DaWeS: DataWarehouse fed with Web Services

John Samuel Samuel, Christophe Rey

DaWeS:
Data Warehouse fed with Web Services

John Samuel, Christophe Rey

LIMOS, CNRS
Université Blaise Pascal
Aubière, France
samuel@isima.fr, christophe.rey@univ-bpclermont.fr

RÉSUMÉ. *Nous présentons un prototype, appelé DaWeS, d'entrepôt de données alimenté par des services web. La spécificité de DaWeS est son approche médiation (intégration de données sans matérialisation) comme outil ETL (extraction, transformation et chargement des données). Cette approche permet d'automatiser une grande partie du processus ETL, tout en facilitant les interventions humaines par l'emploi exclusif de langages déclaratifs (requêtes datalog, SQL, XSD, XSLT). Le contexte de cette étude est celui des standards relatifs aux services web les plus utilisés car les plus simples (HTML, HTTP, REST, XML, JSON), et non des standards plus élaborés mais moins utilisés (SOAP, UDDI, WSDL, SA-WSDL, OWL-S, hRESTS). En termes applicatifs, l'ambition est de permettre à l'administrateur de DaWeS de proposer aux petites et moyennes entreprises un service de stockage et d'interrogation de leurs données métier liées à l'utilisation de services web tiers, sans avoir elles-mêmes à gérer leur propre entrepôt. En particulier, DaWeS permet la définition facile d'indicateurs de performance personnalisés.*

ABSTRACT. *We present a prototype, called DaWeS, which is a Data Warehouse fed with Web Services. The main feature of DaWeS is to use a mediation approach (data integration without materialization) as the ETL tool (data extraction, transformation and loading). This approach enables to automate many steps of the ETL process, while facilitating human interventions by exclusively relying on declarative languages (datalog queries, SQL, XSD, XSLT). The context of this work consists of the mostly used (because the simplest) web services standards (HTML, HTTP, REST, XML, JSON), and not of the more complex but less used ones (SOAP, UDDI, WSDL, SA-WSDL, OWL-S, hRESTS). In terms of applications, the aim is to allow a DaWeS administrator to provide to small and medium companies a service to store and query their business data coming from their usage of third-party services, without having to manage their own warehouse. In particular, DaWeS enables the easy design of personalized performance indicators.*

MOTS-CLÉS : *médiation, entrepôt de données, services web, ETL, intégration de données, réécriture de requêtes*

KEYWORDS: *mediation, data warehouses, web services, ETL, data integration, query rewriting*

1. Introduction

The past two decades have seen the rise of many Web Services (WS) providers offering a reduced subset of services rather than the traditional bloated software applications. These services are heterogeneous, autonomous and ever evolving. Enterprises using WS have no direct control over the underlying data infrastructure and thereby over their own business data. The only convenient mechanism for enterprises to access and manipulate their data is through application programming interface (API) exposed by service providers to allow the clients to build their own internal dashboards. WS APIs differ among each other significantly with respect to the use of different message formats, authentication mechanisms, service level agreements, access patterns, data types, and the choice of input, output and error parameters. APIs are mostly described using human readable (HTML) web pages. Furthermore service providers often make updates to their services (addition and deprecation of resources, change in the API or SLA). These changes may lead to the losing of past enterprise data. All the aforementioned challenges make it difficult for small and medium scale enterprises with lesser human resources and expertise to easily integrate with numerous WS.

Companies traditionally use a data warehouse to perform business analysis, compute performance measures (aka indicators) and track their growth. The purpose of our work is to aid enterprises using WS for their day to day business activities with a data warehouse service. We are building a multi-enterprise *Data Warehouse fed with Web Services (DaWeS)* able to fetch interesting data from various WS and expose them in a manner so that the end users can compute their own interesting performance indicators without having to manage their own warehouse.

In this paper, we present an experimental study of a prototype which aims at being a convenient and realistic semi-automated system. Indeed, our goal is not to build a fully automated WS fed data warehouse, which seems quite impossible, but to ease as much as possible the coding burden of adding and updating new WS (achievable by only a couple of developers or administrators). The feeding of the data warehouse is achieved through the use of mediation techniques associated to a generic wrapper. Though our experiments focus on REST (like) services given their popularity, it can be easily extended to SOAP services and the possible future availability of machine readable standards like WSDL will further reduce the administrators' tasks.

In section 2, we concretely explore three domains to determine what are the mostly used WS standards. Section 3 formally presents the mediation approach, coupled with the generic wrapper, used to feed the data warehouse. Section 4 describes DaWeS architecture, development and the various experiments. Section 6 describes the various related works. Finally we conclude by describing current and future works.

2. Data Warehouse and Web Services

This section surveys 12 WS belonging to three business domains to establish the mostly used WS standards that are effectively used by service providers. The three

studied domains are email marketing, project management and helpdesk (support). Email marketing is a form of direct marketing which uses email campaigns as a means for communicating to a wide (subscribed) audience about new products and technologies. Project management encompasses many activities : planning and estimation of projects, decomposing them to several tasks and tracking their progress. Helpdesk is focused on managing customers' (intended or current) problems, complaints and suggestions on an online web portal internally tracked using tickets.

The 12 surveyed WS are project management (Basecamp : www.basecamp.com, Liquid Planner : www.liquidplanner.com, Teamwork : www.teamworkpm.net and Zoho Projects : www.zoho.com/projects), email marketing (iContact : www.icontact.com, CampaignMonitor : www.campaignmonitor.com and MailChimp : mailchimp.com) and helpdesk (Zendesk : www.zendesk.com, Desk : www.desk.com , Zoho Support : www.zoho.com/support, Uservoice : www.uservoice.com and FreshDesk : www.fresh desk.com). Each of the previous service may propose many operations, each of which has a callable API. The characteristics of these APIs are given in table 2 where WS are classified according to : the language in which APIs are described (i.e., documented), their REST compliance (Fielding, 2000), their version of the API, their authentication method, the resource they deal with (e.g. *task* or *todo* in a project management service, *ticket* in an helpdesk service), their message format, the used service level agreement (constraints on the operations usage) their HTTP access method, the used data types, their handling of dynamic resources (resources which value can evolve), mandatory constraints during operation invocation (e.g. to get all the tasks, it first requires in Teamwork to get all the projects, following retrieving all the task lists in all the projects and finally followed by obtaining the tasks from all the task lists), and their pagination features (i.e., one or many call(s) to retrieve all data).

From these characteristics, an average profile of WS emerges : describing services with HTML, following the REST architecture, using basic HTTP authentication with a GET access, XML or JSON as message format, enumeration and date as data types, dynamic resources and sequence operation invocation. This average profile clearly focuses on simplicity. The consequence is a low level of service management automation. For example, none of these services are described using a computer-oriented language (with or without semantic features) like WSDL (W3C, 2001), SA-WSDL (Kopecký *et al.*, 2007), DAML-S (Burstein *et al.*, 2002), OWL-S (Martin *et al.*, 2007), hRESTS (Kopecký *et al.*, 2008). This situation is also confirmed by ProgrammableWeb (ProgrammableWeb, 2013), a directory which documentes 10,555 APIs and in which a vast majority (around 69%) are REST based WS.

So the existing standards aiming at a better automation of WS management are not really used and widely spread yet. It thus seems important to investigate a semi-automated approach to build a WS fed data warehouse, keeping the requirement of reducing the code burden needed to maintain such a system.

Unlike the traditional WS discovery issue, in the DaWeS context, the services discovery does not really need to be automated since it's up to the user to inform the system about the services he's using. So, the real automation problem resides in the

Tableau 1. *Web Service API Analysis on three domains*

Project Management	**Basecamp**	**LiquidPlanner**	**Teamwork**	**Zoho Projects**
1. API Description	HTML page	HTML page	HTML page	HTML page
2. Conform to REST	REST like	REST like	REST like	Not REST
3. Version	v1	3.0.0	N.A.	N.A.
4. Authentication	Basic HTTP, OAuth 2	Basic HTTP	Basic HTTP	Basic HTTP
5. Resources Involved	Project,Todo List, Todo	Project, Task	Project, Task List, Task	Project, Task List, Task
6. Message Formats	JSON	JSON	XML, JSON	XML, JSON
7. Service Level Agreement	Max 500 requests /10s from same IP address for same account	Max 30 requests /15s for same account	Max 120 requests /1min	Error code :6403 on exceeding the limit
8. HTTP Access	GET	GET	GET	POST
9. Data Types (dt)	Enumerated dt (Project and Todo Status), Date	Enumerated dt (Project and Task Status), Date	Enumerated dt (Project and Task Status), Date	Enumerated dt (Project and Task Status), Date
10. Dynamic nature of the resources	Yes (Project and Task Status)	Yes (Project and Task Status)	Yes (Project and Task Status)	Yes(Project and Task Status)
11. Operation Invocation	Sequence Required	Sequence Not Required	Sequence Required	Sequence Required
12. Pagination	No	No	No	Yes

Email Marketing	**Mailchimp**	**CampaignMonitor**	**iContact**
1. API Description	HTML page	HTML page	HTML page
2. Conform to REST	Not REST	REST like	REST like
3. Version	1.3	v3	2.2
4. Authentication	Basic HTTP	Basic HTTP, OAuth 2	Basic HTTP (with Sandbox)
5. Resources Involved	Campaign, Campaign Statistics	Campaign, Campaign Statistics	Campaign, Campaign Statistics
6. Message Formats	XML, JSON, PHP, Lolcode	XML, JSON	XML, JSON
7. Service Level Agreement	N.A.	N.A.	75,000 requests /24h, with a max of 10,000 requests /1h
8. HTTP Access	GET	GET	GET
9. Data Types (dt)	Enumerated Data types (Campaign Status), Date	Enumerated Data types (Campaign Status), Date	Enumerated Data types (Campaign Status), Date
10. Dynamic nature of the resources	Yes (Campaign Status)	Yes (Campaign Status)	Yes (Campaign Status)
11. Operation Invocation	Sequence Required	Sequence Required	Sequence Not Required
12. Pagination	Yes	No	No

Support	**Zendesk**	**Desk**	**Zoho Support**	**Uservoice**	**Freshdesk**
1. API Description	HTML page	HTML page	HTML page	HTML page	HTML page
2. Conform to REST	REST like	REST	Not REST	REST like	REST like
3. Version	v1	v2	N.A.	v1	N.A.
4. Authentication	Basic HTTP	Basic HTTP, OAuth 1.0a	Basic HTTP	OAuth 1.0	Basic HTTP
5. Resources Involved	Forum, Topic, Ticket	Case	Task	Forum, Topic, Ticket	Forum, Topic, Ticket
6. Message Formats	XML, JSON	JSON	XML, JSON	XML, JSON	JSON
7. Service Level Agreement	Limit exists (but unknown)	60 requests per minute	250 calls /day /org (Free)	N.A.	N.A.
8. HTTP Access	GET	GET	GET	GET	GET
9. Data Types (dt)	Enumerated dt (Ticket Status), Date	Enumerated dt (Case Status), Date	Enumerated dt (Task Status), Date	Enumerated dt (Ticket Status), Date	Enumerated dt (Ticket Status), Date
10. Dynamic resources	Yes (Ticket Status)	Yes (Case Status)	Yes (Task Status)	Yes (Ticket Status)	Yes (Ticket Status)
11. Operation Invocation	Sequence Required	Sequence Required	Sequence Required	Sequence Required	Sequence Required
12. Pagination	Yes	Yes	Yes	Yes	No

automated connection between the warehouse and the known web services that will be its data sources. WSDL is typically a technology that is meant for enabling the automated generation of a wrapper between the system and a WS. But even if we could use such standard, it wouldn't solve the entire problem. Indeed, having a wrapper allows to call the WS. But it does not link the semantics of what the service can do (e.g. what kind of data it can provide) to the semantics of the system which is the one the user knows (typically given by the schema of the warehouse).

The solution we describe in the next section is to manually achieve the connection between DaWeS and WS in a twofold manner : (i) dedicating the greatest part of the manual effort to establish the semantic connection between data in DaWeS and data coming from the WS, and (ii) trying to reduce the daily coding effort to deal with syntactic mismatches. (i) will be obtained via a mediation approach, and (ii) via the building of a generic wrapper and the use of declarative languages only for each manual task.

3. Mediation as ETL

In the data integration field, mediation (Wiederhold, 1992) is the main virtual approach to provide a uniform query interface to multiple heterogeneous and autonomous data sources. Every source has its own (local) schema linked via mappings to the mediated or global schema (i.e., the schema of the mediator) over which the user queries are formulated. The approach is virtual because the global schema does not contain any data. After the user has posed her query over the global schema, this query is reformulated into a set of queries such that each of them can be posed over a special source. After each source has given its results, these are merged into the mediator and presented to the user. Among the different kind of mappings between the local and the global schema (see (Chawathe *et al.*, 1994 ; Duschka et Genesereth, 1997 ; Ullman, 2000 ; Friedman *et al.*, 1999)), the Local As View (LAV) mappings are known to allow easy addition, update and removal of sources (Ullman, 2000). Indeed, adding, updating and removing a LAV mapping can be done without modifying anything else than the mapping itself. This is due to the fact that a LAV mapping is defined as a query over the global schema. Another consequence is that defining a mapping is done using a declarative query language, and not through the programming of a piece of software, which is easier, quicker and less constraining.

Thus, the mediation with LAV mappings approach fits particularly well our need to easily (thus manually) connect to multiple and heterogeneous WS. It becomes the ETL (extraction-transformation-loading) tool of our data warehouse. This implies WS are considered as relational data sources. Since the access to WS is constrained by precise input values, to get output data, then WS must be considered as relational data sources with access patterns that specify which attributes are inputs and which are outputs. More precisely, each operation provided by a WS is modeled as a relation associated to an access pattern (Ullman, 1989) whose size is equal to the number of attributes in a relation. Syntactically, the access pattern is represented by an adornment

Tableau 2. *Helpdesk WS and their operations*

Service	Operation name	Inputs	Outputs
Desk v2 API	Deskv2TotalCases (D2TC)	None	Total nb of tickets : $pgno, pgsize$
	Deskv2Case (D2C)	$pgno$, $pgsize$	One page tickets details : $tkid$, tkn, $tkcd$, tkp, tks
Zendesk v1 API	Zendeskv1Ticket (ZT)	None	All ticket id : $tkid$
	Zendeskv1SolvedTicket (ZST)	None	All closed tickets id : $tkid$
	Zendeskv1TicketDetails (ZTD)	$tkid$	One ticket details : tkn, $tkcd$, $tkdd$, $tkcmpd$, tkp, tks
Uservoice v1 API	Uservoicev1TotalTickets (UTT)	None	Total nb of tickets : $pgno, pgsize$
	Uservoicev1Ticket (UT)	pgn, pgs	One page tickets details : id, tkn, $tkcd$, tkp, tks

being a tuple of b and f letters written besides the relation name. In this tuple, b (for "bound") in ith position says the ith attribute is an input ; f (for "free") says it is an output.

Example 1. : Let us consider three WS in the helpdesk domain : Zendesk, Uservoice and Desk. They allow customers to submit their complaints. These are tracked by tickets. Every ticket has an associated priority and status. Some need immediate attention and therefore have high priority. When a ticket is created, its status is open and when resolved, its status is completed (or closed).

Here are attribute names given to ticket related information. A page is an answer of an API call. $pgno$ is a page number, $pgsize$ is a number of tickets in one page, $limit$ is a number of results in a page, $tkid$ is a ticket identifier, tkn is a ticket name, $tkcd$ is a ticket creation date, $tkdd$ is a ticket due date, $tkcmpd$ is a ticket effective completion date, tkp is a ticket priority and tks is a ticket current status. src is a WS name, and $operation$ is an operation name.

We want DaWeS to be connected to these services so that customers can get performance indicators about the handling of their complaints. Towards this purpose, each WS offers at least one operation callable through its API (see table 1). For these services to be connected to DaWeS, the global schema must contain relations that describe the domain. Here are the two relations extracted from the global schema that describe everything linked to the notion of ticket :

$\mathbf{Ticket}(tkid, src, tkname, tkcdate, tkddate, tkcmpdate, tkpriority, tkstatus)$

$\mathbf{Page}(pgno, src, operation, limit)$

Now, the following queries define the LAV mappings between each operation and the global schema (these are conjunctive queries written in the rule-style syntax) :

$\mathbf{D2TC}^{ff}(pgno, pgsize) \leftarrow \mathbf{Page}(pgno, '\text{ Desk v2 API}', '\text{ Deskv2Case}', pgsize).$

$\mathbf{D2C}^{bbffffff}(pgno, pgsize, tkid, tkn, tkcd, tkp, tks) \leftarrow$
$\mathbf{Page}(pgno, '\text{ Desk v2 API}', '\text{ Deskv2Case}', pgsize),$
$\mathbf{Ticket}(tkid, '\text{ Desk v2 API}', tkn, tkcd, tkdd, tkcmpd, tkp, tks).$

$\mathbf{ZT}^{f}(tkid) \leftarrow \mathbf{Ticket}(tkid, '\text{ Zendesk v1 API}', tkn, tkcd, tkdd, tkcmpd, tkp, tks).$

$\mathbf{ZST}^{f}(tkid) \leftarrow \mathbf{Ticket}(tkid,'\text{ Zendesk v1 API}', tkn, tkcd, tkdd, tkcmpd, tkp,'\text{ Closed}').$

$\mathbf{ZTD}^{bfffffff}(tkid, tkn, tkcd, tkdd, tkcmpd, tkp, tks) \leftarrow$
$\mathbf{Ticket}(tkid,'\text{ Zendesk v1 API}', tkn, tkcd, tkdd, tkcmpd, tkp, tks).$

$\mathbf{UTT}^{ff}(pgno, pgsize) \leftarrow \mathbf{Page}(pgno,'\text{ Uservoice v1 API}','\text{ Uservoicev1Ticket}', pgsize).$

$\mathbf{UT}^{bbffffff}(pgn, pgs, id, tkn, tkcd, tks, tkp) \leftarrow$
$\mathbf{Page}(pgn,'\text{ Uservoice v1 API}','\text{ Uservoicev1Ticket}', pgs),$
$\mathbf{Ticket}(id,'\text{ Uservoice v1 API}', tkn, tkcd, tkdd, tkcmpd, tkp, tks).$

Once the global schema and the LAV mappings are built, the user is able to pose her query over the global schema without dealing with the source relations intricacies. The query will then be automatically transformed by a query rewriting algorithm into a query plan, which describes the sequence of API operation calls (from potentially different WS) needed to answer the user query.

The classical query rewriting algorithms include bucket algorithm (Levy *et al.*, 1996), inverse rules algorithm (Duschka et Genesereth, 1997 ; Duschka *et al.*, 2000) and minicon algorithm (Pottinger et Levy, 2000). They compute the so-called maximally contained rewritings which allow to obtain the certain answers of a query. Informally, this means that the computed answers are not false, at the computation time, in every source. So, for example, if two services deliver different status for the same ticket id, these status does not belong to the certain answers and will not be presented in the query result. In DaWeS, we chose inverse rules algorithm (Duschka et Genesereth, 1997 ; Duschka *et al.*, 2000) since it can handle access patterns in the data sources description and is the only algorithm (up to our knowledge) being able to rewrite datalog recursive queries posed to the global schema (for conjunctive queries as LAV mappings). Moreover it is shown in (Abiteboul et Duschka, 1998) that generating a query plan can be done in polynomial time with respect to the data complexity (i.e., in the sizes of the query and the mappings). This ensures, at least theoretically, quite good performances. Besides, various integrity constraints on the global schema like full and functional dependencies can also be handled by this algorithm.

Example 2. In the context of example 1, we now consider a record definition. Note that we use a special function here called *yesterday()*, which is executed before the query evaluation, to obtain yesterday's date. The record we define is called Daily New Tickets (DNT) : it is the number of tickets that were created yesterday.

$\mathbf{DNT}(tkid, src, tkn, tkp, tks) \leftarrow$
$\mathbf{Ticket}(tkid, src, tkn,' yesterday()', tkdd, tkcmpd, tkp, tks).$

The following program is the query plan which is the rewriting of query DNT.

$\mathbf{Page}(pgno,'\text{ Desk v2 API}','\text{ Deskv2Case}', pgsize) \leftarrow \mathbf{D2TC}^{ff}(pgno, pgsize).$
$\mathbf{DPgNo}(pgno) \leftarrow \mathbf{D2TC}^{ff}(pgno, pgsize).$
$\mathbf{DPgSize}(pgsize) \leftarrow \mathbf{D2TC}^{ff}(pgno, pgsize).$
$\mathbf{Page}(pgno,'\text{ Desk v2 API}','\text{ Deskv2Case}', pgsize) \leftarrow$
$\mathbf{DPgNo}(pgno), \mathbf{DPgNo}(pgsize), \mathbf{D2C}^{bbffffff}(pgno, pgsize, tkid, tkn, tkcd, tkp, tks)$
$\mathbf{Ticket}(tkid,'\text{ Desk v2 API}', tkn, tkcd, f_{D2C,5}(pgno, pgsize, tkid, tkn, tkcd, tkp, tks),$
$f_{D2C,6}(pgno, pgsize, tkid, tkn, tkcd, tkp, tks), tkp, tks) \leftarrow$

$\mathbf{DPgNo}(pgno), \mathbf{DPgSize}(pgsize), \mathbf{D2C}^{bbffffff}(pgno, pgsize, tkid, tkn, tkcd, tkp, tks)$
$\mathbf{Ticket}(tkid,'\text{ Zendesk v1 API}', f_{ZT,3}(tkid), f_{ZT,4}(tkid), f_{ZT,5}(tkid),$
$f_{ZT,6}(tkid), f_{ZT,7}(tkid), f_{ZT,8}(tkid)) \leftarrow \mathbf{ZT}^f(tkid).$
$\mathbf{Ticket}(tkid,'\text{ Zendesk v1 API}', f_{ZST,3}(tkid), f_{ZST,4}(tkid), f_{ZST,5}(tkid), f_{ZST,6}(tkid),$
$f_{ZST,7}(tkid), \text{Closed}') \leftarrow \mathbf{ZST}^f(tkid).$
$\mathbf{ZTID}(tkid) \leftarrow \mathbf{ZST}^f(tkid).$
$\mathbf{Ticket}(tkid,'\text{ Zendesk v1 API}', tkn, tkcd, tkdd, tkcmpd, tkp, tks) \leftarrow$
$\mathbf{ZTID}(tkid), \mathbf{ZTD}^{bfffffff}(tkid, tkn, tkcd, tkdd, tkcmpd, tkp, tks).$
$\mathbf{Page}(pgno,'\text{ Uservoice v1 API}','\text{ Uservoicev1Ticket}', pgsize) \leftarrow \mathbf{UTT}^{ff}(pgno, pgsize).$
$\mathbf{UPgNo}(pgno) \leftarrow \mathbf{UTT}^{ff}(pgno, pgsize).$
$\mathbf{UPgSize}(pgsize) \leftarrow \mathbf{UTT}^{ff}(pgno, pgsize).$
$\mathbf{Page}(pgn,'\text{ Uservoice v1 API}','\text{ Uservoicev1Ticket}', pgs) \leftarrow$
$\mathbf{UPgNo}(pgno), \mathbf{UPgSize}(pgsize), \mathbf{UT}^{bbffffff}(pgn, pgs, id, tkn, tkcd, tks, tkp)$
$\mathbf{Ticket}(id,'\text{ Uservoice v1 API}', tkn, tkcd, f_{UT,5}(pgn, pgs, id, tkn, tkcd, tks, tkp),$
$tkcmpd, tkp, tks) \leftarrow$
$\mathbf{UPgNo}(pgno), \mathbf{UPgSize}(pgsize), \mathbf{UT}^{bbffffff}(pgn, pgs, id, tkn, tkcd, tks, tkp).$

This rewriting is a bit long, but we emphasize the fact that it is a real case which is described here. Now, from the previous records DNT, we can define with SQL queries the following performance indicators : *Total New Tickets Registered in a month*, *Total High Priority Tickets Registered in a month* and *Percentage of High Priority Tickets Registered in a month*. For example the performance indicator *Total High Priority Tickets Registered in a month* definition will be :
SELECT count(tkid) FROM DNT WHERE tkcdate < sysdate and tkcate > sysdate - interval '30' day AND tkpriority='High' ;

In DaWeS, user queries are performance indicators definitions. We have chosen to distinguish two kinds of performance indicators : basic ones, called records, and complex ones, called performance indicators. Indeed, even if the inverse-rules algorithm enables the user to pose recursive datalog queries, the rewriting process is not able to deal with any aggregation function, which are mandatory to define interesting performance indicators. So the idea is first to use mediation with the rewriting process to get data from WS, then to materialize these data in the database of DaWeS, and at last to query these data to generate performance indicators. Records are user queries posed over the virtual global schema, that are rewritten during mediation to query WS and to fetch their data. Performance indicators are user queries posed over the materialized schema built with the record relations. Record queries are datalog queries. Performance queries are full SQL queries, extensible to all possible OLAP operators. Since records are materialized and business performance are computed from them, these can be updated easily when new data for the underlying records are fetched. This two layers query architecture is really interesting since it allows a user to easily change a service provider while still being able to compute her performance indicators with her full dataset (including the old data from the previous providers).

4. DaWeS architecture

The basic underlying architecture of DaWeS is shown in Figure 1. The ETL part of DaWeS is made up with three components : the query rewriter, the answer builder and the generic HTTP WS wrapper. The storage of DaWeS is organized in four schemas : the global schema (virtual), the API schema (virtual), the record schema (materialized) and the performance indicator schema (materialized). The last part is the query evaluator which is given by the underlying DBMS.

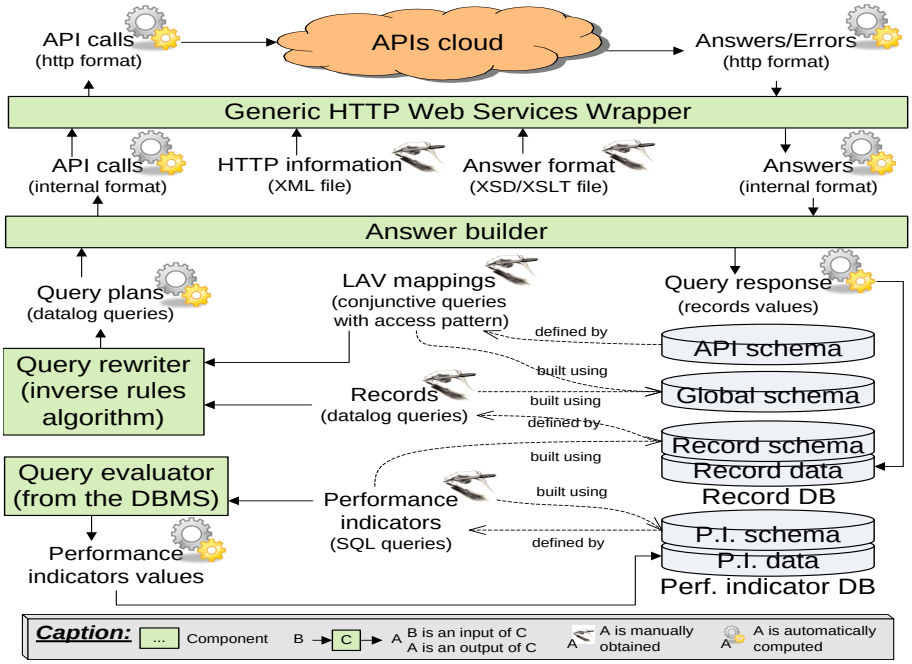

Figure 1. *DaWeS : Basic Architecture*

When a DaWeS administrator wants to add a new WS API operation, he manually gathers all HTTP information (url, authentication, ...) and all operation response formats details (to make the link with DaWeS data formats). Then he manually defines the LAV mapping and the record the operation will provide, both as queries over the global schema. The name and query of each LAV mapping (resp. record) are stored in the API schema (resp. record schema). After this, a DaWeS user can define her own performance indicator by a query over the record and/or the performance indicator schemas. The name and query of the indicator are stored in the performance indicator schema. Here, we emphasize the fact that no human intervention is devoted to programming in a high level (procedural) language (e.g. Java). Instead, only declarative languages are used here.

The automated process can then be executed. First, the query plan is computed by the inverse rule algorithm in the query rewriter from the record definition and the LAV mappings. Then the answer builder, which consists of a datalog query engine,

executes the query plan. It sends to the generic wrapper the API calls in an internal format, which consists of the atoms of the query plan, along with the authentication and input parameters. The generic HTTP WS wrapper is then used to make the WS API operation calls and transform the response in a manner understood by the answer builder. The answer builder combine these answers to get the record values which are stored in the record database. The performance indicator can at last be computed by the underlying DBMS.

The generic HTTP WS wrapper is the component dedicated to effectively execute the query plan. It is generic in the sense that it can call any API operation, provided that some parameter values are given. These values are the HTTP information (URL, method, header and body contents, authentication mechanism) and the answer format (the XSD schema of the response and XSLT translation to transform the API data formats to DaWeS desired data formats). In case of JSON message format, we translate it first to XML and then get its XSD and XSLT. As seen before, all these informations must be manually gathered when the service operations are modeled with respect to the global schema. But after this, the wrapper is able to automatically perform the right API operation call, and get back the results to DaWeS. Internally the wrapper consists of a response validator (using XSD) and a response tranformator (using XSLT). We also used cache in the wrapper in order to reduce the number of calls (the recently made operation response is cached for future use). When the answer builder requests for making an API call to the generic wrapper, the wrapper checks the cache whether the response is available and if available returns this cached response. Else the wrapper frames the HTTP header, method, body and URL for framing the API call with the input parameter values given by the answer builder. Once the response is obtained, the response validator ensures that the response schema is the same as that was registered before (else, it's a sign of possible API/operation level change). If the response is valid, the transformator makes use of the XSLT to transform the response to the desired format. This response is cached and then returned to the answer builder.

A special feature of DaWeS is that the warehouse schema, made up with the record and performance indicator schemas, is dynamic : the set of its relations will evolve in course of time, since we can add or delete new records or indicators. To handle this, we have followed a simple approach which consists of making use of only two big tables to store both the warehouse schema and its associated record and performance indicator data. This can be viewed as an extra logical layer between the schema exposed to the user and the implementation of the DBMS. It implies some small computation overhead, but ensures our warehouse schema can evolve transparently for the user.

DaWeS was tested with Intel(R) Pentium(R) Dual CPU @ 2.16GHz processor, a system memory of 3GiB and Ubuntu 13.04 (32 bits) operating system. We used Oracle 11g (11.2.0.1.0) as the database. DaWeS was developed and run using Java 1.7.0_25. We chose IRIS (Integrated Rule Inference System) (IRIS, 2008) as the datalog engine to perform query evaluation and configured it to make use of the generic HTTP WS wrapper during query evaluation in the answer builder. We chose IRIS considering its capability to handle adornments (to specify access patterns in the relations), func-

Tableau 3. *Characteristics of the Qualitative Tests*

Characteristics	Description
1. Nb of domain of WS	3
2. Nb of Web services considered	12
3. Nb of API operations considered	35 (Operations, so 35 LAV Mappings)
4. Nb of Global Schema Relations	12
5. Nb of Test Organizations	100 (homogeneous test organizations)
6. Message Formats	XML, JSON
7. Authentication Mechanisms	HTTP basic authentication, OAuth 1.0
8. Operation details	No, one or more input parameters ; pagination
9. Nb of Record Definitions	17
10. Nb of Perf. Indicator Queries	20
11. Data types	Strings, Integers, Dates, Enumerated data types

Tableau 4. *Record Definitions*

Project Management	Support (Helpdesk)	Email Marketing
Daily New Projects(1), Daily Active Projects(2), Daily OnHold Projects(3), Daily OnHold or Archived Projects(4), Daily New Tasks(5), Daily Open Tasks(6), Daily Closed Tasks(7), Daily TodoLists(8), Daily Same Status Projects(9)	Daily New Forums(10), Daily All Forums(11), Daily New Topics(12), Daily New Tickets(13), Daily Open Tickets(14), Daily Closed Tickets(15)	Daily New Campaigns(16) and Daily Campaign Statistics(17)

tional symbols (generated with the inverse rules query rewriting), built-in predicates (like equality predicate EQUAL, useful to handle functional dependencies) and the capability to refer external sources during query evaluation.

5. Experiments

We performed various qualitative and quantitative tests on DaWeS. The first step was to create realistic business-like data : we used the web interfaces of each tested service as if we were a company using it. Then we run qualitative test followed by quantitative ones. Qualitative tests aim at checking if the process computes what it is expected to. The characteristics of the tests undertaken by us is summarized in the table 5. We considered the record definitions in Table 5 created similarly as in example 2. The performance indicators considered by us are given in the Table 5.

Grounding our tests on data we had given to the WS via their associated web sites enabled us to easily check if the records and performance indicators computations were right, which was the case. Moreover, the generic wrapper was able to make API calls to any web service. Of course, these results heavily depend on the precise modeling of LAV mappings, HTTP information and answer formats. For example, if the domains of attributes, in predicates with access patterns, are not distinguished accor-

Tableau 5. *Performance Indicator Queries*

Project Management	Support (Helpdesk)	Email Marketing
Total Monthly New Projects, Total Monthly Active Projects, Total Monthly OnHold Projects, Total Monthly Completed Tasks, Average Tasks Completed Daily in a month, Total Monthly New Tasks, Total Todo Lists, Percentage of tasks completed to tasks created in a day	Daily Average Resolution Time, Total New Tickets Registered in a month, Total New Forums Registered in a month, All Forums in a month, Total New Topics Registered in a month, Total High Priority Tickets Registered in a month, Percentage of High Priority Tickets Registered in a month	Total Monthly New Campaigns, Monthly Click Throughs of Campaign, Monthly Forwards of Campaign, Monthly Bounces of Campaign, Total Monthly Solved Tickets

ding to the WS they refer to, then the query plan may imply uninteresting answers. This is the case when output data from operation 1 from service 1 are taken as input for operation 2 from service 2, just because they have the same domain of values. In example 2, if we used only the domains (PgNo, PgSize) instead of (DPgNo, DPgSize, UPgNo, UPgSize), the output page numbers and page sizes from UTT may become input to D2C resulting in unexpected answers. Secondly, the output of UTT and D2C is actually the total number of tickets and cases respectively, these have been transformed to (pgno, pgsize) combination using XSLT. Therefore it requires certain human effort to deal with such cases, where direct composition may not work. Thirdly XSD aren't usually provided in the service documentation and only example XML (JSON) responses are available for reference. It takes additional effort to create and validate XSD from the examples since often there may be discrepancies (extra or missing XML elements or attributes) in the results obtained by actual API calls and the given XML examples. Overall, the time taken to manually read the HTML documentation to declaratively describe the API is 3-4 hours.

Concerning the quantitative tests, figure 2 shows the time taken to compute the records given in table 5. Since our performance results highly depend on network communication times, the computation times were each measured 100 times so as to obtain an average time where the influence of network traffic peaks is limited. In figure 2, the total (average) time of fetching the 17 records was 104.82 seconds. Each of the 17 times are also average times. We can see that the cache implemented in the generic wrapper has a real impact on performances. Indeed, some records (2,3,4,6,7,8,9,11,14,15,17) can use the cached data from other ones (1,5,10,13,16). For example, record 14 can used the cached data of record 13 because the ticket details have already been fetched from the various services during the query evaluation of 13, transformed and cached, the transformed WS responses can as well be used for record 14. That's why we observe on the first chart in figure 2 high values for records 1, 5, 10, 12, 13, 16, and low values for the others. This point is checked also in the second chart which shows the number of API operation calls made during the query evaluation for every record definition. It shows how cache performs optimization by

avoiding the repetition of calls.

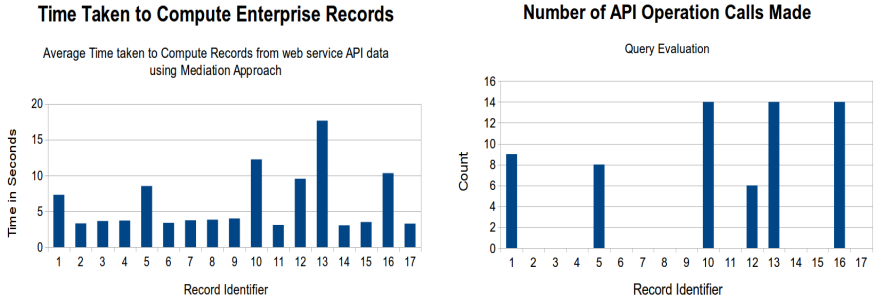

Figure 2. *DaWeS : average times to fetch record data and number of API calls.*

6. Related Works

In DaWeS, 3 important issues are handled : (i) mediation with WS as sources, (ii) generic wrapping to WS, and, (iii) mediation as an ETL tool to feed a data warehouse.

Considering (i), an important approach for performing data integration using WS is ActiveXML (Abiteboul *et al.*, 2002 ; Salem *et al.*, 2013), a language that extends XML to allow the embedding of WS calls. We think that ActiveXML could be a possible extension to DaWeS. The generic wrapper could be extended to create ActiveXML documents so that we can connect to ActiveXML services. In such an extension, we would just use the intentional part of ActiveXML documents, since using the extensional part means storing data besides calls, what we do not need (nor want) to do. So using ActiveXML in this way would be somehow contradictory with one of its main objectives which is to develop a dynamic and powerful data oriented scheme for distributed computation (e.g. peer-to-peer data integration). Indeed, DaWeS is a centralized system, since it is in fine a data warehouse. Other approaches related to (i) and closer to DaWeS than ActiveXML are surveyed in table 6.

About (ii), (Benatallah *et al.*, 2005 ; van den Heuvel *et al.*, 2007) have discussed configurable adapters (wrappers) before to deal with WS replacement and evolution. Compared to DaWeS, their main drawback is they are dependent on (so restricted to) the use of various business standards like BPEL. DaWeS is aimed at working with what is actually used to expose WS API. So focusing on one particular standard, at the exclusion of the others, is not the aim of DaWeS. DaWeS requires manual intervention for translating human readable (HTML web pages) interface description to a desired internal format (LAV Mapping, XSD and XSLT for every WS API operation). Several machine readable interface descriptions to syntactically and semantic describe the WS API operations like WSDL (W3C, 2001), WADL (Hadley, 2006) and hRESTS (Kopecký *et al.*, 2008) are useful for automatic code generation of the wrapper. An industry wide acceptance of these standards is a major concern. A generic wrapper in our context of integrating with numerous WS is easier to manage than having a wrapper for every web service. Also the generic wrapper takes as input a declarative approach to WS, making it furthermore easier to manage.

Tableau 6. *Data Integration and Web Services : State of the Art*

Characteristics	a. DaWeS	b. (Zhu *et al.*, 2004)	c. (Benslimane *et al.*, 2008)	d. (Thakkar *et al.*, 2003)	e. (Barhamgi *et al.*, 2008)
1. Primary aim	Building Data warehouse fed with WS	Large scale data integration from autonomous organizations	Mashups or Composition of two or more WS to generate new service	Mediator As a Web Service Generator	Automatic composition of data providing WS
2. Primary Targeted audience	Business enterprises using WS	Health services	Internet users	Service providers and internet users	Bioinformatics and healthcare systems
3. Underlying mechanism	Mediation approach (query rewriting)	Federated Database System	Web service composition using automated or graphical composition tools	Mediation approach (query rewriting)	Mediation approach (query rewriting)
4. Use of standards	HTTP, XML, JSON, XSD and XSLT	WSDL, UDDI, XML, DAML-S	XML, JSON, HTTP	XML, SOAP	RDF, SPARQL
5. API Operations Handled	Resource access	Resource access	Resource access and manipulation	Resource access	Resource Access
6. Algorithms used	Inverse Rules algorithm	Federated Query Services (query decomposer and query integrator)	Usually manual intervention to create the composition of services	Modified Inverse Rules algorithm	RDF query rewriting algorithm
7. User Schema	Dynamic warehouse schema	Schema generated on the fly	No schema (not needed)	Global schema	Mediated Ontology

Concerning (iii), we recall from (Trujillo et Luján-Mora, 2003) the main tasks characterizing the conceptual UML model of the ETL process : selection of the sources, transformation of the data from the sources, joining the sources to load the data for a target, finding the target, mapping the data source attributes to the target attributes and loading the data in the target. Clearly, DaWeS closely follows these requirements : the query rewriting algorithm ensures the selection and joining of the sources, the wrapper joins the sources to load the data and uses the XSLT files to perform data transformation in accordance to the target (record) schema, and the query response constitutes the data for the target. As a special feature, DaWeS enables the handling of a dynamic schema, which is transparent for the user (cf section 4). On one side, this is very convenient for the user to be able to quickly define new performance indicator. On the other side, it is less straightforward to applying popular data warehouses storage approaches, like the Inmon approach (Inmon, 1992), based on the use of normalized 3NF tables and the Kimball approach (Kimball, 1996), defining the star schema storage organization. For example, it is not clear yet what is the impact of our dynamic schema on the handling of advanced performance indicators like the CUBE operators (used along with the star schema).

7. Conclusion

The growing use of WS among the enterprises cannot be undermined. Our prototype shows it is possible to build a data warehouse fed with web services which is

aimed towards scalability and adaptability and which can be managed using declarative languages only. Mediation as an ETL approach used along with the generic HTTP WS wrapper demonstrates that the extraction of data from WS is not as complex compared to the various web scraping techniques and wrappers for textual sources and legacy databases, even with basic WS standards.

DaWeS has several other components like calibration and error handling that have not been described here due to space limitations. Records and performance indicators are periodically calibrated using various test data and WS data so as to ensure their accuracy. Calibration is used in conjunction with the error handling (errors like unexpected response format) to detect any (unannounced API change) and to trigger a manual intervention (update LAV Mapping, XSD, XSLT...). We are currently working towards a static optimization on the domain rules to reduce the API operation calls for operations that have functional dependencies on their input attributes . We want to extend DaWeS with additional set of constraints in the form of more tuple generating dependencies (TGD) than just functional dependencies. It will be further extended to a cloud infrastructure reaping the benefits of just two tables for the enterprise data.

Remerciements

We thank the Conseil General of the Region of Auvergne (France) and FEDER for funding our research project. We would also like to thank Franck Martin and Lionel Peyron of Rootsystem for their feedback during the development of DaWeS.

8. Bibliographie

Abiteboul S., Benjelloun O., Manolescu I., Milo T., Weber R., « Active XML Peer-to-Peer Data and Web Services Integration », *VLDB*, 2002, p. 1087-1090.

Abiteboul S., Duschka O. M., « Complexity of Answering Queries Using Materialized Views », *PODS*, 1998, p. 254-263.

Barhamgi M., Benslimane D., Ouksel A. M., « Composing and optimizing data providing web services », *WWW*, ACM, 2008, p. 1141-1142.

Benatallah B., Casati F., Grigori D., Nezhad H. R. M., Toumani F., « Developing Adapters for Web Services Integration », *CAiSE*, vol. 3520, 2005, p. 415-429.

Benslimane D., Dustdar S., Sheth A. P., « Services Mashups : The New Generation of Web Applications », *IEEE Internet Computing*, vol. 12, n° 5, 2008.

Burstein M. H., Hobbs J. R., Lassila O., Martin D., McDermott D. V., McIlraith S. A., Narayanan S., Paolucci M., Payne T. R., Sycara K. P., « DAML-S : Web Service Description for the Semantic Web », *Proceedings of ISWC*, 2002, p. 348–363.

Chawathe S., Garcia-Molina H., Hammer J., Ireland K., Papakonstantinou Y., Ullman J., Widom J., « The TSIMMIS Project Integration of Heterogeneous Information Sources », *Proceedings of IPSJ Conference*, 1994, p. 7–18.

Duschka O. M., Genesereth M. R., Levy A. Y., « Recursive Query Plans for Data Integration »,

J. Log. Program., vol. 43, n° 1, 2000, p. 49-73.

Duschka O. M., Genesereth M. R., « Answering Recursive Queries Using Views », *PODS*, 1997.

Fielding R. T., « Architectural Styles and the Design of Network-based Software Architectures », 2000.

Friedman M., Levy A. Y., Millstein T. D., « Navigational Plans For Data Integration », *AAAI/IAAI*, AAAI Press / The MIT Press, 1999.

Hadley M. J., « Web Application Description Language (WADL) », rapport, 2006, Sun Microsystems, Inc., Mountain View, CA, USA.

Inmon W. H., *Building the Data Warehouse*, John Wiley & Sons., New York, NY, USA, 1992.

IRIS, « Integrated Rule Inference System - API and User Guide », 2008.

Kimball R., *The Data Warehouse Toolkit Practical Techniques for Building Dimensional Data Warehouses*, John Wiley, 1996.

Kopecký J., Vitvar T., Bournez C., Farrell J., « SAWSDL Semantic Annotations for WSDL and XML Schema », *IEEE Internet Computing*, vol. 11, n° 6, 2007.

Kopecký J., Gomadam K., Vitvar T., « hRESTS An HTML Microformat for Describing RESTful Web Services », *Proceedings of the IEEE/WIC/ACM*, WI-IAT '08, IEEE Computer Society, 2008, p. 619–625.

Levy A. Y., Rajaraman A., Ordille J. J., « Query-Answering Algorithms for Information Agents », *AAAI/IAAI, Vol. 1*, AAAI Press / The MIT Press, 1996, p. 40-47.

Martin D., Paolucci M., Wagner M., « Bringing Semantic Annotations to Web Services OWL-S from the SAWSDL Perspective », *ISWC/ASWC*, 2007, p. 340–352.

Pottinger R., Levy A. Y., « A Scalable Algorithm for Answering Queries Using Views », *VLDB*, 2000, p. 484-495.

ProgrammableWeb, « http//www.programmableweb.com », December 2013.

Salem R., Boussaïd O., Darmont J., « Active XML-based Web Data Integration », *Information Systems Frontiers*, vol. 15, n° 3, 2013, p. 371–398.

Thakkar S., Knoblock C. A., Ambite J. L., « A View Integration Approach to Dynamic Composition of Web Services », *ICAPS Workshop on Planning for Web Services*, 2003.

Trujillo J., Luján-Mora S., « A UML Based Approach for Modeling ETL Processes in Data Warehouses », *ER*, vol. 2813, 2003, p. 307-320, Springer.

Ullman J. D., *Principles of Database and Knowledge-Base Systems, Volume II*, Computer Science Press, 1989.

Ullman J. D., « Information integration using logical views », *Theor. Comput. Sci.*, vol. 239, n° 2, 2000, p. 189-210.

van den Heuvel W.-J., Weigand H., Hiel M., « Configurable adapters the substrate of self-adaptive web services », *Proceedings of ICEC*, ACM, 2007.

W3C, « Web Service Description Language 1.1 », 2001.

Wiederhold G., « Mediators in the Architecture of Future Information Systems », *Computer*, vol. 25, n° 3, 1992, p. 38–49, IEEE Computer Society Press.

Zhu F., Turner M., Kotsiopoulos I. A., Bennett K. H., Russell M., Budgen D., Brereton P., Keane J. A., Layzell P. J., Rigby M., Xu J., « Dynamic Data Integration Using Web Services », *ICWS*, IEEE Computer Society, 2004, p. 262-269.