Models and Architecture for Smart Data Management

Pierre De Vettor, Michaël Mrissa and Djamal Benslimane

Université de Lyon, CNRS
LIRIS, UMR5205, F-69622, France
E-mail: firstname.surname@liris.cnrs.fr

Abstract—Organizations often hold large amounts of unused data, trapped in fragmented databases, locked in legacy data formats. As well, the Web offers a variety of data sources accessible in diverse ways. There is a lack of approaches to handle this multiplicity of data sources and combine multi-origin data into coherent smart data sets. We therefore define a meta-model that allows flexible modeling of data source diversity, and we propose a resource-oriented approach to handle data access and processing. We designed and evaluated our approach to offer scalability, responsiveness, as well as dynamic and transparent data source management features. We motivate our solution through a live scenario based on the information system of the Audience Labs French company. This paper describes our models and resource-oriented architecture and shows how they adapt to industrial needs and facilitate smart data production and reuse.

Index Terms—resource oriented architecture, data integration, data semantics, smart data

1 Introduction

With the emerging presence of social media and social networking systems, users adapted their online *way of life*, becoming both data providers and consumers. They comment, review, gather interests and wish lists, producing a huge amount of data across the Web, collected as custom profiles. Furthermore, governments and large companies open their databases to the world across the Web, thanks to initiatives such as the open data project [1]. These data sources are typically exposed via Web APIs [2] or SPARQL endpoints that can be combined in service mashups [3] to produce highly valuable services. As an example, the sets of APIs provided by Twitter, Facebook, LinkedIn, Amazon, Youtube, Flickr or Dropbox are reused in thousands of mashups[1]. The huge amount of public data allows organizations and companies to adopt new data-driven strategies. It leads them to open and improve their information systems, drawing benefits from the aggregation of data from the Web. Specific business mashups enrich data with semantics and combines external with internal data sets to improve data exploitation, providing advanced statistics, and useful data for decision support systems.

In this paper, we propose an adaptive solution for combining multi-origin data sets available from internal information systems and Web sources in order to produce smart data sets. We define *smart data* as significant, semantically explicit data, ready to be used to fulfil the stakeholders' objectives.

1. See also http://www.programmableweb.com/

1.1 Objectives and Scientific locks

Our objective is to generate an homogeneous linked data set from diverse data sources in response to a given user request. In addition, our proposed solution must present scalability and responsiveness features Our approach relies on a generic metamodel and a set of models focused on data source description, access and processing. Each data source is described with a combination of specific characteristics that allows to improve data access. Our meta-model allows describing data sources in a flexible way and generating models that in turn provide adapted data processing depending on the scenario. The provided scenario demonstrates how the different models (data source model, data model and data access model) enable specific data processing to handle data source characteristics such as data volume and freshness.

We articulate our approach around different steps which are necessary for completing data aggregation in order to produce smart data. The main steps of the process are extraction from data source, semantic annotation in order to manipulate them as linked data, combination and filtering of produced data in order to remove duplicates and correct malformed pieces. We isolate each task with a different resource enabled by our service oriented architecture implementation.

In order to adapt to diverse scenarios, we provide our architecture with a plugin registry that provides specific implementation for the management of each characteristic. Accordingly, we identify the following challenges and/or scientific locks to address during the data integration process.

- dynamic and transparent data source management:

it must be possible to transparently add or remove a data source at runtime without any need of hard coded information

- scalability and responsiveness: the solution must support a large number of data sources while offering low response time
- dynamic data processing: the solution needs to adapt at runtime to data sources that require different processing (large data volume, frequent update, latency)
- data consistency: provide consistent, error and duplicate-free data

In the following, we illustrate these problems with a set of data sources from our scenario.

1.2 Illustrative Scenario

In the context of our work, we focus on the enrichment and reusability of data handled by a communication company, which has a need for an adaptive system to automatically combine data from their internal information system and enrich it with data from Web sources in order to study the impact of campaign broadcasting over a list of customers. The system must provide decision support tools under the form of recommendations for future broadcasts. The scenario describes the following data sources, each of them presenting different characteristics.

1) an internal linked service giving access to our company business data
2) a SQL database containing a large volume of information (around 100Gb)
3) a database that records user activities (high volume of changing data) with 10.000/20.000 new tuples per day
4) a stream of update requests
5) external APIs (twitter, facebook, dbpedia, etc.)

Each of these data sources present several characteristics. These characteristics are specific to the scenario.

Source 1 is a linked service, i.e. consumes and produces linked data. It provides access to a small data set that describes the business data of the company. Data pieces that come from this source are subject to privacy constraints. **Source 2** is a SQL endpoint to a database that contains millions of tuples with no semantic annotations, this data source has a low update frequency. **Source 3** is a SQL endpoint to a database that contains user daily activities, data from this source are updated regularly, so it requires freshness. **Source 4** is a RSS stream that contains user update requests, it mostly contains data which has to be saved or removed from data results (blacklist information). Other **sources** are represented by a set of APIs (Twitter, Facebook) that help construct interest profiles, as well as a Dbpedia SPARQL endpoint for concept manipulation.

The appearance of one or another characteristic in a data source is unpredictable and may vary from one scenario to another. This unpredictability of variation in scenario clearly illustrate the need for a meta-model in order to fix the limits of data model definition. This meta-model will set the design guideline, and enable the adaptivity of the approach.

1.3 Paper Organization

This paper is organized as follows. Section 2 presents our meta model and models for describing data sources. Section 3 explains our different processing techniques in order to handle the constraints and characteristics data sources provide. Section 4 presents our resource-oriented architecture, details the different components and their orchestration. Section 5 gives an evaluation of our prototype in terms of responsiveness and shows how it responds to user requests with acceptable timings. Section 6 presents related work and highlights the advantages of our approach. Section 7 discusses our results and provides guidelines for future work.

2 Data Source Models

In order to address the above challenges, we introduce a metamodel that allows to describe the characteristics of data and data sources with models according to different characteristics. A *data source model* describes sources in terms of *physical* characteristics such as volume, update frequency, privacy, authentication, semantics, according to the needs of our scenario. A *Data model* defines a scenario-based representation of metadata associated to extracted data from data sources. The data characteristics introduced in this model, when defined, override the properties of the data source model. In addition, we define *data processing workflows* generated with an adaptive algorithm to extract data from data sources according to the characteristics introduced in the models. Hence, these models are used in our smart data architecture that provides a service-oriented solution for data integration.

2.1 A Metamodel for describing data sources

One of the first major challenge that appears during integration of multi-origin data is to handle data source extraction, as data sources are divided into different categories, with different characteristics. Data sources contain or produce data in their own format, responding or not to standardized efforts, e.g. CSV or XML for structured files; tuples for databases; JSON or XML for streams and services. Since data source processing capabilities depend on these characteristics, we build our adaptive integration approach on a flexible data source representation. To do so, we define a meta-model for describing data source models that could easily adapt to any use case. Figure 1 presents this data source meta model.

In this meta-model, we make a difference between a data source and the data extracted from a source. Characteristics appear at different level in this meta-model and can be associated either with the source

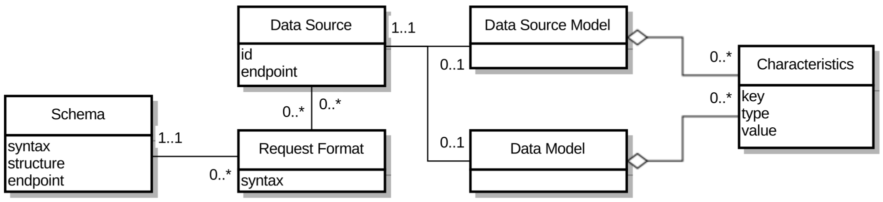

Figure 1. Data source metamodel

itself, which implies that the characteristics are useful to guide the interaction with the data source, or associated with the data to be extracted, in which case they apply to the data schema or to data instances.

Our meta model includes a set of core attributes: URI, request format, data source model and data model. We identify **URI** and **request format** as the mandatory pieces of information needed to manage the data sources. The *request format* characteristic is represented by a *syntax* attribute (e.g. XML, JSON, SQL) and a *schema* defined by three attributes : *endpoint*, *syntax* (e.g. XML, n3, JSON) and *structure* (e.g. RDFS, XSD, JSON Schema).

The *Data source model* is defined as an object that contains all the necessary properties and attributes, that will be used by the client to request the data source. These properties are scenario-specific. The *Data model* defines the set of properties and meta-data information that apply directly to the extracted data. These information are specific to the extracted instances.

While we illustrate the use of this meta-model in the context of our scenario, the former remains appliable to any new data source characteristic and other scenarios.

2.2 Data Source Description Model

Relying on the meta-model presented above, we create an adapted data source model that presents the characteristics which are shared or specific to the different types of data source presented in our scenario (cf Section 1.2). We consider our data sources as defined by a set of the following characteristics: Data Source ID, Endpoint URI (data access transparency), Request format (SQL, SPARQL, JSON, XML etc.), Data volume, Latency, Update period, Authentication, Semantic and Privacy agreement [4]. Fig. 2 presents our scenario-based data source model.

The *URI* characteristic identifies the data source and contains the necessary information to enable the interaction with this data source. An URI is composed by at least : a *protocol*, a *domain* and a *resource*, e.g `file://localhost/home/file1.xml`. The protocol specifies the source connection procedure, such as HTTP, FTP or SGBD connection. URI can also contains authentication information, port number and query parameters. URI can transparently identify any resource, a HTTP URI for a web resource, a file on local system, a database URI, etc.

Request Format defines how to interact with the data source. Most common request formats are SQL, SPARQL, JSON and XML.

Update frequency indicates the recommended average duration between each request to a data source. An update frequency of 0 means that each request may retrieve different data. The update frequency value has an impact on cache or synchronization possibilities.

Volume represents the global quantity of data that a data source manages. Depending on the volume of the data source, specific data access strategies can be adopted. According to the specific strategies to access data, we defined different volume intervals. A `small` volume data source can be accessed directly (less than 20 Mo of data). A `medium` volume data source (from 20Mo to 200Mo) requires cache in order to handle eventual delays. A `high` volume data source (more than 200Mo) may required synchronization systems or big data mechanisms such as map/reduce.

Latency represents the average network time required to obtain a response message to a request on a data source. This value is maintained and updated regularly by statistical measure of delay.

Authentication describes the data source access restriction. This attribute can take different values, or can be disabled if data is publicly accessible. Common values are HTTP-auth, where access is granted by server directives over username/password verifications, OAuth or OpenId, where authentication is handled by a third party server, or SSH public/private keys. In some cases, auth parameters can be specified in the URI, e.g. `http://user:pass@test.com/`.

Semantics aggregates the information required to perform the semantic transformation from raw data to linked data. The semantic description contains : an URI of the linked data graph that describes the data model, an URI of the mapping file that gives information about required data transformation and an attribute identifying the system used to perform the transformation.

Privacy agreements define whether or not data is limited to a specific usage context, according to a set of conditions. Agreements can, as an example, avoid to provide a piece of data to a third party system, or prevent any modification or commercial use of a data piece.

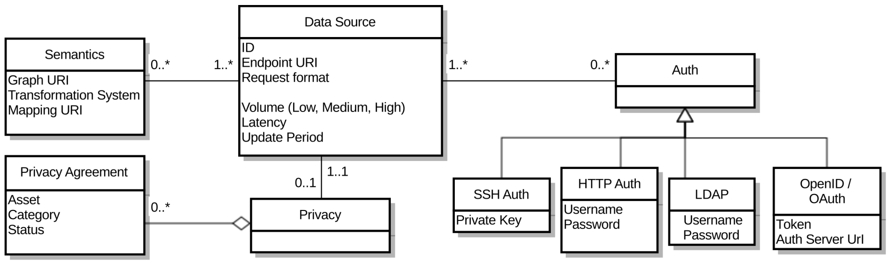

Figure 2. A data source model based on our scenario

2.3 Data Description Model

At the data level, there is a need for a model to describe data characteristics. Based on our scenario, we define here a data model that allows to characterize data sets and instances with specific attributes. The model we devised contains the following attributes that describe data instances: Privacy, Validity, Semantics and Filters. These attributes can be associated with either data tuples or globally with the whole data set.

A set of *privacy* attributes describes privacy requirements that has been given to data values by the data owner. As an example, a user who provides an email address, solely on the condition that she or he does not receive any email, requires a specific data agreement to be associated with the data value.

Validity specifies the lifetime of the data we extract from the data source, in other words it give the date after which the data will be considered as unusable or obsolete. Validity is different from update frequency, as a data source can specify an update frequency of an hour, and specify that the provided data is valid until the end of the year.

Semantics are conceptual metadata which are associated with a data set. Data semantics can be provided together with the data, when accessing the data source, or updated later with a semantic annotation process.

Filters are scenario-based specific attributes, which specify the values in the data set in terms of quality. Filters can specify a detected malformed piece of data, or a forbidden value.

The data constraints introduced in the data model always override the data source characteristics. As an example, a privacy agreement in the data model can specify the recipient allowed for a piece of data, but a data source level privacy agreement specifies a wider recipient will be disabled.

2.4 Data Access Strategies

In the following, we present different data access strategies in order to help our approach handle cases where volume or latency problems hamper data access. A data access model describes several characteristics that affect the way a client connects to and downloads data from data sources. According to characteristic values, different data access policies can be adopted.

Data volume is defined as a discrete scale, as presented before. In case where the data source representation does not specify it, the default volume value is set to small. Technically, low volume sources can be accessed at any times, according to needs. Medium volume sources involve delays and high processing times, so a cache system should be setup. A high volume source can either not be directly queried in a synchronous way, because the volume of data it contains implies a too high response time. In this case, it is recommended to set up a synchronization system, where data is periodically retrieved from the source and saved in a local cache. In the case of big data sources, when data cannot be accessed directly because requests takes to much times, we recommend setting up big data mechanisms such as Map-Reduce to process data in addition to a local cache.

Latency represents the delay (in seconds and milliseconds) between a data request and the received answer, set by default to 0. The system statistically update data source latency value after each request. When latency is high, mechanisms of cache or preloading are set up.

Update period represents the delay between 2 major changes in the data source, set to daily by default. An update period variation will not influence small data sources, but from medium to big data sources, the cache and synchronization systems will be impacted. A short update period will force to increase synchronization delay, and cache will be cleared more often.

We build our access strategies according to two different models: *push* and *pull* strategies.

In pull based strategies, there is no background workflow, data sources are requested directly and on demand. This solution does not imply any storage or synchronization, we request sources, combine results and return the response. This strategy is only available, with low or medium data sources (with or without cache).

In some cases where data source volume is high, or if data source is a stream, data has to be retrieved in background, and stored at a given frequency. We call this a push strategy, it provide some interesting reduction of time and cost for request that involves big data transfer or processing, or in case where the request has a high demand certainty.

In the following section, we present the different processing tasks required to perform the integration process. We identify the needs and propose some orchestration methods in order to optimize computational time and provide a low response time.

3 Data Processing Workflows

From the needs we have identified in the previous sections, we define processing tasks, which we combine in workflows to generate smart data. In this section, we envision different data processing workflows as combinations of functions in different orders. We list in the following the different kind of functions that our architecture manages as resources, before presenting how the different possible workflows are constructed.

3.1 Defining Processing Functions

We consider a data source DS_a, defined by a set of characteristics. We define a download function $DL()$ that extracts a quantity of data D from a data source DS_a.

Definition *1:* $DL(URI_a, S_a) = D_a$ where URI_a, S_a represent the data source DS_a (URI and Model) and D_a the data extracted.

An access function can accept optional parameters (query for databases, authentication parameters etc.). In this case, the download function handles the specific authentication or secure protocol to send the query to the data source. In order to be processed, data needs to be transformed from its raw extracted format to a format we can manipulate. We define a decoding function $Dec()$ which will transform the data into our standard format.

Definition *2:* $Dec(D_{a,r}) = D_{a,f}$ where $D_{a,r}$ is the data extracted into its raw format and $D_{a,f}$ is the transformed data.

In order to combine data from multiple data sources, we need to ensure that concepts from both data source model graph can be compatible. We define this model as G which is a linked data graph. In order to transform the data extracted into an instance of this graph, I_a, we define a mapping function $Sem_a()$ which is defined as:

Definition *3:* $Sem_a(D_a, G) = I_a$ where D_a is the data extracted from data source DS_a, G is the linked data graph, and I_a another linked data graph produced from DS_a.

Once data has been extracted and semantically enhanced, it can easily be combined into a new data set. We define an integration function called $I()$ which takes as input the data sets that have been previously annotated and combines them into a new one.

Definition *4:* $I(G, D_a, D_b, ...) => D_{mix}$ where G is the semantic graph of manipulated data and (D_a, D_b, ...) are semantically transformed extracted data from data source (DS_a, DS_b, ...). Finally, D_{mix} is the smart data set result.

The function relies on graph instances to link the concepts of the different data sets with each other. It analyzes the data pieces which have to be combined and provides on-the-fly mediation by fulfilling the data piece conversions and transformations with help from a set of predefined mediation processes. Based on the domain ontology, the integration function combines the data sets based on their common concepts. The function performs concept matching to link concepts and perform a cartesian product over matching data pieces (i.e. similar to a database join with a pivot). Before performing the combination, the integration function analyzes data and detects heterogeneities, providing mediation based on our previous work with the DMaaS approach [5]. The DMaaS approach proposes an architecture that solves data inconsistencies in service compositions. The approach focuses on service descriptions to analyze conceptual compatibility, and resolves conflictual aspects with help from mediation services.

We define another function $F()$ that removes the malformed part, noise and inconsistencies that may appear in a data set D_a after processing. This function can also take as input a set of conditions to filter data.

Definition *5:* $F(D_a, < Filters >) => D_{a,clean}$

The different processing tasks defined here will help to complete the tasks that participate in the integration process. In the following, we present how these functions can be combined into different processing workflows depending on the characteristics described in the data source and data models.

3.2 Interaction models in the architecture

Figure 3 presents an example of process orchestration in order to integrate data coming from two data sources called S_1 and S_2. Processes are executed from left to right, where boxes represent the previously defined functions. Data is going through the following steps: download (Dl), decode (Dec), transformation into linked data with help from a mapping description (Sem), then both data set are integrated (I) and finally filtered (F).

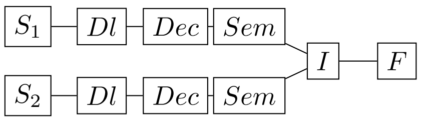

Figure 3. Typical integration workflow

During execution data goes through different states, from the raw original format following data extraction, to our internal format that facilitates manipulation, and finally to the linked data format once annotated. The move from one state to another may have an impact on processing in terms of response time (especially when processing a huge data volume) or data consistency (streams VS static DB).

Therefore, the processing workflow can be envisioned as two connected workflows, where the connection point is the integration function. This way, we define two different parts in the integration workflow: **pre-integration** and **post-integration**. The *pre-integration* part represents the different functions applied to the data set from the extraction from the data source to the integration task. The *post-integration* part begins with data integration and ends with data rendering. This separation helps performing the tasks at different levels, first data preparation aims at preparing data for integration, then the integration task combines data from different sources, and then different functions such as filtering apply to the resulting set.

In a classical workflow, the filtering task is placed after the integration process, so that data cleaning is performed only once. As well, data is typically transformed into linked data before the integration task, because semantic annotations facilitate the integration process.

When it comes to big data volume, combining two huge data sets can be tedious and time-consuming. We then optimize our workflow by swaping components, or by duplicating some of them.

For example, when we combine a really big data set with any other one, it may be interesting to perform a cleaning before integration. In this case, the filtering process can be placed before the integration task or before the semantic transformation task.

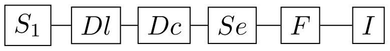

Figure 4. Filtering before integration

Most of the time, with big volume data sources, it can be very interesting to move some tasks forward, in order to reduce the volume we have to process. In some cases, process may be deleted from the pre-integration part and placed in the post-integration part. In the case where the data sources are 2 databases with the same model, or 2 CSV data files that have common fields, performing integration before semantic transformation optimizes the process, because it reduces the size of the data sets to annotate.

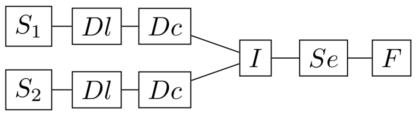

Figure 5. Earlier integration in the workflow

As presented in the previous section, in some cases, specific tasks can be added to the workflow, especially when the semantic model is missing. In this case, the mapping extraction M_{ex} is inserted before the semantic transformation task. Here is an example of a classical integration process where one of the data source does not provide any semantic mapping definition.

In the following, we motivate the need for a resource oriented architecture in order to help task processing as workflows. We extract generic patterns from these workflows and define rules for our orchestrator.

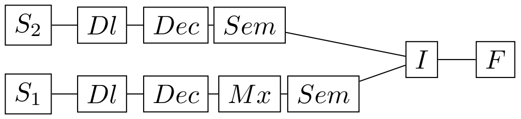

Figure 6. Workflow with a semantic extraction process

3.3 Choice of an Architectural Design

In order to organize and structure the previously introduced tasks into a distributed architecture, we have studied different architecture design patterns, as summarized in the following. The first pattern is an Enterprise Integration Pattern, the next three patterns are related to Service Oriented Architecture, and the last one is related to Resource-Oriented Architecture. We present these patterns and discuss their advantages and drawbacks.

3.3.1 Shared Databases

A shared database architecture [6] is a enterprise integration pattern where different services and components share the same data storage. This type of architecture presents advantages related to the data storage, when enterprises need information to be shared rapidly and consistently. All services and components share the same schema, which helps interaction. Moreover, database studies have shown that this type of architecture is highly adaptable to big volumes, due to database characteristics. Database caching is also widely supported. Nevertheless, this architecture is completely centralized, preventing use of third party services, or components with heterogeneous schemas. Furthermore, the database as a single point of failure becomes an architectural limitation.

3.3.2 Classical SOA architectures

Classical SOA architectures [7] consist of a set of services and static workflows that are compositions of these services. A workflow describes service calls and explicits transformations of data flows. Each use case in the architecture requires a manually crafted and specific workflow. Using such a architecture allows to gain benefits from the principles of SOA, i.e. platform- and location-independent loosely coupled services. It allows to use different service types (SOAP, REST, ...) provided by a variety of third-parties. Despites these advantages, SOA architectures lack adaptivity due to hard-coded workflows. Each task has to be manually integrated into workflows, for example data mediation or caching. Moreover, querying components that are not deployed as services becomes a complex task and requires adapters.

3.3.3 Layered Architecture

A layered architecture [7] gathers services together in layers according to their functional similarity. Each service from a layer may only interact with services of the upper and lower layer. This pattern presents a

	Shared Database	Generic SOA	Layered SOA	ESB	ROA
(+)	- Same schema - Big volumes - Cache	- Independence - Third parties - Service diversity	- Cohesive - Limited coupling - Good reuse - Easy maintenance	- SOA advantages - Easy coupling - Good reuse	- Independence - Uniform interface - Dynamic - Adaptive
(-)	- Centralized - No third party - Database SPOF	- Static workflows - Hard-coded config - Limited to services	- Inflexible - Complexity	- Static routes - Service adapters	- Learning curve - Resource adapters

Table 1
Comparative table of architecture designs

good cohesion within layers, since groups of common featured services are gathered together. This cohesion brings a good separation of concerns, making layers reusable and easy to maintain. Structuring services into layers limits coupling, which simplifies development and facilitates component replacement. Unfortunately, a common schema is needed in the architecture, otherwise it becomes necessary to insert transformation components between each layers. Furthermore, there is a lack of modularity due to this layered data exchange, it is impossible to swap services in workflows for optimization purpose, making the architecture inflexible. And finally, it is difficult to structure layers. If grouping conditions are too strict or too soft, the architecture ends up by having either one layer per service or a global layer which contains all the services.

3.3.4 Enterprise Service Bus

Enterprise service bus (ESB) [8] is a type of architecture based on a message bus where various components connect to a service bus via their service interface. Service composition are managed through the architecture by creating routes. Routes describes service interactions, and a message broker handles the data flowing from and to components. Enterprise service bus architectures present all the advantages of SOAs, including service independence and loose coupling. The usage of a message bus simplifies the integration process. It provides a precise data management in service composition. However, Enterprise service buses have two main drawbacks. First of all, message routes are static, which forbids dynamic composition, and secondly, message bus needs service adapters to connect to different resources and components.

3.3.5 Resource Oriented Architecture

In a resource oriented architecture [9], all the software components must be resources with RESTful interfaces, which means they are accessible through their URI via HTTP methods (GET, POST, PUT, DELETE,...). A RESTful arhcietcture must respect several principles [10], as follows:

- *Uniform interface* ensures that the method information is kept in the HTTP method (we use GET to retrieve a representation of a resource, POST to create a new resource, PUT to upload new representations and DELETE to delete resources), this property also helps to respect statelessness
- *Addressability* ensures that the information about the scope of a resource is kept in the URI, every object will have its own specific URI
- *Statelessness* means that each request happens in complete isolation, and the server does not store any state information, each request contains all the necessary information, thus improving scalability of the solution
- *Representation oriented* means that interactions with resources are made using representations of these resources, request header (such as accept) specifies the desired format

The REST architectural style provides advantages such as a complete independence of underlying platforms and languages, universal interface and access thanks to HTTP methods. Resources can be dynamically composed and reused to fulfill a request.

3.3.6 Our architecture

Table 1 summarizes the advantages and drawbacks of the different architectures presented above. Relying on the previous study, we propose a resource oriented architecture enhanced with a *data bus*. Our architecture handles components as RESTful resources available through a uniform interface via HTTP methods. It overcomes the drawbacks of ESBs, and prevents the use of service adapters. In our architecture, presented in the following, each resource is connected to the bus.

4 A Smart Data Architecture

Our approach relies on a resource oriented architecture in which we define the different tasks required to prepare, semantically annotate and clean data so that it becomes a consistent “smart data” set.

4.1 Global Overview: a Resource-Oriented Architecture

Our architecture follows the principles of SOA [11], which makes our components decoupled, cohesive and reusable, thanks to these properties:

- *Loose Coupling* implies that resources have small or no knowledge of other resources
- *Abstraction* makes components hide their implementation behind a decoupled interface

- *Reusability* maximizes the effort of separating concerns into components in order to reuse them
- *Autonomy* ensures that components have control over what their implementation and are independent from their execution environments
- *Composability* ensures that components can be combined in order to solve different kind of problems

In order to build a completely generic and modular architecture, we deploy our components as RESTful resources, i.e. relying on the REST principles presented above. Thanks to the REST and SOA principles, our architecture is generic, scalable and modular, as it is composed of different resources that can be dynamically orchestrated in different ways as presented in the following.

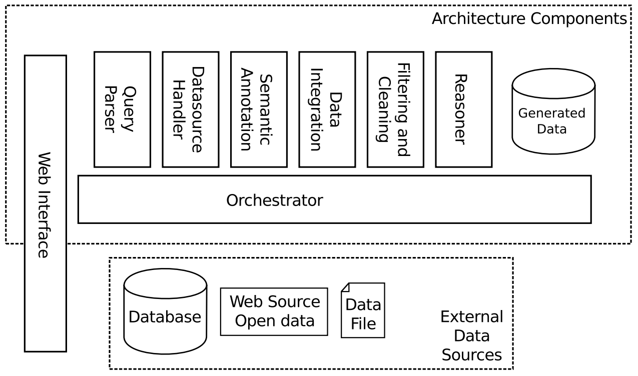

Figure 7. Architecture Resources

We defined a set of architecture components, as shown in Fig. 7, exposed as generic RESTful resources identified by URIs and accessible through HTTP methods. These *business resources* are the core of the architecture, they handle the main data processing tasks. In addition to these core resources, we defined resources that support task configuration and administration, referred to as *management resources*.

4.2 Core Resources

The *data source handler* resource manages the access and data extraction from data sources (DL and Dec tasks of Section 3.1). The *semantic annotation* resource helps to annotate and transform data coming from diverse sources into linked data (Sem task). The *data integration* resource combines multi-origin data and resolves heterogeneities that appear (I task). The *Filtering and Cleaning* resource filters data and removes duplicates as well as malformed pieces of data (F task).

The *Reasoner* (running as a background task) infers new facts from existing data with help from customer-defined business rules. Finally, the *Web Interface* combined with the *Query Parser* handle user interaction and data requests to the architecture.

In the following, we present the *core* resources that handle the tasks presented in Section 3. We then present the *management* resources that allow to configure component usage in the background. Please note that all components do not necessarily have a resource configuration API.

4.2.1 Web Interface and Query Parser

The Web interface is the main entry point to the architecture. User interactions through this interface generate queries that are sent to the architecture, through the orchestrator API. **Data queries** are formatted in SPARQL and involve semantic concepts. Listing 1 gives an example of query that involves a set of concepts belonging to our scenario.

```
PREFIX al: <http://restful.alabs.io/concepts/0.1/>
SELECT ?email_value ?campaign WHERE {
    ?email         a                    al:email ;
        al:has_email_value  ?email_value .
    ?email_value a                      al:email_value .
    ?clic          a                    al:clic ;
        al:clic_email       ?email ;
        al:clic_campaign    ?campaign .
    ?campaign   a                       al:campaign .
}
```

Listing 1. A data query example

Data queries are forwarded to the query parser, which extracts the corresponding algebra, as a set of subgraphs and concepts[2]. Subgraphs and concepts help to detect the different data sources involved. The parser generates a workflow, involving different architecture resources and the data sources that are needed.

4.2.2 Orchestrator

In order to handle data flows between resources, we define an orchestrator, which acts as a data bus and receives HTTP requests from the Web interface. On request reception, the orchestrator extracts the SPARQL query from the request and forwards it to the query parser, which returns a workflow, defined as a sequence of tasks as defined in Section 3. According to this workflow, the orchestrator handles requests to the different architecture resources, retrieving data responses and forging HTTP requests. The orchestrator is defined as a RESTful resource, user queries are sent through HTTP requests. In order to follow the REST principles, we used HTTP GET requests are used for SELECT queries and HTTP POST requests for UPDATE.

```
GET /query?user_token=AS65G&query=SELECT+%3Femail_value
      +%3Fcampaign+WHERE+%7B%0D%0A++%3Femail...
      HTTP/1.1
Host: restful.alabs.io
Keep-Alive: timeout=15
Connection: Keep-Alive
```

Listing 2. Sample of HTTP request embedding a SPARQL query

2. See https://github.com/semsol/arc2/wiki for a documentation about the SPARQL parser we use.

4.2.3 Data Source Handler

The **data source handler** allows to extract data from the different data sources involved in the query subgraphs. This resource accesses each data source and extracts data with the help of the data source description (see Section 2).

In order to extract data from a data source, the handler accepts HTTP requests (GET to read, POST to update, ...) with as parameters the data source representation and the data query. Listing 3 illustrates an example of data source configuration resource.

```
"source":{
    "id" : 1,
    "format":{
        "syntax":"mongodb",
        "schema":{
            "endpoint":"http://153.75.28.26/schema_def",
            "syntax":"JSON",    "structure":"JSON-S"    }    },
    "uri":"mongodb://user1:76ls6h@153.75.28.26:80/myDBendpoint",
    "volume" : "low"
}
```

Listing 3. A data source configuration file example

Relying on this information, the resource retrieves (or deploys, according to access strategies defined in Section 2.4) an adaptive client that handles the characteristics of the data source, authentication, format, volume, etc.

4.2.4 Semantic Annotation Resource

Relying on the semantic information provided in the data source representation, the **semantic annotation** resource will either annotate with concepts or transform data into linked data.

The resource uses different techniques to enhance semantics of raw data, depending on the kind of data source. As an example, for CSV file sources, we rely on the RDF123 approach [12] to transform raw data into linked data. This approach relies on expressions to map the contents of spreadsheet columns to linked data. The semantic annotation resource also relies on the D2R approach [13], in order to transform data from relational databases into linked data. The D2R platform is based on RDF mappings that attach conceptual graphs to SQL requests, giving access to relational data through a SPARQL endpoint. The resource benefits from a management API, presented later in this section, which helps to generate or retrieve semantic mapping information for each data source.

4.2.5 Data Integration Resource

The **data integration** resource interconnects the data sets that have been extracted from sources and annotated with linked data concepts. The integration resource aligns the different concepts, relying on the user data query to construct the graph represented by this query. The architecture analyses the concepts in the query and prepares data for the merging process, detecting the pivot values if they exist, relying on metadata to connect data from the sources involved.

To detect possible heterogeneity problems, could they be syntactic or structural, and to reconcile them, we rely on our previous DMaaS approach [5]. This approach analyzes semantically described data, using a decentralized (peer to peer) repository of mediation services[3]. The DMaaS approach classifies data heterogeneity issues according to the syntactic, structural and semantic levels, and provides adapted mediation along these levels.

4.2.6 Filtering Resources

In our architecture, when data is processed in one resource or another, noise and inconsistencies may appear, as well as duplicate values or instances. The cleaning and filtering resource handles different data cleaning tasks, including data duplicate removal, incomplete data removal, formatting and encoding issue processing and data removal when an issue cannot be solved (damaged data).

In addition to these resources, the user has the possibility to provide specific filtering rules, to ignore specific data values, or to limit fields to range domains. Therefore, the cleaning and filtering resource has a management API, where users can manage their specific rules. Listing 4 presents filtering rules samples.

```
{
    "@context":{
        "@vocab": "http://example.com/filter/",
        "vcard": "http://www.w3.org/TR/vcard-rdf/",},
    {   "@id": "http://example.com/filter/r1",
        "data": "vcard:email", "operator": "not-contain",
        "value": ".org"     },
    {   "@id": "http://example.com/filter/r2",
        "data": "vcard:age", "operator": "more-than",
        "value": "20"       }
}
```

Listing 4. Filtering rules example

4.2.7 Reasoner

Our architecture also integrates an inference engine called **reasoner**. The reasoner runs in background, regularly analyzes data and generates rules and statistical knowledge. It updates a internal database with the statistical data. We integrate such an engine to improve user query response time, by statistically updating data source descriptions, and adapt with changing data source status. We also use the inference engine to generate facts and deal with complex user queries. We use classical AI mechanisms and algorithms such as collaborative filtering (Apache Mahout) and semantic Web inference engine (Jena, Pellet, Euler EYE, HermiT)[4]. This resource has a management API that allows users to manually add specific business rules and facts, that will drive the inference engine.

3. Mediation services are Web services dedicated to data conversion.

4. See http://www.semantic-web-journal.net/sites/default/files/swj120_2.pdf for a comparison of reasoners.

4.2.8 *Authentication and Data Security*

In order to secure the access to resources and data, we overlay our architecture with an authentication system, relying on the oAuth [14] authentication framework. OAuth relies on a authorization server which authenticates user access upon a resource from a server. The framework relies on authentication tokens generated by an authentication server. These tokens can be used to access the resources owned by an information system. The decentralized scheme of the architecture involves authentication of each resource exchange.

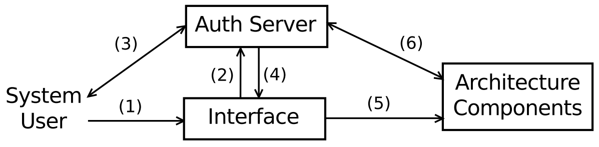

Figure 8. Architecture authentication process

As illustrated in Figure 8, the user connects to the architecture interface (1), the architecture redirects him to the authentication server interface (2). The user logs in through this interface (3). The authentication server generates a token (4) that authorizes the Web interface component to access the different resources on behalf of the user (5), each layer verifies with the authentication server the token freshness and validity (6).

4.3 Management Resources

In order to make architecture management and administration easier, and to avoid manual configurations as much as possible, we provide our architecture with management resources, accessible via a set of APIs. Through these APIs, users can modify resource behaviors and settings, add or remove data sources, request the generation of a mapping for these data sources, plug or unplug core components from the plugin registry, define specific business-oriented rules. We present these management resources in the following.

4.3.1 *Data Source Handler Configuration*

This resource provides an API that allows to perform the four CRUD operations (Create, Read, Update and Delete) over data sources. Data sources are retrieved and manipulated with their *ids* and according to the model instance that describes their characteristics. As an example, a GET request over the data source configuration resource with the `id` of a data source : `GET /datasource/1` returns the JSON object representing the data source with the `id` 1 described in Listing 3. Data sources can be published in the architecture with POST requests, providing the JSON data source description as request body.

4.3.2 *Management of Semantic Mappings*

In order to manage mappings for each data source, we provide our architecture with management resources to create, modify, and delete mapping files and semantic mapping information. Mapping file generation relies on existing third party approaches for semantic annotation and transformation, depending on the type of document or data source we want to annotate or transform into a semantically explicit representation (see Section 4.2.4).

In order to manipulates semantic mappings, we rely on the *id* identifiers of the data sources they enrich. As an example, the mapping information for a data source can be retrieved by a GET request over the semantic resource with the `id` of this data source : `GET /semantics/#id` returns the required mapping file in order to add semantics to data extracted from data source `#id`. A POST request at the same URI is used to submit a new mapping file, DELETE and PUT to destroy and update. Depending on the data sources, the mapping contains different sets of rules, and the URI of the semantic transformation or annotation service. In order to process the data source to extract the corresponding mapping, when available, the following POST request : `GET /semantics/extract/\#1` will to generate and return the mapping file.

4.3.3 *Cleaning and Filtering Rules*

In addition to cleaning and filtering resources, we provide a management resource allowing users to publish their own cleaning rules. These specific rules allow to ignore specific data values, or to limit the range domain of a concept. We define an API to publish, search and remove these rules as follows : The URI `/filter` allows to create, retrieve, delete and update rules according to their id, an example of rule creation is shown below. In addition, `/filters` allows to retrieve a set of rules for a specified data source id, through GET requests. Listing 5 shows an example of HTTP POST request that publishes the filtering rule introduced in section 4.2.6.

```
POST /filter HTTP/1.1
Content−type:application/x−www−form−urlencoded;charset=utf
      −8
Host: restful .alabs. io
Content−length:200
id=r1&data=vcard:email&op=not−contain&val=.org
```

Listing 5. Sample of HTTP request to publish a filtering rules

4.3.4 *Business Rules for Reasoning*

We defined a management resource for the reasoner, which gives the ability to publish, delete, and update business rules that change the behavior of the reasoner. Business rules are defined as specific links between domain concepts. As an example, it is possible to specify hierarchical ($rdfs$: $subClassOf$) or similarity (owl : $sameAs$) relations between interests or concepts. These rules are published as RDF/N3 resources with reasoner's API.

4.4 Example of data flow in the architecture

Figure 9 illustrates an example of data flow during a scenario request. Firstly, the system user creates a data

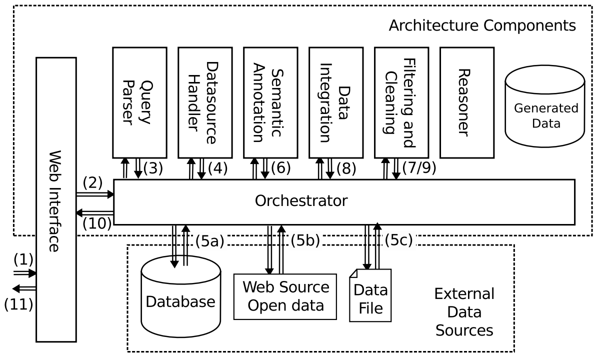

Figure 9. Use Case Data Flow Representation

request through the Web interface (1). The request is sent to the orchestrator (2) that calls the query parser, which returns a resource composition with the data sources involved to cover the query (3). With help from the data source handler, which processes the data source description (4), data is extracted from the sources, could it be a database ($5a$), a Web source ($5b$) or a data file ($5c$). The filtering resource performs the data cleaning and filtering tasks, including duplicate removal, user specific rules, etc. (6). Data is annotated with semantics, i.e. linked data concepts through the semantic resource (7). If needed, another cleaning task is performed to clean data that have been identified as noise after the semantic annotation process (8). Data integration is performed and the multi origin data pieces are combined (9). Data is again cleaned and filtered (10). The generated data set is finally returned to the user through the Web interface $(11, 12)$.

5 Tests and Evaluations

In order to answer the challenges introduced in this paper, the different models to be created based on our meta-model allow for dynamic and transparent data source management and processing. These objectives, as well as the data consistency objective, have been evaluated with the implementation of our Architecture for Integration of Multi-Origin Data Sources (ArchI-MODS).

The scalability to a large number of data source cannot be guaranteed *a priori* by our model nor our implementation. We answer the scalability challenge by evaluating the evolution of request response time over a growing number of data sources. We evaluate our architecture in terms of performance (response time) when answering to a set of complex semantic queries over multiple data sources. We regularly increase the number of data sources and measure the response time.

Relying on our scenario presented above, we create two requests, involving subgraphs containing four concepts. We populate our scenario with a set of data sources covering the different subgraphs. Query 6 involves four subgraphs, implying data sources with different characteristics such as high volume (big database in our scenario) and privacy sensitive information (linked service in our scenario).

```
PREFIX al: <http://restful.alabs.io/concepts/0.1/>
SELECT ?email_value ?campaign WHERE {
    ?email        a                 al:email ;
        al:has_email_value  ?email_value .
    ?email_value a                 al:email_value .
    ?clic         a                 al:clic ;
        al:clic_email        ?email ;
        al:clic_campaign     ?campaign .
    ?campaign    a                 al:campaign .
}
```

Listing 6. Query 1 involving four concepts

Query 7 involves only three subgraphs, but includes user specific filters. This query introduces freshness sensitive data sources (streams in our scenario).

```
PREFIX al: <http://restful.alabs.io/concepts/0.1/>
PREFIX xsd: <http://www.w3.org/TR/xmlschema-2/>
SELECT ?email_value ?campaign WHERE {
    ?email        a                 al:email ;
        al:has_email_value  ?email_value ;
        al: blacklist_status  ?status .
    ?clic         a                 al:clic ;
        al: clic_email       ?email ;
        al: clic_date        ?date .
    FILTER (?status != 1 && ?date >= "1411477450"^^xsd:date)
}
```

Listing 7. Query 2 introducing user specific filters

Fig. 10 shows the evolution of our architecture response time, when the number of data source grows. The graph also presents different composition techniques, that clearly shows the importance of adaptive composition. Workflow $WF1$ composes the steps of integration in a static way, which is quite well adapted for small data source sets, but does not scale when data source number grows. The second workflow $WF2$ introduces a dynamic composition, where the architecture is provided with the possibility to permute components, performing the filtering process before combining data.

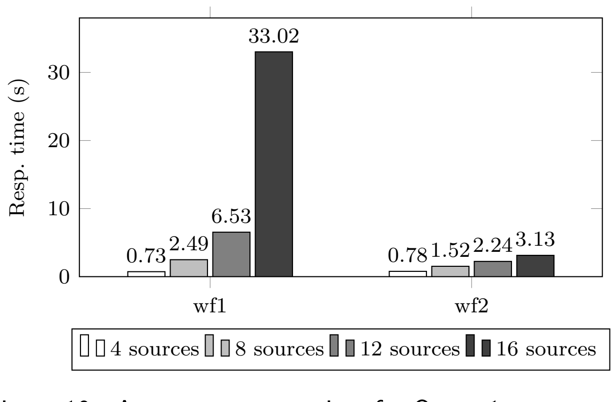

Figure 10. Average response time for Query 1

This graphs shows that our architecture can handle the growth of data source number, as long as we use a dynamic composition model. In the case of the first composition model $wf1$ the combination of data become a time-consuming process, as response time grows exponentially. For more than 20 data sources, with the first workflow, architecture takes minutes to answer the query. With a dynamic composition workflow, avoid composing duplicates and non well formed data severely improves the process.

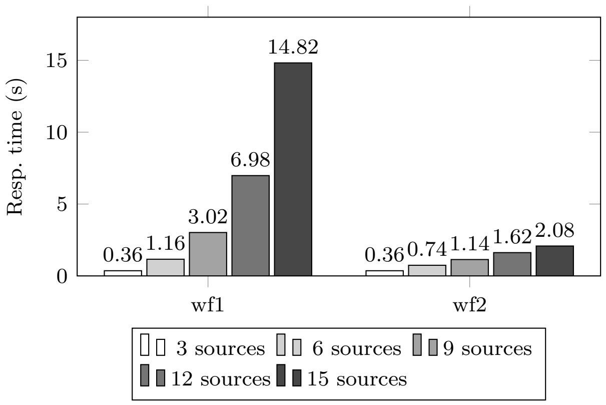

Figure 11. Average response time for Query 2

The second query involves less concepts, and allows the architecture to give better responses with the first workflow, but it still takes more than a minute to answer Query 1 with 20 data sources. The second workflow adapts to the request and provides good results.

6 Related Work

The study and design of architectures to automatically integrate data from diverse resources and produce smart data is currently a hot research topic explored by the community. Smart data has caught the interest of the community as a natural development after the interest around big data. The objective with smart data is focused on producing high-quality data that is directly useful to the users, instead of big data approaches that focused on building solutions to process massive data quantities.

Dustdar et al. present a *peer data network* architecture in [15], where data sources are independent databases. They propose an infrastructure relying on data services where tasks are separated into levels, isolating data management and service integration. Their solution focuses on quality of data and provides service-based optimization, such as peer replication, to resolve data issues. However, the paper does not address data heterogeneity problems, assuming that schema mapping is sufficient.

QuerioCity [16] presents a smart platform to catalog, index and query heterogeneous information from complex systems such as city data portals[5]. They focus on data integration and semantic annotation problems, mainly on issues related to scalability, unpredictability of changing data and impossibility to fully automate the integration process. The proposed approach clearly distinguishes between the data integration and data consumption tasks. In order to harmonize data usage, data fields are converted to a standard format, annotated with metadata and aligned with public ontologies (Dublin Core [17] or FOAF [18]). The effort is placed in management of extracted data, data access challenges and the needs which arise are not part of the scope. The approach focus essentially on extraction of meaning and semantics from datasets as well as provenance in order to provide a harmonized dataset. They do not provide any information about data source classification, and assume that sources have to meet the request format that the architecture supports.

With a more functional and industrial point of view, Atos Worldline propose a solution for the automatic management of data coming from heterogeneous sources called **SmartData.io** [19]. The solution provides a RESTful API through which it is possible to publish data sources and data streams presented in several formats, such as CSV, PDF or RSS. Data is then extracted from files, converted into a pivot language (which is JSON) and then preprocessed by specific applications, which can be internally developed by the company or externally developed by third parties. Depending on the application, the extracted data is filtered by applying patterns or by combining it with additional data. Doing so, only the necessary and correct data is stored into the infrastructure. The presented architecture has very interesting aspects especially about automatic data processing. However, it does not provide any clues about the semantic enrichment of data, neither about how heterogeneities are handled during the integration process.

Apache Metamodel [20] proposes a data access framework, which offers a transparent rich query interface to different types of datastores, which does or does not usually provide this kind of request abilities. The framework provide a uniform connector to many types of datastore types, including several type of databases, data files or objects. They proposes a scripting language for processing updates and transactions via APIs. The architecture provides a uniform driver approach for requesting data sources, but acts as a static process where each source type has a particular adapter. The approach does not address challenges related to data combination and data source configuration at runtime, it provides a universal access to each data source and a language to request these endpoints.

We have been taking into account the strengths and weaknesses of these different approaches to build our proposal, improving the reusability and loose coupling through usage of linked data services, automating the linked data efforts by proposing a distributed approach for the different tasks to perform on data.

In another context, some approaches propose techniques for semantically annotating raw data from heterogeneous sources.

Furth et Al. [21] propose an approach for the semantification (enrichment with semantic description) of technical documents. It relies on different working steps over the document to be enriched. First, it converts the document into a standard format, then it

5. Such as Dublinked http://www.dublinked.ie/

splits the document into segments, and applies natural language processing techniques to the document parts in order to extract a set of ranked concepts. This set of concepts represents the main subject of the document. The strength of this approach is that it does not require a huge set of training data to provide a classification. They provide a performance evaluation tool by adding a manual step allowing domain experts to review results.

Venetis et Al. [22] describe a system that recovers semantics from tables existing on the Web. Their solution relies on the help of a set of millions of other tables to identify the role (or subject) of each column/attribute. The solution stands on performing similarity computation with the corpus of tables and extracting entities with the help of natural language processing over table values. The main drawback of this approach is that it requires a huge amount of objects in the corpus to analyze. Moreover they rely on millions of English-speaking documents to build their relation and entity extractor, which severely decreases the scalability of their approach.

In **TARTAR** [23], Pivk proposes a solution for extracting knowledge from tables picked up from the web. The solution, based on Hurst's table model[24], relies on the analysis of table accross diffferent points, including physical, structural, functional and semantical dimensions. The first step is a regular matrix extraction from physical dimensions. Analyzing the structure allows to determine table reading orientation, to dismember a table into logical subpart and to resolve types. The functional table model helps to deduce the role of each cell. A last step provides semantic concepts and label for each column, using tools such as WordNet. These models and concepts allows to populate a domain ontology from table rows.

The previously introduced approaches show the effort made towards automatic semantic annotation of data. For the sake of simplicity and effectiveness in our approach, we decided to rely on a semi-automated technology in order to add semantics to our data pieces. We present in the following different solutions that rely on the design of map files in order to generate Linked Data directly from data sources. There are different approaches in this area, technical approaches as well as theoretical approaches.

Han et Al. present in **RDF123** [12] an open-source tool for translating spreadsheet data to RDF. They rely on a column-based mapping, where a set of expressions represents the structure and orchestration of cells in a tabular row. They define a whole language to describe these expressions, allowing to define control branches and data manipulations. The generated mapping file containing the set of expressions can be serialized as RDF and placed as a link in spreadsheet metadata, for reusability.

The **Triplify** [25] solution proposed by Auer et Al. allows to attach to a pre-existing system a module that will publish data as a Linked Data store. The solution creates a set of configuration files and associates semantic concepts (URIs) with SQL requests. Once the configuration has been created and the module has been integrated to the system, the module is accessible as a web page within the application and will be registered with a central repositories of data sets. This solution does not allow any flexibility, since each configuration is hard coded in the system. Otherwise, the system does not provide any computational access, and access is only accessible as a generated HTML/JS interface. Accessing the generated linked data pieces in order to manipulate them is not possible without changing the core of the product.

Bizer et Al. propose the **D2R** [13] platform, gives access to relational databases through a SPARQL endpoint. The platform rely on a virtual RDF graph, which associates concepts and relations to SQL requests. D2R gives access to databases relying on a n3 mapping files which can be generated by one of the platform tool. This tool rely on inherent database structure (foreign keys and relations), to deduce relations between fields of tables. In order to make a database available, the configuration has to be generated and the platform launched through an application server.

We rely on these latter approaches to perform our semantic annotation task, improving it by automating the creation of the mapping expression with help from external and third party services for semantics extraction and concept recognition.

7 Conclusion

In this work, we propose an architecture and models to improve smart data management when data comes from different sources with heterogeneity issues, malformed data and duplicates. We propose a flexible solution to model data sources and data according to their characteristics, allowing the use of different data access and processing strategies. To handle data management, we define a scalable and responsive architecture that orchestrates RESTful resources, accessible through their uniform interface to enhance interoperability. Our architecture allows converting and semantically annotating multi-origin data sources in order to produce a smart data set. It aims at being as generic as possible, independent from data sources, and adaptable to any use case. We demonstrate the applicability of our architecture in the context of a scenario that answers the needs of our partner company. We also show its scalability with experiments in real situation. Future work includes performing additional evaluation over large data sets and exploring issues related to data management such as data quality and freshness issues, and reasoning about inconsistent or imprecise data.

References

[1] T. Heath and C. Bizer, *Linked Data: Evolving the Web into a Global Data Space*, ser. Synthesis Lectures on the Semantic Web. Morgan & Claypool Publishers, 2011.

[2] M. Maleshkova, C. Pedrinaci, and J. Domingue, "Investigating web apis on the world wide web," in *ECOWS*, A. Brogi, C. Pautasso, and G. A. Papadopoulos, Eds. IEEE Computer Society, 2010, pp. 107–114.

[3] D. Benslimane, S. Dustdar, and A. P. Sheth, "Services mashups: The new generation of web applications," *IEEE Internet Computing*, vol. 12, no. 5, pp. 13–15, 2008.

[4] H.-L. Truong, S. Dustdar, J. Goetze, T. Fleuren, P. Mueller, S.-E. Tbahriti, M. Mrissa, and C. Ghedira, "Exchanging Data Agreements in the DaaS Model," in *The 2011 IEEE Asia-Pacific Services Computing Conference*. IEEE, Dec. 2011, pp. 153–160.

[5] M. Mrissa, M. Sellami, P. D. Vettor, D. Benslimane, and B. Defude, "A decentralized mediation-as-a-service architecture for service composition," in *WETICE*, S. Reddy and M. Jmaiel, Eds. IEEE, 2013, pp. 80–85.

[6] G. Hohpe and B. Woolf, *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[7] T. Erl, *SOA Design Patterns*, 1st ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2009.

[8] D. A. Chappell, *Enterprise service bus - theory in practice*. O'Reilly, 2004. [Online]. Available: http://www.oreilly.de/catalog/esb/index.html

[9] L. Richardson and S. Ruby, *Restful Web Services*, 1st ed. O'Reilly, 2007.

[10] R. T. Fielding and R. N. Taylor, "Principled design of the modern web architecture," in *Proceedings of the 22Nd International Conference on Software Engineering*, ser. ICSE '00. New York, NY, USA: ACM, 2000, pp. 407–416. [Online]. Available: http://doi.acm.org/10.1145/337180.337228

[11] T. Erl, *Service-Oriented Architecture (SOA): Concepts, Technology, and Design*. Prentice Hall PTRs, Aug. 2005.

[12] L. Han, T. Finin, C. Parr, J. Sachs, and A. Joshi, "Rdf123: From spreadsheets to rdf," in *The Semantic Web - ISWC 2008*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, vol. 5318, pp. 451–466.

[13] C. Bizer, "D2rq - treating non-rdf databases as virtual rdf graphs," in *In Proceedings of the 3rd International Semantic Web Conference (ISWC2004*, 2004.

[14] B. Leiba, "Oauth web authorization protocol." *IEEE Internet Computing*, vol. 16, no. 1, pp. 74–77, 2012. [Online]. Available: http://dblp.uni-trier.de/db/journals/internet/internet16.html#Leiba12

[15] S. Dustdar, R. Pichler, V. Savenkov, and H. L. Truong, "Quality-aware service-oriented data integration: requirements, state of the art and open challenges." *SIGMOD Record*, vol. 41, no. 1, pp. 11–19, 2012.

[16] V. Lopez and S. Kotoulas, "Queriocity: A linked data platform for urban information management." in *International Semantic Web Conference (2)*, ser. Lecture Notes in Computer Science, vol. 7650. Springer, 2012, pp. 148–163.

[17] S. Weibel and T. Koch, "The dublin core metadata initiative : Mission, current activities, and future directions," *D-Lib Magazine*, vol. 6, no. 12, 2000.

[18] D. Brickley and L. Miller, "The Friend Of A Friend (FOAF) vocabulary specification," November 2007, http://xmlns.com/foaf/spec/. [Online]. Available: http://xmlns.com/foaf/spec/

[19] A. Worldline. (2013) Smartdata.io : A dedicated solution to the data management. [Online]. Available: http://api.docs.v2.smartdata.io/

[20] Apache. (2013) Apache metamodel : A data access framework. [Online]. Available: http://metamodel.incubator.apache.org/

[21] S. Furth and J. Baumeister, "Towards the semantification of technical documents," in *FGIR'13: Proceedings of German Workshop of Information Retrieval (at LWA'2013)*, 2013.

[22] P. Venetis, A. Y. Halevy, J. Madhavan, M. Pasca, W. Shen, F. W. 0003, G. Miao, and C. Wu, "Recovering semantics of tables on the web." *PVLDB*, vol. 4, no. 9, pp. 528–538, 2011.

[23] A. Pivk, "Automatic ontology generation from web tabular structures," *AI Communications*, vol. 19, p. 2006, 2005.

[24] M. Hurst, "Layout and language: Challenges for table understanding on the web," in *Proceedings of the International Workshop on Web Document Analysis*, 2001, pp. 27–30.

[25] S. Auer, S. Dietzold, J. Lehmann, S. Hellmann, and D. Aumueller, "Triplify: Light-weight linked data publication from relational databases," in *Proceedings of the 18th International Conference on World Wide Web*, ser. WWW '09. New York, NY, USA: ACM, 2009, pp. 621–630. [Online]. Available: http://doi.acm.org/10.1145/1526709.1526793