Enhancing MIDAS towards a transparent interoperability between SaaS and DaaS

Marcelo Aires Vieira, Elivaldo Lozer Fracalossi Ribeiro, Witã dos Santos Rocha, Babacar Mane, Daniela Barreiro Claro, Joevan Santos de Oliveira and Edmilson Lima
FORMAS - Research Group on Formalisms and Semantic Applications
Institute of Mathematics, Federal University of Bahia
s/n, Adhemar de Barros Ave, Ondina, Zip Code: 40170-110, Salvador, Bahia, Brazil
mairesweb@gmail.com, elivaldolozerfr@gmail.com, witasrocha@gmail.com, mbabacar@gmail.com, dclaro@ufba.br, joevansantos@hotmail.com, edmilsons.s.lima@gmail.com

ABSTRACT

Over the years, Software as a Service (SaaS) has become a common delivery model for many applications. In cloud applications, a huge volume and variety of data can be generated and they can be available for consumption by DaaS (Data as a Service). For this, the data provided by DaaS can be stored in a non-structured (e.g. text), semi-structured (e.g. XML, JSON) or structured format (e.g. Relational Database). However, the access of that kind of DaaS, in a transparent manner, needs substantial efforts due to the lack of interoperability between SaaS and DaaS. In this paper, we propose a new enhanced version of MIDAS, middleware to provide seamlessly and independently interoperability between SaaS and DaaS. First, this new version of MIDAS allows both semi-structure and structure data format from SaaS. It mediates queries from NoSQL (e.g. MongoDB) and SQL (MySQL) databases. Secondly, it was enhanced with `Join` operations, both in SQL and in NOSQL statements. And lastly, other formats were added for the DaaS to fit SaaS requests, such as JSON, XML, and CSV formats. To evaluate this new version of our middleware, we provide three types of experiments to cover critical issues such as execution time, the overhead of our approach, and scalability of MIDAS. Our results show the effectiveness of our approach to tackling interoperability issues in cloud computing environments.

CCS Concepts

•**Information systems** → **Information integration;** *Middleware for databases;* •**Software and its engineering** → Software architectures;

Keywords

MIDAS, Cloud Computing, SaaS, DaaS, Interoperability.

SBSI 2017 June 5^{th} – 8^{th}, 2017, Lavras, Minas Gerais, Brazil

1. INTRODUCTION

Data in the digital universe will double every two years until 2020, and it is estimated to achieve 40 trillion gigabytes [4]. Data management have been facing some challenges to handle this variety and large intensive data. Such data needs to be stored, distributed, available and understandable both to consumers and to organizations. The cloud computing paradigm has emerged to fill some of these requirements. By 2020, nearly 40% of the available data will be managed and stored by a cloud computing provider [4].

Authors in [3] define cloud as the "data center for hardware and software that provides 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) and Data as a Service (DaaS) are instances of service types organized in cloud levels. SaaS are cloud applications that manage data and tackle with business processes. DaaS provides data on demand to a consumer through Application Programming Interfaces (APIs). 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 these data. Thus, 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. However, to access DaaS in different cloud providers, SaaS applications need, in most of the cases, substantial efforts by providing a mechanism to get it. This lock-in situation happens due to the lack of interoperability between SaaS and DaaS. For instance, if demographic researchers need to make studies about census data provided by governments in different DaaS, they will face the difficult to process these data due to the lack of standards and consequently no interoperability between SaaS and DaaS. To accomplish this interoperability issue, we propose a middleware called MIDAS (Middleware for DaaS and SaaS).

Although confusing, note that DaaS and DBaaS (database as a service) are different concepts [6, 10]. DBaaS refers to a complete database service, such as Oracle and MongoDB, and it allows full manipulation of stored information. These services can be in-memory, relational (traditional) or NoSQL databases, and they are offered on demand and hosted on a cloud. On the other hand, DaaS defines a dataset in a cloud service and allows controlled access (usually read-only) to

that data.

MIDAS is responsible for mediating the communication between different SaaS and DaaS providers, making possible that SaaS applications retrieve data seamlessly from a DaaS as it would be querying its datacenter. SaaS applications will be able to get data from DaaS datasets by sending a query to MIDAS and letting it mediate the communication and returning results. Our first version of MIDAS [6] tackled a mono data format. Data queries from SaaS were only made from a relational database (e.g. MySQL). SaaS applications got data from DaaS datasets by sending a query to MIDAS and letting it mediate the communication and return from DaaS providers.

We propose in this paper a new enhanced version of MIDAS with some significant concerns that were developed to provide a transparent interoperability concern. Firstly, this new version of MIDAS allows (i) both semi-structure and structure data format from SaaS. It mediates queries from NoSQL (e.g. MongoDB) and SQL (MySQL) databases. Secondly, (ii) it was enhanced with `Join` operations in SQL statements. Thirdly, results from DaaS providers was extended to other formats to fit SaaS requests, such as JSON, XML, and CSV formats. And lastly, two new scenarios were described to clarify the relevance of MIDAS usage.

We performed some experiments to evaluate our novel approach, considering three important issues: execution time, overhead and scalability. Our results demonstrated that our middleware is effective and efficient, thus providing the correct results in an expected execution time.

The remainder of this paper is organized as follows: Section 2 presents our most relevant related works; Section 3 describes our novel version of MIDAS. Section 4 provides our experiments within two scenarios to clarify the MIDAS usage. Section 5 discusses some results and, Section 6 concludes with some envision work.

2. RELATED WORKS

Some important works have been proposed to provide interoperability solutions, such as [1, 2, 5, 8].

The need to reduce the complexity of heterogeneous data structures is a crucial aspect of Big Data analysis. Authors in [5] state that the issue of interoperability in oil and gas industry is a problem to be overcome since there is no standard. As a solution, authors presented a framework to solve problems of formalization, implementation, and querying of Big Data systems, in the oil and gas area. The proposal was evaluated in a real case study: the OGI Pilot (Oil and Gas Interoperability Pilot), project in the oil and gas domain. The goal of framework is to automate the transfer of information between projects, identifying similarities and differences. Different from our approach, they manipulate data sources in isolation.

Authors in [8] present an initial effort to address interoperability issues of dialysis data since there is a lack of communication between systems. For this, a Federated Database System approach was proposed to build a shared repository. Different from our approach, they do not consider interoperability from data in different formats and in addition, because they use proprietary software (MATrix LABoratory - MATLAB), a lock-in problem may occur.

In the same domain, Cybernetic-Medical Physics Systems (MCPS) are intelligent systems for monitoring and controlling various patient characteristics through embedded devices. MCPS manipulate different types of data, collected by different types of sensors, in order to evaluate the patient's situation and make adequate treatment plan. With the diversity of sensors and formats, the data generated is not interoperable and accessible to all stakeholders, making decisions difficult. For this, authors in [1] studies the health data interoperability in cloud-based MCPS and propose a conceptual framework to provide data interoperability. However, they retrieved data from only one location, making it impossible to link more than one database.

Considering the Cloud domain and different DaaS providers, a middleware or API may be required [9].

In cloud computing, authors in [2] stated that interoperability limits users' actions. Thus, data migration between cloud providers may require significant effort. This study suggests a solution to interoperate different SaaS that would act as a mediator between the layers, called Cloud Interoperability Broker (CIB). The proposal was evaluated and tested on a dataset of the actual enterprise application. Different from our approach, they do not evaluate the performance of the proposal and they not consider the interoperability between SaaS data from different domains.

Park and Moon [7] suggest a solution for heterogeneous DBaaS to share medical data between different institutions. This system stores and shares medical information based on the HL7 (Health Level Seven) standard. However, the authors assume that databases can be manipulated by themselves (which is not our case). This is one of the reasons we chose to interoperate SaaS with DaaS (and not with DBaaS).

The most close approach is MIDAS 1.0 [6]. This middleware provides an interoperability between SaaS and DaaS. However, MIDAS 1.0 has some limitations: (i) Query Decomposer recognizes only SaaS queries with `select`, `from`, `where`, `order by`, and `limit`; (ii) Dataset Information Storage (DIS) only uses a relational database to store information about DaaS, and the information is filled in manually; (iii) Result Formatter returns data only with JSON format; and (iv) middleware only recognizes structured queries. Moreover, the authors do not evaluate the scalability through the middleware.

Our proposal aims to provide interoperability between SaaS and DaaS in a transparent way. Next section describe our enhanced MIDAS in detail, explaining each module.

3. OUR ENHANCED MIDAS

Our enhanced MIDAS is versionned as MIDAS 1.6. This middleware provides an improved version of MIDAS 1.0 in four levels of functionality: (i) data returned; (ii) semi-structured and structure queries, tackling with NoSQL and SQL databases; (iii) lightweight DIS, removing the relational database management system (RDBMS) from MIDAS; and (iv) joined DaaS, executing different DaaS in a single query.

MIDAS 1.6 architecture is depicted in Figure 1. This novel approach is composed of two modules: Request and Result. As the names suggest, Request Module handles the incoming SaaS query while Result Module process DaaS response.

The following subsections describe the whole architecture in detail and identify news functionalities added to each module.

3.1 SaaS Application

Queries sent to MIDAS are provided by consumers or through applications at SaaS level. In MIDAS 1.6 it is pos-

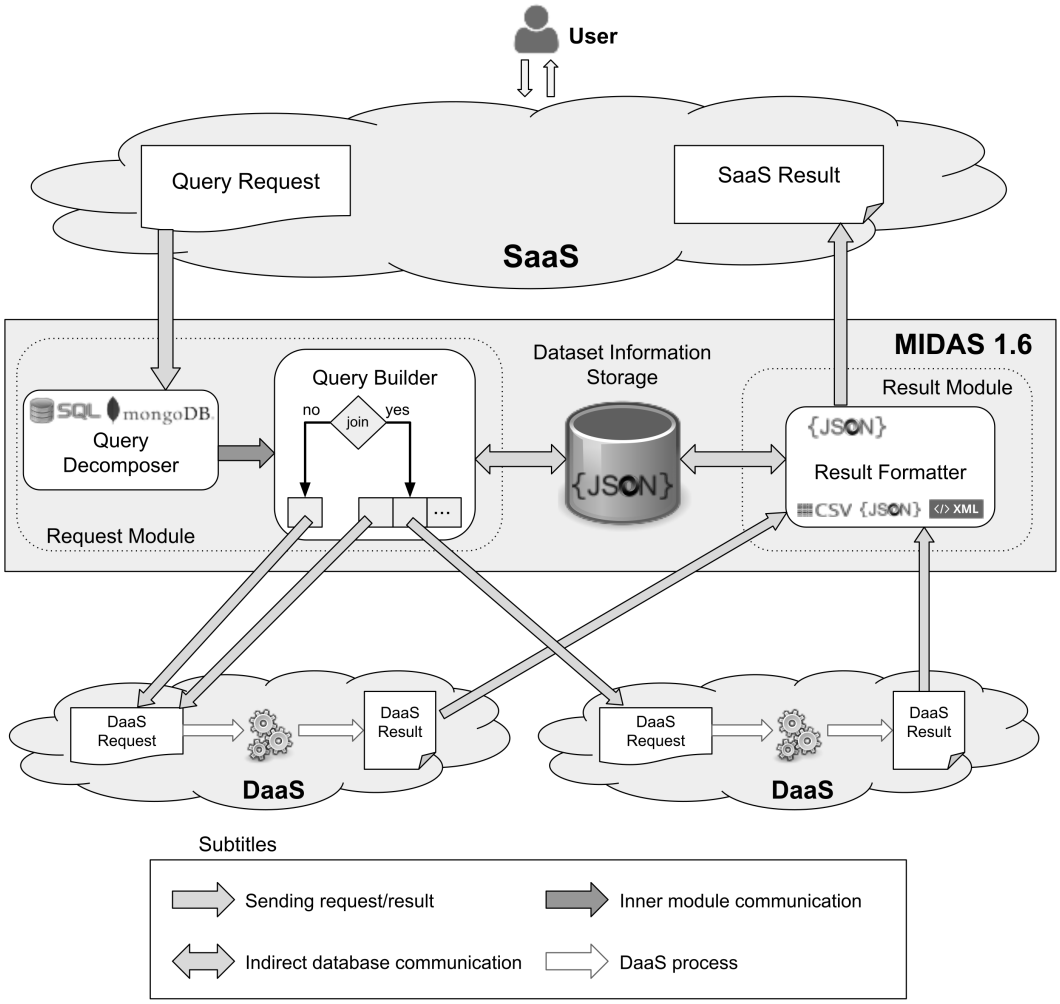

Figure 1: MIDAS 1.6 architecture.

sible to perform either SQL or NoSQL queries considering `join` and lookup clauses respectively.

3.2 MIDAS Middleware

MIDAS aims to provide a transparent communication between either different SaaS and DaaS providers, making possible that SaaS applications retrieve data seamlessly from a DaaS as it would be querying a datacenter.

The Sequence Diagram in Figure 2 describes the interactions among SaaS, MIDAS and DaaS providers. The SaaS user clicks the search button that sends a query SQL or NoSQL to MIDAS. MIDAS receives this query and does all inner steps previewed and then sends the query to DaaS provider. The query is executed by each DaaS provider if there is a `join` clause and returns the result to MIDAS. MIDAS receives this query and does the inner steps previewed and then sends the query to DaaS provider. The query is executed by each DaaS provider if there is a `join` clause and returns the result to MIDAS. MIDAS treats this result and sends it in the appropriate manner to SaaS.

Our middleware is composed of four modules: Query Decomposer, Dataset Information Storage (DIS), Query Builder and Result Formatter. Each module is deeply described in the next subsections.

3.2.1 Query Decomposer

Query Decomposer is responsible for breaking the structure of a query into arrays by mapping the contents of each query statement. For instance, to break a SQL structure like `SELECT name1, name2 FROM table`, the Query Decomposer creates two arrays: one for `select` clauses and the other for `from` clauses. Additionally, it recognizes and breaks a NoSQL query, creating an output equivalent to a structured query. The equivalence between SQL and NOSQL queries can be seen in Figure 3. This format ensures that the Query Builder can build the DaaS request independently of their API.

The Query Decomposer module takes the query as a parameter and creates an array, which collects the first index of each term of the SQL syntax. They are: `from`, `where`, `order by`, `limit`, `inner join`, and `on`. The term `select` is assumed as the first index since it has to head the query in SQL. Next, we store the query columns, the ones between `select` and `from` in the query, as an array inside our inde-

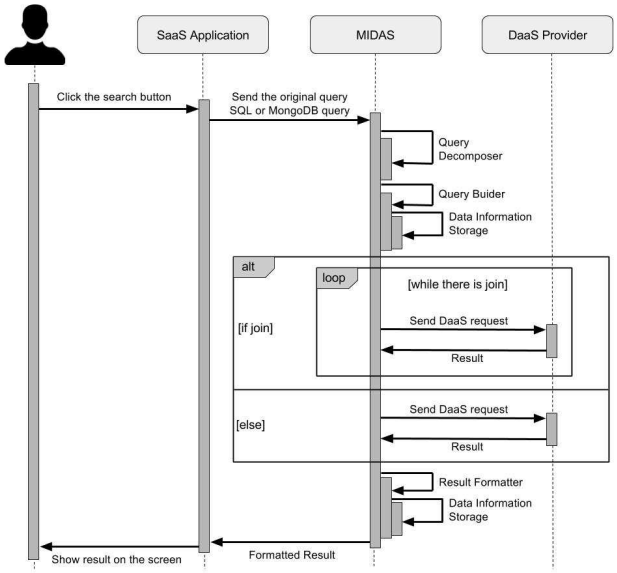

Figure 2: MIDAS sequence diagram.

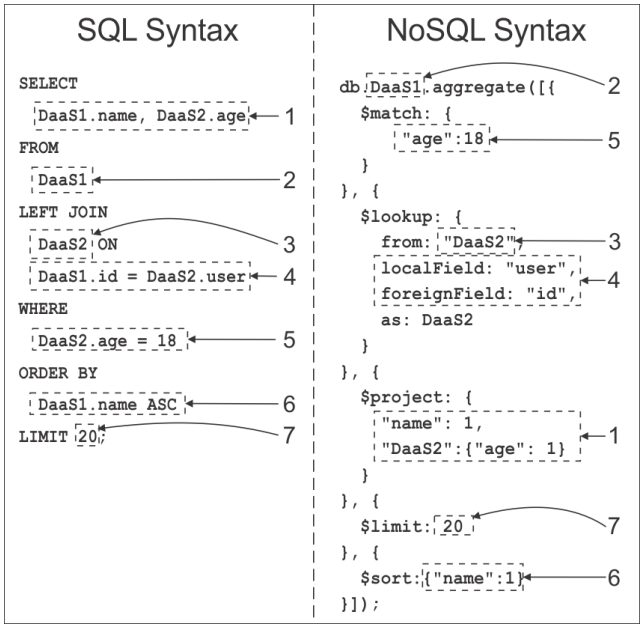

Figure 3: Equivalence between SQL and NOSQL (e.g. MongoDB) queries.

pendent format array in the position `fields`. The `dataset` key relates to the value passed in the `from` clause. This is the dataset 'id' of DaaS provider. This means that the SaaS application queries the dataset as it queries a relational table. The 'filters' key maps the `where` clause , 'order' maps `order by`, the 'limit' key maps the `limit` clause, and `inner join` refers to get data from different tables, and `on` maps the join comparative attributes.

On the other hand to NoSQL (e.g. MongoDB) queries, and the Query Decomposer module also takes the query as a parameter and collects the first index of each syntax term: `db`, `collection`, `find`, `limit`, `sort`, and `lookup`, without an index 0. Next, we store the query columns, the ones between `db` and `collection` in the query, as an array in the position — fields. The 'dataset' key relates to the value passed in the `find` clause, this will be the dataset "id" of the DaaS provider. This means that the SaaS application either queries the dataset as it queries a document. The 'filters' key maps the `limit`, 'order' maps `sort`, the 'limit' key maps the `limit` clause and the 'lookup' key maps the `join` clause.

3.2.2 Query Builder

The Query Builder was enhanced with the `join` clause. If the query does not contain a `join`, the process performed is similar to MIDAS 1.0: Query Builder receives the decomposed query, transforms the SaaS query into a DaaS request, and sends the query to the particular DaaS. On the other hand, if the query contains the `join` clause (one or more), Query Builder must recognize in the array sent by the Query Builder, all differents DaaS that should be consulted and then it sends the queries to each respective DaaS (see Figure 1). In both cases, the Query Builder needs to access the DIS to get information about the provider dataset. DIS information guides the Query Builder to match the right fields it has received from the Query Decomposer, including the parameters that compose the DaaS provider request. After matching, the Query Builder sends the query to the DaaS provider to be executed.

3.2.3 Dataset Information Storage

Dataset Information Storage (DIS) persists the information about DaaS providers' datasets and their APIs. In this new version, we changed the use of an RDBMS into a JSON file. This procedure enables to lightweight DIS and as a consequence, our novel MIDAS approach. Thereby, our novel MIDAS stays independent of any database, avoiding future lock-in. An example of DIS by the use of the JSON file can be seen in Figure 4. Since there is no way to get information about DaaS providers' automatically yet, the task to add new providers and to maintain them up to date must be still done manually. The DIS is composed of the following providers information:

- `dataset`: is a name that uniquely identifies the dataset among the providers;
- `domain`: is the first part of DaaS request URL built by the Query Builder;
- `search_path`: represents the piece of string in the DaaS request that concatenates to the provider domain and gives the path to access the API;
- `dataset_param`, `query_param`, `sort_param`, `limit_param` and `format_param`: they represent respectively the API parameter names to inform dataset, query filters, ordering field, the number of rows and return format in the result.

Each DaaS provider needs to store the relevant information manually, since our crawler is not running yet to obtain the information automatically.

3.2.4 Result Formatter

The Result Formatter added two news functionalities. In our previous version, the only data format to return from DaaS to Result Formatter was JSON. In this novel approach,

dis.json

```
{
    "nyc-wifi-hotspot-locations": {
        "domain": "http://public.opendatasoft.com",
        "search_path": "/api/records/1.0/search/",
        "query_param": "q",
        "sort_param": "sort",
        "limit_param": "rows",
        "dataset_param": "dataset",
        "records_param": "records",
        "fields_param": "fields",
        "format_param": "json"
    },
    "v8qe-fx6p.json": {
        "domain": "http://data.cityofnewyork.us",
        "search_path": "/resource/",
        "query_param": "$where",
        "sort_param": "$order",
        "limit_param": "$limit",
        "dataset_param": "",
        "records_param": "",
        "fields_param": "",
        "format_param": "json"
    }
}
```

Figure 4: Example of DIS within JSON file.

JSON, CSV, and XML are also enabled as possible formats. Considering queries with `join` clause, the Result Formatter must be able to tackle with different DaaS objects. Each return could be in a different format (JSON, CSV or XML). In particular cases, some DaaS have a restriction to project their attributes. This provides an extra overhead to the Result Formatter because it needs to gather the SaaS request to return the same attributes required.

3.3 DaaS Provider

MIDAS supports DaaS providers that return data in XML, JSON, or CSV format. Thereby, the DaaS provider does not have to make any changes in its DaaS service. Thus, the provider will receive the request sent by MIDAS and process into the dataset. After gathering all the data requested, the provider will return the information to MIDAS. The number of DaaS providers is obtained in the SaaS request. As the number of DaaS is different from one, an iteration allows to query each respective DaaS provider.

4. TEST SETS

To evaluate our middleware, we performed two experiments by executing a query firstly without `join` statement and secondly with `join` clause. These two phases determine the relationship type between SaaS and DaaS level. A query provided with `join` statement allows SaaS level to relate to more than one DaaS providers. The number of DaaS depends on how many datasets specified in the `join` clause. A query without `join` statement connects one SaaS to only one DaaS provider.

Three measures are used during our evaluation process: execution time, the overhead caused by MIDAS and scalability.

4.1 Our Case Study

Our middleware MIDAS is being developed in Heroku Cloud. Three mains reasons motivated it: (i) Heroku is an Open Cloud; (ii) it is considered as a complete PaaS platform (support different types of development environment); and finally (iii) Heroku provides enough storage space for our project.

To simulate a SaaS provider, we develop a tourism agency web application based on HTML5 + CSS3 + JavaScript frontend (Bootstrap and jQuery), PHP 5.6 backend and framework Laravel 5.3. This web application is hosted in the Heroku SaaS public instance and it can be accessed at `https://midas-saas.herokuapp.com/`.

Heroku cloud provides the DaaS service level by offering MySQL and MongoDB NoSQL database. The choice of MongoDB database was motivated as it is a native support by Heroku and also one of the most used by NoSQL community.

4.2 Our DaaS instances

Three different DaaS providers are used to perform our tests:

- $\texttt{DaaS}_1$[1]: Times Square Hotels, with 41 instances and 4 attributes;
- $\texttt{Daas}_2$[2]: Health and Hospitals Corporation (HHC) Facilities, with 78 instances and 6 attributes;
- $\texttt{DaaS}_3$[3]: Borough Enrollment Offices, with 13 instances and 6 attributes.

4.3 Experiments

We performed two different experiments to evaluate our approach. The first experiment measures the MIDAS without `join` clause and secondly, we evaluate our MIDAS dealing with the `join` clauses. For each experiment, we measure execution time, overhead of MIDAS, scalability and the effectiveness of our approach by performing 20 queries successively to retrieve 10, 100 and 1000 data records. The average time of each these tasks are registered by Hurl[4] tool.

4.4 Experiment 1: Performing queries without join

In this experiment we perform three tasks:

- 20 SQL queries are submitted directly to DaaS provider;
- 20 SQL queries are performing to DaaS provider through MIDAS; and
- 20 queries of MongoDB statement are executed to DaaS provider through MIDAS.

A DaaS provider we use in this experiment provides a list of hotels in the New York City metropolitan region (in `NYCOpenData`). The aim of this experiment is to evaluate the communication between a SaaS and DaaS provider through MIDAS and analyze the overhead caused by our middleware.

[1] `https://data.cityofnewyork.us/Business/Times-Square-Hotels/v8qe-fx6p`
[2] `https://data.cityofnewyork.us/Health/Health-and-Hospitals-Corporation-HHC-Facilities/f7b6-v6v3`
[3] `https://data.cityofnewyork.us/Education/Borough-Enrollment-Offices/vz8c-29aj`
[4] `https://www.hurl.it`

4.5 Experiment 2: Performing queries with join

In this second experiment, two activities are executed:

- 20 SQL queries with `join` statement are submitted to two DaaS providers through MIDAS; and
- 20 queries of MongoDB with `join` statement are executed to two DaaS providers through MIDAS.

In this case, we use two DaaS providers: one provides service about hospitals and the another about offices (both in `NYCOpenData`).

The objective of this experiment is to evaluate the performance of MIDAs when it relates one SaaS provider with two DaaS provider to perform SQL or MongoDB queries with `join` statement.

5. RESULTS AND DISCUSSION

In this section, we present our results per experiments and make analyses on each one.

5.1 Results from Experiment 1

Results obtained in this experiment are classified based on the value assigned to the `limit` clause of SQL and NoSQL query performed in this test. This value define the number of data records that the query is enable to retrieve. Three categories are specified for our experiment: 10, 100 and 1000 data records.

Firstly, We consider the situation where the 20 queries executed successively return 10 data records. In this case, we have the following results about averages of execution times:

- 236.5 ± 29.98 ms for queries without MIDAS;
- 262.25 ± 35.19 ms for SQL queries through MIDAS; and
- 248.00 ± 30.30 ms for MongoDB queries through MIDAS.

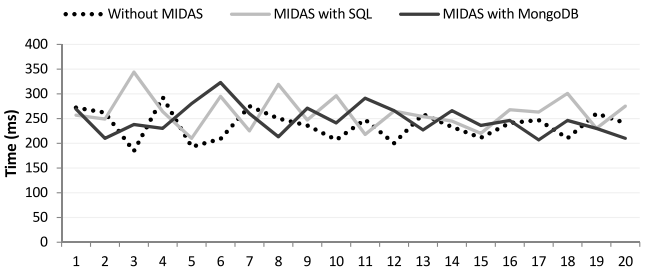

Figure 5: Return time (in ms) for each of the 20 queries submitted with a limit of 10 records.

Secondly for queries with 100 records returned we have the following averages of execution times:

- 269.40 ± 38.96 ms for direct queries to the datasource (without MIDAS);
- 507.88 ± 50.95 ms for SQL queries through MIDAS; and

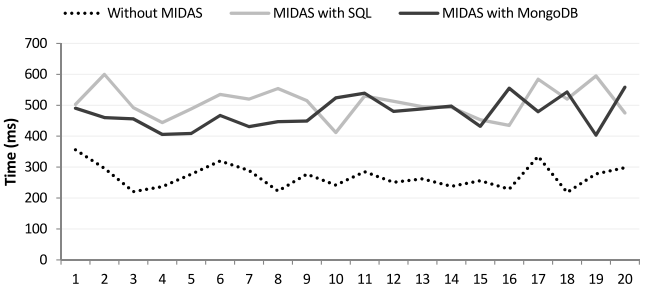

Figure 6: Return time (in ms) for each of the 20 queries submitted with a limit of 100 records.

- 475.65 ± 48.94 ms for MongoDB queries through MIDAS.

Finally, queries of 1000 records present the follow averages of execution times:

- 465.6 ± 65.77 ms for queries without MIDAS;
- 1087.75 ± 126.58 ms for SQL queries through MIDAS; and
- 1010.50 ± 134.53 ms for MongoDB queries through MIDAS.

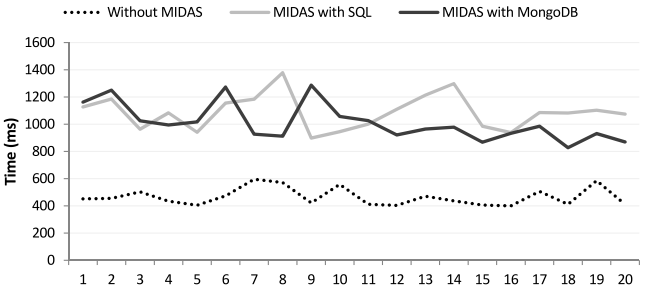

Figure 7: Return time (in ms) for each of the 20 queries submitted with a limit of 1000 records.

The return time of each query can be visualized in figures 5 (`limit` 10), 6 (`limit` 100), and 7 (`limit` 1000).

As expected, overhead is caused by the presence of our middleware - MIDAS. The reasons of this overhead can be distributed between the two components of our MIDAS: Query Decomposer and the Result Formatter.

As describe in this work, the Query Decomposer is considered as one of the most critical components in our middleware. It is responsible for preparing users queries to be built into a URL for executing by a DaaS provider. Therefore, this stage of processing requires a certain amount of time to be performed. Figure 8 shows the importance of Query Decomposer component in the process of treatment of queries in MIDAS. Its responsibility is to transform regardless of the nature and type of query (SQL or MongoDB) sent by the user, in a single format for the query builder component.

The result Formatter also need time processing to organize the results into a different data format (XML, JSON, and CSV) for users.

In short, there is no significant difference in term of results in the three experimental situations considering only

the execution of queries intermediate by the MIDAS. The largest variation occurred in the return of 100 records with MongoDB queries: 8.38% faster than SQL queries. We can conclude that MIDAS is scalable through an increasing number of queries.

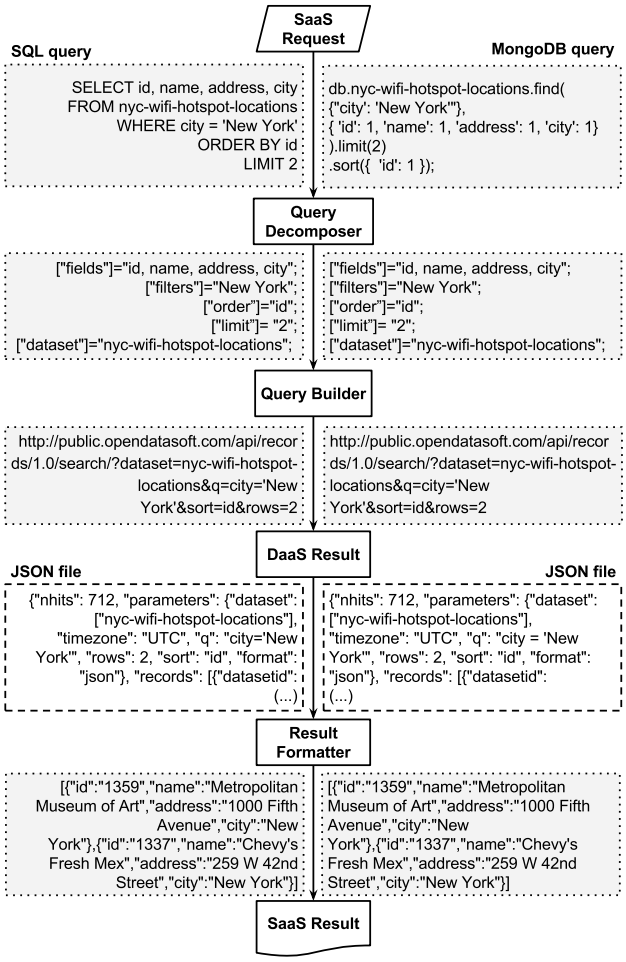

Figure 8: Steps of MIDAS.

5.2 Results from Experiment 2

In this experiment, we performed a query with `join` statements that allow accessing two different DaaS providers (Enrollment Offices and Hospitals).

As presented in Figure 9, the averages of execution times are:

- 201.36 ± 26 ms for SQL query; and
- 240.7 ± 31.36 ms for MongoDB query.

These two experiments allow us to evaluate and analyze the effectiveness, scalability, overhead, and execution times of our middleware MIDAS to ensure interoperability between SaaS and DaaS providers. Concerning the measures obtained, we found that the differences obtained in performing the queries are not perceptible from the users. A proper evaluation of MIDAS scalability characteristics allows us to suggest in the future integration of our middleware to Big Data solutions. These observations will be considered for next improvement of our middleware.

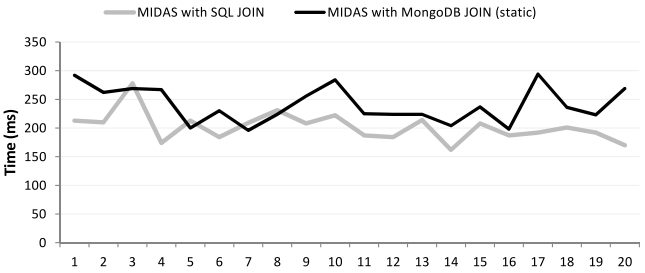

Figure 9: Return time (in ms) for each of the 20 queries with join submitted.

6. CONCLUSION AND FUTURE WORK

In this paper, we proposed a new version of MIDAS whose purpose was to address important issues left open in the first version. These requirements were presented in four new types of functionalities: (i) more robust support to SQL Standard queries by adding `join` statement; (ii) returned data in different formats from DaaS to Result Formatter of MIDAS; (iii) recognized a simple queries to a MongoDB NoSQL database; and (iv) changed the DIS from relational database (MySQL) to JSON file to lightweight our MIDAS and to avoid lock-in future problems. Within this new version of MIDAS, SaaS applications continue to query DaaS datasets transparently as it is querying a database with a minimal adaptation for SaaS and DaaS providers.

Our results show the effectiveness of our approach to tackling interoperability issues in Cloud Computing environments.

In future work, we intend to continue improving MIDAS by adding news characteristics such as (i) recognizes queries in SPARQL and other NoSQL; (ii) returns data in XML, JSON, and CSV formats by Result Formatter to SaaS applications or users; and (iii) develop a crawler to automate the search of DIS information.

7. ACKNOWLEDGMENTS

The authors would like to thank FAPESB (Foundation for Research Support of the State of Bahia) and CNPq (National Counsel of Technological and Scientific Development) for financial support.

8. REFERENCES

[1] M. A. Alhumud, M. A. Hossain, and M. Masud. Perspective of health data interoperability on cloud-based medical cyber-physical systems. In *Multimedia & Expo Workshops (ICMEW), 2016 IEEE International Conference on*, pages 1–6. IEEE, 2016.

[2] H. Ali, R. Moawad, and A. A. F. Hosni. A cloud interoperability broker (cib) for data migration in saas. In *Cloud Computing and Big Data Analysis (ICCCBDA), 2016 IEEE International Conference on*, pages 250–256. IEEE, 2016.

[3] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[4] J. Gantz and D. Reinsel. 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, 2012.

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

[6] T. Marinho, V. Cidreira, D. B. Claro, and B. Mane. Midas: A middleware to provide interoperability between saas and daas. In *Proceedings of the XII Brazilian Symposium on Information Systems on Brazilian Symposium on Information Systems: Information Systems in the Cloud Computing Era-Volume 1*, page 53. Brazilian Computer Society, 2016.

[7] H.-K. Park and S.-J. Moon. Dbaas using hl7 based on xmdr-dai for medical information sharing in cloud. *International Journal of Multimedia and Ubiquitous Engineering*, 10(9):111–120, 2015.

[8] D. Vito, G. Casagrande, C. Bianchi, and M. L. Costantino. An interoperable common storage system for shared dialysis clinical data. In *Student Conference (ISC), 2016 IEEE EMBS International*, pages 1–4. IEEE, 2016.

[9] Z. Zhang, C. Wu, and D. W. Cheung. A survey on cloud interoperability: taxonomies, standards, and practice. *ACM SIGMETRICS Performance Evaluation Review*, 40(4):13–22, 2013.

[10] Z. Zheng, J. Zhu, and M. R. Lyu. Service-generated big data and big data-as-a-service: An overview. In *2013 IEEE International Congress on Big Data*, pages 403–410, June 2013.