*Journal of Universal Computer Science, vol. 27, no. 4 (2021), 387-412*
*submitted: 14/8/2019, accepted: 6/4/2021, appeared: 28/4/2021 CC BY-ND 4.0*

# Integration Model between Heterogeneous Data Services in a Cloud

**Marcelo Aires Vieira**
(FORMAS Research Group, Computer Science Department, IME/LaSiD, Federal University of Bahia (UFBA), Salvador, Brazil
https://orcid.org/0000-0002-7681-5279, mairesweb@gmail.com)

**Elivaldo Lozer Fracalossi Ribeiro**
(FORMAS Research Group, Computer Science Department, IME/LaSiD, Federal University of Bahia (UFBA), Salvador, Brazil, and
Federal University of Southern Bahia (UFSB), Porto Seguro, Brazil
https://orcid.org/0000-0002-1697-9570, elivaldolozerfr@gmail.com)

**Daniela Barreiro Claro**
(FORMAS Research Group, Computer Science Department, IME/LaSiD, Federal University of Bahia (UFBA), Salvador, Brazil
https://orcid.org/0000-0001-8586-1042, dclaro@ufba.br)

**Babacar Mane**
(FORMAS Research Group, Computer Science Department, IME/LaSiD, Federal University of Bahia (UFBA), Salvador, Brazil
https://orcid.org/0000-0002-9519-2847, mbabacar@gmail.com)

**Abstract:** With the growth of cloud services, many companies have begun to persist and make their data available through services such as Data as a Service (DaaS) and Database as a Service (DBaaS). The DaaS model provides on-demand data through an Application Programming Interface (API), while DBaaS model provides on-demand database management systems. Different data sources require efforts to integrate data from different models. These model types include unstructured, semi-structured, and structured data. Heterogeneity from DaaS and DBaaS makes it challenging to integrate data from different services. In response to this problem, we developed the Data Join (DJ) method to integrate heterogeneous DaaS and DBaaS sources. DJ was described through canonical models and incorporated into a middleware as a proof-of-concept. A test case and three experiments were performed to validate our DJ method: the first experiment tackles data from DaaS and DBaaS in isolation; the second experiment associates data from different DaaS and DBaaS through one join clause; and the third experiment integrates data from three sources (one DaaS and two DBaaS) based on different data type (relational, NoSQL, and NewSQL) through two join clauses. Our experiments evaluated the viability, functionality, integration, and performance of the DJ method. Results demonstrate that DJ method outperforms most of the related work on selecting and integrating data in a cloud environment.

**Keywords:** Cloud Computing, Data Integration, DaaS, DBaaS
**Categories:** H.3.1, H.3.2, H.3.3, H.3.7, H.5.1
**DOI:** 10.3897/jucs.67046

## 1 Introduction

Digitally stored data has grown exponentially, with an estimated 175 Zettabytes in 2025 [Reinsel et al. 2018]. Currently, internet data traffic exceeds five billion gigabytes per day [Internet Live Stats 2020]. This amount of data, distribution, availability, and plurality makes the aggregation of data a new challenge. Cloud computing minimizes some of these requirements by providing services with high availability. Cloud computing also facilitates the publication and distribution of data [Mell and Grance 2011].

Cloud computing is a ubiquitous network of on-demand services, composed of applications, platforms, and hardware in a hierarchical model. Usually, a cloud is described in three levels: (i) Infrastructure as a Service (IaaS); (ii) Platform as a Service (PaaS); and (iii) Software as a Service (SaaS) [Mell and Grance 2011, Armbrust at al. 2010]. Based on these levels, cloud providers enhanced other service models, such as Data as a Service (DaaS) and Database as a Service (DBaaS) [Li et al. 2012, Zheng et al. 2013]. These two data-based models are conceptually different [Zheng et al. 2013]. DaaS provides data on demand through interfaces based on Web Services Description Language (WSDL) or Representational State Transfer (REST) [Barros et al. 2018], such as Brazilian Open Data Portal[1] and DATA.GOV[2]. DBaaS provides Database Management Systems (DBMSs) for organizations to store, access, and manipulate their databases [Zheng 2018]. Examples of DBaaS are ClearDB[3] (based on MySQL), ElephantSQL[4] (based on PostgreSQL), and BD Cosmos[5] (based on NoSQL). Table 1 presents some differences between both service models.

| **Feature** | **DaaS** | **DBaaS** |
|---|---|---|
| Goal | Provide data on-demand | Provide DBMS on-demand |
| Access Mode | API | Specific Drivers |
| Data Model Returned | Semi-structured and unstructured | Semi-structured and structured |
| Authentication | Optional | Required |
| Query Language | Based on REST or WSDL | Relational (SQL) or specific (NoSQL) languages |

*Table 1: Some differences between DaaS and DBaaS models [Zheng et al. 2013, Zheng 2018, Hacigumus et al. 2002, Li et al. 2012]*

The DaaS and DBaaS services aim to provide data and DBMS on-demand, respectively. DaaS obtains the data by an API (Application Programming Interface), and DBaaS receives the data by employing specific drivers. Both returns are in semi-structured data. Authentication is optional in DaaS and mandatory in DBaaS. Finally, the query language depends on the data model.

Several organizations persist their data through DaaS and DBaaS models, such as governments and institutions (public and private). These data need to interact and be

[1] http://dados.gov.br/
[2] https://www.data.gov/
[3] http://w2.cleardb.net/
[4] https://www.elephantsql.com/
[5] https://azure.microsoft.com/services/cosmos-db/

available for access by users/applications uniform and transparently. For instance, when two companies merge and they need to aggregate data from different sources, their information may be heterogeneous (schema, syntax, semantics, data types, data formats, and data constraints). This problem is common due to the lack of standards between applications and data sources (DaaS and DBaaS) [Colomb and Orlowska 1995]. Our approach integrates DaaS and/or DBaaS at a syntactic level from heterogeneous sources for SaaS transparently.

Despite the fact that integration and interoperability have similar objectives (enable the common use of data), in our opinion, they have some relevant differences. Interoperability is the ability of multiple systems to work together, ensuring communication between services and exchanging information effectively and efficiently. Integration describes the mechanism by which multiple systems can communicate. Moreover, integration is one solution to achieves interoperability among systems [Gravina et al. 2017]. Since a single cloud service is unlikely to meet the full requirements of an enterprise [Li et al. 2013], integration and interoperability are essential tasks in modern applications.

In this paper, we present Data Join (DJ) method, a solution for joining data regardless of the service (*e.g.*, DaaS and DBaaS) and model (*e.g.*, relational, key-value). Differently from our related work, the DJ method joins cloud services data regardless of the data model. Our method recognizes relational, document-based NoSQL, NewSQL, and semi-structured models (*e.g.*, JSON, CSV, and XML). Few works provide cloud solutions respecting their sources to avoid inconsistency. Other studies afford integration by the adoption of a data standard format, which is not our case as we let each client (SaaS) and data provider (DaaS) deal with its own format. This transparency minimizes the effort of synchronization and consistency of distributed data in cloud services. Our significant contributions are: (i) a solution to integrate data among different DaaS and DBaaS; (ii) a transparent and unified method to access different DaaS and DBaaS through SQL, NoSQL, and NewSQL queries; (iii) a lightweight formal description of our method based on trees and key-values, and (iv) an in-depth description of how our method performs the different join clauses.

Our approach performs data join between two or more data services. For this, our DJ method: (i) receives the data in different formats; (ii) transforms the data into metadata; and (iii) associates the data collected (by DaaS/DBaaS) with the attributes defined in the query. Rescue systems [Barros et al. 2016], Data Lake systems [Fang 2015], and clinical systems [Jayaratne et al. 2019] have a single repository that stores and integrates data in a way that is significantly different from our approach. We deal with data directly from the sources, avoiding obsolete and inconsistent data treatment.

We incorporated our method into a middleware for DaaS and SaaS (MIDAS), and we performed some experiments (functionality, performance, and overhead). In the first experiment, we performed the same query 100 times in different DaaS and DBaaS, varying the number of instances to 100, 1,000, and 10,000. The second experiment performed three different queries, 100 times each, with a combination of two heterogeneous DaaS and DBaaS, measuring response time. We use (i) one Relational DBaaS, (ii) one NoSQL DBaaS, (iii) one NewSQL DBaaS, and (iv) one DaaS to perform joins among different data sources. Finally, the third experiment performed three different queries, 100 times each, integrating three different datasets: one DaaS and two DBaaS.

The remainder of this paper is organized as follows: Section 2 describes the current version of MIDAS; Section 3 describes the model of our approach; Section 4 presents a proof of concept, incorporating our method into a middleware; Section 5 presents some experiments and results; Section 6 presents related works; and Section 7 summarizes our conclusions and identifies areas for further study.

## 2 The Current MIDAS

The MIDAS with DJ method architecture is depicted in Figure 1. This novel approach is composed of eight components:

– **Query Decomposer** receives a SQL or NoSQL query sent from a SaaS and decomposes it into an array;

– **Query Builder** builds a query based on REST for DaaS (or DBaaS);

– **Dataset Information Storage (DIS)** stores the attribute information about DaaS (or DBaaS);

– **Crawler** keeps DIS up-to-date;

– **Mapping**[6] simulates DBaaS with characteristics of a DaaS and forwards the query to a dataset;

– **Join**[6] receives the data and connects it according to fields/attributes described in join clause (SQL join or NoSQL references clause);

– **Filtering** filters and sorts the data as requested by SaaS; and

– **Formatter** formats the data for the model requested by SaaS.

DJ method increases two new components to MIDAS: *Mapping* and *Join*. The *Mapping* component (i) simulates a DBaaS with characteristics of a DaaS, (ii) identifies the data source of query, and (iii) forwards the query to the respective DaaS or DBaaS. This component provides an access interface that obtains DBaaS data through a process that is similar to obtain DaaS data through REST request. *Mapping* component allows MIDAS to query DBaaS and DaaS similarly. This component identifies the dataset to which the query refers (DaaS or DBaaS) and then it forwards the query to the data service.

The *Join* component is triggered only when a SaaS submits a query with a join clause. Upon receiving the data, this component connects the data according to the fields/attributes described in join clause (SQL join or NoSQL references). As this component runs through all data, its complexity is $O(n^2)$.

Besides these two components, *DIS* component was also modified. The new *DIS* component stores credentials information from data services. We include a set of attributes for each data service: (i) driver: generic interface describing a specific service provider implementation; (ii) host: machine location; (iii) port: port of communication; (iv) database: name of the database; (v) username: username for authentication; (vi) password: user password for authentication; (vii) charset: characters coding on the database; and (viii) from: name of the dataset, *e.g.*, table and document.

Other components (Query Decomposer, Query Builder, Crawler, Filtering, and Formatter) runs similar as previous version [Ribeiro et al. 2019].

[6] New component increased by DJ method into MIDAS

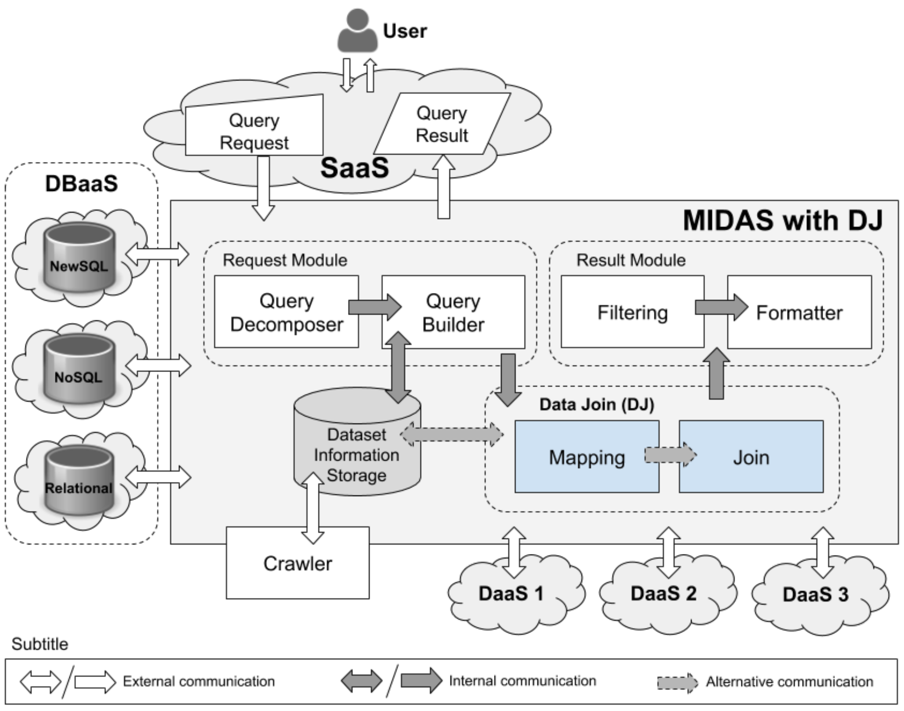


*Figure 1: Overview of MIDAS with our DJ method*

## 3 Model of DJ method

This section describes our method in canonical models. A canonical model [Schreiner et al. 2015] is an universal data model composed of trees and key-values. Although it is a generic model, our work adapted canonical models because they can define and represent different data models, *e.g.*, DaaS and DBaaS.

**Definition 1 (mDJ)** *mDJ (Figure 2) is a tuple $mDJ = (mDictionary, mMapping, mData, mJoin)$, where: $mDictionary$ is the canonical model of DIS, $mMapping$ is the canonical model for identifying queries, $mData$ is the canonical model that maps DBaaS (or DaaS) return(s), and $mJoin$ is the canonical model that maps the join clause.*

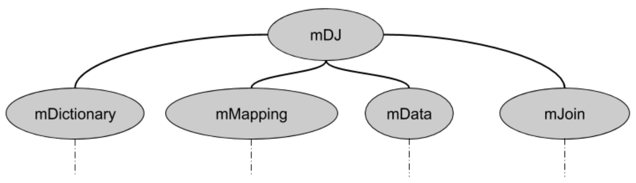


*Figure 2: mDJ canonical model*

**Definition 2 (mDictionary)** *The canonical model of DIS ($mDictionary$) is a tuple $mDictionary = (N_{root}, DSERVICES)$, where: $N_{root}$ is the model identification name, and $DSERVICES$ is the set of data service models (dservice set).*

**Definition 3 (dservice)** *The canonical model that represents the i-th dataset (dservice $\in DSERVICES$) is a tuple dservice $= (N_{root_{dservice}}, KEYS)$, where: $N_{root_{dservice}}$ is the name of the dataset, and $KEYS$ is a predefined set of keys for each dataset, in which $KEYS$ = {domain, search_path, query, filter, sort, limit, dataset, records, fields, amount, format, credentials}. Keys are defined for each dataset. The set $KEYS$ is between 2 and 12 keys (i.e., $2 \leq keys \leq 12$). The service needs to specify at least a $domain$ (virtual address to be located) and a $dataset$ (dataset identifier). In these cases, $KEYS$ contains two keys ($KEYS$ = {domain, dataset}). The remaining options for $keys$ can be either mandatory or optional, depending on the data.*

**Definition 4 (key)** *A $key$ ($key \in KEYS$) consists of dataset information and it is defined as $key = (N_{root_{key}}, INFO)$, where: $N_{root_{key}}$ names a specific attribute of a $dservice$ ($keys.Nroot_{keys} \in KEYS$), and $INFO$ is a set of information ($info$) about the $key$ attribute. Thus, for all $keys$, there is a corresponding $info$.*

For each dataset, the following attributes are defined:

– $domain$: address of a data service;

– $search\_path$: address complement;

– $query$: project data;

– $filter$: filter information;

– $sort$: sort attribute;

– $limit$: amount data returned;

– $dataset$ name of the dataset;

– $records$: required when only data are necessary (no extra information);

– $fields$: attributes of a dataset;

– $amount$: maximum number of data in a source;

– $format$: data format options; and

– $credentials$: set of credentials for authentication.

**Definition 5 (info)** *An information $info$ ($info \in INFO$) is a tuple $info = (N_{root_{info}}, DATA)$, where: $DATA$ is a set of data of the i-th attribute, and $N_{root_{info}}$ defines a specific information or attribute of a key ($info.N_{root_{info}} \in INFO$). Information $info$ can be empty, atomic or multivalued. If $\nexists keys$, then $\nexists info$.*

**Definition 6 (data)** *A $data$ ($data \in DATA$) corresponds to a data point referring to $info$, in which data can be empty ($DATA = \emptyset$) or binary ($DATA = \{dataI_{info}, dataV_{info}\}$), where: $dataI_{info}$ is an information $info$, and $dataV_{info}$ is a value of $info$.*

For instance, $mDictionary$ of two sets of data (*dset1* and *dset2*) is shown in Figure 3: (i) the main node consists of an identifier of the model; (ii) each node in $dservice$ level stores the name of each dataset; and (iii) nodes in $info$ level store information about $key$ level, immediately above. The dataset *dset1* stores information about people in a MongoDB in the domain of the Federal University of Bahia (http://ufba.br). The model presents information about domain, credentials, filters, among other elements.

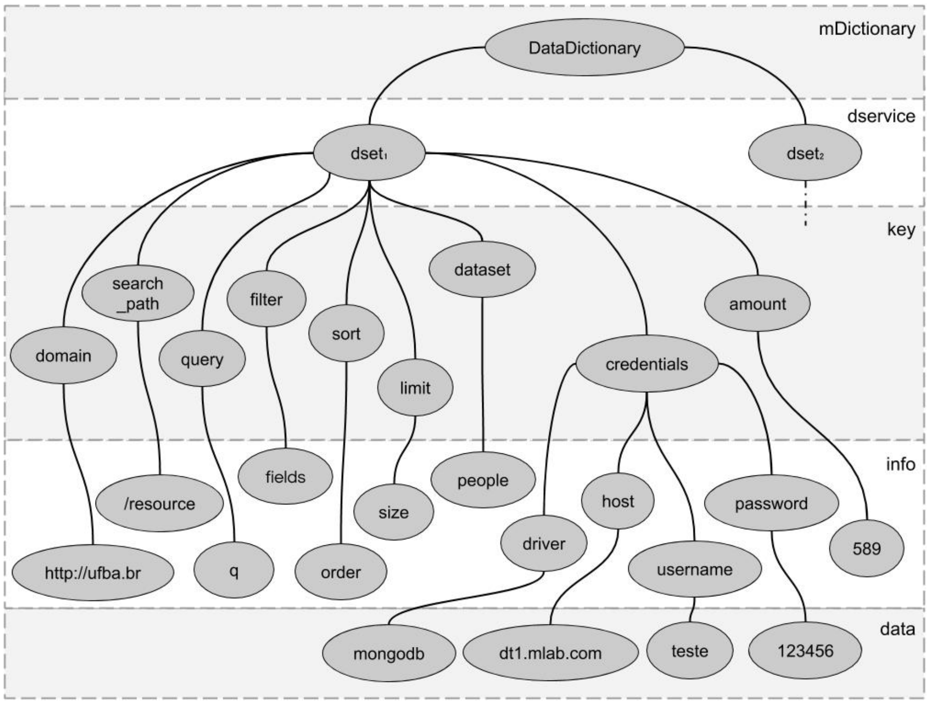


*Figure 3: Example of mDictionary canonical model*

**Definition 7 (mMapping)** *The canonical model $mMapping$ is defined by $mMapping$ = ($N_{root}$, $PARAMS$), where: $N_{root}$ is the identification name of the dataset, and PARAMS is the set of parameters ($param$ set) to map the attributes and query operations to DaaS (or DBaaS). There is one mMapping for each integrate dataset.*

**Definition 8 (param)** *Each parameter ($param \in PARAMS$) is a tuple $param = (N_{root_{param}}, VALUES)$, in which: $N_{root_{param}}$ is the name of the query clause, where $param$ ={$dataset$, $fields$, $where$, $order$, $limit$}, and $VALUES$ is a set of values of each clause.*

**Definition 9 (value)** *A $value$ ($value \in VALUES$) represents information for each clause of the query. Depending on the $param$, $value$ can be empty, atomic or multivalued. Thus, $VALUES = \emptyset$ or $VALUES = \{value_1, value_2, \ldots, value_w\}$, where: $value_i$ is the i-th $value$ and $w$ is the number of $values$ in $VALUES$ set.*

Figure 4 shows an example of $mMapping$. The first level presents the main node with the dataset identifier. The nodes in the $param$ level consist of the query clauses, and the last level nodes ($value$) contain the $param$ level information. Figure 4 shows an example of a possible result for the following SQL query:

```
SELECT name, age, id
FROM dset1
WHERE age > 8
ORDER BY name
LIMIT 10
```

where: $dataset$ is the attribute of $FROM$, $fields$ are the attributes of $SELECT$, $where$ is the attribute of $WHERE$ (where clause conditions), $order$ is the attribute of $ORDER\ BY$, and $limit$ is the attribute of $LIMIT$.

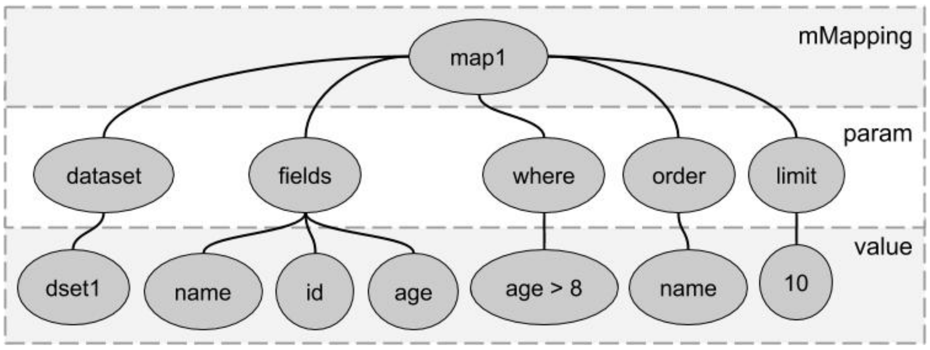


*Figure 4: Example of a mMapping canonical model*

**Definition 10 (mData)** *The $mData$ canonical model is a tuple $mData = (N_{root}, FIELDS)$, where: $N_{root}$ is the identification name of the dataset, and $FIELDS$ is the set of attributes of the returned data ($field$ set).*

**Definition 11 (field)** *Each field ($field \in FIELDS$) is a tuple $field = (N_{root_{field}}, REC)$, where: $N_{root_{field}}$ identifies each field, and $REC$ is a set of data records (rec), in which the $field$ can be either empty or atomic.*

**Definition 12 (rec)** *A record $rec$ ($rec \in REC$) represents all information of a $field$, following the same order as data received (top-down). The number of $rec$ is equal to the number of tuples returned. Thus, $rec = \{tuple_1, tuple_2, \ldots, tuple_q\}$, where: $tuple_i$ is the i-th tuple for each $rec$ of $field$ and $q$ is the amount of tuples returned.*

For instance, Figure 5 presents a $mData$ canonical model: the main node identifies the data source, and subtrees are formed by the field identifier node ($field$) and the data returned ($rec$). This canonical model is an example of data return (Table 2) in response to query: $dataset = ``dset1"$, $fields = \{``name", ``age", ``id"\}$, $where = ``age > 8"$, $order = ``name"$, and $limit = 3$.

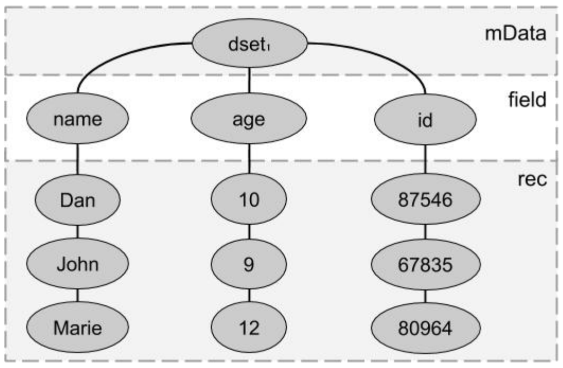


*Figure 5: Example of mData of a query to $dset1$ dataset*

| dset1 | | |
|---|---|---|
| **name** | **age** | **id** |
| Dan | 10 | 87546 |
| John | 9 | 67835 |
| Marie | 12 | 80964 |

*Table 2: Example of returned data*

**Definition 13 (mJoin)** *$mJoin$ is a tuple $mJoin = (N_{root}, JC)$, where: $N_{root}$ is the name that identifies the dataset, and $JC$ is the set of distinct values of the join condition ($jc$ set) in the corresponding relationship.*

**Definition 14 (jc)** *An information $jc$ ($jc \in JC$) consists of a value that join condition admits in the corresponding relation. For such, $jc$ is a tuple $jc = (N_{root_{jc}}, DJ)$, where: $N_{root_{jc}}$ is the name that identifies $jc$ value, and $DJ$ is a set of join data (dj set) with all attributes in $jc$.*

**Definition 15 (dj)** *A data join dj (dj $\in DJ$) contains all the information of the same tuple that $jc$ is a part of, following the relationship's order of occurrence (from left to right). The number of dj $\in DJ$ reflects directly on the number of tuple attributes, thus dj $= \{a_1, a_2, \ldots, a_n\}$, where: $a_i$ is the i-th attribute for each dj, and $n$ is the number of tuples $a \in$ dj.*

As an example, it can be assumed that both sets presented in Table 3 result from the following query ($q$):

```
SELECT dset1.name, dset1.age, dset2.address
FROM dset1 LEFT OUTER JOIN dset2 ON dset1.id = dset2.client_id
LIMIT 5
```

Breaking down $q$, it is assumed that: $q.SELECT =$ *{dset1.name, dset1.age, dset2.address}*; $q.FROM = dset1$; $q.JOIN =$ *{dset1.id = dset2.client_id}*; and $q.LIMIT = 5$. This query results in two $mMapping$s: $dset_1$ and $dset_2$ (Figure 6). In this case, each $mData$ canonical model is shown in Figure 7.

| dset1 | | | dset2 | |
|---|---|---|---|---|
| **name** | **age** | **id** | **address** | **client_id** |
| Dan | 10 | 87546 | Street A | 67835 |
| John | 9 | 67835 | Street C | 87546 |
| Marie | 12 | 80964 | Street D | 87546 |
| ... | ... | ... | ... | ... |
| name$_n$ | age$_n$ | id$_n$ | address$_n$ | client_id$_n$ |

*Table 3: Example of returned data*

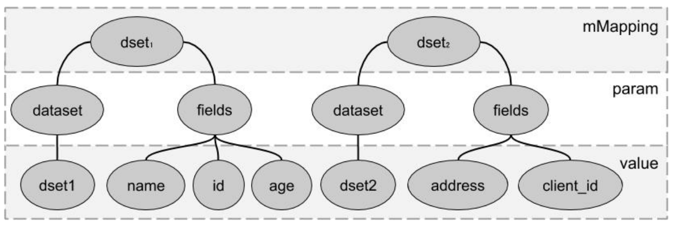


*Figure 6: Example of mMapping with a join clause between two datasets*

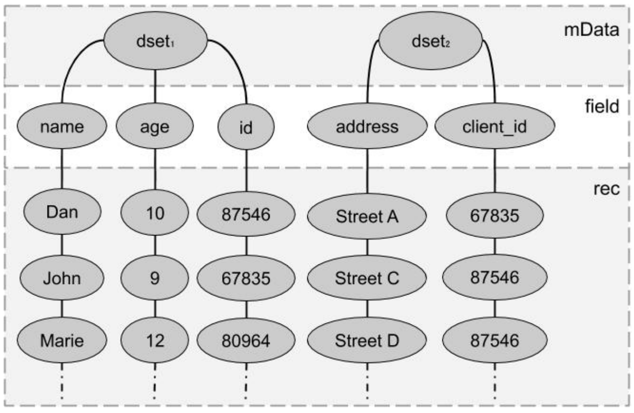


*Figure 7: Example of $mData$ canonical models of Table 3*

When preparing for a join clause, the values in the relationship are retained for later aggregation. The first step is to separate the values from the relationship ($jc$) from the other values (Figure 8(a)). Afterwards, it is assumed that: (i) $lch(p)$ is a function that results in the last child of a $p$ node; (ii) $ch(p)$ is a function that results in the contents of the first child of $p$ node; and (iii) $con(p_1, p_2)$ is a function that connects $p_1$ node to $p_2$ node. Join clauses are performed as follows:

1. For **left join**:
   (a) $\forall jc_1 \in ch(dset_1)$ and $\forall jc_2 \in ch(dset_2)$, then $con(lch(dset_1.jc_1), ch(dset_2.$

$jc_2)$), when $\forall jc_1 = jc_2$ (Figure 8(b));

   (b) $\forall jc_1 \notin q.SELECT$, $con(dset1, ch(dset_1.jc_1))$ and remove $jc_1$ (Figure 8(c)).

2. For **right join**:

   (a) $\forall jc_1 \in ch(dset_1)$ and $\forall jc_2 \in ch(dset_2)$, then $con(lch(dset_2.jc_2), ch(dset_1.jc_1))$, when $\forall jc_1 = jc_2$;

   (b) $\forall jc_2 \notin q.SELECT$, $con(dset_2, ch(dset_2.jc_2))$ and remove $jc_2$.

3. For **inner join**:

   (a) $\forall jc_1 \in ch(dset_1)$, if $jc_1 \notin ch(dset_2)$ then remove $jc_1$;

   (b) $\forall jc_2 \in ch(dset_2)$, if $jc_2 \notin ch(dset_1)$ then remove $jc_2$;

   (c) $\forall jc_1 \in ch(dset_1)$ and $\forall jc_2 \in ch(dset_2)$, then $con(lch(dset_1.jc_1), ch(dset_2.jc_2))$, $\forall jc_1 = jc_2$ and $\forall jc_2 = jc_1$;

   (d) $\forall jc_1 \notin q.SELECT$, $con(dset_1, ch(dset_1.jc_1))$ and remove $jc_1$.

4. For **full join**:

   (a) $\forall jc_1 \in ch(dset_1)$ and $\forall jc_2 \in ch(dset_2)$, $con(dset_1, dset_2.jc_2)$;

   (b) $\forall jc_1 \notin q.SELECT$, $con(dset_1, ch(dset_1.jc_1))$ and remove $jc_1$.

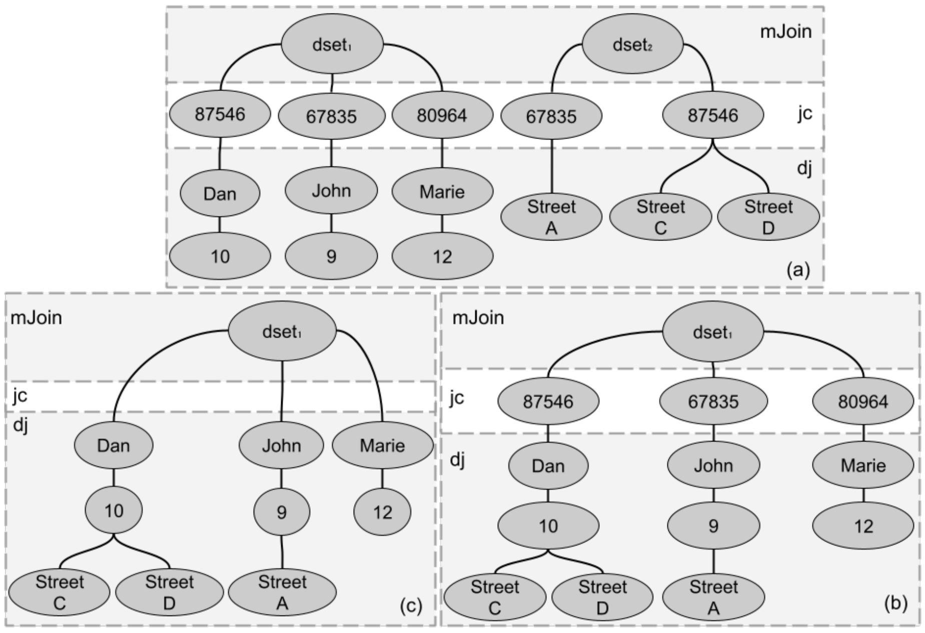
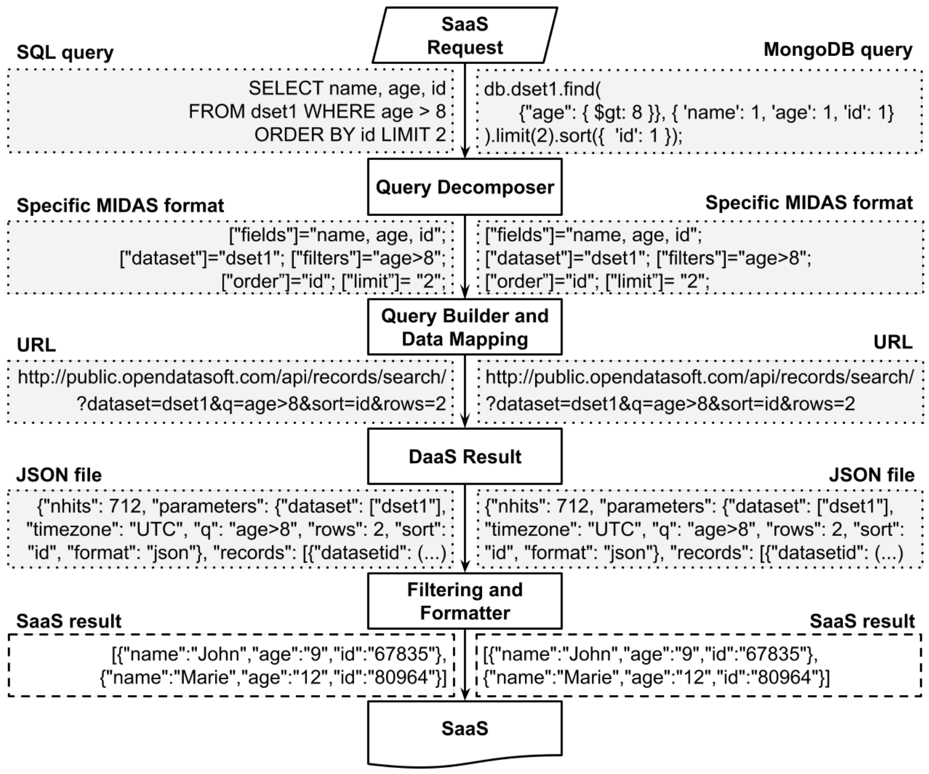


*Figure 8: Example of $mJoin$ steps (left join)*

We are aware that there is no native support for the join clause in NoSQL queries, in contrast to SQL and NewSQL queries. MIDAS loads data from NoSQL data sources, and then the middleware deals internally with the reference clauses involving these data. Our method performs the reference clause without problem for two reasons. First, MIDAS with DJ recognizes queries from document-based NoSQL, for instance, the lookup clause from MongoDB. Second, when MIDAS with DJ receives an SQL or document-based NoSQL query, the Query Decomposer module decomposes the query into a specific MIDAS format, enabling MIDAS to deal with Relational/NewSQL and NoSQL queries. Therefore, both SQL or document-based NoSQL queries are treated equally after Query Decomposer.

Figure 9 shows how MIDAS with DJ translates SQL and NoSQL (MongoDB) queries without a join clause into the specific MIDAS format. MIDAS receives the request from SaaS and then translates the query into a specific format. MIDAS analyzes the query statements and maps the clauses to instances according to the data model. After Query Decomposer, all outputs are identical.



```
SELECT name, age, id
FROM dset1 WHERE age > 8
ORDER BY id LIMIT 2
```

```
db.dset1.find(
    {"age": { $gt: 8 }}, { 'name': 1, 'age': 1, 'id': 1}
).limit(2).sort({  'id': 1 });
```

```
["fields"]="name, age, id";
["dataset"]="dset1"; ["filters"]="age>8";
["order"]="id"; ["limit"]= "2";
```

```
["fields"]="name, age, id";
["dataset"]="dset1"; ["filters"]="age>8";
["order"]="id"; ["limit"]= "2";
```

```
http://public.opendatasoft.com/api/records/search/
?dataset=dset1&q=age>8&sort=id&rows=2
```

```
http://public.opendatasoft.com/api/records/search/
?dataset=dset1&q=age>8&sort=id&rows=2
```

```
{"nhits": 712, "parameters": {"dataset": ["dset1"],
"timezone": "UTC", "q": "age>8", "rows": 2, "sort":
"id", "format": "json"}, "records": [{"datasetid": (...)
```

```
{"nhits": 712, "parameters": {"dataset": ["dset1"],
"timezone": "UTC", "q": "age>8", "rows": 2, "sort":
"id", "format": "json"}, "records": [{"datasetid": (...)
```

```
[{"name":"John","age":"9","id":"67835"},
{"name":"Marie","age":"12","id":"80964"}]
```

```
[{"name":"John","age":"9","id":"67835"},
{"name":"Marie","age":"12","id":"80964"}]
```

*Figure 9: Steps of MIDAS*

We emphasize that DaaS and DBaaS are accessed from a URL according to the service API [Ribeiro et al. 2019]. MIDAS translates SQL or NoSQL query into one or more URLs, depending on the existence of a join clause.

## 4 Proof of Concept

As a proof of concept, our method was integrated into a middleware to interoperate SaaS and heterogeneous DaaS/DBaaS. We incorporate our DJ method into MIDAS since we have access to the middleware source code and this allows us to improve and evaluate features using the existing framework.

Figure 10 presents the execution flow of MIDAS with our DJ method. The process starts with a SQL or NoSQL query by the *Query Decomposer*. The query is decomposed into an array that identifies the projection, data set, joins, filters, sort, and limit. *Query Builder* receives this array and it accesses *DIS* to obtain the information to construct queries. *Mapping* selects the service (DaaS or DBaaS) and checks for authentication. If the requested service is a DBaaS, *Mapping* recognizes the query parameters and converts them into an API of the specific databases (Relational, NoSQL, or NewSQL). Services are then accessed to obtain the data sets. Finally, *Filtering* component filters and sorts the data, and *Formatter* component formats and sends these data to SaaS.

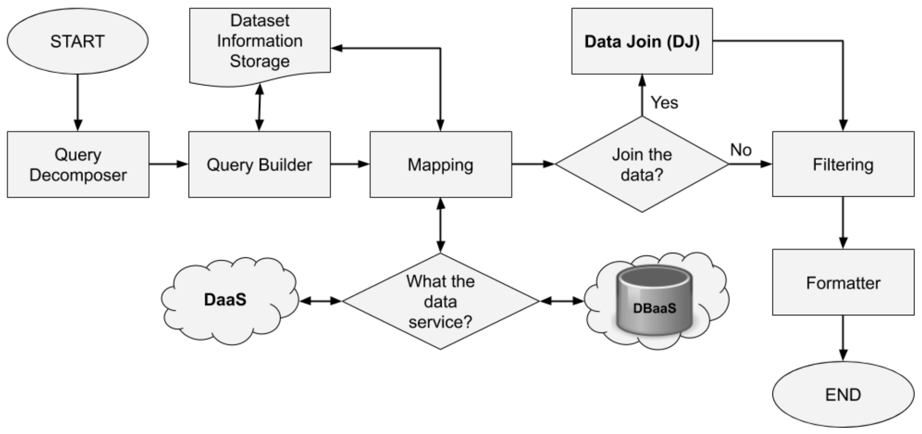


*Figure 10: Execution flow of MIDAS with our DJ method*

An example of the process executed by MIDAS with DJ method starts with SaaS sending a query as follows:

```
SELECT people.name, people.age,
    places.street, places.number, places.city
FROM people LEFT JOIN places
    ON people.id = places.person_id
WHERE people.city = "New York"
ORDER BY people.age
LIMIT 3
```

*Query Decomposer* groups the query parts as shown in Figure 11. *Query Builder* component checks the information about the dataset in data dictionary (*DIS*). This information is stored in a JSON file, as shown in Figure 12.

```
array( {
        "select" =>  'people.name, people.age,places.street, places.number, places.city',
        "from" => 'people',
        "join" => array( {
                "type" => 'LEFT JOIN', "dataset1" => 'people',
                "dataset2" => 'places', "att1" => 'id', "att2" => 'person_id'
        } ),
        "where" => 'people.city = "New York"', "order" => 'people.age',     "limit" => 3
} )
```

*Figure 11: Query Decomposer output example*

```
"people": {
        "domain": "http://dpe.com", "search_path": "/resource",
        "query": "$where", "filter": "$select", "sort": "$order",
        "limit": "$limit", "dataset": "people", "amount": "500"
}, "places": {
        "domain": "http://127.0.0.1", "query": "q",  "filter": "fields",
        "sort": "order", "limit": "l", "dataset": "dataset", "amount": "3850",
        "credentials": {
                "driver" => "mongodb", "host" => "dt1.mlab.com",
                "port" => "19678", "database" => "db1", "username" => "teste",
                "password" => "123456", "collection" => "places"
        }
}
```

*Figure 12: DIS example*

As a way of validating *mDictionary* canonical model, Figure 13 describes the information displayed in *DIS*.

*Query Builder* builds the request from the query. Since the information is from two datasets, results are obtained through two distinct data requests. Results of Query Builder are:

i) *http://dpe.com/resource/?people&$select=name,age,id&$order=age*

ii) *http://127.0.0.1/?dataset=places&fields=street,number,city,id_people&q=city= "New York"*.

After constructing the requests, they are forwarded to *Mapping*. This component identifies the dataset from the URL (Uniform Resource Locator) parameter. When a URL is input with the domain *http://127.0.0.1* (*e.g.*, *http://127.0.0.1/ ?dataset=places...*), *Mapping* recognizes a DBaaS. When *Mapping* identifies a DaaS, the request is sent directly. When it identifies a DBaaS, the URL is fragmented into parameters and transformed into a query, which is sent to different DBaaS models. The function parameters are described in the *mMapping* canonical model presented in Figure 14.

Before sending the query to DBaaS, *Mapping* collects information about access credentials which were forwarded by *Query Builder*. These credentials are used to authenticate access to the database.

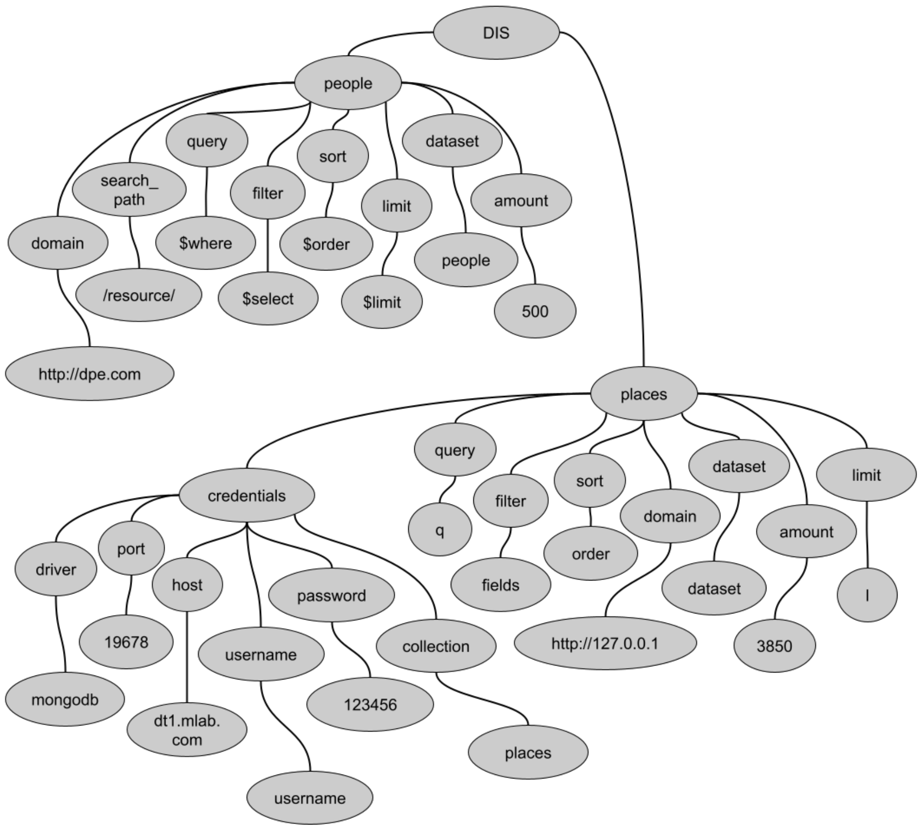


*Figure 13: Example of mDictionary canonical model for Figure 12*

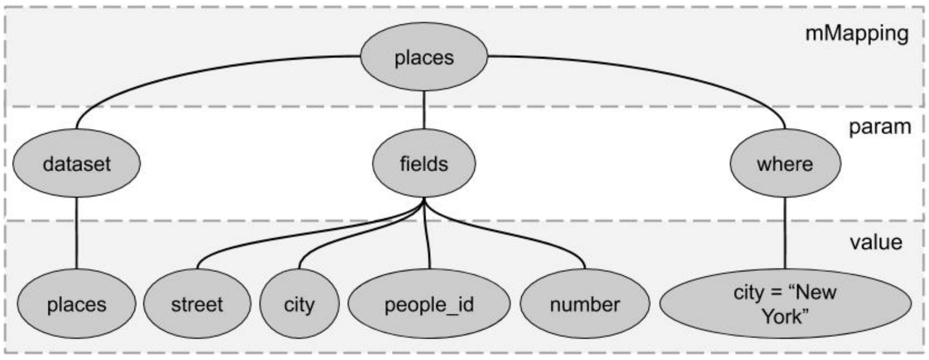


*Figure 14: mMapping model describing the parameters in Mapping*

The last step of *Mapping* is to transform the decomposed parameters from the URL into a query. The following example is about a NoSQL DBaaS based on MongoDB, using the $driver$ credential $mongodb$. The result is:

```
db.places.find(
    {
        "city", "New York"
    },
    {
        "street": 1,
        "city": 1,
        "number": 1,
        "people_id": 1
    }
)
```

in which the emphasized words are equivalent to parameters in standard function of *Mapping* as described in Figure 14.

At this stage, queries are sent to DaaS and DBaaS. The two mData results are depicted in Figure 15. This representation omits the heterogeneities of data models that our approach eliminates.

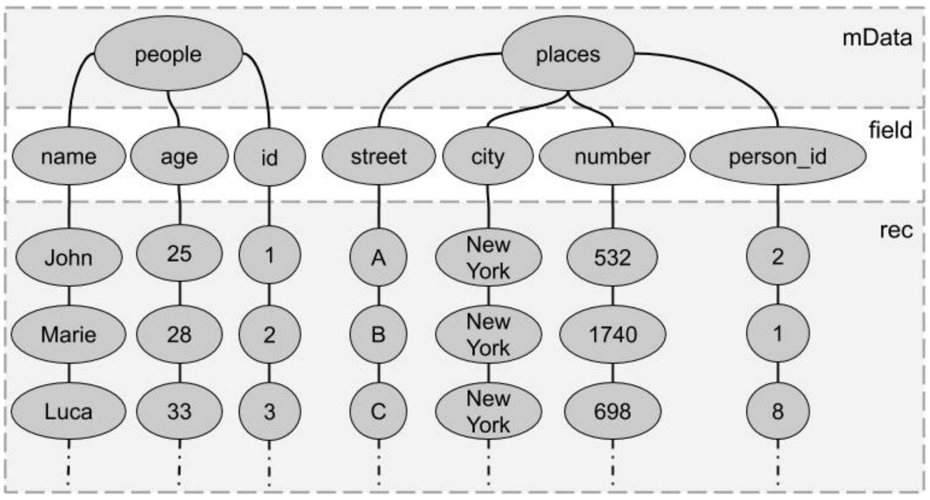


*Figure 15: Example of two mData: people and place*

Since SaaS sent a query with the join clause, *Join* component is trigger. This component then checks which join clause to perform. Results are described by mJoin in Figure 16.

Finally, *Filtering* and *Formatter* components remove the unusual data and format the result as requested by SaaS. Formats currently available by MIDAS are JSON, CSV, and XML.

The next section describes a set of experiments to evaluate our approach.

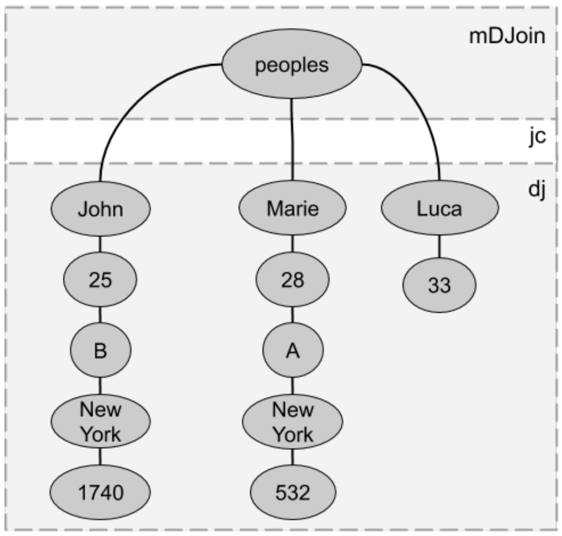


*Figure 16: mJoin result from mData (Figure 15)*

## 5 Evaluation

We performed three experiments to evaluate our method. These experiments aim to validate the (i) overhead caused by DJ method and (ii) performance of data join. The first experiment compares queries with and without a join clause. Results measure the response time between SaaS and data services. In the second experiment, we perform a query with only one join clause to integrate two different data services. The third experiment compares queries with more than one join clause to integrate three different data services.

The first experiment aims to analyze the overhead of MIDAS with DJ when accessing DaaS/DBaaS. In this experiment, we performed a query directly to the data services and then we compared these results with the query performs through MIDAS with D method. The same query was sent: (i) 100 times directly to DaaS/DBaaS, and (ii) 100 times to DaaS or DBaaS through MIDAS with our DJ method. Queries returned 100, 1,000, and 10,000 records.

The second experiment evaluates the overhead with a join clause between two datasets. For this, we performed 100 queries on (i) two DaaS, (ii) two DBaaS, and (iii) one DaaS and one DBaaS. Returned data was not limited.

In the third experiment, we performed the join clauses on three datasets: (i) one DaaS, one Relational DBaaS, and one NoSQL DBaaS; (ii) one DaaS, one NoSQL DBaaS, and one NewSQL DBaaS; (iii) one DaaS, one Relational DBaaS, and one NewSQL DBaaS; and (iv) one Relational DBaaS, one NewSQL DBaaS, and one NoSQL DBaaS.

Although MIDAS recognizes NoSQL query languages from different formats due to the Query Decompose module's generalization, we experimented with document-based NoSQL. From such type of NoSQLs, we chose the most employed document-based NoSQL[7], which is MongoDB.

[7] According to https://db-engines.com/en/ranking, last accessed: February 23, 2021

We performed all tasks with native functions. The execution time measure was restricted to the overload of our algorithm. No external tools were used to avoid any interference in our evaluation process.

## 5.1 Our Case Study

We developed MIDAS with DJ method using open source technologies in Heroku Cloud Platform[8]. The Cloud Application Platform of Heroku consists of a complete development environment.

Our SaaS instance was developed through a web application that queries MIDAS with DJ method. Our cloud instance has one CPU core, 512MB of RAM, and limited storage. This application is hosted in the Heroku Cloud and it can be accessed by *https://midastests.herokuapp.com/*. This SaaS performs queries with and without join clause applied to different DaaS and DBaaS models.

## 5.2 Experiments and Results

Datasets used in the experiments are provided by three different DaaS providers (NYC Open Data[9], DATA.NY.GOV[10], and OpenDataSoft[11]):

- $D_1$: *Transportation Sites*, with 13,7 thousand instances and 18 attributes;
- $D_2$: *Health and Hospitals Corporation (HHC) Facilities*, with 78 instances e 6 attributes; and
- $D_3$: *NYC Wi-Fi Hotspot Locations*, with 3,179 instances and 29 attributes.

We choose these datasets because they have an intersection attribute that contains related information: *zip code* attribute.

With respect to DBaaS, the same datasets $D_1$, $D_2$, and $D_3$ were persisted in three different DBaaS providers:

- $DB_1$: mLab[12] based on MongoDB (NoSQL);
- $DB_2$: ClearDB[13] based on MySQL (relational); and
- $DB_3$: Hetzner[14] based on MemSQL (NewSQL).

### 5.2.1 Experiment 1

Our first experiment analyzed the overhead through a single DaaS and DBaaS. We performed a query 100 times, successively, varying the number of results (100, 1,000, and 10,000) for each source. The datasets were $DB_1$, $DB_2$ and $DB_3$. The average response time (in seconds) is shown in Figure 17.

[8] https://www.heroku.com/
[9] https://goo.gl/mVLdDh
[10] https://goo.gl/Au5GtQ
[11] https://goo.gl/gC4wnz
[12] https://www.mlab.com/
[13] http://w2.cleardb.net
[14] https://hetzner.com/

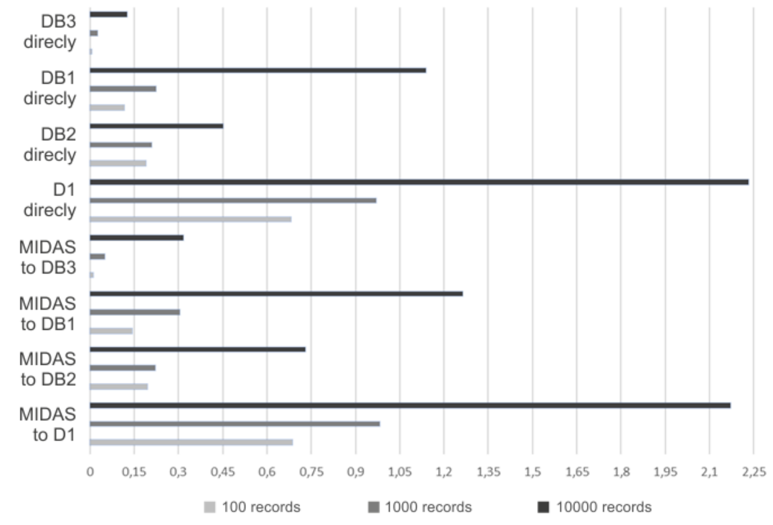


*Figure 17: Average response time (X-axis) for each DaaS or DBaaS (Y-axis)*

Figure 17 depicts how the number of returned data impacts the response time. In some cases, the time difference for 100 and 1,000 instances is minimal. On the other hand, when increasing from 1,000 to 10,000 instances, the response time increased more than double. When analyzing the MIDAS with DJ method overhead, we obtained the following results: (i) $DB_2$ was 23.34% faster without MIDAS; (ii) $DB_1$ was 23.93%; and (iii) $DB_3$ was 102.79%. In the $D_1$ test no overhead occurred, since the middleware was 0.29% faster. This occurs because communication between our cloud and DaaS provider has high latency and this reflects directly on the response time.

We compared ZQL [Xu et al. 2016] with our results because this is the only with similar experiments. Although their computational power is more significant than provided by Heroku in our experiments, our performance is better. For instance, their response time for 10,000 instances of RDBMS in MySQL is about 2 seconds, while our method is about 0.72 seconds. We do not consider the number of dataset attributes since ZQL does not provide this information. Additionally, our DJ method addresses with data formatting, in contrast to ZQL.

### 5.2.2 Experiment 2

This experiment evaluates overhead when integrating two datasets. Response time was estimated by performing data join between two different DaaS and DBaaS models in distinct clouds. Data sources was integrated based on *zip code* attribute, in which it is represented in $D_1$ and $D_2$ by the attribute *zip* and *zip_code*, respectively. For this, we perform:

- 100 queries with join clause for two DaaS providers ($D_1$ e $D_2$);
- 100 queries with join clause for two DBaaS providers ($DB_1$ e $DB_2$); and

– 100 queries with join clause for one DaaS provider ($D_1$) and one DBaaS provider ($DB_1$).

Figure 18 shows the response time of each query. Each query on average was:

– $36.87 \pm 6.66$s for queries on two DaaS;

– $27.22 \pm 13.97$s for queries on two DBaaS; and

– $32.84 \pm 18.13$s for queries on one DaaS and one DBaaS.

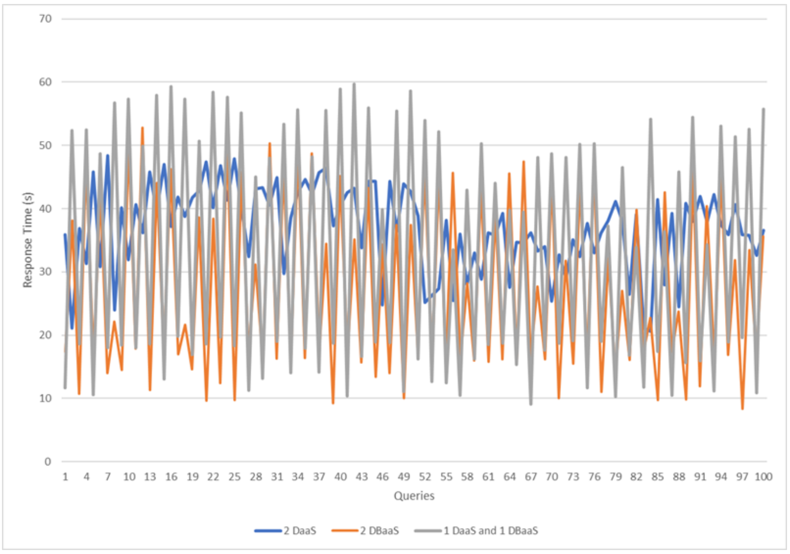


*Figure 18: Response time (Y-axis) for each query (X-axis) on cloud*

Some of our results have a high standard deviation. We observed that this was due to DBaaS access. Conversely, we realize that the shortest average response time is associated with DBaaS. Analyzing step-by-step, we noticed that the communication between MIDAS with DJ and data service is responsible for this abnormal performance. We also observed that the time for DBaaS to return to a request is higher than the response for the previous requests. This unexpected response time may have occurred due to service overload or network latency between MIDAS with DJ and the service. We performed the queries locally with the same data sets to avoid external interference, such as network latency. The results are shown in Figure 19.

The experiment shows that in local queries, the standard deviation and the average response time are smaller compared to the cloud queries. These results demonstrates the impact of external communication among services caused by network latency or overhead of data services.

### 5.2.3 Experiment 3

Our third experiment evaluates integration and overhead among more than two data services. We presented the response time to integrate data on three different DaaS and

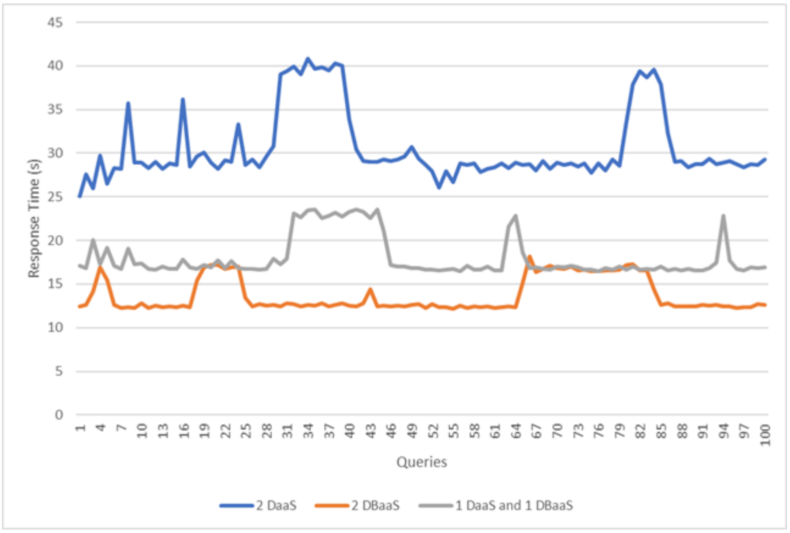


*Figure 19: Response time (Y-axis) for each query (X-axis) locally*

DBaaS from heterogeneous clouds. Queries contain both inner and left join clauses. Data was integrated through the *zip code* attribute, represented in $D_1$, $D_2$ and $D_3$ by the attribute *zip*, *zip_code*, and *postal_code*, respectively. For this, we performed:

- 100 queries with join clause for $D_1$, $DB_2$, and $DB_1$;
- 100 queries with join clause for $D_1$, $DB_1$, and $DB_3$;
- 100 queries with join clause for $D_1$, $DB_2$, and $DB_3$; and
- 100 queries with join clause for $DB_1$, $DB_2$, and $DB_3$.

Figure 20 depicts the response time of each query. Each mean is described as follows:

- $44.57 \pm 8.19$s for $D_1$, $DB_2$, and $DB_1$;
- $52.02 \pm 8.43$s for $D_1$, $DB_1$, and $DB_3$;
- $42.34 \pm 8.99$s for $D_1$, $DB_2$, and $DB_3$; and
- $35.19 \pm 6.27$s for $DB_1$, $DB_2$, and $DB_3$.

Results show that integrating DaaS has a longer response time than integrating DBaaS. This fact is expected because DBaaS models are based on DBMS models, while DaaS is a service without any prior recommendations nor rigorous study. The number of datasets and time are directly proportional. However, queries with $DB_1$, $DB_2$, and $DB_3$ have time very similar to experiment 2.

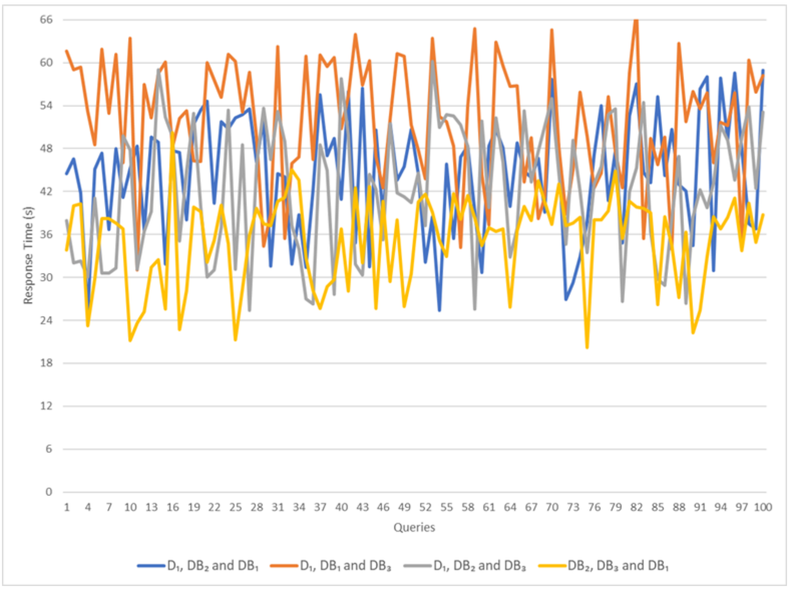


*Figure 20: Response time (Y-axis) for each query (X-axis)*

### 5.3 Methodological concerns

Possible concerns regarding our methodology:

- only *left*, *right*, *inner* and *full outer join* are recognized in our DJ method;
- *lookup* clause (MongoDB queries) recognized in our DJ method allows only basic functions;
- the computational effort of the MIDAS with DJ and increased data are directly proportional;
- all DaaS and DBaaS must be included in DIS;
- in case of the failure of DaaS or DBaaS during the query, MIDAS with DJ method may return an error or not return data;
- a join clause is performed only when there is an equality condition between two attributes; and
- SaaS application needs a brief understanding of data attributes before perform a query.

The next section positions our paper with respect to our related works.

## 6 Related Work

Although there is much literature in the area of data join, specific work on cloud environments has not been thoroughly provided, as far as we know. Most of the recent related works deal with problems in other areas (*e.g.*, medical area). Despite the complexity involving integration, some authors have proposed solutions similar to our approach.

[Jayaratne et al. 2019] propose an open data integration platform for patient, clinical, medical, and historical data persisted into multiple health information systems. Their focus is on patient-centered care to provide heterogeneous medical information. However, they need to synchronize their data, and they only use a relational database to persist them, which differs from our approach.

[Park and Moon 2015] propose a solution for heterogeneous DBaaS that shares medical data from different institutions. Unlike our approach, their data follows the Health Level Seven (HL7) standard, minimizing efforts related to heterogeneity. Moreover, the authors do not present or discuss data join.

[Barros et al. 2016] present an integration environment to solve emergency events, called RESCUE. The objective is to integrate data from heterogeneous sources through ontologies. Differently from our approach, RESCUE stores data from multiple sources into a single repository to integrate them. We use data pulled directly from the sources, without storing them in a temporary repository. This minimizes the risk of inconsistency.

Both studies in [Sellami et al. 2014] and [Xu et al. 2016] provide an API and a middleware, respectively, to manage different relational and NoSQL databases. These authors do not perform data join as we do in this study. [Ma et al. 2017] proposed to extract data from heterogeneous sources, integrate them, and transform them into graph instance data. Even though they integrate data automatically into a graph database, different from our approach, we let the data in their sources, minimizing inconsistencies and synchronization tasks.

Solutions for data interoperability using multiple systems are somewhat Data Lake [Fang 2015]. This concept consists of a single repository that stores data sets in different formats. Data Lake-based systems require both high storage space and require substantial efforts to integrate and refine data. [Kadadi et al. 2014] present other challenges to providing data integration and interoperability. However, authors do not mention Data Lake concern. Contrarily to Data Lake, our approach accesses data directly from the source.

Considering that the whole data integration process can be very complex and consists of three main tasks, Schema Matching, Entity Resolution, and Data Fusion [Dong and Srivastava 2015], results are combined to produce an integrated view of distributed data. Each of these tasks has been the subject of various studies in last years and several solutions have been proposed. However, given the recent advances in how data is published and the broadly use of cloud environments, new solutions for data integration and interoperability systems are still needed.

The most similar work is summarized by [Ribeiro et al. 2019]. The authors achieve interoperability between SaaS and DaaS/DBaaS through middleware MIDAS 1.9. Unlike our work, MIDAS 1.9 is limited to two DaaS and/or DBaaS. We improve their work by expanding the type of join clause and integrating more than two datasets. Due to our DJ method, from now on MIDAS (i) recognizes structured and semi-structured data, (ii) supports DaaS and DBaaS, (iii) integrates data with more than two datasets, (iv) supports different join clause, *e.g.*, right, left, inner and full, (v) is independent of domain; and, (vi) avoids inconsistent data.

Table 4 presents the main differences between the related work and our approach. We employed eight comparison criteria:

(a) **Recognizes structured data**: ability to collect and manipulate structured data, *e.g.*, tables;

(b) **Recognizes semi-structured data**: ability to collect and manipulate semi-structured data, *e.g.*, JSON and CSV;

(c) **DaaS support**: ability to access and collect data from DaaS;

(d) **DBaaS support**: ability to access and collect data from DBaaS;

(e) **Data join**: ability to join data of different formats;

(f) **Joins more than two datasets**: ability to join data from more than two datasets in a single query;

(g) **No specific domain**: solution does not mention a specific domain, *e.g.*, health field; and

(h) **Types of joins**: ability to perform different types of joins, e.g. left, full join, among others.

| **Related Work / Criteria** | (a) | (b) | (c) | (d) | (e) | (f) | (g) | (h) |
|---|---|---|---|---|---|---|---|---|
| [Jayaratne et al. 2019] | X | | | | X | X | | |
| [Park and Moon 2015] | X | | | X | | | | |
| [Barros et al. 2016] | X | | | X | X | ** | | |
| [Sellami et al. 2014] | X | X | | X | | | X | |
| [Xu et al. 2016] | X | X | | X | | | X | |
| [Fang 2015] | X | X | | X | X* | X* | X | |
| [Ribeiro et al. 2019] | | X | X | X | X | | X | |
| **Our DJ method** | X | X | X | X | X | X | X | X |

* Requires manual effort to develop data integration

** Not specified

*Table 4: Major differences between related work and DJ method*

Contrary to our related work, the DJ method performs different types of joins to integrate cloud data services, regardless of the data model and the number of data sources. Our solution recognizes structured (*e.g.*, table) and semi-structured (*e.g.*, JSON, CSV, and XML) data, DaaS, and DBaaS. Additionally, the DJ method supports relational, NewSQL, and document-based NoSQL models. Our method is domain independent.

## 7 Conclusions and Future Work

Our method integrates data from heterogeneous sources through a merge between DaaS and DBaaS models. The DJ method accesses, obtains, and integrates data directly from the source. By collecting data at runtime, our method avoids the use of extra computational resources for storage and prevents obsolete/inconsistent data collection.

To complete our research, we describe our approach to canonical models. Each canonical model describes a (part of) MIDAS component. We considered four join clauses: left, right, inner, and full join. As proof of concept, we integrated our DJ method into MIDAS middleware. Our evaluation consisted of three experiments with and without data join.

The results show that our DJ method into MIDAS middleware requires minimal knowledge to collect and integrate data between different DaaS and DBaaS. Therefore, our approach for data join advances the state-of-the-art with regards to interoperability.

In the future, we intend to continue adding and improving features. In particular, we aim to (i) optimize our algorithm and (ii) incorporate semantics in the mapping and integrating components.

**Acknowledgements**

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

## References

[Armbrust at al. 2010] Armbrust, M., Fox, A., Griffith, R., Joseph, A., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., Zaharia, M.: "A View of Cloud Computing"; Commun. ACM, 4, 53 (2010), 50-58.

[Barros et al. 2016] Barros, R., Vieira, V., Salvador, L., Almeida, R.: "Interoperabilidade Semântica entre Sistemas de Resposta a Emergências"; Proc. $31^{rd}$ Simpósio Brasileiro de Banco de Dados (SBBD), SBC, Salvador (2016), 199-204.

[Barros et al. 2018] Barros, V., Estrella, J., Prates, L., Bruschi, L.: "An IoT-DaaS Approach for the Interoperability of Heterogeneous Sensor Data Sources"; Proc. $21^{st}$ ACM International Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems (ACM MSWiM), ACM, Montreal (2018), 275-279.

[Colomb and Orlowska 1995] Colomb, R., Orlowska, M.: "Interoperability in information systems"; Information Systems Journal, 1, 5 (1995), 37-50.

[Dong and Srivastava 2015] Dong, X., Srivastava, D.: "Big Data Integration"; Morgan & Claypool Publishers, Brisbane (2015).

[Fang 2015] Fang, H.: "Managing data lakes in big data era: What's a data lake and why has it became popular in data management ecosystem"; Proc. $5^{th}$ IEEE International Conference on Cyber Technology in Automation, Control, and Intelligent Systems (CYBER), IEEE, Shenyang (2015), 820-824.

[Gantz and Reinsel 2002] Gantz, J., Reinsel, D.: "The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east"; IDC iView: IDC Analyze the future (2002), 1-16.

[Gravina et al. 2017] Gravina, R., Palau, C., Manso, M., Liotta, A., Fortino, G.: "Integration, Interconnection, and Interoperability of IoT Systems"; Springer, Heidelberg (2017).

[Hacigumus et al. 2002] Hacigumus, H., Iyer, B., Mehrotra, S.: "Providing database as a Service"; Proc. $18^{th}$ International Conference on Data Engineering (ICDE), IEEE, San Jose (2002), 29-38.

[Internet Live Stats 2020] Internet Live Stats: Internet traffic today (2020), https://www.internetlivestats.com

[Jayaratne et al. 2019] Jayaratne, M., Nallaperuma, D., Silva, D., Alahakoon, D., Devitt, B., Webster, K., Chilamkurti, N.: "A data integration platform for patient-centered e-healthcare and clinical decision support"; Future Generation Computer Systems, 1, 92 (2019), 996-1008.

[Kadadi et al. 2014] Kadadi, A., Agrawal, R., Nyamful, C., Atiq, R.: "Challenges of data integration and interoperability in big data"; Proc. 2$^{nd}$ IEEE International Conference on Big Data (BigData), IEEE, Washington (2014), 38-40.

[Li et al. 2012] Li, S., Xu, L., Wang, X., Wang, J.: "Integration of hybrid wireless networks in cloud services oriented enterprise information systems"; Enterprise Information Systems, 2, 6 (2012), 165-187.

[Li et al. 2013] Li, Q., Wang, Z.-Y., Li, W.-H., Li, J., Wang, C., Du, R.-Y.: "Applications integration in a hybrid cloud computing environment: modelling and platform"; Enterprise Information Systems, 3, 7 (2013), 237-271.

[Loutas et al. 2011] Loutas, N., Kamateri, E., Bosi, F., Tarabanis, K.: "Cloud Computing Interoperability: The State of Play"; Proc. 3$^{rd}$ IEEE International Conference on Cloud Computing Technology and Science (CLOUDCOM), IEEE, Athens (2011), 752-757.

[Ma et al. 2017] Ma, B., Jiang, T., Zhou, X., Zhao, F., Yang, Y.: "A Novel Data Integration Framework Based on Unified Concept Model"; IEEE Access, 1, 5 (2017), 5713-5722.

[Mell and Grance 2011] Mell, P., Grance, T.: "The NIST Definition of Cloud Computing"; SP 800-145, Gaithersburg (2011).

[Park and Moon 2015] Park, H.-K., Moon, S.-J.: "DBaaS Using HL7 Based on XMDR-DAI for Medical Information Sharing in Cloud"; International Journal of Multimedia and Ubiquitous Engineering, 9, 10 (2015), 111-120.

[Reinsel et al. 2018] Reinsel, D., Gantz, J., Rydning, J.: "The Digitization of the World from Edge to Core"; IDC White Paper (2018), 1-28.

[Ribeiro et al. 2018] Ribeiro, E., Vieira, M., Claro, D., Silva, N.: "Transparent Interoperability Middleware between Data and Service Cloud Layers"; Proc. 8$^{th}$ International Conference on Cloud Computing and Services Science (CLOSER), SCITEPRESS, Funchal (2018), 148-157.

[Ribeiro et al. 2019] Ribeiro, E., Vieira, M., Claro, D., Silva, N.: "Interoperability Between SaaS and Data Layers: Enhancing the MIDAS Middleware"; in Muñoz, V., Ferguson, D., Helfert, M., Pahl, C. (eds.), chapter 6, Springer, Cham (2019), 102-125.

[Schreiner et al. 2015] Schreiner, G., Duarte, D., Mello, R.: "SQLtoKeyNoSQL: a layer for relational to key-based NoSQL database mapping"; Proc. 17$^{th}$ International Conference on Information Integration and Web-based Applications & Services (iiWAS2019), ACM, Brussels (2015).

[Sellami et al. 2014] Sellami, R., Bhiri, S., Defude, B.: "ODBAPI: a unified REST API for relational and NoSQL data stores"; Proc. 2$^{nd}$ IEEE International Congress on Big Data (BigData), Control, and Intelligent Systems (CYBER), IEEE, Anchorage (2014), 653-660.

[Xu et al. 2016] Xu, J., Shi, M., Chen, C., Zhang, Z., Fu, J., Liu, C.: "ZQL: A unified middleware bridging both relational and NoSQL databases"; Proc. 14$^{th}$ IEEE Intl. Conf. on Dependable, Autonomic and Secure Computing, 14$^{th}$ Intl. Conf. on Pervasive Intelligence and Computing, 2$^{nd}$ Intl. Conf. on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech), IEEE, Auckland (2016), 730-737.

[Zheng et al. 2013] Zheng, Z., Zhu, J., Lyu, M.: "Service-Generated Big Data and Big Data-as-a-Service: An Overview"; Proc. 1$^{st}$ IEEE International Congress on Big Data (BigData), IEEE, Santa Clara (2013), 403-410.

[Zheng 2018] Zheng, X.: "Database as a Service - Current Issues and Its Future"; CoRR, 4, 8 (2018), 1-4.