	p1
2	Bruce	35	1	p2
1	Carl	13	30	p3
3	Dylan	34	2	p4
2	Erik	65	5	p5

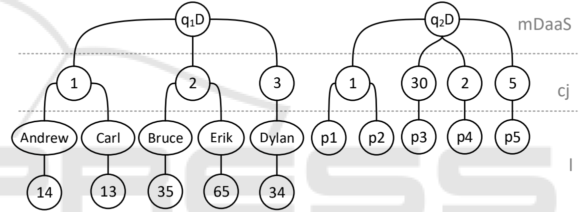

Figure 8: Example of *mDaaS* for query in Table 2.

Once the *mDaaS* has been generated, the join can be done. The next step depends on the value of $ch(param.TypeJoin)$. For this, in addition to the functions already mentioned, we assume that: $lch(p)$ is a function that returns the last child of a p node; and $con(p_1, p_2)$ is a function that connects the node p_1 to the node p_2.

If $ch(param.TypeJoin) =$ '$left\ outer$', the join is performed as follows:

a) $\forall cj_1 \in ch(q_1D)$ e $\forall cj_2 \in ch(q_2D)$, $con(lch(q_1D.cj_1), ch(q_2D.cj_2)), \forall cj_1 = cj_2$;

b) case $cj_1 \notin ch(q_1.Projection)$, then (i) $con(q_1D, ch(q_1D.cj_1))$ is performed and (ii) cj_1 is removed;

c) if there is $ch(param.OrderBy)$, this node is sorted;

d) if there is $ch(param.Limit)$, this must be the total of $ch(q_1D)$; and finally

e) q_1D is converted to $ch(param.Return)$ and it is sent to SaaS.

Considering the *mDaaS* of Fig. 8, the execution of the described steps should result in Fig. 9.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

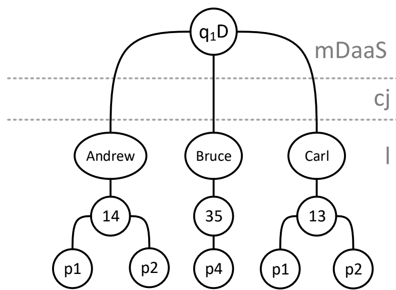

Figure 9: Example of *mDaaS* for query presented in Table 2 after left outer join.

5 EVALUATION

To evaluate our middleware, we performed a set of three experiments. These experiments delimit the relationship between SaaS and DaaS/DBaaS. A query without join (or aggregation) statement connects one SaaS to only one DaaS or one DBaaS provider (Experiments 1 and 2). Queries with join (or with aggregation) statements allow SaaS level to relate more than one DaaS and/or more than one DBaaS providers (Experiment 3).

Firstly, we evaluate the overhead of our middleware. We submitted 100 queries directly to both DaaS and DBaaS and, we compared the results with MIDAS access. Queries were performed to return 100, 1000, and 10000 records. Secondly, we evaluate whether the query language (SQL and NoSQL) influences the access time to different data sources (DaaS and DBaaS). Through MIDAS, we have submitted 100 queries: (i) With MongoDB to DaaS; (ii) SQL to DaaS; (iii) MongoDB to DBaaS; and (iv) SQL to DBaaS. Thirdly, we evaluate the interoperability of our proposal. In this experiment, we submit 100 queries to more than one data source: (i) 2 DBaaS; (ii) 2 DaaS; and (iii) 1 DaaS and 1 DBaaS.

In experiment 1 we evaluated overhead; experiment 3 we evaluated interoperability; and in all experiments (1, 2 and 3) we evaluated function and execution time. The average time (in ms) of each task was registered by Apache JMeter tool (http://jmeter.apache.org/).

5.1 Our Case Study

Our current MIDAS is based on open source technologies that are found in any cloud with PHP support. It was developed in Heroku cloud (https://www.heroku.com/) because it is an open cloud with sufficient storage space and a complete Platform as a Service (PaaS) for our project. To simulate a SaaS provider, we develop a Demographic Statistics by NY Hospital's web application based on PHP. This web application is hosted in Heroku SaaS instance, and it can be accessed at <https://midas-saas.herokuapp.com>.

Regarding DaaS service level, three different DaaS providers are used to perform our tests and experiments (P_1: <https://goo.gl/7sVsZB>; P_2: <https://goo.gl/E4YmYH>; and P_3: <https://goo.gl/vJomwT>):

- P_1: Transportation Sites, with 13600 instances and 18 attributes;
- P_2: Hospital General Information, with 4812 instances and 29 attributes; and
- P_3: Demographic Statistics By Zip Code, with 236 instances and 46 attributes.

The same dataset provided by DaaS were persisted into two DBaaS: P_1 in JawsDB (https://www.jawsdb.com/) and P_2 in mLab (https://www.mlab.com/). The DBaaS are based on MySQL and MongoDB, respectively. The choice for MySQL and MongoDB was motivated by being the most widely used Relational and NoSQL available and free (according to ranking https://db-engines.com/en/ranking). Our application (simulating SaaS) performs a join between P_2 and P_3.

5.2 Experiments

To evaluate our middleware, we performed three experiments: (E_1) overhead; (E_2) performance of different queries; and (E_3) data join and interoperability.

In the first experiment, we submitted 100 queries to both data sources (DaaS and DBaaS) with and without MIDAS. We vary the number of records returned (100, 1000, and 10000). This allows evaluating the influence of MIDAS on the communication between SaaS and DaaS/DBaaS. For this, in the first experiment we submit:

- 100 queries directly to DaaS provider;
- 100 queries to DaaS provider through MIDAS;
- 100 queries directly to DBaaS provider; and
- 100 queries to DBaaS provider through MIDAS.

As stated, we evaluated whether the query language influences access time depending on the data source. Thus, in the second experiment we submit:

- 100 MongoDB queries to the DaaS provider through MIDAS;
- 100 SQL queries to the DaaS provider through MIDAS;

- 100 MongoDB queries to the DBaaS provider through MIDAS; and
- 100 SQL queries to the DBaaS provider through MIDAS.

Finally, our third experiment evaluates the interoperability of MIDAS. We estimate the average execution time required for MIDAS to relate data from different sources, through the join (or aggregation) statement. The association of the data was made through a zip code field. having in dataset P_1 the attribute as *Zip* and in the dataset P_2 the attribute as *Zip Code*. For this, we submit:

- 100 queries with join statement to two DaaS providers through MIDAS;
- 100 queries with join statement to two DBaaS providers through MIDAS; and
- 100 queries with join statement to one DaaS and one DBaaS providers through MIDAS.

6 RESULTS

In this section, we present the results of our experiments, and we discuss them.

6.1 Results from Experiment 1

The results obtained from experiment 1 were classified based on the value assigned to the query limit. This value defines the number of records returned and it was restricted up to 100, 1000 and 10000 data records.

Firstly, we submitted 100 queries to return 100 data records. In this case, Fig. 10 shows the average of the execution time:

- 1267.77 ± 276.22 ms for queries without MIDAS to DaaS;
- 2052.37 ± 2658.98 ms for queries through MIDAS to DaaS;
- 489.76 ± 367.30 ms for queries without MIDAS to DBaaS; and

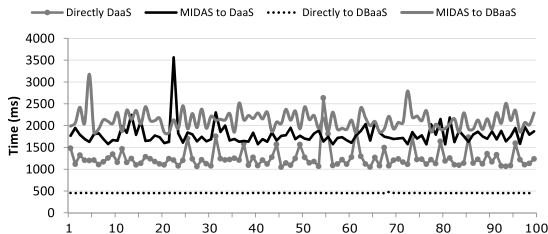

Figure 10: Return time (y-axis) for each of the 100 queries submitted (x-axis) with a limit of 100 records.

- 2128.15 ± 219.87 ms for queries through MIDAS to DBaaS.

Secondly, we submitted 100 queries to return 1000 data records. In this case, Fig. 11 shows the average of execution time:

- 1372.92 ± 275.70 ms for queries without MIDAS to DaaS;
- 3071.09 ± 585.30 ms for queries through MIDAS to DaaS;
- 896.51 ± 22.83 ms for queries without MIDAS to DBaaS; and
- 2813.19 ± 198.26 ms for queries through MIDAS to DBaaS.

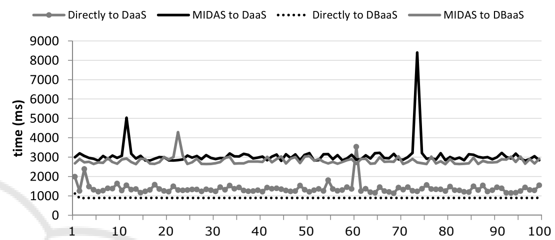

Figure 11: Return time (y-axis) for each of the 100 queries submitted (x-axis) with a limit of 1000 records.

Finally, we submitted 100 queries to return 10000 data records. In this case, Fig. 12 shows the average of execution time:

- 7917.02 ± 1045.84 ms for queries without MIDAS to DaaS;
- 35039.22 ± 1420.75 ms for queries through MIDAS to DaaS;
- 4260.8 ± 61.25 ms for queries without MIDAS to DBaaS; and
- 30023.41 ± 1213.57 ms for queries through MIDAS to DBaaS.

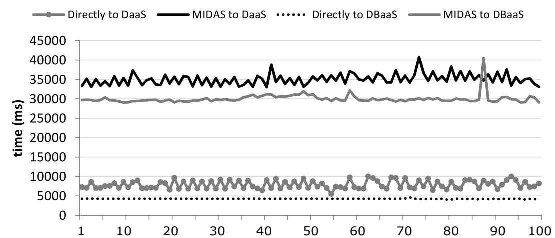

Figure 12: Return time (y-axis) for each of the 100 queries submitted (x-axis) with a limit of 10000 records.

Regarding the overhead caused by MIDAS, we can observe that the average differences of direct queries to DaaS and DBaaS, respectively, when compared to the access through MIDAS were: (i) 42.4%

and 368.8%, for 100 data records; (ii) 123.7% and 213.8%, for 1000 data records; and (iii) 342.6% and 604.6%, for 10000 records. Time values are affected by (i) data traffic on the Internet and (ii) MIDAS infrastructure. These results demonstrate that the algorithms need optimizations, not being the scope of this work.

6.2 Results from Experiment 2

In this experiment, we combine two query languages (SQL and NoSQL) with both sources (DaaS and DBaaS).

As Fig. 13 shows, the following averages of execution time were obtained:

- 33569.03 ± 2663.39 ms for MongoDB queries through MIDAS to DaaS;
- 35039.22 ± 1420.75 ms for SQL queries through MIDAS to DaaS;
- 29415.03 ± 1065.52 ms for MongoDB queries through MIDAS to DBaaS; and
- 30023.41 ± 1213.57 ms for SQL queries through MIDAS to DBaaS.

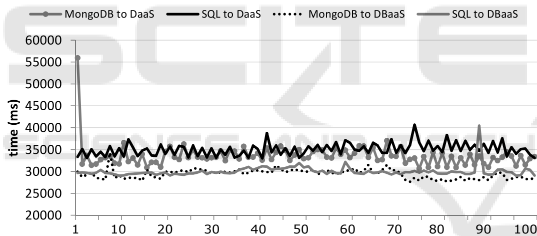

Figure 13: Return time (y-axis) for each of the 100 queries submitted (x-axis) from different languages to different data sources.

We can observe that: (i) For access to DaaS, SQL queries were 4.4% slower; while (ii) for DBaaS access, SQL queries were 2% slower. The time difference between the two types of queries is minimal, not representing losses in the choice of which to use.

6.3 Results from Experiment 3

In this experiment, we performed a query with join statements that access two different DaaS, two different DBaaS and one DaaS with one DBaaS.

Figure 14 depicts the average of the execution time.

- 12357.08 ± 6831.42 ms for two DaaS providers;
- 126957.46 ± 55870.66 ms for two DBaaS providers; and

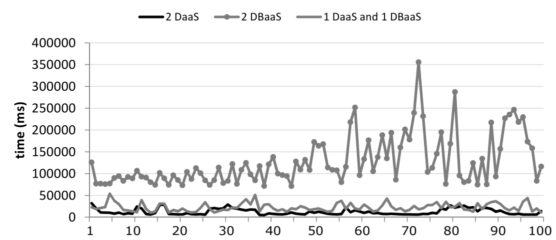

Figure 14: Return time (y-axis) for each of the 100 queries (x-axis) with join (or aggregation) statement.

- 22707.84 ± 9324.02 ms for one DaaS and one DBaaS

In this experiment, we can observe that (i) the average query time to 2 DBaaS is 459% slower than 1 DaaS and 1 DBaaS queries, and 927.4% slower than 2 DaaS queries. The average time for queries to 1 DaaS and 1 DBaaS is 83.8% slower than queries to 2 DaaS. When using DBaaS, the time values are higher than those presented by DaaS, due to the process of accessing and processing the data in the DBaaS.

6.4 Discussions

Our case study evaluates MIDAS through its overhead and different languages and data sources.

Despite the fact that the execution time was proportional to the submitted query, in the first experiment the results show that MIDAS inputs an extra overhead regarding direct queries. This depreciation was expected because of the new layer introduced between SaaS and DaaS. It is noteworthy that network bandwidth, cloud providers, and latency might also influence those results.

Considering DbaaS, we observed that the result from a direct access is more rapid than through MIDAS. In fact, MIDAS deals with DBaaS as a DaaS, through the Data Mapping module.

The second experiment states that the language used by a SaaS (i.e., SQL, NoSQL) does not influence the query performance or the return time with both data (i.e., DaaS, DbaaS).

Finally, the third experiment states that DbaaS needs to be deeply analyzed. Despite the fact that the join clause has a complexity $O(n^2)$ (2: number of data source), the join execution time decreases the performance in almost 1 minute. On the other hand, results on DaaS were less than 23 seconds. We can state that the benefits of our approach to interoperate different data sources by the use of join clauses outperforms the time spent on gathering the results.

There is one threat of validity: all data sources were public. Thus, offline data for any cause can compromise our approach.

7 CONCLUSIONS AND FUTURE WORK

In this paper, we propose a new version of MIDAS, describing a formal model to provide interoperability among cloud layers. We performed some experiments to validate our results and to show the effectiveness of our proposal.

Our middleware requires a minimum adaptation from SaaS applications despite the complexity of dealing with interoperability problem between application services and heterogeneous data in cloud environments. As contributions, unlike the previous version (1.6) our solution (i) even promotes the joining of data from different DaaS and DBaaS, enabling gathering data from various data sources; (ii) automatically populates and maintains updated the DIS; and (iii) considers other SaaS return formats in addition to JSON.

As a future work, we intend to continue improving MIDAS by adding new characteristics, such as (i) recognization of SPARQL queries and other types of NoSQL; (ii) automate the Crawler for searching novel DaaS and disambiguate data from heterogeneous data sources, and (iii) improve algorithms for better results.

ACKNOWLEDGEMENTS

The authors would like to thank FAPESB (Foundation for Research Support of the State of Bahia) for financial support.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

REFERENCES

Ali, H., Moawad, R., and Hosni, A. A. F. (2016). A Cloud Interoperability Broker (CIB) for data migration in SaaS. In *2016 IEEE International Conference on Cloud Computing and Big Data Analysis (ICCCBDA)*, pages 250–256.

Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., and Zaharia, M. (2010). A view of cloud computing. *Commun. ACM*, 53(4):50–58.

Barouti, S., Alhadidi, D., and Debbabi, M. (2013). Symmetrically-private database search in cloud computing. In *Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International Conference on*, volume 1, pages 671–678. IEEE.

Gantz, J. and Reinsel, D. (2012). The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east. *IDC iView: IDC Analyze the future*, 2007(2012):1–16.

Hacigumus, H., Iyer, B., and Mehrotra, S. (2002). Providing database as a service. In *Data Engineering, 2002. Proceedings. 18th International Conference on*, pages 29–38. IEEE.

Igamberdiev, M., Grossmann, G., Selway, M., and Stumptner, M. (2016). An integrated multi-level modeling approach for industrial-scale data interoperability. *Software & Systems Modeling*, pages 1–26.

Loutas, N., Kamateri, E., Bosi, F., and Tarabanis, K. (2011). Cloud computing interoperability: The state of play. In *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on*, pages 752–757. IEEE.

Mell, P., Grance, T., et al. (2011). The NIST definition of cloud computing.

Park, H.-K. and Moon, S.-J. (2015). DBaaS using HL7 based on XMDR-DAI for medical information sharing in cloud. *International Journal of Multimedia and Ubiquitous Engineering*, 10(9):111–120.

Schreiner, G. A., Duarte, D., and Mello, R. d. S. (2015). SQLtoKeyNoSQL: a layer for relational to key-based nosql database mapping. In *Proceedings of the 17th International Conference on Information Integration and Web-based Applications & Services*, page 74. ACM.

Sellami, R., Bhiri, S., and Defude, B. (2014). ODBAPI: a unified REST API for relational and NoSQL data stores. In *Big Data (BigData Congress), 2014 IEEE International Congress on*, pages 653–660. IEEE.

Silva, G. C., Rose, L. M., and Calinescu, R. (2013). A systematic review of cloud lock-in solutions. In *Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International Conference on*, volume 2, pages 363–368. IEEE.

Vieira, M. A., Ribeiro, E. L. F., Rocha, W. S., Mane, B., Claro, D. B., Oliveira, J. S., and Lima, E. (2017). Enhancing midas towards a transparent interoperability between saas and daas. In *Proceedings of the XIII Brazilian Symposium on Information Systems*, pages 356–363.

Xu, J., Shi, M., Chen, C., Zhang, Z., Fu, J., and Liu, C. H. (2016). ZQL: A unified middleware bridging both relational and nosql databases. In *Dependable, Autonomic and Secure Computing, 14th Intl Conf on Pervasive Intelligence and Computing, 2nd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech), 2016 IEEE 14th Intl C*, pages 730–737. IEEE.