Feeding a data warehouse with data coming from web services. A mediation approach for the DaWeS prototype

John Samuel

N° d'ordre : D.U: 2493
EDSPIC : 669

Université Blaise Pascal - Clermont II
ECOLE DOCTORALE
SCIENCES POUR L'INGENIEUR DE CLERMONT-FERRAND

THÈSE

Presentée par

John SAMUEL

pour obtenir le grade de

Docteur d'Université

Spécialité : Informatique

Feeding a Data Warehouse with Data coming from Web Services. A Mediation Approach for the DaWeS prototype.

Thèse dirigée par Farouk Toumani, Christophe Rey

préparée au LIMOS - UMR CNRS 6158

soutenue publiquement le

6 Octobre, 2014

Devant le jury:

Rapporteurs :	Pr. Omar Boucelma (Président du jury)	Aix-Marseille Université
	Pr. Jérôme Darmont	Université Lumière Lyon 2
Examinateurs:	Dr. Emmanuel Coquery	Université Claude Bernard Lyon 1
	Pr. Farouk Toumani (Directeur de thèse)	Université Blaise Pascal
	Dr. Christophe Rey (Co-directeur de thèse)	Université Blaise Pascal
Invité:	M. Franck Martin	Rootsystem, Vichy

N° d'ordre : D.U: 2493
EDSPIC : 669

Feeding a Data Warehouse with Data coming from Web Services. A Mediation Approach for the DaWeS prototype.

by

John SAMUEL

A Thesis

submitted to the Graduate School of Engineering Sciences

of the Blaise Pascal University - Clermont II

in fulfilment to the requirements for the degree of

Doctor of Philosophy
in Computer Science

Supervised by Farouk Toumani, Christophe Rey

at the LIMOS laboratory - UMR CNRS 6158

publicly defended on

October 6, 2014

Before the jury:

Reviewers :	Pr. Omar Boucelma (President of jury)	Aix-Marseille Université
	Pr. Jérôme Darmont	Université Lumière Lyon 2
Examinators:	Dr. Emmanuel Coquery	Université Claude Bernard Lyon 1
	Pr. Farouk Toumani (Directeur of thesis)	Université Blaise Pascal
	Dr. Christophe Rey (Co-directeur de thesis)	Université Blaise Pascal
Invitee	M. Franck Martin	Rootsystem, Vichy

Abstract

The role of data warehouse for business analytics cannot be undermined for any enterprise, irrespective of its size. But the growing dependence on web services has resulted in a situation where the enterprise data is managed by multiple autonomous and heterogeneous service providers. We present our approach and its associated prototype DaWeS [Samuel, 2014; Samuel and Rey, 2014; Samuel et al., 2014], a DAta warehouse fed with data coming from WEb Services to extract, transform and store enterprise data from web services and to build performance indicators from them (stored enterprise data) hiding from the end users the heterogeneity of the numerous underlying web services. Its ETL process is grounded on a mediation approach usually used in data integration. This enables DaWeS (i) to be fully configurable in a declarative manner only (XML, XSLT, SQL, datalog) and (ii) to make part of the warehouse schema dynamic so it can be easily updated. (i) and (ii) allow DaWeS managers to shift from development to administration when they want to connect to new web services or to update the APIs (Application programming interfaces) of already connected ones. The aim is to make DaWeS scalable and adaptable to smoothly face the ever-changing and growing web services offer. We point out the fact that this also enables DaWeS to be used with the vast majority of actual web service interfaces defined with basic technologies only (HTTP, REST, XML and JSON) and not with more advanced standards (WSDL, WADL, hRESTS or SAWSDL) since these more advanced standards are not widely used yet to describe real web services. 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 (as SQL Queries) of personalized performance indicators. We present in detail this mediation approach for ETL and the architecture of DaWeS.

Besides its industrial purpose, working on building DaWeS brought forth further scientific challenges like the need for optimizing the number of web service API opera-

tion calls or handling incomplete information. We propose a bound on the number of calls to web services. This bound is a tool to compare future optimization techniques. We also present a heuristics to handle incomplete information.

Keywords Web Services, Application Programming Interface (API), Data Integration, Incomplete Information, Datalog Query, Limited Access Patterns, Adaptability, Data Warehouse, Conjunctive Query

Acknowledgement

First of all, I would like to thank the Region of Auvergne and FEDER for funding our research project. I am extremely grateful to Christophe Rey and Farouk Toumani for their guidance and direction. The research work under their direction has helped me to further expand my horizons in the fields of data integration and web services. I am especially thankful to Christophe Rey for the very fruitful day-long discussions on several scientific challenges encountered during the research period. His guidance also enabled me to explore the DaWeS development from various points of view. I am also thankful to Frank Martin and Lionel Peyron of Rootsystem, Vichy, France for their feedback during the development of DaWeS.

I also express my gratitude to all the other members of the LIMOS laboratory, especially Alain Quillot, Béatrice Bourdieu, Martine Caccioppoli and Pascale Gouinaud for their technical and administrative support. I am especially thankful to my colleagues Benjamin Momège, Karima Ennaoui, Lakhdar Akroun and Salsabil Grouche for their active participation in several scientific discussions. I would also like to express my gratitude to various past and current colleagues of LIMOS especially Hicham Reguieg, Jonathan Passerat-Palmbach, Kahina Gani, Li Haizhou, Pierre-Antoine Papon and Yahia Chabane.

I would like to express my sincere gratitude to Mathieu d'Aquin (Knowledge Media Institute, The Open University, United Kingdom) and Steffen Staab (Institute for Web Science and Technologies - WeST, Universität Koblenz-Landau, Germany) for their feedback during 11^{th} ESWC PhD Symposium, Crete, Greece. I would also like to acknowledge various anonymous reviewers of the conference papers and the respective attendees for their reviews and comments. I also acknowledge the reviewers of the thesis report for their feedback.

I would specially like to thank all my past teachers and professors since my childhood for their guidance. I am extremely grateful to Vineeth Paleri and Vinod Pathari

for their inspiration that encouraged me to pursue research and development as a career path.

I am thankful to Anthony for enabling me to be who I am. I am also thankful to my family members especially my parents and grandfather (and my late grandmother) for all their love and care. In particular, I would like to thank my sister Sheeba for her utmost support and being the bridge for communication.

I would like to thank Alice, Amélie, Anne-Lise, Annick, Anthony, Aurélie, Barbara, Betty, Charlotte, Delphine, Elise, Elsa, Elvire, Frédérique, Gegemon, Jisha, Joe, Joel, Matylda, Morgaine, Nicolas, Patrice, Prathaban, Romain, Sophie, Swathy, Tiphaine, Victor and Zélie for their support, love and care.

Last, but not the least, I am thankful to my friends from the animal kingdom: Odin, Lilith and Simone who made everyday special to me.

To
Anthony, for helping me discover myself through lines and colors transcending bits and bytes
Sheeba, my sister for being the pillar of support in all times of need,
Samuel and Leelamma, my parents for implanting in me the desire to explore and fulfill my dreams
and Simone, the cute little cat for being so gentle and sweet.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

General Context

In order to reach a wider audience, many major service providers have switched from the traditional desktop software model to making services available over the internet. This trending approach, also known as Software as a Service (SaaS) has enabled the service providers to reach a large number of potential customers. The growing use of internet has fueled a complete new generation of web service providers and users. While the desktop software model limits the usage of the software to a particular operating system, services over the internet, particularly those that are accessible using the HTTP [Berners-Lee et al., 1996; Fielding et al., 1999] protocol, help the customers to easily access them over their web browsers using any operating system.

Many small and medium scale enterprises have started using the web services for their daily transactions. Enterprises are now moving from the traditional bloated one-software-fits-all model to the more specialized web services suiting their specific needs. These enterprises have now been able to focus on their core business rather than involving themselves to the cumbersome task of maintaining their own internal softwares.

But the internal softwares and applications have the benefit that the entire business data is within the reach of individual business owners. In spite of being managed by different departments of the enterprise, the enterprise owners have the ability to have direct control over their entire data infrastructure and hence over their own business data.

Enterprises using web services have no direct control over the underlying data in-

frastructure of the service providers 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 applications and dashboards. A majority of web service providers expose the API so that both the clients and third party users authorized by the clients can access and manipulate the client data. APIs of several service providers can be used together to create interesting new web services [Thakkar et al., 2004] often called service mashups [Benslimane et al., 2008]. A lot of service mashups have appeared in the market during past several years offering various interesting services to the web service users making use of their user data.

Web services APIs differ among each other significantly with respect to the use of different message formats, authentication mechanisms, service level agreements (SLA), access limitations, data types, and the choice of input, output and error parameters. APIs are mostly described using human readable (HTML) web pages. Therefore it requires a significant amount of development effort to read these web pages and create service mashups. Thus most of these mashups are limited to a very few web service APIs. Taking this into account, the research and industrial community had previously proposed various machine readable standards to describe the web service APIs, particularly WSDL [W3C, 2001] and WADL [Hadley, 2006]. But the lack of their widespread industrial adoption still poses a major challenge.

Furthermore service providers often make updates to their services like addition and deprecation of resources, change in the API or SLA. They deprecate the older versions of the API making them no longer available to the external world. Sometimes these changes force the clients to change their service providers due to various business reasons (e.g., financial). In some cases, web service providers may shutdown. All the above situations may lead to the loss of past enterprise data and also requires changes in the internal applications and dashboards created with the older web service API. Though the use of web services have helped a lot for the small and medium scale enterprises in reducing their workload, it has made it difficult for them with lesser human resources and expertise to seamlessly integrate with numerous web services.

Business analytics form the prime pillars for every enterprise willing to track its growth and performance. Integration of enterprise data spread across various data sources (like various department databases) is important towards this goal. Indeed a (central) data warehouse [Inmon, 1992; Kimball, 1996] helps the enterprises to have an integrated view of all the relevant enterprise data obtained from the various departments (or business sub-units) using wrappers [Roth and Schwarz, 1997]. But the

dependence on web services has resulted in a situation where seamless integration with these heterogeneous, autonomous and ever-evolving data sources poses a major problem.

Problem Studied and Contributions

Our work focuses on creating a solution able to easily provide an online and personalized data warehouse for small and medium scale enterprises using web services. The purpose is two fold: first, providing a store for past historical enterprise data and second, enabling the easy computation of business performance indicators hiding from them the underlying intricacies and heterogeneity of the web services.

Scientifically speaking, the first and important problem is how to feed a data warehouse with data coming from web services. Also we have constraints like having very few developers and the need to work with real web services. We would like to ease the burden of developers by shifting the focus from development to administration, i.e. every new web service API or any change in web service API (operation) is declaratively described in the platform. Towards this objective, we propose to address the entire ETL (extraction, transformation and loading) of relevant web service enterprise data to the data warehouse by choosing the mediation approach, a well-known virtual data integration approach. Fully automating ETL from web services is a very complex task especially given the lack of usage of machine readable web service description languages. Mediation enables the use of declarative languages to handle the update/adding of new web services as data sources to data warehouse.

From an industrial point of view, we designed and developed DaWeS, a DAta warehouse fed with data coming from WEb Services [Samuel, 2014; Samuel and Rey, 2014; Samuel et al., 2014]: a complete multi-enterprise data warehouse platform to be fed with the enterprise data from the web services. We designed and implemented a complete ETL module based on mediation along with a generic HTTP Web Service API wrapper to extract and transform data from the web services using their API. The result is a data warehouse with a partly dynamic global schema able to store relevant enterprise data and compute user-defined performance indicator queries using these data.

From a research perspective, we validated our approach by using actual web services from three different domains. Certain web service operations require input arguments, the values of which must be obtained from other web service operations. We consider web service operations as relations with access patterns (an access pattern characterizes the input and output arguments of a relation). We defined the precise operational

semantics of conjunctive query with access patterns and use it to define an upper bound on the number of accesses for conjunctive query with access patterns. Since every access corresponds to one web service API operation call, this bound allows to have a measure on the number of accesses (i.e. calls) during ETL. Web service API operation calls are expensive since they are both priced and also made over the internet (thereby consuming internet bandwidth). The goal of this measure is to help future efforts to reduce the number of these (expensive) API operation calls made over the internet; thereby it plays an important role in comparing various query optimization algorithms. Besides we face another problem: not every web service provides us all the information required to answer the queries formulated over the data warehouse schema. We thus propose a heuristics to handle the incomplete information arising out of such web services.

Outline

Chapter 2 presents the problem statement, describing the industrial problem and the various constraints. In particular, it gives an analysis of API of actual web services from three different domains: project management, email marketing and support-/helpdesk.

In chapter 3, we present the state of art and compare our work with various other works that aimed to integrate with the web services. We position our work with other very closely related domains.

Chapter 4 presents the preliminaries. It first introduces the notion of ETL (extraction, transformation and loading), a classical approach to feed the data warehouse and discusses how the mediation approach can be used as an ETL tool to feed the data warehouse with the web service data. Traditionally in data integration, wrappers are used to extract data from the data sources. We discuss how this idea has been extended to the domain of web services to build a generic wrapper using only basic web service technologies that fits well with mediation.

Chapter 5 describes in detail how the mediation approach is used to build DaWeS. DaWeS architecture is discussed in detail along with its development.

We discuss in Chapter 6 various open problems encountered during the study and development of DaWeS. We present the study on how to obtain the upper bound on the number of accesses for conjunctive query with access patterns. We also discuss heuristics to handle incomplete information coming from web services.

We then move on to discussing how DaWeS can be used. Chapter 7 discusses its

usage both from the point of view of a DaWeS administrator and a DaWeS (client enterprise) user. We discuss the methodologies to be followed in deciding the relevant enterprise data to be stored, describing web service API operations and performance indicators. We also present qualitative and quantitative experiments done using real web services from three domains and discuss the obtained results.

In chapter 8, we describe the future course of actions from an industrial and scientific context. We also discuss how advancements in the closely related domains can be used to extend the capabilities of DaWeS. Finally we conclude our work.

In addition to these, Appendix A gives the glossary. Appendix B describes the analysis of the API of web services in detail. Various examples explaining how to add a new web service, how to formulate queries to extract relevant data from web services, and how to create a performance indicator query are described in Appendix C. DaWeS manual has been summarized in Appendix D, also describing the various relational tables used in DaWeS for storing web service description, performance indicators and enterprise data.

Chapter 2

Problem Statement

Web services are heterogeneous, autonomous and ever-evolving. Small and medium scale enterprises are dependent on numerous such web services for their various day to day requirements. Integrating with multiple such web services with lesser human resources is currently a daunting task. But such an integration is important not only to get an integrated view of the enterprise data spread across numerous web services but also to be able to perform various business analytics over this integrated data. In this chapter, we will first present the industrial problem and discuss how it can be related to a data warehouse fed with web services. We will discuss the various constraints especially the fact that the current generation of web services do not use advanced web service description languages.

2.1 Industrial Problem

The initial problem was given by Rootsystem [Rootsystem, 2014], an enterprise based in Vichy, France. Their product LittleCrowd [LittleCrowd, 2014] wants to be one-place portal for all the quality management requirements of enterprises. Business performance measure is a key aspect. They specially target small and medium scale enterprises using web services for their daily requirements. The requirements are summarized below:

- Periodically or on trigger of events (manual or automated), collect relevant information (records or periodic snapshots) of the client data from various web services and store them (if permitted by the clients) in a local database.

- Compute business performance measures periodically or on trigger of events (manual or automated) using the client records stored in the the database.
- In addition to default performance indicators, enable clients to easily define and compute new ones using their available records.
- Integrate easily with a large number of web services.
- Take into account the evolution of web services.
- Cater to the fact that clients change their web services from time to time and must still have access to their old enterprise data from their previous web services and achieve continuity with the new web service data.
- Periodically verify and guarantee the process of accurate fetching of the enterprise web service data.
- Periodically verify accurate computation of performance indicators.
- Support searching of supported web services and performance indicators and incorporate the popularity and folksonomies related to the search results.

2.2 DaWeS: Data warehouse fed with data coming from Web Services

A data warehouse is commonly used in the industry for the purpose of business data analytics. It provides an integrated view of the relevant enterprise data obtained from various departments (or business sub-units) and is later used for computing business indicators. In our case, the data sources are the web services. Therefore the goal of our work is to build a DAta warehouse fed with data coming from WEb Services. We call it DaWeS.

The aim of DaWeS is captured in the Figure 2.1. While an enterprise user continues to use the web services for the daily transactions, DaWeS periodically extracts relevant enterprise data from these web services using their API and with the authorization of the enterprise owner. Apart from storing the historical enterprise data, it also computes some default (and popular) performance indicators using these data and also enable the enterprise user to easily define and compute new performance indicators (for performance dashboards). A DaWeS administrator periodically watches and manages API of various supported web services for any announced changes (eg., official blogs and forums of the service providers) and add support for new web services considering the client demand.

A typical business use case of DaWeS can be described as follows. Alice, a small

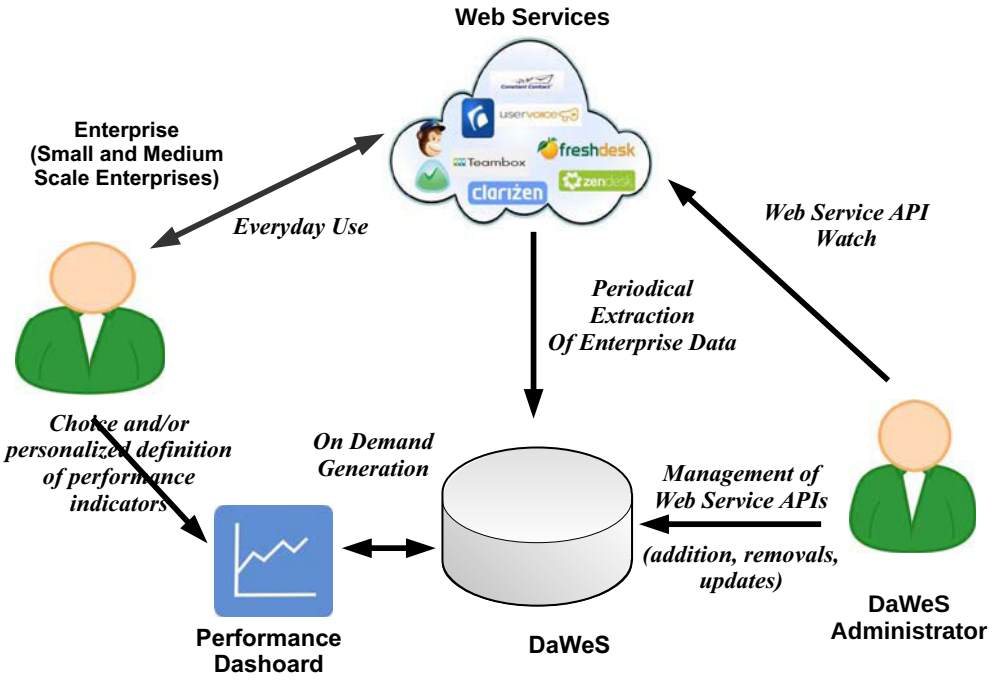

Figure 2.1: DaWeS for Enterprises using Web Services

scale enterprise owner has just begun her enterprise and is currently using a project management service P_1 to manage her several different projects. Her second product will be released in a couple of months. She wants to track the progress of various tasks in the different projects, especially the open tasks. She periodically checks the website of P_1, but the website doesn't offer her any statistics on the number of open tasks. She checks every project and verifies the individual task status and counts the open tasks to finally get the number she wants. Being a small company, she also manages on her own various email campaigns to study the market demand of her upcoming product. For this purpose, she is using two email marketing web services E_1, a free service and E_2, a premium service. She wants to compare the performance of her various trial campaigns on both these services. She manually checks the website of E_1 and E_2 to know the latest campaign statistics and later compares them using a spreadsheet. She had recently set up a team for getting customer feedback and feature requests and was using a support/helpdesk service S_1. She was happy with the services provided by S_1 until recently when they changed their prices. She heard about S_2, another helpdesk

service and decided to give it a try. But she wonders how she can get a one-portal view of all her enterprise data spread across P_1, E_1, E_2, S_1 and S_2. Then she heard about DaWeS, a solution to her requirements: a one-portal view for business analytics from web service enterprise data. With DaWeS, she can not only get a consolidated statistics of all the campaigns, projects and complaints (feature requests) across web services P_1, E_1, E_2, S_1 and S_2., but also create her own business performance measures like the monthly number of clicks she had received for her campaign. Thanks to DaWeS, she can have a snapshot of the data from her previous web service (eg. S_1) and continue using them even with a new web service (eg. her trial with S_2).

For DaWeS to be a generic solution for all the enterprises, it cannot assume working with a couple of services from selected domains like email marketing, project management or support (helpdesk) services. It must not only be able to support any increasing number of domains but it must also be possible to easily integrate with any number of web services from these domains. In a nut shell, DaWeS must provide the following features:

- **Scalable and Adaptable platform**: DaWeS must be scalable, that is it must be easily possible to integrate with numerous web services. It must also be adaptable, that is when a web service changes its interface, it must be very easy to make this update to DaWeS.
- **Historical Data Storage and Performance Indicators**: DaWeS must give an option to the enterprises to periodically store selected enterprise data from web services and allow them to create interesting business measures using the stored enterprise data.
- **Continuity of Enterprise Data**: Enterprises may change their web service providers or the web services may shut down. Enterprises must be able to easily work with their past enterprise data along with the data coming from the new web service (from the same domain). Accordingly the performance indicators must continue to work even with the change of web services.

2.3 Constraints

We also have the following constraints to solving the above industrial problem

1. Very few developers to manage thousands of web services
2. Consider actual web services

A very small enterprise doesn't have a lot of developers to write programs to integrate with thousands of ever-evolving web services. Therefore it must be simpler to add and update new web service API to the underlying platform[1]. This explains constraint 1.

Constraint 2 is quite natural. As a third party user, the only available interface to access the enterprise data stored by the web services is by the application programming interface (API) of these services. We now focus our attention to the characteristics of API of the web services. The main conclusion of our analysis is that the use of advanced web service description languages cannot be assumed.

We present in this section a summary of the survey of seventeen web services belonging to three business domains to establish the mostly used web service standards that are effectively used by service providers. The complete analysis is given in Appendix B (pages 183-194). 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. Each of the previous service may propose many operations, each of which has a callable API. Refer page 183 for the complete description of the three domains.

The 17 surveyed web services are[2]: project management (Basecamp, Liquid Planner, Teamwork, Zoho Projects, Wrike and Teambox), email marketing (MailChimp, Campaign Monitor, iContact, Constant Contact and AWeber) and helpdesk (Zendesk, Desk, Zoho Support, Uservoice, FreshDesk and Kayako). We summarize our analysis with these web services in Tables 2.1, 2.2 and 2.3. The web services are classified according to: the language in which APIs are described (i.e., documented), their REST compliance [Fielding, 2000], their (current analyzed) version of the API, their authentication method, the resources 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

[1]This constraint also comes from Rootsystem that has only 2 permanent employees

[2]http://www.basecamp.com, http://www.liquidplanner.com, http://www.teamworkpm.net, http://www.zoho.com/projects, http://www.wrike.com/, http://www.teambox.com, http://www.mailchimp.com, http://www.campaignmonitor.com, http://www.icontact.com, http://www.constantcontact.com, http://www.aweber.com, http://www.zendesk.com, http://www.desk.com, http://www.zoho.com/support, http://www.uservoice.com, http://www.freshdesk.com, http://www.kayako.com

agreement (constraints on the operations usage) their HTTP access method, the used data types, their handling of dynamic resources (resources whose 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 same API operation call(s) with different parameters to retrieve all data).

From these characteristics, an average profile of web services emerges: describing services with HTML, (a limited) following of the REST architecture style, using basic HTTP authentication with a GET access, XML or JSON as message format, strings, enumeration and date as data types, dynamic resources and sequence of 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, 2012], a directory which documents 10,555 APIs and in which a vast majority (around 69%) are REST based web services.

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

The solution we describe in chapter 5 is to manually achieve the connection between DaWeS and web services in a two-fold manner: (i) dedicating the greatest part of the manual effort to establish the semantic connection between data in DaWeS and data coming from the web services, 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 only declarative languages for every manual task.

Table 2.1: Web Service API Analysis on Project Management Services

Web Service	**Basecamp**	**Liquid Planner**	**Teamwork**	**Zoho Projects**	**Wrike**	**Teambox**
1. API Description	HTML page	HTML page	HTML page	HTML page	HTML page	HTML page
2. Conformance to REST	REST like	REST like	REST like	Not REST	Not REST	REST like
3. Version	v1	3.0.0	N.A.	N.A.	v2	1
4. Authentication	Basic HTTP, OAuth 2	Basic HTTP	Basic HTTP	Basic HTTP	OAuth 1.0	OAuth 2.0
5. Resources Involved	Project,Todo List, Todo	Project, Task	Project, Task List, Task	Project, Task List, Task	Task	Project, Task
6. Message Formats	JSON	JSON	XML, JSON	XML, JSON	XML, JSON	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	N.A	N.A.
8. HTTP Resource Access	GET	GET	GET	POST	POST	GET
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	Enumerated dt (Task Status), Date	Enumerated dt (Project and Task Status), Date
10. Dynamic nature of the resources	Yes (Project and Todo Status)	Yes (Project and Task Status)	Yes (Project and Task Status)	Yes(Project and Task Status)	Yes (Task Status)	Yes (Project and Task Status)
11. Operation Invocation Sequence Required	Yes	No	Yes	Yes	Yes	Yes
12. Pagination	No	No	No	Yes	Yes	No

Table 2.2: Web Service API Analysis on Email Marketing services

Web Service	**Mailchimp**	**Campaign Monitor**	**iContact**	**Constant Contact**	**AWeber**
1. API Description	HTML page	HTML page	HTML page	HTML page	HTML page
2. Conformance to REST	Not REST	REST like	REST like	REST	REST
3. Version	1.3	v3	2.2	N.A.	1.0
4. Authentication	Basic HTTP	Basic HTTP, OAuth 2	Basic HTTP (with Sandbox)	OAuth 2.0	OAuth 1.0
5. Resources Involved	Campaign, Campaign Statistics	Campaign, Campaign Statistics	Campaign, Campaign Statistics	Campaign, Campaign Statistics	Campaign, Campaign Statistics
6. Message Formats	XML, JSON, PHP, Lolcode	XML, JSON	XML, JSON	XML	JSON
7. Service Level Agreement	N.A.	N.A.	75,000 requests /24h, with a max of 10,000 requests /1h	N.A	60 requests per minute
8. HTTP Resource Access	GET	GET	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	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)	Yes (Campaign Status)	Yes (Campaign Status)
11. Operation Invocation Sequence Required	Yes	Yes	No	Yes	Yes
12. Pagination	Yes	No	No	Yes	Yes

Table 2.3: Web Service API Analysis on Support/Helpdesk Services

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

Chapter 3

State of the Art

DaWeS is a data warehouse fed with the enterprise data from the web services using web service API. It uses the mediation approach coming from the data integration field to describe the web service API operations and their access patterns (to distinguish between input and output attributes). The mediation approach chosen by us allows us to expose to the end users a mediated (or global) schema hiding from them the underlying heterogeneity of the numerous web services. The end user formulates the queries using only the global schema. There are several ways of describing sources in mediation based data integration systems (section 3.3.1). DaWeS uses Local as View Mapping (LaV) mapping to describe the web service API operations given its scalable nature. DaWeS internally uses query rewriting algorithm to translate query formulated over the global schema to query formulated using the web service API operations. This translated query is evaluated to extract data from the web services and store them to the underlying relation database of DaWeS. This enables DaWeS to use the DBMS capabilities to compute performance indicators (written as SQL queries) using the stored enterprise data.

In this chapter, we discuss various related works associated with integrating data coming from the web services. We also position our work with respect to other closely related domains.

3.1 DaWeS and Data Warehousing

In this section we locate DaWeS with respect to three main dimensions in data warehouse: the schema (data organization), the ETL process and the wrappers. There are two very popular approaches commonly used for building a data warehouse: bottom up approach and top down approach. In a bottom-up approach, data marts (departmental view of information) are used (or created) first and later combined to build a data warehouse. Therefore a data warehouse in this context is a union of all the data marts. A top-down approach starts from the overall data warehouse schema and then goes down to the design of individual data marts. We discuss two popular industrial data warehouse approaches: Kimball approach and Inmon approach. Kimball approach [Kimball, 1996] considers a data warehouse as *a copy of transaction data specifically structured for query and analysis.* It is a bottom up approach and use fact (metrics) tables and dimension tables to store the data. A fact table stores the numerical performance measurements of a business and the data is obtained from a business process. Dimension tables have numerous attributes giving the detailed information (textual descriptors) of the business. Inmon approach [Inmon, 1992] is a top down approach and uses 3NF (third normal form) tables to store extracted (and transformed) data from the data sources. Inmon considers a data warehouse is a *subject-oriented, integrated, time-varying, non-volatile collection of data in support of the management's decision-making process.* We consider DaWeS as a top-down approach since we start with designing the global schema and then link the sources to it. Though global schema (or the data warehouse schema) is created only after understanding the client requirements, performing the market study and understanding the API operations (local schema) of the web services, we first build it and later map the API operations to it. But DaWeS doesn't materialize the global schema instead use it for querying. DaWeS doesn't employ any star schema as seen in the Kimball approach. Instead the global schema relations are simply table entries (section 5.2). On one side, this is very convenient for the user to be able to quickly define new global schema relations making DaWeS partly dynamic. On the other side, it is less straightforward to applying the above popular data warehouse storage approaches. For example, handling of advanced performance indicators like the CUBE operators (used along with the star schema) needs to be further explored.

Xyleme [Xyleme, 2001] is another closely related work to DaWeS. It proposes building a data warehouse using a large number of crawlable XML web pages. After crawling, it proposes to keep a complete copy of these XML pages and to provide a view mechanism (abstract DTD [Bosak et al., 1998]) where a single schema is used for querying

hiding the heterogeneity of the underlying heterogeneous XML pages. It also supports change control [Mignet et al., 2000]. DAWAX [Baril et al., 2003] presents a working prototype related to building a data warehouse using view model for XML sources. The data warehouse is defined as a set of views. A view serves the purpose of selecting various XML sources and building composite views out of them. Another purpose of a view is to aid the building of a global integrated schema. DAWAX presents graphical tools to build such views and manage the data warehouse. Unlike DaWeS, both DAWAX and Xyleme store the data obtained from XML sources for the purpose of querying them later. They don't use mediation for the purpose of extraction and transformation of relevant information to a standard internal format (RDBMS) and to deal with access patterns. DAWAX though stores XML data into relational database to make use of the querying capabilities of the RDBMS. DaWeS only stores the query response obtained after the query evaluation in RDBMS and doesn't keep a local copy of entire enterprise data. Also the enterprise data on the web services are not easily crawlable sources. The complete discussion on DaWeS and other data warehousing approaches has been summarized in Table 3.1.

Table 3.1: DaWeS and Data Warehousing

Name	Primary Data Source	Warehousing Approach	Warehouse Schema	Local Copy of External Data Sources
Kimball Approach [Kimball, 1996]	Enterprise Operational data	Bottom-up	Star Schema	Yes
Inmon Approach [Inmon, 1992]	Enterprise Operational data	Top-down	3NF Normalized tables	Yes
Xyleme [Xyleme, 2001]	Crawlable XML Data Sources	Bottom-up	Abstract DTD	Yes
DAWAX [Baril et al., 2003]	XML Data sources specified by user	Bottom-up	Mediated Schema	Yes
DaWeS	Web Services API (XML and JSON) authorized by users	Top-down	Mediated Schema	No (Limited, only query responses)

There are several other works [Berti-Equille and Moussouni, 2005; Calvanese et al., 1998, 1999, 2001a; Guérin et al., 2005; Hammer et al., 1995; Zhou et al., 1995] that deal with warehousing using data integration. DWQ (Data Warehouse Quality) Project [Calvanese et al., 1998, 1999, 2001a] is closely related to DaWeS. They also suggest a declarative approach to the overall problem of integrating data from various sources to feed the data warehouse. They employ wrappers and mediators and also use Local as View mapping for describing the external data sources with access patterns. Unlike DaWeS, they also take into consideration interschema correspondences of the various schemas (source schema and data warehouse schema) involved. These are done by various programs that perform the matching, conversion and the reconciliation of the data. Another major difference is that DaWeS doesn't materialize any warehouse schema relation but only the responses of the queries formulated over the warehouse schema. The reason comes from the fact since DaWeS aims to be a platform for a warehouse service for multiple enterprises. Enterprises may not be willing to entrust DaWeS with their complete (confidential corporate) data, but only certain selected data (current generation of services have started offering API that allows third party users only certain selected data authorized by their owners). Secondly, complete materialization of warehouse schema relations is also expensive for the enterprises from cost perspective. But the reader may also note that it is possible to materialize the warehouse schema relation by creating a conjunctive or datalog query with the warehouse schema relation in the body of the query.

Another key requirement of the data warehouse is the incremental update of the warehouse. When new data are added into the sources, the warehouse must also reflect it. In DaWeS, we make use of scheduler that periodically evaluates the queries to extract information from the web services. H20 [Zhou et al., 1996, 1995] is another data warehousing approach that aims to deal with the incremental update of the sources. But they assume 'active' modules, i.e., softwares whose behavior can be defined using rules. This enables them to manipulate both the sources and the warehouse whenever a new update is made to any data source. In the current version of DaWeS, we only make use of the API operations that can be used to access the enterprise data. H20, based on object-model also uses a declarative data integration approach to build a data warehouse using Global as View (GAV) mapping. Mediators are generated using Integration Specification Language (ISL). ISL is used to specify the source schema relations, various criteria for matching objects from various source classes and the derived classes from the mediator. GEDAW (Gene Expression Data Warehouse) [Berti-Equille and Moussouni, 2005; Guérin et al., 2005], a specialized warehouse to store and manage relevant information for analyzing gene expression measurement also utilizes GAV map-

ping to define XML sources and in-house database. WHIPS, the Stanford Warehousing Project [Hammer et al., 1995] also stresses the need for warehousing approach to the information integration for the purpose of collection of scientific data, maintenance of enterprise historical data and caching of frequently requested information.

Webhouse [Kimball and Merz, 2000; Lopes and David, 2006; Zorrilla et al., 2005] is a data warehouse that stores user clickstream on a website and other contextual information. Webhouses play a significant role for site owners to improve user experience. They store the user behavior on a website, like how a user navigates a web page, clicks various links, web browsers used by the users, the underlying operating system and various other information corresponding to a user session on a website.

Another closely related work associated to handling web pages over the internet are the web warehouses [Bhowmick et al., 1999, 2003; Cheng et al., 2000; Vrdoljak et al., 2003; Yu et al., 2008], data warehouses to store hyperlinked web documents. In most of the above works, the prime sources of information were in-house databases and crawlable HTML/XML pages. DaWeS only targets the web services with API exposed to third party users. We target those web services where the message format used for the communication is XML/JSON.

Data warehouses are fed with data from the sources using ETL tools. We recall from [Trujillo and 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 uses the XSLT files to perform data transformation in accordance to the target (global) schema, and the query response constitutes the data for the target. These are currently automated in DaWeS. But there are several manual steps. Take for example: the XSLT files used for the transformation of the operation response is manually created. Similarly, the local schema relations are manually described and mapped using the target (global) schema with LAV mapping.

Active data warehouses are updated frequently to reflect the latest changes in the sources. ETL queues [Karakasidis et al., 2005] have been suggested for real-time data warehousing. Any changes in the source is collected as blocks and propagated to the warehouse. The ETL workflow consists of activities called ETL queues, each of them pipelining blocks of tuples to subsequent activities. Once the ETL processing (cleaning

and transformation) is over, a receiving web service for every table or materialized view is used for populating the warehouse. DaWeS doesn't assume the availability of such sources (or web services) that can inform the warehouse about the changes in the source. DaWeS works in an offline fashion i.e., it periodically extracts enterprise data from the web services by evaluating queries and not at real-time. It polls the web services periodically to obtain the latest information (e.g. tasks that were created yesterday).

The data wrappers [Roth and Schwarz, 1997] encouraged the enterprises not to scrap their legacy data stores but rather wrap them with the help of data wrappers in order to make use of their (historical or legacy) data sources. Wrapping a data source corresponds to querying these legacy data stores and inferring various information from them in a desired format. Wrappers for web services can also be automatically generated with the help of advanced web service description languages. Examples include Axis [Axis, 2012] for SOAP web services and Jersey [jersey, 2012] for REST web services. Generic wrapper (section 5.2.2.6) for web services in DaWeS is to handle the ability to work with numerous web services in a declarative manner under the absence of advanced description languages.

[Benatallah et al., 2005; van den Heuvel et al., 2007] have discussed configurable adapters (wrappers) before to deal with web services replacement and evolution. But their primary audience was the services using business standards like BPEL. DaWeS however targets any web service using the basic web standards (HTTP, XML, JSON, Resource-oriented) to expose their API. Response validation and calibration in DaWeS help the administrators to detect any unannounced API changes.

3.2 Integrating Data coming from the Web Services

[Hansen et al., 2002] study data integration using web services built over web standards like XML, HTTP, SOAP [Box et al., 1999], WSDL, UDDI [McKee et al., 2001] for the purpose of aggregation. They define an aggregator as *an entity that transparently collects and analyzes information from different data sources, resolves the semantic and contextual differences in the information, addresses one or more of the following aggregation purposes: content aggregation, comparison aggregation, relationship aggregation, process aggregation.* DaWeS can be seen as an implementation addressing more specifically the content aggregation (i.e., DaWeS extracts information from various sources on a specific topic and provides analytics) and relationship aggregation (i.e., DaWeS

provides a single point of contact between the user and various business services) issues using standards like XML and HTTP.

We classify various approaches to integration of data from web services on two dimensions: whether they use virtual or materialized approach and whether they follow a centralized architecture or a decentralized one. Materialized data integration is often called data warehouse. In a virtual data integration, the warehouse schema is used only for querying and is not materialized (i.e., they have no associated database extension). In a centralized integration approach, there's a single point of contact for the end users to get an integrated view of numerous heterogeneous data sources. But in a decentralized setting, integration is performed by all the participating entities. We compare various works with DaWeS in Table 3.2. [Pérez et al., 2008] surveys various works on integrating data warehouses with web data.

Table 3.2: DaWeS and Integration with Web Services

Name	Virtual or Materialized Integration	Centralized or Decentralized Integration
[Zhu et al., 2004]	Virtual (Federated)	Centralized
ActiveXML [Abiteboul et al., 2002]	Materialized	Decentralized
[Barhamgi et al., 2008; Thakkar et al., 2003]	Virtual	Centralized
ActiveXML Warehouse [Salem et al., 2013]	Materialized	Centralized
Kimball Approach [Kimball, 1996]	Materialized	Centralized
Inmon Approach [Inmon, 1992]	Materialized	Centralized
Xyleme [Xyleme, 2001]	Virtual	Centralized
DAWAX [Baril et al., 2003]	Virtual	Centralized
DaWeS	Virtual	Centralized

ActiveXML [Abiteboul et al., 2002] is a language that extends XML to allow the embedding of web services calls. It has been proposed to develop a dynamic and powerful data oriented scheme for distributed computation (e.g. peer-to-peer data integration). Active XML has also been proposed [Salem et al., 2013, 2010] for building

a (centralized) data warehouse. Indeed, DaWeS is a centralized system, since it is in fine a data warehouse for enterprise records and performance indicators and uses virtual data integration approach to obtain them.

[Barhamgi et al., 2008] makes use of semantic web standards like RDF and SPARQL and the mediation approach with (ontological) query rewriting to provide on-demand automated integration of data providing web services. IBHIS (Integration Broker for Heterogeneous Information Sources) project [Zhu et al., 2004] is built using the web services in the healthcare domain. It employs a federated database system [Sheth and Larson, 1990]. It exposes a federated schema. In a federated query, the user must explicitly specify the data sources. The query decomposer module decomposes the query formulated over the federated schema into a set of local queries which are then executed. Web mashups [Benslimane et al., 2008] compose two or more web services to create interesting new web services. They are created using graphical composition tools or are dependent on advanced web standards for their automated generation. These works are dependent on advanced standards (WSDL, UDDI, DAML-S [Burstein et al., 2002], SOAP, RDF, SPARQL), an assumption that we cannot make. But mashups can also manipulate the resources. [Thakkar et al., 2003] uses the inverse rules query rewriting algorithm to creating mediator as a web service and mediator as a web service generator. Similar to [Thakkar et al., 2003], we chose inverse rules algorithm given its capability to automatically compose data providing web services.

We analyzed some of these approaches based on their primary aim and targeted audience, their underlying approach, the use of standards, API operations handled, algorithms used and the schema and summarized in Table 3.3.

3.3 DaWeS and Mediation

Mediation is a virtual data integration approach that provides a uniform query interface to autonomous and heterogeneous data sources. In this section, we locate DaWeS with respect to three main dimensions in mediation: describing the data sources, query rewriting algorithms and query evaluation optimization.

Table 3.3: Data Integration and Web Services: State of the Art

Characteristics	DaWeS	[Zhu et al., 2004]	[Benslimane et al., 2008]	[Thakkar et al., 2003]	[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, WSDL, hRESTS	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

3.3.1 Describing Data Sources

In mediation systems, we recall that there are three ways by which the sources (the local schema relations) can be mapped to the global schema relations: Global-as-View(GAV) mapping [Adali et al., 1996; Chawathe et al., 1994; Halevy, 2001], Local as view(LAV) mapping [Duschka and Genesereth, 1997; Ullman, 2000] and Global-Local as view mapping (GLAV) [Friedman et al., 1999; Koch, 2004]. In GAV, each relation of the global schema is defined as a query over the source relations. In LAV, each source relation is defined as a query over the global schema relation. GAV mediators are known to offer good query answering properties, while facing an evolution in the sources may be difficult (e.g., adding a new source implies to potentially updating many relation definitions in the global schema). LAV mediators are known to easily handle source changes, while query answering is algorithmically more difficult. Indeed, the user query posed to the global schema must be rewritten into queries that can be posed to the source. And rewriting algorithms have a high complexity (NP-Complete at least). In GLAV, the global and local schema relations are mapped using a set of tuple generating dependencies (TGDs). LAV is easier than GLAV with respect to an algorithmic point of view. In DaWeS, we chose LAV because we want to integrate with a large number of web services that are not only ever-evolving but also new players come to the market on a frequent basis. Choosing GAV would amount to changing the mappings on a frequent basis. Whereas in case of LAV, a new web service API operation will simply require a new LAV mapping and no change in the existing mappings are required.

There are several information systems like Infomaster [Genesereth et al., 1997], Infosleuth [Bayardo et al., 1997], EMERAC [Kambhampati et al., 2004], Ariadne [Knoblock et al., 2001], TSIMMIS [Chawathe et al., 1994] and Information Manifold[Kirk et al., 1995]. Infomaster also integrates data from various sources using wrappers. It uses rules and constraints to describe the various data sources. Infosleuth uses a network of computing agents and mediation to answer user queries. EMERAC uses optimized inverse rules algorithm. Ariadne uses query rewriting approach. TSIMMIS uses GAV mapping whereas Information manifold considers the query planning under the LAV settings.

Enosys XML Integration Platform (EXIP) [Papakonstantinou et al., 2003; Papakonstantinou and Vassalos, 2002] is an XQuery-based integration platform. [Gardarin et al., 2002; Manolescu et al., 2001] also use XML and XQuery to integrate heterogeneous data sources. [Cautis et al., 2011b] deals with querying XML data sources exporting large set of views. A view in their context is a set of possible queries exposed

as web services. They take into consideration how to handle queries when the number of views exposed is exponential in the size of the schema or even infinite. In DaWeS, we considered those XML (and JSON) data sources whose output can be transformed using XSLT to a set of tuples that can be later fed to the datalog query engine. A detailed discussion on data integration, data warehousing, working with XML documents, data validation with XSD schemas, querying XML documents using XPath and XQuery and transforming XML documents to any other document format has been done in [Abiteboul et al., 2012].

In DaWeS, we used the manual approach to perform the mappings. But there have been research efforts to automate the generation of these mappings. Automated schema matching has been surveyed in [Rahm and Bernstein, 2001].

3.3.2 Query Rewriting

Query rewriting [Calvanese et al., 2000b, 2001b; Levy, 1999; Levy et al., 1996a; Ullman, 2000] involves translating a query formulated over the global schema relations to queries formulated over the local schema relations. The classical query rewriting algorithm includes the bucket algorithm [Ullman, 2000], minicon algorithm [Pottinger and Halevy, 2001; Pottinger and Levy, 2000] and the inverse rules algorithm [Duschka and Genesereth, 1997; Duschka et al., 2000]. A modified form of the bucket algorithm called the shared variable bucket algorithm [Mitra, 2001] enhances the bucket algorithm by using some extra buckets to avoid the conjunctive query containment test. The complexity of answering queries using materialized views is discussed in [Abiteboul and Duschka, 1998]. In DaWeS, we chose the inverse rules algorithm since it can generate a query rewriting in polynomial time, handle access patterns, handle recursive datalog queries and also support various integrity constraints on the global schema. Recent advancements in ontological query answering [Gottlob and Schwentick, 2012] can be used to support additional tuple generating dependencies (TGDs). There are various researches related to query rewriting as shown in Figure 3.1. We will postpone the discussion related to query rewriting under access patterns to the next section 3.3.3.

Incomplete Information: Incomplete information [Abiteboul et al., 1995, 1991; Imieliński and Lipski, 1984; van der Meyden, 1998; Vardi, 1986] is an important aspect that needs to be considered in the context of query rewriting. There are several ways by which the incomplete information is represented: using Codd nulls, marked nulls and horn tables. Even though horn tables are shown [Grahne, 1989] to be an efficient tool for handling incomplete information in databases, they are difficult to be used along

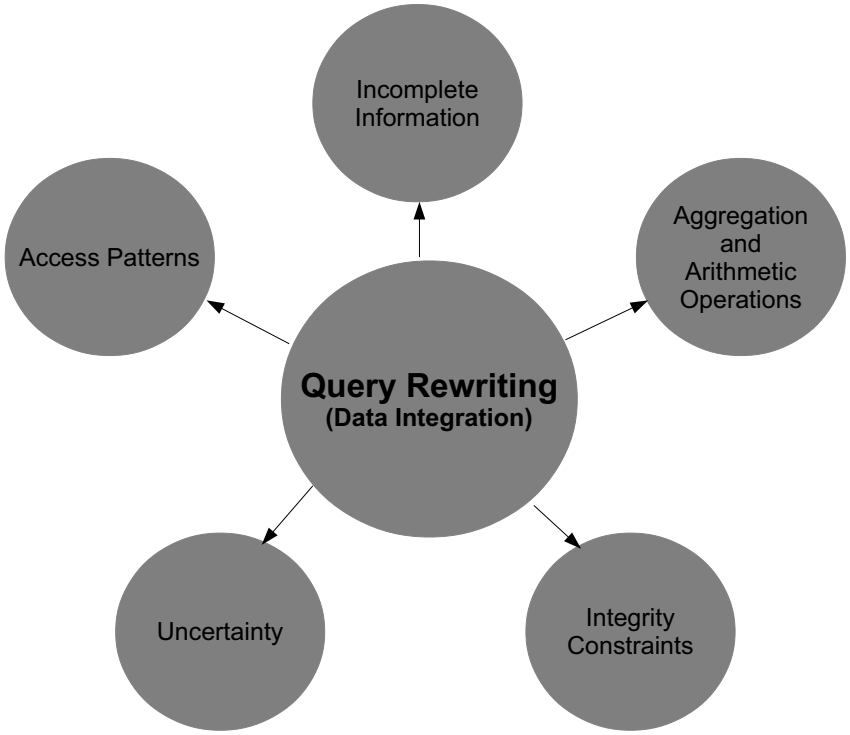

Figure 3.1: Query Rewriting Considered under Various Dimensions

with the current relational databases. Codd nulls do not cover various interesting information (when two unknowns are known to have the same values). [Li et al., 2001] brings the notion of p-containment showing that the traditional query containment [Chandra and Merlin, 1977] is not useful in while answering queries using views. Prior to that the tableau techniques for querying information sources through global schemas has been discussed in [Grahne and Mendelzon, 1999]. Unlike relation tables that are a set of facts, tableaus are used to store a set of atoms. Marked nulls are useful and they have been used for the purpose of query rewriting [Grahne and Kiricenko, 2002, 2003, 2004] by bringing in the notion of p-containment, under the LAV settings. [Grahne and Kiricenko, 2002] also introduces a modified form of the bucket algorithm: the p-bucket algorithm to handle conjunctive queries. Our heuristic in DaWeS (section 6.1) handles datalog queries and needs to be further examined towards a formal proof.

Integrity Constraints in the global schema relations We considered query rewriting under two sets of dependencies on the global schema relations: the functional

dependencies and full dependencies. Various other dependencies like the inclusion dependencies, join dependencies exist. More recently, a lot of interest has been on query rewriting under various constraints [Afrati and Kiourtis, 2008; Calì et al., 2002, 2004, 2013, 2003; Calvanese et al., 2000a; Christiansen and Martinenghi, 2004; Gottlob et al., 2011a,b; Gryz, 1999], especially for the different classes of tuple generating dependencies. [Bai et al., 2006] proposes a bucket based approach to query rewriting in the presence of inclusion dependencies.

Handling Arithmetic Operations In DaWeS, we didn't consider any LAV mapping that involved the arithmetic comparison operations. Query answering using views with arithmetic (comparison) operations [Afrati et al., 2002] is also another area of research and algorithms based on shared-variable buckets[Mitra, 2001] and minicon algorithm have been proposed.

Uncertainity When data from multiple heterogeneous sources are integrated, there is a high possibility that there some sources do not provide the latest or accurate information. During query answering, an additional field may be useful explaining the data provenance and the probability that the information is accurate. Various researches [Agarwal et al., 1995; Wolf et al., 2009] in this direction is also being currently done.

3.3.3 Optimization

Optimizations have been proposed for conjunctive queries using sources with access patterns [Calì and Martinenghi, 2008] by making use of the optimized dependency graph in order to reduce the number of accesses to the external data sources. [Calì et al., 2009a] suggests dynamic query optimization under the functional dependencies existing among the attributes of the relation, especially considering the case where a relation have multiple access patterns. In DaWeS, we consider relation with single access pattern. If there are multiple access patterns, we rename the relations so that each relation has only one access pattern.

Going beyond the classical access patterns [Cautis et al., 2011a; Halevy, 2000, 2001; Kwok and Weld, 1996; Levy et al., 1996b; Millstein et al., 2003; Rajaraman et al., 1995] involving input and output attributes, [Yerneni et al., 1999] considered additional adornments like unspecifiable and optional attributes. DaWeS currently considers only input and output attributes of API operations.

[Duschka et al., 2000; Li and Chang, 2000] use domain rules to handle access pat-

terns. [Li and Chang, 2000] also optimizes the query answering problem by removing any useless accesses to the sources that don't contribute to the query's answer. [Li, 2003; Li and Chang, 2001b] compute complete answers to queries in the presence of limited access patterns. A complete answer is an answer obtained to a query when all the tuples of the relation could be obtained. It is studied by considering the stability of the various classes of queries. A query is considered to be stable when for any instance of the relation, the complete answer can be computed under the considered access patterns.

[Nash and Ludäscher, 2004a] considers first order queries under limited access patterns whereas [Nash and Ludäscher, 2004b] considers union of conjunctive queries with negation under limited access patterns. In this work, we do not handle the adornment for the query predicate or the intensional predicates. [Yang et al., 2006] discusses about the feasible binding pattern for the intensional predicates and the ordering of the subgoals. Query rewriting using views with access patterns under integrity constraints is discussed in [Deutsch et al., 2007].

[Thakkar et al., 2005] proposes the use of inverse rules algorithm to perform web service composition and suggest the use of two optimizations one in the form of tuple-level filtering and an algorithm that transforms the query plan into dataflow-style streaming execution plan in order to reduce the number of calls to the web services and executing the generated query plan efficiently. Source selection during query answering is an interesting problem and access to sources that are not useful for query evaluation need not be considered. In case of semantic web, the reformulation trees [Li and Heflin, 2010] have been proposed to optimize queries over distributed heterogeneous sources.

Most of the previous works have worked on optimizing the number of accesses under various conditions. We cannot find any work in the literature that directly work on the defining the semantics of $\mathcal{CQ}^{\alpha_{Last}}$, $Datalog^{\alpha_{Last}}$ and $\mathcal{CQ}^{\alpha}$. By considering the operational semantics of these, we are able to define an upper bound on the number of accesses. In the field of static optimization, this upper bound aims at being a tool to evaluate and compare any query evaluation algorithm with respect to the number of accesses they imply.

3.4 DaWeS and Web Standards

Two popular architecture styles employed by the industry for the web services are SOA (Service oriented Architecture) [He, 2003] and REST (Representational State

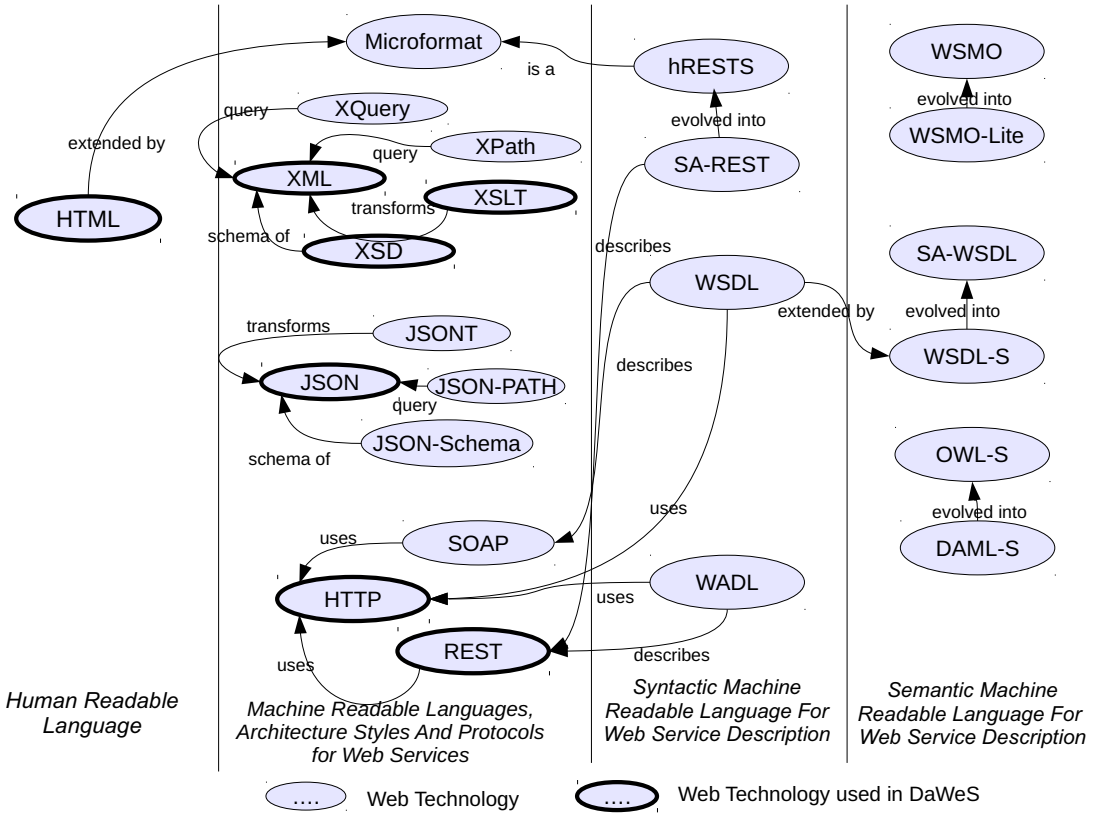

Figure 3.2: Languages for Describing Web Service Description

Transfer) [Fielding, 2000]. ProgrammableWeb [ProgrammableWeb, 2012], a directory of vast number of web service APIs have approximately documented 10,555 Web Service APIs (December, 2013), a vast majority of which (around 69%) are REST based web services.

Web services can be described in two ways: in a machine readable way or in a human readable way. They are mainly described in human readable web pages that are later used by the developers to develop client (programs) to integrate with the web services. Besides the primary aim of the machine readable interface descriptions is to create self-describing web services that can be used for web service discovery and composition without any (or least) human-intervention. Web services based on SOAP [Box et al., 1999] use WSDL [W3C, 2001] for web service interface description. WSDL describes the service, the service endpoint, the interface (the operations) and the data types of the input and output using XSD (XML Schema). WSDL have been

quite popular for SOAP based web services. Extensions to WSDL like SAWSDL (Semantic Annotations for WSDL and XML Schema) [Kopecký et al., 2007] (evolved from WSDL-S [Akkiraju et al., 2005]) have been proposed to WSDL in order to semantically annotate WSDL with additional information like the description of various elements used in the services and their relation to ontology, semantics of the various operations, describing the semantics of the quality of the service and the verification of the process (execution semantics). Web services must sharing these additional information beyond the regular interface descriptions are useful when they register themselves to web service registry. They are very useful for data mediation, web service discovery, composition and orchestration especially in a dynamic SOA environment [Sheth, 2003; Sheth et al., 2008]. SAWSDL supports XSLT transformation, that allows the translation between the source schema and the target schema, thereby proving useful in the data mediation. Other proposals include OWL-S [Martin et al., 2009], WSMO [Lausen et al., 2005] and WSMO-Lite [Vitvar et al., 2008]. Another study [Maleshkova et al., 2010] takes a survey about the different authentication mechanisms used by the web services and suggests enhancement to add semantics to represent different authentication mechanisms. WADL [Hadley, 2006; W3C, 2009] is another proposal catered towards describing resource-oriented HTTP-based web applications.

Microformats are additional attributes used along with the existing HTML tags. Microformats [Microformats, 2012] are *designed for humans first and machines second, microformats are a set of simple, open data formats built upon existing and widely adopted standards* hRESTS [Kopecký et al., 2008] (evolved from SA-REST [Gomadam et al., 2010]) is a microformat to describe the RESTful web services by annotating service descriptions in HTML. hRESTS seems to be an interesting proposal since it makes use of the current manner of describing the web service interface using HTML (human-readable) pages and microformats to make the web page machine-readable.

We see that in general the web services considered in the Table B.7 use XML [Bray et al., 1997] and JSON [Crockford, 2006] formats for the communication. Compared to XML, JSON is a much recent player in the field of communication and is gaining popularity given its light-weight nature. XML has been well researched and has been in use for more than a decade. In order to define the possible content in a given XML document, XSD [Gao et al., 2009] is used to define the schema. To query the XML documents, various standards like XPath [Berglund et al., 2007] and XQuery [Fernández et al., 2002] have been proposed. XSLT [Kay et al., 2007] is used to transform an XML document to another XML or any other format (like HTML). In DaWeS, we use XSD to define the schema of the API operation response and use it to validate the operation

responses. We also make use of XSLT to transform the response obtained from the web services to a set of tuples (understood by the IRIS datalog engine). Our choice of XML, XSD and XSLT is due to the maturity in this field. In case of the web services using JSON, we first transform the JSON response to XML and use XSD and XSLT like the other XML based web service API responses. There are some proposals within the JSON community like JSON-Schema [Zyp et al., 2010] to describe the schema of JSON documents, JSONT [Goessner, 2006] to transform the JSON documents to any desired format and JSONPath [Goessner, 2007] to query the JSON documents. Inspite of the above limitations, JSON is also a good choice to be considered as an intermediate format given its lightweight nature. These web technologies has been summed up in the Figure 3.2 highlighting as well the technologies used in DaWeS.

Chapter 4

Preliminaries

4.1 Theoretical Preliminaries

We now recall basic concepts of the relational model, and then focus on the notion of access and associated concepts.

4.1.1 Relational Model Recalls

These recalls are mainly taken from [Abiteboul et al., 1995; Grahne and Kiricenko, 2004]. Consider *dom* to be a countably infinite set of values (a.k.a. constants). These constants are noted with small lettres a, b and c, with or without indices. Small letter p, q and r, with or without indices, are set to be relation names. Other small letters names may be given to relation names. We may also use "relation" instead of "relation name". Each relation name r is associated to an arity $arity(r)$ which is an integer. We note this $r^{(arity(r))}$. A relational schema is a finite set of relation names. Relation arities may not be noted when writing a relational schema. For instance $\{r_1, r_2, \ldots, r_n\}$, where $n \in N$, is a possible relational schema. $\{r_1^{(arity(r_1))}, r_2^{(arity(r_2))}, \ldots, r_m^{(arity(r_m))}\}$, where $m \in N$ is another. From *dom* and the relation names, we built up a universe consisting of all expressions of the form $r(c_1, c_2, \ldots, c_{arity(r)})$ where r is a relation name and the c_i's are constants. This is the Herbrand universe. Such expressions are called facts. A database instance $\mathcal{D}$ for a schema $schema(\mathcal{D}) = \{r_1, r_2, \ldots, r_n\}$ is a finite set of facts which relation names are taken in $schema(\mathcal{D})$. We may use "database" instead of "database instance". The instance of a relation r belonging to $schema(\mathcal{D})$ is the set of all facts which relation name is r. This set is noted $r(\mathcal{D})$.

Consider now $\mathcal{V}$ a countably infinite set of variables. We mainly use capital letters X, Y and Z, with or without indices, to denote variables. Other capital letters may be used (like V and T for instance). We use the same capital letters with an overline to denote tuples of variables, like $\overline{X}$ for example. The function *var* applied to a tuple of variables returns the set of all variables that are in this tuple. An expression of the form $r(d_1, d_2, \ldots, d_{arity(r)})$ where r is a relation name and the d_j's are either variables or constants is called an atom. A valuation is a mapping from $\mathcal{V}$ into *dom*, straightforwardly extended to tuples of variables, and extended to be the identity on *dom*.

Definition 4.1.1 (Conjunctive Query Syntax)**.** *A conjunctive query ψ is an expression of the following form, defined according to the database instance $\mathcal{D}$*

$$\psi : q(\overline{Z}) \leftarrow r_1(\overline{X_1}), \ldots, r_n(\overline{X_n})$$

where $r_i \in schema(\mathcal{D}), \forall i \in \{1, \ldots, n\}$ and $\overline{Z}, \overline{X_1}, \ldots, \overline{X_n}$ are tuples of variables and constants from $\mathcal{V} \cup dom$.

The atom on the left side of $\leftarrow$ is called the head, referred to as $head(\psi)$ and the right hand side consisting of atoms $r_1(\overline{X_1}), \ldots, r_n(\overline{X_n})$ is called the body of the conjunctive query, referred to as $body(\psi)$. The set $\{r_1, \ldots, r_n, q\}$ is called the schema of ψ, also noted $schema(\psi)$. We only consider safe conjunctive queries in which all the variables that are present in the head are also present in the body, i.e. $\overline{Z} \subseteq \overline{X_1} \cup \cdots \cup \overline{X_n}$. We call $\mathcal{CQ}$ the set of conjunctive queries.

Definition 4.1.2 (Conjunctive Query Semantics)**.** *Consider a conjunctive query $\psi : q(\overline{Z}) \leftarrow r_1(\overline{X_1}), \ldots, r_n(\overline{X_n})$ and a database instance $\mathcal{D}$, the semantics of ψ with respect to a database instance $\mathcal{D}$ is given by*

$$\psi(\mathcal{D}) = \{q(\nu(\overline{Z})) | \nu \text{ is a valuation and } r_i(\nu(\overline{X_i})) \in \mathcal{D}, \forall i \in \{1, \ldots, n\}\}.$$

Example 4.1.3. *Consider the following conjunctive query*

$$q(pid, tid) \leftarrow project(pid, pname, pstatus), task(pid, tid, tname, tstatus)$$

where $project(pid, pname, pstatus)$ and $task(pid, tid, tname, tstatus)$ are two relations and constitute the body. The head is $q(pid, tid)$. The conjunctive query returns the tuples with project identifiers pid and their associated task identifiers tid for which there are not only an associated project name pname and project status pstatus, but also a task name tname and task status tstatus.

It is now well-known [Abiteboul et al., 1995] that conjunctive query are monotonic, i.e. for two database instances $\mathcal{D}_1$,$\mathcal{D}_2$ over the same schema such that $\mathcal{D}_1 \subseteq \mathcal{D}_2$, then $q(\mathcal{D}_1) \subseteq q(\mathcal{D}_2)$ for q a conjunctive query..

A *Datalog* query Ψ is a finite set of conjunctive queries. One relation name among the head atoms is defined to be the query predicate. The schema of Ψ is the union of the schema of each conjunctive query composing Ψ. It is noted $schema(\Psi)$. Due to space limitations, we refer to [Abiteboul et al., 1995] for the precise semantics of Datalog query. Each answer of a Datalog query is a fact which relation name is the query predicate.

Example 4.1.4. *Consider the following datalog query*

$q(x, y) \leftarrow edge(x, y)$
$q(x, z) \leftarrow edge(x, y), q(y, z)$

This datalog query describes the transitivity of the relation edge between vertices in a graph. It is recursive, since q, present in the first conjunctive query is also present in the body of the second conjunctive query.■

Let *dom* contains a infinite number of subsets called *domains*. Examples of *domains* include string, integer, floating point numerals, dblp publication identifier, project identifier, etc. Each argument i (or position i) in a relation is associated to a name A_i, which is called an attribute. By convention, attributes will be represented by capital letters A, B and C with or without indices. Overlined capital letters $\overline{A}$, $\overline{B}$ and $\overline{C}$, with or without indices denote tuples of attributes. Each attribute A_i takes its value in one domain, denoted as $dom(A_i)$, which is a subset of *dom*. Each relation r is associated to a tuple of attribute $\left\langle A_1, \ldots, A_{arity(r)} \right\rangle$ which is called its schema, given by $schema(r)$. We also note this as the following expression $r(A_1, \ldots, A_{arity(r)})$. Thus, for a $n-$ary relation name r, and a database instance $\mathcal{D}$, there is:
$\forall r(c_1, \ldots, c_k) \in \mathcal{D}$, if $schema(r) = \langle A_1, \ldots, A_k \rangle$, then $\forall i \in \{1, \ldots, k\}$, $c_i \in dom(A_i) \subseteq dom$.
Let $t \in r(\mathcal{D})$ for a database instance $\mathcal{D}$ with $schema(r) = \langle A_1, \ldots, A_n \rangle$. Then $t[A_{i_1}, \ldots, A_{i_j}]$, with $A_{i_k} \in schema(r)$, $\forall k \in \{1, \ldots, j\}$, is called the projection of t onto attributes $A_{i_1}, \ldots, A_{i_j}$. Moreover $r[\left\langle A_{i_1}, \ldots, A_{i_j} \right\rangle]$ is defined as the set of all different tuples $t[A_{i_1}, \ldots, A_{i_j}]$ for $t \in r(\mathcal{D})$. $r[\left\langle A_{i_1}, \ldots, A_{i_j} \right\rangle]$ is called the projection of r onto attributes $A_{i_1}, \ldots, A_{i_j}$.

Data Dependencies: Data dependencies [Abiteboul et al., 1995]. are used to specify constraints on the various relations. A functional dependency on r, noted $\overline{A} \rightarrow$

$\overline{B}$, exists between two tuples of attributes $\overline{A} = \langle A_1, \ldots, A_m \rangle$ and $\overline{B} = \langle B_1, \ldots, B_n \rangle$, with $A_i \in schema(r)$, $\forall i \in \{1, \ldots, m\}$ and $B_j \in schema(r)$, $\forall j \in \{1, \ldots, n\}$, if

$$\forall (t_1, t_2) \in r(\mathcal{D})^2, \text{if } t_1[A_1, \ldots, A_m] = t_2[A_1, \ldots, A_m], \text{then } t_1[B_1, \ldots, B_n] = t_2[B_1, \ldots, B_n]$$

Example 4.1.5. *Consider a relation $project(pid, pname, pstatus)$. If there's a functional dependency $pid \rightarrow pname$, it signifies that if there are two tuples with same project identifiers, the values for the attribute pname in the two tuples will also be the same.*

The above functional dependency can also be expressed by a full dependency

$$project(pid_1, pname_1, pstatus_1), project(pid_2, pname_2, pstatus_2), pid_1 = pid_2 \Rightarrow pname_1 = pname_2 \blacksquare$$

4.1.2 Relational Model with Access Patterns

Now we consider the relational model with access patterns. We define the notions of access patterns and accesses for relations. We recall that in this thesis we study relations which access is constrained by input and output attributes, since they represent web services API operations. That is, each input position in a relation having an access pattern corresponds to an input argument of the associated web service operation and must be valuated before accessing the relation.

Definition 4.1.6 (Access Pattern [Ullman, 1989b])**.** *Let r be a relation name with $schema(r) = \langle A_1, \ldots, A_n \rangle$. An access pattern α of r is a string made up with letters i and o which length is n such that if the k^{th} letter of the access pattern is an i, then A_k is an input attribute of r, otherwise (i.e. the letter is o) it is an output attribute of r. We then note r^α to say α is the access pattern of r.*

In the litterature, letters b and f are also used for i and o respectively. We define $input(r^\alpha)$ (resp. $output(r^\alpha)$) the function that returns the tuple of input (resp. output) attributes of r^α. $input(r^\alpha)$ is called the input tuple, and $output(r^\alpha)$ the output tuple of r^α. For a precise atom $r^\alpha(\overline{X})$, we define $inVar(r^\alpha)$ (resp. $outVar(r^\alpha)$) the functions that return the tuple of variables for the input (resp. output) attributes of r^α. Moreover, a variable which corresponds to an input (resp. output) attribute is called an input (resp. output) variable.

Definition 4.1.7 (Instance of a relation with acccess pattern)**.** *Consider a database instance $\mathcal{D}$. Let $r^\alpha \in schema(\mathcal{D})$ with $\overline{A_i}$ the input tuple of r^α and $\overline{A_o}$ the output tuple*

of r^α. We suppose r^α corresponds to a web service operation oper. The instance of r^α in $\mathcal{D}$ denoted by $r^\alpha(\mathcal{D})$ is the set of facts $r(\bar{c})$, with $|\bar{c}| = |\overline{A_i}| + |\overline{A_o}|$ and $\bar{c}$ is the tuple of constants obtained as a result when calling oper with the constants of $\bar{c}$ that are in input attributes in r^α as inputs of oper.

Each conjunctive query can be seen as a conjunctive query having access patterns on its relations since a relation having no access pattern can be considered as having one access pattern with only o's (i.e. all attributes are output attributes).

Definition 4.1.8 (Access)**.** *An access to a relation r^α with $\overline{A_i} = \langle A_i^1, \ldots, A_i^n \rangle$ its input tuple is any tuple of constants $\bar{c} = \langle c_1, \ldots, c_n \rangle$ such that $|\overline{A_i}| = |\bar{c}|$ and $c_k \in dom(A_i^k)$, $\forall k \in \{1, \ldots, n\}$.*

If α is made up with o's only, then the only access is the empty tuple.

The first definition of access is given in [Calì et al., 2009a]. Despite it is defined using a query with one atom in the body, it is basically the same notion as our definition. Now we focus on the subclass of $\mathcal{CQ}$ we work on in this thesis: the class of conjunctive queries that have access patterns and are executable.

Definition 4.1.9 (Executable Conjunctive Query - $\mathcal{CQ}^\alpha$ syntax)**.** *Let ψ be in $\mathcal{CQ}$ having access patterns on its body relations. ψ is an executable conjunctive query if, for every relation r_i, $\forall i \in \{1, \ldots, n\}$, and for every term t (i.e. variable or constant) associated to an input attribute of r_i, t is either a constant or a variable associated to an output attribute of one or more r_k, for $k \in \{1, \ldots, i-1\}$.*

We note $\mathcal{CQ}^\alpha$ the set of all safe executable conjunctive queries expressed with relations having access patterns.

Thus, each element ψ of $\mathcal{CQ}^\alpha$ has the following form: $\psi : q(\overline{Z}) \leftarrow r_1^{\alpha_1}(\overline{X_1}), \ldots, r_n^{\alpha_n}(\overline{X_n})$ where $\overline{Z}$,$\overline{X_1}, \ldots \overline{X_n}$ are tuples of variables from $\mathcal{V}$ or constants from *dom* linked to the domain of the associated attributes of q and $r_i, i \in \{1, \ldots, n\}$. $\alpha_1, \ldots, \alpha_n$ corresponds to the access patterns of the relations $r_1, \ldots, r_n$ respectively.

The notion of executable conjunctive query reflects the notion of ordered calls of a sequence of operations, in which some values which are outputs of some operations are used as inputs for others.

4.2 ETL of Data coming from Web Services

The only interface available to us as a third party user to get the enterprise data from the web services is through the API. Our analysis of the web service API in section 2.3 showed that the API of web services are significantly different from each other. Our goal is to extract, transform and load relevant enterprise data to the data warehouse using the APIs with minimum development effort.

4.2.1 ETL in Classical Data Warehouse

A typical data warehouse is able to generate summarized and aggregated data from detailed records. Therefore a data warehouse plays a significant role in the performance measurement requirements of an enterprise. Obtaining the business analytics (often using the data warehouses) is referred to as online analytical processing (OLAP) [Chaudhuri and Dayal, 1997]. To this purpose, a data warehouse uses certain tools for extracting data from various sources, often referred to as ETL(extract-transform-load) tools. ETL program [Vassiliadis and Simitsis, 2009] is "*any kind of data processing software that reshapes or filters records, calculates new values, and populates another data store than the original one*". ETL process [Trujillo and Luján-Mora, 2003] involves 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. Various methods involved during the process of extraction, transformation and loading have been extensively studied in [Vassiliadis, 2011].

4.2.2 Web Services as Data sources

For DaWeS, we consider the web service APIs as the data sources since they are the only authorized interfaces available to us. The crucial point is to view every web service API operation as a relation with access pattern. For a web service API, an access pattern describes:

- input and output parameters of every API operation.
- and data types of both the input and output parameters.

4.2.3 Recalls about Data Integration

In the data integration field, mediation is a virtual approach in which a uniform query interface is provided to a multitude of heterogeneous and autonomous data sources. Every data source has its own schema. There is a mediated or global schema over which the user queries are formulated. Global schema relations and the local schema relations are linked with the help of mappings which are relation definitions expressed with respect to either global schema relations or source schema relations. Any query posed over the global schema is translated to a query involving the source schemas.

The data source relations constitute the local schema. The data sources are the original sources of information. Every data source has its own schema. The users of a data integration system may or may not have direct control over the schema of the data sources.

Every data source must be mapped to the global schema. There are various mapping mechanisms like Global as View, Local as View and Global-Local as View Mapping. We consider here the local as view mapping given its scalable nature that is the data integration system can easily work when a new data source is added. In a local as view mapping, the local schema is defined using the global schema. There are several ways to do this mapping. But we consider here the mapping using conjunctive query.

Recall that the queries are formulated over the global schema. But the actual data is situated in the data source. Therefore a query using the global schema must be translated to a query using the local schema. This query translation is called query rewriting. Classical query rewriting algorithms include bucket algorithm, minicon algorithm [Pottinger and Halevy, 2001] and inverse rules algorithm [Duschka and Genesereth, 1997]. A detailed analysis of these algorithms and various other data integration mechanisms has been done in [Halevy, 2001].

Example 4.2.1. *LAV based data integration is shown in the Figure 4.1. Consider two sources source* 1 *with relations Article, Ref and source* 2 *with relations BiblioAuthor. These relations cannot be directly accessed but by rather four views* $V1, V2, V3$ *and* $V4$ *defined in SQL. Only the source relations are materialized and not the view definitions. We have two virtual global schema relations Cite and SameTopic, hence they have no materialized data. The end user is exposed only to the global schema and formulates her query over it. In LAV mapping, the views are defined using the global schema relations. We assume open world assumption (OWA) where any data not present in the database may be true unlike close world assumption (CWA) where such absent data are assumed to be false. We are only interested in certain answers, i.e., the query response*

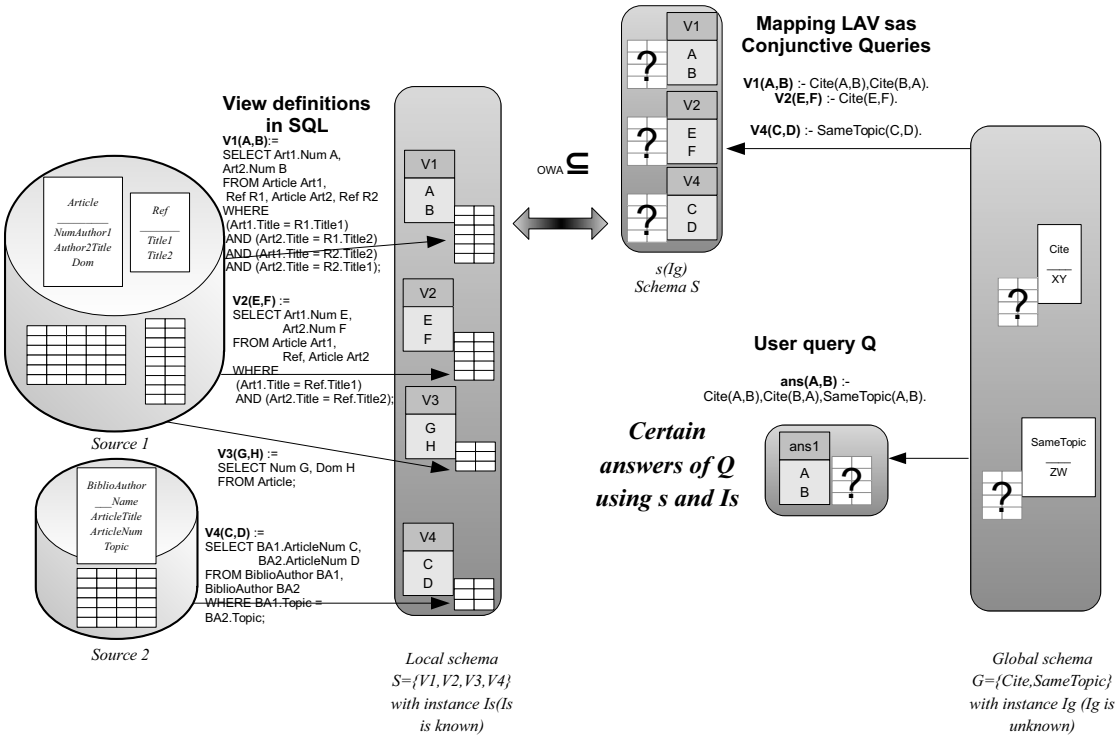

Figure 4.1: LAV based Data Integration

is true in all the data sources that check LAV mappings. Queries formulated over the global schema (here, $ans(A, B) : -Cite(A, B), Cite(B, A), SameTopic(A, B)$.) must be translated or rewritten using the view definitions. ? signifies that the associated relation is virtual (i.e., there is no associated extension). The query rewriting thus obtained is evaluated to obtain the certain answers. ■

4.2.4 Mediation as ETL

To deal with ETL for data coming from web services, we propose to use mediation. For every web service, each API operation can be viewed as a relation with an access pattern, then mediation can be adopted to these sources provided access patterns can be handled in the rewriting process. We recall that it is possible in the next section. So this proposition is a virtual approach for data materialization to implement the first tier of DaWeS.

Access Pattern: A relation can also be associated to an access pattern [Ullman, 1989b] whose size is equal to the number of attributes in a relation. Syntactically, the access pattern is represented by an adornment being a tuple of i (or b) and o (or f) letters written besides the relation name. In this tuple, i (or b for "bound") in i^{th} position says the i^{th} attribute is an input ; o (or f for "free") says it is an output.

Executable Conjunctive Query: A conjunctive query is called an executable conjunctive query if, for every relation with access pattern in the body of the query, the input parameters are present as an output parameters in one or more previous relation (while checking from left to right of the body) or are themselves constants.

Example 4.2.2. *In Figure 4.2, we consider the domain* Project Management *that manages mainly two resources, Project and Task; hence two global schema relations:*

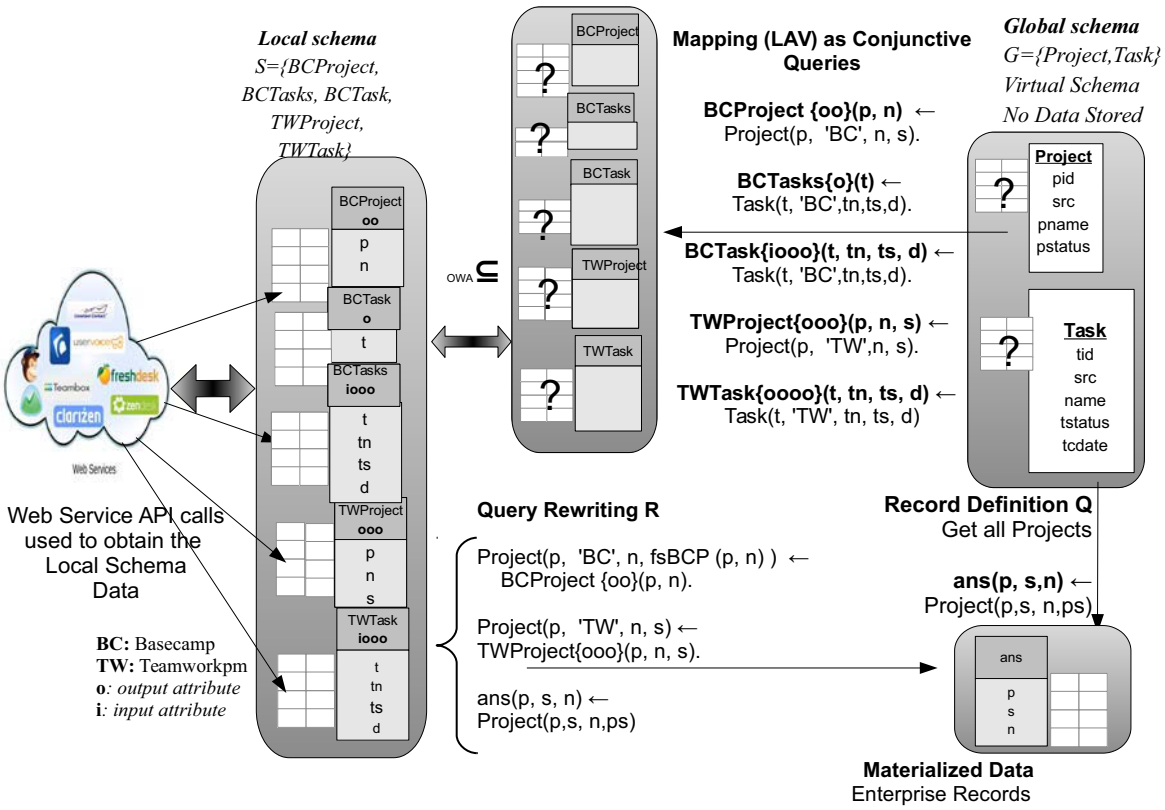

Figure 4.2: Mediation as ETL

$Project(pid, src, pname, pstatus)$ *and* $Task(tid, src, name, tstatus, tcdate)$.

We consider two web services: Basecamp and Teamwork. We consider here a simplified version of their API operations. Basecamp has one operation BCProject that

returns all the projects taking no input attributes. It also provides two operations related to task: the first operation BCTasks provides all the task identifiers and the second BCTask requires the task identifier as input to give the complete task details. This information is captured by the access patterns. Consider LAV mapping $BCTask^{iooo}(t, tn, ts, td) \leftarrow Task(t,' BC', tn, ts, td)$. *It corresponds to the fact that the operation BCTask takes as input the task identifier t and gives the details of the task (name, status and creation date). The operation has been mapped to the global schema relation Task with source value as BC to signify Basecamp, its source. Now consider a record definition q:* Get all Projects*, (a conjunctive query formulated over the global schema).*

$$q(p, s, n) \leftarrow Project(p, s, n, ps).$$

This query must be rewritten using the source relations. Here is an example query rewriting (obtained by inverse rules query rewriting discussed below). This is a simplified rewriting and detailed version will be discussed in example 4.2.3.

$Project(p,' BC', n, f_{BCP,4}(p, n)) \leftarrow BCProject^{oo}(p, n).$
$Project(p,' TW', n, s) \leftarrow TWProject^{ooo}(p, n, s).$
$q(p, s, n) \leftarrow Project(p, s, n, ps).$ ■

Query answering under limited access patterns has also been studied in [Rajaraman et al., 1995], [Li and Chang, 2001a], [Duschka et al., 2000], [Deutsch et al., 2007]. We choose the inverse rules algorithm given its unique capability [Duschka et al., 2000] to handle at the same time recursive datalog query, limited access patterns in the database and full and functional dependencies in the mediated schema.

4.2.5 Inverse Rules Query Rewriting

Inverse Rules query rewriting algorithm computes the maximally contained rewriting that has the property of computing the certain answers. Certain answers signifies those answers that are true for any database instance of the global schema that is compatible with LAV mappings (i.e., local schema relations that can be defined using the global schema relations). Certain answers follow very strict semantics. The steps of the algorithm has been described in Figure 4.3.

We consider an example to explain the steps of inverse rules query rewriting.

Example 4.2.3. *Consider a record definition q:* Get all Projects from Teamwork*, (a*

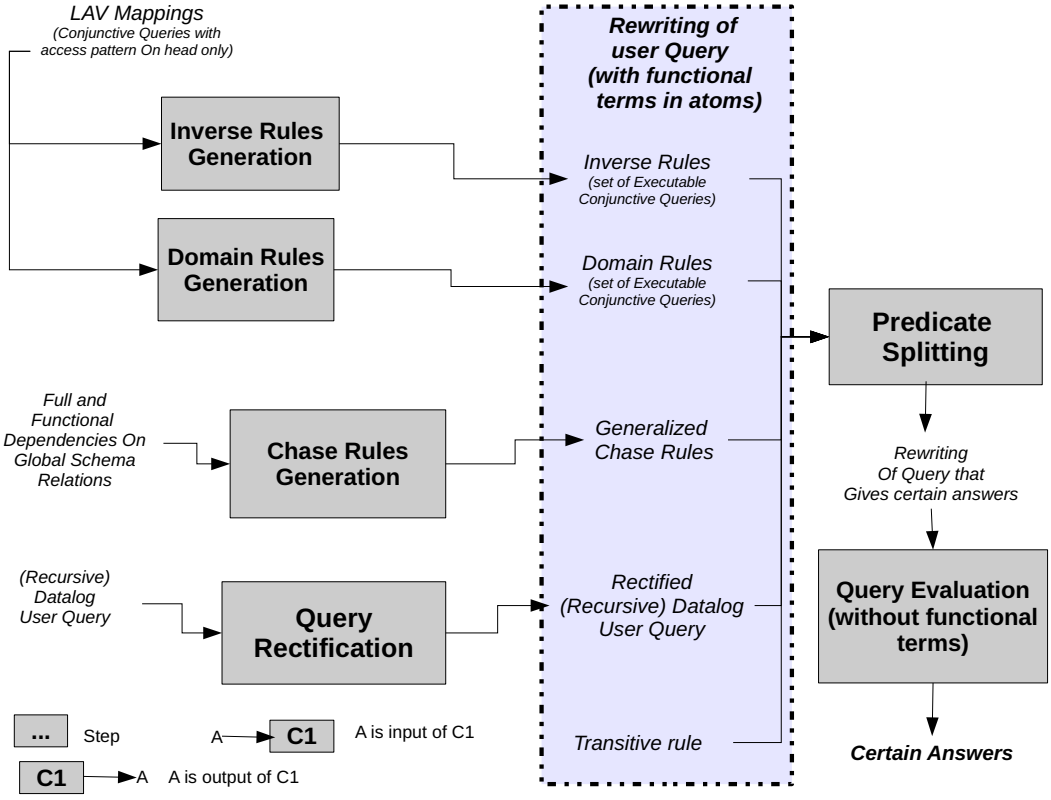

Figure 4.3: Inverse Rules Query Rewriting Algorithm

conjunctive query formulated over the global schema) continuing with the same sources as in example 4.2.2.

$$q(p, n, ps) \leftarrow Project(p,'TW', n, ps).$$

Consider the Figure 4.3, Inverse Rules generation *generates the following rules (set of executable conjunctive queries). For every LAV mapping, it generates rules with every body atom in the LAV mapping as the head atom and the head of the LAV mapping as the body of the new rules. The unsafe variables in the head of the rule so created are replaced by functional terms. But the rules so generated may be unexecutable and hence domain rules (discussed below) are used to generate an executable query plan.*

$$Project(p,'BC', n, f_{BCP,4}(p, n)) \leftarrow BCProject^{oo}(p, n).$$

$Project(p,'TW',n,ps) \leftarrow TWProject^{ooo}(p,n,ps).$

Note that the two above rules are executable (since they don't have any input attribute). See example 4.2.4 that demonstrates a query plan with input attributes.

Domain Rules generation *generates the following rules:*

$dom_p(p) \leftarrow BCProject^{oo}(p,n).$
$dom_n(n) \leftarrow BCProject^{oo}(p,n).$
$dom_p(p) \leftarrow TWProject^{ooo}(p,n,ps).$
$dom_n(n) \leftarrow TWProject^{ooo}(p,n,ps).$
$dom_{ps}(ps) \leftarrow TWProject^{ooo}(p,n,ps).$

If we have the following functional dependencies over the global schema relation $Project$*:*

$Project : p,s \rightarrow n$
$Project : p,s \rightarrow ps,$

they are used by Chase Rules Generation *to generate generalized chase rules.*

$e(n1,n2) \leftarrow Project(p1,s1,n1,ps1), Project(p2,s2,n2,ps2), e(p1,p2), e(s1,s2)$
$e(ps1,ps2) \leftarrow Project(p1,s1,n1,ps1), Project(p2,s2,n2,ps2), e(p1,p2), e(s1,s2)$

where e*, the equality predicate is a binary relation whose intended meaning is* $=$ *i.e.,* $e(p_1,p_2)$ *signifies* $p_1 = p_2$*.*

The transitive rule is $e(X,Z) \leftarrow e(X,Y), e(Y,Z)$*. This rule signifies the transitivity of the equality predicate.*

The user query is then rectified:

$$q(p1,n1,ps1) \leftarrow Project(p1,s1,n1,ps1), e(p1,p), e(n1,n), e(ps1,ps), e(s1,'TW').$$

Therefore the complete rewriting of the user query is

$Project(p,'BC',n,f_{BCP,4}(p,n)) \leftarrow BCProject^{oo}(p,n).$
$Project(p,'TW',n,ps) \leftarrow TWProject^{ooo}(p,n,ps).$
$dom_p(p) \leftarrow BCProject^{oo}(p,n).$
$dom_n(n) \leftarrow BCProject^{oo}(p,n).$
$dom_p(p) \leftarrow TWProject^{ooo}(p,n,ps).$
$dom_n(n) \leftarrow TWProject^{ooo}(p,n,ps).$

$dom_{ps}(ps) \leftarrow TWProject^{ooo}(p, n, ps).$
$e(n1, n2) \leftarrow Project(p1, s1, n1, ps1), Project(p2, s2, n2, ps2), e(p1, p2), e(s1, s2)$
$e(ps1, ps2) \leftarrow Project(p1, s1, n1, ps1), Project(p2, s2, n2, ps2), e(p1, p2), e(s1, s2)$
$e(X, Z) \leftarrow e(X, Y), e(Y, Z)$
$q(p1, n1, ps1) \leftarrow Project(p1, s1, n1, ps1), e(p1, p), e(n1, n), e(ps1, ps), e(s1,'TW').$

Datalog query must not contain any functional terms. So the predicate splitting step [Duschka et al., 2000] is used to remove the functional terms. Then the resulting datalog program is evaluated by a datalog engine to get the query response (certain answers). ■

In the above example, the domain rules are not used since the domain predicates *dom* didn't appear in the body of any executable conjunctive query. In the following example, we see how domain rules and executable conjunctive queries can be used to handle pagination (for the sake of simplicity, we don't show any query rectification, generalized chase rules and transitive rule). Operations that require pagination signifies that the same operation call must be repeated multiple times (with different parameters like page number and page size) to obtain the complete details of the resource. An example for this is the search engine results page. The search engine returns the total number of results and splits the results into multiple pages. To get all the search results, one must check every individual page. Following example shows how we handle pagination in DaWeS for such operations. The example also shows how query responses can be further used to build performance indicators.

Example 4.2.4. *: Let us consider three web services 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 web services name, and operation is an operation name.

We want to be connected to these services so that customers can get performance indicators about the handling of their complaints. Towards this purpose, each web services offers at least one operation callable through its API (see table 4.2.4). The global

Table 4.1: Helpdesk web services 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*

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):

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

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

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

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

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

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

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

In the context of example 4.2.4, 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.*

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

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

$\mathbf{Page}(pgno,' Desk\ v2\ API',' Deskv2Case', pgsize) \leftarrow \mathbf{D2TC}^{oo}(pgno, pgsize).$
$\mathbf{DPgNo}(pgno) \leftarrow \mathbf{D2TC}^{oo}(pgno, pgsize).$
$\mathbf{DPgSize}(pgsize) \leftarrow \mathbf{D2TC}^{oo}(pgno, pgsize).$
$\mathbf{Page}(pgno,' Desk\ v2\ API',' Deskv2Case', pgsize) \leftarrow$
$\quad \mathbf{DPgNo}(pgno), \mathbf{DPgNo}(pgsize), \mathbf{D2C}^{iiooooo}(pgno, pgsize, tkid, tkn, tkcd, tkp, tks)$
$\mathbf{Ticket}(tkid,' Desk\ v2\ API', tkn, tkcd, f_{D2C,5}(pgno, pgsize, tkid, tkn, tkcd, tkp, tks),$
$\quad f_{D2C,6}(pgno, pgsize, tkid, tkn, tkcd, tkp, tks), tkp, tks) \leftarrow$
$\quad \mathbf{DPgNo}(pgno), \mathbf{DPgSize}(pgsize), \mathbf{D2C}^{iiooooo}(pgno, pgsize, tkid, tkn, tkcd, tkp, tks)$
$\mathbf{Ticket}(tkid,' Zendesk\ v1\ API', f_{ZT,3}(tkid), f_{ZT,4}(tkid), f_{ZT,5}(tkid),$
$\quad f_{ZT,6}(tkid), f_{ZT,7}(tkid), f_{ZT,8}(tkid)) \leftarrow \mathbf{ZT}^{o}(tkid).$
$\mathbf{Ticket}(tkid,' Zendesk\ v1\ API', f_{ZST,3}(tkid), f_{ZST,4}(tkid), f_{ZST,5}(tkid), f_{ZST,6}(tkid),$
$\quad f_{ZST,7}(tkid), Closed') \leftarrow \mathbf{ZST}^{o}(tkid).$
$\mathbf{ZTID}(tkid) \leftarrow \mathbf{ZST}^{o}(tkid).$
$\mathbf{Ticket}(tkid,' Zendesk\ v1\ API', tkn, tkcd, tkdd, tkcmpd, tkp, tks) \leftarrow$
$\quad \mathbf{ZTID}(tkid), \mathbf{ZTD}^{iooooo}(tkid, tkn, tkcd, tkdd, tkcmpd, tkp, tks).$
$\mathbf{Page}(pgno,' Uservoice\ v1\ API',' Uservoicev1Ticket', pgsize) \leftarrow \mathbf{UTT}^{oo}(pgno, pgsize).$
$\mathbf{UPgNo}(pgno) \leftarrow \mathbf{UTT}^{oo}(pgno, pgsize).$
$\mathbf{UPgSize}(pgsize) \leftarrow \mathbf{UTT}^{oo}(pgno, pgsize).$
$\mathbf{Page}(pgn,' Uservoice\ v1\ API',' Uservoicev1Ticket', pgs) \leftarrow$
$\quad \mathbf{UPgNo}(pgno), \mathbf{UPgSize}(pgsize), \mathbf{UT}^{iiooooo}(pgn, pgs, id, tkn, tkcd, tks, tkp)$
$\mathbf{Ticket}(id,' Uservoice\ v1\ API', tkn, tkcd, f_{UT,5}(pgn, pgs, id, tkn, tkcd, tks, tkp),$
$\quad tkcmpd, tkp, tks) \leftarrow$
$\quad \mathbf{UPgNo}(pgno), \mathbf{UPgSize}(pgsize), \mathbf{UT}^{iiooooo}(pgn, pgs, id, tkn, tkcd, tks, tkp).$

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';

DNT when executed daily for a period (like 30 days) can be used to extract all the tickets that were created on the previous day on various web services for the specified period (like 30 days). The above performance indicator when executed gives the count of the tickets that were created during the past 30 days using the query responses from DNT. ■

4.3 Generic Wrapper for Web Services

Mediation traditionally uses wrappers to make sources appear as relations. Wrappers [Roth and Schwarz, 1997] encouraged the enterprises not to scrap their legacy data stores but to rather wrap them to make use of them. Wrapping a data source corresponds to querying these legacy data stores and inferring various information from them in a desired format. Thus we require a wrapper for our web services. For a single enterprise working with ten different web services, developing a wrapper for each of the different web service is feasible, but in the multi-enterprise and multi-web service context, this is not feasible except if some advanced languages (WSDL) are used to automatically generate wrappers. Thus we need a generic wrapper that is able to wrap the different web services in the most declarative manner. Wrappers are the place where some heterogeneity can be handled (e.g., date formats, enumerations).

Thus our proposal for the generic data wrapper must make use of the web service interface (web service API) to extract the data from these web services.

So we require a generic wrapper for extracting data from various web services using their API with features like declarative approach and easier handling of the web services and their heterogeneous and autonomous nature. It is possible since web services are less heterogeneous compared to the classical mediation systems which has to deal with various diverse sources like relational, semi-structured and textual sources. Web services are already computer operational sources (and not human language only sources). That's why we talk about a 'Generic Wrapper'. We can't generate them automatically since web services are not described with advanced languages. So we study a manual solution:

- one wrapper that is configurable for each web service.

- to make it feasible to configure a lot of web services, this configurable step must be done using declarative languages and not (imperative) programming languages.

To configure the generic wrapper, we consider the services that use XML (or JSON) as message formats for the web service API communication. To validate the response obtained from the web services, we require the expected schema of the operation reponse generally available as XSD [Gao et al., 2009]. To extract desired information from the operation response, we also need a transformer using XQuery [Fernández et al., 2002] or XSLT [Kay et al., 2007].

Conclusion Our goal is to shift ETL from development to administration by using declarative languages to link sources to the global schema and to configure the generic wrapper.

Chapter 5

DaWeS: Data Warehouse fed with Web Services

A data warehouse is fed with the data coming from different data sources using various ETL (Extraction, Transformation and Loading) tools. In a classical data warehouse settings, the data sources are the legacy databases of various departments (or business units), textual documents, spreadsheets and sometimes web pages. Wrappers are used to extract data from these sources. Our data sources are the web services used by the enterprises whose data can be accessible using the respective web service APIs.

Mediation approach in data integration field provides a uniform query interface to heterogeneous and autonomous data sources. We use the mediation approach to feed DaWeS. In this chapter, we will discuss how the mediation approach along with a generic wrapper can be used as an ETL for the data coming from the web services. We will also discuss the detailed architecture and development of DaWeS.

5.1 Two Tiered Architecture of DaWeS

DaWeS employs a two-tiered architecture as shown in Figure 5.1. There are primarily two main categories of users: DaWeS administrators and DaWeS users (enterprises). DaWeS administrator looks for new web service APIs to integrate them with DaWeS. She also defines new record definitions which when executed using the ETL tool gives the (historical) enterprise data from the web services. These enterprise data form the enterprise records. She also defines new performance indicators using the record

schema (record definitions). The performance indicator definitions are evaluated by the *Query processing* making use of the enterprise records. It produces enterprise performance indicators (aggregated data). The ETL component makes the web service API operation calls, receives the operation responses, transforms them to a desired format and store the result to the enterprise record database.

Record schema and enterprise records are managed in the *Record Tier* and performance indicators schema and values (enterprise performance indicators) are managed by the *Performance indicator Tier*.

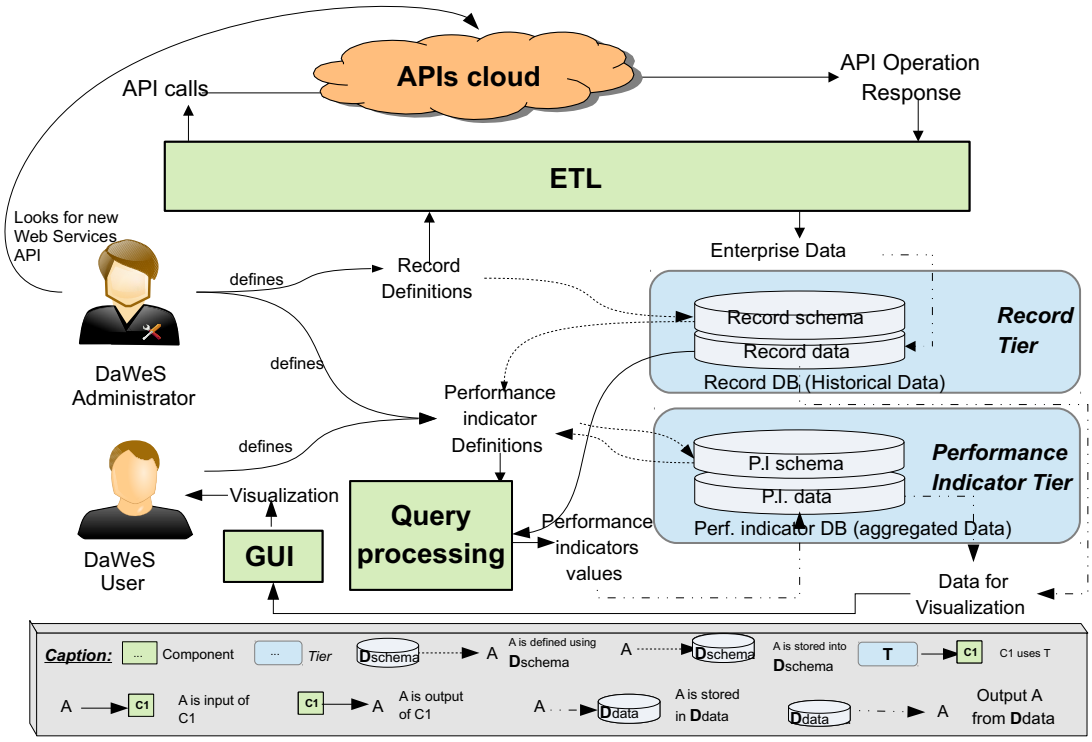

Figure 5.1: DaWeS: Overall Picture of Two Tiered Architecture

A DaWeS user makes use of the *GUI (Graphical User Interface)* to visualize the performance indicators and records of her enterprise. She can also define new performance indicators using the record schema. She (and DaWeS administrator) can also define new performance indicator definitions using the already defined performance indicator definitions. In the upcoming sections, we will take a detailed look at all these components.

5.2 Architecture and Development

Before delving deep into the architecture of DaWeS, we first present the basic overview of its architecture (see Figure 5.2). Then we go into a detailed analysis of its individual components.

5.2.1 Overview

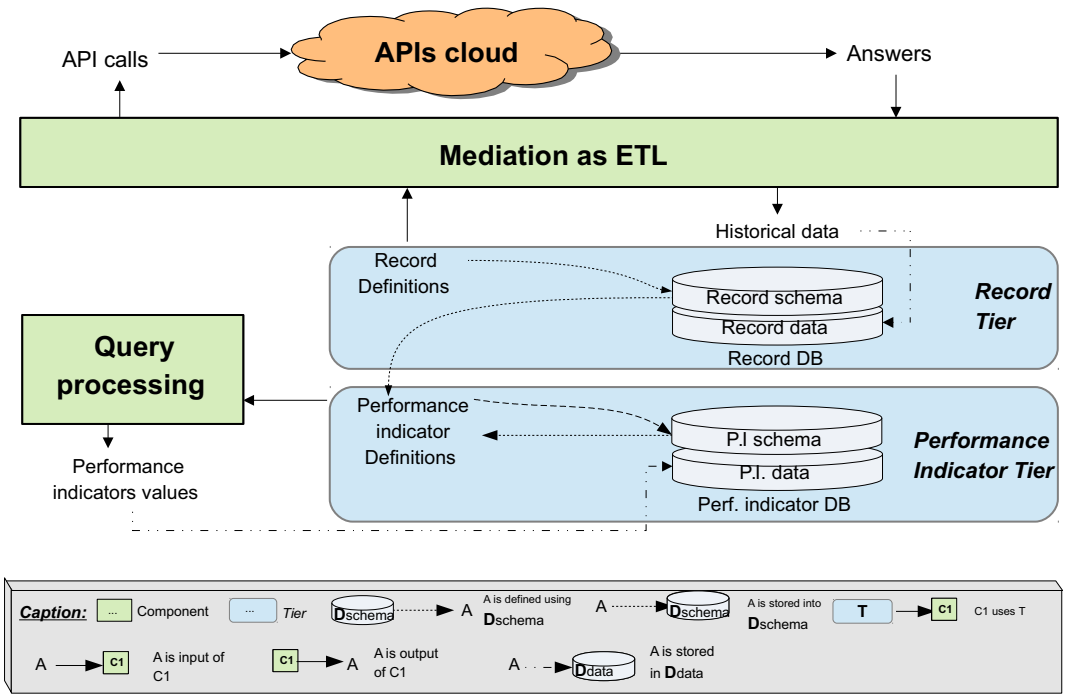

Figure 5.2: DaWeS: Two Tiered Architecture

Records and Performance indicators constitute the two-tiered architecture of DaWeS.

Two-Tiered Architecture Tier One, also referred to as the *Records Tier* manages the (historical) enterprise data obtained from the web services API. Tier two (*Performance Indicators Tier*) uses the enterprise records to compute business performance indicators. Mediation is used as an ETL approach to obtain data from the web service API. Mediation handles query rewriting, web service API operation calls using a

generic wrapper (section 5.2.2.6) and query evaluation. The two tiered architecture is shown in Figure 5.2.

The two tiered architecture is further explained by Fig. 5.3 that shows the basic architecture of DaWeS. The figure also differentiates between various automated and manual efforts. Web service API operations are manually defined. For every web service API operation, DaWeS administrator refers the API documentation to get the details of the operation like HTTP information (HTTP url, body and header of operation request), the expected response schema (XSD) and the transformation (XSLT) required to extract the relevant information from the operation response. DaWeS administrator also creates record definitions and some (default) performance indicator queries. All of these constitute the manual effort. Handling of queries (both record definitions and performance indicator queries) and their responses and making web service API operation calls are automated.

Query rewriter rewrites the query formulated over the global schema relations to query plan with web service API operations using inverse rules algorithm. Mediation as an ETL is achieved by the generic HTTP web services wrapper and answer builder. The answer builder executes the query plan and makes API operation calls using the generic wrapper. The query responses constitute the enterprise records (e.g., *Daily New Tickets* in example 4.2.4). The records so computed are used to compute the performance indicators. Query evaluator uses the underlying DBMS to evaluate performance indicator queries (or SQL queries) to compute enterprise performance indicators (e.g. *Total High Priority Tickets Registered in a month* in example 4.2.4).

DaWeS Tier One: Record Tier - Extracting relevant information from web services) When DaWeS has all the information required (e.g., authentication parameters) from an enterprise to fetch the relevant information from the web services, DaWeS executes every query formulated over the global schema. The query evaluation engine performs various tasks given below:

1. Reads the query (datalog query formulated over the global schema) from the DaWeS database
2. Rewrites this query formulated over the global schema relations into a query expressed with respect to web service API operations using the inverse rules query rewriting algorithm
3. During the process of query evaluation (more precisely, the query evaluation step is the evaluation of the rewriting), the following steps are performed for every local schema relation (web service API operation):

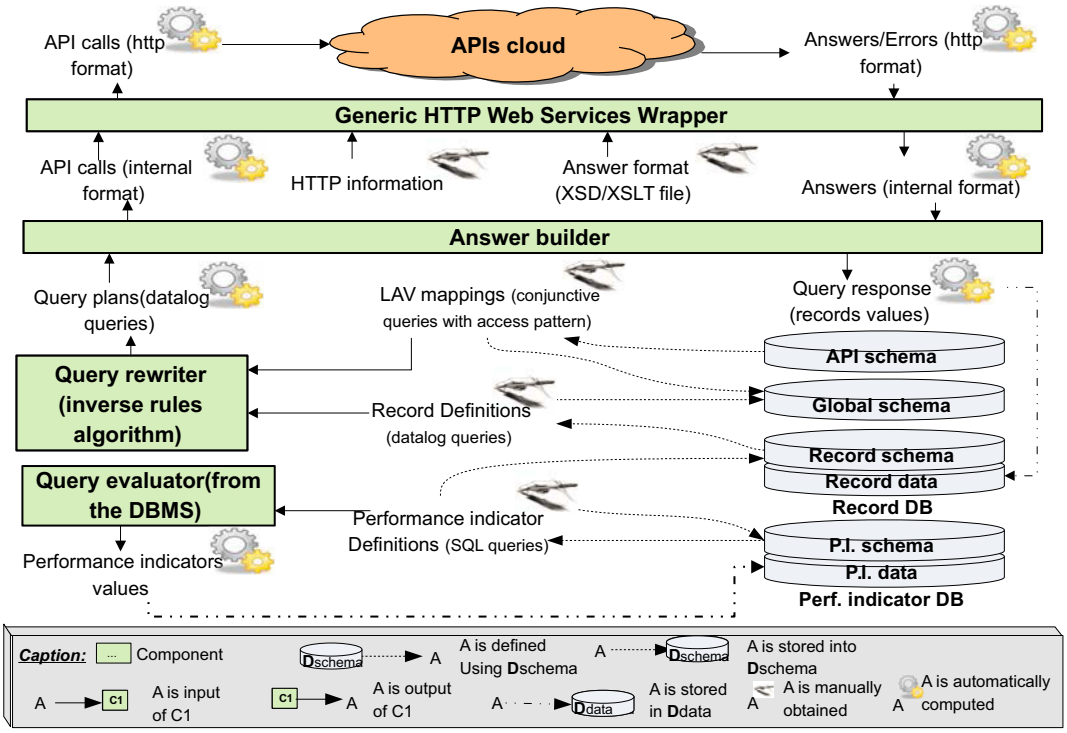

Figure 5.3: Basic Architecture

(a) On receiving the operation name, the generic wrapper performs the following tasks:

i. Check the cache to see whether the operation call is recently made. If the result is available, return the cached response.

ii. Read the authentication and authorization parameters of the corresponding enterprise

iii. Form the HTTP request call (the HTTP URL, headers and body)

iv. Make the request and wait for the response (or error)

(b) A non-error response is validated using the expected operation response schema

(c) A validated response is transformed to a format understood by the query evaluation engine using transformers

(d) The transformed response is stored in the cache for future use

(e) Return the transformed response for the query evaluation

4. The query response (or error) is stored in the database as an enterprise record

DaWeS Tier 2: Performance Indicator Tier- Computing business performance measures The records obtained from the evaluation of the query formulated over the global schema are used to compute the business performance measures. Business performance measures are SQL queries and are evaluated by making use of the enterprise records. The business performance measures are computed in the following manner

1. Read the business performance query definition from the DaWeS database
2. For every enterprise record required to compute this business performance
   (a) Read the corresponding enterprise records for the given period from the database
   (b) Make sure that the enterprise records for the given period are available
   (c) Ensure that none of the enterprise records for the given period have errors (every enterprise record have an associated field that tells whether the record computation was a success or a failure)
3. Compute the performance measure and store the result (error) in the DaWeS database

Configuring DaWeS

There are two types of users: DaWeS administrators and enterprise users. DaWeS administrators decide which web services they want to integrate DaWeS with. They find the corresponding domain of these web services. For every domain, they decide the relevant global schema relations (considering the various interesting business measures that an enterprise may be interested in). Once the global schema relations and the relevant attributes are created, every relevant web service operation (operation request and operation response) are mapped to the global schema using the LAV mapping. Queries are formulated over the global schema. Using only these queries, various business measures (queries) are created and exposed to the end users.

Enterprise users search for the available web services and authorize DaWeS to query the web services. Authorization and authentication parameters are stored in DaWeS. Enterprise users also decide what information they want to extract from the web services and store in the DaWeS database. Precisely it means that they choose what record definitions they authorize DaWeS to execute. They also choose interesting business measure queries from the default offer or create new ones that suit their requirements.

5.2.2 Detailed Architecture

We now discuss the detailed architecture of DaWeS. Figure 5.4 shows the various modules and also differentiates between automated and manual efforts. The main modules involved concern DaWeS Database, scheduler, enterprise business performance measure computation, rewriting generator, answer builder, generic web service wrapper (response validation, valid response transformation, response cache, generic HTTP Web Service API Handler, failure handling), calibration and search.

5.2.2.1 DaWeS Database

DaWeS Database is used for storing web service description, global (mediated) schema, record schema, performance indicators queries, enterprise authentication parameters, enterprise records and performance and indicators. They are shown in the Figures 5.5, 5.6, 5.7, 5.8, 5.9 and 5.10. In all these tables ID is the primary key. A detailed look at the various relational tables is discussed in section D.2. Here we present a quick overview of these tables.

Figure 5.5 shows the information related to the web service. For every web service, we collect the information regarding the various categories (or domains) it belongs to. Examples of categories include *Project Management*, *Email Marketing* etc. The administrator also registers the various API of the web services and the details of the service providers. Figure 5.6 describes the various information that is essential to describe the web service API. For every API, the administrator must collect the information related to the message formats, state (current, deprecated or active), authentication parameters required from the enterprises and from DaWeS Administrator (for OAuth 1.0). It also shows the various details captured for every API operation like the expected response schema (XSD), the desired transformation (using XSLT) and HTTP details of request. Every API operation has an associated local schema relation as shown in Figure 5.7. Local schema relations are described using the global schema relations using LAV mapping (conjunctive query). Both local schema and global schema relations have their attributes and the corresponding data types described. Figure 5.8 shows how records (datalog queries) and performance indicator queries are stored. Records definitions are (recursive) datalog queries. Every record definition and every performance indicator has an associated frequency of execution that tells the scheduler (section 5.2.2.2) how often they must be computed. They also have associated calibration test data (section 5.2.2.3) to ensure their proper computation. Figure 5.9

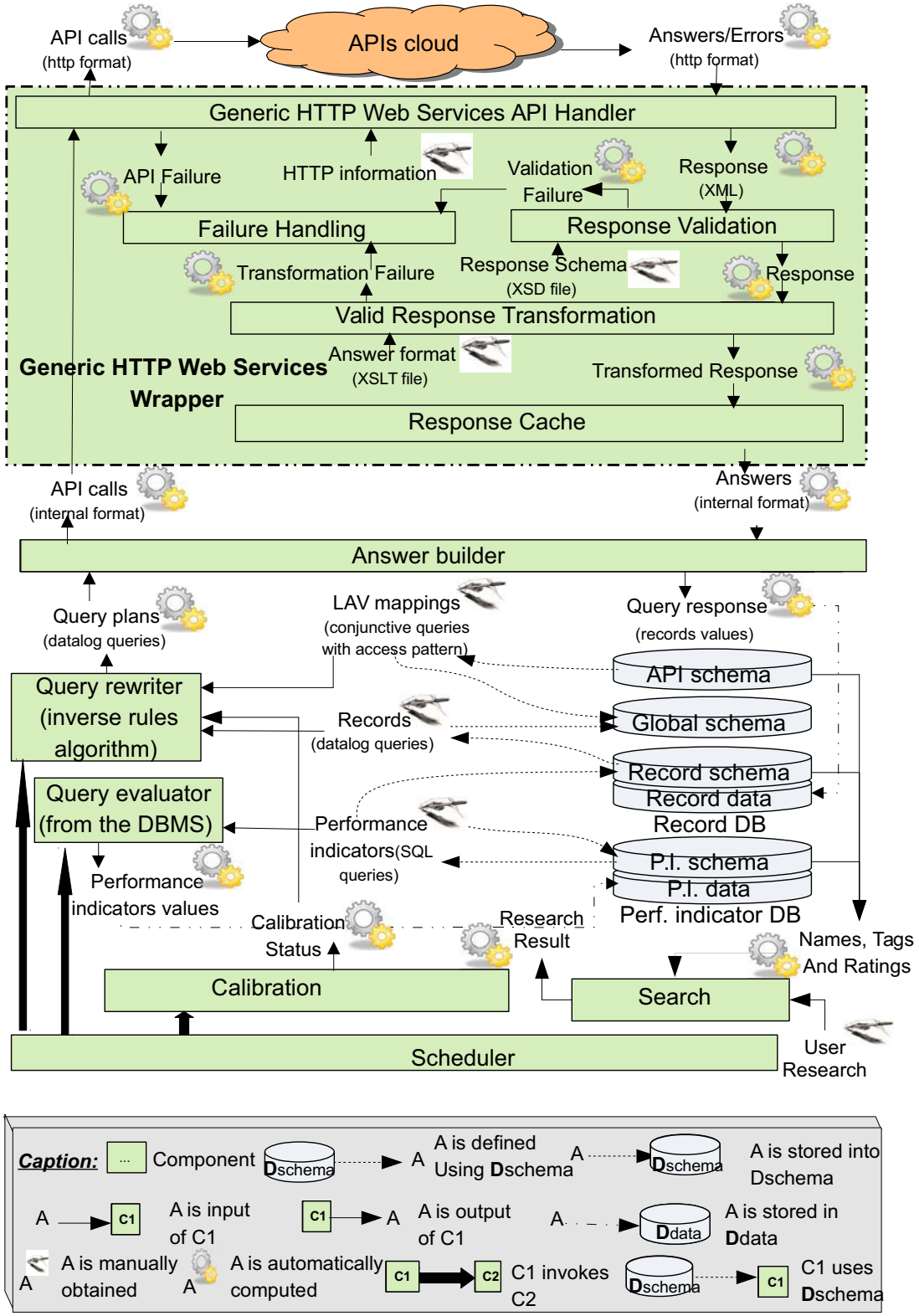

Figure 5.4: DaWeS: Detailed Architecture

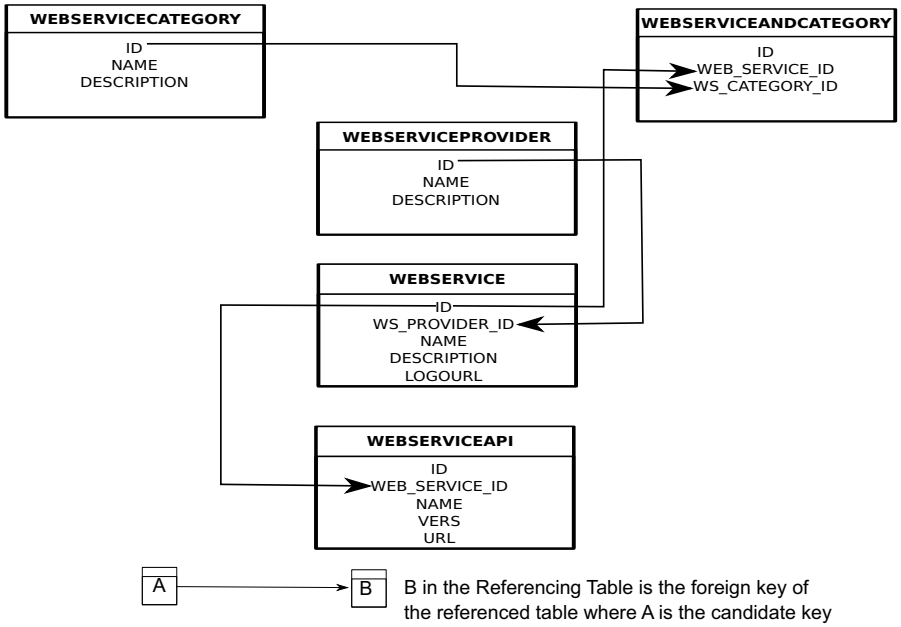

Figure 5.5: Web Service

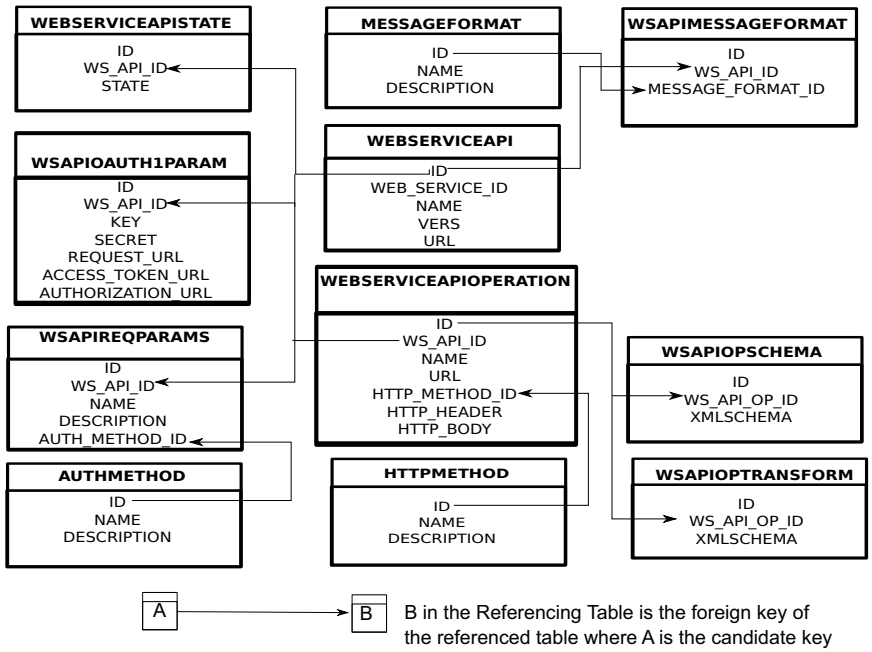

Figure 5.6: Web Service API

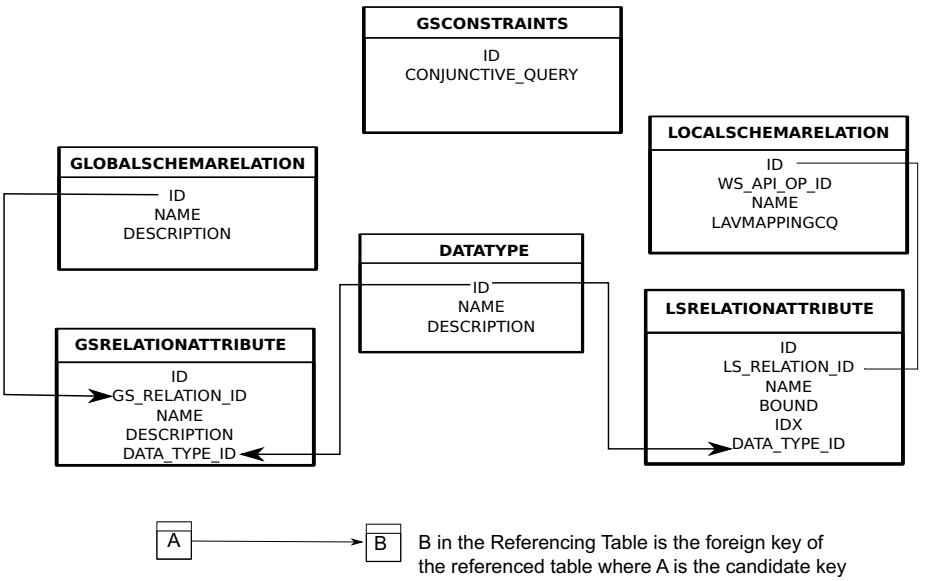

Figure 5.7: Local and Global Schema

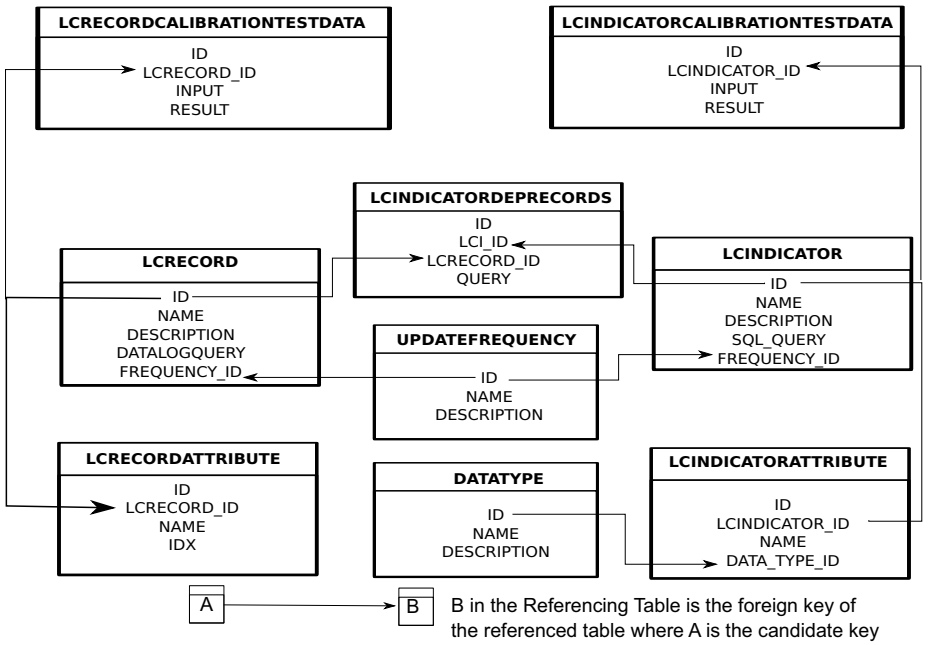

Figure 5.8: Record Definitions and Performance Indicators

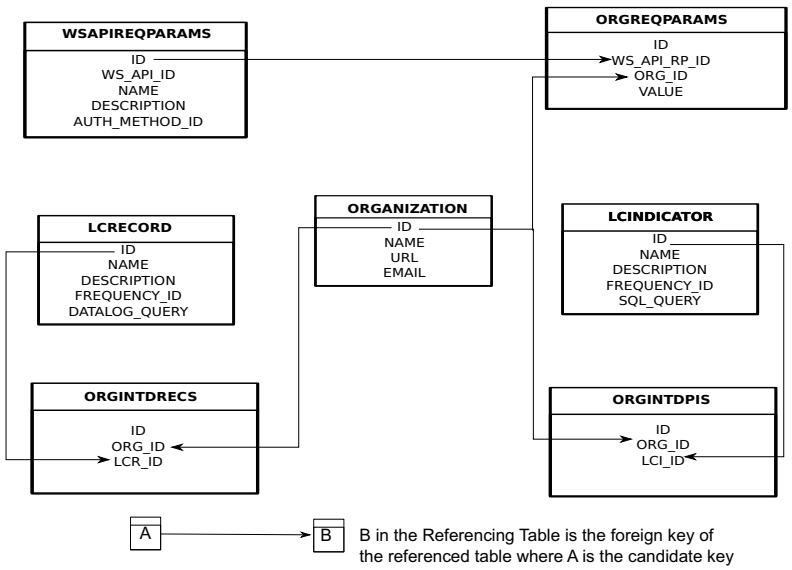

Figure 5.9: Organization, its authentication params and interested Record Definitions and Performance Indicator

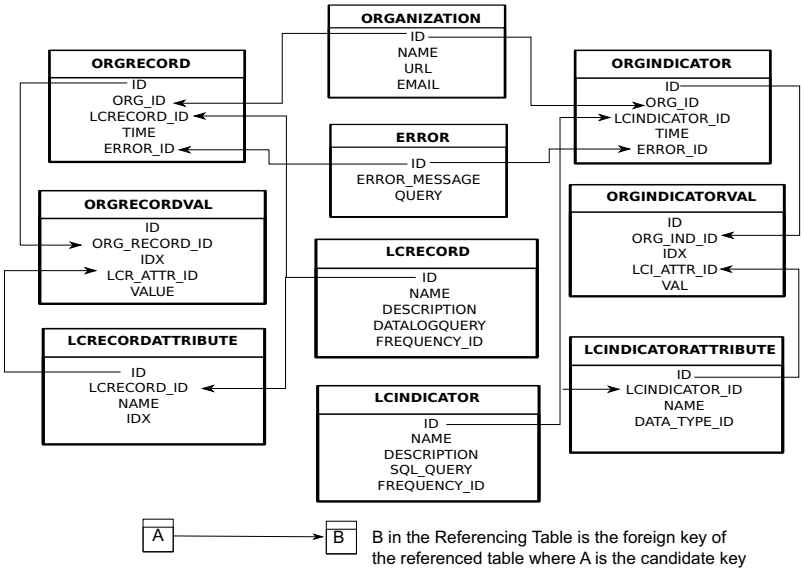

Figure 5.10: Organization Data

captures every information required from the enterprise (or organization), i.e., organization details, authentication parameters for the web services, the interested records and performance indicators. Figure 5.10 shows how enterprise records and performance

indicators are stored. In some cases, record or performance indicator may result in failures. This information is also captured. In addition, there are tables that deal with tags, ratings and calibration status of record definitions and performance indicator definitions. Figure D.1 shows how organization can tag and rate the web services, the records and performance indicators. Figure D.2 shows how the current calibration (section 5.2.2.3) status and performance indicators are handled.

Discussion: In Tables 5.7 and 5.8, the readers may have noticed how we handle the global schema relations, record definitions and performance indicator queries. In a classical data warehouse settings, every data warehouse schema relation is a table in itself. Since we are using the virtual data integration, where the global schema (here data warehouse schema) is not materialized, every global schema relation is a table entry (with details of its attributes stored in another table). *GLOBALSCHEMARELATION* is used to store the relation name and description. *GSRELATIONATTRIBUTE* is used to store the global schema relation attributes. Thus it makes it easier to add new global schema relations when new domains are identified. It also makes the data warehouse partly dynamic. The impact of update and deletion of global schema relations is currently handled with the help of triggers and needs to be further explored.

Record definitions are stored in two tables *LCRECORD* and *LCRECORDATTRIBUTE*. The former is used to store the query predicate name, description and the frequency of execution (like in example 4.2.4, *Daily New Tickets(DNT)* has an associated frequency: Daily to signify that this record definition or datalog query must be evaluated daily). The latter is used to stored the details of the query attributes (like *tkid, src, tkn, tkp, tks* for *DNT*). The organization (or enterprise) records are stored in two tables *ORGRECORD* and *ORGRECORDVAL*, with the former storing the details concerning the record definition (identifier), the timestamp, the organization (identifier) and the error (identifier) occurred during the query evaluation and the latter storing the tuples of query response.

The same two table idea is used to handle the performance indicator, i.e., *LCINDICATOR* and *LCINDICATORATTRIBUTE* to store the details of the performance indicator query name, description, frequency and attributes. *ORGINDICATOR* and *ORGINDICATORVAL* are used to store the enterprise performance indicator values.

Example 5.2.1. *In example 4.2.4, we saw that the record definition* Daily New Tickets *or* DNT *was seen as a relation and used to create performance indicator (or SQL query). But with this table design, the (Oracle) SQL query needs to reflect this idea and the new SQL query takes the following form:*

```
SELECT count (*) FROM OrgRecordVal
WHERE org_record_id IN
(
  SELECT id FROM OrgRecord
  WHERE time < sysdate
    AND
  time > sysdate  - interval '30' day
    AND
  lcrecord_id =
  (
    SELECT ID FROM LCRECORD
    WHERE NAME LIKE 'Daily New Tickets'
  )
    AND
  org_id = $orgID
)
  AND
lcr_attr_id =
(
  SELECT ID FROM LCRECORDATTRIBUTE
  WHERE NAME LIKE 'tkp'
    AND
  LCRECORD_ID =
  (
    SELECT ID FROM LCRECORD
    WHERE NAME LIKE 'Daily New Tickets'
  )
)
  AND
value LIKE '''High'''
```

We now analyse the above query in detail. Note in the above query $orgID *is replaced by the identifier of the respective organization before query evaluation. The following SQL query excerpt ensures that the enterprise records of last 30 days corresponding to the record definition* Daily New Tickets *are taken into consideration:*

```
...
  SELECT id FROM OrgRecord
  WHERE time < sysdate
    AND
  time > sysdate  - interval '30' day
    AND
  lcrecord_id =
  (
    SELECT ID FROM LCRECORD
    WHERE NAME LIKE 'Daily New Tickets'
  )
    AND
  org_id = $orgID
...
```

Following SQL query excerpt ensures that we take into consideration tkp, an attribute of Daily New Tickets *and check whether it's value (in the enterprise record) is*

High*:*

```
...
lcr_attr_id =
(
  SELECT ID FROM LCRECORDATTRIBUTE
  WHERE NAME LIKE 'tkp'
    AND
  LCRECORD_ID =
  (
    SELECT ID FROM LCRECORD
    WHERE NAME LIKE 'Daily New Tickets'
  )
)
  AND
value LIKE '''High'''
```

The following SQL query excerpt makes the count of the tuples of enterprise records satisfying the above conditions:

```
SELECT count (*) FROM OrgRecordVal
WHERE org_record_id IN
......
```

Thus we make use of record definitions to formulate performance indicator definitions. ■

5.2.2.2 Scheduler

Both the record definitions (query formulated over the global schema) and the performance indicator queries have an associated frequency of evaluation that determines the periodicity of computation of the corresponding queries. Frequency values include daily, fortnightly, monthly, quarterly, yearly. In example 4.2.4, *DNT* has the frequency value: Daily and the performance indicator query *Total High Priority Tickets Registered in a month* has the frequency value: Monthly. Scheduler does the following:

1. Get all the organizations currently using DaWeS.
2. For every organization, perform the following steps:
   (a) First compute or form the enterprise records. Find the relevant (or interesting) record definitions for the organization. For every record definition,
      i. Get the corresponding frequency of evaluation.

ii. If the desired record is already computed for the specified period, skip and move on to the next enterprise record definition.
iii. Invoke the record computation module for evaluating the query.

(b) Now compute the enterprise performance measures. Find the relevant (or interesting) performance measure queries. For every performance measure query:
i. Get the corresponding frequency of evaluation.
ii. If the desired performance measure is already computed for the specified period, skip and move on to the next enterprise performance measure query and
iii. Invoke the performance indicator computation module for evaluating the query.

3. Sleep and wait till the next invocation.

5.2.2.3 Calibration

There are primarily two major computations: computation of records (after fetching data from the web services) and computation of performance indicators from records. While fetching data from web services, there are various internal and external factors that can lead to the corruption of data. So measures must be taken to ensure that there is no data corruption and every record computation or performance indicator is computed as expressed by its definition. Record computation involves fetching the data from the web services, validating the web service response, transforming the validated response and finally compute the value of the record. We split this whole process into three levels of calibration steps:

1. **Level 1** Ensuring valid response from Web services: For every web service API operation, there is an associated (XSD) schema that ensures whether all the required information is present in the web service API response and there is no change in their data types. (example: refer section 5.2.2.7).
2. **Level 2** Ensuring a record computes what it is defined to do: For every record definition, there is an associated calibration test data that has both the input and the desired result. Both the input data and desired output data are stored in the DaWeS database. Every record definition can have one or more calibration test data. A record passes the calibration test if the computation of the record produces the expected response. This is to capture any undesired (accidental) changes to the record definitions.

3. **Level 3** Ensuring integrating Web service and record computation also performs the desired computation: This is another level of test where calibration is performed on every record after integrating with the web services. For this level of test, every calibration test data has a desired result and there is no input specified. Unlike the previous level, where both the input and desired output data are stored in the database, in this level the input data is not stored in the database, but rather they come from the web services. This is an end to end calibration of the overall *Records Tier*.

All the above three levels are used to perform the calibration of records

We take a look at *Level 2* calibration with an example. It means we have both the input test data and the expected query response. We feed the test data and the record definitions to the answer builder. The response so obtained is compared with the expected response. If it passes, we say that one calibration test passed. If all the calibration test for a record definition passes, the (most recent) status of the calibration is considered as *passed*, else *failed*.

Example 5.2.2. *This is the Input of a Calibration test Data for the Record Daily New Projects defined by the following datalog query*

```
q(pid,src,pname,pstatus):-Project(pid,src,pname,'yesterday()', pstatus).
```

The input for the calibration is the following

```
Project('1','a','project 1','yesterday()','Open').
Project('2','a','project 2','yesterday()','Open').
Project('2','a','project 2','yesterday(3)','Open').
```

yesterday() *is transformed before query evaluation to yesterday's date and yesterday*(3) *is transformed to date three days before yesterday. And the desired output is the following*

```
q('1','a','project 1','Open').
q('2','a','project 2','Open').
```

■

We store both the input and output calibration test data to the DaWeS database (level 2 as discussed above). When there is no input specified for the record definition calibration (level 3), it corresponds to the situation where web service API operation calls have to be made to obtain the result. The result thus obtained is checked with the desired output.

Calibration of Indicators is performed in the similar manner. A performance indicator is defined using the records. Like the records, for every performance indicator, there are a set of calibration test data with input and desired result specified. The calibration check computes the performance indicator using the specified input data and compares it with the desired result. When all the calibration test data result matches, a performance indicator can be said to pass the calibration test. Calibration of performance indicator may seem to be not as important as calibration of records due to the lack of external factors (web services) but they are also important to catch any unexpected changes to the performance indicator queries due to human errors.

Example 5.2.3. *This is the calibration test data for Total Monthly New Projects The input for the calibration is the following*

```
DailyNewProjects('yesterday()','1','a','project 1','Open').
DailyNewProjects('yesterday()','2','a','project 2','Open').
DailyNewProjects('yesterday(3)','3','a','project 3','Open').
```

And the desired output is the following

```
q('3').
```

Note that the first term in the input test data for the calibration test data corresponds to the time (to imitate the time when a record was saved in the database). Also note that the name used for the relation name is the record name with all spaces removed ■

Calibration test data are stored in the database and they remain unchanged until when some API operation change and it results in calibration test failure(s). The administrator will then change web service description in DaWeS and also the calibration test data to reflect the API changes. Thus calibration also plays an important role to detect the evolution of web services.

5.2.2.4 Rewriting Generator

Rewriting generator performs the query rewriting, that is translating the query formulated over the global schema relations to a query formulated over the local schema relations (or web service api operations). The rewriting generator gives the following output (datalog query plan) corresponding to the given record identifier.

1. Rectified Query
2. Inverse Rules (for the LAV Mapping)
3. Domain Rules
4. Generalized Chase rules (for full and functional dependencies defined over the global schema relation)
5. Transitive rule

The above steps are done in accordance with the Inverse rules algorithm. Also see Figure 5.11 for the role of rewriting generator in query evaluation. Refer example 4.2.3 for the query rewriting generated by inverse rules algorithm.

5.2.2.5 Answer Builder

Answer builder plays a significant role in the evaluation of the record definition. The inputs to the answer builder consists of the record definition identifier and the organization identifier. The main constituent of the answer builder is the datalog engine which evaluates the (recursive) datalog query using the query rewritings. The answer builder performs the following tasks

1. It reads the concerned record definition from the database.
2. Any translation of in-built functions used by DaWeS for the record definitions is performed (Example: yesterday(), see section C.2).
3. It requests for the query rewritings from the rewriting generator (section 5.2.2.4).
4. It loads the query rewritings and the rectified query (record definition) into the datalog engine.
5. Datalog engine during query evaluation on encountering a local schema, request the generic http web service wrapper (section 5.2.2.6) to make the web service API operation call. providing it the values for input parameters. Datalog engine during query evaluation makes available the values for these input parameters.

6. If no errors occurred during the web service API operation calls and during the query evaluation, the datalog engine computes the answer and returns the query result.
7. Store the query response (or error result) to the database.

Figure 5.11 illustrates the role of answer builder in the query evaluation.

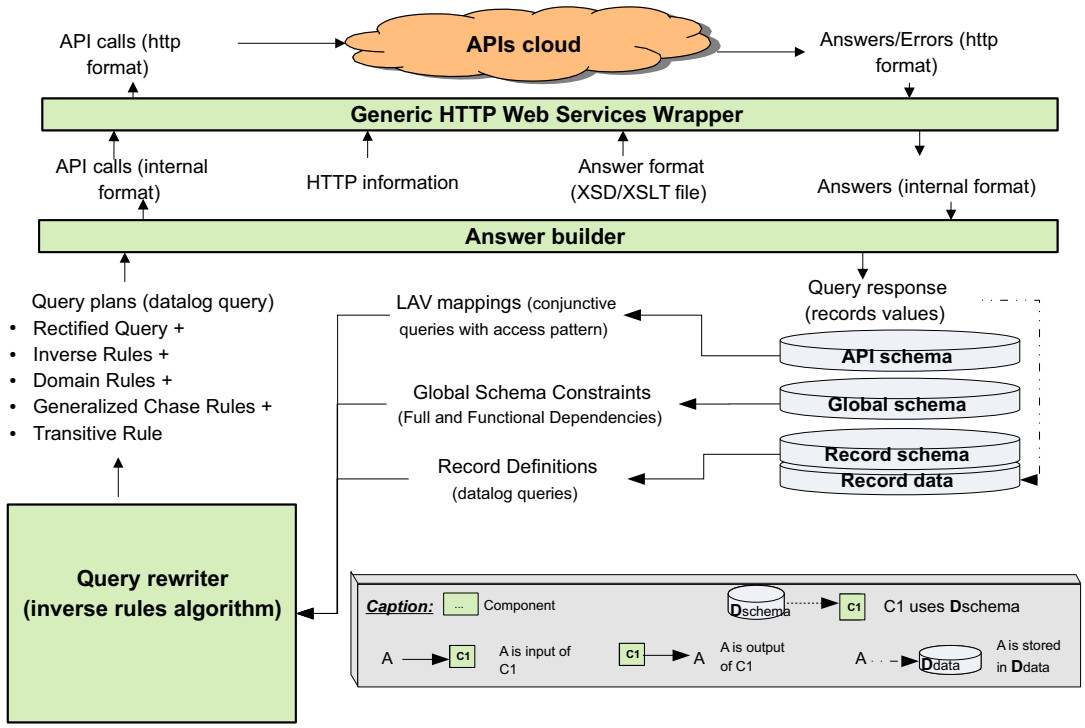

Figure 5.11: DaWeS: Query Evaluation Engine

5.2.2.6 Generic HTTP Web Service API Handler

One of the main components in generic wrapper is Generic HTTP web service API handler that is described in Figure 5.12. The purpose of the generic http web service API handler is to be able to make API calls to any web service. The generic API handler takes as input the organization identifier, the web service operation identifier (name) and the required input parameters for making the web service operation call. The API handler performs the following steps

1. It forms the HTTP URL, HTTP header and the HTTP body using the authentication parameters and the given input parameters. (Example: $https://basecamp.com/\$orgID/api/v1/projects.json$ is transformed to $https://basecamp.com/12345/api/v1/projects.json$ replacing $orgID$ by the organization identifier in Basecamp service)
2. It makes the web service API operation call and waits for the response.
3. When it receives an error/response from the web service, it modifies it to a desired intermediate format (Done for web service operation responses in JSON; such messages are transformed to XML since XML has been well researched for years and several packages associated to schema validation (XSD) and transformation (XSLT) are present).

During any of the above steps, if the API handler encounters an error, it sends the corresponding information to the failure handling module (section 5.2.2.10).

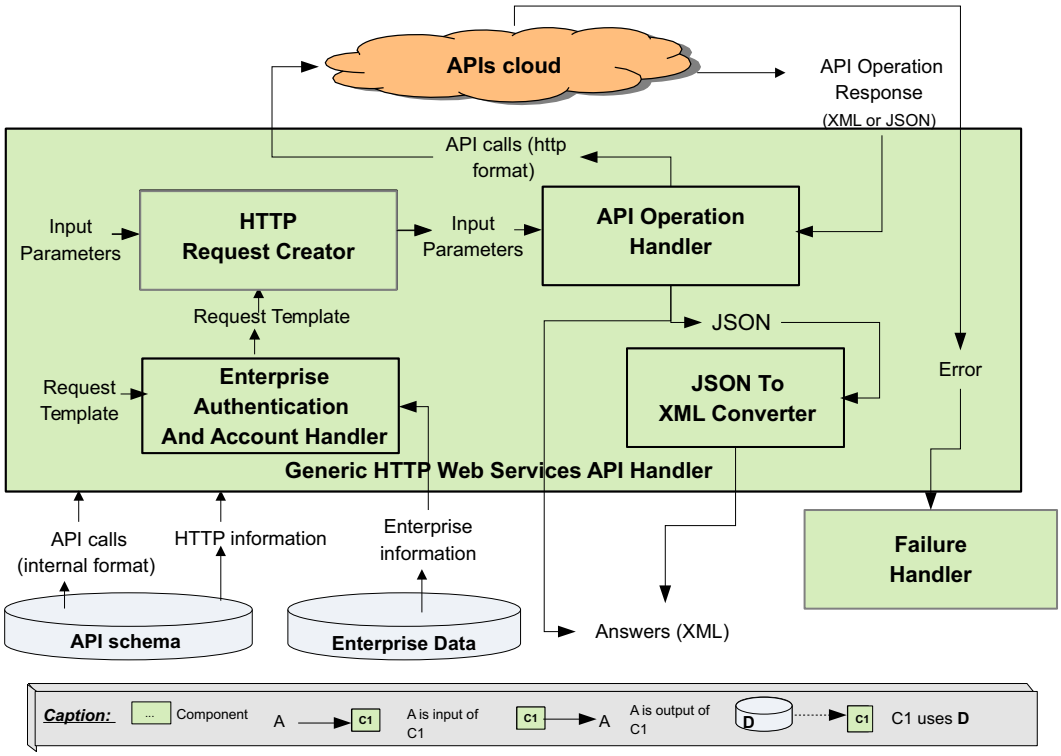

Figure 5.12: DaWeS: Generic HTTP Web Service API Handler

5.2.2.7 Response Validation

The response obtained after making a web service operation call has then to be validated. For this purpose, XSD is used. While defining every API operation, it is also required to define the (expected) schema of the operation response. In this step, it is made sure that the obtained response is in accordance to the expected response schema. If the response is not valid (i.e., the obtained response schema doesn't match with the expected response schema), this module sends the corresponding information to the failure handling module (section 5.2.2.10). In DaWeS, when the response schema is not given in the web service API documentation (in our experience, mostly an example web service API response is given), we make use of the given examples to create the expected schema manually or using online tools like [XSD Generator: FreeFormatter, 2012].

5.2.2.8 Valid Response Transformation

If the response obtained from the web service API operation call is valid, we perform the transformation of the response to a desired format understood by the datalog engine. For this purpose, we use XSLT files created manually by the DaWeS administrators. The output of the this module is a list of tuples having comma separated values. On encountering a failure, failure handling module (section 5.2.2.10) is invoked.

Example 5.2.4. *In this example, we see how the operation response for UTT considered in the example 4.2.4 is validated using XSD and the total number of tickets so obtained from the operation response is transformed to (page number, page size) combination.*

***XSD**: We are only interested to validate whether there exists a field total_records whose XPath is given below:*

1. *Total Tickets (response/response_data/total_records)*

The XSD for the operation UTT response is given below:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="response">
    <xs:complexType>
      <xs:all>
        <xs:element name="response_data">
```

```
          <xs:complexType>
            <xs:all>
              <xs:element name="total_records">
                <xs:complexType>
                  <xs:simpleContent>
                    <xs:extension base="xs:string">
                      <xs:attribute name="type" type="xs:string">
                      </xs:attribute>
                    </xs:extension>
                  </xs:simpleContent>
                </xs:complexType>
              </xs:element>
              <xs:any maxOccurs="unbounded" processContents="lax"/>
            </xs:all>
          </xs:complexType>
        </xs:element>
        <xs:any maxOccurs="unbounded" processContents="lax"/>
      </xs:all>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Note how we aren't interested in any other fields in the operation response. This XSD is used by Response Validation (section 5.2.2.7) to validate the response obtained after making the call to UTT. Next we see the transformation using XSLT.

***XSLT**: The obtained response from UTT contains the total number of tickets. We are interested in extracting the page number, entries per page from the operation response. The default number of entries per page is set as 25. We transform the total number of tickets to tuples of (page number, 25) as shown below. The idea behind the following program is to start from the total number of tickets, produce a page number starting from 1, decrease page size from the total number of tickets and repeat the whole process until the total number of tickets is less than or equal to zero. (For e.g., 100 will give values (1,25),(2,25),(3,25),(4,25)).*

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:fn="http://www.w3.org/2005/xpath-functions" version="2.0">
  <xsl:output method="text" omit-xml-declaration="yes"
    cdata-section-elements="namelist"/>
  <xsl:template name="for-loop">
    <xsl:param name="total" select="1"/>
    <xsl:param name="increment" select="1"/>
    <xsl:param name="page" select="1"/>
    <xsl:if test="$total &gt; 0">
      <xsl:value-of select="$page"/>,
      <xsl:value-of select="$increment"/>
      <xsl:text>&#xa;</xsl:text>
      <xsl:call-template name="for-loop">
```

```
        <xsl:with-param name="total" select="$total - $increment"/>
        <xsl:with-param name="increment" select="$increment"/>
        <xsl:with-param name="page" select="$page+1"/>
      </xsl:call-template>
    </xsl:if>
  </xsl:template>
  <xsl:template match="/">
    <xsl:variable name="default">25</xsl:variable>
    <xsl:variable name="page">
      <xsl:copy-of select="$default"/>
    </xsl:variable>
    <xsl:variable name="total">
      <xsl:value-of select="response/response_data/total_records"/>
    </xsl:variable>
    <xsl:call-template name="for-loop">
      <xsl:with-param name="total" select="$total"/>
      <xsl:with-param name="increment" select="$default"/>
      <xsl:with-param name="page" select="1"/>
    </xsl:call-template>
  </xsl:template>
</xsl:stylesheet>
```

This XSLT is used by Valid Response Transformation (section 5.2.2.7) to transform the response obtained after making the call to UTT to tuples of (page number, page size). ■

5.2.2.9 Response Cache

Response cache is used by the generic wrapper to store the transformed responses for later use. The response cache is checked before making any new web service API operation calls. The cache is a key-value storage mechanism, where for every key, there is an associated value. In our case, the operation name, operation parameters and the organization identifier together forms the key. The transformed operation response is the value. Every key-value constitutes an entry. Therefore the main operations of the cache is to create new entries, update the existing entries and get the value of the existing entries. On obtaining the key, the corresponding value is given. Whenever an operation call is made, the response cache is checked whether any transformed response is cached for that operation and organization along with the given set of input parameters.

5.2.2.10 Web Service API Operation Failure Handling

Following are the possible failures encountered during the web service API operation calls and the associated processes: web service internal error, operation request time-

out, account revoked permission, API deprecation, service enforced policies (breach of quality contract), API operation response Changes, incorrect passage of operation request(input) parameters, permission denied, internal web service temporarily unavailable, change in message formats (XML, JSON, plain-text, signed content) other than previously supported ones., operation deprecation etc. Any error above leads to the abortion of the current evaluation of the query (record definition) and error status is recorded for the corresponding record of the organization.

5.2.2.11 Enterprise Business Performance Indicators Computation

Performance measure computation works in the similar manner as records are computed. Following are the steps involved in the performance measure computation. It takes as input the performance measure query identifier and the organization identifier.

1. Verify whether both the performance measure query identifier and the organization identifier are valid.
2. Check the latest calibration (section 5.2.2.3) status of the performance measure query.
3. If the performance measure query is well calibrated (section 5.2.2.3),
   (a) Get the corresponding frequency of evaluation.
   (b) If the desired performance indicator is already computed for the specified period, return this result.
   (c) For every enterprise record required to compute this business performance.
      i. Read the corresponding enterprise records for the given period from the database.
      ii. Make sure that the enterprise records for the given period are available.
      iii. Make sure that none of the enterprise records for the given period have errors (web service API error response or failure to compute the enterprise record). Recall that every enterprise record in *ORGRECORD* has an attribute called *ERROR_ID* to signify this.
   (d) Compute the performance measure and store the result (error) in the DaWeS database. This computation of SQL query is handled by the underlying DBMS.
   (e) Save the query response (or error status) to the database.

5.2.2.12 Search

Every web service has an associated web service category. Users(organizations) can also rate a web service or tag the service based on their usage pattern. Rating values range from 1 to 10 (10 being the best rating). Tags are (popular) labels that users use to label the web services. Take for example, a DaWeS administrator tagged a project management service as *Project Management*. But regular users of such services may often refer such services as *PM*. Such popular terms are useful for search since a user search like *PM* in DaWeS will return all project management services taking into account the user tag *PM* for one such service. Rating values can be used in conjunction to return the most popular web service(s) at the top.

Similarly a performance indicator and a record can have associated tags and rating defined by the users(organizations). When an application user searches for a web service, record or performance indicator, (s)he must be shown other relevant information that can enable him/her to see other options available in the DaWeS (such results are popularly known as search suggestions). Search works in the following manner. It takes two options: type and the pattern to search. Type can be web service, performance indicator or record. The pattern to search can be a string of characters or use special characters like regular expression pattern. We explain the search procedure with the type web service below. It can be extended to record and performance indicator search with the exception that records and performance indicators have no associated category. Following are the steps:

1. First look for the web services with the exact name or has the given pattern in its name.
   (a) Get the rating for each web service and arrange the web service in the descending order of rating with top rated web service at the top.
   (b) Get the web services having the same tag and category as the web service(s) found in 1.a.
2. Search for web service category that match the user pattern.
   (a) Get the rating for each web service and arrange the web service in the descending order of rating with top rated web service at the top.
   (b) For each matching category, get all the web services
3. Search for tags that match the user pattern. For each matching tag, get all the web services that have the same tag.
4. For every web service obtained in 1.b, 2.b and 3.b, find the category and tag.

(a) Get the web services belonging to this category and tag.
(b) Get the rating of all the web services and their rating and arrange them in the descending order.

5.2.3 Development

Table 5.1: Development Environment

System details	
Processor	Intel(R) Pentium(R) Dual CPU @ 2.16GHz
System Memory	3GiB
Operating System	Ubuntu [Ubuntu, 2012] 13.04 (32 bits)
Database details	
Database	Oracle 11g [Oracle Database, 2012] (11.2.0.1.0)
SQL Interface	Oracle SQL Developer [SQL Developer, 2012] v3.1.07
Development details	
IDE	Eclipse IDE [Eclipse, 2012] Version 3.8
Programming Language	Java [Java SE, 2012] (Version: Java 1.7.0_25)
XSD and XSLT	
XSD	1.1 [Gao et al., 2009]
XSLT	2.0 [Kay et al., 2007]
Libraries Used and their Versions	
IRIS [IRIS, 2008]	v0.60
Ehcache [ehcache, 2012]	2.6.5
Hibernate [hibernate, 2012]	4.1.9
Scribe [Fernandez, 2013]	1.3.5

Table 5.1 summarizes the development environment. The system was developed and tested in Ubuntu 13.04 (32 bits) operating system using Java 1.7.0_25 and Oracle 11g database. We chose IRIS (Integrated Rule Inference System) [IRIS, 2008] as the datalog engine considering its capability to handle adornments (to specify access patterns in the relations) and functional terms (generated by the inverse rules algorithm for the LAV mappings). IRIS also has various built-in predicates like EQUAL to specify the equality predicates. It can also be configured to refer external data sources during query evaluation. It also supports a lot of optimization techniques that can be used to

optimize the datalog query evaluation. Following are the IRIS configurations used by us:

1. We enabled standard rule safety in IRIS so as to receive an exception whenever an unsafe conjunctive query is encountered.
2. For the purpose of optimizations, we enabled rule filters along with magic sets [Abiteboul et al., 1995]. This ensures that only those rules that are useful for the query evaluation are taken into consideration and the remaining are removed.
3. In addition to the support for pre-loading of facts (tuples of constants) to IRIS, it also supports configuring external sources so that the facts can be loaded during the query evaluation. In our case, we don't have any preloaded facts from the web services and the generic wrapper (section 5.2.2.6) is the only external source that can give access to the enterprise data over the web services. Hence we configured IRIS to make use of the generic wrapper during query evaluation.
4. For the evaluation of the datalog queries, we configured IRIS to use bottom-up evaluation strategy along with semi-naive evaluation. Bottom up evaluation (along with magic sets) performs better than top down evaluation [IRIS, 2008; Ullman, 1989a] and so is the semi-naive evaluation compared to the naive approach.

During development, we used Hibernate (an object-relational mapping library for Java) to easily work with the various SQL tables used in DaWeS. We used the scribe library for the web services using OAuth 1.0 to access the enterprise data. Ehcache is used to cache transformed API operation responses. We use [Json-lib, 2012] Java library to convert JSON responses to XML messages. For the XML serialization, validation and transformation use use the Javax.xml library [javax.xml, 2012].

Chapter 6

Optimizing Query Processing in DaWeS

In this chapter, we present two open problems that have arisen while implementing DaWeS. The first problem is related to the management of incomplete information, which means unknown information in the context of DaWeS. The second is related to the bounding of the number of API operation calls, which can be expensive both in time and money. In section 4.1, we recall theoretical elements needed to understand our proposals to these problems. These are then studied in section 6.1 and 6.2.

In [Rajaraman et al., 1995], semantics of executable conjunctive queries has been defined intuitively in the following manner: make accesses to every relation in $body(\psi)$ from left to right. While accessing a relation with input attributes, make accesses with the values of output attributes obtained by making accesses to the previous relations in $body(\psi)$. Precisely defining the semantics of $\mathcal{CQ}^{\alpha}$ is the aim of section 6.2.2.

6.1 Handling Incomplete Information

As seen in chapter 5, using mediation with LAV mappings is convenient since it allows to easily add, remove and update API operations. However, the semantics it relies on, the certain answer semantics, is quite constraining. One of the implied constraints is that incomplete tuples (that is tuples in which some data would be missing) are not considered as answers. This means that potentially many data cannot be exploited because the lack of a few others. We illustrate this point in the following example

(where access patterns have not been taken into account, for a sake of clarity).

Example 6.1.1. *Consider the case given in the Figure 4.1. We recall the following elements:*

- *the global schema is* $\{cite^{(2)}, sameTopic^{(2)}\}$
- *the LAV mappings are (v3 is not used here):*

  $v1(A, B) \leftarrow cite(A, B), cite(B, A)$
  $v2(E, F) \leftarrow cite(E, F)$
  $v4(C, D) \leftarrow sameTopic(C, D)$

- *the query is* $ans(A, B) \leftarrow cite(A, B), cite(B, A), sameTopic(A, B)$

Suppose that the extensions of $v1$, $v2$ *and* $v4$ *are the following (given in a tabular style):*

$v1$

2	4
3	6

$v2$

3	4
1	3
1	2

$v4$

1	2
3	4
2	4

Since the database instances of the global schema must be compatible with the source extensions, the they must contain at least the following tuples:

$cite$

3	4
1	3
1	2
2	4
4	2
3	6
6	3

$sameTopic$

1	2
3	4
2	4

It means that each database instance for the global schema contains at least these tuples, but may contain other tuples. The rewriting given by the inverse-rules algorithm is the following Datalog query (the first rule is the original query, and the other ones are the inverse-rules):
$ans(A, B) \leftarrow cite(A, B), cite(B, A), sameTopic(A, B)$
$cite(A, B) \leftarrow v1(A, B)$
$cite(A, B) \leftarrow v1(B, A)$
$cite(E, F) \leftarrow v2(E, F)$
$sameTopic(C, D) \leftarrow v4(C, D)$
In this configuration, evaluating this rewriting does not imply any problem. The obtain answers, which are the certain answers, are:

ans		
	2	*4*

We recall the certain answers are the intersection of all sets of answers obtained by evaluating the query on all database instances of the global schema that contain the tuples of the source relations.

Let's now suppose that we change the LAV mappings (keeping the same global schema and query):

- *the global schema is* $\{cite^{(2)}, sameTopic^{(2)}\}$
- *the LAV mappings is:*

  $v5(A, B) \leftarrow cite(A, C), cite(C, B), sameTopic(B, C)$

  with the following extension

$v5$		
	1	*4*
	3	*3*

- *the query is* $ans(A, B) \leftarrow cite(A, B), cite(B, A), sameTopic(A, B)$

Then the instances of the global schema must contain at least the following tuples:

cite	
1	*f(1,4)*
f(1,4)	*4*
3	*f(3,3)*
f(3,3)	*3*

sameTopic	
4	*f(1,4)*
3	*f(3,3)*

The functional terms express the existential variable C used in the LAV mapping. For example, since the tuple $\langle 1, 4\rangle$ *is in the extension of* $v5$*, it means that there exists some C which depends functionaly from* $\langle 1, 4\rangle$ *(that's why we note it* $f(1, 4)$*) such that* $\langle 1, f(1, 4)\rangle$ *and* $\langle f(1, 4), 4\rangle$ *are in the extension of cite and* $\langle 4, f(1, 4)\rangle$ *is in the extension of sameTopic. Due to the presence of functional terms, database instances are not database instances any more (since there is no functional terms in a database instance). These are "more general" instances. Since it expresses an existential variable, each functional term may replace many values (at least one). So this is a sort of generic description of many possible databases. [Grahne and Kiricenko, 2004] call it an incomplete database. Strictly speaking the evaluation of the rewriting given by the inverse-rules algorithm, which is the following Datalog query:*

$ans(A, B) \leftarrow cite(A, B), cite(B, A), sameTopic(A, B)$

$cite(A, f(A, B)) \leftarrow v5(A, B)$

$cite(f(A, B), B) \leftarrow v5(A, B)$

$sameTopic(B, f(A, B)) \leftarrow v5(A, B)$
cannot use these functional terms. It means in our case that the set of certain answers is empty. But what a pity not to be able to use the tuples in the sources ! Indeed, it would still be interesting to return to the user the following answers:

ans		
	3	*f(3,3)*

As in [Grahne and Kiricenko, 2004], the solution we adopted in DaWeS is a relaxation of the certain answers semantics to allow dealing with functional terms. It is well-known [Ullman, 1989b] how to evaluate a *Datalog* program with functions. Here the fact that functional terms can only arise in the head of inverse-rules (which are non recursive) ensures that the naive bottom-up evaluation always stops. And then, results from [Grahne and Kiricenko, 2004] ensure that the obtain answers contain both the certain answers and the answers built with functional terms.

Concretely, to handle incomplete values:

- the last step of the inverse-rules algorithm called "predicate splitting", which aims at getting rid of functional terms, is removed,
- the rewriting with functional terms is evaluated by the IRIS reasoner which is able to handle functional terms during datalog evaluation,
- and functional terms in answers are then replaced by marked null [Ullman, 1989b] because marked nulls are null values seen as (special) constants and are used to represent unknown values. We remove functional terms by marked nulls because they can be easily handled with relational databases by treating them as constants.

This is illustrated in figure 6.1

The last step is the adding of a heuristics to take advantage of generalized chase rules on marked nulls. Indeed take the following example:

Example 6.1.2. *Consider three sources BCOtherProject, BCMoreProject and BCSomeProject having the details of projects. The three of them have some common projects. Take for example we consider the following tables*

BCSomeProject			
	1	*Design*	*Open*
	2	*Development*	*Open*

BCOtherProject		
	1	*Design*
	4	*Testing*

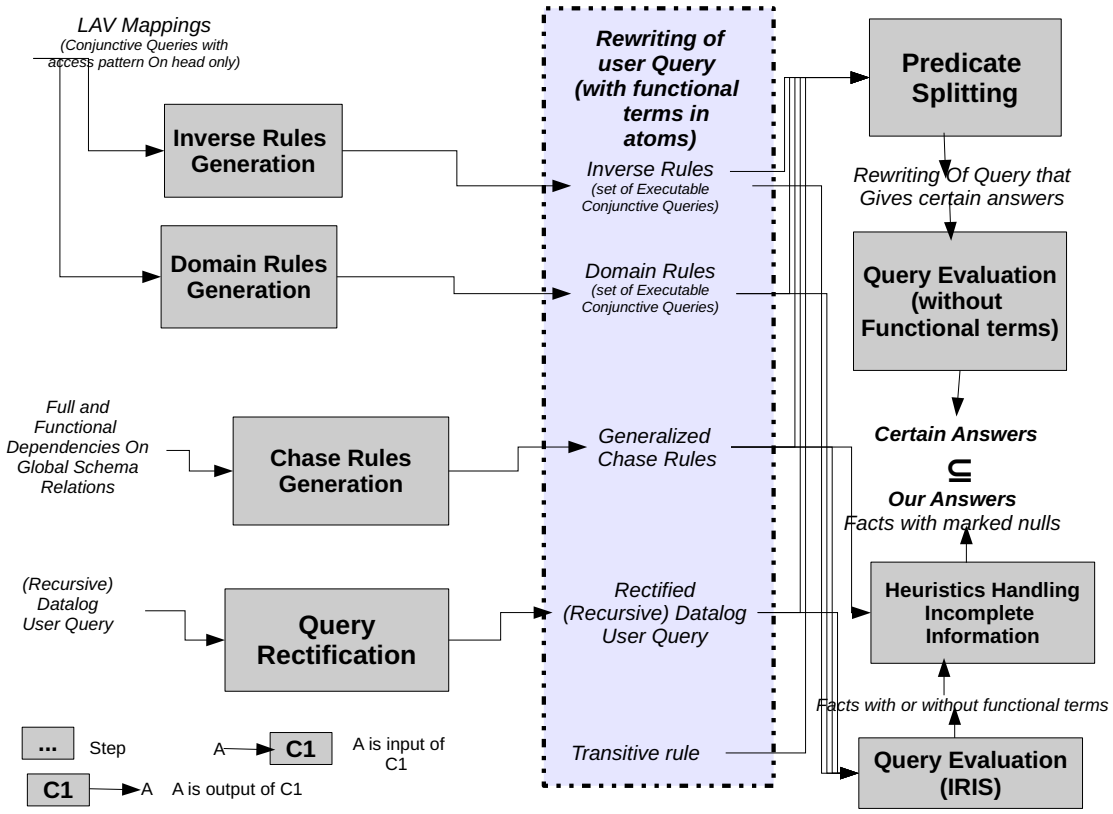

Figure 6.1: The adapted Inverse-Rules Algorithm

$BCMoreProject$		
	1	*Design*

Consider the following rewriting (simplified, without domain rules). Note how we use the EQUAL builtin predicate of IRIS to specify the functional dependencies and query rectification in the body and use 'e' in the head. This is because adding IRIS builtin function EQUAL in the head will lead to infinite query evaluation (i.e., f(f(f(f...))) terms will be generated because of generalized chase rules). Rather we are interested in extension of 'e' by making use of the modified generalized chase rules to use them later for removing functional terms.

$Project(p,'BC', n, f_{BCOP,4}(p,n)) \leftarrow BCOtherProject^{oo}(p,n).$
$Project(p,'BC', n, f_{BCMP,4}(p,n)) \leftarrow BCMoreProject^{oo}(p,n).$
$Project(p,'BC', n, ps) \leftarrow BCSomeProject^{ooo}(p,n,ps).$

$e(n1, n2) \leftarrow Project(p1, s1, n1, ps1), Project(p2, s2, n2, ps2), EQUAL(p1, p2),$
$EQUAL(s1, s2)$
$e(ps1, ps2) \leftarrow Project(p1, s1, n1, ps1), Project(p2, s2, n2, ps2), EQUAL(p1, p2),$
$EQUAL(s1, s2)$
$q(p1, n1, ps1) \leftarrow Project(p1, s1, n1, ps1), EQUAL(p1, p),$
$EQUAL(n1, n), EQUAL(ps1, ps), EQUAL(s1, s).$

After the query evaluation of the above query rewriting, we get the following

q		
1	*Design*	*Open*
2	*Development*	*Open*
1	*Design*	$f_{BCMP,4}$*(1 , Design)*
1	*Design*	$f_{BCOP,4}$*(1 , Design)*
3	*Documentation*	$f_{BCOP,4}$*(3 , Documentation)*
4	*Testing*	$f_{BCOP,4}$*(4 , Testing))*

Next we make generate the equality predicates (predicate name as 'e') using the generalized chase rules.

e	
Open	*Open*
Open	$f_{BCMP,4}$*(1,Design)*
Open	$f_{BCOP,4}$*(1,Design)*
$f_{BCMP,4}$*(1,Design)*	*Open*
$f_{BCMP,4}$*(1,Design)*	$f_{BCMP,4}$*(1,Design)*
$f_{BCMP,4}$*(1,Design)*	$f_{BCOP,4}$*(1,Design)*
$f_{BCOP,4}$*(1,Design)*	*Open*
$f_{BCOP,4}$*(1,Design)*	$f_{BCMP,4}$*(1,Design)*
$f_{BCOP,4}$*(1,Design)*	$f_{BCOP,4}$*(1,Design)*
$f_{BCOP,4}$*(3,Documentation)*	$f_{BCOP,4}$*(3,Documentation)*
$f_{BCOP,4}$*(4,Testing)*	$f_{BCOP,4}$*(4,Testing)*
Design	*Design*
Development	*Development*
Documentation	*Documentation*
Testing	*Testing*

The e predicates are used to remove the functional terms and marked nulls or constant are generated for every functional term. Take for example $f_{BCOP,4}$*(4,Testing) generates a marked null* $\perp_2$*. When we encounter* $f_{BCOP,4}$*(1,Design) or* $f_{BCMP,4}$*(1,Design),*

we generate 'Open'.

This is the final output

q			
	1	*Design*	*Open*
	2	*Development*	*Open*
	3	*Documentation*	$\perp_1$
	4	*Testing*	$\perp_2$

■

Our heuristics that simplifies the result as in the previous example is given in figure 6.2:

We have four major components

1. *e Predicate Generation*: This component generates the facts with predicate name 'e' using the generalized chase rules defined in the manner in example 6.1.2.
2. *Marked Null Generation*: This component consists of a function that returns a new marked null for every functional term
3. *Removal Of Functional Terms*: This component takes facts with functional terms and facts with predicate names as 'e' to generate facts with marked nulls or facts without functional terms.
4. *Facts Generation* ensures that no facts are repeated. It takes both the facts without functional terms and facts with marked null and generates set of facts (with marked nulls or without functional terms). This step is important since replacing of functional terms with marked nulls or constants may lead to a fact already generated.

6.2 Bounding the Number of Accesses

6.2.1 Motivation

Apart from dealing with incomplete information, a second problem arises in DaWeS due to the online context of our mediation. Since each API operation call can be expensive, reducing the number of these is important. When generating the domain rules (the ones that will imply API operation calls), the inverse rules algorithm does not try to minimize the number of implied calls. For example, functional dependencies

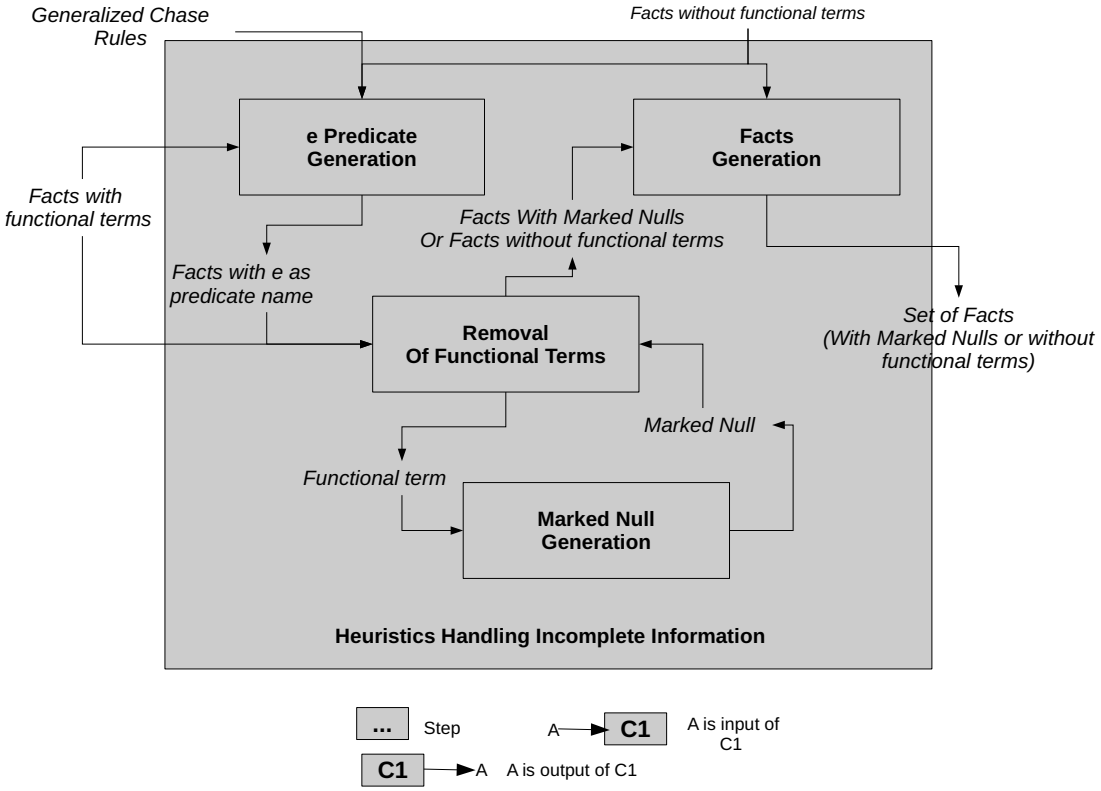

Figure 6.2: Heuristics to handle Incomplete Information

existing among attributes are not taken into account while some useful information can be exploited from them to optimize domain rules so that the number of implied calls is reduced.

Example 6.2.1. *Consider a relation* $YearlyPublication(Year, Title)$ *with attributes* $Year$ *and* $Title$ *which contains couples* (Y, T) *such that the paper having title* T *has been published in year* Y*. A query like*

$$ans(Y, T) \leftarrow YearlyPublication(Y, T)$$

can be used to get all the publications. A single access thus gives all the tuples of the relation. This access is the empty tuple since there is no input attribute.

But if the relation $YearlyPublication$ *has access limitation like* Year *as an input attribute, noted* $YearlyPublication^{io}(Year, Title)$*, it signifies that the user must specify*

a year to get the publications of the corresponding year. Thus, to obtain all publication titles, we have to make use of the following query:

$$ans(T) \leftarrow Y =?, YearlyPublication(Y, T)$$

meaning that values must be given to Y to obtain values for T. An access is one such value. One possible way is to specify a range of years, e.g. 1900-2014, which implies 115 different accesses. Restricting to a particular range may not yield all the results, if for example the publisher might be publishing long before 1900. Another possible problem is that if the publisher might have only started publishing since 1980, it results in 80 spurious accesses (i.e., values of years between 1900-1979). Such spurious accesses are wasteful, especially in our context where an access corresponds to a remote API operation call that is priced by the SLA of the service provider. Such queries could be avoided if for example, the publisher also exposes another relation $Volumes^{oo}(Year, Volno)$ with attributes year *and* volno *without any access limitations. volno corresponds to the volume number. Thus we can obtain all the publications with the query:*

$$ans(T) \leftarrow Volumes(Y, V), YearlyPublication(Y, T)$$

Here the domain of values for the input attribute $Year$ of $Publication$ is given by the first atom of the query. It is easy to see that the number of accesses to $YearlyPublication$ is the same as the number of different values for the attribute $Year$ in $Volumes$ since it is the only input attribute. So the number of accesses implied in ths query is bounded by $|Volumes| + 1$, i.e. the total number of tuples in $Volumes$ plus one access to query $Volumes$. Here $Volumes$ is said to be the domain of the input Y. ■

For relations with more than one input, the basic approach is to find the domain of all the input attributes and take the cartesian product of them, in order to create all possible input tuples.

Example 6.2.2. *Let's consider the relation $Publication^{iio}(Year, Volno, Title)$, with three attributes $Year$, $Volno$ and $Title$. The two input attributes ($Year$ and $Volno$) are to be specified to obtain all the titles published in a volume of a publication. Therefore we have two domains year and volume number, which can be obtained from the relation $Volumes^{oo}(Year, Volno)$. Let's consider the following table of values shown in Table 6.1. Note however that the values of Publication without specifying the corresponding input values cannot be obtained. The query:*

$$ans(Y) \leftarrow Volumes(Y, V)$$

gives all the publication years (2010, 2011 and 2012). *The query:*

$$ans(V) \leftarrow Volumes(Y, V)$$

gives all the volume numbers (v1,v2,v3,v4,v5,v6). *Thus we have three years and six*

Table 6.1: Tables: Volumes and Publication

Volumes	
Year	Volno
2010	v1
2010	v2
2011	v3
2011	v4
2012	v5
2012	v6

Publication		
Year	Volno	Title
2010	v1	a
2010	v2	b
2011	v3	c
2011	v4	d
2012	v5	e
2012	v6	f
2012	v7	g

volume numbers, which are used to build the following set of accesses to Publication: {(2010,v1), (2010,v2), (2010,v3), (2010,v4), (2010,v5), (2010,v6), ... (2012,v6)}. *The query to obtain all the publication details is*

$$ans(Y, V, T) \leftarrow Volumes(Y, V1), Volumes(Y1, V), Publication(Y, V, T)$$

The number of accesses for the above query is bounded by $36 + 2$ *(i.e.* $|Volumes| \times |Volumes|+2$*).If the query is executed, the real number of accesses is* $18+2$ *(since same accesses are not used many times). However, here, it is easy to see that the following query:*

$$ans(Y, V, T) \leftarrow Volumes(Y, V), Publication(Y, V, T)$$

returns the same answers for only 6 *accesses.* ■

As shown in the previous example, a simple optimization on the query itself can drastically reduce the value of both the upper bound on the number of accesses and the number of really executed accesses. Our aim in this section is not to study this kind of query optimizations, but rather to define a theoretical upper bound on the number of accesses that will be used to compare these optimizations. The kind of queries

that are concerned by this bound are both the queries that are outputs of the inverse-rules algorithm (thereafter called $Datalog^{\alpha_{Last}}$ queries), and also executable conjunctive queries (called $\mathcal{CQ}^{\alpha}$ queries). Studying $\mathcal{CQ}^{\alpha}$ queries, and not only $Datalog^{\alpha_{Last}}$ queries, is a way to generalize a bit this work, knowing that these two languages are really close. Indeed, as shown in the next section, $\mathcal{CQ}^{\alpha}$ semantics is defined in terms of $Datalog^{\alpha_{Last}}$ semantics.

6.2.2 $\mathcal{CQ}^{\alpha}$ Queries Operational Semantics

We now focus on defining a semantics for $\mathcal{CQ}^{\alpha}$ queries. Since our aim is to bound the number of accesses needed when evaluating a $\mathcal{CQ}^{\alpha}$ query, we focus on an operational semantics which gives a model of the query evaluation process, especially by precising when accesses are done (i.e. when operations are called). Our aim here is to formalize what has been previously intuitively enounced (eg. in [Rajaraman et al., 1995]) concerning the semantics of $\mathcal{CQ}^{\alpha}$ queries. Let's explain the idea underlying the operational semantics with an example.

Example 6.2.3. *Suppose we have to evaluate the following $\mathcal{CQ}^{\alpha}$ query:*
$\phi : \mathbf{ans}(X, Y) \leftarrow \mathbf{p_1}^{o}(X), \mathbf{p_2}^{ioo}(X, Y, Z1), \mathbf{p_6}^{io}(Z1, Z3), \mathbf{p_2}^{ioo}(Z3, Z2, X), \mathbf{p_4}^{o}(V),$
$\mathbf{p_5}^{o}(V), \mathbf{p_3}^{iiio}(Y, Z1, V, T).$
This query is executable since, each variable in an input attribute appears in an output attribute of a previous atom, when reading the body from left to right. This means that a first access (the empty tuple) is needed for $\mathbf{p_1}^{o}(X)$ to get values for X. Then each singleton tuple containing a value for X is an access for $\mathbf{p_2}^{ioo}(X, Y, Z1)$ to get couples for $(Y, Z1)$. Then values for $Z1$ are used to build accesses for $\mathbf{p_6}^{io}(Z1, Z3)$ to get values for $Z3$, and so on. The process is recursive since from values for $Z3$ and $\mathbf{p_2}^{ioo}(Z3, Z2, X)$, we can get new values for X, which can lead to new values for Y and $Z1$ via $\mathbf{p_2}^{ioo}(X, Y, Z1)$, and so on. However this recursive process will eventually stops because no new constant is ever generated and because instances of relations are all finite. This query is equivalent to the following Datalog query, having ans as its query predicate:

$\mathbf{dom_X}(X) \leftarrow \mathbf{p_1}^{o}(X).$
$\mathbf{dom_Y}(Y) \leftarrow \mathbf{dom_X}(X), \mathbf{p_2}^{ioo}(X, Y, Z_1).$
$\mathbf{dom_{Z_1}}(Z_1) \leftarrow \mathbf{dom_X}(X), \mathbf{p_2}^{ioo}(X, Y, Z_1).$
$\mathbf{dom_{Z_3}}(Z_3) \leftarrow \mathbf{dom_{Z_1}}(Z_1), \mathbf{p_6}^{io}(Z_1, Z_3).$
$\mathbf{dom_{Z_2}}(Z_2) \leftarrow \mathbf{dom_{Z_3}}(Z_3), \mathbf{p_2}^{ioo}(Z_3, Z_2, X).$
$\mathbf{dom_X}(X) \leftarrow \mathbf{dom_{Z_3}}(Z_3), \mathbf{p_2}^{ioo}(Z_3, Z_2, X).$

$\mathbf{dom_V}(V) \leftarrow \mathbf{p_4}^o(V).$
$\mathbf{dom_V}(V) \leftarrow \mathbf{p_5}^o(V).$
$\mathbf{dom_T}(T) \leftarrow \mathbf{dom_Y}(Y), \mathbf{dom_{Z_1}}(Z_1), \mathbf{dom_V}(V), \mathbf{p_3}^{iiio}(Y, Z_1, V, T).$
$\mathbf{p'_1}(X) \leftarrow p_1^o(X).$
$\mathbf{p'_2}(X, Y, Z_1) \leftarrow \mathbf{dom_X}(X), \mathbf{p_2}^{ioo}(X, Y, Z_1).$
$\mathbf{p'_6}(Z_1, Z_3) \leftarrow \mathbf{dom_{Z_1}}(Z_1), \mathbf{p_6}^{io}(Z_1, Z_3).$
$\mathbf{p'_2}(Z_3, Z_2, X) \leftarrow \mathbf{dom_{Z_3}}(Z_3), \mathbf{p_2}^{ioo}(Z_3, Z_2, X).$
$\mathbf{p'_4}(V) \leftarrow \mathbf{p_4}^o(V).$
$\mathbf{p'_5}(V) \leftarrow \mathbf{p_5}^o(V).$
$\mathbf{p'_3}(Y, Z_1, V, T) \leftarrow \mathbf{dom_Y}(Y), \mathbf{dom_{Z_1}}(Z_1), \mathbf{dom_V}(V), \mathbf{p_3}^{iiio}(Y, Z_1, V, T).$
$\phi' : \mathbf{ans}(X, Y) \leftarrow \mathbf{p'_1}(X), \mathbf{p'_2}(X, Y, Z_1), \mathbf{p'_6}(Z_1, Z_3), \mathbf{p'_2}(Z_3, Z_2, X), \mathbf{p'_4}(V),$
$\mathbf{p'_5}(V), \mathbf{p'_3}(Y, Z_1, V, T).$

In this query, the rules with head predicates $\mathbf{dom}_{...}$ *are called the domain rules: their purpose is to store every possible value obtainable for the associated variable (the one in subscript). From these* $\mathbf{dom}_{...}$ *relations and from the inital relations* $\mathbf{p_i}$*, the set of* $\mathbf{p'_i}$ *relations are obtained by applying the process discussed above. The purpose of these* $\mathbf{p'_i}$ *relations is to be a materialized copy of* $\mathbf{p_i}$ *relations, so that they can be queries without needing any call to any remote operation. Then the initial query can be evaluated from* $\mathbf{p'_i}$ *relations with a classical conjunctive query evaluation process since* $\mathbf{p'_i}$ *relations have no access pattern. This is the purpose of the last rule* ϕ'*. The full set of rules is a Datalog query because of the recursive process discussed above. We remark however that each rule in this Datalog query is simpler than a query in* $\mathcal{CQ}^\alpha$ *in the sense that either they have no access pattern, or they have a single access pattern for their last atom (i.e. the right outermost atom).* ■

The evaluation process examplified above is intuitively based on an operational semantics for $\mathcal{CQ}^\alpha$ queries. Up to our knowledge, this semantics has not been precisely defined in the literature [Duschka et al., 2000; Rajaraman et al., 1995]. This is the purpose of the rest of this section.

We begin by defining the operational semantics of safe conjunctive queries having at most one access pattern on their last (i.e. right outermost) atom. Thus, this set of queries includes safe conjunctive queries without any access pattern. This set is noted $\mathcal{CQ}^{\alpha_{Last}}$. It is clear that $\mathcal{CQ} \subseteq \mathcal{CQ}^{\alpha_{Last}} \subseteq \mathcal{CQ}^\alpha$. Then we define the operational semantics of $Datalog^{\alpha_{Last}}$ queries which are *Datalog* queries which rules are all in $\mathcal{CQ}^{\alpha_{Last}}$. At last we show how $\mathcal{CQ}^\alpha$ semantics is defined using $Datalog^{\alpha_{Last}}$ semantics.

Definition 6.2.4 ($\mathcal{CQ}^{\alpha_{Last}}$ syntax)**.** *A* $\mathcal{CQ}^{\alpha_{Last}}$ *query is a* $\mathcal{CQ}^\alpha$ *query with its last*

atom only having an access pattern. Thus, it is an expression of the following form:
$\phi : ans(\overline{Z}) \leftarrow r_1(\overline{X_1}), ..., r_q(\overline{X_q}), r^{\alpha}_{q+1}(\overline{X}, \overline{Y})$
such that:

- $var(\overline{X}) \subseteq (\bigcup_{i=1}^{q} var(\overline{X_i}))$ *and* $var(\overline{Z}) \subseteq (\bigcup_{i=1}^{q} var(\overline{X_i})) \cup var(\overline{Y})$,
- α *is the access pattern of the last atom with* $\overline{X}$*, the input variables tuple and* $\overline{Y}$ *the output variables tuple*
- *and there is no access pattern for atoms* $r_1, \ldots, r_q$.

In the previous definition, since the last body atom is written $r^{\alpha}_{q+1}(\overline{X}, \overline{Y})$, we may think the input attributes always come before the output attributes. In fact this is not the case: input and output attributes can be found at any position in this atom. This notation is just a convenient shortcut to use the input and output tuples names.

Definition 6.2.5 ($Datalog^{\alpha_{Last}}$ syntax)**.** *A* $Datalog^{\alpha_{Last}}$ *query is a Datalog query which rules are all in* $\mathcal{CQ}^{\alpha_{Last}}$.

In the following, we consider a $Datalog^{\alpha_{Last}}$ query Φ as a set $\{\phi_j\}$ where each ϕ_j is a rule of Φ. For a $Datalog^{\alpha_{Last}}$ query Φ, we note $schema(\Phi)$ the set of all relations used in Φ. We now define the semantics of $\mathcal{CQ}^{\alpha_{Last}}$. The idea is simple: we just extend the classical $\mathcal{CQ}$ semantics by making a special focus on the last atom (the one with an access pattern) so that a naive evaluation procedure can easily be derived.

Definition 6.2.6 ($\mathcal{CQ}^{\alpha_{Last}}$ semantics)**.** *Consider a* $\mathcal{CQ}^{\alpha_{Last}}$ *query* ϕ *using the notations of definition 6.2.4, ie.* $\phi : ans(\overline{Z}) \leftarrow r_1(\overline{X_1}), ..., r_q(\overline{X_q}), r^{\alpha}_{q+1}(\overline{X}, \overline{Y})$*. Let* $\mathcal{D}$ *be a database instance such that* $schema(\mathcal{D}) = \{r_1, \ldots, r_q, r^{\alpha}_{q+1}\}$*. Let* $r_i(\mathcal{D})$ *be the extension of* r_i *in* $\mathcal{D}$*, for each* $i \in \{1, \ldots, q\}$ *and* $r^{\alpha}_{q+1}(\mathcal{D})$ *be the extension of* r^{α}_{q+1} *in* $\mathcal{D}$*. The operational semantics of* ϕ *is defined as follows:*

$$\phi(\mathcal{D}) := \{ans(v(\overline{Z}))|v \text{ valuation on } \left(\bigcup_{i=1}^{q} var(\overline{X_i})\right) \cup var(\overline{Y}) \text{ such that } \left(\forall i \in \{1, \ldots, q\}\ r_i(v(\overline{X_i})) \in r_i(\mathcal{D})\right) \text{ and } \left(r^{\alpha}_{q+1}(v(\overline{X}), v(\overline{Y})) \in r^{\alpha}_{q+1}(\mathcal{D})\right)\}$$

This semantics can be considered as operational since we can derive a basic way to evaluate such queries: first valuations are found for the same query without its last atom (the one with an access pattern), then these valuations are tested on the inputs of the last atom. Each valuation is extended by each output tuple it generates from the last atom. At last each extended valuation gives one answer of the initial query. The application of the valuations computed during the first step on input attributes of the last atom defines the so-called set of accesses implied in the evaluation of a $\mathcal{CQ}^{\alpha_{Last}}$ query ϕ over a database instance $\mathcal{D}$.

Definition 6.2.7 (Set of accesses)**.** *Consider a $\mathcal{CQ}^{\alpha_{Last}}$ query ϕ using the notations of definition 6.2.4, ie. $\phi : ans(\overline{Z}) \leftarrow r_1(\overline{X_1}), ..., r_q(\overline{X_q}), r^{\alpha}_{q+1}(\overline{X}, \overline{Y})$. Let $\mathcal{D}$ be a database instance. The set of accesses implied in the evaluation of ϕ over $\mathcal{D}$, noted $Accesses(\phi, \mathcal{D})$ is defined as follows:*

$$Accesses(\phi, \mathcal{D}) = \{v(\overline{X}) \mid v \text{ valuation on } \left(\bigcup_{i=1}^{q} var(\overline{X_i})\right) \text{ such that } \left(\forall i \in \{1, \ldots, q\}\ r_i(v(\overline{X_i})) \in r_i(\mathcal{D})\right)\}$$

Defining ϕ' the $\mathcal{CQ}$ query as $ans'(\overline{Z}) \leftarrow r_1(\overline{X_1}), ..., r_q(\overline{X_q})$, it is straightforward to see that:

$$Accesses(\phi, \mathcal{D}) = \phi'(\mathcal{D})$$

Since this operational semantics encompasses $\mathcal{CQ}$ and $\mathcal{CQ}^{\alpha_{Last}}$ queries, in the rest of this thesis, we will assume the use of this semantics for both kinds of queries. Since $\mathcal{CQ}^{\alpha_{Last}}$ queries are just a slight extension of $\mathcal{CQ}$ queries, then it is natural that they are still monotonous.

Lemma 6.2.8. *$\mathcal{CQ}^{\alpha_{Last}}$ queries are monotonous.*

Proof. Consider two database instances $\mathcal{D}_1$,$\mathcal{D}_2$ over the same schema, such that $\mathcal{D}_1 \subseteq \mathcal{D}_2$. Let ϕ be a $\mathcal{CQ}^{\alpha_{Last}}$ query with the same notation as in definition 6.2.4. Consider a fact $t \in \phi(\mathcal{D}_1)$. According to definition 6.2.6, there exists v valuation on $\left(\bigcup_{i=1}^{q} var(\overline{X_i})\right) \cup var(\overline{Y})$ such that
$\left(\forall i \in \{1, \ldots, q\}\ r_i(v(\overline{X_i})) \in r_i(\mathcal{D}_1)\right)$ and $\left(r_{q+1}(v(\overline{X}), v(\overline{Y})) \in r^{\alpha}_{q+1}(\mathcal{D}_1)\right)$.
Since $\mathcal{D}_1 \subseteq \mathcal{D}_2$ then $\forall i \in \{1, \ldots, q+1\}\ r_i(\mathcal{D}_1) \subseteq r_i(\mathcal{D}_2)$ and then:
$\left(\forall i \in \{1, \ldots, q\}\ r_i(v(\overline{X_i})) \in r_i(\mathcal{D}_2)\right)$ and $\left(r_{q+1}(v(\overline{X}), v(\overline{Y})) \in r^{\alpha}_{q+1}(\mathcal{D}_2)\right)$.
And thus $t \in \phi(\mathcal{D}_2)$. This shows the monotonicity of ϕ. □

It is now possible to define the operational semantics of $Datalog^{\alpha_{Last}}$ queries. As for *Datalog* queries [Abiteboul et al., 1995], we define it as the fixpoint of an immediate consequence operator.

Definition 6.2.9 (Immediate consequence of a $Datalog^{\alpha_{Last}}$ query)**.** *Let Φ be a $Datalog^{\alpha_{Last}}$ query such that $\Phi = \{\phi_1, \ldots, \phi_n\}$, each ϕ_j belonging to $\mathcal{CQ}^{\alpha_{Last}}$. Let $\mathcal{D}$ be a database instance with $schema(\mathcal{D}) = schema(\Phi)$. A fact f is an immediate consequence of Φ and $\mathcal{D}$ if*

- $\exists r_i \in schema(\Phi)$ *such that* $f \in r_i(\mathcal{D})$ *(or* $f \in r_i^{\alpha}(\mathcal{D})$ *if* r_i *has an access pattern* α*)*
- *and/or* $f \in \phi_j(\mathcal{D})$ *for some rule* $\phi_j, j \in \{1, \ldots, n\}$ *of* Φ.

Definition 6.2.10 (Imm. consequence operator for a $Datalog^{\alpha_{Last}}$ query)**.** *Let* Φ *be a* $Datalog^{\alpha_{Last}}$ *query such that* $\Phi = \{\phi_1, \ldots, \phi_n\}$*, each* ϕ_j *belonging to* $\mathcal{CQ}^{\alpha_{Last}}$*. Let* $\mathcal{D}$ *be a database instance with* $schema(\mathcal{D}) = schema(\Phi)$*. The immediate consequence operator of* Φ*, noted* T_Φ *is a mapping defined as follows:*
T_Φ :

$$\begin{array}{rcl} \{db\ instances\ on\ schema(\Phi)\} & \rightarrow & \{db\ instances\ on\ schema(\Phi)\} \\ \mathcal{D} & \mapsto & T_\Phi(\mathcal{D}) = \{immediate\ consequences\ of\ \Phi\ and\ \mathcal{D}\} \end{array}$$

By definition 6.2.9, it is easy to see that we have:

- $\mathcal{D} \subseteq T_\Phi(\mathcal{D})$,
- $\forall \phi_j \in \Phi$, $\phi_j(\mathcal{D}) \subseteq T_\Phi(\mathcal{D})$
- and there isn't anything else in $T_\Phi(\mathcal{D})$.

This proves the following lemma.

Lemma 6.2.11. *Let* Φ *be a* $Datalog^{\alpha_{Last}}$ *query such that* $\Phi = \{\phi_1, \ldots, \phi_n\}$*, each* ϕ_j *belonging to* $\mathcal{CQ}^{\alpha_{Last}}$*. Let* $\mathcal{D}$ *be a database instance with* $schema(\mathcal{D}) = schema(\Phi)$*. We have:* $T_\Phi(\mathcal{D}) = \mathcal{D} \cup \bigcup_{j=1}^{n} \phi_j(\mathcal{D})$

We can now show this operation is monotonous.

Lemma 6.2.12. *The immediate consequence operator for a* $Datalog^{\alpha_{Last}}$ *query is monotonous.*

Proof. Let Φ be a $Datalog^{\alpha_{Last}}$ query such that $\Phi = \{\phi_1, \ldots, \phi_n\}$. Let $\mathcal{D}$ be a database instance with $schema(\mathcal{D}) = schema(\Phi)$. To prove that the operator T_Φ is monotonous, we need to prove that for each database instance couple $(\mathcal{D}_1, \mathcal{D}_2)$ on $schema(\Phi)$, if $\mathcal{D}_1 \subseteq \mathcal{D}_2$, then $T_\Phi(\mathcal{D}_1) \subseteq T_\Phi(\mathcal{D}_2)$.

We have $T_\Phi(\mathcal{D}_1) = \mathcal{D}_1 \cup \bigcup_{j=1}^{n} \phi_j(\mathcal{D}_1)$ and $T_\Phi(\mathcal{D}_2) = \mathcal{D}_2 \cup \bigcup_{j=1}^{n} \phi_j(\mathcal{D}_2)$. We assume $\mathcal{D}_1 \subseteq \mathcal{D}_2$. We have $\forall i \in \{1, \ldots, n\}, \phi_i(\mathcal{D}_1) \subseteq \phi_i(\mathcal{D}_2)$ thanks to lemma 6.2.8. The results follows straightforwardly. □

Lemma 6.2.13. *Let Φ be a Datalog$^{\alpha_{Last}}$ query such that $\Phi = \{\phi_1, \ldots, \phi_n\}$. Let $\mathcal{D}$ be a database instance with schema($\mathcal{D}$) = schema(Φ). There exists a fixpoint for the operator T_Φ containing $\mathcal{D}$.*

Proof. Φ contains a finite number of $\mathcal{CQ}^{\alpha_{Last}}$ queries. By definition, $\mathcal{D}$ contains a finite number of facts. Since no new constant is ever generated during the evaluation of a $\mathcal{CQ}^{\alpha_{Last}}$ query (see definition 6.2.6), then the set of all different facts of which $\phi_j(\mathcal{D})$ is a subset if finite, $\forall \phi_j \in \Phi$.

We have seen that $T_\Phi(\mathcal{D}) = \mathcal{D} \cup \bigcup_{j=1}^{n} \phi_j(\mathcal{D})$. Thus $\mathcal{D} \subseteq T_\Phi(\mathcal{D})$. Since T_Φ is monotonous, then there is $T_\Phi(\mathcal{D}) \subseteq T_\Phi(T_\Phi(\mathcal{D}))$. And thus $\mathcal{D} \subseteq T_\Phi^n(\mathcal{D}) \subseteq T_\Phi^{n+1}(\mathcal{D})$, for all integer $n \geq 1$.

From the last argument of the previous two paragraphs, we derive easily that there exists an integer n_0 such that $T_\Phi^{n_0}(\mathcal{D}) = T_\Phi^{n_0+1}(\mathcal{D})$, i.e. $T_\Phi^{n_0}(\mathcal{D})$ is a fix point of T_Φ. Last argument of the previous paragraph shows this fixpoint contains $\mathcal{D}$. Obviously, we can suppose n_0 is minimum, i.e. $T_\Phi^{n_0-1}(\mathcal{D}) \subsetneq T_\Phi^{n_0}(\mathcal{D})$. □

We can now define the semantics of *Datalog*$^{\alpha_{Last}}$ queries.

Definition 6.2.14 (*Datalog*$^{\alpha_{Last}}$ operational semantics)**.** *Let Φ be a Datalog$^{\alpha_{Last}}$ query. Let $\mathcal{D}$ be a database instance with schema($\mathcal{D}$) = schema(Φ). The semantics of Φ is defined as follows: $\Phi(\mathcal{D}) = T_\Phi^{n_0}(\mathcal{D})$ with (i) T_Φ is the immediate consequence operator of Φ, and (ii) n_0 is the least integer such that $T_\Phi^{n_0}(\mathcal{D}) = T_\Phi^{n_0+1}(\mathcal{D})$ (which always exists as shown in lemma 6.2.13).*

Algorithm 6.2 is the algorithmic translation of the previous definition. It is structured as follows:

- Lines 1 to 4 and 31 to 35 describe the iterative process associated to the operator T_Φ.
- Lines 5 to 30 describe the computation of $T_\Phi(\mathcal{D}_{new})$ (one iteration of the process) according to the formula given in lemma 6.2.11.
  - Line 5: $\mathcal{D}_{new}$ is saved into $T_\Phi(\mathcal{D}_{new})$ to keep the result of the previous iteration.
  - Lines 6 to 26: for each $\phi \in \Phi$, with $\phi \in \mathcal{CQ}^{\alpha_{Last}}$ and $\phi \notin \mathcal{CQ}$, we compute $\phi(\mathcal{D}_{new})$. Following definition 6.2.6, the idea is to "materialize" the relation with an access pattern by first computing all possible inputs and then by

calling the associated operation for each of them (lines 7 to 20). Once the relation is materialized by storing the returned results, it can be used in standard conjunctive query evaluation (lines 21 to 25).
- Lines 27 to 29: for each $\phi \in \Phi$, with $\phi \notin \mathcal{CQ}^{\alpha_{Last}}$ and $\phi \in \mathcal{CQ}$, we compute $\phi(\mathcal{D}_{new})$.
- Line 30: this is the replacement of the old database instance $\mathcal{D}_{new}$ by the newly computed one $T_\Phi(\mathcal{D}_{new})$.

Now, we can define the semantics of a $\mathcal{CQ}^\alpha$ query ϕ as the semantics of a $Datalog^{\alpha_{Last}}$ query Φ built from ϕ as it is done in example 6.2.3. A generalization of the construction explained in this example is given in the algorithm that generates Φ from ϕ. This is algorithm 6.1. It is clear that algorithm 6.1 generates a $Datalog^{\alpha_{Last}}$ program since every generated rule is either a $\mathcal{CQ}^{\alpha_{Last}}$ query or a $\mathcal{CQ}$ query. We now end this section by giving the semantics of $\mathcal{CQ}^\alpha$ queries in definition 6.2.15.

Data: A $\mathcal{CQ}^\alpha$ query ϕ
Result: A $Datalog^{\alpha_{Last}}$ query Φ with which the semantics of ϕ is defined.
$\Phi := \emptyset$;
for $f \in body(\phi)$ **do**
  for *output variable* Y *in* f **do**
    $\Phi := \Phi \cup \{dom_Y(Y) \leftarrow (\bigwedge_{X \text{ input variable in } f} dom_X(X)) \wedge f\}$;
for $f \in body(\phi)$ *with* $f = r^\alpha(\overline{X})$ **do**
  $\Phi := \Phi \cup \{r'(\overline{X}) \leftarrow (\bigwedge_{X \text{ input variable in } f} dom_X(X)) \wedge f\}$;
$\Phi := \Phi \cup \{\phi' : head(\phi) \leftarrow (\bigwedge_{f \in body(\phi) \text{ with } f = r^\alpha(\overline{X})} r'(\overline{X}))\}$;
Replace all "$\wedge$" symbols in Φ by "," to respect the rule notation ;
Return Φ;

Algorithm 6.1: The $ExecTransform$ algorithm

Definition 6.2.15 ($\mathcal{CQ}^\alpha$ operational semantics)**.** *Let ϕ be a $\mathcal{CQ}^\alpha$ query. Let $\mathcal{D}$ be a database instance. The semantics of ϕ is defined as follows:*

$$\phi(\mathcal{D}) = (ExecTransform(\phi))(\mathcal{D})$$

Data: A database instance $\mathcal{D}$, a $Datalog^{\alpha_{Last}}$ query $\Phi = \{\phi_1, \ldots, \phi_n\}$ such that $schema(\mathcal{D}) = schema(\Phi)$ and $\forall r \in schema(\Phi)$, $r(\mathcal{D}) \neq \emptyset$ if r has an access pattern, and $r(\mathcal{D}) = \emptyset$ otherwise.

Result: $\Phi(\mathcal{D})$ according to definition 6.2.14.

$\mathcal{D}_{new} := \mathcal{D}$;
$stop := false$;
while $stop = false$ **do**
  $\mathcal{D}_{old} := \mathcal{D}_{new}$;
  $T_\Phi(\mathcal{D}_{new}) := \mathcal{D}_{new}$;
  for $\phi : ans(\overline{Z}) \leftarrow r_1(\overline{X_1}), \ldots, r_q(\overline{X_q}), r^\alpha_{q+1}(\overline{X}, \overline{Y}) \in \Phi$ **do**
    if ϕ *is such that*
      . α *has* ***at least one "i"***
      . *and oper is the callable operation associated to* r^α_{q+1}
    then
      $\phi(\mathcal{D}_{new}) := \emptyset$;
      $\phi' := ans'(\overline{X}) \leftarrow r_1(\overline{X_1}), \ldots, r_q(\overline{X_q})$;
      $Accesses(\phi, \mathcal{D}_{new}) := \phi'(\mathcal{D}_{new})$;
      $r^*_{q+1}(\mathcal{D}_{new}) := \emptyset$;
      for $\overline{c} \in Accesses(\phi, \mathcal{D}_{new})$ **do**
        $r^*_{q+1}(\mathcal{D}_{new}) := r^*_{q+1}(\mathcal{D}_{new}) \cup \{r^*_{q+1}(\overline{d}) |\ \overline{d} = \overline{ce}$ and $\overline{e}$ is obtained by calling *oper* with input $\overline{c}\}$;
    if ϕ *is such that*
      . α *is* ***made up with "o"'s only***
      . *and oper is the callable operation associated to* r^α_{q+1}
    then
      $r^*_{q+1}(\mathcal{D}_{new}) := \{r^*_{q+1}(\overline{d}) | \overline{d}$ is obtained by calling *oper* with no input$\}$;
    if $r^*_{q+1}(\mathcal{D}_{new}) \neq \emptyset$ **then**
      $\phi^* := ans(\overline{Z}) \leftarrow r_1(\overline{X_1}), \ldots, r_q(\overline{X_q}), r^*_{q+1}(\overline{X}, \overline{Y})$;
      $\phi(\mathcal{D}_{new}) := \phi^*(\mathcal{D}_{new})$;
      $T_\Phi(\mathcal{D}_{new}) := T_\Phi(\mathcal{D}_{new}) \cup \phi(\mathcal{D}_{new})$;
  for $\phi \in \Phi$ *with* $\phi \in \mathcal{CQ}$ **do**
    $T_\Phi(\mathcal{D}_{new}) := T_\Phi(\mathcal{D}_{new}) \cup \phi(\mathcal{D}_{new})$;
  $\mathcal{D}_{new} := T_\Phi(\mathcal{D}_{new})$;
  if $\mathcal{D}_{new} = \mathcal{D}_{old}$ **then**
    $stop := true$;
return S_{new};

Algorithm 6.2: Operational Semantics of $Datalog^{\alpha_{Last}}$ translated as a naive $Datalog^{\alpha_{Last}}$ query evaluation algorithm

6.2.3 $\mathcal{CQ}^{\alpha}$ Queries Accesses

In the previous section, we have proposed a definition for the operational semantics of $\mathcal{CQ}^{\alpha}$ queries, and we have given an algorithm to evaluate such queries according to this semantics. Now we want to determine an upper bound on the number of accesses implied in the evaluation process. Indeed, in DaWeS, we want to use this upper bound to compare future algorithms or heuristics made to optimize queries so that their evaluation implies less accesses. Since during evaluation, calling web services can have a major impact on the overall evaluation time, then it is natural to try to limitate the number of such calls, i.e. accesses. We have chosen to express this bound wrt the sizes of the relations having access patterns, i.e. wrt the maximum number of tuples the associated web service operations can return. In fact, we do not know their precise values, but we do not need to know them. We just define our bound according to them, so that two bounds corresponding to two differently optimized queries can be compared wrt the same parameters.

Since the semantics of a $\mathcal{CQ}^{\alpha}$ query Ψ is based on the semantics of its associated $Datalog^{\alpha_{Last}}$ query Φ (with $\Phi = ExecTransform(\Psi)$), we define the set of accesses implied in the evaluation of Ψ over a database instance $\mathcal{D}$ as the set of accesses implied in the evaluation of Φ over $\mathcal{D}$.

Definition 6.2.16 (Set of accesses for a $Datalog^{\alpha_{Last}}$ query)**.** *Consider a $Datalog^{\alpha_{Last}}$ query $\Phi = \{\phi_j, j \in \{1, \ldots, m\}\}$. Let $\mathcal{D}_0$ be a database instance. Let n_0 be the number of iteration needed to reach the fixpoint during the evaluation of Φ over $\mathcal{D}_0$. Let $\mathcal{D}_1, \ldots, \mathcal{D}_{n_0}$ be databases instances obtained at the end of iteration $1, \ldots, n_0$, respectively. The set of accesses implied in the evaluation of Φ over $\mathcal{D}_0$, noted $Accesses(\Phi, \mathcal{D}_0)$ is defined as follows:*

$$Accesses(\Phi, \mathcal{D}_0) = \bigcup_{i=0}^{n_0} \bigcup_{j=1}^{m} \{Accesses(\phi_j, \mathcal{D}_i)\}$$

This definition implies $Accesses(\Phi, \mathcal{D}_0)$ may contain many times the same access if this one is generated at many iterations. A maybe more precise, but a bit less intuitive, definition could have been:
$Accesses(\Phi, \mathcal{D}_0) = \bigcup_{j=1}^{m} \{ \bigcup_{i=0}^{n_0} Accesses(\phi_j, \mathcal{D}_i)\}$.
Anyway, we are especially interested in bounding the cardinality of this set. And in both cases, we can easily state the following lemma.

Lemma 6.2.17. *Consider a $Datalog^{\alpha_{Last}}$ query $\Phi = \{\phi_j, j \in \{1, \ldots, m\}\}$. Let $\mathcal{D}_0$ be a database instance. Let n_0 be the number of iterations needed to reach the fixpoint*

during the evaluation of Φ *over* $\mathcal{D}_0$*. Let* $\mathcal{D}_1, \ldots, \mathcal{D}_{n_0}$ *be databases instances obtained at the end of iteration* $1, \ldots, n_0$*, respectively. We have:*

$$|Accesses(\Phi, \mathcal{D}_0)| \leq \sum_{i=0}^{n_0} \sum_{j=1}^{m} |Accesses(\phi_j, \mathcal{D}_i)|$$

Now we can define the set of accesses for a $\mathcal{CQ}^\alpha$ query over a database instance $\mathcal{D}$.

Definition 6.2.18 (Set of accesses for a $\mathcal{CQ}^\alpha$ query)**.** *Consider a* $\mathcal{CQ}^\alpha$ *query* Ψ*. Let* $\mathcal{D}$ *be a database instance. The set of accesses implied in the evaluation of* Ψ *over* $\mathcal{D}$*, noted* $Accesses(\Psi, \mathcal{D})$ *is defined as follows:*

$$Accesses(\Psi, \mathcal{D}) = Accesses(ExecTransform(\Psi), \mathcal{D})$$

Lemma 6.2.17 is the starting point of our study of how to bound the size of the number of accesses. This study folllows on the three following sections. In section 6.2.4, we see how to bound the size of the result (i.e. the number of tuples in the result) of a $\mathcal{CQ}$ query (without access pattern) evaluated on a database instance. In section 6.2.5, we study how to use the previous bound to bound the number of accesses of a $\mathcal{CQ}^\alpha$ query and of a $Datalog^{\alpha_{Last}}$ query. To this purpose, by looking at algorithm 6.2, two steps are necessary:

- computing a bound on the number of accesses necessary at each iteration
- and computing a bound on each relation in the database instance on which the operator T_Φ is applied, at each iteration.

At last, in section 6.2.6, we apply our results to obtain a precise upper bound for the number of accesses of the output of the inverse-rules algorithm which is a $Datalog^{\alpha_{Last}}$ query with special properties.

6.2.4 Bounding the Number of $\mathcal{CQ}$ Query Answers

Let $\mathcal{D}$ be a database instance. Let $\phi : q(X_1, X_2, X_3) \leftarrow r_1(X_1, X_2), r_2(X_2, X_3), r_3(X_3)$ be a conjunctive query (cf. definition 4.1.1). We suppose $\{r_1, r_2, r_3\} \subseteq schema(\mathcal{D})$. According to definition 4.1.2, evaluating ϕ on $\mathcal{D}$ amounts to finding all valuations v such that $r_1(v(X_1), v(X_2)) \in r_1(\mathcal{D})$, $r_2(v(X_2), v(X_3)) \in r_2(\mathcal{D})$ and $r_3(v(X_3)) \in r_3(\mathcal{D})$. Then for all these valuations v, $q(v(X_1), v(X_2), v(X_3))$ is an answer of ϕ. It is quite obvious that the number of these answers depends on the size of the relations, i.e. on $|r_1(\mathcal{D})|$, $|r_2(\mathcal{D})|$ and $|r_3(\mathcal{D})|$. In our example, the value of X_1 can only come from a

tuple of r_1, so there are at most $|r_1(\mathcal{D})|$ different values for X_1. The value of X_2 can come from r_1 or r_2. Since there is a join, each value for X_2 must be in both relations. So there are at most $Min(|r_1(\mathcal{D})|, |r_2(\mathcal{D})|)$ possible values for X_2. Similarly, there are at most $Min(|r_2(\mathcal{D})|, |r_3(\mathcal{D})|)$ possible values for X_3. Thus, a first bound would be:

$$|\phi(\mathcal{D})| \leq |r_1(\mathcal{D})| * Min(|r_1(\mathcal{D})|, |r_2(\mathcal{D})|) * Min(|r_2(\mathcal{D})|, |r_3(\mathcal{D})|)$$

This bound is however a bit rough. Indeed we have not taken into account that values for X_1 and X_2 that come from r_1 are linked (inside tuples of r_1). Thus the previous bound may count these values twice. In fact, it appears that we have considered each variable independently. In other words, we have only considered the following partition of the variables of the head of ϕ:

$$\{\{X_1\}, \{X_2\}, \{X_3\}\}$$

For each element of this partition, we have taken the minimum cardinality of all relations that contain this element. So, to be complete, we have to envision all partitions of these variables and take the cardinality of all relations that contain this element. For example, let's take the following partition:

$$\{\{X_1, X_2\}, \{X_3\}\}$$

Couples of values for both X_1 and X_2 can only come from r_1 (since X_1 is not in $r_2(X_2, X_3)$). Since values for X_3 still come from r_2 or r_3, we can derive the following bound:

$$|\phi(\mathcal{D})| \leq |r_1(\mathcal{D})| * Min(|r_2(\mathcal{D})|, |r_3(\mathcal{D})|)$$

This bound is better than the first. If we generalized the process to all partitions, and keep only the minimal bound (more precisely, one of the minimal bounds since there may be many), then we have a still better upper bound to the number of answers. By generalizing the previous reasoning, it is possible to prove the following lemma.

Lemma 6.2.19. *Let $\mathcal{D}$ be a database instance. Let $\phi : q(\overline{X}) \leftarrow r_1(\overline{X_1}), \ldots, r_q(\overline{X_q}) \in \mathcal{CQ}$. Let $\mathcal{B}(\phi, \mathcal{D})$ be defined as follows:*

$$\mathcal{B}(\phi, \mathcal{D}) = \min_{\substack{\mathcal{X} \text{ partition of } var(\overline{X}) \text{ such that} \\ \forall x \in \mathcal{X}, \exists i \in \{1,\ldots,q\} | x \subseteq var(\overline{X_i})}} \left(\prod_{x \in \mathcal{X}} \left(\min_{\substack{i \in \{1,\ldots,q\}| \\ x \subseteq var(\overline{X_i})}} |r_i(\mathcal{D})| \right) \right)$$

Then we have:

$$|\phi(\mathcal{D})| \leq \mathcal{B}(\phi, \mathcal{D})$$

6.2.5 Bounding the Number of $\mathcal{CQ}^{\alpha}$ Query Accesses

We now see how to use the previous bound to bound the number of accesses needed during the evaluation of a $\mathcal{CQ}^{\alpha}$ query or a $Datalog^{\alpha_{Last}}$ one. We recall that there are two steps to evaluate a $\mathcal{CQ}^{\alpha}$ query Ψ: first it needs to be transformed with algorithm 6.1 into a $Datalog^{\alpha_{Last}}$ query Φ (which is a set of $\mathcal{CQ}^{\alpha_{Last}}$ queries), and then this $Datalog^{\alpha_{Last}}$ query is evaluated using algorithm 6.2. During algorithm 6.2, for any ϕ in Φ that has an access pattern α on its last atom, we have:

- Either there is at least one input in α, and then the accesses needed when evaluating ϕ on $\mathcal{D}$ are the answers of ϕ' evaluated on $\mathcal{D}$ (see line 10 for the definition of ϕ' and line 11 for the evaluation of ϕ' in algorithm 6.2). I.e. $Accesses(\phi, \mathcal{D}) = \phi'(\mathcal{D})$ (cf. definition 6.2.7). Since $\phi' \in \mathcal{CQ}$, then $|\phi'(\mathcal{D})| \leq \mathcal{B}(\phi', \mathcal{D})$ according to lemma 6.2.19. So $|Accesses(\phi, \mathcal{D})| \leq \mathcal{B}(\phi', \mathcal{D})$.
- Or there is no input in α and then there is only one access needed during the evaluation of ϕ on $\mathcal{D}$ (see line 19 of algorithm 6.2).
- Or there is no relation having an access pattern. In this case there is no access.

However, this is not sufficient to define the researched bound since all $\mathcal{B}(\phi', \mathcal{D})$ bounds are expressed using relations used to define ϕ' queries, and these relations (thereafter called temporary relations) are not the original relations (those that define Ψ and that have access patterns).

Fortunately, algorithm 6.2 gives a way of linking temporary and original relations. In the iterative process implemented in algorithm 6.2, for each iteration, we can bound the cardinality of temporary relations by the sum of the same cardinality obtained at the previous iteration and the number of new tuples for these relations computed at the current iteration. More precisely, at line 6, for one $\phi \in \Phi$, we can see that $\{ans, r_1, \ldots, r_q\}$ is the set of temporary relations and r^{α}_{q+1} is the only original relation. The computation for new tuples for ans is achieved at line 22, after the materialization of r^{α}_{q+1} into r^{*}_{q+1} (i.e. after doing all needed calls to the corresponding web service operation and saving the returned results). This materialization step transforms ϕ into ϕ^* which is a $\mathcal{CQ}$ query. So, according to lemma 6.2.19, we can bound $|\phi^*(\mathcal{D})|$ by $\mathcal{B}(\phi^*, \mathcal{D})$. If we suppose that $\mathcal{D}_i$ is the database instance obtained after the i^{th} iteration, and $\mathcal{D}_{i+1}$ the instance obtained after the $i + 1^{th}$ iteration, then we have, for any temporary relation ans:

$$ans(\mathcal{D}_{i+1}) = ans(\mathcal{D}_i) \cup \bigcup_{\substack{\phi \in \Phi \text{ such that} \\ ans \text{ is the head relation of } \phi}} \phi^*(\mathcal{D}_i)$$

Using lemma 6.2.19, it is easy to derive:

$$|ans(\mathcal{D}_{i+1})| \leq |ans(\mathcal{D}_i)| + \sum_{\substack{\phi \in \Phi \text{ such that} \\ ans \text{ is the head relation of } \phi}} |\phi^*(\mathcal{D}_i)|$$

$$|ans(\mathcal{D}_{i+1})| \leq |ans(\mathcal{D}_i)| + \sum_{\substack{\phi \in \Phi \text{ such that} \\ ans \text{ is the head relation of } \phi}} \mathcal{B}(\phi^*, \mathcal{D}_i)$$

Let's summary what we've seen in the next lemma.

Lemma 6.2.20. *Consider a $Datalog^{\alpha_{Last}}$ query $\Phi = \{\phi_j, j \in \{1, \ldots, m\}\}$. Let $\mathcal{D}_0$ be a database instance. Let n_0 be the number of iteration needed to reach the fixpoint during the evaluation of Φ over $\mathcal{D}_0$. Let $\mathcal{D}_1, \ldots, \mathcal{D}_{n_0}$ be databases instances obtained at the end of iteration $1, \ldots, n_0$, respectively.*
We suppose for each $\phi_j \in \Phi$ we have built (cf. algorithm 6.2) two queries ϕ'_j and ϕ^*_j. ϕ'_j is the same as ϕ_j without its last atom, and ϕ_j* is the same as ϕ_j with a materialized relation for its last atom (and not the relation with an access pattern).

After the i^{th} iteration we have, for each $\phi_j \in \Phi$, and for each temporary relation ans:

(i) $|Accesses(\phi_j, \mathcal{D}_i)| \leq \mathcal{B}(\phi'_j, \mathcal{D}_i)$,
or $|Accesses(\phi_j, \mathcal{D}_i)| = 1$ *if there is no input in the access pattern,*
or $|Accesses(\phi_j, \mathcal{D}_i)| = 0$ *if there is no relation with an access pattern in* ϕ.

(ii) $$|ans(\mathcal{D}_{i+1})| \leq |ans(\mathcal{D}_i)| + \sum_{\substack{\phi \in \Phi \text{ such that} \\ ans \text{ is the head relation of } \phi}} \mathcal{B}(\phi^*, \mathcal{D}_i)$$

In the next section, we see how to use this lemma in the special case of the inverse-rules algorithm output to obtain an upper bound expressed wrt original relations only.

6.2.6 Bounding the Number of Accesses for the Inverse-Rules Algorithm Output

We now focus on using the previous result to the special case of the $Datalog^{\alpha_{Last}}$ query obtained as the result of the inverse-rules algorithm. As for $Datalog^{\alpha_{Last}}$ queries obtained after applying algorithm 6.1 to a $\mathcal{CQ}^{\alpha}$ query, the query obtained as the output of the inverse-rules algorithm has the following properties:

- temporary relations (ie. domain relations) are unary relations

- and $\mathcal{CQ}^{\alpha_{Last}}$ rules (inside the $Datalog^{\alpha_{Last}}$ query) have only one variable in their head.

These characteristics of the output of the inverse-rules algorithm allow to express the bound on the number of accesses wrt the cardinalities of the original relations (ie. relations having an access pattern). This is the objective of this section.

In lemma 6.2.20, in (i), $\mathcal{B}(\phi', \mathcal{D}_i)$ is expressed with cardinalities of temporary relations only. These relations are bounded by (ii) at each iteration. Besides, in (ii), $\mathcal{B}(\phi^*, \mathcal{D}_i)$ is also expressed with cardinalities of temporary relations but also with the cardinality of one original (materialized) relation. Since the iterative process mandatorily stops after a finite number n_0 of iterations (this comes from lemma 6.2.13) and since at the beginning $ans(\mathcal{D}_0) = \emptyset$ (thus $|ans(\mathcal{D}_0)| = 0$) for all temporary relation ans, then it is possible to bound the number of accesses needed when evaluating Φ on $\mathcal{D}$ by an expression using only the cardinalities of the original relations, i.e. the maximum number of tuples that are obtainable from the corresponding web service operations.

Example 6.2.21. *Suppose we have the following $\mathcal{CQ}^{\alpha}$ query Ψ:*

$$q(X_1, X_3, X_5) \leftarrow r_1^o(X_2), r_2^o(X_1), r_3^{iioo}(X_1, X_2, X_3, X_4), r_4^{ioo}(X_4, X_5, X_1)$$

After algorithm 6.1, we obtain the $Datalog^{\alpha_{Last}}$ query Φ (q is the query predicate):

$\phi_1 : dom_{X_1}(X_1) \leftarrow r_2^o(X_1)$
$\phi_2 : dom_{X_2}(X_2) \leftarrow r_1^o(X_2)$
$\phi_3 : dom_{X_3}(X_3) \leftarrow dom_{X_1}(X_1), dom_{X_2}(X_2), r_3^{iioo}(X_1, X_2, X_3, X_4)$
$\phi_4 : dom_{X_4}(X_4) \leftarrow dom_{X_1}(X_1), dom_{X_2}(X_2), r_3^{iioo}(X_1, X_2, X_3, X_4)$
$\phi_5 : dom_{X_5}(X_5) \leftarrow dom_{X_4}(X_4), r_4^{ioo}(X_4, X_5, X_1)$
$\phi_6 : dom_{X_1}(X_1) \leftarrow dom_{X_4}(X_4), r_4^{ioo}(X_4, X_5, X_1)$
$\phi_7 : rr_1(X_2) \leftarrow r_1^o(X_2)$
$\phi_8 : rr_2(X_1) \leftarrow r_2^o(X_1)$
$\phi_9 : rr_3(X_1, X_2, X_3, X_4) \leftarrow dom_{X_1}(X_1), dom_{X_2}(X_2), r_3^{iioo}(X_1, X_2, X_3, X_4)$
$\phi_{10} : rr_4(X_4, X_5, X_1) \leftarrow dom_{X_4}(X_4), r_4^{ioo}(X_4, X_5, X_1)$
$\phi_{11} : q(X_1, X_3, X_5) \leftarrow rr_1(X_2), rr_2(X_1), rr_3(X_1, X_2, X_3, X_4), rr_4(X_4, X_5, X_1)$

Here, the original relations are:
$\{r_1^o, r_2^o, r_3^{iioo}, r_4^{ioo}\}$,
and the temporary relations are:
$\{dom_{X_1}, dom_{X_2}, dom_{X_3}, dom_{X_4}, dom_{X_5}, rr_1, rr_2, rr_3, rr_4\}$.

During algorithm 6.2, the following $\mathcal{CQ}^{\alpha_{Last}}$ queries are build to generate the accesses sets:

$\phi'_3 : ans'_3(X_1, X_2) \leftarrow dom_{X_1}(X_1), dom_{X_2}(X_2)$
$\phi'_4 : ans'_4(X_1, X_2) \leftarrow dom_{X_1}(X_1), dom_{X_2}(X_2)$
$\phi'_5 : ans'_5(X_4) \leftarrow dom_{X_4}(X_4)$
$\phi'_6 : ans'_6(X_4) \leftarrow dom_{X_4}(X_4)$
$\phi'_9 : ans'_9(X_1, X_2) \leftarrow dom_{X_1}(X_1), dom_{X_2}(X_2)$
$\phi'_{10} : ans'_{10}(X_4) \leftarrow dom_{X_4}(X_4)$

Moreover, for each $j \in \{1, \ldots, 10\}$, ϕ^*_j queries are built. These are the same as ϕ_j queries except that each relation with access pattern r^α_{q+1} has been materialized into r^*_{q+1}. After the end of the i^{th} iteration, we have (according to (i)):

$|Accesses(\phi_1, \mathcal{D}_i)| = 1$
$|Accesses(\phi_2, \mathcal{D}_i)| = 1$
$|Accesses(\phi_3, \mathcal{D}_i)| \leq \mathcal{B}(\phi'_3, \mathcal{D}_i) = |dom_{X_1}(\mathcal{D}_i)| * |dom_{X_2}(\mathcal{D}_i)|$
$|Accesses(\phi_4, \mathcal{D}_i)| \leq \mathcal{B}(\phi'_4, \mathcal{D}_i) = |dom_{X_1}(\mathcal{D}_i)| * |dom_{X_2}(\mathcal{D}_i)|$
$|Accesses(\phi_5, \mathcal{D}_i)| \leq \mathcal{B}(\phi'_5, \mathcal{D}_i) = |dom_{X_4}(\mathcal{D}_i)|$
$|Accesses(\phi_6, \mathcal{D}_i)| \leq \mathcal{B}(\phi'_6, \mathcal{D}_i) = |dom_{X_4}(\mathcal{D}_i)|$
$|Accesses(\phi_7, \mathcal{D}_i)| = 1$
$|Accesses(\phi_8, \mathcal{D}_i)| = 1$
$|Accesses(\phi_9, \mathcal{D}_i)| \leq \mathcal{B}(\phi'_9, \mathcal{D}_i) = |dom_{X_1}(\mathcal{D}_i)| * |dom_{X_2}(\mathcal{D}_i)|$
$|Accesses(\phi_{10}, \mathcal{D}_i)| \leq \mathcal{B}(\phi'_{10}, \mathcal{D}_i) = |dom_{X_4}(\mathcal{D}_i)|$
$|Accesses(\phi_{11}, \mathcal{D}_i)| = 0$

and also (according to (ii)):

$$\begin{aligned}|dom_{X_1}(\mathcal{D}_{i+1})| &\leq |dom_{X_1}(\mathcal{D}_i)| + \mathcal{B}(\phi^*_1, \mathcal{D}_i) + \mathcal{B}(\phi^*_6, \mathcal{D}_i) \\ &\leq |dom_{X_1}(\mathcal{D}_i)| + |r^o_2(\mathcal{D}_i)| + |r^{ioo}_4(\mathcal{D}_i)| \\ &\leq |dom_{X_1}(\mathcal{D}_i)| + |r^o_2(\mathcal{D}_0)| + |r^{ioo}_4(\mathcal{D}_0)|\end{aligned}$$
$|dom_{X_2}(\mathcal{D}_{i+1})| \leq |dom_{X_2}(\mathcal{D}_i)| + |r^o_1(\mathcal{D}_0)|$
$|dom_{X_3}(\mathcal{D}_{i+1})| \leq |dom_{X_3}(\mathcal{D}_i)| + |r^{iioo}_3(\mathcal{D}_0)|$
$|dom_{X_4}(\mathcal{D}_{i+1})| \leq |dom_{X_4}(\mathcal{D}_i)| + |r^{iioo}_3(\mathcal{D}_0)|$
$|dom_{X_5}(\mathcal{D}_{i+1})| \leq |dom_{X_5}(\mathcal{D}_i)| + |r^{iioo}_4(\mathcal{D}_0)|$

Now, according to the previous bounds of $|Accesses(\phi_j, \mathcal{D}_i)|$, we only need to precise $|dom_{X_1}(\mathcal{D}_i)|$, $|dom_{X_2}(\mathcal{D}_i)|$ and $|dom_{X_4}(\mathcal{D}_i)|$ to express these bounds with a few $|r^\alpha(\mathcal{D}_i)|$ only, with the r^αs some original relations. We recall that since these relations are

accessed, then no tuple can be added to them during any iteration of the evaluation. So $|r^{\alpha}(\mathcal{D}_i)| = |r^{\alpha}(\mathcal{D}_0)|$ *for all iteration* i*. Let's focus on* $|dom_{X_1}(\mathcal{D}_i)|$*. For* n_0 *the number of the last iteration (during which the fixpoint is generated), we have:*

$|dom_{X_1}(\mathcal{D}_0)| = 0$
$|dom_{X_1}(\mathcal{D}_1)| \leq 0 + |r_2^o(\mathcal{D}_0)| + |r_4^{ioo}(\mathcal{D}_0)|$
$|dom_{X_1}(\mathcal{D}_2)| \leq 2 * (|r_2^o(\mathcal{D}_0)| + |r_4^{ioo}(\mathcal{D}_0)|)$
...
$|dom_{X_1}(\mathcal{D}_i)| \leq i * (|r_2^o(\mathcal{D}_0)| + |r_4^{ioo}(\mathcal{D}_0)|)$
...
$|dom_{X_1}(\mathcal{D}_{n_0})| \leq n_0 * (|r_2^o(\mathcal{D}_0)| + |r_4^{ioo}(\mathcal{D}_0)|)$

Similarly, we have:
$|dom_{X_2}(\mathcal{D}_i)| \leq i * |r_1^o(\mathcal{D}_0)|$
$|dom_{X_4}(\mathcal{D}_i)| \leq i * |r_3^{iioo}(\mathcal{D}_0)|$

So, the total number of accesses, for all iterations and all query ϕ_j *is:*

$$
\begin{aligned}
|Accesses(\Phi, \mathcal{D}_0)| &= \sum_{i=0}^{n_0} \sum_{j=1}^{11} |Accesses(\phi_j, \mathcal{D}_i)| \\
&\leq \sum_{i=0}^{n_0} (4 + 3 * (|dom_{X_1}(\mathcal{D}_i)| * |dom_{X_2}(\mathcal{D}_i)|) + 3 * |dom_{X_4}(\mathcal{D}_i)|) \\
&\leq \sum_{i=0}^{n_0} (4 + 3 * (i * (|r_2^o(\mathcal{D}_0)| + |r_4^{ioo}(\mathcal{D}_0)|) * i * |r_1^o(\mathcal{D}_0)|) + 3 * i * |r_3^{iioo}(\mathcal{D}_0)|) \\
&\overset{since\ i \leq n_0}{\leq} n_0(4 + 3 * (n_0 * (|r_2^o(\mathcal{D}_0)| + |r_4^{ioo}(\mathcal{D}_0)|) * n_0 * |r_1^o(\mathcal{D}_0)|) + 3 * n_0 * |r_3^{iioo}(\mathcal{D}_0)|) \\
&\leq 4n_0 + 3n_0^3 * (|r_2^o(\mathcal{D}_0)| + |r_4^{ioo}(\mathcal{D}_0)|) * |r_1^o(\mathcal{D}_0)|) + 3 * n_0^2 * |r_3^{iioo}(\mathcal{D}_0)|)
\end{aligned}
$$

If we consider n_0 *as a parameter, we can conclude that:*
$|Accesses(\Phi, \mathcal{D}_0)| = \mathcal{O}\left((|r_2^o(\mathcal{D}_0)| + |r_4^{ioo}(\mathcal{D}_0)|) * |r_1^o(\mathcal{D}_0)| + |r_3^{iioo}(\mathcal{D}_0)|\right)$ ■

We generalize the reasoning developed in the previous example with the following lemma.

Lemma 6.2.22. *Consider a* $Datalog^{\alpha_{Last}}$ *query* $\Phi = \{\phi_j, j \in \{1, \ldots, m\}\}$*. Let* $\mathcal{D}_0$ *be a database instance. We assume that* Φ *has the following properties:*

(P1) in the $\mathcal{CQ}^{\alpha_{Last}}$ *rules of* Φ*, the body atoms without access pattern (ie. all body atoms except the last one) are made up with unary relations only*
(P2) and the head of $\mathcal{CQ}^{\alpha_{Last}} \backslash_{\mathcal{CQ}}$ *rules of* Φ *has only one variable.*

We have:

$$|Accesses(\Phi, \mathcal{D}_0)| \leq \mathcal{O}(\sum_{\substack{\phi_j \in \Phi \mid \text{there is at least one input in the access pattern on the last atom of } \phi_j}} (\prod_{\substack{\text{atom } p(\overline{Y}) \mid \\ p(\overline{Y}) \in body(\phi_j) \\ \text{with } p(\overline{Y}) \\ \text{without any access pattern}}} (\sum_{\substack{\phi_k \in \Phi \mid \\ p \text{ is the head predicate of } \phi_k \\ \text{and } r_k^\alpha \text{ is the only relation of } \phi_k \text{ having an access pattern}}} |r_k^\alpha(\mathcal{D}_0)|)))$$

Proof. As stated in lemma 6.2.17:
$|Accesses(\Phi, \mathcal{D}_0)| \leq \sum_{i=0}^{n_0} \sum_{j=1}^{m} |Accesses(\phi_j, \mathcal{D}_i)|$
with n_0 be the number of iterations needed to reach the fixpoint during the evaluation of Φ over $\mathcal{D}_0$, and $\mathcal{D}_1, \ldots, \mathcal{D}_{n_0}$ be databases instances obtained at the end of iterations $1, \ldots, n_0$, respectively.

Now, since in lemma 6.2.20 there is (i), we have:
$|Accesses(\phi_j, \mathcal{D}_i)| \leq \mathcal{B}(\phi'_j, \mathcal{D}_i)$ when there is at least one input in the access pattern of the last atom of ϕ_j
$|Accesses(\phi_j, \mathcal{D}_i)| = 1$ when there is no input in the access pattern of the last atom of ϕ_j
$|Accesses(\phi_j, \mathcal{D}_i)| = 0$ when there is no access pattern in ϕ_j

According to lemma 6.2.19, and since $(P1)$ is verified, we have:
$$\mathcal{B}(\phi'_j, \mathcal{D}_i) = \prod_{\substack{\text{atom } p(\overline{Y}) \mid \\ p(\overline{Y}) \in body(\phi_j) \\ \text{with } p(\overline{Y}) \\ \text{without any} \\ \text{access pattern}}} |p(\mathcal{D}_i)|$$
Thus:
$$|Accesses(\Phi, \mathcal{D}_0)| \leq \sum_{i=0}^{n_0} (\sum_{\substack{\phi_j \in \Phi \mid \text{there} \\ \text{is at least one} \\ \text{input in the} \\ \text{access pattern} \\ \text{on the last} \\ \text{atom of } \phi_j}} (\prod_{\substack{\text{atom } p(\overline{Y}) \mid \\ p(\overline{Y}) \in body(\phi_j) \\ \text{with } p(\overline{Y}) \\ \text{without any} \\ \text{access pattern}}} |p(\mathcal{D}_i)|) + \sum_{\substack{\phi_j \in \Phi \mid \text{there} \\ \text{is no no input} \\ \text{in the access} \\ \text{pattern} \\ \text{on the last} \\ \text{atom of } \phi_j}} 1)$$

Now, since in lemma 6.2.20 there is (ii), we have:
$$|p(\mathcal{D}_i)| \leq |p(\mathcal{D}_{i-1})| + \sum_{\substack{\phi_k \in \Phi \text{ such that} \\ p \text{ is the head relation of } \phi_k}} \mathcal{B}(\phi_k^*, \mathcal{D}_{i-1})$$
with ϕ_k^* is built from ϕ_k as in algorithm 6.2 (ie. ϕ_k^* is the same as ϕ_k except that the last atom relation has been materialized).
Let $\phi_k : p(\overline{X}) \leftarrow r_1(\overline{X_1}), \ldots, r_q(\overline{X_q}), r_{q+1}^\alpha(\overline{X_{q+1}})$.
So $\phi_k^* : p(\overline{X}) \leftarrow r_1(\overline{X_1}), \ldots, r_q(\overline{X_q}), r_{q+1}^*(\overline{X_{q+1}})$.

Since ϕ_k^* is a rule in the output of the inverse-rules algorithm, we have: (a) the head atom of each ϕ_k^* has all its variables present in its last atom (ie. $\overline{X} \subseteq \overline{X_{q+1}}$), and (b) all other atoms (other than the last one) have only one variable. So there is always a partition of $var(\overline{X})$, which is the singleton $\{var(\overline{X})\}$, such that $\forall x \in \{var(\overline{X})\}, \exists h = q+1 \in \{1,\ldots,q+1\} | x \subseteq var(\overline{X_h})$. Then, there is, according to lemma 6.2.19:

$$\mathcal{B}(\phi_k^*, \mathcal{D}_{i-1}) = \min_{\substack{\mathcal{X} \text{ partition of } var(\overline{X}) \\ \text{such that} \\ \forall x \in \mathcal{X}, \exists h \in \{1,\ldots,q+1\} | \\ x \subseteq var(\overline{X_h})}} \left(\prod_{x \in \mathcal{X}} \left(\min_{\substack{h \in \{1,\ldots,q+1\} | \\ x \subseteq var(\overline{X_h})}} |r_h(\mathcal{D}_{i-1})| \right) \right)$$

$$\mathcal{B}(\phi_k^*, \mathcal{D}_{i-1}) = |r_{q+1}^*(\mathcal{D}_{i-1})|$$

Since the extension of relations with access patterns always stays the same, then, at each iteration, materialized relations have the same extension as during the first iteration. This means that, for all $i \geq 1$:

$$\mathcal{B}(\phi_k^*, \mathcal{D}_{i-1}) = |r_{q+1}^*(\mathcal{D}_0)|$$

With the original relation name (the one with access pattern), it gives:

$$\mathcal{B}(\phi_k^*, \mathcal{D}_{i-1}) = |r_{q+1}^{\alpha}(\mathcal{D}_0)|$$

We then derive:

$$|p(\mathcal{D}_i)| \leq |p(\mathcal{D}_{i-1})| + \sum_{\substack{\phi_k \in \Phi \\ \text{such that} \\ p \text{ is the head relation of } \phi_k \\ \text{and } r_k^{\alpha} \text{ the only relation of} \\ \phi_k \text{ having an access pattern}}} |r_k^{\alpha}(\mathcal{D}_0)|$$

As $|p(\mathcal{D}_0)| = 0$, we have:

$$|p(\mathcal{D}_i)| \leq i * \sum_{\substack{\phi_k \in \Phi \\ \text{such that} \\ p \text{ is the head relation of } \phi_k \\ \text{and } r_k^{\alpha} \text{ the only relation of} \\ \phi_k \text{ having an access pattern}}} |r_k^{\alpha}(\mathcal{D}_0)|$$

We obtain:

$$|Accesses(\Phi, \mathcal{D}_0)| \leq$$

$$\sum_{i=0}^{n_0} \Bigg(\sum_{\substack{\phi_j \in \Phi \mid \text{there} \\ \text{is at least one} \\ \text{input in the} \\ \text{access pattern} \\ \text{on the last} \\ \text{atom of } \phi_j}} \Bigg(\prod_{\substack{\text{atom } p(\overline{Y}) \mid \\ p(\overline{Y}) \in body(\phi_j) \\ \text{with } p(\overline{Y}) \\ \text{without any} \\ \text{access pattern}}} \Big(i * \sum_{\substack{\phi_k \in \Phi \\ \text{such that} \\ p \text{is the} \\ \text{head relation} \\ \text{of } \phi_k \\ \text{and } r_k^{\alpha} \\ \text{the only} \\ \text{relation of} \\ \phi_k \text{ having} \\ \text{an access} \\ \text{pattern}}} |r_k^{\alpha}(\mathcal{D}_0)| \Big) \Bigg) + \sum_{\substack{\phi_j \in \Phi \mid \text{there} \\ \text{is no no input} \\ \text{in the access} \\ \text{pattern} \\ \text{on the last} \\ \text{atom of } \phi_j}} 1 \Bigg)$$

Thus:

$$|Accesses(\Phi, \mathcal{D}_0)| \leq$$

$$n_0 * \Big(\sum_{\substack{\phi_j \in \Phi \mid \text{there is at least one input in the access pattern on the last atom of } \phi_j}} \Big(\prod_{\substack{\text{atom } p(\overline{Y}) \mid \\ p(\overline{Y}) \in body(\phi_j) \\ \text{with } p(\overline{Y}) \\ \text{without any access pattern}}} (n_0 * \sum_{\substack{\phi_k \in \Phi \\ \text{such that } p \text{is the head relation of } \phi_k \\ \text{and } r_k^\alpha \\ \text{the only relation of } \phi_k \text{ having an access pattern}}} |r_k^\alpha(\mathcal{D}_0)|) \Big) + \sum_{\substack{\phi_j \in \Phi \mid \text{there is no no input in the access pattern on the last atom of } \phi_j}} 1 \Big)$$

So:

$$|Accesses(\Phi, \mathcal{D}_0)| \leq \mathcal{O}\Big(\sum_{\substack{\phi_j \in \Phi \mid \text{there is at least one input in the access pattern on the last atom of } \phi_j}} \Big(\prod_{\substack{\text{atom } p(\overline{Y}) \mid \\ p(\overline{Y}) \in body(\phi_j) \\ \text{with } p(\overline{Y}) \\ \text{without any access pattern}}} \Big(\sum_{\substack{\phi_k \in \Phi \\ \text{such that } p \text{is the head relation of } \phi_k \\ \text{and } r_k^\alpha \\ \text{the only relation of } \phi_k \text{ having an access pattern}}} |r_k^\alpha(\mathcal{D}_0)| \Big) \Big) \Big)$$

□

6.2.7 Discussion

The bound on the number of accesses given in lemma 6.2.22 applies on $Datalog^{\alpha_{Last}}$ queries which follow properties $(P1)$ and $(P2)$. This kind of queries encompasses the outputs of the inverse-rules algorithm and of algorithm 6.1. Precisely, we showed it encompasses the outputs of the inverse-rules algorithm when its input query doesn't contain any functions. However, this results can be straightforwardly generalized to all inputs of the inverse-rules algorithm. Indeed, what is missing to obtain this result is a proof that the immediate consequence operator, defined for $Datalog^{\alpha_{Last}}$ queries, still has a fixpoint when the query has functions, with the restriction that functions can only appear in heads of non recursive $\mathcal{CQ}^{\alpha_{Last}}$ rules. Remember, in the inverse-rules algorithm, that functions can only appear after reversing view definitions, which are $\mathcal{CQ}$ queries. So they cannot appear in recursive rules. Since the operational semantics of $Datalog^{\alpha_{Last}}$ queries defines a naive evaluation method (given in algorithm 6.2) which is clearly bottom-up, then we can use the same argument as the one given in the proof of lemma 7 in [Duschka et al., 2000]. This argument says that this query structure implies that terms having an infinitely number of nested functions cannot be generated during a bottom-up evaluation. That's why all previous results can still be used when functions are present in heads of non-recursive $\mathcal{CQ}^{\alpha_{Last}}$ rules.

Proof of lemma 6.2.22 gives a method specifying how to use results given in 6.2.20 to a particuliar configuration of a $Datalog^{\alpha_{Last}}$ query to obtain a precise bound on the

number of accesses implied by this query. In the case of lemma 6.2.22, the particuliar configuration is expressed through properties $(P1)$ and $(P2)$. It means that, if we keep the same naive evaluation algorithm, any processing that aims at optimizing the number of implied accesses by transforming the initial $Datalog^{\alpha_{Last}}$ query (provided that the results of both evaluations is the same), any such processing may be given a precise bound on the number of these accesses (in the style of the bound guiven in lemma 6.2.22). And then, two such bounds will make easier the comparison between the initial query and the optimized one, from which it will be possible to checked whether the optimization algorithm is useful or not. So we think this bound may an interesting tool to help the study of new algorithms to optimize (ie. reduce) the number of implied accesses during $Datalog^{\alpha_{Last}}$ query evaluation by trasforming the initial query obtained after the inverse-rules algorithm.

The last remark on the bound on the number of accesses given in lemma 6.2.22 is related to the number of iterations until fixpoint during evaluation. This number is hidden inside the bound (ie. considered as a constant) despite it is tightly linked to the number of implied accesses. Hiding this number of iterations does not mean that we consider it as a non key parameter to evaluate the number of accesses. Instead, it means that we assume this number will not vary too much between two different bounds obtained for example from an output of the inverse-rules algorithm and from the same output optimized by some heuristics. This is clearly a simplification but that is justified in that the number of iterations before fixpoint is a parameter that may be different from one query to its optimizations. So if we keep it in the bound, it will be of no use in order to compare optimization algorithms that could be applied to the output of the inverse-rules algorithm. In some sense, this number of iterations is a dynamic parameter since it depends on the query, while our bound is useful to statically study inverse-rules algorithm optimizations.

Chapter 7

Using DaWeS

In this chapter, we discuss the features offered by DaWeS to its users. We also discuss how web service API operations, global schema relations, enterprise record definitions and performance indicator queries are defined. We present the various qualitative and quantitative experiments done on DaWeS.

7.1 DaWeS Features

We discuss what DaWeS offers to its administrators and enterprise users.

7.1.1 Administrators

DaWeS provides the following features to the administrators:

1. Modeling the web service domains that involves adding the global schema relations and their corresponding attributes and adding the global schema constraints.
2. Adding new data types (to use for describing the local schema and global schema attributes).
3. Adding new web services belonging to a particular service domain and defining the relevant web service API operations and mapping them to the global schema using LAV mapping. Also for every web service API operation, identifying the schema of the response and the desired transformation.

4. Creating new record definitions (queries formulated over the global schema).
5. Creating new performance indicators (queries formulated using the record definitions).
6. Creating calibration tests for the record definitions and the performance indicator queries.
7. Monitoring the web service API operation changes.
8. Monitoring the calibration status for all the record definitions and performance measure queries.

The role of DaWeS administrator has been summarized in the Figure 7.1.

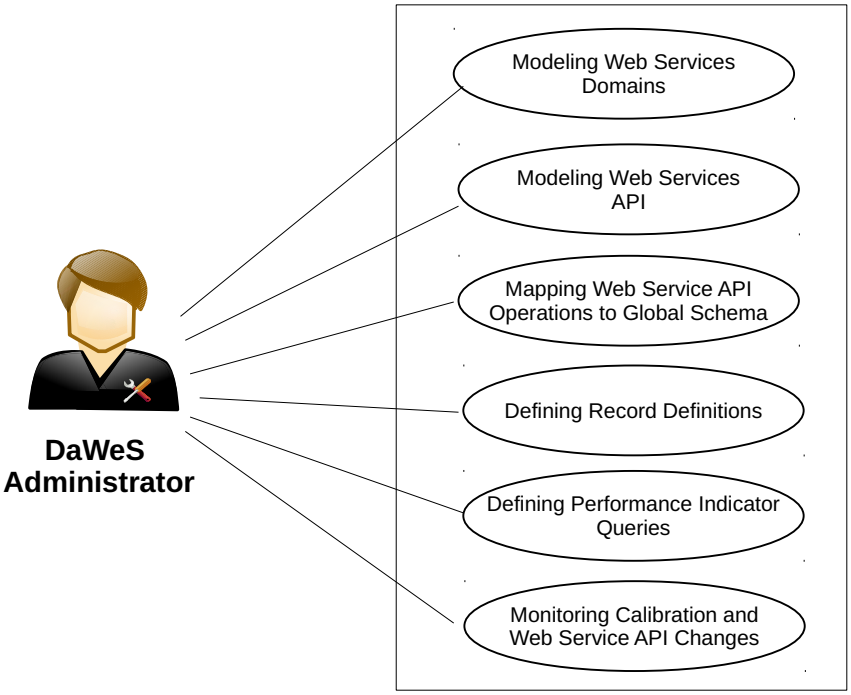

Figure 7.1: Role of DaWeS Administrator

7.1.2 Enterprise Users

Enterprise users has the following features provided by DaWeS:

1. Ability to search the supported web services
2. Specifying the authentication and authorization parameters for the desired web services
3. Ability to search the supported record definitions and performance measure queries
4. Choosing the relevant (or interesting) record definitions and performance measure queries
5. Creating new performance indicators (queries formulated using the record definitions)
6. Ability to view their own enterprise records and performance indicator results

The features available to DaWeS enterprise user has been summarized in the Figure 7.2.

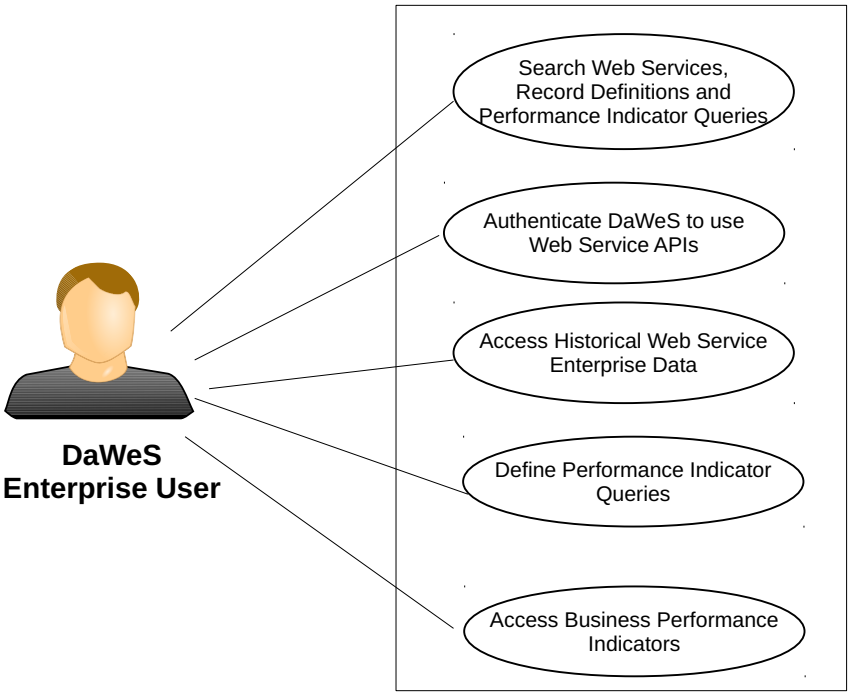

Figure 7.2: Role of DaWeS Enterprise User

7.2 Methodology for Modeling Web Service Interfaces in a Data Integration Approach

A data integration system involving web services is defined by the pentuple $\mathcal{W} = (\mathcal{D}, \mathcal{G}, \mathcal{L}, \mathcal{M}, \mathcal{C})$, where $\mathcal{D}$ is the set of domain (note to the reader, this is different from the domain or categories of web services) of various attributes used in the system, $\mathcal{G}$ is the set of global schema relations, $\mathcal{L}$ is the set of various web service API operations considered $\mathcal{M}$ is the set of source descriptions where every web service API operation is defined over the global schema relations using safe conjunctive query LAV mapping and $\mathcal{C}$ is the set of constraints on the global schema relations (currently only full and functional dependencies).

7.2.1 Global Schema Relations

When support for a new domain of web service(s) is required, the following guidelines are taken into account in order to identify the various relations and their corresponding attributes:

1. **Identification of the resources**: Various web services belonging to the studied domain are analyzed to identify the various resources managed by them. These resources are named and form the initial set of global schema relation names. The various information captured by each of these resources like name, identifier, creation date, status form their corresponding attributes.
2. **Client Requirements**: The client feedback is also considered in deciding the global schema relations. A client may be interested in various performance indicators. Once the client requirements are known, it is verified whether they can be computed from the existing global schema relations and the corresponding attributes or whether new relations would be required.
3. **Market Study of Performance Indicators**: Another source of information for the performance indicators in a particular domain is the associated market study. It also helps in figuring out the global schema relations.
4. **Resource/Attribute name synonyms(semantics)**: Various resources are referred to by different names in different web services. Referring these resources by these different names will ultimately result in having a large number of global schema relations. Therefore, while working with heterogeneous web services, we ensured whether the different resource names are in actual synonyms. If two or

more resource names are synonyms or semantically similar, one of these names is kept and it is taken into account during the LAV Mapping. For example, in the domain of project management web services, todo and task are synonyms used interchangeably in various web services. The same procedure is applied while considering the attribute names.

5. **Transient features**: Some of the information present in a resource representation are transient in nature. The most common attributes that shows highly volatile nature in its values is the status. Example project status, task status, campaign status. Other attributes like the resource names usually don't change their values. Therefore such transient information is considered as an attribute of a relation. Take for example, we have two different web service operations like *Get Open Tasks* and *Get Closed Tasks*. Instead of creating two different global schema relations *OpenTasks* and *Closed Tasks*, the status of the tasks is considered as an attribute of the relation *Tasks*. Identifying these transient information from the resource name and description is a little fuzzy. But it helps to significantly reduce the number of global schema relations.
6. **Resource identifier, source and creation time**: For every global schema relation, it is made sure that all of them have the following information: resource identifier, source and the creation time of the resource. These three information are quite useful since they are in-volatile information. They can be used together to uniquely identify a particular source of information. Source is used to identify the web service API and the associated version.

7.2.2 Local Schema Relations and LaV Mapping

When support for a new web service has to be added to the system, it is first ensured whether it belongs to one of the existing domains of the web services in DaWeS or share some common features to an existing domain. If there are some new features or if they don't belong to an existing domain, the first step involved is to define the global schema relations corresponding to this domain.

In the context of data integration, the relations in the data sources constitute the local schema. As we have seen throughout our discussion, the web service operations(request and response parameters) form our local schema relations. Thus for every web service, we will take a look into the relations (the different API operations) and their attributes (the input/request and the output/response parameters). Defining the local schema involves the following steps:

- All the relevant operations that can be mapped to the global schema relations are identified.
- The data type of all the attributes (both input and output attributes) are identified. New data types (if required) are created, especially for those input attributes that concern only a particular web service.

Note that for every relation name given in the table, our naming convention followed this manner:

1. Source (Web Service Name)
2. Version (Version of the Web Service API under consideration)
3. Resource Name (After considering the output of the web service API operation)

Example 7.2.1. *: Let us consider a data integration system* $\mathcal{W} = (\mathcal{D}, \mathcal{G}, \mathcal{L}, \mathcal{M}, \mathcal{C})$ *involving two web services: Basecamp[Basecamp, 2012] and Teamwork [Teamwork, 2012], two project management web services that manage various projects and tasks of the enterprises. Every projects has a number of task lists and a task list has a number of projects.*

The set of domains include $\mathcal{D}$ =

$$\{\textit{String, Project Status, Task Status, Basecamp Project Identifier, Basecamp Todo List Identifier, Basecamp Todo Identifier, Teamworkpm Project Identifier, Teamworkpm Task List Idenitifier, Teamworkpm Task Identifier}\}$$

The set of global schema relations $\mathcal{G}$ =

$$\{\ Project(pid, src, pname, pcdate, pstatus), TaskList(pid, src, tlid), Task(tlid, tid, src, tname, tcdate, tddate, tcmpdate, tstatus)\ \}$$

$\mathcal{G}$ *describes three relations:* $Project$, $TaskList$ *and* $Task$*; Project with attributes project identifier pid, source src, name pname, creation date pcdate and status pstatus,* $TaskList$ *with attributes project identifier pid, source src and task list identifier tlid and and* $Task$ *with attributes task list identifier tlid, task identifier tid, source src, task name tname, task creation date tcdate, task due date tddate, task completion date tcmpdate and task status tstatus.*

The set of global schema constraints $\mathcal{C}$ =

$$\{\ Project : pid, src \rightarrow pname, Project : pid, src \rightarrow pcdate,$$
$$Project : pid, src \rightarrow pstatus, TaskList : tlid, src \rightarrow pid,$$
$$Task : tid, src \rightarrow tlid, Task : tid, src \rightarrow tname,$$
$$Task : tid, src \rightarrow tcdate, Task : tid, src \rightarrow tddate,$$
$$Task : tid, src \rightarrow tcmpdate, Task : tid, src \rightarrow tstatus\ \}$$

Consider the functional dependency $Project : pid, src \rightarrow pname$. *If two tuples in Project have the same values pid and src, then the corresponding values for pname must be the same.*

We now look LAV mappings $\mathcal{M}$ *to define the some web service API operations* $\mathcal{L}$ *along with the access patterns (adornments) using the global schema relations. We explain the LAV mapping of* $Basecampv1Projects$ *and the present the LAV mapping of the rest without detailed explanation.*

Basecampv1Projects*: This operation gives the details of all the projects: project identifier pid, name pname, status pstatus and creation date pcdate. Figure 7.3 shows the screenshot of Basecamp Project obtained from their API documentation. The goal is to translate the information from API documentation on web pages to LAV mapping (along with XSD and XSLT).*

$$Basecampv1Projects^{oooo}(pid, pname, pstatus, pcdate) \leftarrow$$
$$Project(pid,' Basecamp\ API', pname, pcdate, pstatus). \tag{7.1}$$

The access pattern oooo in the above LAV mapping describes that the API operation $Basecampv1Projects$ *has no input attributes. Also note the source src attribute in Project corresponds to 'Basecamp API'. For all other attributes in* $Basecampv1Projects$*, there is one-to-one mapping with the attributes of Project.*

Basecampv1TodoLists*: This operation takes as input the project identifiers pid and gives the todo list identifiers tid present in the corresponding project identified by the project identifier.*

$$Basecampv1TodoLists^{io}(pid, tlid) \leftarrow$$
$$TaskList(pid,' Basecamp\ API', tlid). \tag{7.2}$$

```
[
  {
    "id": 605816632,
    "name": "BCX",
    "description": "The Next Generation",
    "updated_at": "2012-03-23T13:55:43-05:00",
    "url": "https://basecamp.com/999999999/api/v1/projects/605816632-bcx.json",
    "archived": false,
    "starred": true,
    "trashed": false,
    "is_client_project": false,
    "color": "3185c5"
  },
  {
    "id": 684146117,
    "name": "Nothing here!",
    "description": null,
    "updated_at": "2012-03-22T16:56:51-05:00",
    "url": "https://basecamp.com/999999999/api/v1/projects/684146117-nothing-here.json",
    "archived": false,
    "starred": false,
    "trashed": false,
    "is_client_project": true,
    "color": "3185c5"
  }
]
```


Figure 7.3: Screenshot of Basecamp Project

Basecampv1Tasks: *This operation takes as input the project identifier pid and todo list identifier tlid and gives as output the tasks present in the given todolist. It gives the following details like task identifer tid, name tname, status tstatus and creation date tcdate.*

$$Basecampv1Tasks^{iiooooo}(pid, tlid, tid, tname, tstatus, tcdate) \leftarrow TaskList(pid,' Basecamp\ API', tlid), Task(tlid, tid,' Basecamp\ API', tname, tcdate, tddate, tcmpdate, tstatus). \quad (7.3)$$

Basecampv1CompletedTasks: *This operation takes as input the project identifier pid and todo list identifier tlid and gives as output the completed tasks present in the given todolist. It gives the following details like task identifer tid, name tname, status tstatus and creation date tcdate.*

$$Basecampv1CompletedTasks^{iiooooo}(pid, tlid, tid, tname, tcdate, tcmpdate) \leftarrow TaskList(pid,' Basecamp\ API', tlid), Task(tlid, tid,' Basecamp\ API', tname, tcdate, tddate, tcmpdate,' Completed'). \quad (7.4)$$

TeamworkpmProjects*: This operation gives the details of all the projects: project identifier pid, name pname, status pstatus and creation date pcdate.*

$$TeamworkpmProjectAPI^{oooo}(pid, pname, pstatus, pcdate) \leftarrow Project(pid,' Teamworkpm\ API', pname, pcdate, pstatus). \quad (7.5)$$

TeamworkpmTodoLists*: This operation takes as input the project identifiers pid and gives the todo list identifiers tid present in the corresponding project identified by the project identifier.*

$$TeamworkpmTodoLists^{io}(pid, tlid) \leftarrow Project(pid,' Teamworkpm\ API', pname, pcdate, pstatus), TaskList(pid,' Teamworkpm\ API', tlid). \quad (7.6)$$

TeamworkpmTasks*: This operation takes as input the todo list identifier tlid and gives as output the tasks present in the given todolist. It gives the following details like task identifertid, name tname, project identifier pid, status tstatus, creation date tcdate and task due date tddate.*

$$TeamworkpmTasks^{iooooooo}(tlid, tid, tname, pid, tstatus, tcdate, tddate) \leftarrow Project(pid,' Teamworkpm\ API', pname, pcdate, pstatus), TaskList(pid,' Teamworkpm\ API', tlid), Task(tlid, tid,' Teamworkpm\ API', tname, tcdate, tddate, tcmpdate, tstatus). \quad (7.7)$$

7.2.2.1 XML Schema: Response Validation

When we make a web service API operation call, we get a response from the web service. We need to make sure that the response is in accordance to what we had expected. For this purpose, we use the schema of the expected response. In the example web

services that we considered, a vast majority of them did not expose any XML schema (XSD), but with a couple of examples given in the (human-readable) documentation, we created the XSD. Since XSD schema takes a considerable amount of space, we present here only one of the XSD schema. Below is the XSD schema of the operation Basecampv1Projects (Refer the LAVMapping C.10). We validate the four elements that we are interested in (id, name, created_at and archived). We have mentioned the expected types.

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="json">
    <xs:complexType>
     <xs:sequence>
      <xs:element name="array" maxOccurs="unbounded" minOccurs="0">
        <xs:complexType>
          <xs:all minOccurs="1" maxOccurs="1">
       <!-- Project Identifier -->
            <xs:element name="id">
              <xs:complexType>
                <xs:simpleContent>
                  <xs:extension base="xs:string">
                    <xs:attribute name="type" type="xs:string">
                    </xs:attribute>
                  </xs:extension>
                </xs:simpleContent>
              </xs:complexType>
            </xs:element>
       <!-- Project Name -->
            <xs:element name="name">
              <xs:complexType>
                <xs:simpleContent>
                  <xs:extension base="xs:string">
                    <xs:attribute name="type" type="xs:string">
                    </xs:attribute>
                  </xs:extension>
                </xs:simpleContent>
              </xs:complexType>
            </xs:element>
       <!-- Project Creation Date -->
            <xs:element name="created_at">
              <xs:complexType>
                <xs:simpleContent>
                  <xs:extension base="xs:dateTime">
                    <xs:attribute name="type" type="xs:string">
                    </xs:attribute>
                  </xs:extension>
                </xs:simpleContent>
              </xs:complexType>
            </xs:element>
       <!-- Project Status: Archived(true or false) -->
            <xs:element name="archived">
              <xs:complexType>
                <xs:simpleContent>
```

```
                <xs:extension base="xs:boolean">
                  <xs:attribute name="type" type="xs:string">
                  </xs:attribute>
                </xs:extension>
              </xs:simpleContent>
            </xs:complexType>
          </xs:element>
          <xs:any maxOccurs="unbounded" processContents="lax"/>
        </xs:all>
        <xs:attribute type="xs:string" name="class"/>
      </xs:complexType>
    </xs:element>
   </xs:sequence>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

7.2.2.2 XSLT: Response Transformation

Next, we use XSLT transformation to transform the result to a desired format. Again we continue the example with Basecampv1Projects (LAV Mapping C.10). Note how we extract only the desired information: project identifier, name, status and creation date. As we saw earlier that this operation returns the status of project as either archived or not (true or false), but our global schema relation requires that the project status is either 'Archived' or 'Active'. Therefore we make this data transformation for this element. Also take a look at how we extract the first ten characters from the date (YYYY-MM-DD). The transformed data consists of tuples of project, where each project detail is in a new line. Project details are comma separated (i.e., 'identifier', 'name', 'status', 'date' newline).

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:fn="http://www.w3.org/2005/xpath-functions" version="2.0">
  <xsl:output method="text" omit-xml-declaration="yes"
    cdata-section-elements="namelist"/>
  <xsl:template match="/">
    <xsl:for-each select="json/array">
      <xsl:value-of select="id"/>,<xsl:value-of select="name"/>,
      <!-- Project Status: Archived or Active -->
      <xsl:if test="archived = 'false'">Active</xsl:if>
      <xsl:if test="archived ='true'">Archived</xsl:if>,
      <!-- Project Creation date -->
      <xsl:value-of select="fn:substring(current()/created_at,1,10)"/>
      <xsl:text>&#xa;</xsl:text>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

7.2.3 Enterprise Record Definitions and Performance Indicator Queries

Performance Indicators or business performance measures are useful for every enterprise to track its growth and well-being. It is also useful to keep a track on various business units and their working. Business performance measures are usually an aggregation (or a summary) of large enterprise data. Take for example if various projects undertaken by an enterprise constitutes the enterprise data, examples of possible business performance measures include:

- Total number of projects
- Total number of completed projects
- Total number of pending projects
- Total number of late running projects
- Total number of projects completed before time
- Percentage of late running projects
- Average number of projects completed in a month

One of the key characteristic of the performance indicators is that they are calculated periodically. These are expressed in numbers with varying units of measures. Numbers can signify financial measures, count or a comparison. [Lawyer and Chowdhury, 2004] discusses several best practices in data warehousing particularly in relation to data warehouse attribution and keys and data warehouse loading. In particular they suggest not to select a single frequency (hourly, daily, monthly) for loading the data warehouse. During the design and development of DaWeS, we made sure that various queries have an associated frequency that decides when to fetch the relevant information from the web services. The frequencies included the most common ones: daily, weekly, fortnightly, monthly, quarterly half-yearly and yearly. We take a look on the various characteristics of performance indicators

1. *Frequency*: Different indicators are computed at various periods of time. For example Total number of new subscribers to a magazine in a month, day or a week. Some of the popular frequencies commonly used in businesses are hourly, daily, weekly, fortnightly, monthly, quarterly, yearly or a specific period (examples: last 10 days, between some specified, dates). Some enterprises, especially the internet service providers(ISPs) or large web services even go to much more smaller periods of times like computing the availability of their services every

minute (or second). Such real time monitoring of various business measures is also getting very common these days.

2. *Significance of the numerals*: What does a performance indicator signify. In most cases, the performance indicators are usually numerals (integers or floating point) or multiple numerals along with some associated informations. For example: Number of new subscriptions or unsubscription for an internet magazine or after an email marketing campaign, Average number of clicks registered on an email marketing campaign on a day, top five active forums. The significant measures include
   - Count (Financial Measure, Numerical Count)
   - Mean, Mode, Median, and Standard Deviation
   - Ratio, Percentage
   - 95 or 99 percentile
   - Ordinal (Position, Top 5 or 10)
3. *Granularity of Information*: One other interesting characteristic of performance indicator is to figure out the manner and amount of aggregation to be performed. Total number of pending projects in an enterprise with hundreds of employees is an important business metric for business executives controlling the overall functioning of various business units. What's more interesting for a manager is to track the number of pending projects in her team. Similarly, consider another example where a forum consists of various entries (comments), and every entry constitutes an activity. Different types of users require different levels of (granular) information. A manager of Customer support team would like to know the number of unanswered comments in the forums managed by the team, but the customer support team member will be more concerned about the number of unanswered comments directly assigned to him/her.
4. *State*: In most of the resource representations, state is an important data field. This state field is quite often used for the computation of performance indicator, often used as a criterion for the aggregation (as discussed above). Information of various states are computed at different points of time. A user before starting his work on a day would like to see the number of pending tasks assigned to him on a particular day. But at the end of a week/month, he prefers to see the total number of tickets solved by him. So the state of information (here tickets) is another important factor. Some examples of the states include
   - Solved Tickets
   - Unsolved Tickets
   - Answered Forums

- Unanswered Forums
- Active Projects
- Archived Projects

5. *Category of the Requester*: The indicators requested by different persons differs based on his/her work profile. A manager requests for a different set of indicators than an employee in his/her team. Following are some examples of performance indicators with the (possible) requester(s) given in the brackets

   - Total tickets overdue on him/her (Employee and Manager)
   - Total tickets overdue on the team (Manager)
   - Total tickets solved by the team in the quarter (Manager)

6. *Choice of Performance Indicators*: When users choose an indicator, various reasons are behind their choice. A choice comes along with the associated information of the person who made the choice (job profile, team, company, domain of work). This information in turn is useful to suggest identifiers when a new company/organization looks for identifiers in a category (or a service). Some of the reasons behind the choice of a performance indicator are already discussed above (Refer Category of the Requester, State and Granularity).

7.3 Experiments

In this section, we describe the various experiments done on DaWeS, both qualitative and quantitative and report our observations.

7.3.1 Experiment Description

We took into consideration 35 different operations (refer section C.2) of 12 different web services belonging to three different domains(project management, email marketing and support/helpdesk). We considered 17 record definitions (refer section C.3) formulated over the global schema relations (defined in the section C.1). We make use of the record definitions to define our performance indicators (refer section C.4 to see the 20 such performance indicators) and then use the enterprise records to compute the performance indicators.

7.3.1.1 Setup

The details of the *system*, *database* and the *cache* are given in the Table 7.3.1.1.

Table 7.1: DaWeS Experiments: Setup

DaWeS Settings	
Ehcache	Version 2.6.5
Cache Eviction Policy	Least Frequently Used
Allowed Number of entries (on Heap)	10000
Allowed Number of entries (on Disk)	100
Time to Live	600 seconds

For real-life scenario, we considered the test data as described in the section C.5 and entered them into the respective web services using web browsers. We use the DaWeS command line options to run our tests (refer section D.3). We use (INFO Logging mode) logging to log the time taken by various functions in DaWeS. Twelve different web services and the associated number of operation considered by us is given in the Table 7.2.

Table 7.2: Total Web Service API Operations considered for the Tests

Project Management Services		
1.	Basecamp	4
2.	Liquid Planner	2
3.	Teambox	3
4.	Zoho Projects	1
Email Marketing Services		
5.	MailChimp	3
6.	Campaign Monitor	5
7.	iContact	1
Support (Helpdesk) Services		
8.	Zendesk	5
9.	Desk	4
10.	Zoho Support	1
11.	Uservoice	2
12.	FreshDesk	4

We created 100 test organizations (one of them was created manually and all others made use of the same authentication parameters) to simulate a multi-enterprise environment. We considered that all the 100 test organizations have the same information of authentication parameters, the interested record definitions and the interested performance indicator queries. Thus we created 100 (homogeneous) enterprises for performing our tests. For a given organization, we performed the following steps:

1. For every web services considered in the Table 7.2, do the following
   (a) Create an account in the respective web service
   (b) Add test data (with the details given in section C.5)
   (c) Add the organization authentication parameters and other required information (like the subdomain in the URL) for the web service to DaWeS.
2. Add the information (to DaWeS) that this organization is interested in all the record definitions considered in the Table C.4.
3. Add the information (to DaWeS) that this organization is interested in all the performance indicators considered in the Table C.5.

7.3.1.2 Qualitative Tests

We have tested the following features: *Scheduler*, *Calibration*, *Search*, *Fetching of Records from web services* and *Performance Indicator computation*. Our focus is on the query evaluation (fetching of records from the web services) and the performance indicator computation.

For the qualitative tests, we test the capability of our system to handle different types of queries and performance indicators. We make sure that various web service operations are handled properly, especially the operations requiring pagination and those that need to follow some specific operation invocation sequence (both are handled by the inverse rules algorithm using the domain rules). The web service API operations that are considered have one or more of the following characteristics:

1. Requires no input arguments
2. Requires one or more input arguments
3. Requires pagination

We also tested the system with the following authentication mechanisms:

1. Basic HTTP Authentication (username/password combination)
2. Basic HTTP Authentication (username/password combination and additional information from the users like the (sub)domain name of their service)
3. OAuth v1.0 Authentication (with additional information from the user like the subdomain)

The test data are detailed in the section C.5. They have the following features

1. Multiple resources with same names (or titles) within a single web service. Example: Two tasks having the same name in Basecamp.
2. Multiple resources with same names (or titles) among different web services. Example: Two tasks in Basecamp and Teamworkpm having the same name.
3. Multiple resources with different names (or titles) within a single web services or among different web services

We use the following types of queries (defined over the global schema relations)

1. Conjunctive Query
2. Union of Conjunctive Query
3. (Recursive) Datalog Query

We also test our system for different types of performance indicators

1. Total/Count of entries
2. Average
3. Percentage
4. Tuples (useful to create charts)

7.3.1.3 Quantitative Tests

In this set of tests, we check the time required to compute the record definitions (queries formulated over the global schema relations) and performance indicator queries. Especially we focus our attention to the queries on the global schema relations and calculate the different times for query throughput (throughput here refers to the time taken to compute a query):

1. *Query throughput (one query at a time)*

2. *Query throughput, when run with the scheduler*
3. *Query throughput, when response is available in the database (from previous computation), so using the cache*

As mentioned above, we are interested mostly in the two main features of DaWeS: query evaluation (fetching of records from web services) and performance indicator computation. Therefore for the query evaluation, we take into consideration the following times (averaged for 100 organizations):

1. (Hibernate) Setup Time for query evaluation
2. Performing various checks (whether the record definition identifier is valid)
3. Get the latest record for the organization (if any)
4. Get the latest Calibration Status
5. Setting up the IRIS Datalog engine (Example: loading the domain rules, inverse rules)
6. Query Evaluation (or Execution) time. This includes
   (a) Reading the data from the cache
   (b) Making Web service API operation calls
   (c) Response Validation
   (d) Response Transformation
7. Performing Chase (Heuristics handling incomplete information) and Frame Response
8. Saving the result to the database

For performance indicator computation, we are interested in computing the following times (averaged for 100 organizations):

1. (Hibernate) Setup Time
2. Performing various checks (whether the performance indicator identifier is valid)
3. Get the latest performance indicator value for the organization (if any)
4. Get the latest Calibration Status
5. Check whether the dependent records of organization are valid
6. Execute (SQL query) and Frame Response
7. Saving the result to the database

7.3.2 Experiment Results

7.3.2.1 Qualitative Tests

We use some screen-shots to show the responses of DaWeS when various operations (section D.3) are performed.

Figure 7.4 shows DaWeS command line options.

```
$ ./dawes -h
Usage:dawes
-i : Initialize the System(first test run with default examples)
-c r|pi -o org [-n Identifier] : Compute all(a specified) record or perf
ormance indicator
-s ws|r|pi -p string: Search a web service, record or performance indica
tor
-l all|ws|r|pi [-n Identifier] : Run unit tests and Calibrate all(a spec
ified) web service, record or performance indicator
-r : Run the Scheduler
-h : Help
-v : Version

$
```

Figure 7.4: DaWeS: Help

1. **Search**: Figure 7.5 shows how a web service is searched using DaWeS. In the given example, we search for project management web services, but simply use project as the search pattern. The results shows all the project management web services currently supported with DaWeS (along with the current ratings).

```
$ ./dawes -s ws -p "project"
Searching..
Identifier                                                    Name     Rating
================================================================================
     80153                                                 Basecamp        5.0
     80154                                              LiquidPlanner      5.0
     80156                                               ZohoProjects      5.0
     80155                                                 Teamworkpm      5.0
$
```

Figure 7.5: DaWeS: Searching a web service

Figure 7.6 shows how a record definition is searched using DaWeS. In the given example, we search for the record definitions which captures the notion of all

forums. DaWeS not only returns the record definitions that contains the pattern *all forums*, but also some other record definitions that may be useful to the enterprise, belonging to the same domain or are somewhat related.

```
$ ./dawes -s r -p "all forums"
Searching..
Identifier                                                          Name     Rating
================================================================================
     89160                                                Daily All Forums        5.0
     89162                                              Daily New Tickets        5.0
     89159                                               Daily New Forums        5.0
     89164                                          Daily Closed Tickets        5.0
     89163                                            Daily Open Tickets        5.0
     89161                                               Daily New Topics        5.0
$
```

Figure 7.6: DaWeS: Searching a record definition

Figure 7.7 shows how DaWeS is used to search for interesting performance indicators. In the given example, we search for performance indicators that deal with *campaigns*. DaWeS returns the relevant performance indicators and their current ratings.

```
$ ./dawes -s pi -p "campaign"
Searching..
Identifier                                                          Name     Rating
================================================================================
     75158                                  Total Monthly New Campaigns        5.0
     75160                                 Monthly Forwards of Campaign        5.0
     75161                                  Monthly Bounces of Campaign        5.0
     75159                               Monthly Click Throughs of Campaign        5.0
$
```

Figure 7.7: DaWeS: Searching a Performance Indicator

2. **Calibration**: Figure 7.8 shows how DaWeS is used to calibrate all the record definitions and the latest calibration status is stored in the database (which will be used before the fetching of record; the record will be fetched if the status is passed)
   Figure 7.9 shows how DaWeS is used to calibrate all the performance indicator computation and the latest calibration status is stored in the database (which will be used before the performance indicator computation the computation will be performed if the status is passed)
3. **Fetching of Records from web services**: Figure 7.10 shows how DaWeS is used to fetch and form a record after the query evaluation. In this case, the

```
$ ./dawes -l r
Performing the Calibration of All LCRecords
Completed and Updated the Database
$
```

Figure 7.8: DaWeS: Performing Calibration of Record Definitions

```
$ ./dawes -l pi
Performing the Calibration of All Performance Indicator
Completed and Updated the Database
$
```

Figure 7.9: DaWeS: Performing Calibration of Performance Indicators

concerned record is *Daily New Projects*. The example shows the capability of DaWeS to evaluate a particular query of an enterprise.

```
=======================================================
                    Query Results
=======================================================
('a1be77490ecffc4d51b9834c23a5bf186755b1c1035e80e0', 'ZohoProjects API', 'Client communication', 'Active')
('83103', 'Teamworkpm API', 'Documentation', 'Active')
('83105', 'Teamworkpm API', 'System Requirement and Analysis', 'Archived')
('4539776', 'Basecamp API', 'Design', 'Active')
('4539803', 'Basecamp API', 'Development', 'Active')
('12261354', 'LiquidPlanner API', 'Integrated Testing', 'Active')
('12261408', 'LiquidPlanner API', 'Unit Testing', 'Active')
=======================================================
```

Figure 7.10: DaWeS: Fetching a Record: Daily New Projects

Figure 7.11 shows how DaWeS is used to evaluate the record definition *Daily Open Tasks*.

Figure 7.12 shows how DaWeS is used to evaluate the record definition *Daily Open Tickets*.

4. **Performance Indicator Computation**: Figure 7.13 shows how DaWeS is used to compute the performance indicator of an enterprise. In this case, the concerned performance indicator is *Monthly Forwards of Campaign*

   Figure 7.14 shows how DaWeS is used to compute the performance indicator *Total high priority tickets Registered in a month*.

   Figure 7.15 shows how DaWeS is used to compute the performance indicator: Percentage of high priority tickets Registered in a month

```
=========================================================
                      Query Results
=========================================================
('12261382', 'LiquidPlanner API', 'Testing the calibration')
('12261384', 'LiquidPlanner API', 'Scheduler')
('12261411', 'LiquidPlanner API', 'Answer Builder')
('12261418', 'LiquidPlanner API', 'Generic Wrapper')
('12261421', 'LiquidPlanner API', 'Performance Indicator Computation')
('74468190', 'Basecamp API', 'Design Answer Builder')
('74468205', 'Basecamp API', 'Design Generic Wrapper')
('74468219', 'Basecamp API', 'Scheduler')
('74468236', 'Basecamp API', 'Performance Indicator Computation')
('74467977', 'Basecamp API', 'Relations for storing the web service descriptions')
('74468046', 'Basecamp API', 'Relations related to organization and their data')
('74468432', 'Basecamp API', 'Develop Answer Builder')
('74468446', 'Basecamp API', 'Develop Generic Wrapper')
('74468467', 'Basecamp API', 'Scheduler')
('74468518', 'Basecamp API', 'Performance Indicator Computation')
('1870634', 'Teamworkpm API', 'Document Generic Wrapper')
('1870635', 'Teamworkpm API', 'Scheduler')
('1870604', 'Teamworkpm API', 'Describe the relations for storing the record definitions')
('1870605', 'Teamworkpm API', 'Describe the relations for storing the performance indicators')
=========================================================
```

Figure 7.11: DaWeS: Fetching a Record: Daily Open Tasks

```
=====================================================
                    Query Results
=====================================================
('1000256021', 'Freshdesk API', 'Add new subsections Setup and Results in the section Experiments', 'High')
('1000256017', 'Freshdesk API', 'Add a new section called Experiments in the documentation', 'High')
('1000256013', 'Freshdesk API', 'Correct typos in the documentation', 'Medium')
('38077000000050007', 'ZohoSupport API', 'Add support for a new record definition', 'Low')
('38077000000050005', 'ZohoSupport API', 'Add support for a new performance indicator', 'High')
('38077000000050003', 'ZohoSupport API', 'Add support for a new web service', 'Medium')
('38077000000050001', 'ZohoSupport API', 'Request for a new feature', 'Low')
('Make a call to the web service provider', 'Desk v2 API', 'Make a call to the web service provider', 'Low')
('Make a request to open an account', 'Desk v2 API', 'Make a request to open an account', 'High')
('Request the web service provider for a feature request', 'Desk v2 API', 'Request the web service provider for a f
eature request', 'Low')
('14746217', 'Uservoice v1 API', 'Add a new domain of web services', 'Low')
('14746200', 'Uservoice v1 API', 'Study the new web service feature request', 'Low')
('6', 'Zendesk v2 API', 'Add support for a new web service', 'Medium')
('7', 'Zendesk v2 API', 'Add support for a new performance indicator', 'High')
('8', 'Zendesk v2 API', 'Add support for a new record definition', 'Medium')
=====================================================
```

Figure 7.12: DaWeS: Fetching a Record: Daily Open Tickets

5. **Scheduler**: Figure 7.16 shows how DaWeS is used to run the scheduler. As mentioned before and as seen in the figure, DaWeS performs the calibration and then computation of record definitions and performance indicators for every enterprise.

7.3.2.2 Quantitative Tests

We continue with tests performed using 100 test organizations.

```
==========================================================
                      Query Results
==========================================================
('Beta Release Campaign', '0')
('Alpha Release Campaign', '0')
('Product Release CampaignMonitor Campaign', '0')
==========================================================
```

Figure 7.13: DaWeS: Computing a Performance Indicator: Monthly Forwards of Campaign

```
==========================================================
                      Query Results
==========================================================
('5')
==========================================================
```

Figure 7.14: DaWeS: Computing a Performance Indicator: Total high priority tickets Registered in a month

```
==========================================================
                      Query Results
==========================================================
('38.461538461538461538461538461538461538461538461538')
==========================================================
```

Figure 7.15: DaWeS: Computing a Performance Indicator: Percentage of high priority tickets Registered in a month

```
$ ./dawes -r
Running the scheduler
Organization..72700
Running Unit Tests
Fri Nov 29 01:53:21 CET 2013
Running Record Calibration Tests
Computation of Records
==========================================================
                      Query Results
```

Figure 7.16: DaWeS: Running the scheduler

We will present here once again (refer section C.2 for details) all the record definitions.

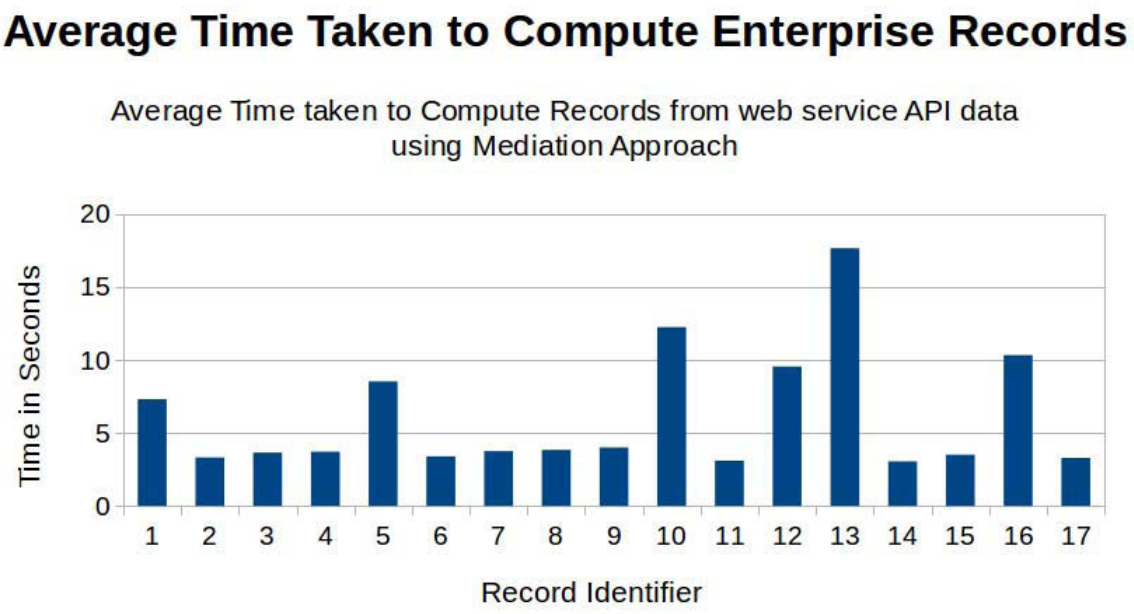

Figure 7.17: DaWeS: Scheduler performing Query Evaluation using web service API Operations

1. Daily New Projects
2. Daily Active Projects
3. Daily OnHold Projects
4. Daily OnHold or Archived Projects
5. Daily New Tasks
6. Daily Open Tasks
7. Daily Closed Tasks
8. Daily TodoLists
9. Daily Same Status Projects
10. Daily New Forums
11. Daily All Forums
12. Daily New Topics
13. Daily New Tickets
14. Daily Open Tickets
15. Daily Closed Tickets
16. Daily New Campaigns
17. Daily Campaign Statistics

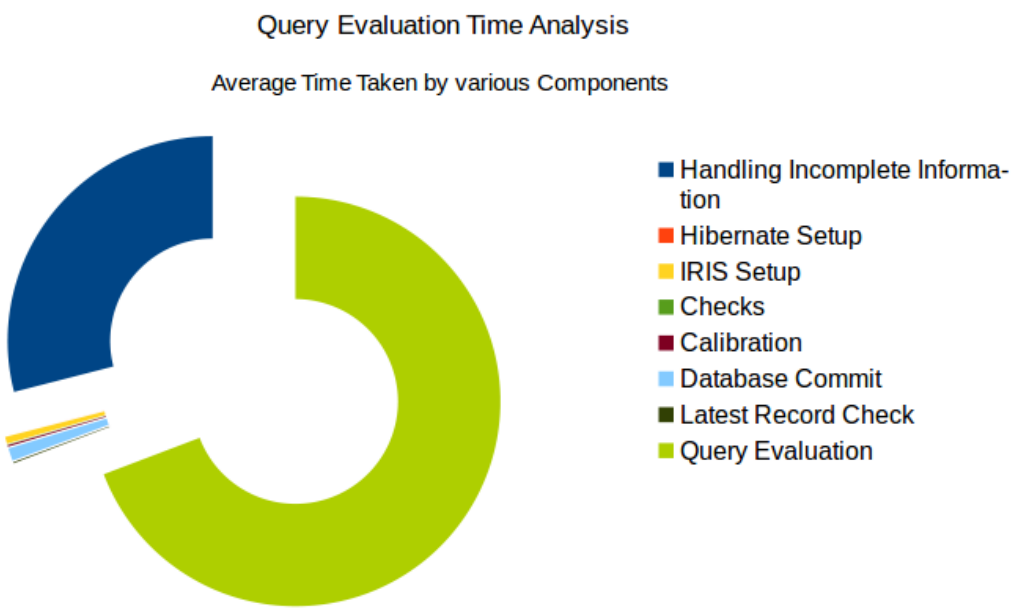

Figure 7.18: DaWeS: Analysis of Query Evaluation of All Queries

Figure 7.17 shows the average time taken for 100 organizations to compute the records using the web service API operations with the help of a scheduler. Recall that the scheduler computes the records of an enterprise (thus making the most of the cache). Note the interesting spikes for the following records

- 1. Daily New Projects
- 5. Daily New Tasks
- 10. Daily New Forums
- 12. Daily New Topics
- 13. Daily New Tickets
- 16. Daily New Campaigns

This is because the data is not available locally in the cache and the (time-expensive) web service API operations have to be made. For the subsequent operations (depending on the data already fetched and cache), the computation time is low.

Figure 7.18 shows the query analysis. Query evaluation takes significant amount of time followed by making use of the chase algorithm. We see in Figure 7.19 the time taken by the wrapper during query evaluation. It is clear that the most amount of time is spent making API operation calls and getting the response.

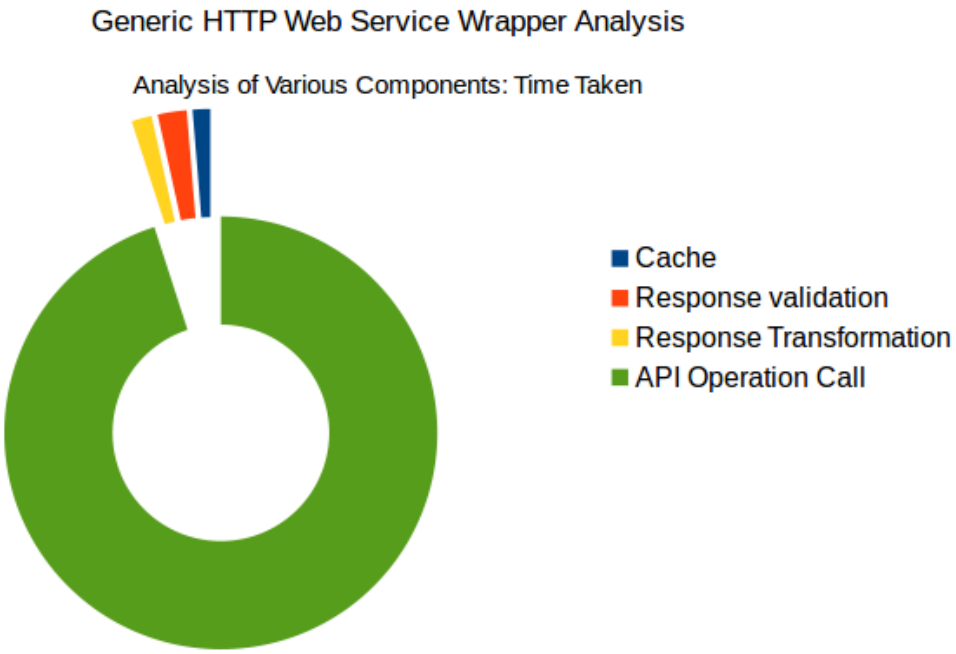

Figure 7.19: DaWeS: Analysis of Various Components of Generic HTTP Web Service Wrapper

Next we see the computation of performance indicators using the records. The performance indicators considered are given below (refer section C.4 for details).

1. Total Monthly New Projects
2. Total Monthly Active Projects
3. Total Monthly OnHold Projects
4. Total Monthly Completed Tasks
5. Average Tasks Completed Daily in a month
6. Total Monthly New Tasks
7. Total Todo Lists
8. Percentage of tasks completed to tasks created in a day
9. Total Monthly New Campaigns
10. Monthly Click Throughs of Campaign
11. Monthly Forwards of Campaign
12. Monthly Bounces of Campaign
13. Total Monthly Solved Tickets
14. Daily Average Resolution Time

15. Total New Tickets Registered in a month
16. Total New Forums Registered in a month
17. All Forums in a month
18. Total New Topics Registered in a month
19. Total High Priority Tickets Registered in a month
20. Percentage of High Priority Tickets Registered in a month

Figure 7.20 shows the average time taken to compute performance indicators. Recall that the performance indicators make use of the underlying (Oracle) DBMS for the query evaluation. We check that evaluating performance indicators is two orders of magnitude quicker than fetching the data. This was expected because of fetching the data from web services require network communication times to achieve calls whereas performance indicators are computed by highly optimized query evaluation techniques in Oracle using the enterprise records stored in the database.

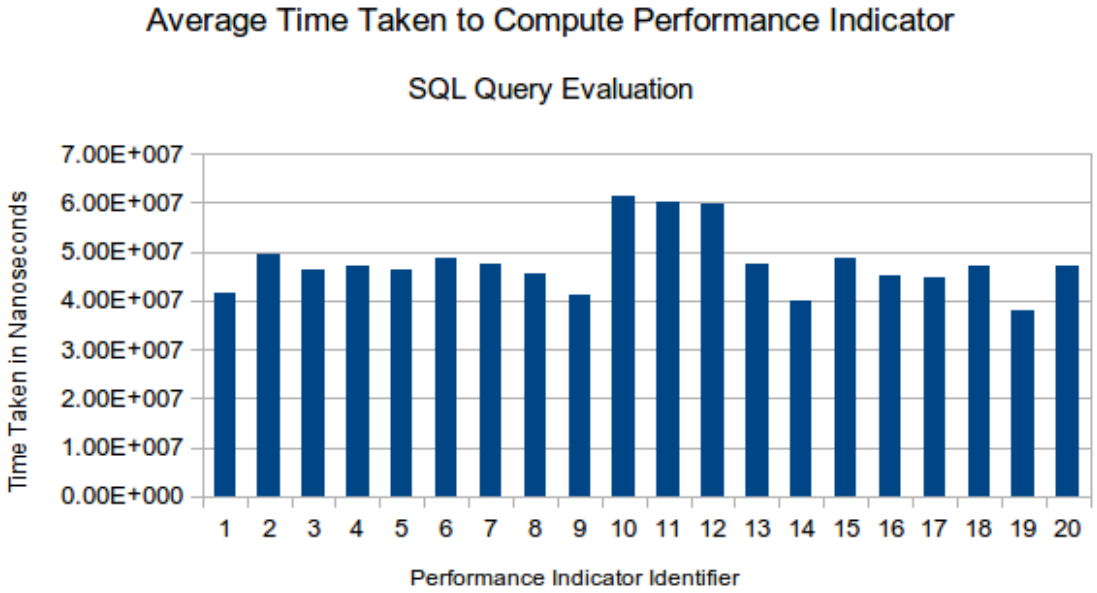

Figure 7.20: DaWeS: Performance Indicator Computation

Current tests: We are currently working on creating a thread pool for the scheduler so that record computation of multiple enterprises can be done in parallel. We are also currently working on a test suite to test DaWeS with thousands of API operations randomly generated. We want to see how much load (number of web service operations) can a single instance of DaWeS handle without performance degradation.

Chapter 8

Future Works and Conclusion

There are several scopes for extending the capabilities of DaWeS. From experimental perspective, DaWeS needs to be industrially tested with additional domains and the associated web services. Secondly, one of the reasons for choosing two tables for storing the enterprise data and performance indicators was to easily migrate these data to a columnar storage based cloud infrastructure. This possibility of migration needs to be tested with columnar stores along with the associated changes required in performance indicator queries. A visual interface for creating performance indicator queries is another interesting direction that can further reduce the burden of working with the complex SQL queries.

From a research perspective, we presented a heuristics for handling the incomplete information in DaWeS using the inverse rules algorithm. The precise characterisation of the heuristics is useful especially to understand the query composition capabilities of DaWeS. Currently, the inverse rules algorithm handles full and functional dependencies and we utilize this capability to specify the primary keys of the global schema relations. Datalog has caught the attention of the research community and various extensions [Calì et al., 2009b, 2012, 2011; Calvanese et al., 2007; Gottlob et al., 2012] to datalog have been proposed especially to deal with the various constraints used in the ontology. Ontological queries can be rewritten into small non-recursive datalog programs [Gottlob and Schwentick, 2011]. Support for additional constraints in the global schema is very interesting. Another possible extension is by making use of certain aggregate queries with datalog [Shkapsky et al., 2013].

We showed that even with basic web technologies, we can reduce a significant amount of coding effort and automate the feeding of data warehouse with web ser-

vice data. A further study is required that can compare what is done with the basic languages and what could be done with advanced languages (eg., automating the generation of the LAV mapping). [Yerneni et al., 1999] considered the case of search forms usually found in web sites and studied additional adornments like unspecifiable and optional. Certain web service API operations also have these arguments; extending DaWeS taking into account these is another interesting direction.

Columnar storage is gaining a lot of traction during the past few years. This gradual transition from relational databases to columnar storage opens up a lot of new research problems particularly handling of the OLAP CUBE queries, previously suggested with the star schema. Any such advancement can be used to enhance the capabilities of DaWeS to handle CUBE queries. DaWeS doesn't follow multidimensional modeling. But in some cases, the customers of DaWeS may be interested to have hierarchies in some dimensions to perform cubing and OLAP. This needs to be further explored.

From the perspective of business requirements, a more automated approach to handling the errors is needed. Currently we periodically perform the calibration of records and performance indicators to ensure their accurate computation. Calibration of records help DaWeS adminstrators to get quick information when any web service API undergoes any (unannounced) change. But (unannounced) API change is one of the possible failures.

We discussed about various failures that can occur while making API operation calls in section 5.2.2.10. Web service APIs don't follow any standard HTTP errors that can be used to distinguish between various categories of errors. Distinguishing various categories of errors is important so as to take appropriate action. Take for example, if network or the service provider is temporally down, it signifies that the API operation call can be repeated after a period of time. If the operation call returns an error 'resource not found', the operation call need not be repeated and it must be reported and diagnosed why such an operation call was made since API calls are expensive. DaWeS needs to be further enhanced with automated error handling mechanism such as with the help of an ontology of API failures and associated measures towards correcting errors.

The growing use of web services among the enterprises cannot be undermined. DaWeS is a need of the hour for the enterprises using multiple web services for their day to day transactions. DaWeS provides them an integrated view of their enterprise data spread across multiple web services and also enables them to create and compute interesting business performance measures. DaWeS also points to a new generation

of web services that make interesting and promising use of the enterprise data spread across other web services.

DaWeS aims towards building a scalable and adaptable service for integration with ever-evolving web services supporting not only the ease of use in building interesting business measures but also enabling the continuity of enterprise data in the event of web service shutdowns or switching. With DaWeS, we show how mediation as ETL is effective in integration with web services. Given its scalable and adaptable nature, it can be easily adapted by small and medium scale enterprises considering the minimum amount of coding effort while handling web service API. Generic HTTP wrapper shows that even with the lack of machine readable interfaces, it is possible to work with numerous web services (and the problem is not as complex as the wrappers for textual sources, legacy databases where a new wrapper is created for every new data source).

Given the interest of cloud computing within the industry, our current approach of storing the complete web service API enterprise data on just two big tables may be adapted to the columnar storage based cloud infrastructure.

Our proposed upper bound on the number of API operation calls can be used to compare various optimization algorithms to reduce the number of (expensive) web service API operation calls. Finally the proposed heuristics handling incomplete information shows another interesting feature of inverse rules algorithm towards handling incomplete answers and query composition.

DaWeS also bridges the current gap between the industry and research community considering the lack of usage of machine readable syntactic and semantic standards like WSDL, WADL, hRESTS, SAWSDL. We believe that this situation will change in the coming years and DaWeS can be further automated like automated generation of LAV mapping. DaWeS also presents to the research community a platform for testing various recent scientific advancements in the fields of web services, data integration and data warehousing.

Bibliography

Abiteboul, S., Benjelloun, O., Manolescu, I., Milo, T., and Weber, R. (2002). Active xml: Peer-to-peer data and web services integration. In *VLDB*, pages 1087–1090. Morgan Kaufmann. (Cited on page 43.)

Abiteboul, S. and Duschka, O. M. (1998). Complexity of answering queries using materialized views. In Mendelzon, A. O. and Paredaens, J., editors, *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 1-3, 1998, Seattle, Washington, USA*, pages 254–263. ACM Press. (Cited on page 47.)

Abiteboul, S., Hull, R., and Vianu, V. (1995). *Foundations of Databases*. Addison-Wesley. (Cited on pages 47, 55, 57, 99 and 114.)

Abiteboul, S., Kanellakis, P. C., and Grahne, G. (1991). On the representation and querying of sets of possible worlds. *Theor. Comput. Sci.*, 78(1):158–187. (Cited on page 47.)

Abiteboul, S., Manolescu, I., Rigaux, P., Rousset, M.-C., Senellart, P., et al. (2012). *Web data management*. Cambridge University Press. (Cited on page 47.)

Adali, S., Candan, K. S., Papakonstantinou, Y., and Subrahmanian, V. S. (1996). Query caching and optimization in distributed mediator systems. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, SIGMOD '96, pages 137–146, New York, NY, USA. ACM. (Cited on page 46.)

Afrati, F. N. and Kiourtis, N. (2008). Query answering using views in the presence of dependencies. In *NTII*, pages 8–11. (Cited on page 49.)

Afrati, F. N., Li, C., and Mitra, P. (2002). Answering queries using views with arithmetic comparisons. In Popa, L., Abiteboul, S., and Kolaitis, P. G., editors, *PODS*, pages 209–220. ACM. (Cited on page 49.)

Agarwal, S., Keller, A. M., Wiederhold, G., and Saraswat, K. (1995). Flexible relation: An approach for integrating data from multiple, possibly inconsistent databases. In *In IEEE International Conference on Data Engineering*, pages 495–504. (Cited on page 49.)

Akkiraju, R., Farrell, J., Miller, J. A., Nagarajan, M., Sheth, A., and Verma, K. (2005). Web service semantics-wsdl-s. (Cited on page 52.)

Aweber (2012). Aweber. `http://www.aweber.com`. (Cited on page 184.)

Axis, A. (2012). Apache web services project. `hhttp://axis.apache.org/axis2/java/core/`. (Cited on page 42.)

Bai, Q., Hong, J., McTear, M. F., and Wang, H. (2006). A bucket-based approach to query rewriting using views in the presence of inclusion dependencies. *Journal of Research and Practice in Information Technology*, 38(3):251–266. (Cited on page 49.)

Barhamgi, M., Benslimane, D., and Ouksel, A. M. (2008). Composing and optimizing data providing web services. In Huai, J., Chen, R., Hon, H.-W., Liu, Y., Ma, W.-Y., Tomkins, A., and Zhang, X., editors, *WWW*, pages 1141–1142. ACM. (Cited on pages 43, 44 and 45.)

Baril, X., Bellahsène, Z., et al. (2003). Designing and managing an xml warehouse. *XML Data Management: Native XML and XML-Enabled Database Systems, Addison Wesley*, pages 455–473. (Cited on pages 39 and 43.)

Basecamp (2012). Basecamp. `http://www.basecamp.com`. (Cited on pages 136 and 184.)

Bayardo, R. J., Jr., Bohrer, W., Brice, R., Cichocki, A., Fowler, J., Helal, A., Kashyap, V., Ksiezyk, V. K. T., Martin, G., Nodine, M., Rashid, M., Rusinkiewicz, M., Shea, R., Unnikrishnan, C., Unruh, A., and Woelk, D. (1997). Infosleuth: Agent-based semantic integration of information in open and dynamic environments. (Cited on page 46.)

Benatallah, B., Casati, F., Grigori, D., Nezhad, H. R. M., and Toumani, F. (2005). Developing adapters for web services integration. In Pastor, O. and e Cunha, J. F., editors, *CAiSE*, volume 3520 of *Lecture Notes in Computer Science*, pages 415–429. Springer. (Cited on page 42.)

Benslimane, D., Dustdar, S., and Sheth, A. P. (2008). Services mashups: The new generation of web applications. *IEEE Internet Computing*, 12(5). (Cited on pages 22, 44 and 45.)

Berglund, A., Boag, S., Chamberlin, D., Fernandez, M. F., Kay, M., Robie, J., and Siméon, J. (2007). Xml path language (xpath) 2.0. *W3C recommendation*, 23. (Cited on page 52.)

Berners-Lee, T., Fielding, R., and Frystyk, H. (1996). Hypertext transfer protocol–http/1.0. (Cited on page 21.)

Berti-Equille, L. and Moussouni, F. (2005). Quality-aware integration and warehousing of genomic data. In Naumann, F., Gertz, M., and Madnick, S. E., editors, *IQ*. MIT. (Cited on page 40.)

Bhowmick, S., Madria, S., Ng, W.-K., and Lim, E.-P. (1999). Web warehousing: Design and issues. In Kambayashi, Y., Lee, D.-L., Lim, E.-p., Mohania, M., and Masunaga, Y., editors, *Advances in Database Technologies*, volume 1552 of *Lecture Notes in Computer Science*, pages 93–104. Springer Berlin Heidelberg. (Cited on page 41.)

Bhowmick, S. S., Madria, S. K., and Ng, W. K. (2003). Representation of web data in a web warehouse. *Comput. J.*, 46(3):229–262. (Cited on page 41.)

Bosak, J., Bray, T., Connolly, D., Maler, E., Nicol, G., Sperberg-McQueen, C., Wood, L., and Clark, J. (1998). W3c xml specification dtd. (Cited on page 38.)

Box, D., Ehnebuske, D., Kakivaya, G., Layman, A., Mendelsohn, N., Nielsen, H. F., Thatte, S., and Winer, D. (1999). Soap: Simple object access protocol. *HTTP Working Group Internet Draft*. (Cited on pages 42 and 51.)

Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., and Yergeau, F. (1997). Extensible markup language (xml). *World Wide Web Journal*, 2(4):27–66. (Cited on pages 52 and 189.)

Burstein, M. H., Hobbs, J. R., Lassila, O., Martin, D., McDermott, D. V., McIlraith, S. A., Narayanan, S., Paolucci, M., Payne, T. R., and Sycara, K. P. (2002). Daml-s: Web service description for the semantic web. In *Proceedings of ISWC*, pages 348–363. (Cited on pages 32 and 44.)

Calì, A., Calvanese, D., Giacomo, G. D., and Lenzerini, M. (2002). On the role of integrity constraints in data integration. *IEEE Data Eng. Bull.*, 25(3):39–45. (Cited on page 49.)

Calì, A., Calvanese, D., Giacomo, G. D., and Lenzerini, M. (2004). Data integration under integrity constraints. *Inf. Syst.*, 29(2):147–163. (Cited on page 49.)

Calì, A., Calvanese, D., and Lenzerini, M. (2013). Data integration under integrity constraints. In Jr., J. A. B., Krogstie, J., Pastor, O., Pernici, B., Rolland, C., and Sølvberg, A., editors, *Seminal Contributions to Information Systems Engineering*, pages 335–352. Springer. (Cited on page 49.)

Calì, A., Calvanese, D., and Martinenghi, D. (2009a). Dynamic query optimization under access limitations and dependencies. *J. UCS*, 15(1):33–62. (Cited on pages 49 and 59.)

Calì, A., Gottlob, G., and Lukasiewicz, T. (2009b). Tractable query answering over ontologies with datalog+/-. In Grau, B. C., Horrocks, I., Motik, B., and Sattler, U., editors, *Description Logics*, volume 477 of *CEUR Workshop Proceedings*. CEUR-WS.org. (Cited on page 159.)

Calì, A., Gottlob, G., and Lukasiewicz, T. (2012). A general datalog-based framework for tractable query answering over ontologies. *J. Web Sem.*, 14:57–83. (Cited on page 159.)

Calì, A., Gottlob, G., and Pieris, A. (2011). New expressive languages for ontological query answering. In Burgard, W. and Roth, D., editors, *AAAI*. AAAI Press. (Cited on page 159.)

Calì, A., Lembo, D., and Rosati, R. (2003). Query rewriting and answering under constraints in data integration systems. In *IJCAI*, pages 16–21. (Cited on page 49.)

Calì, A. and Martinenghi, D. (2008). Querying data under access limitations. In *ICDE*, pages 50–59. (Cited on page 49.)

Calvanese, D., De Giacomo, G., Lenzerini, M., Nardi, D., and Rosati, R. (1998). Source integration in data warehousing. In *DEXA Workshop*, pages 192–197. (Cited on page 40.)

Calvanese, D., De Giacomo, G., Lenzerini, M., Nardi, D., and Rosati, R. (1999). A principled approach to data integration and reconciliation in data warehousing. In Gatziu, S., Jeusfeld, M. A., Staudt, M., and Vassiliou, Y., editors, *DMDW*, volume 19 of *CEUR Workshop Proceedings*, page 16. CEUR-WS.org. (Cited on page 40.)

Calvanese, D., De Giacomo, G., Lenzerini, M., Nardi, D., and Rosati, R. (2001a). Data integration in data warehousing. *Int. J. Cooperative Inf. Syst.*, 10(3):237–271. (Cited on page 40.)

Calvanese, D., Giacomo, G. D., Lembo, D., Lenzerini, M., and Rosati, R. (2007). Tractable reasoning and efficient query answering in description logics: The *dl-lite* family. *J. Autom. Reasoning*, 39(3):385–429. (Cited on page 159.)

Calvanese, D., Giacomo, G. D., and Lenzerini, M. (2000a). Answering queries using views over description logics knowledge bases. In Kautz, H. A. and Porter, B. W., editors, *AAAI/IAAI*, pages 386–391. AAAI Press / The MIT Press. (Cited on page 49.)

Calvanese, D., Giacomo, G. D., Lenzerini, M., and Vardi, M. Y. (2000b). What is view-based query rewriting? In Bouzeghoub, M., Klusch, M., Nutt, W., and Sattler, U., editors, *Proceedings of the 7th International Workshop on Knowledge Representation meets Databases (KRDB 2000), Berlin, Germany, August 21, 2000*, volume 29 of *CEUR Workshop Proceedings*, pages 17–27. CEUR-WS.org. (Cited on page 47.)

Calvanese, D., Lembo, D., and Lenzerini, M. (2001b). Survey on methods for query rewriting and query answering using views. (Cited on page 47.)

Campaign Monitor (2012). Campaign monitor. `http://www.campaignmonitor.com`. (Cited on page 184.)

Cautis, B., Deutsch, A., and Onose, N. (2011a). Querying data sources that export infinite sets of views. *Theory Comput. Syst.*, 49(2):367–428. (Cited on page 49.)

Cautis, B., Deutsch, A., Onose, N., and Vassalos, V. (2011b). Querying xml data sources that export very large sets of views. *ACM Trans. Database Syst.*, 36(1):5. (Cited on page 46.)

Chandra, A. K. and Merlin, P. M. (1977). Optimal implementation of conjunctive queries in relational data bases. In *STOC*, pages 77–90. (Cited on page 48.)

Chaudhuri, S. and Dayal, U. (1997). An overview of data warehousing and olap technology. *SIGMOD Record*, 26(1):65–74. (Cited on page 60.)

Chawathe, S., Garcia-Molina, H., Hammer, J., Ireland, K., Papakonstantinou, Y., Ullman, J., and Widom, J. (1994). The tsimmis project: Integration of heterogeneous information sources. In *In Proceedings of IPSJ Conference*, pages 7–18. (Cited on page 46.)

Cheng, K., Kambayashi, Y., Lee, S. T., and Mohania, M. K. (2000). Functions of a web warehouse. In *Kyoto International Conference on Digital Libraries*, pages 372–379. (Cited on page 41.)

Christiansen, H. and Martinenghi, D. (2004). Simplification of integrity constraints for data integration. In Seipel, D. and Torres, J. M. T., editors, *FoIKS*, volume 2942 of *Lecture Notes in Computer Science*, pages 31–48. Springer. (Cited on page 49.)

ConstantContact (2012). Constantcontact. `http://www.constantcontact.com`. (Cited on page 184.)

Crockford, D. (2006). Json. (Cited on pages 52 and 189.)

Desk (2012). Desk. `http://www.desk.com`. (Cited on page 184.)

Deutsch, A., Ludäscher, B., and Nash, A. (2007). Rewriting queries using views with access patterns under integrity constraints. *Theor. Comput. Sci.*, 371(3):200–226. (Cited on pages 50 and 64.)

Duschka, O. M. and Genesereth, M. R. (1997). Answering recursive queries using views. In *PODS*, pages 109–116. (Cited on pages 46, 47 and 61.)

Duschka, O. M., Genesereth, M. R., and Levy, A. Y. (2000). Recursive query plans for data integration. *J. Log. Program.*, 43(1):49–73. (Cited on pages 47, 49, 64, 67, 112 and 129.)

Eclipse (2012). Eclipse ide. `http://www.eclipse.org/eclipse/development/readme_eclipse_3.8.html`. (Cited on page 98.)

ehcache (2012). Ehcache. `http://ehcache.org/`. (Cited on page 98.)

Fernández, M., Malhotra, A., Marsh, J., Nagy, M., and Walsh, N. (2002). Xquery 1.0 and xpath 2.0 data model. *W3C working draft*, 15. (Cited on pages 52 and 71.)

Fernandez, P. (2013). Scribe. `https://github.com/fernandezpablo85/scribe-java`. (Cited on page 98.)

Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and Berners-Lee, T. (1999). Hypertext transfer protocol–http/1.1. (Cited on pages 21 and 190.)

Fielding, R. T. (2000). Architectural styles and the design of network-based software architectures. (Cited on pages 31, 51, 181 and 186.)

Freshdesk (2012). Freshdesk. `http://www.freshdesk.com`. (Cited on page 184.)

Friedman, M., Levy, A. Y., and Millstein, T. D. (1999). Navigational plans for data integration. In Hendler, J. and Subramanian, D., editors, *AAAI/IAAI*, pages 67–73. AAAI Press / The MIT Press. (Cited on page 46.)

Gao, S., Sperberg-McQueen, C. M., Thompson, H. S., Mendelsohn, N., Beech, D., and Maloney, M. (2009). W3c xml schema definition language (xsd) 1.1 part 1: Structures. *W3C Candidate Recommendation*, 30. (Cited on pages 52, 71 and 98.)

Gardarin, G., Mensch, A., Tuyet Dang-Ngoc, T., and Smit, L. (2002). Integrating heterogeneous data sources with xml and xquery. In *Database and Expert Systems Applications, 2002. Proceedings. 13th International Workshop on*, pages 839–844. IEEE. (Cited on page 46.)

Genesereth, M. R., Keller, A. M., and Duschka, O. M. (1997). Infomaster: An information integration system. In Peckham, J., editor, *SIGMOD Conference*, pages 539–542. ACM Press. (Cited on page 46.)

Goessner, S. (2006). Jsont - transforming json. `http://goessner.net/articles/jsont/`. (Cited on page 53.)

Goessner, S. (2007). Jsonpath - xpath for json. `http://goessner.net/articles/JsonPath/`. (Cited on page 53.)

Gomadam, K., Ranabahu, A., and Sheth, A. (2010). Sa-rest: Semantic annotation of web resources. *W3C Member Submission*, 5. (Cited on page 52.)

Gottlob, G., Orsi, G., and Pieris, A. (2011a). Ontological queries: Rewriting and optimization. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 2–13. IEEE. (Cited on page 49.)

Gottlob, G., Orsi, G., and Pieris, A. (2011b). Ontological query answering via rewriting. In Eder, J., Bieliková, M., and Tjoa, A. M., editors, *ADBIS*, volume 6909 of *Lecture Notes in Computer Science*, pages 1–18. Springer. (Cited on page 49.)

Gottlob, G., Orsi, G., Pieris, A., and Simkus, M. (2012). Datalog and its extensions for semantic web databases. In Eiter, T. and Krennwallner, T., editors, *Reasoning Web*, volume 7487 of *Lecture Notes in Computer Science*, pages 54–77. Springer. (Cited on page 159.)

Gottlob, G. and Schwentick, T. (2011). Rewriting ontological queries into small nonrecursive datalog programs. *Description Logics*, 745. (Cited on page 159.)

Gottlob, G. and Schwentick, T. (2012). Rewriting ontological queries into small nonrecursive datalog programs. In Brewka, G., Eiter, T., and McIlraith, S. A., editors, *KR*. AAAI Press. (Cited on page 47.)

Grahne, G. (1989). Horn tables - an efficient tool for handling incomplete information in databases. In [Silberschatz, 1989], pages 75–82. (Cited on page 47.)

Grahne, G. and Kiricenko, V. (2002). Obtaining more answers from information integration systems. In *Proc. Fifth International Workshop on the Web and Databases (WebDB '02)*, pages 67–76. (Cited on page 48.)

Grahne, G. and Kiricenko, V. (2003). Partial answers in information integration systems. In *WIDM*, pages 98–101. (Cited on page 48.)

Grahne, G. and Kiricenko, V. (2004). Towards an algebraic theory of information integration. *Inf. Comput.*, 194(2):79–100. (Cited on pages 48, 55, 103 and 104.)

Grahne, G. and Mendelzon, A. O. (1999). Tableau techniques for querying information sources through global schemas. In *In Proc. of the 7th Int. Conf. on Database Theory (ICDT'99), volume 1540 of Lecture Notes in Computer Science*, pages 332–347. Springer. (Cited on page 48.)

Gryz, J. (1999). Query rewriting using views in the presence of functional and inclusion dependencies. *Inf. Syst.*, 24(7):597–612. (Cited on page 49.)

Guérin, E., Marquet, G., Burgun, A., Loréal, O., Berti-Equille, L., Leser, U., and Moussouni, F. (2005). Integrating and warehousing liver gene expression data and related biomedical resources in gedaw. In Ludäscher, B. and Raschid, L., editors, *DILS*, volume 3615 of *Lecture Notes in Computer Science*, pages 158–174. Springer. (Cited on page 40.)

Hadley, M. J. (2006). Web application description language (wadl). Technical report, Mountain View, CA, USA. (Cited on pages 22 and 52.)

Halevy, A. Y. (2000). Theory of answering queries using views. *SIGMOD Record*, 29(4):40–47. (Cited on page 49.)

Halevy, A. Y. (2001). Answering queries using views: A survey. *VLDB J.*, 10(4):270–294. (Cited on pages 46, 49 and 61.)

Hammer, J., Garcia-Molina, H., Widom, J., Labio, W., and Zhuge, Y. (1995). The stanford data warehousing project. *IEEE Data Eng. Bull.*, 18(2):41–48. (Cited on pages 40 and 41.)

Hansen, M., Madnick, S. E., and Siegel, M. (2002). Data integration using web services. In Lacroix, Z., editor, *DIWeb*, pages 3–16. University of Toronto Press. (Cited on page 42.)

He, H. (2003). What is service-oriented architecture. *Publicação eletrônica em*, 30. (Cited on page 50.)

hibernate (2012). Hibernate. `www.hibernate.org`. (Cited on page 98.)

iContact (2012). icontact. `http://www.icontact.com`. (Cited on page 184.)

Imieliński, T. and Lipski, Jr., W. (1984). Incomplete information in relational databases. *J. ACM*, 31(4):761–791. (Cited on page 47.)

Inmon, W. H. (1992). *Building the Data Warehouse*. John Wiley & Sons, Inc., New York, NY, USA. (Cited on pages 22, 38, 39, 43 and 181.)

IRIS (2008). *Integrated Rule Inference System - API and User Guide*. (Cited on pages 98, 99 and 293.)

Java SE (2012). Java se development kit 7, update 25 (jdk 7u25). `www.oracle.com/technetwork/java/javase/7u25-relnotes-1955741.html`. (Cited on page 98.)

javax.xml (2012). javax.xml: Core xml constants and functionality from the xml specifications. `http://docs.oracle.com/javase/1.5.0/docs/api/javax/xml/package-summary.html`. (Cited on page 99.)

jersey (2012). Jersey-restful web services in java. `https://jersey.java.net/`. (Cited on page 42.)

Json-lib (2012). Json-lib. `http://json-lib.sourceforge.net/`. (Cited on page 99.)

Kambhampati, S., Lambrecht, E., Nambiar, U., Nie, Z., and Gnanaprakasam, S. (2004). Optimizing recursive information gathering plans in emerac. *J. Intell. Inf. Syst.*, 22(2):119–153. (Cited on page 46.)

Karakasidis, A., Vassiliadis, P., and Pitoura, E. (2005). Etl queues for active data warehousing. In Berti-Equille, L., Batini, C., and Srivastava, D., editors, *IQIS*, pages 28–39. ACM. (Cited on page 41.)

Kay, M. et al. (2007). Xsl transformations (xslt) version 2.0. *W3C Recommendation*, 23. (Cited on pages 52, 71 and 98.)

Kayako (2012). Kayako. `http://www.kayako.com`. (Cited on page 184.)

Kimball, R. (1996). *The Data Warehouse Toolkit: Practical Techniques for Building Dimensional Data Warehouses*. John Wiley. (Cited on pages 22, 38, 39 and 43.)

Kimball, R. and Merz, R. (2000). *The data webhouse toolkit: building the web-enabled data warehouse*. Wiley New York. (Cited on page 41.)

Kirk, T., Levy, A. Y., Sagiv, Y., and Srivastava, D. (1995). The information manifold. In *In Proceedings of the AAAI 1995 Spring Symp. on Information Gathering from Heterogeneous, Distributed Enviroments*, pages 85–91. (Cited on page 46.)

Knoblock, C. A., Minton, S., Ambite, J. L., Ashish, N., Muslea, I., Philpot, A., and Tejada, S. (2001). The ariadne approach to web-based information integration. *Int. J. Cooperative Inf. Syst.*, 10(1-2):145–169. (Cited on page 46.)

Koch, C. (2004). Query rewriting with symmetric constraints. *AI Commun.*, 17(2):41–55. (Cited on page 46.)

Kopecký, J., Gomadam, K., and Vitvar, T. (2008). hrests: An html microformat for describing restful web services. In *Proceedings of the 2008 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology - Volume 01*, WI-IAT '08, pages 619–625, Washington, DC, USA. IEEE Computer Society. (Cited on pages 32, 52 and 185.)

Kopecký, J., Vitvar, T., Bournez, C., and Farrell, J. (2007). Sawsdl: Semantic annotations for wsdl and xml schema. *IEEE Internet Computing*, 11(6):60–67. (Cited on pages 32 and 52.)

Kwok, C. T. and Weld, D. S. (1996). Planning to gather information. In *In Proceedings of the AAAI Thirteenth National Conference on Artificial Intelligence*, pages 32–39. (Cited on page 49.)

Lausen, H., Polleres, A., and Roman, D. (2005). Web service modeling ontology (wsmo). *W3C Member Submission*, 3. (Cited on page 52.)

Lawyer, J. and Chowdhury, S. (2004). Best practices in data warehousing to support business initiatives and needs. In *HICSS*. (Cited on page 142.)

Levy, A. Y. (1999). Logic-based techniques in data integration. (Cited on page 47.)

Levy, A. Y., Rajaraman, A., and Ordille, J. J. (1996a). Query-answering algorithms for information agents. In Clancey, W. J. and Weld, D. S., editors, *AAAI/IAAI, Vol. 1*, pages 40–47. AAAI Press / The MIT Press. (Cited on page 47.)

Levy, A. Y., Rajaraman, A., and Ordille, J. J. (1996b). Querying heterogeneous information sources using source descriptions. In Vijayaraman, T. M., Buchmann, A. P., Mohan, C., and Sarda, N. L., editors, *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*, pages 251–262. Morgan Kaufmann. (Cited on page 49.)

Li, C. (2003). Computing complete answers to queries in the presence of limited access patterns. *VLDB J.*, 12(3):211–227. (Cited on page 50.)

Li, C., Bawa, M., and Ullman, J. D. (2001). Minimizing view sets without losing query-answering power. In den Bussche, J. V. and Vianu, V., editors, *ICDT*, volume 1973 of *Lecture Notes in Computer Science*, pages 99–113. Springer. (Cited on page 48.)

Li, C. and Chang, E. Y. (2000). Query planning with limited source capabilities. In *ICDE*, pages 401–412. (Cited on pages 49 and 50.)

Li, C. and Chang, E. Y. (2001a). Answering queries with useful bindings. *ACM Trans. Database Syst.*, 26(3):313–343. (Cited on page 64.)

Li, C. and Chang, E. Y. (2001b). On answering queries in the presence of limited access patterns. In *ICDT*, pages 219–233. (Cited on page 50.)

Li, Y. and Heflin, J. (2010). Using reformulation trees to optimize queries over distributed heterogeneous sources. In Patel-Schneider, P. F., Pan, Y., Hitzler, P., Mika, P., Zhang, L., Pan, J. Z., Horrocks, I., and Glimm, B., editors, *International Semantic Web Conference (1)*, volume 6496 of *Lecture Notes in Computer Science*, pages 502–517. Springer. (Cited on page 50.)

LiquidPlanner (2012). Liquidplanner. `http://www.liquidplanner.com`. (Cited on page 184.)

LittleCrowd (2014). Littlecrowd. `http://www.littlecrowd.com/`. (Cited on page 27.)

Lopes, C. T. and David, G. (2006). Higher education web information system usage analysis with a data webhouse. In Gavrilova, M. L., Gervasi, O., Kumar, V., Tan, C. J. K., Taniar, D., Laganà, A., Mun, Y., and Choo, H., editors, *ICCSA (4)*, volume 3983 of *Lecture Notes in Computer Science*, pages 78–87. Springer. (Cited on page 41.)

Mailchimp (2012). Mailchimp. `http://www.mailchimp.com`. (Cited on page 184.)

Maleshkova, M., Pedrinaci, C., Domingue, J., Alvaro, G., and Martinez, I. (2010). Using semantics for automating the authentication of web apis. In *Proceedings of the 9th International Semantic Web Conference on The Semantic Web - Volume Part I*, ISWC'10, pages 534–549, Berlin, Heidelberg. Springer-Verlag. (Cited on page 52.)

Manolescu, I., Florescu, D., and Kossmann, D. (2001). Answering xml queries over heterogeneous data sources. In Mouaddib, N., editor, *BDA*. (Cited on page 46.)

Martin, D., Burstein, M., Hobbs, J., Lassila, O., McDermott, D., McIlraith, S., Narayanan, S., Paolucci, M., Parsia, B., Payne, T., et al. (2009). Owl-s: Semantic markup for web services (2004). *Latest version available from: http://www.ai.sri.com/daml/services/owl-s/1.2*. (Cited on page 52.)

Martin, D., Paolucci, M., and Wagner, M. (2007). Bringing semantic annotations to web services: Owl-s from the sawsdl perspective. In *Proceedings of the 6th International The Semantic Web and 2Nd Asian Conference on Asian Semantic Web Conference*, ISWC'07/ASWC'07, pages 340–352, Berlin, Heidelberg. Springer-Verlag. (Cited on page 32.)

McKee, B., Ehnebuske, D., and Rogers, D. (2001). Uddi version 2.0 api specification. *UDDI. org*. (Cited on page 42.)

Microformats (2012). Microformats. `http://microformats.org/`. (Cited on page 52.)

Mignet, L., Abiteboul, S., Ailleret, S., Amann, B., Marian, A., and Preda, M. (2000). Acquiring xml pages for a webhouse. In Doucet, A., editor, *BDA*. (Cited on page 39.)

Millstein, T. D., Halevy, A. Y., and Friedman, M. (2003). Query containment for data integration systems. *J. Comput. Syst. Sci.*, 66(1):20–39. (Cited on page 49.)

Mitra, P. (2001). An algorithm for answering queries efficiently using views. In *ADC*, pages 99–106. (Cited on pages 47 and 49.)

Nash, A. and Ludäscher, B. (2004a). Processing first-order queries under limited access patterns. In *PODS*, pages 307–318. (Cited on page 50.)

Nash, A. and Ludäscher, B. (2004b). Processing unions of conjunctive queries with negation under limited access patterns. In Bertino, E., Christodoulakis, S., Plexousakis, D., Christophides, V., Koubarakis, M., Böhm, K., and Ferrari, E., editors, *EDBT*, volume 2992 of *Lecture Notes in Computer Science*, pages 422–440. Springer. (Cited on page 50.)

Oracle Database (2012). Oracle database express edition. `http://www.oracle.com/technetwork/database/database-technologies/express-edition/index.html`. (Cited on page 98.)

Papakonstantinou, Y., Borkar, V. R., Orgiyan, M., Stathatos, K., Suta, L., Vassalos, V., and Velikhov, P. (2003). Xml queries and algebra in the enosys integration platform. *Data Knowl. Eng.*, 44(3):299–322. (Cited on page 46.)

Papakonstantinou, Y. and Vassalos, V. (2002). Architecture and implementation of an xquery-based information integration platform. *IEEE Data Eng. Bull.*, 25(1):18–26. (Cited on page 46.)

Pérez, J. M., Llavori, R. B., Aramburu, M. J., and Pedersen, T. B. (2008). Integrating data warehouses with web data: A survey. *IEEE Trans. Knowl. Data Eng.*, 20(7):940–955. (Cited on page 43.)

Pottinger, R. and Halevy, A. (2001). Minicon: A scalable algorithm for answering queries using views. *The VLDB Journal*, 10(2-3):182–198. (Cited on pages 47 and 61.)

Pottinger, R. and Levy, A. Y. (2000). A scalable algorithm for answering queries using views. In El Abbadi, A., Brodie, M. L., Chakravarthy, S., Dayal, U., Kamel, N., Schlageter, G., and Whang, K.-Y., editors, *VLDB*, pages 484–495. Morgan Kaufmann. (Cited on page 47.)

ProgrammableWeb, M. (2012). Apis, and the web as platform. *URL: http://www.programmableweb. com*. (Cited on pages 32 and 51.)

Raggett, D., Le Hors, A., Jacobs, I., et al. (1999). Html 4.01 specification. *W3C recommendation*, 24. (Cited on page 189.)

Rahm, E. and Bernstein, P. A. (2001). A survey of approaches to automatic schema matching. *VLDB J.*, 10(4):334–350. (Cited on page 47.)

Rajaraman, A., Sagiv, Y., and Ullman, J. D. (1995). Answering queries using templates with binding patterns (extended abstract). In *Proceedings of the fourteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, PODS '95, pages 105–112. (Cited on pages 49, 64, 101, 111 and 112.)

Rootsystem (2014). Rootsystem. `http://www.rootsystem.fr`. (Cited on page 27.)

Roth, M. T. and Schwarz, P. M. (1997). Don't scrap it, wrap it! a wrapper architecture for legacy data sources. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, VLDB '97, pages 266–275, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc. (Cited on pages 22, 42 and 70.)

Salem, R., Boussaïd, O., and Darmont, J. (2013). Active xml-based web data integration. *Information Systems Frontiers*, 15(3):371–398. (Cited on page 43.)

Salem, R., Darmont, J., and Boussaïd, O. (2010). Toward active xml data warehousing. In Ben-Abdallah, H. and Feki, J., editors, *EDA*, volume B-6 of *RNTI*, pages 65–79. Cépaduès. (Cited on page 43.)

Samuel, J. (2014). Towards a data warehouse fed with web services. In Presutti, V., d'Amato, C., Gandon, F., d'Aquin, M., Staab, S., and Tordai, A., editors, *ESWC*, volume 8465 of *Lecture Notes in Computer Science*, pages 874–884. Springer. (Cited on pages 1 and 23.)

Samuel, J. and Rey, C. (2014). Dawes: Data warehouse fed with web services. In *INFORSID*. (Cited on pages 1 and 23.)

Samuel, J., Rey, C., Martin, F., and Peyron, L. (2014). Mediation-based web services fed data warehouse. In Bimonte, S., d'Orazio, L., and Negre, E., editors, *EDA*, RNTI. Hermann. (Cited on pages 1 and 23.)

Sheth, A. (2003). Semantic web process lifecycle: role of semantics in annotation, discovery, composition and orchestration. (Cited on page 52.)

Sheth, A. P., Gomadam, K., and Ranabahu, A. (2008). Semantics enhanced services: Meteor-s, sawsdl and sa-rest. *IEEE Data Eng. Bull.*, 31(3):8–12. (Cited on page 52.)

Sheth, A. P. and Larson, J. A. (1990). Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Comput. Surv.*, 22(3):183–236. (Cited on page 44.)

Shkapsky, A., Zeng, K., and Zaniolo, C. (2013). Graph queries in a next-generation datalog system. *PVLDB*, 6(12):1258–1261. (Cited on page 159.)

Silberschatz, A., editor (1989). *Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, March 29-31, 1989, Philadelphia, Pennsylvania, USA*. ACM Press. (Cited on pages 170 and 177.)

SQL Developer (2012). Oracle sql developer. `http://www.oracle.com/technetwork/developer-tools/sql-developer/overview/index.html`. (Cited on page 98.)

Teambox (2012). Teambox. `http://www.teambox.com`. (Cited on page 184.)

Teamwork (2012). Teamwork. `http://www.teamworkpm.net`. (Cited on pages 136 and 184.)

Thakkar, S., Ambite, J. L., and Knoblock, C. A. (2004). A data integration approach to automatically composing and optimizing web services. In *In Proceedings of the ICAPS Workshop on Planning and Scheduling for Web and Grid Services*. (Cited on page 22.)

Thakkar, S., Ambite, J. L., and Knoblock, C. A. (2005). Composing, optimizing, and executing plans for bioinformatics web services. *The VLDB Journal*, 14(3):330–353. (Cited on page 50.)

Thakkar, S., Knoblock, C. A., and Ambite, J. L. (2003). A view integration approach to dynamic composition of web services. In *In Proceedings of 2003 ICAPS Workshop on Planning for Web Services*. (Cited on pages 43, 44 and 45.)

Trujillo, J. and Luján-Mora, S. (2003). A uml based approach for modeling etl processes in data warehouses. In Song, I.-Y., Liddle, S. W., Ling, T. W., and Scheuermann, P., editors, *ER*, volume 2813 of *Lecture Notes in Computer Science*, pages 307–320. Springer. (Cited on pages 41 and 60.)

Ubuntu (2012). Ubuntu. `http://ubuntu.com/`. (Cited on page 98.)

Ullman, J. D. (1989a). Bottom-up beats top-down for datalog. In [Silberschatz, 1989], pages 140–149. (Cited on page 99.)

Ullman, J. D. (1989b). *Principles of Database and Knowledge-Base Systems, Volume II*. Computer Science Press. (Cited on pages 58, 63 and 104.)

Ullman, J. D. (2000). Information integration using logical views. *Theor. Comput. Sci.*, 239(2):189–210. (Cited on pages 46 and 47.)

Uservoice (2012). Uservoice. `http://www.uservoice.com`. (Cited on page 184.)

van den Heuvel, W.-J., Weigand, H., and Hiel, M. (2007). Configurable adapters: the substrate of self-adaptive web services. In *Proceedings of the ninth international conference on Electronic commerce*, ICEC '07, pages 127–134, New York, NY, USA. ACM. (Cited on page 42.)

van der Meyden, R. (1998). Logical approaches to incomplete information: A survey. In Chomicki, J. and Saake, G., editors, *Logics for Databases and Information Systems*, pages 307–356. Kluwer. (Cited on page 47.)

Vardi, M. Y. (1986). Querying logical databases. *J. Comput. Syst. Sci.*, 33(2):142–160. (Cited on page 47.)

Vassiliadis, P. (2011). A survey of extract-transform-load technology. In Taniar, D. and Chen, L., editors, *Integrations of Data Warehousing, Data Mining and Database Technologies*, pages 171–199. Information Science Reference. (Cited on page 60.)

Vassiliadis, P. and Simitsis, A. (2009). Extraction, transformation, and loading. In Liu, L. and Özsu, M. T., editors, *Encyclopedia of Database Systems*, pages 1095–1101. Springer US. (Cited on page 60.)

Vitvar, T., Kopecký, J., Viskova, J., and Fensel, D. (2008). Wsmo-lite annotations for web services. In *ESWC*, pages 674–689. (Cited on page 52.)

Vrdoljak, B., Banek, M., and Rizzi, S. (2003). Designing web warehouses from xml schemas. In Kambayashi, Y., Mohania, M. K., and Wöß, W., editors, *DaWaK*, volume 2737 of *Lecture Notes in Computer Science*, pages 89–98. Springer. (Cited on page 41.)

W3C (2001). *Web Service Description Language 1.1.* (Cited on pages 22, 32, 51 and 185.)

W3C (2004). *Web Services Architecture.* (Cited on pages 181 and 185.)

W3C (2009). *Web Application Description Language.* (Cited on page 52.)

Wolf, G., Kalavagattu, A., Khatri, H., Balakrishnan, R., Chokshi, B., Fan, J., Chen, Y., and Kambhampati, S. (2009). Query processing over incomplete autonomous databases: query rewriting using learned data dependencies. *VLDB J.*, 18(5):1167–1190. (Cited on page 49.)

Wrike (2012). Wrike. `http://www.wrike.com`. (Cited on page 184.)

XSD Generator: FreeFormatter (2012). Xsd generator: Freeformatter. `http://www.freeformatter.com/xsd-generator.html`. (Cited on page 93.)

Xyleme, L. (2001). A dynamic warehouse for xml data of the web. *IEEE Data Eng. Bull.*, 24(2):40–47. (Cited on pages 38, 39 and 43.)

Yang, G., Kifer, M., and Chaudhri, V. K. (2006). Efficiently ordering subgoals with access constraints. In *PODS*, pages 183–192. (Cited on page 50.)

Yerneni, R., Li, C., Garcia-Molina, H., and Ullman, J. D. (1999). Computing capabilities of mediators. In *SIGMOD Conference*, pages 443–454. (Cited on pages 49 and 160.)

Yu, L., Huang, W., Wang, S., and Lai, K. K. (2008). Web warehouse - a new web information fusion tool for web mining. *Information Fusion*, 9(4):501–511. (Cited on page 41.)

Zendesk (2012). Zendesk. `http://www.zendesk.com`. (Cited on page 184.)

Zhou, G., Hull, R., and King, R. (1996). Generating data integration mediators that use materialization. *Journal of Intelligent Information Systems*, 6(2-3):199–221. (Cited on page 40.)

Zhou, G., Hull, R., King, R., and Franchitti, J.-C. (1995). Data integration and warehousing using h2o. *IEEE Data Eng. Bull.*, 18(2):29–40. (Cited on page 40.)

Zhu, F., Turner, M., Kotsiopoulos, I. A., Bennett, K. H., Russell, M., Budgen, D., Brereton, P., Keane, J. A., Layzell, P. J., Rigby, M., and Xu, J. (2004). Dynamic data integration using web services. In *ICWS*, pages 262–269. IEEE Computer Society. (Cited on pages 43, 44 and 45.)

Zoho Projects (2012). Zoho projects. `http://www.zoho.com/projects`. (Cited on page 184.)

Zoho Support (2012). Zoho support. `http://www.zoho.com/support`. (Cited on page 184.)

Zorrilla, M., Millan, S., and Menasalvas, E. (2005). Data web house to support web intelligence in e-learning environments. In *Granular Computing, 2005 IEEE International Conference on*, volume 2, pages 722–727. IEEE. (Cited on page 41.)

Zyp, K. et al. (2010). A json media type for describing the structure and meaning of json documents. *draft-zyp-json-schema-02 (work in progress)*. (Cited on page 53.)

Appendix A

Glossary

Resource: [Fielding, 2000] defines *resource R as a temporally varying membership function $M_R(t)$, which for time t maps to a set of entities, or values, which are equivalent. The values in the set may be resource representations and/or resource identifiers.*

Web Services: W3C defines a web service [W3C, 2004] as *a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards*

Data Warehouse: A data warehouse [Inmon, 1992] is a *subject-oriented, integrated, time-varying, non-volatile collection of data in support of the management's decision-making process.*

Appendix B

Analysis of Web Service API

We considered actual web services from three different business domains for our study. We found the domain of the analyzed web services explicitly mentioned on their respective websites. We analyzed these web services on various criteria in order to understand how much human effort is required to integrate with them.

B.1 Web services Analyzed

The three domains that we considered are project management, email marketing and support (helpdesk). In order to understand the three domains, we briefly describe them below

Email Marketing web services: Email marketing is a form of direct marketing which uses email as a means of communicating to a wide audience. It is commonly used by companies to inform the (subscribed) audience about new products and newly released features. Companies use this marketing technique to give offers to a selected group of customers. Email marketing services offer features like creating new campaigns, managing the list of subscribers, and collecting the campaign statistics. Campaign statistics include the information like the number of people who opened the email campaign, the count of people who clicked the campaign or forwarded it to others, the count of people who marked it as an abuse and also the number of people who unsubscribed after receiving a campaign.

Project Management web services: An enterprise has a number of projects that its employees work on. A project has a number of tasks assigned to different

employees. Every task has an associated status like open, due or closed to respectively signify whether it is still not complete, it is overdue or it is complete. A project management service provides features to track the progress of a project and the associated tasks. It serves many purposes: planning and estimation of a project, resource allocation, collaboration among team members, tracking the progress, communication and documentation of all activities,

Support (Helpdesk) web services A helpdesk web service helps customers (intended or current) to raise their complaints and problems regarding various products. The support (helpdesk) web services help the customers to file their complaints, concerns and suggestions on an online web portal. They also help the companies to track the progress on how the user request has been responded internally. For this purpose, every user request has an associated ticket. A ticket gives the complete details of who made the request, how was the request made (online web portal, social networking websites, email or phone), who (or which internal team) is acting on it and what is the due date for the resolution of the problem.

In addition to these, they also provide features like online forums, where the customers can find resolutions to the commonly occurring problems.

The names of the web services analyzed and their respective domains are given in the Table B.1

Table B.1: Web services

Project Management Services	
1.	Basecamp [Basecamp, 2012]
2.	Liquid Planner [LiquidPlanner, 2012]
3.	Teambox [Teambox, 2012]
4.	Wrike [Wrike, 2012]
5.	Zoho Projects [Zoho Projects, 2012]
6.	TeamWork [Teamwork, 2012]
Email Marketing Services	
7.	MailChimp [Mailchimp, 2012]
8.	Campaign Monitor [Campaign Monitor, 2012]
9.	Constant Contact [ConstantContact, 2012]
10.	iContact [iContact, 2012]
11.	AWeber [Aweber, 2012]
Support (Helpdesk) Services	
12.	Zendesk [Zendesk, 2012]
13.	Desk [Desk, 2012]
14.	Kayako [Kayako, 2012]
15.	Zoho Support [Zoho Support, 2012]
16.	Uservoice [Uservoice, 2012]
17.	FreshDesk [Freshdesk, 2012]

B.2 Criteria of Analysis

Next we take a look at the API of each of the individual web services mentioned in the Table B.1 and analyze them on various criteria described in the Table B.2. This analysis is used to understand how much manual effort is required to integrate with these web services and what aspects of the web service integration can be generalized.

Table B.2: Web Service API Information Template

1.	Category (Domain) of the web services
2.	Use of Web Service Description Language
3.	Conformance to the REST
4.	Support for Versions
5.	Authentication Methods
6.	Resources Involved
7.	Message Formats for Communication
8.	Service Level Agreement/ Terms and Conditions of Use
9.	Interface Details (Generic details of all the operations)
10.	Data Types
11.	Dynamic Nature of the Resources
12.	Operation Invocation Sequence
13.	Pagination

B.2.1 Category (Domain) of web services

By classifying the APIs into various categories based on their functions, it is easier to compare similar services. For the current set of web services that we have taken into consideration, their domains are summarized in Table B.1.

B.2.2 Use of Web Service Definition Languages

WSDL (Web Service Definition Language) [W3C, 2001] was proposed to handle automatic web service integration, web service discovery and web service composition [W3C, 2004]. But for the current set of web services taken into account, we found that none of them have exposed any WSDL files. All of them have given their web service description using HTML in human readable format. Another proposal was to combine the idea of machine readability and human readability into a single format called hRESTS, an HTML microformat [Kopecký et al., 2008]. But use of hRESTS is also missing in the considered web services. We considered the use of machine readable

standards like WSDL, hRESTS to understand how much of human effort is required to work with the given web services.

We also took into consideration some other web services other than those mentioned above and we found that barring few exceptions, none of the web services exposed a WSDL file describing their interface. Thus we require some manual efforts to transform the web service API description written in human readable format on the respective web sites of the web service to any chosen internal format.

B.2.3 Conformance to the REST

REST (Representational State Transfer) [Fielding, 2000] architecture style was proposed for the web services to support automatic integration and avoid tight coupling between clients and servers. REST [Fielding, 2000] is defined by four interface constraints: *"identification of resources; manipulation of resources through representations; self-descriptive messages; and, hypermedia as the engine of application state."*

An API may or may not be adherent to the REST architectural style. In most of the non adherent APIs, hypermedia is not used as a state of application engine or (and) the messages are not self-descriptive. Both of these features make the client and the server tightly coupled (thereby losing the REST benefits).

The conformance to REST for the considered web services is summarized in the table B.3. Note that, our use of the term *REST like* is based on the current industrial practice. It has become a common industry practice to declare a web service RESTful when it is completely adherent to the REST architecture style and *REST like* when some principles are not followed.

B.2.4 Support of Versions

Considering the fact that not all web services are REST compliant, we need to consider how web services evolve from time to time and how they inform this change in their interfaces to the API users. A common practice among the web service providers is to allow versions of the API. The major changes between two API versions include the following: change in message formats (XML, JSON, plain-text, signed content), change in authentication and authorization mechanisms, operation interface (input, output or error) changes, deprecation of one or more operations, change in service level

Table B.3: Web services and their Conformance to REST

Project Management Services		
1.	Basecamp	REST like (Not hypermedia driven)
2.	Liquid Planner	REST like (Not hypermedia driven)
3.	Teambox	REST like (Not hypermedia driven)
4.	Wrike	Not REST
5.	Zoho Projects	Not REST
6.	TeamWork	REST like (Not hypermedia driven)
Email Marketing Services		
7.	MailChimp	Not REST
8.	Campaign Monitor	REST like (Not Hypermedia driven)
9.	Constant Contact	REST
10.	iContact	REST like (Not Hypermedia driven)
11.	AWeber	REST
Support (Helpdesk) Services		
12.	Zendesk	REST like (Not hypermedia driven)
13.	Desk	Not REST
14.	Kayako	REST
15.	Zoho Support	Not REST
16.	Uservoice	REST like (Not Hypermedia driven)
17.	FreshDesk	REST like (Not Hypermedia driven)

agreement or terms of conditions of use, resource representation changes, addition of new resources and addition of one or more operations.

Versioning helps the service providers to release new features and also gives the developers some time before actually shifting on to the latest version. The service providers can gradually terminate the older versions when a significant number of developers or web service clients have migrated to the latest version.

The support for versions including the current versions is summarized in the Table B.4. This table also give the reader an idea about the version of the web service API that we took into consideration. We also show how the service providers differentiates between two versions of API (eg., in HTTP header, body or URL)

B.2.5 Authentication Methods

Web services need to authenticate and authorize the use of third party users. Various methods of authentication include: basic HTTP authentication (username and password) and open authentication standards (OAuth 1.0, OAuth 2.0).

The use of various authentication mechanisms have been summarized in the Table B.5. Some require an additional parameter called the key often known as the API

Table B.4: Web services and the Support for Versions

Project Management Services		
1.	Basecamp	Yes, in URL (Current Version:v1)
2.	Liquid Planner	Yes, in Header (X-API-Version) (Current Version: 3.0.0)
3.	Teambox	Yes, in URL (Current Version:1)
4.	Wrike	Yes, in URL (Current Version:v2)
5.	Zoho Projects	None
6.	TeamWork	None
Email Marketing Services		
7.	MailChimp	Yes, in URL (Current Version:1.3)
8.	Campaign Monitor	Yes, in URL (Current Version:v3)
9.	Constant Contact	None
10.	iContact	Yes, in URL (Current Version:2.2)
11.	AWeber	Yes, in URL (Current Version:1.0)
Support (Helpdesk) Services		
12.	Zendesk	None
13.	Desk	Yes, in URL (Current Version:v1)
14.	Kayako	None
15.	Zoho Support	None
16.	Uservoice	Yes, in URL (Current Version:v1)
17.	FreshDesk	None

key for tracking the API usage of the third party users (for pricing purposes).

B.2.6 Resources Involved

In REST, the key abstraction of information is a resource. REST defines a resource as any information that can be named. Continuing with the examples from the three domains, examples of resources include daily open tasks, all campaigns, daily campaign statistics, daily closed tickets, open tickets, all forums, all projects, archived projects etc.

A static resource refer to the same set of values or entities irrespective of the time. Examples include new projects, forums or campaigns created on January 24, 2012. Examples of non-static resources include daily open tasks, daily campaign statistics since their values vary from time to time.

The primary resources related to project management services are: company, clients, project, task, user, user generated content like conversations, comments. The primary resources of a support(helpdesk) service: company, groups, tickets, attachments, users, tags, forums and comments. A closer look of email marketing services help us to determine the following primary resources: subscriber, client, campaign and statistics.

Table B.5: Web services and the Authentication Mechanisms

Project Management Services		
1.	Basecamp	OAuth 2.0
2.	Liquid Planner	HTTP Basic Authentication (Email and Password)
3.	Teambox	OAuth 2.0
4.	Wrike	OAuth 1.0
5.	Zoho Projects	Basic HTTP Authentication
6.	TeamWork	HTTP Basic Authentication (API Token and Bogus Text as password)
Email Marketing Services		
7.	MailChimp	HTTP Basic Authentication: API Key and DC (location of the server)
8.	Campaign Monitor	HTTP Basic Authentication
9.	Constant Contact	OAuth2.0
10.	iContact	Basic HTTP Authentication (App ID, Sandbox Username, Password)
11.	AWeber	OAuth 1.0
Support (Helpdesk) Services		
12.	Zendesk	HTTP basic authentication
13.	Desk	OAuth 1.0a
14.	Kayako	Basic HTTP Authentication (API Key, Salt, Signature). Content must be signed with the Key
15.	Zoho Support	HTTP Basic Authentication (API Key and Ticket ID)
16.	Uservoice	OAuth 1.0
17.	FreshDesk	Basic HTTP Authentication (Username, Password)

APIs are used to get access to these resources through various methods and operations. The resources that we obtain from various web services are detailed in the Table B.6.

B.2.7 Message Formats for Communication

Communication between a web service client and a web service provider includes the request for desired resource (with optional parameters like the resource identifier) by the client and the web service provider response in the form of a resource representation or sometimes error (diagnostics) while accessing the requested resource. The commonly found message formats for the communication include JSON [Crockford, 2006], XML [Bray et al., 1997], HTML [Raggett et al., 1999]. JavaScript Object Notation (JSON) is a lightweight, text-based, language-independent data interchange format. It was derived from the ECMAScript Programming Language Standard. XML (Extensible Markup Language) is a markup language designed to be both human readable and machine readable. HTML (HyperText Markup Language) is another markup language commonly used to create web pages and viewed using the web browser. The use of

Table B.6: Web services and the Resources

Project Management Services		
1.	Basecamp	Project, Task
2.	Liquid Planner	Project, Task
3.	Teambox	Project, Task
4.	Wrike	Task
5.	Zoho Projects	Project, Task
6.	TeamWork	Project, Task
Email Marketing Services		
7.	MailChimp	Campaign, Statistics
8.	Campaign Monitor	Campaign, Statistics
9.	Constant Contact	Campaign, Statistics
10.	iContact	Campaign, Statistics
11.	AWeber	Campaign, Statistics
Support (Helpdesk) Services		
12.	Zendesk	Forum, Topic, Ticket
13.	Desk	Ticket
14.	Kayako	Ticket, Topic
15.	Zoho Support	Ticket
16.	Uservoice	Forum, Topic, Ticket
17.	FreshDesk	Forum, Topic, Ticket

various message formats among the considered web services are summarized in the Table B.7.

B.2.8 Service Level Agreement/ Terms and Conditions of Use

Every service limits the usage of its API like the number of calls that can be made on a particular day or a period of time. The terms of a service also determines how certain information from the service can be used.

The SLA of various web services is described in the Table B.8. This information is crucial since the number of API calls (a limited resource) must be efficiently used.

B.2.9 Interface Details

Web service API supports multiple operations to access and manipulate the various resources. They generally use the HTTP [Fielding et al., 1999] protocol for the communication. Every HTTP request requires a URL(Uniform Resource Location), header, method and body. The HTTP URL corresponds to the location of the resource. The HTTP headers are used to specify the content type of the communication, the authen-

Table B.7: Web services and the Message Formats

Project Management Services		
1.	Basecamp	JSON
2.	Liquid Planner	JSON
3.	Teambox	JSON
4.	Wrike	XML, JSON
5.	Zoho Projects	XML, JSON
6.	TeamWork	XML, JSON
Email Marketing Services		
7.	MailChimp	XML, JSON, PHP, Lolcode
8.	Campaign Monitor	XML, JSON
9.	Constant Contact	XML
10.	iContact	XML, JSON
11.	AWeber	JSON
Support (Helpdesk) Services		
12.	Zendesk	XML, JSON
13.	Desk	JSON
14.	Kayako	XML
15.	Zoho Support	XML, JSON
16.	Uservoice	XML, JSON
17.	FreshDesk	XML

tication parameters and also to specify certain client capabilities (whether compression supported by the clients). HTTP methods include GET, POST, PUT and DELETE respectively used to get, create, update and delete a resource in the server. There are other HTTP methods, but the above four are commonly used methods. The HTTP body contains the request parameters from the client and response from the service providers.

Therefore when we analyze the operations of a web service, we take into account the following

1. Operation name
2. Operation request parameters: For every operation request, the following information is required
   (a) The URL of the resource/operation
   (b) HTTP Method
   (c) HTTP Headers (example: authentication and content-type)
   (d) HTTP Body (other information like the resource identifier)
3. Operation response parameters
4. Operation error parameters

We also made an observation during our study that the use of HTTP methods is

Table B.8: Web services and the Service Level Agreement

Project Management Services		
1.	Basecamp	500 requests per 10 second period from the same IP address for the same account
2.	Liquid Planner	Each user account may make up to 30 requests per 15 seconds
3.	Teambox	N.A.
4.	Wrike	N.A.
5.	Zoho Projects	Error code:6403 on exceeding the limit
6.	TeamWork	120 requests per minute
Email Marketing Services		
7.	MailChimp	Stream Timeout: 300 seconds and Connection Timeout:30 seconds
8.	Campaign Monitor	N.A.
9.	Constant Contact	N.A.
10.	iContact	N.A.
11.	AWeber	60 requests per minute
Support (Helpdesk) Services		
12.	Zendesk	Limit Exists (Value not specifically mentioned)
13.	Desk	60 requests per minute
14.	Kayako	N.A.
15.	Zoho Support	250 calls per day / Organization (Free)
16.	Uservoice	N.A.
17.	FreshDesk	N.A.

not uniform. Take for example, we have observed the use of HTTP POST used to retrieve or delete a resource. Thus a generic assumption like HTTP GET is used for resource access cannot be made.

B.2.10 Data Types

Apart from the data types like integers, strings and floating point numerals, we also considered other data types: resource identifier, date (or time) and enumerated data types.

Resources are usually identified by their identifiers and this information is usually present in the resource representation. While extracting this information from various representation from various web services, we notice that resource identifiers are not unique across web services. The use of URI (uniform resource identifiers) to identify the resources is missing. A resource identifier like a ticket identifier from Zendesk is identical to the ticket identifier from Freshdesk, that is, both Zendesk and Freshdesk may have the same ticket identifier 1. Thus while considering the tickets from various web services, one must also take into account the source of the information and not solely depend on the resource identifiers.

Various web services use different formats to represent data and time. There is no particular standard (like ISO 8601) employed across the web services. Web services use different ways to represent enumerated values. Take for example, 0, 1, 2 are used to respectively specify the priority of a ticket as low, medium and high.

B.2.11 Dynamic Nature of the Resources

As described before, a resource is a mapping between time and a set of values or entities. The resources that can be accessed and manipulated by the web service API operation are usually dynamic in nature. Take for example, Freshdesk has an API operation to get all the open tickets. Note that this resource *open tickets* is dynamic in nature, since a ticket that was in the open state and retrieved yesterday may not be present if the same API operation is called today since the ticket may have been resolved during this period.

B.2.12 Operation Invocation Sequence

Certain operation calls cannot be made directly unless we have some additional information that must be passed as parameters to the corresponding operation calls. This additional information is obtained from some other sources or by making a different operation call of the same (or different) web service API. Thus for accessing some resources, there is an underlying operation invocation sequence (not explicitly mentioned, sometimes).

Take for example (for the clarity of the reader, we have simplified the situation), Basecamp has three operation calls

1. Get all projects (It takes no input parameters)
2. Get all Todo lists for a project(It takes as input the project identifier)
3. Get all Todo items of a Todo list (It takes as input the Todo list identifier)

If a client is interested in getting all the todo items, there is no way that the third operation can be directly called (unless of course, the client somehow knows all the todo item identifiers). Thus the client needs to make the second call that again requires an argument, the values for which can only be obtained from the first call. Thus for accessing or manipulating certain resources, an invocation sequence must be followed.

B.2.13 Pagination

Pagination is a special case of operation invocation sequence, where the same operation call must be made multiple times to obtain the complete representation of a resource. Resource representation sometimes requires a lot of space. In order to reduce the size of the output response in a single API call, most web services support paginated operation requests. We observed the use of pagination in web service APIs like Mailchimp and Zendesk.

B.3 Conclusion

We summarize our analysis with these web services in Tables B.9, B.10 and B.11. In addition to demonstrating the significant heterogeneity existing among the web service API (in terms of the use of message formats, service level agreements, data types, operation and resource handling etc.), the analysis also shows that a lot of service providers are still using the basic web technologies (HTTP, XML, JSON) to expose their API to third party users and the use of machine readable web service description hasn't still caught up as a common practice in the industry.

Table B.9: Web Service API Analysis on Project Management Services

Web Service	**Basecamp**	**Liquid Planner**	**Teamwork**	**Zoho Projects**	**Wrike**	**Teambox**
1. API Description	HTML page	HTML page	HTML page	HTML page	HTML page	HTML page
2. Conformance to REST	REST like	REST like	REST like	Not REST	Not REST	REST like
3. Version	v1	3.0.0	N.A.	N.A.	v2	1
4. Authentication	Basic HTTP, OAuth 2	Basic HTTP	Basic HTTP	Basic HTTP	OAuth 1.0	OAuth 2.0
5. Resources Involved	Project,Todo List, Todo	Project, Task	Project, Task List, Task	Project, Task List, Task	Task	Project, Task
6. Message Formats	JSON	JSON	XML, JSON	XML, JSON	XML, JSON	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	N.A	N.A.
8. HTTP Resource Access	GET	GET	GET	POST	POST	GET
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	Enumerated dt (Task Status), Date	Enumerated dt (Project and Task Status), Date
10. Dynamic nature of the resources	Yes (Project and Todo Status)	Yes (Project and Task Status)	Yes (Project and Task Status)	Yes(Project and Task Status)	Yes (Task Status)	Yes (Project and Task Status)
11. Operation Invocation Sequence Required	Yes	No	Yes	Yes	Yes	Yes
12. Pagination	No	No	No	Yes	Yes	No

Table B.10: Web Service API Analysis on Email Marketing services

Web Service	**Mailchimp**	**Campaign Monitor**	**iContact**	**Constant Contact**	**AWeber**
1. API Description	HTML page	HTML page	HTML page	HTML page	HTML page
2. Conformance to REST	Not REST	REST like	REST like	REST	REST
3. Version	1.3	v3	2.2	N.A.	1.0
4. Authentication	Basic HTTP	Basic HTTP, OAuth 2	Basic HTTP (with Sandbox)	OAuth 2.0	OAuth 1.0
5. Resources Involved	Campaign, Campaign Statistics	Campaign, Campaign Statistics	Campaign, Campaign Statistics	Campaign, Campaign Statistics	Campaign, Campaign Statistics
6. Message Formats	XML, JSON, PHP, Lolcode	XML, JSON	XML, JSON	XML	JSON
7. Service Level Agreement	N.A.	N.A.	75,000 requests /24h, with a max of 10,000 requests /1h	N.A	60 requests per minute
8. HTTP Resource Access	GET	GET	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	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)	Yes (Campaign Status)	Yes (Campaign Status)
11. Operation Invocation Sequence Required	Yes	Yes	No	Yes	Yes
12. Pagination	Yes	No	No	Yes	Yes

Table B.11: Web Service API Analysis on Support/Helpdesk Services

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

Appendix C

DaWeS: Examples

In this chapter, we will see what are the steps involved when a new web service has to be added, how to formulate a query over the global schema and how to define performance indicator queries.

C.1 Global Schema

We present here the global schema relations and the corresponding attributes identified during our analysis of the three domains: project management, email marketing and support(helpdesk). The detailed description of the relations and their attributes are given in the Table C.1

Table C.1: Global Schema Relations an their attributes

Project Management Services		
Project	pid	Project Identifier
	src	Source
	pname	Project Name
	pcdate	Project Creation Date
	pstatus	Project Status
Task List	pid	Project Identifier
	src	Source
	tlid	Task List Identifier
	tlid	Task List Identifier
	tid	Task Identifier

Task

	src	Source
	tname	Task Name
	tcdate	Task Creation Date
	tddate	Task Due Date
	tcmpdate	Task Completion Date
	tstatus	Task Status
Email Marketing Services		
Client	clid	Client Identifier
	src	Source
Campaign	cmid	Campaign Identifier
	src	Source
	cmname	Campaign Name
	cmcdate	Campaign Creation Date
	cmstatus	Campaign Status
Campaign Statistics	cmid	Campaign Identifier
	src	Source
	cmar	Campaign Abuse Reports
	cmctr	Campaign Click-through Reports
	cmbr	Campaign block Reports
	cmfr	Campaign forward Reports
Support (Helpdesk) Services		
Forum Category	fcid	Forum Category Identifier
	src	Source
	fcname	Forum Category Name
Forum	fid	Forum Identifier
	src	Source
	fname	Forum Name
	fcdate	Forum Creation Date
Topic	tpid	Topic Identifier
	src	Source
	tpname	Topic Name
	fid	Forum Identifier
	tpcate	Topic Creation Date
Ticket	tkid	Ticket Identifier
	src	Source
	tkname	Ticket Name
	tkcdate	Ticket Creation Date
	tkddate	Ticket Updation Date
	tkcmpdate	Ticket Completion Date
	tkpriority	Ticket Priority

	tkstatus	Ticket Status
Miscellaneous		
Page	pgno	Page Number
	src	Source
	operation	Web service operation
	Limit	Page Limit
Next Page	pglink	Page Link
	src	Source
	operation	Web service operation

Next we need to specify data constraints on the global schema relations. Considering the fact that the Inverse rules algorithm can support full and functional dependencies, we make use of this to support the functional dependenices, especially the key dependencies. For every global schema relation, we specify its respective primary key as given in Table C.2.

Table C.2: Primary Key for Global Schema Relations

S.No.	Global Schema Relation	Key
1.	Project(pid, src, pname, pcdate, pstatus)	pid,src
2.	TaskList(pid, src, tlid)	tlid,src
3.	Task(tlid, tid, src, tname, tcdate, tddate, tcmpdate, tstatus)	tid,src
4.	Campaign(cmid, src, cmname, cmcdate, cmstatus)	cmid,src
5.	Client(clid,src)	
6.	CampaignStatistics(cmid, src, cmar, cmctr, cmfr, cmbr)	cmid,src
7.	Forum(fid, src, fname, fcdate)	fid,src
8.	ForumCategory(fcid, src, fcname)	fcid,src
9.	Topic(tpid, src, tpname, tpcdate, fid)	tpid,src
10.	Ticket(tkid, src, tkname, tkcdate, tkddate, tkcmpdate, tkpriority, tkstatus)	tkid,src
11.	Page(pgno, src, operation, limit)	
12.	NextPage(pglink, src, operation)	

Example C.1.1. *To specify the primary keys of the global schema relation* Forum*, we make use of the* EQUAL *predicate of IRIS. To specify following two functional dependencies in global schema relation Forum:*

$Forum : fid, src \rightarrow fname$
$Forum : fid, src \rightarrow fcdate,$

we specify it in the following manner:

```
E(?fname1,?fname2):-Forum(?fid1, ?src1, ?fname1, ?fcdate1),
                    Forum(?fid2, ?src2, ?fname2, ?fcdate2),
                    EQUAL(?fid1,?fid2),EQUAL(?src1,?src2).
E(?fcdate1,?fcdate2):-Forum(?fid1, ?src1, ?fname1, ?fcdate1),
                      Forum(?fid2, ?src2, ?fname2, ?fcdate2),
                      EQUAL(?fid1,?fid2),EQUAL(?src1,?src2).
```

■

C.2 Web Service API Operations

We use the local schema relations mentioned in the Table C.3. For every web service API operation considered for testing, we present its Local as View Mapping, the XSD schema and XSLT transformation. Note that in the examples below, the lines between *<xsl:for-each select⋯ >* and *</xsl:for-each>* must be written in a single line. For the reasons of readability and clarity, we have put them on separate lines. We also specify the attributes of every local schema relation along with its access pattern. We use f for the free(output) attributes corresponding to the response of a web service API operation and b for the bound(input) attributes corresponding to the request parameters.

1. **CampaignMonitorv3SentCampaign**:This web service API operation takes as input the concerned client identifier(*clid*), the possible values of which are obtained from the web service operation C.4. The response consists of all the sent campaigns. We are interested in the campaign identifiers *cmid* and campaign names *cmname*.
   (a) **LAVMapping**:

$$\begin{aligned} CampaignMonitorv3SentCampaign^{bff}(clid, cmid, cmname) \leftarrow \\ Client(clid,' \text{CampaignMonitor v3 API}'), \\ Campaign(cmid,' \text{CampaignMonitor v3 API}', cmname, cmcdate,' \text{Sent}'). \end{aligned} \tag{C.1}$$

   (b) **XSD Schema**: As discussed in the LAV Mapping C.1 above, we are interested in the following elements (XPath given) from the response schema

Table C.3: Local Schema Relations

S.No.	Local Schema Relation
1.	$CampaignMonitorv3SentCampaign$
2.	$CampaignMonitorv3ScheduledCampaign$
3.	$CampaignMonitorv3DraftCampaign$
4.	$CampaignMonitorv3Client$
5.	$CampaignMonitorv3CampaignStatistics$
6.	$IContactv2\_2Campaign$
7.	$Mailchimpv1\_3TotalCampaign$
8.	$Mailchimpv1\_3Campaign$
9.	$Mailchimpv1\_3CampaignStatistics$
10.	$Basecampv1Projects$
11.	$Basecampv1TodoLists$
12.	$Basecampv1Tasks$
13.	$Basecampv1CompletedTasks$
14.	$LiquidPlannerv3\_0\_0Projects$
15.	$LiquidPlannerv3\_0\_0Tasks$
16.	$TeamworkpmProjects$
17.	$TeamworkpmTodoLists$
18.	$TeamworkpmTasks$
19.	$ZohoProjectsProjects$
20.	$FreshdeskForum$
21.	$FreshdeskTopic$
22.	$FreshdeskForumCategory$
23.	$FreshdeskTicket$
24.	$Uservoicev1Ticket$
25.	$Uservoicev1TotalTickets$
26.	$Zendeskv2Forum$
27.	$Zendeskv2Topic$
28.	$Zendeskv2Ticket$
29.	$Zendeskv2SolvedTicket$
30.	$Zendeskv2TicketDetails$
31.	$ZohoSupportTask$
32.	$Deskv2Topic$
33.	$Deskv2TotalTopics$
34.	$Deskv2TotalCases$
35.	$Deskv2Case$

i. Campaign identifier ($Campaigns/Campaign/CampaignID$)
ii. Campaign name ($Campaigns/Campaign/Name$).

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Campaigns">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Campaign" maxOccurs="unbounded" minOccurs="0">
          <xs:complexType>
            <xs:all minOccurs="1" maxOccurs="1">
              <xs:element name="CampaignID">
                <xs:complexType>
                  <xs:simpleContent>
                    <xs:extension base="xs:string">
                      <xs:attribute name="type" type="xs:string">
                      </xs:attribute>
                    </xs:extension>
                  </xs:simpleContent>
                </xs:complexType>
              </xs:element>
              <xs:element name="Name">
                <xs:complexType>
                  <xs:simpleContent>
                    <xs:extension base="xs:string">
                      <xs:attribute name="type" type="xs:string">
                      </xs:attribute>
                    </xs:extension>
                  </xs:simpleContent>
                </xs:complexType>
              </xs:element>
              <xs:any maxOccurs="unbounded" processContents="lax"/>
            </xs:all>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

(c) **XSLT**: We only extract $CampaignID$ and $Name$ from the operation response as shown below.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:fn="http://www.w3.org/2005/xpath-functions" version="2.0">
  <xsl:output method="text" omit-xml-declaration="yes"
    cdata-section-elements="namelist"/>
  <xsl:template match="/">
    <xsl:for-each select="Campaigns/Campaign">
      <xsl:value-of select="CampaignID"/>,
      <xsl:value-of select="Name"/>
      <xsl:text>&#xa;</xsl:text>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

2. **CampaignMonitorv3ScheduledCampaign**: This web service operation takes as input the concerned client identifier(*clid*), the possible values of which are obtained from the web service operation C.4. The response consists of all the scheduled campaigns. We are interested in the campaign identifiers *cmid*, campaign names *cmname* and campaign creation dates *cmdate*.

   (a) **LAV Mapping**:

$$CampaignMonitorv3ScheduledCampaign^{bfff}(clid, cmid, cmname, cmdate) \leftarrow Client(clid, '\text{ CampaignMonitor v3 API}'), Campaign(cmid, '\text{ CampaignMonitor v3 API}', cmname, cmdate, '\text{ Scheduled}'). \quad (C.2)$$

   (b) **XSD**: As discussed in the LAV Mapping C.2, we are interested in the following elements (XPath) from the response schema

      i. Campaign identifier $ScheduledCampaigns/Campaign/CampaignID$
      ii. Campaign name $ScheduledCampaigns/Campaign/Name$.
      iii. Campaign Creation Date $ScheduledCampaigns/Campaign/DateCreated$.

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="ScheduledCampaigns">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Campaign" maxOccurs="unbounded" minOccurs="0">
          <xs:complexType>
            <xs:all minOccurs="1" maxOccurs="1">
              <xs:element name="CampaignID">
                <xs:complexType>
                  <xs:simpleContent>
                    <xs:extension base="xs:string">
                      <xs:attribute name="type" type="xs:string">
                      </xs:attribute>
                    </xs:extension>
                  </xs:simpleContent>
                </xs:complexType>
              </xs:element>
              <xs:element name="Name">
                <xs:complexType>
                  <xs:simpleContent>
                    <xs:extension base="xs:string">
                      <xs:attribute name="type" type="xs:string">
                      </xs:attribute>
                    </xs:extension>
                  </xs:simpleContent>
                </xs:complexType>
              </xs:element>
              <xs:element name="DateCreated">
                <xs:complexType>
                  <xs:simpleContent>
                    <xs:extension base="xs:string">
                      <xs:attribute name="type" type="xs:string">
                      </xs:attribute>
                    </xs:extension>
                  </xs:simpleContent>
```

```
            </xs:complexType>
          </xs:element>
          <xs:any maxOccurs="unbounded" processContents="lax"/>
        </xs:all>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
 </xs:complexType>
</xs:element>
</xs:schema>
```

(c) **XSLT**: We only extract $CampaignID$, $Name$ and $DateCreated$ from the operation response as shown below. Note how we extract only the first ten characters of $DateCreated$ which corresponds to YYYY-MM-DD (date in year-month-day format).

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:fn="http://www.w3.org/2005/xpath-functions" version="2.0">
  <xsl:output method="text" omit-xml-declaration="yes"
   cdata-section-elements="namelist"/>
  <xsl:template match="/">
    <xsl:for-each select="ScheduledCampaigns/Campaign">
      <xsl:value-of select="CampaignID"/>,
      <xsl:value-of select="Name"/>,
      <xsl:value-of select="fn:substring(current()/DateCreated,1,10)"/>
      <xsl:text>&#xa;</xsl:text>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

3. **CampaignMonitorv3DraftCampaign**: This web service operation takes as input the concerned client identifier($clid$), the possible values of which are obtained from the web service operation C.4. The response consists of all the draft campaigns. We are interested in the campaign identifiers $cmid$, campaign names $cmname$ and campaign creation dates $cmdate$.

(a) **LAV Mapping**:

$$\begin{aligned} CampaignMonitorv3DraftCampaign^{bfff}(clid, cmid, cmname, cmdate) \leftarrow \\ Client(clid, '\text{ CampaignMonitor v3 API}'), \\ Campaign(cmid, '\text{ CampaignMonitor v3 API}', cmname, cmdate, '\text{ Draft}'). \end{aligned} \tag{C.3}$$

(b) **XSD**: As discussed in the LAV Mapping C.3, we are interested in the following elements (along with their XPath) from the response schema

i. Campaign identifier $Drafts/Campaign/CampaignID$
ii. Campaign name $Drafts/Campaign/Name$.
iii. Campaign Creation Date $Drafts/Campaign/DateCreated$.

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Drafts">
```

```
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Campaign" maxOccurs="unbounded" minOccurs="0">
          <xs:complexType>
            <xs:all minOccurs="1" maxOccurs="1">
              <xs:element name="CampaignID">
                <xs:complexType>
                  <xs:simpleContent>
                    <xs:extension base="xs:string">
                      <xs:attribute name="type" type="xs:string">
                      </xs:attribute>
                    </xs:extension>
                  </xs:simpleContent>
                </xs:complexType>
              </xs:element>
              <xs:element name="Name">
                <xs:complexType>
                  <xs:simpleContent>
                    <xs:extension base="xs:string">
                      <xs:attribute name="type" type="xs:string">
                      </xs:attribute>
                    </xs:extension>
                  </xs:simpleContent>
                </xs:complexType>
              </xs:element>
              <xs:element name="DateCreated">
                <xs:complexType>
                  <xs:simpleContent>
                    <xs:extension base="xs:string">
                      <xs:attribute name="type" type="xs:string">
                      </xs:attribute>
                    </xs:extension>
                  </xs:simpleContent>
                </xs:complexType>
              </xs:element>
              <xs:any maxOccurs="unbounded" processContents="lax"/>
            </xs:all>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

(c) **XSLT**: We only extract *CampaignID*, *Name* and *DateCreated* from the operation response as shown below. Note how we extract only the first ten characters of *DateCreated* which corresponds to YYYY-MM-DD (date in year-month-day format).

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:fn="http://www.w3.org/2005/xpath-functions" version="2.0">
  <xsl:output method="text" omit-xml-declaration="yes"
    cdata-section-elements="namelist"/>
    <xsl:template match="/">
```

```
      <xsl:for-each select="Drafts/Campaign">
        <xsl:value-of select="CampaignID"/>,
        <xsl:value-of select="Name"/>,
        <xsl:value-of select="fn:substring(current()/DateCreated,1,10)"/>
      <xsl:text>&#xa;</xsl:text>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

4. **CampaignMonitorv3Client**: This web service API operation takes no input and as output gives all the client identifiers.

   (a) **LAV Mapping**:

$$CampaignMonitorv3Client^{f}(clid) \leftarrow Client(clid,' \text{CampaignMonitor v3 API}'). \tag{C.4}$$

   (b) **XSD**: As discussed in the LAV Mapping C.4, we are interested in the following elements (along with their XPath) from the response schema

      i. Client identifier $Clients/Client/CampaignID$

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Clients">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Client" maxOccurs="unbounded" minOccurs="0">
          <xs:complexType>
            <xs:all minOccurs="1" maxOccurs="1">
              <xs:element name="ClientID">
                <xs:complexType>
                  <xs:simpleContent>
                    <xs:extension base="xs:string">
                      <xs:attribute name="type" type="xs:string">
                      </xs:attribute>
                    </xs:extension>
                  </xs:simpleContent>
                </xs:complexType>
              </xs:element>
              <xs:any maxOccurs="unbounded" processContents="lax"/>
            </xs:all>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

   (c) **XSLT**: We only extract $ClientID$ from the operation response as shown below.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:fn="http://www.w3.org/2005/xpath-functions" version="2.0">
  <xsl:output method="text" omit-xml-declaration="yes"
```

```
    cdata-section-elements="namelist"/>
    <xsl:template match="/">
      <xsl:for-each select="Clients/Client">
        <xsl:value-of select="ClientID"/>
        <xsl:text>&#xa;</xsl:text>
      </xsl:for-each>
    </xsl:template>
  </xsl:stylesheet>
```

5. **CampaignMonitorv3CampaignStatistics**: This web service API operation takes as input the campaign identifier *cmid*, the values of which are obtained from the various operations C.3, C.2 and C.1. The response include the detailed statistics corresponding to every campaign identified by its identifier. It includes the abuse rate *cmar*, click rate *cmctr*, forward rate $cmfr$ and bounce rate *cmbr*.

   (a) **LAV Mapping**:

$$\begin{aligned} &CampaignMonitorv3CampaignStatistics^{bfffff}(cmid, cmar, cmctr, cmfr, cmbr) \leftarrow \\ &\quad Campaign(cmid,' \text{ CampaignMonitor v3 API}', cmname, cmcdate, cmstatus), \\ &\quad CampaignStatistics(cmid,' \text{ CampaignMonitor v3 API}', cmar, cmctr, cmfr, cmbr). \end{aligned} \tag{C.5}$$

   (b) **XSD**: As discussed in the LAV Mapping C.5, we are interested in the following elements (along with their XPath) from the response schema

   i. Bounced count (*Summary/Bounced*)
   ii. Clicks count (*Summary/Clicks*)
   iii. Forwards count (*Summary/Forwards*)
   iv. SpamComplaints count (*Summary/SpamComplaints*)

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Summary">
    <xs:complexType>
      <xs:all minOccurs="1" maxOccurs="1">
        <xs:element name="Bounced">
          <xs:complexType>
            <xs:simpleContent>
              <xs:extension base="xs:integer">
                <xs:attribute name="type" type="xs:string">
                 </xs:attribute>
              </xs:extension>
            </xs:simpleContent>
          </xs:complexType>
        </xs:element>
        <xs:element name="Clicks">
          <xs:complexType>
            <xs:simpleContent>
              <xs:extension base="xs:integer">
                <xs:attribute name="type" type="xs:string">
                </xs:attribute>
              </xs:extension>
            </xs:simpleContent>
          </xs:complexType>
        </xs:element>
        <xs:element name="Forwards">
```

```
          <xs:complexType>
            <xs:simpleContent>
              <xs:extension base="xs:integer">
                <xs:attribute name="type" type="xs:string">
                </xs:attribute>
              </xs:extension>
            </xs:simpleContent>
          </xs:complexType>
        </xs:element>
        <xs:element name="SpamComplaints">
          <xs:complexType>
            <xs:simpleContent>
              <xs:extension base="xs:integer">
                <xs:attribute name="type" type="xs:string">
                </xs:attribute>
              </xs:extension>
            </xs:simpleContent>
          </xs:complexType>
        </xs:element>
        <xs:any maxOccurs="unbounded" processContents="lax"/>
      </xs:all>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

(c) **XSLT**: We only extract *SpamComplaints*, *Clicks*, *Forwards* and *Bounces* from the operation response as shown below.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:fn="http://www.w3.org/2005/xpath-functions" version="2.0">
  <xsl:output method="text" omit-xml-declaration="yes"
    cdata-section-elements="namelist"/>
    <xsl:template match="/">
      <xsl:for-each select="Summary">
        <xsl:value-of select="SpamComplaints"/>,
        <xsl:value-of select="Clicks"/>,
        <xsl:value-of select="Forwards"/>,
        <xsl:value-of select="Bounced"/>
        <xsl:text>&#xa;</xsl:text>
      </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

6. **IContactv2_2Campaign**: This web service API operation of IContact takes no input and gives all the campaigns. The response includes the identifier *cmid*, name *cmname* and the status *cmstatus* of all the campaigns.

   (a) **LAV Mapping**:

$$IContactv2\_2Campaign^{fff}(cmid, cmname, cmstatus) \leftarrow Campaign(cmid, \text{'IContact v2.2 API'}, cmname, cmcdate, cmstatus). \tag{C.6}$$

   (b) **XSD**: As discussed in the LAV Mapping C.6, we are interested in the following elements (XPath) from the response schema

i. Campaign identifier *response/campaigns/campaign/campaignId*
ii. Campaign name *response/campaigns/campaign/name*.
iii. Campaign status *response/campaigns/campaign/archiveByDefault*.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="response">
    <xs:complexType>
      <xs:sequence>
      <xs:element name="campaigns">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="campaign" maxOccurs="unbounded" minOccurs="0">
              <xs:complexType>
                <xs:all minOccurs="1" maxOccurs="1">
                  <xs:element name="campaignId">
                    <xs:complexType>
                      <xs:simpleContent>
                        <xs:extension base="xs:string">
                          <xs:attribute name="type" type="xs:string">
                          </xs:attribute>
                        </xs:extension>
                      </xs:simpleContent>
                    </xs:complexType>
                  </xs:element>
                  <xs:element name="name">
                    <xs:complexType>
                      <xs:simpleContent>
                        <xs:extension base="xs:string">
                          <xs:attribute name="type" type="xs:string">
                          </xs:attribute>
                        </xs:extension>
                      </xs:simpleContent>
                    </xs:complexType>
                  </xs:element>
                  <xs:element name="archiveByDefault">
                    <xs:complexType>
                      <xs:simpleContent>
                        <xs:extension base="xs:string">
                          <xs:attribute name="type" type="xs:string">
                          </xs:attribute>
                        </xs:extension>
                      </xs:simpleContent>
                    </xs:complexType>
                  </xs:element>
                  <xs:any maxOccurs="unbounded" processContents="lax"/>
                </xs:all>
              </xs:complexType>
            </xs:element>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

(c) **XSLT**: We only extract *campaignId*, *name* and *archiveByDefault* from the operation response as shown below. Note how we transform the value of the *archiveByDefault* to our internal format.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:fn="http://www.w3.org/2005/xpath-functions" version="2.0">
  <xsl:output method="text" omit-xml-declaration="yes"
    cdata-section-elements="namelist"/>
    <xsl:template match="/">
      <xsl:for-each select="response/campaigns/campaign">
        <xsl:value-of select="campaignId"/>,
        <xsl:value-of select="name"/>,
        <xsl:if test="archiveByDefault = '0'">Active</xsl:if>
        <xsl:if test="archiveByDefault ='1'">Archived</xsl:if>
        <xsl:text>&#xa;</xsl:text>
      </xsl:for-each>
   </xsl:template>
</xsl:stylesheet>
```

7. **Mailchimpv1_3TotalCampaign**: This web service API operation of Mailchimp takes no input and gives the total number of campaigns. In order to support the pagination, our internal transformation makes sure that the count of campaigns is changed to (page number, page size) combination.

   (a) **LAV Mapping**:

$$Mailchimpv1\_3TotalCampaign^{ff}(pgno, pgsize) \leftarrow Page(pgno, '\text{ MailChimp v1.3 API}', '\text{ Mailchimpv1\_3Campaign}', pgsize). \tag{C.7}$$

   (b) **XSD**: Consider the web service operation defined using the LAVMapping C.7. We are interested in the following information

      i. Total Campaigns Count ($MCAPI/total$)

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="MCAPI">
    <xs:complexType>
      <xs:all>
        <xs:element name="total">
          <xs:complexType>
            <xs:simpleContent>
              <xs:extension base="xs:string">
                <xs:attribute name="type" type="xs:string">
                 </xs:attribute>
              </xs:extension>
            </xs:simpleContent>
          </xs:complexType>
        </xs:element>
        <xs:any maxOccurs="unbounded" processContents="lax"/>
      </xs:all>
      <xs:attribute type="xs:string" name="type"/>
    </xs:complexType>
```

```
  </xs:element>
</xs:schema>
```

(c) **XSLT**: Note how we transform the total number of entries to page number, entries per page combination. Refer our discussion in section C.2.1. We have set the default entries per page as 2.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:fn="http://www.w3.org/2005/xpath-functions" version="2.0">
  <xsl:output method="text" omit-xml-declaration="yes"
   cdata-section-elements="namelist"/>
  <xsl:template name="for-loop">
    <xsl:param name="total" select="1"/>
    <xsl:param name="increment" select="1"/>
    <xsl:param name="page" select="1"/>
    <xsl:if test="$total &gt; 0">
      <xsl:value-of select="$page"/>,
      <xsl:value-of select="$increment"/>
      <xsl:text>&#xa;</xsl:text>
      <xsl:call-template name="for-loop">
        <xsl:with-param name="total" select="$total - $increment"/>
        <xsl:with-param name="increment" select="$increment"/>
        <xsl:with-param name="page" select="$page+1"/>
       </xsl:call-template>
    </xsl:if>
  </xsl:template>
  <xsl:template match="/">
    <xsl:variable name="default">2</xsl:variable>
    <xsl:variable name="page">
      <xsl:copy-of select="$default"/>
    </xsl:variable>
    <xsl:variable name="total">
      <xsl:value-of select="MCAPI/total"/>
    </xsl:variable>
    <xsl:call-template name="for-loop">
      <xsl:with-param name="total" select="$total"/>
      <xsl:with-param name="increment" select="$default"/>
      <xsl:with-param name="page" select="0"/>
    </xsl:call-template>
  </xsl:template>
</xsl:stylesheet>
```

8. **Mailchimpv1_3Campaign**: This operation takes as input (page number, page size) combination and gives the details of all the campaigns. The list of (page number, page size) combination is obtained from the operation considered in the LAV mapping C.7. The outputs include the campaign identifier *cmid*, name *cmname*, creation date *cmcdate* and status *cmstatus*.

   (a) **LAV Mapping**:

$$Mailchimpv1\_3Campaign^{bbffff}(pgno, pgsize, cmid, cmname, cmcdate, cmstatus) \leftarrow Page(pgno,' \text{MailChimp v1.3 API}', \text{'Mailchimpv1\_3Campaign}', pgsize), Campaign(cmid,' \text{MailChimp v1.3 API}', cmname, cmcdate, cmstatus). \quad (C.8)$$

(b) **XSD**: Consider the web service operation defined using the LAVMapping C.8. We are interested in the following elements (XPath)

i. Campaign identifier $MCAPI/data/struct/id$
ii. Campaign name $MCAPI/data/struct/title$.
iii. Campaign Creation date $MCAPI/data/struct/create\_time$.
iv. Campaign status $MCAPI/data/struct/status$.

Note how we created a new data type *campaignStatus* to validate the allowed values of the campaign status.

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:simpleType name="campaignStatus">
    <xs:restriction base="xs:string">
      <xs:enumeration value="sent"/>
      <xs:enumeration value="save"/>
      <xs:enumeration value="paused"/>
      <xs:enumeration value="schedule"/>
      <xs:enumeration value="sending"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:element name="MCAPI">
    <xs:complexType>
      <xs:all>
        <xs:any maxOccurs="unbounded" processContents="lax"/>
        <xs:element name="data">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="struct" maxOccurs="unbounded" minOccurs="0">
                <xs:complexType>
                  <xs:all minOccurs="1" maxOccurs="1">
                    <xs:element name="id">
                      <xs:complexType>
                        <xs:simpleContent>
                          <xs:extension base="xs:string">
                            <xs:attribute name="type" type="xs:string">
                            </xs:attribute>
                          </xs:extension>
                        </xs:simpleContent>
                      </xs:complexType>
                    </xs:element>
                    <xs:element name="title">
                      <xs:complexType>
                        <xs:simpleContent>
                          <xs:extension base="xs:string">
                            <xs:attribute name="type" type="xs:string">
                            </xs:attribute>
                          </xs:extension>
                        </xs:simpleContent>
                      </xs:complexType>
                    </xs:element>
```

```
                    <xs:element name="create_time">
                      <xs:complexType>
                        <xs:simpleContent>
                          <xs:extension base="xs:string">
                            <xs:attribute name="type" type="xs:string">
                            </xs:attribute>
                          </xs:extension>
                        </xs:simpleContent>
                      </xs:complexType>
                    </xs:element>
                    <xs:element name="status">
                      <xs:complexType>
                        <xs:simpleContent>
                          <xs:extension base="campaignStatus">
                            <xs:attribute name="type" type="xs:string">
                            </xs:attribute>
                          </xs:extension>
                        </xs:simpleContent>
                      </xs:complexType>
                    </xs:element>
                    <xs:any maxOccurs="unbounded" processContents="lax"/>
                  </xs:all>
                  <xs:attribute type="xs:string" name="type"/>
                  <xs:attribute type="xs:string" name="key"/>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
            <xs:attribute type="xs:string" name="type"/>
          </xs:complexType>
        </xs:element>
      </xs:all>
      <xs:attribute type="xs:string" name="type"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

(c) **XSLT**: We are interested in extracting the *id*, *title*, *create_time* and *status* from the operation response as shown below. Note how we transform the campaign status to our desired internal format using various *if* conditions.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:fn="http://www.w3.org/2005/xpath-functions" version="2.0">
  <xsl:output method="text" omit-xml-declaration="yes"
    cdata-section-elements="namelist"/>
  <xsl:template match="/">
    <xsl:for-each select="MCAPI/data/struct">
      <xsl:value-of select="id"/>,
      <xsl:value-of select="title"/>,
      <xsl:value-of select="fn:substring(current()/create_time,1,10)"/>,
      <xsl:if test="status ='sent'">Sent</xsl:if>
      <xsl:if test="status ='save'">Scheduled</xsl:if>
      <xsl:if test="status ='paused'">Scheduled</xsl:if>
      <xsl:if test="status ='sending'">Sent</xsl:if>
      <xsl:text>&#xa;</xsl:text>
```

```
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

9. **Mailchimpv1_3CampaignStatistics**: This operation takes as input the campaign identifier and gives the campaign statistics: campaign abuse count *cmar*, click count *cmctr*, forward count *cmfr* and bounce count *cmbr*.

   (a) **LAV Mapping**:

$$\begin{aligned} &Mailchimpv1\_3CampaignStatistics^{bfffff}(cmid, cmar, cmctr, cmfr, cmbr) \leftarrow \\ &\quad Campaign(cmid,' \text{ MailChimp v1.3 API}', cmname, cmcdate, cmstatus), \\ &\quad CampaignStatistics(cmid,' \text{ MailChimp v1. 3 API}', cmar, cmctr, cmfr, cmbr). \end{aligned} \tag{C.9}$$

   (b) **XSD**: Consider the web service operation defined using the LAVMapping C.9. We are interested in the following elements (XPath)

      i. Campaign Abuse Reports $MCAPI/abuse\_reports$
      ii. Campaign Click Count $MCAPI/clicks$.
      iii. Campaign Forwards Count $MCAPI/forwards$.
      iv. Campaign Hard Bounces $MCAPI/hard\_bounces$.
      v. Campaign Soft Bounces $MCAPI/soft\_bounces$.

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="MCAPI">
    <xs:complexType>
      <xs:all minOccurs="1" maxOccurs="1">
        <xs:element name="abuse_reports">
          <xs:complexType>
            <xs:simpleContent>
              <xs:extension base="xs:double">
                <xs:attribute name="type" type="xs:string">
                </xs:attribute>
              </xs:extension>
            </xs:simpleContent>
          </xs:complexType>
        </xs:element>
        <xs:element name="clicks">
          <xs:complexType>
            <xs:simpleContent>
              <xs:extension base="xs:double">
                <xs:attribute name="type" type="xs:string">
                </xs:attribute>
              </xs:extension>
            </xs:simpleContent>
          </xs:complexType>
        </xs:element>
        <xs:element name="forwards">
          <xs:complexType>
            <xs:simpleContent>
              <xs:extension base="xs:double">
                <xs:attribute name="type" type="xs:string">
                </xs:attribute>
              </xs:extension>
            </xs:simpleContent>
```

```
          </xs:complexType>
        </xs:element>
        <xs:element name="hard_bounces">
          <xs:complexType>
            <xs:simpleContent>
              <xs:extension base="xs:double">
                <xs:attribute name="type" type="xs:string">
                </xs:attribute>
              </xs:extension>
            </xs:simpleContent>
          </xs:complexType>
        </xs:element>
        <xs:element name="soft_bounces">
          <xs:complexType>
            <xs:simpleContent>
              <xs:extension base="xs:double">
                <xs:attribute name="type" type="xs:string">
                </xs:attribute>
              </xs:extension>
            </xs:simpleContent>
          </xs:complexType>
        </xs:element>
        <xs:any maxOccurs="unbounded" processContents="lax"/>
      </xs:all>
       <xs:attribute type="xs:string" name="type"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

(c) **XSLT**: We are interested in extracting the *abuse_reports*, *clicks*, *forwards*, *soft_bounces* and *hard_bounces* from the operation response as shown below. Note how we sum up the count of soft bounces and hard bounces to get the desired bounce count.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:fn="http://www.w3.org/2005/xpath-functions" version="2.0">
  <xsl:output method="text" omit-xml-declaration="yes"
    cdata-section-elements="namelist"/>
  <xsl:template match="/">
    <xsl:for-each select="MCAPI">
      <xsl:value-of select="abuse_reports"/>,
      <xsl:value-of select="clicks"/>,
      <xsl:value-of select="forwards"/>,
      <xsl:value-of select="soft_bounces+hard_bounces"/>
      <xsl:text>&#xa;</xsl:text>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

10. **Basecampv1Projects**: This operation gives the details of all the projects: project identifier *pid*, name *pname*, status *pstatus* and creation date *pcdate*.

    (a) **LAV Mapping**:

$$Basecampv1Projects^{ffff}(pid, pname, pstatus, pcdate) \leftarrow Project(pid,' \text{Basecamp API}', pname, pcdate, pstatus). \tag{C.10}$$

(b) **XSD**: Consider the web service operation defined using the LAVMapping C.10. We are interested in the following elements (XPath)

i. Project identifier $json/array/id$
ii. Project name $json/array/name$.
iii. Project Creation date $json/array/created\_at$.
iv. Project status $json/array/archived$.

Note how the root element is json. This is because we convert the json response to xml response. During this conversion, the desired root element is json.

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="json">
    <xs:complexType>
     <xs:sequence>
      <xs:element name="array" maxOccurs="unbounded" minOccurs="0">
        <xs:complexType>
          <xs:all minOccurs="1" maxOccurs="1">
            <xs:element name="id">
              <xs:complexType>
                <xs:simpleContent>
                  <xs:extension base="xs:string">
                    <xs:attribute name="type" type="xs:string">
                    </xs:attribute>
                  </xs:extension>
                </xs:simpleContent>
              </xs:complexType>
            </xs:element>
            <xs:element name="name">
              <xs:complexType>
                <xs:simpleContent>
                  <xs:extension base="xs:string">
                    <xs:attribute name="type" type="xs:string">
                    </xs:attribute>
                  </xs:extension>
                </xs:simpleContent>
              </xs:complexType>
            </xs:element>
            <xs:element name="created_at">
              <xs:complexType>
                <xs:simpleContent>
                  <xs:extension base="xs:dateTime">
                    <xs:attribute name="type" type="xs:string">
                    </xs:attribute>
                  </xs:extension>
                </xs:simpleContent>
              </xs:complexType>
            </xs:element>
            <xs:element name="archived">
              <xs:complexType>
                <xs:simpleContent>
```

```
                <xs:extension base="xs:boolean">
                  <xs:attribute name="type" type="xs:string">
                  </xs:attribute>
                </xs:extension>
              </xs:simpleContent>
            </xs:complexType>
          </xs:element>
          <xs:any maxOccurs="unbounded" processContents="lax"/>
        </xs:all>
        <xs:attribute type="xs:string" name="class"/>
      </xs:complexType>
    </xs:element>
   </xs:sequence>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

(c) **XSLT**: We are interested in extracting the *id*, *name*, *created_at* and *archived* from the operation response as shown below. Note how we tranform the boolean *archived* to the desired internal format.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xs="http://www.w3.org/2001/XMLSchema"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 xmlns:fn="http://www.w3.org/2005/xpath-functions" version="2.0">
  <xsl:output method="text" omit-xml-declaration="yes"
   cdata-section-elements="namelist"/>
    <xsl:template match="/">
      <xsl:for-each select="json/array">
        <xsl:value-of select="id"/>,
      <xsl:value-of select="name"/>,
      <xsl:if test="archived = 'false'">Active</xsl:if>
      <xsl:if test="archived ='true'">Archived</xsl:if>,
      <xsl:value-of select="fn:substring(current()/created_at,1,10)"/>
      <xsl:text>&#xa;</xsl:text>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

11. **Basecampv1TodoLists**: This operation takes as input the project identifiers *pid* and gives the todo list identifiers *tid* present in the corresponding project identified by the project identifier.

    (a) **LAV Mapping**:

$$Basecampv1TodoLists^{bf}(pid, tlid) \leftarrow TaskList(pid, \text{'Basecamp API'}, tlid). \tag{C.11}$$

    (b) **XSD**: Consider the web service operation defined using the LAVMapping C.11. We are interested in the following element (XPath)

        i. Todo List identifier $json/array/id$

    Note how the root element is json. This is because we convert the json response to xml response. During this conversion, the desired root element

is json.

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="json">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="array" maxOccurs="unbounded" minOccurs="0">
          <xs:complexType>
            <xs:all minOccurs="1" maxOccurs="1">
              <xs:element name="id">
                <xs:complexType>
                  <xs:simpleContent>
                    <xs:extension base="xs:string">
                      <xs:attribute name="type" type="xs:string">
                      </xs:attribute>
                    </xs:extension>
                  </xs:simpleContent>
                </xs:complexType>
              </xs:element>
              <xs:any maxOccurs="unbounded" processContents="lax"/>
            </xs:all>
            <xs:attribute type="xs:string" name="class"/>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

(c) **XSLT**: We are interested in extracting the *id* from the operation response as shown below.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:fn="http://www.w3.org/2005/xpath-functions" version="2.0">
  <xsl:output method="text" omit-xml-declaration="yes"
    cdata-section-elements="namelist"/>
  <xsl:template match="/">
    <xsl:for-each select="json/array">
      <xsl:value-of select="id"/>
      <xsl:text>&#xa;</xsl:text>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

12. **Basecampv1Tasks**: This operation takes as input the project identifier *pid* and todo list identifier *tlid* and gives as output the tasks present in the given todolist. It gives the following details like task identifer *tid*, name *tname*, status *tstatus* and creation date *tcdate*.

    (a) **LAV Mapping**:

$$Basecampv1Tasks^{bbfffff}(pid, tlid, tid, tname, tstatus, tcdate) \leftarrow \\ TaskList(pid,' \text{ Basecamp API}', tlid), \\ Task(tlid, tid,' \text{ Basecamp API}', tname, tcdate, tddate, tcmpdate, tstatus). \tag{C.12}$$

(b) **XSD**: Consider the web service operation defined using the LAVMapping C.12. We are interested in the following elements (XPath)

i. Taskidentifier ($json/todos/remaining/array/id$)
ii. Task name ($json/todos/remaining/array/content$).
iii. Task Creation date ($json/todos/remaining/array/created\_at$).
iv. Task status ($json/todos/remaining/array/completed$).

Note how the root element is json. This is because we convert the json response to xml response. During this conversion, the desired root element is json.

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="json">
    <xs:complexType>
      <xs:all minOccurs="1" maxOccurs="1">
        <xs:element name="todos" maxOccurs="unbounded" minOccurs="0">
          <xs:complexType>
            <xs:all minOccurs="1" maxOccurs="1">
              <xs:element name="remaining" maxOccurs="unbounded" minOccurs="0">
                <xs:complexType>
                  <xs:all minOccurs="1" maxOccurs="1">
                    <xs:element name="array" maxOccurs="unbounded" minOccurs="0">
                      <xs:complexType>
                        <xs:all minOccurs="1" maxOccurs="1">
                          <xs:element name="id">
                            <xs:complexType>
                              <xs:simpleContent>
                                <xs:extension base="xs:string">
                                  <xs:attribute name="type" type="xs:string">
                                  </xs:attribute>
                                </xs:extension>
                              </xs:simpleContent>
                            </xs:complexType>
                          </xs:element>
                          <xs:element name="content">
                            <xs:complexType>
                              <xs:simpleContent>
                                <xs:extension base="xs:string">
                                  <xs:attribute name="type" type="xs:string">
                                  </xs:attribute>
                                </xs:extension>
                              </xs:simpleContent>
                            </xs:complexType>
                          </xs:element>
                          <xs:element name="created_at">
                            <xs:complexType>
                              <xs:simpleContent>
                                <xs:extension base="xs:string">
                                  <xs:attribute name="type" type="xs:string">
                                  </xs:attribute>
```

```
                                </xs:extension>
                              </xs:simpleContent>
                            </xs:complexType>
                          </xs:element>
                          <xs:element name="completed">
                            <xs:complexType>
                              <xs:simpleContent>
                                <xs:extension base="xs:boolean">
                                  <xs:attribute name="type" type="xs:string">
                                  </xs:attribute>
                                </xs:extension>
                              </xs:simpleContent>
                            </xs:complexType>
                          </xs:element>
                          <xs:any maxOccurs="unbounded" processContents="lax"/>
                        </xs:all>
                        <xs:attribute type="xs:string" name="class"/>
                      </xs:complexType>
                    </xs:element>
                  </xs:all>
                  <xs:attribute type="xs:string" name="class"/>
                </xs:complexType>
              </xs:element>
              <xs:any maxOccurs="unbounded" processContents="lax"/>
            </xs:all>
            <xs:attribute type="xs:string" name="class"/>
          </xs:complexType>
        </xs:element>
        <xs:any maxOccurs="unbounded" processContents="lax"/>
      </xs:all>
      <xs:attribute type="xs:string" name="type"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

(c) **XSLT**: We are interested in extracting the *id*, *content*, *created_at* and *completed* from the operation response as shown below. Note how we transform the boolean *completed* to our desired internal format.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:fn="http://www.w3.org/2005/xpath-functions" version="2.0">
  <xsl:output method="text" omit-xml-declaration="yes"
    cdata-section-elements="namelist"/>
  <xsl:template match="/">
    <xsl:for-each select="json/todos/remaining/array">
      <xsl:value-of select="id"/>,
      <xsl:value-of select="content"/>,
      <xsl:if test="completed = 'false'">Open</xsl:if>
      <xsl:if test="completed ='true'">Completed</xsl:if>,
      <xsl:value-of select="fn:substring(current()/created_at,1,10)"/>
      <xsl:text>&#xa;</xsl:text>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

13. **Basecampv1CompletedTasks**: This operation takes as input the project identifier *pid* and todo list identifier *tlid* and gives as output the completed tasks present in the given todolist. It gives the following details like task identifer *tid*, name *tname*, status *tstatus* and creation date *tcdate*.

    (a) **LAV Mapping**:

$$Basecampv1CompletedTasks^{bbfffff}(pid, tlid, tid, tname, tcdate, tcmpdate) \leftarrow TaskList(pid, '\text{ Basecamp API}', tlid), \quad Task(tlid, tid, '\text{ Basecamp API}', tname, tcdate, tddate, tcmpdate, '\text{ Completed}'). \tag{C.13}$$

    (b) **XSD**: Consider the web service operation defined using the LAVMapping C.13. We are interested in the following elements (XPath)

    i. Task identifier ($json/todos/completed/array/id$)
    ii. Task name ($json/todos/completed/array/content$)
    iii. Task Creation date ($json/todos/completed/array/created\_at$)
    iv. Task Completion date ($json/todos/completed/array/completed\_at$)

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="json">
    <xs:complexType>
      <xs:all minOccurs="1" maxOccurs="1">
        <xs:element name="todos" maxOccurs="unbounded" minOccurs="0">
          <xs:complexType>
            <xs:all minOccurs="1" maxOccurs="1">
              <xs:element name="completed" maxOccurs="unbounded" minOccurs="0">
                <xs:complexType>
                  <xs:all minOccurs="1" maxOccurs="1">
                    <xs:element name="array" maxOccurs="unbounded" minOccurs="0">
                      <xs:complexType>
                        <xs:all minOccurs="1" maxOccurs="1">
                          <xs:element name="id">
                            <xs:complexType>
                              <xs:simpleContent>
                                <xs:extension base="xs:string">
                                  <xs:attribute name="type" type="xs:string">
                                  </xs:attribute>
                                </xs:extension>
                              </xs:simpleContent>
                            </xs:complexType>
                          </xs:element>
                          <xs:element name="content">
                            <xs:complexType>
                              <xs:simpleContent>
                                <xs:extension base="xs:string">
                                  <xs:attribute name="type" type="xs:string">
                                  </xs:attribute>
                                </xs:extension>
                              </xs:simpleContent>
                            </xs:complexType>
                          </xs:element>
                          <xs:element name="created_at">
                            <xs:complexType>
                              <xs:simpleContent>
```

```
                                <xs:extension base="xs:string">
                                  <xs:attribute name="type" type="xs:string">
                                  </xs:attribute>
                                </xs:extension>
                              </xs:simpleContent>
                            </xs:complexType>
                          </xs:element>
                          <xs:element name="completed_at">
                            <xs:complexType>
                              <xs:simpleContent>
                                <xs:extension base="xs:string">
                                  <xs:attribute name="type" type="xs:string">
                                  </xs:attribute>
                                </xs:extension>
                              </xs:simpleContent>
                            </xs:complexType>
                          </xs:element>
                          <xs:any maxOccurs="unbounded" processContents="lax"/>
                        </xs:all>
                        <xs:attribute type="xs:string" name="class"/>
                      </xs:complexType>
                    </xs:element>
                  </xs:all>
                  <xs:attribute type="xs:string" name="class"/>
                </xs:complexType>
              </xs:element>
              <xs:any maxOccurs="unbounded" processContents="lax"/>
            </xs:all>
            <xs:attribute type="xs:string" name="class"/>
          </xs:complexType>
        </xs:element>
        <xs:any maxOccurs="unbounded" processContents="lax"/>
      </xs:all>
      <xs:attribute type="xs:string" name="type"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

(c) **XSLT**: We are interested in extracting the *id*, *content*, *created_at* and *completed_at* from the operation response as shown below.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xs="http://www.w3.org/2001/XMLSchema"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 xmlns:fn="http://www.w3.org/2005/xpath-functions" version="2.0">
  <xsl:output method="text" omit-xml-declaration="yes"
   cdata-section-elements="namelist"/>
    <xsl:template match="/">
      <xsl:for-each select="json/todos/completed/array">
       <xsl:value-of select="id"/>,
       <xsl:value-of select="content"/>,
       <xsl:value-of select="fn:substring(current()/created_at,1,10)"/>,
       <xsl:value-of select="fn:substring(current()/completed_at,1,10)"/>
       <xsl:text>&#xa;</xsl:text>
     </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

14. **LiquidPlannerv3_0_0Projects**: This operation gives the details of all the projects: project identifier *pid*, name *pname*, status *pstatus* and creation date *pcdate*.

    (a) **LAV Mapping**:

$$LiquidPlannerv3\_0\_0Projects^{ffff}(pid, pname, pstatus, pcdate) \leftarrow Project(pid,' \text{ LiquidPlanner API}', pname, pcdate, pstatus). \tag{C.14}$$

    (b) **XSD**: Consider the web service operation defined using the LAVMapping C.14. We are interested in the following elements (XPath)

        i. Project identifier ($json/array/id$)
        ii. Project name ($json/array/name$)
        iii. Project Creation date ($json/array/created\_at$)
        iv. Project status ($json/array/is\_on\_hold$)

    Note how the root element is json. This is because we convert the json response to xml response. During this conversion, the desired root element is json.

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="json">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="array" maxOccurs="unbounded" minOccurs="0">
          <xs:complexType>
            <xs:all minOccurs="1" maxOccurs="1">
              <xs:element name="id">
                <xs:complexType>
                  <xs:simpleContent>
                    <xs:extension base="xs:string">
                      <xs:attribute name="type" type="xs:string">
                      </xs:attribute>
                    </xs:extension>
                  </xs:simpleContent>
                </xs:complexType>
              </xs:element>
              <xs:element name="name">
                <xs:complexType>
                  <xs:simpleContent>
                    <xs:extension base="xs:string">
                      <xs:attribute name="type" type="xs:string">
                      </xs:attribute>
                    </xs:extension>
                  </xs:simpleContent>
                </xs:complexType>
              </xs:element>
              <xs:element name="created_at">
                <xs:complexType>
                  <xs:simpleContent>
                    <xs:extension base="xs:string">
                      <xs:attribute name="type" type="xs:string"/>
                      <xs:attribute name="class" type="xs:string"/>
                    </xs:extension>
```

```
            </xs:simpleContent>
          </xs:complexType>
        </xs:element>
        <xs:element name="is_on_hold">
          <xs:complexType>
            <xs:simpleContent>
              <xs:extension base="xs:boolean">
                <xs:attribute name="type" type="xs:string">
                </xs:attribute>
              </xs:extension>
            </xs:simpleContent>
          </xs:complexType>
        </xs:element>
        <xs:any maxOccurs="unbounded" processContents="lax"/>
      </xs:all>
      <xs:attribute type="xs:string" name="class"/>
    </xs:complexType>
  </xs:element>
 </xs:sequence>
</xs:complexType>
</xs:element>
</xs:schema>
```

(c) **XSLT**: We are interested in extracting the *id*, *name*, *created_at* and *is_on_hold* from the operation response as shown below. Note how we tranform the boolean *is_on_hold* to our desired internal format.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:fn="http://www.w3.org/2005/xpath-functions" version="2.0">
  <xsl:output method="text" omit-xml-declaration="yes"
    cdata-section-elements="namelist"/>
  <xsl:template match="/">
    <xsl:for-each select="json/array">
      <xsl:value-of select="id"/>,
      <xsl:value-of select="name"/>,
      <xsl:if test="is_on_hold = 'false'">Active</xsl:if>
      <xsl:if test="is_on_hold ='true'">OnHold</xsl:if>,
      <xsl:value-of select="fn:substring(current()/created_at,1,10)"/>
      <xsl:text>&#xa;</xsl:text>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

15. **LiquidPlannerv3__0__0Tasks**: This operation gives the details of all the tasks: task identifier *tid*, name *tname*, project identifier *pid*, status *tstatus*, creation date *tcdate* and due date *tddate*.

(a) **LAV Mapping**:

$$\begin{aligned} &LiquidPlannerv3\_0\_0Tasks^{fffffff}(tid, tname, pid, tstatus, tcdate, tddate) \leftarrow \\ &\quad Project(pid,' \text{LiquidPlanner API}', pname, pcdate, pstatus), \\ &\quad TaskList(pid,' \text{LiquidPlanner API}', tlid), \\ &\quad Task(tlid, tid,' \text{LiquidPlanner API}', tname, tcdate, tddate, tcmpdate, tstatus). \end{aligned} \tag{C.15}$$

(b) **XSD**: Consider the web service operation defined using the LAVMapping C.15. We are interested in the following elements (XPath)

i. Task identifier ($json/array/id$)
ii. Task name ($json/array/name$)
iii. Task Project identifier ($json/array/project\_id$)
iv. Task status ($json/array/is\_done$)
v. Task Creation date ($json/array/created\_at$)
vi. Task Due date ($json/array/expected\_finish$)

Note how the root element is json. This is because we convert the json response to xml response. During this conversion, the desired root element is json.

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="json">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="array" maxOccurs="unbounded" minOccurs="0">
          <xs:complexType>
            <xs:all minOccurs="1" maxOccurs="1">
              <xs:element name="id">
                <xs:complexType>
                  <xs:simpleContent>
                    <xs:extension base="xs:string">
                      <xs:attribute name="type" type="xs:string">
                      </xs:attribute>
                    </xs:extension>
                  </xs:simpleContent>
                </xs:complexType>
              </xs:element>
              <xs:element name="name">
                <xs:complexType>
                  <xs:simpleContent>
                    <xs:extension base="xs:string">
                      <xs:attribute name="type" type="xs:string">
                      </xs:attribute>
                    </xs:extension>
                  </xs:simpleContent>
                </xs:complexType>
              </xs:element>
              <xs:element name="created_at">
                <xs:complexType>
                  <xs:simpleContent>
                    <xs:extension base="xs:string">
                      <xs:attribute name="type" type="xs:string"/>
                      <xs:attribute name="class" type="xs:string"/>
                    </xs:extension>
                  </xs:simpleContent>
                </xs:complexType>
              </xs:element>
              <xs:element name="expected_finish">
                <xs:complexType>
                  <xs:simpleContent>
                    <xs:extension base="xs:string">
                      <xs:attribute name="type" type="xs:string"/>
```

```
                    <xs:attribute name="class" type="xs:string"/>
                  </xs:extension>
                </xs:simpleContent>
              </xs:complexType>
            </xs:element>
            <xs:element name="done_on">
              <xs:complexType>
                <xs:simpleContent>
                  <xs:extension base="xs:string">
                    <xs:attribute name="type" type="xs:string"/>
                    <xs:attribute name="class" type="xs:string"/>
                    <xs:attribute name="null" type="xs:string"/>
                  </xs:extension>
                </xs:simpleContent>
              </xs:complexType>
            </xs:element>
            <xs:element name="is_done">
              <xs:complexType>
                <xs:simpleContent>
                  <xs:extension base="xs:boolean">
                    <xs:attribute name="type" type="xs:string">
                    </xs:attribute>
                  </xs:extension>
                </xs:simpleContent>
              </xs:complexType>
            </xs:element>
            <xs:any maxOccurs="unbounded" processContents="lax"/>
          </xs:all>
          <xs:attribute type="xs:string" name="class"/>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
</xs:schema>
```

(c) **XSLT**: We are interested in extracting the *id*, *name*, *project_id*, *is_done* *created_at* and $expected_finish$ from the operation response as shown below. Note the transformation on the boolean *is_done* to our desired internal format.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:fn="http://www.w3.org/2005/xpath-functions" version="2.0">
  <xsl:output method="text" omit-xml-declaration="yes"
    cdata-section-elements="namelist"/>
  <xsl:template match="/">
    <xsl:for-each select="json/array">
      <xsl:value-of select="id"/>,
      <xsl:value-of select="name"/>,
      <xsl:value-of select="project_id"/>,
      <xsl:if test="is_done = 'false'">Open</xsl:if>
      <xsl:if test="is_done ='true'">Completed</xsl:if>,
      <xsl:value-of select="fn:substring(current()/created_at,1,10)"/>,
      <xsl:value-of select="fn:substring(current()/expected_finish,1,10)"/>
```

```
      <xsl:text> &#xa;</xsl:text>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

16. **TeamworkpmProjects**: This operation gives the details of all the projects: project identifier *pid*, name *pname*, status *pstatus* and creation date *pcdate*.

    (a) **LAV Mapping**:

$$TeamworkpmProjectAPI^{ffff}(pid, pname, pstatus, pcdate) \leftarrow Project(pid, '\text{ Teamworkpm API}', pname, pcdate, pstatus). \tag{C.16}$$

    (b) **XSD**: Consider the web service operation defined using the LAVMapping C.16. We are interested in the following elements (XPath)

    i. Project identifier (*projects/project/id*)
    ii. Project name (*projects/project/name*)
    iii. Project status (*projects/project/status*)
    iv. Project Creation date (*projects/project/created-on*)

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:simpleType name="projectStatus">
    <xs:restriction base="xs:string">
      <xs:enumeration value="active"/>
      <xs:enumeration value="on-hold"/>
      <xs:enumeration value="archived"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:element name="projects">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="project" maxOccurs="unbounded" minOccurs="0">
          <xs:complexType>
            <xs:all minOccurs="1" maxOccurs="1">
              <xs:element name="id">
                <xs:complexType>
                  <xs:simpleContent>
                    <xs:extension base="xs:string">
                      <xs:attribute name="type" type="xs:string">
                      </xs:attribute>
                    </xs:extension>
                  </xs:simpleContent>
                </xs:complexType>
              </xs:element>
              <xs:element name="name">
                <xs:complexType>
                  <xs:simpleContent>
                    <xs:extension base="xs:string">
                      <xs:attribute name="type" type="xs:string">
                      </xs:attribute>
                    </xs:extension>
                  </xs:simpleContent>
                </xs:complexType>
              </xs:element>
              <xs:element name="created-on">
```

```
                <xs:complexType>
                  <xs:simpleContent>
                    <xs:extension base="xs:dateTime">
                      <xs:attribute name="type" type="xs:string">
                      </xs:attribute>
                    </xs:extension>
                  </xs:simpleContent>
                </xs:complexType>
              </xs:element>
              <xs:element name="status">
                <xs:complexType>
                  <xs:simpleContent>
                    <xs:extension base="projectStatus">
                      <xs:attribute name="type" type="xs:string">
                      </xs:attribute>
                    </xs:extension>
                  </xs:simpleContent>
                </xs:complexType>
              </xs:element>
              <xs:any maxOccurs="unbounded" processContents="lax"/>
            </xs:all>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
      <xs:attribute type="xs:string" name="type"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

(c) **XSLT**: We are interested in extracting the *id*,*name*, *created-on* and *status* from the operation response as shown below. Note how we perform the case change on the *status*.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:fn="http://www.w3.org/2005/xpath-functions" version="2.0">
  <xsl:output method="text" omit-xml-declaration="yes"
    cdata-section-elements="namelist"/>
  <xsl:template match="/">
    <xsl:for-each select="projects/project">
      <xsl:value-of select="id"/>,
      <xsl:value-of select="name"/>,
      <xsl:if test="status = 'active'">Active</xsl:if>
      <xsl:if test="status ='archived'">Archived</xsl:if>,
      <xsl:value-of select="fn:substring(current()/created-on,1,10)"/>
      <xsl:text>&#xa;</xsl:text>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

17. **TeamworkpmTodoLists**: This operation takes as input the project identifiers *pid* and gives the todo list identifiers *tid* present in the corresponding project identified by the project identifier.

(a) **LAV Mapping**:

$$\begin{aligned} &TeamworkpmTodoLists^{bf}(pid, tlid) \leftarrow \\ &Project(pid,' \text{Teamworkpm API}', pname, pcdate, pstatus), \\ &\qquad TaskList(pid,' \text{Teamworkpm API}', tlid). \end{aligned} \tag{C.17}$$

(b) **XSD**: Consider the web service operation defined using the LAVMapping C.17. We are interested in the following element (XPath)

i. Todo List identifier *todo-lists/todo-list/id*

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="todo-lists">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="todo-list" maxOccurs="unbounded" minOccurs="0">
          <xs:complexType>
            <xs:all minOccurs="1" maxOccurs="1">
              <xs:element name="id">
                <xs:complexType>
                  <xs:simpleContent>
                    <xs:extension base="xs:string">
                      <xs:attribute name="type" type="xs:string">
                      </xs:attribute>
                    </xs:extension>
                  </xs:simpleContent>
                </xs:complexType>
              </xs:element>
              <xs:any maxOccurs="unbounded" processContents="lax"/>
            </xs:all>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
      <xs:attribute type="xs:string" name="type"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

(c) **XSLT**: We are interested in extracting the *id* from the operation response as shown below.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:fn="http://www.w3.org/2005/xpath-functions" version="2.0">
  <xsl:output method="text" omit-xml-declaration="yes"
    cdata-section-elements="namelist"/>
  <xsl:template match="/">
    <xsl:for-each select="todo-lists/todo-list">
      <xsl:value-of select="id"/>
      <xsl:text>&#xa;</xsl:text>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

18. **TeamworkpmTasks**: This operation takes as input the todo list identifier *tlid* and gives as output the tasks present in the given todolist. It gives the following details like task identifier*tid*, name *tname*, project identifier *pid*, status *tstatus*, creation date *tcdate* and task due date *tddate*.

    (a) **LAV Mapping**:

$$\begin{aligned} &TeamworkpmTasks^{bfffffff}(tlid, tid, tname, pid, tstatus, tcdate, tddate) \leftarrow \\ &\quad Project(pid,' \text{Teamworkpm API}', pname, pcdate, pstatus), \\ &\quad TaskList(pid,' \text{Teamworkpm API}', tlid), \\ &\quad Task(tlid, tid,' \text{Teamworkpm API}', tname, tcdate, tddate, tcmpdate, tstatus). \end{aligned} \tag{C.18}$$

    (b) **XSD**: Consider the web service operation defined using the LAVMapping C.18. We are interested in the following elements (XPath)

        i. Task identifier (*todo-items/todo-item/id*)
        ii. Task name (*todo-items/todo-item/content*)
        iii. Task Project identifier (*todo-items/todo-item/project-id*)
        iv. Task status (*todo-items/todo-item/completed*)
        v. Task Creation date (*todo-items/todo-item/start-date*)
        vi. Task Due date (*todo-items/todo-item/due-date*)

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="todo-items">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="todo-item" maxOccurs="unbounded" minOccurs="0">
          <xs:complexType>
            <xs:all minOccurs="1" maxOccurs="1">
              <xs:element name="id">
                <xs:complexType>
                  <xs:simpleContent>
                    <xs:extension base="xs:string">
                      <xs:attribute name="type" type="xs:string">
                      </xs:attribute>
                    </xs:extension>
                  </xs:simpleContent>
                </xs:complexType>
              </xs:element>
              <xs:element name="project-id">
                <xs:complexType>
                  <xs:simpleContent>
                    <xs:extension base="xs:string">
                      <xs:attribute name="type" type="xs:string">
                      </xs:attribute>
                    </xs:extension>
                  </xs:simpleContent>
                </xs:complexType>
              </xs:element>
              <xs:element name="content">
                <xs:complexType>
                  <xs:simpleContent>
                    <xs:extension base="xs:string">
                      <xs:attribute name="type" type="xs:string">
```

```
                    </xs:attribute>
                  </xs:extension>
                </xs:simpleContent>
              </xs:complexType>
            </xs:element>
            <xs:element name="start-date">
              <xs:complexType>
                <xs:simpleContent>
                  <xs:extension base="xs:string">
                    <xs:attribute name="type" type="xs:string">
                    </xs:attribute>
                  </xs:extension>
                </xs:simpleContent>
              </xs:complexType>
            </xs:element>
            <xs:element name="due-date">
              <xs:complexType>
                <xs:simpleContent>
                  <xs:extension base="xs:string">
                    <xs:attribute name="type" type="xs:string">
                    </xs:attribute>
                  </xs:extension>
                </xs:simpleContent>
              </xs:complexType>
            </xs:element>
            <xs:element name="completed">
              <xs:complexType>
                <xs:simpleContent>
                  <xs:extension base="xs:boolean">
                    <xs:attribute name="type" type="xs:string">
                    </xs:attribute>
                  </xs:extension>
                </xs:simpleContent>
              </xs:complexType>
            </xs:element>
            <xs:any maxOccurs="unbounded" processContents="lax"/>
          </xs:all>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
    <xs:attribute type="xs:string" name="type"/>
  </xs:complexType>
</xs:element>
</xs:schema>
```

(c) **XSLT**: We are interested in extracting the *id*, *content*, *project-id*, *completed*, *start-date* and *due-date* from the operation response as shown below. Note the transformation on the boolean *completed*. Also note how we transform the two dates (*start-date* and *due-date*) to a desired YYYY-MM-DD format.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:fn="http://www.w3.org/2005/xpath-functions" version="2.0">
  <xsl:output method="text" omit-xml-declaration="yes"
    cdata-section-elements="namelist"/>
```

```
  <xsl:template match="/">
    <xsl:for-each select="todo-items/todo-item">
      <xsl:value-of select="id"/>,
      <xsl:value-of select="content"/>,
      <xsl:value-of select="project-id"/>,
      <xsl:if test="completed = 'false'">Open</xsl:if>
      <xsl:if test="completed ='true'">Completed</xsl:if>,
      <xsl:value-of select="concat(fn:substring(current()/start-date,1,4),
        concat( concat('-', fn:substring(current()/start-date,5,2)),
        concat('-',fn:substring(current()/start-date,7,2))))"/>,
      <xsl:value-of select="concat(fn:substring(current()/due-date,1,4),
        concat( concat('-', fn:substring(current()/due-date,5,2)),
        concat('-',fn:substring(current()/due-date,7,2))))"/>
      <xsl:text>&#xa;</xsl:text>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

19. **ZohoProjectsProjects**: This operation gives the details of all the projects: project identifier *pid*, name *pname*, status *pstatus* and creation date *pcdate*.

    (a) **LAV Mapping**:

$$ZohoProjectsProjects^{ffff}(pid, pname, pstatus, pcdate) \leftarrow Project(pid, \text{'ZohoProjects API'}, pname, pcdate, pstatus). \tag{C.19}$$

    (b) **XSD**: Consider the web service operation defined using the LAVMapping C.19. We are interested in the following elements (XPath)

    i. Project identifier (*response/result/ProjectDetails/ProjectDetail/project_id*)
    ii. Project name (*response/result/ProjectDetails/ProjectDetail/project_name*)
    iii. Project status (*response/result/ProjectDetails/ProjectDetail/project_status*)
    iv. Project Creation date (*response/result/ProjectDetails/ProjectDetail/proj_created*)

```
<xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified" xmlns:xs="http://www.w3.org/2001/X
  <xs:element name="response">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="result">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="ProjectDetails">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="ProjectDetail" maxOccurs="unbounded" minOccurs="0">
                      <xs:complexType>
                        <xs:all>
                          <xs:element type="xs:string" name="project_id"/>
                          <xs:element type="xs:string" name="project_name"/>
                          <xs:element type="xs:string" name="project_status"/>
                          <xs:element type="xs:string" name="proj_created"/>
                          <xs:any maxOccurs="unbounded" processContents="lax"/>
                        </xs:all>
                      </xs:complexType>
                    </xs:element>
                  </xs:sequence>
                </xs:complexType>
```

```
            </xs:element>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element type="xs:string" name="uri"/>
    </xs:sequence>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

(c) **XSLT**: We are interested in extracting the *project_id*, *project_name*, *project_created* and *project_status* from the operation response as shown below. Note the transformation on the boolean *project_status* to convert to an internal format. Also note the tranformation on the date *project_created* to YYYY-MM-DD format.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:fn="http://www.w3.org/2005/xpath-functions" version="2.0">
  <xsl:output method="text" omit-xml-declaration="yes"
    cdata-section-elements="namelist"/>
  <xsl:template match="/">
    <xsl:for-each select="response/result/ProjectDetails/ProjectDetail">
      <xsl:value-of select="project_id"/>,
      <xsl:value-of select="project_name"/>,
      <xsl:if test="project_status ='active'">Active</xsl:if>
      <xsl:if test="project_status ='archived'">Archived</xsl:if>,
      <xsl:value-of select="concat(fn:substring(current()/proj_created,7,4),
        concat( concat('-', fn:substring(current()/proj_created,1,2)),
        concat('-',fn:substring(current()/proj_created,4,2))))"/>
      <xsl:text>&#xa;</xsl:text>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

20. **FreshdeskForumCategory**: This operation gives all the forum categories : their identifiers $fcid$ and names $fcname$.

(a) **LAV Mapping**:

$$FreshdeskForumCategory^{ff}(fcid, fcname) \leftarrow ForumCategory(fcid,' \text{Freshdesk API}', fcname). \tag{C.20}$$

(b) **XSD**: Consider the web service operation defined using the LAVMapping C.20. We are interested in the following elements (XPath)

i. Forum identifier ($forum\text{-}categories/forum\text{-}category/id$)
ii. Forum name ($forum\text{-}categories/forum\text{-}category/name$)

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="forum-categories">
    <xs:complexType>
      <xs:sequence>
```

```
        <xs:element name="forum-category" maxOccurs="unbounded" minOccurs="0">
          <xs:complexType>
            <xs:all minOccurs="1" maxOccurs="1">
              <xs:element name="id">
                <xs:complexType>
                  <xs:simpleContent>
                    <xs:extension base="xs:string">
                      <xs:attribute name="type" type="xs:string">
                      </xs:attribute>
                    </xs:extension>
                  </xs:simpleContent>
                </xs:complexType>
              </xs:element>
              <xs:element name="name">
                <xs:complexType>
                  <xs:simpleContent>
                    <xs:extension base="xs:string">
                      <xs:attribute name="type" type="xs:string">
                      </xs:attribute>
                    </xs:extension>
                  </xs:simpleContent>
                </xs:complexType>
              </xs:element>
              <xs:any maxOccurs="unbounded" processContents="lax"/>
            </xs:all>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
      <xs:attribute type="xs:string" name="count"/>
      <xs:attribute type="xs:string" name="type"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

(c) **XSLT**: We are interested in extracting the *id* and *name* from the operation response as shown below.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:fn="http://www.w3.org/2005/xpath-functions" version="2.0">
  <xsl:output method="text" omit-xml-declaration="yes"
   cdata-section-elements="namelist"/>
  <xsl:template match="/">
    <xsl:for-each select="forum-categories/forum-category">
      <xsl:value-of select="id"/>,
      <xsl:value-of select="name"/>
      <xsl:text>&#xa;</xsl:text>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

21. **FreshdeskForum**: This operation takes as input the forum category identifier $fcid$ (obtained by the operation considered in the LAV mapping C.20) and gives as output all the forums belonging to that category: with their identifiers fid

and name $fname$.

(a) **LAV Mapping**:

$$\begin{aligned} &FreshdeskForum^{bff}(fcid, fid, fname) \leftarrow \\ &ForumCategory(fcid,' \text{Freshdesk API}', fcname), \\ &\quad Forum(fid,' \text{Freshdesk API}', fname, fcdate). \end{aligned} \tag{C.21}$$

(b) **XSD**: Consider the web service operation defined using the LAVMapping C.21. We are interested in the following elements (XPath)

i. Forum identifier ($forum\text{-}category/forums/forum/id$)
ii. Forum name ($forum\text{-}category/forums/forum/title$)

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="forum-category">
    <xs:complexType>
      <xs:all>
        <xs:any maxOccurs="unbounded" processContents="lax"/>
        <xs:element name="forums">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="forum" maxOccurs="unbounded" minOccurs="0">
                <xs:complexType>
                  <xs:all minOccurs="1" maxOccurs="1">
                    <xs:element name="id">
                      <xs:complexType>
                        <xs:simpleContent>
                          <xs:extension base="xs:string">
                            <xs:attribute name="type" type="xs:string">
                            </xs:attribute>
                          </xs:extension>
                        </xs:simpleContent>
                      </xs:complexType>
                    </xs:element>
                    <xs:element name="name">
                      <xs:complexType>
                        <xs:simpleContent>
                          <xs:extension base="xs:string">
                            <xs:attribute name="type" type="xs:string">
                            </xs:attribute>
                          </xs:extension>
                        </xs:simpleContent>
                      </xs:complexType>
                    </xs:element>
                    <xs:any maxOccurs="unbounded" processContents="lax"/>
                  </xs:all>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
            <xs:attribute type="xs:string" name="count"/>
            <xs:attribute type="xs:string" name="type"/>
          </xs:complexType>
        </xs:element>
      </xs:all>
      <xs:attribute type="xs:string" name="count"/>
      <xs:attribute type="xs:string" name="type"/>
```

```
    </xs:complexType>
  </xs:element>
</xs:schema>
```

(c) **XSLT**: We are interested in extracting the *id* and *name* from the operation response as shown below.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:fn="http://www.w3.org/2005/xpath-functions" version="2.0">
  <xsl:output method="text" omit-xml-declaration="yes"
    cdata-section-elements="namelist"/>
  <xsl:template match="/">
    <xsl:for-each select="forum-category/forums/forum">
      <xsl:value-of select="id"/>,
      <xsl:value-of select="name"/>
      <xsl:text>&#xa;</xsl:text>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

22. **FreshdeskTopic**: This operation takes as input the forum category identifier $fcid$ and forum identifier fid and gives as output all the topics belonging to the forum. It gives as output the topic identifier *tpid*, name *tpname* and the creation date *tpcdate*.

(a) **LAV Mapping**:

$$\begin{aligned} FreshdeskTopic^{bbfff}(fcid, fid, tpid, tpname, tpcdate) \leftarrow \\ ForumCategory(fcid,' \text{Freshdesk API}', fcname), \\ Forum(fid,' \text{Freshdesk API}', fname, fcdate), \\ Topic(tpid,' \text{Freshdesk API}', tpname, tpcdate, fid). \end{aligned} \tag{C.22}$$

(b) **XSD**: Consider the web service operation defined using the LAVMapping C.22. We are interested in the following elements (XPath)

i. Topic identifier ($forum/topics/topic/id$)
ii. Topic name ($forum/topics/topic/title$)
iii. Topic Creation date ($forum/topics/topic/created\text{-}at$)

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:element name="forum">
    <xs:complexType>
      <xs:all>
        <xs:any maxOccurs="unbounded" processContents="lax"/>
        <xs:element name="topics">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="topic" maxOccurs="unbounded" minOccurs="0">
                <xs:complexType>
                  <xs:all minOccurs="1" maxOccurs="1">
                    <xs:element name="id">
                      <xs:complexType>
```

```
                    <xs:simpleContent>
                      <xs:extension base="xs:string">
                        <xs:attribute name="type" type="xs:string">
                        </xs:attribute>
                      </xs:extension>
                    </xs:simpleContent>
                  </xs:complexType>
                </xs:element>
                <xs:element name="title">
                  <xs:complexType>
                    <xs:simpleContent>
                      <xs:extension base="xs:string">
                        <xs:attribute name="type" type="xs:string">
                        </xs:attribute>
                      </xs:extension>
                    </xs:simpleContent>
                  </xs:complexType>
                </xs:element>
                <xs:element name="created-at">
                  <xs:complexType>
                    <xs:simpleContent>
                      <xs:extension base="xs:dateTime">
                        <xs:attribute name="type" type="xs:string">
                        </xs:attribute>
                      </xs:extension>
                    </xs:simpleContent>
                  </xs:complexType>
                </xs:element>
                <xs:any maxOccurs="unbounded" processContents="lax"/>
              </xs:all>
            </xs:complexType>
          </xs:element>
        </xs:sequence>
        <xs:attribute type="xs:string" name="count"/>
        <xs:attribute type="xs:string" name="type"/>
      </xs:complexType>
    </xs:element>
  </xs:all>
  <xs:attribute type="xs:string" name="count"/>
  <xs:attribute type="xs:string" name="type"/>
 </xs:complexType>
</xs:element>
</xs:schema>
```

(c) **XSLT**: We are interested in extracting the *id*, *title*, and *created-at* from the operation response as shown below.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:fn="http://www.w3.org/2005/xpath-functions" version="2.0">
  <xsl:output method="text" omit-xml-declaration="yes"
    cdata-section-elements="namelist"/>
  <xsl:template match="/">
    <xsl:for-each select="forum/topics/topic">
      <xsl:value-of select="id"/>,
      <xsl:value-of select="title"/>,
```

```
      <xsl:value-of select="fn:substring(current()/created-at,1,10)"/>
      <xsl:text>&#xa;</xsl:text>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

23. **FreshdeskTicket**: This operation gives all the open tickets giving the following details: ticket identifier *tkid*, name *tkname*, creation date *tkcdate*, due date *tkddate* and priority *tkpriority*.

    (a) **LAV Mapping**:

$$FreshdeskTicket^{fffff}(tkid, tkname, tkcdate, tkddate, tkpriority) \leftarrow Ticket(tkid, '\text{Freshdesk API}', tkname, tkcdate, tkddate, tkcmpdate, tkpriority, 'Open'). \tag{C.23}$$

    (b) **XSD**: Consider the web service operation defined using the LAVMapping C.23. We are interested in the following elements (XPath)

    i. Ticket identifier (*helpdesk-tickets/helpdesk-ticket/id*)
    ii. Ticket name (*helpdesk-tickets/helpdesk-ticket/subject*)
    iii. Ticket Creation date (*helpdesk-tickets/helpdesk-ticket/created-at*)
    iv. Ticket Due date (*helpdesk-tickets/helpdesk-ticket/due-by*)
    v. Ticket status (*helpdesk-tickets/helpdesk-ticket/status*)
    vi. Ticket priority (*helpdesk-tickets/helpdesk-ticket/priority*)

    Note how we created new data types *ticketStatus* and *ticketPriority*to validate the ticket status and ticket priority respectively.

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:simpleType name="ticketStatus">
    <xs:restriction base="xs:string">
      <xs:enumeration value="2"/>
      <xs:enumeration value="3"/>
      <xs:enumeration value="4"/>
      <xs:enumeration value="5"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="emptyDate">
    <xs:union memberTypes="xs:dateTime">
      <xs:simpleType>
        <xs:restriction base="xs:string">
          <xs:enumeration value=""/>
        </xs:restriction>
      </xs:simpleType>
    </xs:union>
  </xs:simpleType>
  <xs:simpleType name="ticketPriority">
    <xs:restriction base="xs:string">
      <xs:enumeration value="1"/>
      <xs:enumeration value="2"/>
      <xs:enumeration value="3"/>
      <xs:enumeration value="4"/>
    </xs:restriction>
  </xs:simpleType>
```

```
<xs:element name="helpdesk-tickets">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="helpdesk-ticket" maxOccurs="unbounded" minOccurs="0">
        <xs:complexType>
          <xs:all minOccurs="1" maxOccurs="1">
            <xs:element name="id">
              <xs:complexType>
                <xs:simpleContent>
                  <xs:extension base="xs:string">
                    <xs:attribute name="type" type="xs:string">
                     </xs:attribute>
                  </xs:extension>
                </xs:simpleContent>
              </xs:complexType>
            </xs:element>
            <xs:element name="subject">
              <xs:complexType>
                <xs:simpleContent>
                  <xs:extension base="xs:string">
                    <xs:attribute name="type" type="xs:string">
                     </xs:attribute>
                  </xs:extension>
                </xs:simpleContent>
              </xs:complexType>
            </xs:element>
            <xs:element name="created-at">
              <xs:complexType>
                <xs:simpleContent>
                  <xs:extension base="xs:dateTime">
                    <xs:attribute name="type" type="xs:string">
                     </xs:attribute>
                  </xs:extension>
                </xs:simpleContent>
              </xs:complexType>
            </xs:element>
            <xs:element name="due-by" nillable="true">
              <xs:complexType>
                <xs:simpleContent>
                  <xs:extension base="emptyDate">
                    <xs:attribute name="type" type="xs:string">
                     </xs:attribute>
                    <xs:attribute type="xs:string" name="nil"/>
                  </xs:extension>
                </xs:simpleContent>
              </xs:complexType>
            </xs:element>
            <xs:element name="status">
              <xs:complexType>
                <xs:simpleContent>
                  <xs:extension base="ticketStatus">
                    <xs:attribute name="type" type="xs:string">
                     </xs:attribute>
                  </xs:extension>
                </xs:simpleContent>
              </xs:complexType>
            </xs:element>
```

```
            <xs:element name="priority">
              <xs:complexType>
                <xs:simpleContent>
                  <xs:extension base="ticketPriority">
                    <xs:attribute name="type" type="xs:string">
                     </xs:attribute>
                  </xs:extension>
                </xs:simpleContent>
              </xs:complexType>
            </xs:element>
            <xs:any maxOccurs="unbounded" processContents="lax"/>
          </xs:all>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
    <xs:attribute type="xs:string" name="type"/>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

(c) **XSLT**: We are interested in extracting the *id*, *subject*, *created-at*, *due-by* and *priority* from the operation response as shown below. Note the transformation on the *priority* to the internal format.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:fn="http://www.w3.org/2005/xpath-functions" version="2.0">
  <xsl:output method="text" omit-xml-declaration="yes"
    cdata-section-elements="namelist"/>
  <xsl:template match="/">
    <xsl:for-each select="helpdesk-tickets/helpdesk-ticket">
      <xsl:value-of select="id"/>,
      <xsl:value-of select="subject"/>,
      <xsl:value-of select="fn:substring(current()/created-at,1,10)"/>,
      <xsl:value-of select="fn:substring(current()/due-by,1,10)"/>,
      <xsl:if test="priority ='1'">Low</xsl:if>
      <xsl:if test="priority ='2'">Medium</xsl:if>
      <xsl:if test="priority ='3'">High</xsl:if>
      <xsl:if test="priority ='4'">High</xsl:if>
      <xsl:text>&#xa;</xsl:text>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

24. **Uservoicev1TotalTickets**: This web service operation takes no input and gives the total number of tickets. In order to support the pagination, our internal transformation makes sure that the count of tickets is changed to (page number, page size) combination.

    (a) **LAV Mapping**:

$$Uservoicev1TotalTickets^{ff}(pgno, pgsize) \leftarrow Page(pgno, '\text{Uservoice v1 API}', '\text{Uservoicev1Ticket}', pgsize). \tag{C.24}$$

(b) **XSD**: Consider the web service operation defined using the LAVMapping C.24. We are interested in the following elements (XPath)

i. Total Tickets (*response/response_data/total_records*)

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="response">
    <xs:complexType>
      <xs:all>
        <xs:element name="response_data">
          <xs:complexType>
            <xs:all>
              <xs:element name="total_records">
                <xs:complexType>
                  <xs:simpleContent>
                    <xs:extension base="xs:string">
                      <xs:attribute name="type" type="xs:string">
                      </xs:attribute>
                    </xs:extension>
                  </xs:simpleContent>
                </xs:complexType>
              </xs:element>
              <xs:any maxOccurs="unbounded" processContents="lax"/>
            </xs:all>
          </xs:complexType>
        </xs:element>
        <xs:any maxOccurs="unbounded" processContents="lax"/>
      </xs:all>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

(c) **XSLT**: We are interested in extracting the page number, entries per page from the operation response as shown below. Refer our discussion on page handling (section C.2.1). The default number of entries per page is set as 25.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:fn="http://www.w3.org/2005/xpath-functions" version="2.0">
  <xsl:output method="text" omit-xml-declaration="yes"
    cdata-section-elements="namelist"/>
  <xsl:template name="for-loop">
    <xsl:param name="total" select="1"/>
    <xsl:param name="increment" select="1"/>
    <xsl:param name="page" select="1"/>
    <xsl:if test="$total &gt; 0">
      <xsl:value-of select="$page"/>,
      <xsl:value-of select="$increment"/>
      <xsl:text>&#xa;</xsl:text>
      <xsl:call-template name="for-loop">
        <xsl:with-param name="total" select="$total - $increment"/>
        <xsl:with-param name="increment" select="$increment"/>
        <xsl:with-param name="page" select="$page+1"/>
```

```
      </xsl:call-template>
    </xsl:if>
  </xsl:template>
  <xsl:template match="/">
    <xsl:variable name="default">25</xsl:variable>
    <xsl:variable name="page">
      <xsl:copy-of select="$default"/>
    </xsl:variable>
    <xsl:variable name="total">
      <xsl:value-of select="response/response_data/total_records"/>
    </xsl:variable>
    <xsl:call-template name="for-loop">
      <xsl:with-param name="total" select="$total"/>
      <xsl:with-param name="increment" select="$default"/>
      <xsl:with-param name="page" select="1"/>
    </xsl:call-template>
  </xsl:template>
</xsl:stylesheet>
```

25. **Uservoicev1Ticket**: This operation takes as input the (page number, page size) combination obtained from the operation considered in the LAV mapping C.24. It gives all the tickets with the following details: ticket identifier *tkid*, name *tkname*, creation date *tkcdate*, status *tkstatus* and priority *tkpriority*.

    (a) **LAV Mapping**:

$$\begin{aligned} &Uservoicev1Ticket^{bbffffff}(pgno, pgsize, tkid, tkname, tkcdate, tkstatus, tkpriority) \leftarrow \\ &\qquad Page(pgno, '\text{ Uservoice v1 API}', '\text{ Uservoicev1Ticket}', pgsize), \\ &\quad Ticket(tkid, '\text{ Uservoice v1 API}', tkname, tkcdate, tkddate, tkcmpdate, tkpriority, tkstatus). \end{aligned} \tag{C.25}$$

    (b) **XSD**: Consider the web service operation defined using the LAVMapping C.25. We are interested in the following elements (XPath)

        i. Ticket identifier (*response/tickets/ticket/id*)
        ii. Ticket name (*response/tickets/ticket/subject*)
        iii. Ticket Creation date (*response/tickets/ticket/created_at*)
        iv. Ticket status (*response/tickets/ticket/state*)

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="response">
    <xs:complexType>
      <xs:all>
        <xs:element name="tickets"  maxOccurs="unbounded" minOccurs="0">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="ticket"  maxOccurs="unbounded" minOccurs="0">
                <xs:complexType>
                  <xs:all>
                    <xs:element name="id">
                      <xs:complexType>
                        <xs:simpleContent>
                          <xs:extension base="xs:string">
                            <xs:attribute name="type" type="xs:string">
```

```
                    </xs:attribute>
                  </xs:extension>
                </xs:simpleContent>
              </xs:complexType>
            </xs:element>
            <xs:element name="subject">
              <xs:complexType>
                <xs:simpleContent>
                  <xs:extension base="xs:string">
                    <xs:attribute name="type" type="xs:string">
                    </xs:attribute>
                  </xs:extension>
                </xs:simpleContent>
              </xs:complexType>
            </xs:element>
            <xs:element name="state">
              <xs:complexType>
                <xs:simpleContent>
                  <xs:extension base="xs:string">
                    <xs:attribute name="type" type="xs:string">
                    </xs:attribute>
                  </xs:extension>
                </xs:simpleContent>
              </xs:complexType>
            </xs:element>
            <xs:element name="created_at">
              <xs:complexType>
                <xs:simpleContent>
                  <xs:extension base="xs:string">
                    <xs:attribute name="type" type="xs:string">
                    </xs:attribute>
                  </xs:extension>
                </xs:simpleContent>
              </xs:complexType>
            </xs:element>
            <xs:any maxOccurs="unbounded" processContents="lax"/>
          </xs:all>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
    <xs:attribute name="type" type="xs:string"/>
  </xs:complexType>
</xs:element>
<xs:any maxOccurs="unbounded" processContents="lax"/>
</xs:all>
</xs:complexType>
</xs:element>
</xs:schema>
```

(c) **XSLT**: We are interested in extracting the *id*, *subject*, *created_at* and *state* from the operation response as shown below. Note the transformation on *state* to the internal format.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
```

```
  xmlns:fn="http://www.w3.org/2005/xpath-functions" version="2.0">
  <xsl:output method="text" omit-xml-declaration="yes"
    cdata-section-elements="namelist"/>
  <xsl:template match="/">
    <xsl:for-each select="response/tickets/ticket">
      <xsl:value-of select="id"/>,
      <xsl:value-of select="subject"/>,
      <xsl:value-of select="fn:substring(current()/created_at,1,10)"/>,
      <xsl:if test="state ='open'">Open</xsl:if>
      <xsl:if test="state ='closed'">Closed</xsl:if>,Low
     <xsl:text>&#xa;</xsl:text>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

26. **Zendeskv2Forum**: This operation gives all the forums with the following details: forum identifier fid and name $fname$.

    (a) **LAV Mapping**:

$$Zendeskv2Forum^{ff}(fid, fname) \leftarrow Forum(fid,' \text{Zendesk v2 API}', fname, fcdate). \tag{C.26}$$

    (b) **XSD**: Consider the web service operation defined using the LAVMapping C.26. We are interested in the following elements (XPath)

        i. Forum identifier ($forums/forum/id$)
        ii. Forum name ($forums/forum/name$)

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="forums">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="forum" maxOccurs="unbounded" minOccurs="0">
          <xs:complexType>
            <xs:all minOccurs="1" maxOccurs="1">
              <xs:element name="id">
                <xs:complexType>
                  <xs:simpleContent>
                    <xs:extension base="xs:string">
                      <xs:attribute name="type" type="xs:string">
                      </xs:attribute>
                    </xs:extension>
                  </xs:simpleContent>
                </xs:complexType>
              </xs:element>
              <xs:element name="name">
                <xs:complexType>
                  <xs:simpleContent>
                    <xs:extension base="xs:string">
                      <xs:attribute name="type" type="xs:string">
                      </xs:attribute>
                    </xs:extension>
                  </xs:simpleContent>
                </xs:complexType>
              </xs:element>
```

```
            <xs:any maxOccurs="unbounded" processContents="lax"/>
          </xs:all>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
    <xs:attribute type="xs:string" name="count"/>
    <xs:attribute type="xs:string" name="type"/>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

(c) **XSLT**: We are interested in extracting the *id* and *name* from the operation response as shown below.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:fn="http://www.w3.org/2005/xpath-functions" version="2.0">
  <xsl:output method="text" omit-xml-declaration="yes"
    cdata-section-elements="namelist"/>
  <xsl:template match="/">
    <xsl:for-each select="forums/forum">
      <xsl:value-of select="id"/>,
      <xsl:value-of select="name"/>
      <xsl:text>&#xa;</xsl:text>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

27. **Zendeskv2Topic**: This operation takes as input the forum identifier fid and gives as output all the topics belonging to the forum. It gives as output the topic identifier *tpid*, name *tpname* and the creation date *tpcdate*.

(a) **LAV Mapping**:

$$\begin{aligned} &Zendeskv2Topic^{bfff}(fid, tpid, tpname, tpcdate) \leftarrow \\ &Topic(tpid, '\text{Zendesk v2 API}', tpname, tpcdate, fid), \\ &\quad Forum(fid, '\text{Zendesk v2 API}', fname, fcdate). \end{aligned} \tag{C.27}$$

(b) **XSD**: Consider the web service operation defined using the LAVMapping C.27. We are interested in the following elements (XPath)

i. Topic identifier (*entries/entry/id*)
ii. Topic name (*entries/entry/title*)
iii. Topic Creation date (*entries/entry/created-at*)

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="entries">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="entry" maxOccurs="unbounded" minOccurs="0">
          <xs:complexType>
            <xs:all minOccurs="1" maxOccurs="1">
              <xs:element name="id">
```

```
                <xs:complexType>
                  <xs:simpleContent>
                    <xs:extension base="xs:string">
                      <xs:attribute name="type" type="xs:string">
                      </xs:attribute>
                    </xs:extension>
                  </xs:simpleContent>
                </xs:complexType>
              </xs:element>
              <xs:element name="title">
                <xs:complexType>
                  <xs:simpleContent>
                    <xs:extension base="xs:string">
                      <xs:attribute name="type" type="xs:string">
                      </xs:attribute>
                    </xs:extension>
                  </xs:simpleContent>
                </xs:complexType>
              </xs:element>
              <xs:element name="created-at">
                <xs:complexType>
                  <xs:simpleContent>
                    <xs:extension base="xs:dateTime">
                      <xs:attribute name="type" type="xs:string">
                      </xs:attribute>
                    </xs:extension>
                  </xs:simpleContent>
                </xs:complexType>
              </xs:element>
              <xs:any maxOccurs="unbounded" processContents="lax"/>
            </xs:all>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
      <xs:attribute type="xs:string" name="count"/>
      <xs:attribute type="xs:string" name="type"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

(c) **XSLT**: We are interested in extracting the *id*, *title* and *created-at* from the operation response as shown below.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:fn="http://www.w3.org/2005/xpath-functions" version="2.0">
  <xsl:output method="text" omit-xml-declaration="yes"
    cdata-section-elements="namelist"/>
  <xsl:template match="/">
    <xsl:for-each select="entries/entry">
      <xsl:value-of select="id"/>,
      <xsl:value-of select="title"/>,
      <xsl:value-of select="fn:substring(current()/created-at,1,10)"/>
      <xsl:text>&#xa;</xsl:text>
    </xsl:for-each>
  </xsl:template>
```

```
</xsl:stylesheet>
```

28. **Zendeskv2Ticket**: This operation gives all the ticket identifiers *tkid*.

    (a) **LAV Mapping**:

$$Zendeskv2Ticket^{f}(tkid) \leftarrow Ticket(tkid,'\text{Zendesk v2 API}', tkname, tkcdate, tkddate, tkcmpdate, tkpriority, tkstatus). \tag{C.28}$$

    (b) **XSD**: Consider the web service operation defined using the LAVMapping C.28. We are interested in the following elements (XPath)

        i. Ticket identifier (*records/record/nice-id*)

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="records">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="record" maxOccurs="unbounded" minOccurs="0">
          <xs:complexType>
            <xs:all minOccurs="1" maxOccurs="1">
              <xs:element name="nice-id">
                <xs:complexType>
                  <xs:simpleContent>
                    <xs:extension base="xs:string">
                      <xs:attribute name="type" type="xs:string">
                      </xs:attribute>
                    </xs:extension>
                  </xs:simpleContent>
                </xs:complexType>
              </xs:element>
              <xs:any maxOccurs="unbounded" processContents="lax"/>
            </xs:all>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
      <xs:attribute type="xs:string" name="count"/>
      <xs:attribute type="xs:string" name="type"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

    (c) **XSLT**: We are interested in extracting the *nice-id* from the operation response as shown below.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:fn="http://www.w3.org/2005/xpath-functions" version="2.0">
  <xsl:output method="text" omit-xml-declaration="yes"
    cdata-section-elements="namelist"/>
  <xsl:template match="/">
    <xsl:for-each select="records/record">
      <xsl:value-of select="nice-id"/>
      <xsl:text>&#xa;</xsl:text>
```

```
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

29. **Zendeskv2SolvedTicket**: This operation gives all the solved ticket identifiers *tkid*.

    (a) **LAV Mapping**:

$$Zendeskv2SolvedTicket^{f}(tkid) \leftarrow Ticket(tkid,' \text{Zendesk v2 API}', tkname, tkcdate, tkddate, tkcmpdate, tkpriority,' \text{Closed}'). \tag{C.29}$$

    (b) **XSD**: Consider the web service operation defined using the LAVMapping C.29. We are interested in the following elements (XPath)

        i. Ticket identifier (*tickets/ticket/nice-id*)

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="tickets">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="ticket" maxOccurs="unbounded" minOccurs="0">
          <xs:complexType>
            <xs:all minOccurs="1" maxOccurs="1">
              <xs:element name="nice-id">
                <xs:complexType>
                  <xs:simpleContent>
                    <xs:extension base="xs:string">
                      <xs:attribute name="type" type="xs:string">
                      </xs:attribute>
                    </xs:extension>
                  </xs:simpleContent>
                </xs:complexType>
              </xs:element>
              <xs:any maxOccurs="unbounded" processContents="lax"/>
            </xs:all>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
      <xs:attribute type="xs:string" name="count"/>
      <xs:attribute type="xs:string" name="type"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

    (c) **XSLT**: We are interested in extracting the *nice-id* from the operation response as shown below.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:fn="http://www.w3.org/2005/xpath-functions" version="2.0">
  <xsl:output method="text" omit-xml-declaration="yes"
    cdata-section-elements="namelist"/>
```

```
  <xsl:template match="/">
    <xsl:for-each select="tickets/ticket">
      <xsl:value-of select="nice-id"/>
      <xsl:text>&#xa;</xsl:text>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

30. **Zendeskv2TicketDetails**: This operation takes as input the ticket identifers (obtained from the operations like the ones considered in the LAVMapping C.28 and C.29. It gives the ticket details: ticket name *tkname*, creation date *tkcdate*, due date *tkddate*, completion date *tkcmpdate*, status *tkstatus* and priority *tkpriority*.

    (a) **LAV Mapping**:

$$Zendeskv2TicketDetails^{bfffffff}(tkid, tkname, tkcdate, tkddate, tkcmpdate, tkpriority, tkstatus) \leftarrow Ticket(tkid,' \text{Zendesk v2 API}', tkname, tkcdate, tkddate, tkcmpdate, tkpriority, tkstatus). \tag{C.30}$$

    (b) **XSD**: Consider the web service operation defined using the LAVMapping C.30. We are interested in the following elements (XPath)

    i. Ticket identifier (*ticket/nice-id*)
    ii. Ticket name (*ticket/subject*)
    iii. Ticket Creation date (*ticket/created-at*)
    iv. Ticket Due date (*ticket/due-date*)
    v. Ticket Completion date (*ticket/solved-at*)
    vi. Ticket priority (*ticket/priority-id*)
    vii. Ticket status (*ticket/status-id*)

    Note how we created new data types *ticketStatus* and *ticketPriority*to validate the ticket status and ticket priority respectively.

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:simpleType name="ticketStatus">
    <xs:restriction base="xs:string">
      <xs:enumeration value="0"/>
      <xs:enumeration value="1"/>
      <xs:enumeration value="2"/>
      <xs:enumeration value="3"/>
      <xs:enumeration value="4"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="emptyDate">
    <xs:union memberTypes="xs:dateTime">
      <xs:simpleType>
        <xs:restriction base="xs:string">
          <xs:enumeration value=""/>
        </xs:restriction>
      </xs:simpleType>
    </xs:union>
  </xs:simpleType>
  <xs:simpleType name="ticketPriority">
    <xs:restriction base="xs:string">
```

```
    <xs:enumeration value="0"/>
    <xs:enumeration value="1"/>
    <xs:enumeration value="2"/>
    <xs:enumeration value="3"/>
    <xs:enumeration value="4"/>
  </xs:restriction>
</xs:simpleType>
<xs:element name="ticket">
  <xs:complexType>
    <xs:all minOccurs="1" maxOccurs="1">
      <xs:element name="nice-id">
        <xs:complexType>
          <xs:simpleContent>
            <xs:extension base="xs:string">
              <xs:attribute name="type" type="xs:string">
              </xs:attribute>
            </xs:extension>
          </xs:simpleContent>
        </xs:complexType>
      </xs:element>
      <xs:element name="subject">
        <xs:complexType>
          <xs:simpleContent>
            <xs:extension base="xs:string">
              <xs:attribute name="type" type="xs:string">
              </xs:attribute>
            </xs:extension>
          </xs:simpleContent>
        </xs:complexType>
      </xs:element>
      <xs:element name="created-at">
        <xs:complexType>
          <xs:simpleContent>
            <xs:extension base="xs:dateTime">
              <xs:attribute name="type" type="xs:string">
              </xs:attribute>
            </xs:extension>
          </xs:simpleContent>
        </xs:complexType>
      </xs:element>
      <xs:element name="solved-at" nillable="true">
        <xs:complexType>
          <xs:simpleContent>
            <xs:extension base="emptyDate">
              <xs:attribute name="type" type="xs:string">
              </xs:attribute>
              <xs:attribute type="xs:string" name="nil"/>
            </xs:extension>
          </xs:simpleContent>
        </xs:complexType>
      </xs:element>
      <xs:element name="due-date" nillable="true">
        <xs:complexType>
          <xs:simpleContent>
            <xs:extension base="emptyDate">
              <xs:attribute name="type" type="xs:string">
              </xs:attribute>
```

```
            <xs:attribute type="xs:string" name="nil"/>
          </xs:extension>
        </xs:simpleContent>
      </xs:complexType>
    </xs:element>
    <xs:element name="status-id">
      <xs:complexType>
        <xs:simpleContent>
          <xs:extension base="ticketStatus">
            <xs:attribute name="type" type="xs:string">
            </xs:attribute>
          </xs:extension>
        </xs:simpleContent>
      </xs:complexType>
    </xs:element>
    <xs:element name="priority-id">
      <xs:complexType>
        <xs:simpleContent>
          <xs:extension base="ticketPriority">
            <xs:attribute name="type" type="xs:string">
            </xs:attribute>
          </xs:extension>
        </xs:simpleContent>
      </xs:complexType>
    </xs:element>
    <xs:any maxOccurs="unbounded" processContents="lax"/>
  </xs:all>
 </xs:complexType>
</xs:element>
</xs:schema>
```

(c) **XSLT**: We are interested in extracting the *subject*, *created-at*, *due-date*, *solved-at*, *priority-id* and *status-id* from the operation response as shown below. Note the transformation on the *priority-id* and *status-id* to get the desired internal format.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:fn="http://www.w3.org/2005/xpath-functions" version="2.0">
  <xsl:output method="text" omit-xml-declaration="yes"
    cdata-section-elements="namelist"/>
  <xsl:template match="/">
    <xsl:for-each select="ticket">
      <xsl:value-of select="subject"/>,
      <xsl:value-of select="fn:substring(current()/created-at,1,10)"/>,
      <xsl:value-of select="fn:substring(current()/due-date,1,10)"/>,
      <xsl:value-of select="fn:substring(current()/solved-at,1,10)"/>,
      <xsl:if test="priority-id ='0'">Low</xsl:if>
      <xsl:if test="priority-id ='1'">Low</xsl:if>
      <xsl:if test="priority-id ='2'">Medium</xsl:if>
      <xsl:if test="priority-id ='3'">High</xsl:if>
      <xsl:if test="priority-id ='4'">High</xsl:if>,
      <xsl:if test="status-id ='0'">Open</xsl:if>
      <xsl:if test="status-id ='1'">Open</xsl:if>
      <xsl:if test="status-id ='2'">Closed</xsl:if>
```

```
      <xsl:if test="status-id ='3'">Closed</xsl:if>
      <xsl:if test="status-id ='4'">Closed</xsl:if>
      <xsl:text>&#xa;</xsl:text>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

31. **ZohoSupportTask**: This operation gives the details of all the tasks: ticket identifer *tkid*, name *tkname*, due date *tkddate*, status *tkstatus* and priority *tkpriority*.

    (a) **LAV Mapping**:

$$ZohoSupportTask^{fffff}(tkid, tkname, tkddate, tkpriority, tkstatus) \leftarrow Ticket(tkid,' \text{ZohoSupport API}', tkname, tkcdate, tkddate, tkcmpdate, tkpriority, tkstatus). \quad (C.31)$$

    (b) **XSD**: Consider the web service operation defined using the LAVMapping C.31. We are interested in the following elements (XPath)

    i. Ticket identifier ($response/result/Tasks/row/fl[@val =' ACTIVITYID']/node()$)
    ii. Ticket name ($response/result/Tasks/row/fl[@val =' Subject']/node()$)
    iii. Ticket Due date ($response/result/Tasks/row/fn : substring(fl[@val =' DueDate']/node()$)
    iv. Ticket Priority ($response/result/Tasks/row/fl[@val =' Priority']/node()$)
    v. Ticket Status ($response/result/Tasks/row/fl[@val =' Status']/node()$)

    Note how we created new data types *ticketStatus* and *ticketPriority*to validate the ticket status and ticket priority respectively.

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="response">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="result">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="Tasks">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="row" maxOccurs="unbounded" minOccurs="0">
                      <xs:complexType>
                        <xs:sequence>
                          <xs:element name="fl" maxOccurs="unbounded" minOccurs="0">
                            <xs:complexType>
                              <xs:simpleContent>
                                <xs:extension base="xs:string">
                                  <xs:attribute type="xs:string" name="val" use="optional"/>
                                </xs:extension>
                              </xs:simpleContent>
                            </xs:complexType>
                          </xs:element>
                        </xs:sequence>
                        <xs:attribute type="xs:byte" name="no" use="optional"/>
                      </xs:complexType>
                    </xs:element>
```

```
                </xs:sequence>
              </xs:complexType>
            </xs:element>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
    <xs:attribute type="xs:string" name="uri"/>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

(c) **XSLT**: Note how we extract the desired information from the operation response and how we transform the status and priority to the desired internal format.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:fn="http://www.w3.org/2005/xpath-functions" version="2.0">
  <xsl:output method="text" omit-xml-declaration="yes"
    cdata-section-elements="namelist"/>
  <xsl:template match="/">
    <xsl:for-each select="response/result/Tasks/row">
      <xsl:value-of select="fl[@val = 'ACTIVITYID']/node()"/>,
      <xsl:value-of select="fl[@val = 'Subject']/node()"/>,
      <xsl:value-of select="fn:substring(fl[@val = 'Due Date']/node(),1,10)"/>,
      <xsl:if test="fl[@val = 'Priority']/node() ='Highest'">High</xsl:if>
      <xsl:if test="fl[@val = 'Priority']/node() ='High'">High</xsl:if>
      <xsl:if test="fl[@val = 'Priority']/node() ='Lowest'">Low</xsl:if>
      <xsl:if test="fl[@val = 'Priority']/node() ='Low'">Low</xsl:if>
      <xsl:if test="fl[@val = 'Priority']/node() ='Normal'">Medium</xsl:if>,
      <xsl:if test="fl[@val = 'Status']/node() ='Not Started'">Open</xsl:if>
      <xsl:if test="fl[@val = 'Status']/node() ='Deferred'">Open</xsl:if>
      <xsl:if test="fl[@val = 'Status']/node() ='In Progress'">Open</xsl:if>
      <xsl:if test="fl[@val = 'Status']/node() ='Waiting on someone else'">Open</xsl:if>
      <xsl:if test="fl[@val = 'Status']/node() ='Completed'">Closed</xsl:if>
      <xsl:text>&#xa;</xsl:text>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

32. **Deskv2TotalTopics**: This web service API operation of Desk takes no input and gives the total number of topics. In order to support the pagination, our internal transformation makes sure that the count of topics is changed to (page number, page size) combination.

(a) **LAV Mapping**:

$$Deskv2TotalTopics^{ff}(pgno, pgsize) \leftarrow Page(pgno, '\text{ Desk v2 API}', '\text{Deskv2Topic}', pgsize). \tag{C.32}$$

(b) **XSD**: Consider the web service operation defined using the LAVMapping C.32. We are interested in the following elements (XPath)

i. Total Topics (*json/total_entries*)

Note how the root element is json. This is because we convert the json response to xml response. During this conversion, the desired root element is json.

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="json">
    <xs:complexType>
      <xs:all>
        <xs:element name="total_entries">
          <xs:complexType>
            <xs:simpleContent>
              <xs:extension base="xs:string">
                <xs:attribute name="type" type="xs:string">
                 </xs:attribute>
              </xs:extension>
            </xs:simpleContent>
          </xs:complexType>
        </xs:element>
        <xs:any maxOccurs="unbounded" processContents="lax"/>
      </xs:all>
      <xs:attribute type="xs:string" name="type"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

(c) **XSLT**: We are interested in extracting the page number, entries per page from the operation response as shown below. Refer our discussion on page handling (section C.2.1). The default number of entries per page is set as 25.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:fn="http://www.w3.org/2005/xpath-functions" version="2.0">
  <xsl:output method="text" omit-xml-declaration="yes"
    cdata-section-elements="namelist"/>
  <xsl:template name="for-loop">
    <xsl:param name="total" select="1"/>
    <xsl:param name="increment" select="1"/>
    <xsl:param name="page" select="1"/>
    <xsl:if test="$total &gt; 0">
      <xsl:value-of select="$page"/>,
      <xsl:value-of select="$increment"/>
      <xsl:text>&#xa;</xsl:text>
      <xsl:call-template name="for-loop">
        <xsl:with-param name="total" select="$total - $increment"/>
        <xsl:with-param name="increment" select="$increment"/>
        <xsl:with-param name="page" select="$page+1"/>
      </xsl:call-template>
    </xsl:if>
  </xsl:template>
  <xsl:template match="/">
```

```
    <xsl:variable name="default">25</xsl:variable>
    <xsl:variable name="page">
      <xsl:copy-of select="$default"/>
    </xsl:variable>
    <xsl:variable name="total">
      <xsl:value-of select="json/total_entries"/>
    </xsl:variable>
    <xsl:call-template name="for-loop">
      <xsl:with-param name="total" select="$total"/>
      <xsl:with-param name="increment" select="$default"/>
      <xsl:with-param name="page" select="1"/>
    </xsl:call-template>
  </xsl:template>
</xsl:stylesheet>
```

33. **Deskv2Topic**: This operation takes as input (page number, page size) combination obtained from the operation considered in the LAV mapping C.32 and gives all the topics with the following details: the topic identifier *tpid*, name *tpname* and the creation date *tpcdate*.

    (a) **LAV Mapping**:

$$\begin{aligned} &Deskv2Topic^{bbfff}(pgno, pgsize, tpid, tpname, tpcdate) \leftarrow \\ &\quad Page(pgno, '\text{ Desk v2 API}', '\text{ Deskv2Topic}', pgsize), \\ &\quad\quad Topic(tpid, '\text{ Desk v2 API}', tpname, tpcdate, fid). \end{aligned} \tag{C.33}$$

    (b) **XSD**: Consider the web service operation defined using the LAVMapping C.33. We are interested in the following elements (XPath)

    i. Topic identifier (*json/_embedded/entries/array/position*)
    ii. Topic name (*json/_embedded/entries/array/title*)
    iii. Topic Creation date (*json/_embedded/entries/array/created_at*)

    Note how the root element is json. This is because we convert the json response to xml response. During this conversion, the desired root element is json.

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="json">
    <xs:complexType>
      <xs:all minOccurs="1" maxOccurs="1">
        <xs:element name="_embedded">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="entries" maxOccurs="unbounded" minOccurs="0">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="array" maxOccurs="unbounded" minOccurs="0">
                      <xs:complexType>
                        <xs:all minOccurs="1" maxOccurs="1">
                          <xs:element name="position">
                            <xs:complexType>
                              <xs:simpleContent>
                                <xs:extension base="xs:string">
```

```
                                    <xs:attribute name="type" type="xs:string">
                                    </xs:attribute>
                                  </xs:extension>
                                </xs:simpleContent>
                              </xs:complexType>
                            </xs:element>
                            <xs:element name="name">
                              <xs:complexType>
                                <xs:simpleContent>
                                  <xs:extension base="xs:string">
                                    <xs:attribute name="type" type="xs:string">
                                    </xs:attribute>
                                  </xs:extension>
                                </xs:simpleContent>
                              </xs:complexType>
                            </xs:element>
                            <xs:element name="created_at">
                              <xs:complexType>
                                <xs:simpleContent>
                                  <xs:extension base="xs:dateTime">
                                    <xs:attribute name="type" type="xs:string">
                                    </xs:attribute>
                                  </xs:extension>
                                </xs:simpleContent>
                              </xs:complexType>
                            </xs:element>
                            <xs:any maxOccurs="unbounded" processContents="lax"/>
                          </xs:all>
                        <xs:attribute type="xs:string" name="class"/>
                        </xs:complexType>
                      </xs:element>
                    </xs:sequence>
                  <xs:attribute type="xs:string" name="class"/>
                  </xs:complexType>
                </xs:element>
              </xs:sequence>
            <xs:attribute type="xs:string" name="class"/>
            </xs:complexType>
          </xs:element>
          <xs:any maxOccurs="unbounded" processContents="lax"/>
        </xs:all>
      </xs:complexType>
    </xs:element>
</xs:schema>
```

(c) **XSLT**: Consider the web service operation defined using the LAVMapping C.33. We are interested in making use of four elements from the response: task identifier *id*, name *content*, creation date *created_at* and completion date *completed_at*.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:fn="http://www.w3.org/2005/xpath-functions" version="2.0">
  <xsl:output method="text" omit-xml-declaration="yes"
    cdata-section-elements="namelist"/>
```

```
  <xsl:template match="/">
    <xsl:for-each select="json/_embedded/entries/array">
      <xsl:value-of select="position"/>,
      <xsl:value-of select="name"/>,
      <xsl:value-of select="fn:substring(current()/created_at,1,10)"/>
      <xsl:text>&#xa;</xsl:text>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

34. **Deskv2TotalCases**: This web service API operation of Desk takes no input and gives the total number of cases. In order to support the pagination, our internal transformation makes sure that the count of cases is changed to (page number, page size) combination.

    (a) **LAV Mapping**:

$$Deskv2TotalCases^{ff}(pgno, pgsize) \leftarrow Page(pgno, '\text{ Desk v2 API}', '\text{ Deskv2Case}', pgsize). \tag{C.34}$$

    (b) **XSD**: Consider the web service operation defined using the LAVMapping C.34. We are interested in the following elements (XPath)

        i. Total Cases ($json/total\_entries$)

    Note how the root element is json. This is because we convert the json response to xml response. During this conversion, the desired root element is json.

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="json">
    <xs:complexType>
      <xs:all>
        <xs:element name="total_entries">
          <xs:complexType>
            <xs:simpleContent>
              <xs:extension base="xs:string">
                <xs:attribute name="type" type="xs:string">
                </xs:attribute>
              </xs:extension>
            </xs:simpleContent>
          </xs:complexType>
        </xs:element>
        <xs:any maxOccurs="unbounded" processContents="lax"/>
      </xs:all>
      <xs:attribute type="xs:string" name="type"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

    (c) **XSLT**: We are interested in extracting the page number, entries per page from the operation response as shown below. Refer our discussion on page

handling (section C.2.1). The default number of entries per page is set as 25.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:fn="http://www.w3.org/2005/xpath-functions" version="2.0">
  <xsl:output method="text" omit-xml-declaration="yes"
    cdata-section-elements="namelist"/>
  <xsl:template name="for-loop">
    <xsl:param name="total" select="1"/>
    <xsl:param name="increment" select="1"/>
    <xsl:param name="page" select="1"/>
    <xsl:if test="$total &gt; 0">
      <xsl:value-of select="$page"/>,
      <xsl:value-of select="$increment"/>
      <xsl:text>&#xa;</xsl:text>
      <xsl:call-template name="for-loop">
        <xsl:with-param name="total" select="$total - $increment"/>
        <xsl:with-param name="increment" select="$increment"/>
        <xsl:with-param name="page" select="$page+1"/>
      </xsl:call-template>
    </xsl:if>
  </xsl:template>
  <xsl:template match="/">
    <xsl:variable name="default">25</xsl:variable>
    <xsl:variable name="page">
      <xsl:copy-of select="$default"/>
    </xsl:variable>
    <xsl:variable name="total">
      <xsl:value-of select="json/total_entries"/>
    </xsl:variable>
    <xsl:call-template name="for-loop">
      <xsl:with-param name="total" select="$total"/>
      <xsl:with-param name="increment" select="$default"/>
      <xsl:with-param name="page" select="1"/>
    </xsl:call-template>
  </xsl:template>
</xsl:stylesheet>
```

35. **Deskv2Case**: This operation takes as input (page number, page size) combination obtained from the operation considered in the LAV mapping C.34 and gives all the cases with the following details: case identifer *tkid*, name *tkname*, creation date *tkcdate*, status *tkstatus* and priority *tkpriority*.

    (a) **LAV Mapping**:

$$\begin{aligned} Deskv2Case^{bbffffff}(pgno, pgsize, tkid, tkname, tkcdate, tkpriority, tkstatus) \leftarrow \\ Page(pgno, '\text{ Desk v2 API}', '\text{ Deskv2Case}', pgsize), \\ Ticket(tkid, '\text{ Desk v2 API}', tkname, tkcdate, tkddate, tkcmpdate, tkpriority, tkstatus). \end{aligned} \tag{C.35}$$

    (b) **XSD**: Consider the web service operation defined using the LAVMapping C.35. We are interested in the following elements (XPath)

i. Ticket identifier (*json/_embedded/entries/array/subject*)
ii. Ticket name (*json/_embedded/entries/array/subject*)
iii. Ticket Creation date (*json/_embedded/entries/array/created_at*)
iv. Ticket Priority (*json/_embedded/entries/array/priority*)
v. Ticket Status (*json/_embedded/entries/array/status*)

Note how we make use of the ticket name as the ticket identifier.

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="json">
    <xs:complexType>
      <xs:all minOccurs="1" maxOccurs="1">
        <xs:element name="_embedded">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="entries" maxOccurs="unbounded" minOccurs="0">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="array" maxOccurs="unbounded" minOccurs="0">
                      <xs:complexType>
                        <xs:all minOccurs="1" maxOccurs="1">
                          <xs:element name="priority">
                            <xs:complexType>
                              <xs:simpleContent>
                                <xs:extension base="xs:string">
                                  <xs:attribute name="type" type="xs:string">
                                  </xs:attribute>
                                </xs:extension>
                              </xs:simpleContent>
                            </xs:complexType>
                          </xs:element>
                          <xs:element name="status">
                            <xs:complexType>
                              <xs:simpleContent>
                                <xs:extension base="xs:string">
                                  <xs:attribute name="type" type="xs:string">
                                  </xs:attribute>
                                </xs:extension>
                              </xs:simpleContent>
                            </xs:complexType>
                          </xs:element>
                          <xs:element name="subject">
                            <xs:complexType>
                              <xs:simpleContent>
                                <xs:extension base="xs:string">
                                  <xs:attribute name="type" type="xs:string">
                                  </xs:attribute>
                                </xs:extension>
                              </xs:simpleContent>
                            </xs:complexType>
                          </xs:element>
                          <xs:element name="created_at">
                            <xs:complexType>
                              <xs:simpleContent>
                                <xs:extension base="xs:dateTime">
                                  <xs:attribute name="type" type="xs:string">
                                  </xs:attribute>
```

```
                                </xs:extension>
                              </xs:simpleContent>
                            </xs:complexType>
                          </xs:element>
                          <xs:any maxOccurs="unbounded" processContents="lax"/>
                        </xs:all>
                      <xs:attribute type="xs:string" name="class"/>
                      </xs:complexType>
                    </xs:element>
                  </xs:sequence>
                <xs:attribute type="xs:string" name="class"/>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          <xs:attribute type="xs:string" name="class"/>
          </xs:complexType>
        </xs:element>
        <xs:any maxOccurs="unbounded" processContents="lax"/>
      </xs:all>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

(c) **XSLT**: We are interested in extracting the *subject*, *created_at*, *priority* and *status* from the operation response as shown below. Note how we make use of the ticket name as the ticket identifier. Also note the transformation on the ticket status and ticket priority to the desired internal format.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:fn="http://www.w3.org/2005/xpath-functions" version="2.0">
  <xsl:output method="text" omit-xml-declaration="yes"
    cdata-section-elements="namelist"/>
  <xsl:template match="/">
    <xsl:for-each select="json/_embedded/entries/array">
      <xsl:value-of select="subject"/>,
      <xsl:value-of select="subject"/>,
      <xsl:value-of select="fn:substring(current()/created_at,1,10)"/>,
      <xsl:if test="priority ='1'">Low</xsl:if>
      <xsl:if test="priority ='2'">Low</xsl:if>
      <xsl:if test="priority ='3'">Low</xsl:if>
      <xsl:if test="priority ='4'">Low</xsl:if>
      <xsl:if test="priority ='5'">Medium</xsl:if>
      <xsl:if test="priority ='6'">Medium</xsl:if>
      <xsl:if test="priority ='7'">Medium</xsl:if>
      <xsl:if test="priority ='8'">High</xsl:if>
      <xsl:if test="priority ='9'">High</xsl:if>
      <xsl:if test="priority ='10'">High</xsl:if>,
      <xsl:if test="status='new'">Open</xsl:if>
      <xsl:if test="status ='pending'">Open</xsl:if>
      <xsl:if test="status ='resolved'">Closed</xsl:if>
      <xsl:if test="status ='open'">Open</xsl:if>
      <xsl:text>&#xa;</xsl:text>
    </xsl:for-each>
  </xsl:template>
```

```
</xsl:stylesheet>
```

C.2.1 HandlingPagination

Pagination is a special case and we describe it in some detail below:

Example C.2.1. *Pagination comes in various ways. One commonly used mechanism is to give the page number and the desired count of entries per page. This is captured in the global schema relation discussed above. It has four attributes*

1. *Page Number* pgno*: This attribute corresponds to the page number.*
2. *Source* src*: This attribute (like many other global schema relations) captures the source of the information: here the web service API and version*
3. *Operation* operation*: It captures the corresponding web service API operation which required paginated input parameters*
4. *Limit* limit*: It captures the notion of maximum (or default) number of entries per page.*

We discussed about some internal transformation that converts the total number of entries to page number and number of entries (page number, page limit combination) for the LAV mappings C.7, C.24, C.32 and C.34. This (page number,page limit) combination is used by the operations considered in the LAV mappings C.8, C.25, C.33 and C.35 respectively.

Now we show how the total count of (campaigns, entries, topics or cases) is transformed to a (page number, page limit) combination. We show here one example XSLT for Deskv2TotalTopics considered in the LAV mapping C.32. The other examples are done similarly (except for the change in the element names)

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:fn="http://www.w3.org/2005/xpath-functions" version="2.0">
  <xsl:output method="text" omit-xml-declaration="yes"
cdata-section-elements="namelist"/>
  <xsl:template name="for-loop">
    <xsl:param name="total" select="1"/>
    <xsl:param name="increment" select="1"/>
    <xsl:param name="page" select="1"/>
    <xsl:if test="$total &gt; 0"><xsl:value-of select="$page"/>,
      <xsl:value-of select="$increment"/><xsl:text>&#xa;</xsl:text>
       <xsl:call-template name="for-loop">
```

```
        <xsl:with-param name="total" select="$total - $increment"/>
        <xsl:with-param name="increment" select="$increment"/>
        <xsl:with-param name="page" select="$page+1"/>
      </xsl:call-template>
    </xsl:if>
  </xsl:template>
  <xsl:template match="/">
    <xsl:variable name="default">25</xsl:variable>
    <xsl:variable name="page">
      <xsl:copy-of select="$default"/>
    </xsl:variable>
    <xsl:variable name="total">
      <xsl:value-of select="json/total_entries"/>
    </xsl:variable>
    <xsl:call-template name="for-loop">
      <xsl:with-param name="total" select="$total"/>
      <xsl:with-param name="increment" select="$default"/>
      <xsl:with-param name="page" select="1"/>
    </xsl:call-template>
  </xsl:template>
</xsl:stylesheet>
```

The idea behind here is to divide the total count by the desired number of entries per page (here the default is 25). Therefore if the count is 100, this transformation will give the following

```
(1,25)
(2,25)
(3,25)
(4,25)
```

■

C.3 Enterprise Records

We will discuss how to form the queries over the global schema relations. The queries formulated over the global schema relation form the record definition

We considered the record definitions in the table C.4. Note that every record definition we considered takes into account the state of the resource as reported yesterday (the day before the evaluation of the query/record definition).

We will see how each of the record definition names considered in the table C.4 is actually formulated over the global schema relations. Recall that inverse rules algorithm can handle recursive datalog queries. We will consider in our examples

Table C.4: Records from Web Services

Project Management Services		
1.	Daily New Projects	New projects created yesterday
2.	Daily Active Projects	Active projects reported yesterday
3.	Daily OnHold Projects	Onhold projects reported yesterday
4.	Daily OnHold or Archived Projects	Active or Onhold projects reported yesterday
5.	Daily Same Status Projects	Projects with same status reported yesterday
6.	Daily TodoLists	TodoLists reported as of yesterday
7.	Daily New Tasks	New Tasks created yesterday
8.	Daily Open Tasks	Tasks reported to be completed yesterday
9.	Daily Closed Tasks	Tasks reported to be solved yesterday
Email Marketing Services		
10.	Daily New Campaign	New campaigns created yesterday
11.	Daily Campaign Statistics	Campaign statistics reported yesterday
Support (Helpdesk) Services		
12.	Daily New Tickets	New tickets created yesterday
13.	Daily Open Tickets	Tickets remaining unsolved as reported yesterday
14.	Daily Closed Tickets	Tickets reported to be solved yesterday
15.	Daily New Forums	New forums created yesterday
16.	Daily All Forums	All forums as reported yesterday
17.	Daily New Topics	New topics created yesterday

1. Conjunctive query
2. Union of Conjunctive query (Example: Record definition C.39)
3. Recursive Datalog query (Example: Record definition C.40)

Note that in all our queries, we use $yesterday()$, a special function that before evaluation is transformed to the date before today's evaluation. We also have a function like $yesterday(n)$, which gives the date n days before yesterday.

1. **Daily New Projects**

$$q(pid, src, pname, pstatus) : -Project(pid, src, pname,' yesterday()', pstatus). \quad \text{(C.36)}$$

2. **Daily Active Projects**

$$q(pid, src, pname) : -Project(pid, src, pname, pcdate,' Active'). \quad \text{(C.37)}$$

3. **Daily OnHold Projects**

$$q(pid, src, pname) : -Project(pid, src, pname, pcdate, 'OnHold'). \tag{C.38}$$

4. **Daily OnHold or Archived Projects**: This is an example of a union of conjunctive query.

$$\begin{aligned} q(pid, src, pname) &: -Project(pid, src, pname, pcdate, 'Active'). \\ q(pid, src, pname) &: -Project(pid, src, pname, pcdate, 'OnHold'). \end{aligned} \tag{C.39}$$

5. **Daily Same Status Projects**: This is an example of recursive datalog query.

$$\begin{aligned} q(pid, src, pid, src, status) &: -Project(pid, src, pname, pcdate, status). \\ q(pid1, src1, pid2, src2, status) &: -Project(pid1, src1, pname1, pcdate1, status), \\ &\qquad q(pid2, src2, pid3, src3, status). \end{aligned} \tag{C.40}$$

6. **Daily TodoLists**

$$\begin{aligned} q(pid, tlid) : -Project(pid, src, pname, pcdate, pstatus), \\ TaskList(pid, src, tlid). \end{aligned} \tag{C.41}$$

7. **Daily New Tasks**

$$\begin{aligned} q(tid, src, tname, tstatus) : -Project(pid, src, pname, pcdate, pstatus), \\ TaskList(pid, src, tlid), \\ Task(tlid, tid, src, tname, 'yesterday()', tddate, tcmpdate, tstatus). \end{aligned} \tag{C.42}$$

8. **Daily Open Tasks**

$$\begin{aligned} q(tid, src, tname) : -Task(tlid, tid, src, tname, tcdate, tddate, \\ tcmpdate, 'Open'). \end{aligned} \tag{C.43}$$

9. **Daily Closed Tasks**

$$\begin{aligned} q(tid, src, tname) : -Project(pid, src, pname, pcdate, pstatus), \\ TaskList(pid, src, tlid), \\ Task(tlid, tid, src, tname, tcdate, tddate, 'yesterday()', 'Completed'). \end{aligned} \tag{C.44}$$

10. **Daily New Forums**

$$q(fid, src, fname) : -Forum(fid, src, fname, 'yesterday()'). \tag{C.45}$$

11. **Daily All Forums**

$$q(fid, src, fname) : -Forum(fid, src, fname, fcdate). \tag{C.46}$$

12. **Daily New Topics**

$$q(tpid, src, tpname) : -Topic(tpid, src, tpname,' yesterday()', fid). \tag{C.47}$$

13. **Daily New Tickets**

$$\begin{aligned} &q(tkid, src, tkname, tkpriority, tkstatus) : - \\ &Ticket(tkid, src, tkname,' yesterday()', tkddate, tkcmpdate, \\ &\qquad tkpriority, tkstatus). \end{aligned} \tag{C.48}$$

14. **Daily Open Tickets**

$$\begin{aligned} &q(tkid, src, tkname, tkpriority) : - \\ &\quad Ticket(tkid, src, tkname, tkcdate, tkddate, tkcmpdate, tkpriority,' Open'). \end{aligned} \tag{C.49}$$

15. **Daily Closed Tickets**

$$\begin{aligned} &q(tkid, src, tkname, tkcdate, tkddate, tkpriority) : - \\ &Ticket(tkid, src, tkname, tkcdate, tkddate,' yesterday()', \\ &\qquad tkpriority,' Closed') \end{aligned} \tag{C.50}$$

16. **Daily New Campaigns**

$$\begin{aligned} &q(cmid, src, cmname, cmstatus) : -Campaign(cmid, src, cmname, \\ &\qquad 'yesterday()', cmstatus). \end{aligned} \tag{C.51}$$

17. **Daily Campaign Statistics**

$$\begin{aligned} &q(cmid, src, cmname, cmctr, cmfr, cmbr) : - \\ &\quad Campaign(cmid, src, cmname, cmcdate, cmstatus), \\ &\quad CampaignStatistics(cmid, src, cmar, cmctr, cmfr, cmbr). \end{aligned} \tag{C.52}$$

C.4 Performance Indicators

We demonstrate in this section how to create performance indicator queries using the records. For this purpose, we would also request the reader to refer the relational tables used in DaWeS (refer section D.2). We show how we use SQL query to compute our desired performance indicator.

For our experiments, we considered the performance indicators given in the table C.5. We see different types of performance indicators

1. Count
2. Average

3. Percentage
4. A list of tuples (a table, useful for creating charts)

Table C.5: Performance Indicators

Project Management Services		
1.	Total Monthly New Projects	New projects created during the last 30 days
2.	Total Monthly Active Projects	Active projects reported during the last 30 days
3.	Total Monthly OnHold Projects	Onhold projects reported during the last 30 days
4.	Total Todo Lists	Lately reported total todo lists
5.	Total Monthly New Tasks	New tasks created during the last 30 days
6.	Total Monthly Completed Tasks	Tasks completed during the last 30 days
7.	Average Tasks Completed Daily in a month	Average number of tasks completed during the last 30 days
8.	Monthly percentage of tasks completed to tasks created in a day	Percentage of tasks completed to tasks created in a day during the last 30 days
Email Marketing Services		
9.	Total Monthly New Campaigns	New campaigns created during the last 30 days
10.	Monthly Bounces of Campaign	Monthly bounces of every campaign reported during the last 30 days (latest reported)
11.	Monthly Click Throughs of Campaign	Monthly clicks of every campaign reported during the last 30 days (latest reported)
12.	Monthly Forwards of Campaign	Monthly forwards of every campaign reported during the last 30 days (latest reported)
Support (Helpdesk) Services		
13.	Total New Tickets Registered in a month	Total count of tickets created during the last 30 days
14.	Total Monthly Solved Tickets	Total count of solved tickets during the last 30 days

15.	Total New Forums Registered in a month	Total forums created during the last 30 days
16.	All Forums in a month	Total forums (latest) reported during the last 30 days
17.	Total New Topics Registered in a month	Total new topics created during the last 30 days
18.	Total High Priority Tickets Registered in a month	Total number of high priority tickets created during the last 30 days
19.	Percentage of High Priority Tickets Registered in a month	Percentage of high priority tickets created during the last 30 days
20.	Daily Average Resolution Time	Average resolution time of the tickets solved during the last day

As explained in section D.2, we use the two relations $OrgRecord$ and $OrgRecordVal$ to store the records of every organization. The performance indicator queries considers this fact to define a SQL query using these organization records.

To make our SQL query generic (to be able to use with any organization), we use a variable $\$orgID$ that is internally transformed to the identifier of the concerned organization before executing the SQL query. Note that the relations $LCRecord$ and $LCRECORDATTRIBUTE$ stores the record definition and the record attribute details.

Given below is the SQL query to compute total monthly new projects. Note that it makes use of the record definition C.36 (Daily New Projects). The idea is to take the count of all the project identifiers that were created during the last 30 days.

```
SELECT count(value) FROM OrgRecordVal
WHERE org_record_id IN
(
  SELECT id FROM OrgRecord
  WHERE time < sysdate
    AND
  time > sysdate  - interval '30' day
    AND
  lcrecord_id =
  (
    SELECT ID FROM LCRECORD
    WHERE NAME LIKE 'Daily New Projects'
  )
    AND
  org_id = $orgID
```

```
)
  AND
lcr_attr_id =
(
  SELECT ID FROM LCRECORDATTRIBUTE
  WHERE NAME LIKE 'pid'
    AND
  LCRECORD_ID =
  (
    SELECT ID FROM LCRECORD
    WHERE NAME LIKE 'Daily New Projects'
  )
)
```

Note here we are using the record 'Daily New Projects' and its attribute 'pid' to get the count of projects created in a month.

So for every performance indicator, we must take care of the following

1. Record Definition and Attribute: We must be aware of the relevant records and the specific attribute
2. Period: We must be aware of the period from which the desired records are needed
3. Supported SQL Aggregate functions

Now we take another example *Monthly Average resolution time*, where we make use of two different attributes of a record (the completion time of the ticket and the creation time of the ticket). We make use of the record definition C.44 (Daily Closed Tickets) and its two attributes (*tkcdate* and *tkcmpdate*,). We use some of the Oracle in-built functions to create the following performance indicator.

```
SELECT sum
(
  TO_NUMBER
  (
    trunc( sysdate) - to_date ( replace( o.value,'''',''), 'yyyy-mm-dd')
  )
) / count (o.value)
FROM OrgRecordVal o, Dual
WHERE org_record_id in
(
  SELECT id FROM OrgRecord
  WHERE time < sysdate
    AND
  time > sysdate  - interval '30' day
    AND
  lcrecord_id =
```

```
  (
    SELECT ID FROM LCRECORD
    WHERE NAME LIKE 'Daily Closed Tickets'
  )
    AND
  org_id=$orgID
)
  and
lcr_attr_id =
(
  SELECT ID FROM LCRECORDATTRIBUTE
  WHERE NAME LIKE 'tkcdate'
    AND
  LCRECORD_ID =
  (
    SELECT ID FROM LCRECORD
    WHERE NAME LIKE 'Daily Closed Tickets'
  )
)
```

Now we see the SQL queries of all the considered performance indicators.

1. **Monthly Bounces of Campaign**: This performance indicator is defined using the record definition C.52 (Daily Campaign Statistics). The indicator makes use of the organization records corresponding to the daily registered campaign statistics for the last day. It selects the values for the attributes *cmid*, campaign identifier and *cmbr*, the campaign abuse count from these records.

```
WITH c AS
(
  SELECT * FROM OrgRecordVal
  WHERE org_record_id IN
  (
    SELECT id FROM OrgRecord
    WHERE time < sysdate
      AND
    time > sysdate  - interval '1' day
      AND
    lcrecord_id  =
    (
      SELECT ID FROM LCRECORD
      WHERE NAME LIKE 'Daily Campaign Statistics'
    )
      AND
    org_id = $orgID
  )
    AND lcr_attr_id  =
    (
       SELECT ID FROM LCRECORDATTRIBUTE
       WHERE NAME LIKE 'cmname'
          AND
```

```
       LCRECORD_ID  =
       (
         SELECT ID FROM LCRECORD
         WHERE NAME LIKE 'Daily Campaign Statistics'
       )
    )
  ),
d AS
(
  SELECT * FROM OrgRecordVal
  WHERE org_record_id IN
  (
    SELECT id FROM OrgRecord
    WHERE time < sysdate
      AND
    time > sysdate  - interval '1' day
      AND
    lcrecord_id =
    (
      SELECT ID FROM LCRECORD
      WHERE NAME LIKE 'Daily Campaign Statistics'
    )
      AND
    org_id = $orgID
  )
    AND
  lcr_attr_id =
  (
    SELECT ID FROM LCRECORDATTRIBUTE
    WHERE NAME LIKE 'cmbr'
       AND
    LCRECORD_ID =
    (
      SELECT ID FROM LCRECORD
      WHERE NAME LIKE 'Daily Campaign Statistics'
    )
  )
)
SELECT c.value AS cmid ,d.value AS bounces FROM c,d
  WHERE
c.org_record_id = d.org_record_id
  AND
c.idx = d.idx
```

2. **Total Monthly Solved Tickets**: This performance indicator is defined using the record definition C.50 (Daily Closed Tickets). The indicator makes use of the organization records corresponding to the tickets solved daily for the last 30 days. It selects the values for the attribute *tkid*, ticket identifier and computes their total count.

```
SELECT count(*) FROM
(
  SELECT distinct value FROM OrgRecordVal
  WHERE org_record_id IN
```

```
  (
    SELECT id FROM OrgRecord
    WHERE time < sysdate
      AND
    time > sysdate  - interval '30' day
      AND
    lcrecord_id =
    (
      SELECT ID FROM LCRECORD
      WHERE NAME LIKE 'Daily Closed Tickets'
    )
     AND
    org_id = $orgID
  )
    AND
  lcr_attr_id =
  (
    SELECT ID FROM LCRECORDATTRIBUTE
    WHERE NAME LIKE 'tkid'
      AND
    LCRECORD_ID =
    (
      SELECT ID FROM LCRECORD
      WHERE NAME LIKE 'Daily Closed Tickets'
    )
  )
)
```

3. **Daily Average Resolution Time**: This performance indicator is defined using the record definition C.50 (Daily Closed Tickets). The indicator makes use of the organization records corresponding to the tickets solved during the last day. It selects the values for the attribute *tkcdate*, ticket creation date. For every ticket, it computes the days passed sinced the creation of the ticket. Once the count for every ticket is obtained, they are summed and the divided by the total number of tickets (number of values considered for the ticket creation date).

```
SELECT sum (TO_NUMBER
  (
    trunc (sysdate) -to_date (replace (o.value,'''',''') ,
            'yyyy-mm-dd')
  )
) /count (o.value)
FROM OrgRecordVal o, Dual
WHERE org_record_id IN
(
  SELECT id FROM OrgRecord
  WHERE time < sysdate
    AND
  time > sysdate  - interval '1' day
    AND
  lcrecord_id =
  (
    SELECT ID FROM LCRECORD
```

```
    WHERE NAME LIKE 'Daily Closed Tickets'
  )
    AND
  org_id = $orgID
)
  AND
lcr_attr_id =
(
  SELECT ID FROM LCRECORDATTRIBUTE
  WHERE NAME LIKE 'tkcdate'
    AND
  LCRECORD_ID =
  (
     SELECT ID FROM LCRECORD
     WHERE NAME LIKE 'Daily Closed Tickets'
  )
)
```

4. **Total New Tickets Registered in a month**: This performance indicator is defined using the record definition C.48 (Daily New Tickets). The indicator makes use of the organization records corresponding to the new tickets created daily for the last 30 days. It selects the values for the attribute *tkid*, ticket identifier and computes their total count.

```
SELECT count (*) FROM
(
  SELECT distinct value FROM OrgRecordVal
  WHERE org_record_id IN
  (
    SELECT id FROM OrgRecord
    WHERE time < sysdate
      AND
    time > sysdate  - interval '30' day
      AND
    lcrecord_id =
    (
      SELECT ID FROM LCRECORD
      WHERE NAME LIKE 'Daily New Tickets'
    )
      AND
    org_id = $orgID
  )
    AND
  lcr_attr_id =
  (
    SELECT ID FROM LCRECORDATTRIBUTE
    WHERE NAME LIKE 'tkid'
      AND
    LCRECORD_ID =
    (
      SELECT ID FROM LCRECORD
      WHERE NAME LIKE 'Daily New Tickets'
    )
  )
```

```
)
```

5. **Total New Forums Registered in a month**: This performance indicator is defined using the record definition C.45 (Daily New Forums). The indicator makes use of the organization records corresponding to the new forums created daily for the last 30 days. It selects the values for the attribute *fid*, forum identifier and computes their total count.

```
SELECT count (*) FROM
(
  SELECT distinct value FROM OrgRecordVal
  WHERE org_record_id IN
  (
    SELECT id FROM OrgRecord
    WHERE time < sysdate
      AND
    time > sysdate  - interval '30' day
      AND
    lcrecord_id =
    (
      SELECT ID FROM LCRECORD
      WHERE NAME LIKE 'Daily New Forums'
    )
      AND
    org_id = $orgID
  )
    AND
  lcr_attr_id =
  (
    SELECT ID FROM LCRECORDATTRIBUTE
    WHERE NAME LIKE 'fid'
      AND
    LCRECORD_ID =
    (
       SELECT ID FROM LCRECORD
       WHERE NAME LIKE 'Daily New Forums'
    )
  )
)
```

6. **All Forums in a month**: This performance indicator is defined using the record definition C.46 (Daily New Forums). The indicator makes use of the organization records corresponding to the differnent forum reported daily for the last 30 days. It selects the values for the attribute *fid*, forum identifier and computes their total count.

```
SELECT count (*) FROM
(
  SELECT distinct value FROM OrgRecordVal
  WHERE org_record_id IN
```

```
  (
    SELECT id FROM OrgRecord
    WHERE time < sysdate
      AND
    time > sysdate  - interval '1' day
      AND
    lcrecord_id =
    (
      SELECT ID FROM LCRECORD
      WHERE NAME LIKE 'Daily All Forums'
    )
      AND
    org_id = $orgID
  )
    AND
  lcr_attr_id =
  (
    SELECT ID FROM LCRECORDATTRIBUTE
    WHERE NAME LIKE 'fid'
      AND
    LCRECORD_ID =
    (
      SELECT ID FROM LCRECORD
      WHERE NAME LIKE 'Daily All Forums'
    )
  )
)
```

7. **Total New Topics Registered in a month**: This performance indicator is defined using the record definition C.47 (Daily New Topics). The indicator makes use of the organization records corresponding to the new topics created daily for the last 30 days. It selects the values for the attribute *tpid*, topic identifier and computes their total count.

```
SELECT count (*) FROM
(
  SELECT distinct value FROM OrgRecordVal
  WHERE org_record_id IN
  (
     SELECT id FROM OrgRecord
     WHERE time < sysdate
       AND
     time > sysdate  - interval '30' day
       AND
     lcrecord_id =
     (
       SELECT ID FROM LCRECORD
       WHERE NAME LIKE 'Daily New Topics'
     )
       AND
     org_id = $orgID
   )
     AND
   lcr_attr_id =
```

```
   (
     SELECT ID FROM LCRECORDATTRIBUTE
     WHERE NAME LIKE 'tpid'
       AND
     LCRECORD_ID =
     (
       SELECT ID FROM LCRECORD
       WHERE NAME LIKE 'Daily New Topics'
    )
  )
)
```

8. **Total High Priority Tickets Registered in a month**: This performance indicator is defined using the record definition C.48 (Daily New Tickets). The indicator makes use of the organization records corresponding to the new tickets created daily for the last 30 days. It selects the values for the attribute *tkpriority*, ticket priority and computes the total count of the tickets having the value for *tkpriority* as 'High'.

```
SELECT count (*) FROM OrgRecordVal
WHERE org_record_id IN
(
  SELECT id FROM OrgRecord
  WHERE time < sysdate
    AND
  time > sysdate  - interval '30' day
    AND
  lcrecord_id =
  (
    SELECT ID FROM LCRECORD
    WHERE NAME LIKE 'Daily New Tickets'
  )
    AND
  org_id = $orgID
)
  AND
lcr_attr_id =
(
  SELECT ID FROM LCRECORDATTRIBUTE
  WHERE NAME LIKE 'tkpriority'
    AND
  LCRECORD_ID =
  (
    SELECT ID FROM LCRECORD
    WHERE NAME LIKE 'Daily New Tickets'
  )
)
  AND
value LIKE '''High'''
```

9. **Percentage of High Priority Tickets Registered in a month** This performance indicator is defined using the record definition C.48 (Daily New Tickets).

The indicator makes use of the organization records corresponding to the new tickets created daily for the last 30 days. It selects the values for the attribute *tkpriority*, ticket priority and computes the total count of the tickets having the value for *tkpriority* as 'High' and the total count of all the tickets created during the last 30 days (simply using the count of tkpriority). It finally computes the percentage using the two counts.

```
WITH count1 AS
(
  SELECT count (*) c1 FROM OrgRecordVal
  WHERE org_record_id in
  (
    SELECT id FROM OrgRecord
    WHERE time < sysdate
      AND
    time > sysdate  - interval '30' day
      AND
    lcrecord_id =
    (
      SELECT ID FROM LCRECORD
      WHERE NAME LIKE 'Daily New Tickets'
    )
      AND
    org_id = $orgID
  )
    AND
  lcr_attr_id =
  (
    SELECT ID FROM LCRECORDATTRIBUTE
    WHERE NAME LIKE 'tkpriority'
      AND
    LCRECORD_ID =
    (
      SELECT ID FROM LCRECORD
      WHERE NAME LIKE 'Daily New Tickets'
    )
  )
    AND
  value LIKE '''High'''),
count2 AS
(
  SELECT count (*) c2 FROM OrgRecordVal
  WHERE org_record_id IN
  (
    SELECT id FROM OrgRecord
    WHERE time < sysdate
      AND
    time > sysdate  - interval '30' day
      AND
    lcrecord_id =
    (
      SELECT ID FROM LCRECORD
      WHERE NAME LIKE 'Daily New Tickets'
    )
      AND
```

```
    org_id = $orgID
  )
    AND
  lcr_attr_id =
  (
    SELECT ID FROM LCRECORDATTRIBUTE
    WHERE NAME LIKE 'tkpriority'
      AND
    LCRECORD_ID =
    (
      SELECT ID FROM LCRECORD
      WHERE NAME LIKE 'Daily New Tickets'
    )
  )
)
SELECT c1/c2*100 FROM count1,count2
```

10. **Total Monthly New Projects**: This performance indicator is defined using the record definition C.36 (Daily New Projects). The indicator makes use of the organization records corresponding to the new projects created daily for the last 30 days. It selects the values for the attribute *pid*, project identifier and computes their total count.

```
SELECT count(value) FROM OrgRecordVal
WHERE org_record_id IN
(
  SELECT id FROM OrgRecord
  WHERE time < sysdate
    AND
  time > sysdate  - interval '30' day
    AND
  lcrecord_id =
  (
    SELECT ID FROM LCRECORD
    WHERE NAME LIKE 'Daily New Projects'
  )
    AND
  org_id = $orgID
)
  AND
lcr_attr_id =
(
  SELECT ID FROM LCRECORDATTRIBUTE
  WHERE NAME LIKE 'pid'
    AND
  LCRECORD_ID =
  (
    SELECT ID FROM LCRECORD
    WHERE NAME LIKE 'Daily New Projects'
  )
)
```

11. **Total Monthly Active Projects**: This performance indicator is defined using the record definition C.37 (Daily Active Projects). The indicator makes use of

the organization records corresponding to the active projects reported daily for the last 30 days. It selects the values for the attribute *pid*, project identifier and computes their total count (considering only the unique values).

```
SELECT count (*) FROM
(
  SELECT distinct value FROM OrgRecordVal
  WHERE org_record_id IN
  (
    SELECT id FROM OrgRecord
    WHERE time < sysdate
      AND
    time > sysdate  - interval '30' day
      AND
    lcrecord_id =
    (
      SELECT ID FROM LCRECORD
      WHERE NAME LIKE 'Daily Active Projects'
    )
      AND
    org_id = $orgID
  )
    AND
  lcr_attr_id =
  (
    SELECT ID FROM LCRECORDATTRIBUTE
    WHERE NAME LIKE 'pid'
      AND
    LCRECORD_ID =
    (
      SELECT ID FROM LCRECORD
      WHERE NAME LIKE 'Daily Active Projects'
    )
  )
)
```

12. **Total Monthly OnHold Projects**: This performance indicator is defined using the record definition C.38 (Daily OnHold Projects). The indicator makes use of the organization records corresponding to the onhold projects reported daily for the last 30 days. It selects the values for the attribute *pid*, project identifier and computes their total count (considering only the unique values).

```
SELECT count (*) FROM
(
  SELECT distinct value FROM OrgRecordVal
  WHERE org_record_id IN
  (
    SELECT id FROM OrgRecord
    WHERE time < sysdate
      AND
    time > sysdate  - interval '30' day
      AND
    lcrecord_id =
```

```
    (
      SELECT ID FROM LCRECORD
      WHERE NAME LIKE 'Daily OnHold Projects'
    )
      AND
    org_id = $orgID
  )
    AND
  lcr_attr_id =
  (
    SELECT ID FROM LCRECORDATTRIBUTE
    WHERE NAME LIKE 'pid'
      AND
    LCRECORD_ID =
    (
      SELECT ID FROM LCRECORD
      WHERE NAME LIKE 'Daily OnHold Projects'
    )
  )
)
```

13. **Total Monthly Completed Tasks**: This performance indicator is defined using the record definition C.44 (Daily Closed Tasks). The indicator makes use of the organization records corresponding to the tasks solved daily for the last 30 days. It selects the values for the attribute *tid*, task identifier and computes their total count.

```
SELECT count (*) FROM
(
  SELECT distinct value FROM OrgRecordVal
  WHERE org_record_id IN
  (
     SELECT id FROM OrgRecord
     WHERE time < sysdate
       AND
     time > sysdate  - interval '30' day
       AND
     lcrecord_id =
     (
       SELECT ID FROM LCRECORD
       WHERE NAME LIKE 'Daily Closed Tasks'
     )
       AND
     org_id = $orgID
  )
    AND
  lcr_attr_id =
  (
    SELECT ID FROM LCRECORDATTRIBUTE
    WHERE NAME LIKE 'tid'
      AND
    LCRECORD_ID =
    (
      SELECT ID FROM LCRECORD
```

```
      WHERE NAME LIKE 'Daily Closed Tasks'
    )
  )
)
```

14. **Average Tasks Completed Daily in a month**: This performance indicator is defined using the record definition C.44 (Daily Closed Tasks). The indicator makes use of the organization records corresponding to the tasks solved daily for the last 30 days. It selects the values for the attribute *tid*, task identifier and computes their total count. It then computes the average by dividing the total count by 30.

```
SELECT count (*)/30 FROM
(
  SELECT distinct value FROM OrgRecordVal
  WHERE org_record_id IN
  (
    SELECT id FROM OrgRecord
    WHERE time < sysdate
      AND
    time > sysdate  - interval '30' day
      AND
    lcrecord_id =
    (
      SELECT ID FROM LCRECORD
      WHERE NAME LIKE 'Daily Closed Tasks'
    )
      AND
    org_id = $orgID
  )
    AND
  lcr_attr_id =
  (
    SELECT ID FROM LCRECORDATTRIBUTE
    WHERE NAME LIKE 'tid'
      AND
    LCRECORD_ID =
    (
      SELECT ID FROM LCRECORD
      WHERE NAME LIKE 'Daily Closed Tasks'
    )
  )
)
```

15. **Total Monthly New Tasks**: This performance indicator is defined using the record definition C.42 (Daily New Tasks). The indicator makes use of the organization records corresponding to the new tasks created daily for the last 30 days. It selects the values for the attribute *tid*, task identifier and computes their total count.

```
SELECT count (*) FROM
```

```
(
  SELECT distinct value FROM OrgRecordVal
  WHERE org_record_id IN
  (
    SELECT id FROM OrgRecord
    WHERE time < sysdate
      AND
    time > sysdate  - interval '30' day
      AND
    lcrecord_id =
    (
      SELECT ID FROM LCRECORD
      WHERE NAME LIKE 'Daily New Tasks'
    )
      AND
    org_id = $orgID
  )
    AND
  lcr_attr_id =
  (
    SELECT ID FROM LCRECORDATTRIBUTE
    WHERE NAME LIKE 'tid'
      AND
    LCRECORD_ID =
    (
      SELECT ID FROM LCRECORD
      WHERE NAME LIKE 'Daily New Tasks'
    )
  )
)
```

16. **Total Todo Lists**: This performance indicator is defined using the record definition C.41 (Daily TodoLists). The indicator makes use of the organization records corresponding to the todo lists reported lately. It selects the values for the attribute *tlid*, todo list identifier and computes their total count.

```
SELECT count (*) FROM
(
  SELECT distinct value FROM OrgRecordVal
  WHERE org_record_id IN
  (
    SELECT id FROM OrgRecord
    WHERE time < sysdate
      AND
    time > sysdate  - interval '1' day
      AND
    lcrecord_id =
    (
      SELECT ID FROM LCRECORD
      WHERE NAME LIKE 'Daily TodoLists'
    )
      AND
    org_id = $orgID
  )
    AND
```

```
  lcr_attr_id =
  (
    SELECT ID FROM LCRECORDATTRIBUTE
    WHERE NAME LIKE 'tlid'
      AND
    LCRECORD_ID =
    (
      SELECT ID FROM LCRECORD
      WHERE NAME LIKE 'Daily TodoLists'
    )
  )
)
```

17. **Percentage of tasks completed to tasks created in a day** This performance indicator is defined using two record definitions C.42 (Daily New Tasks) and C.44 (Daily Closed Tasks). The indicator makes use of the organization records corresponding to the new tasks created daily as well as the tasks completed daily for the last 30 days. It selects the values for the attribute *tid*, task identifier for both different types of records and computes the total count of each to compute the percentage

```
WITH count1 AS
(
  SELECT count (*) c1 FROM
  (
    SELECT distinct value FROM OrgRecordVal
    WHERE org_record_id IN
    (
      SELECT id FROM OrgRecord
      WHERE time < sysdate
        AND
      time > sysdate  - interval '30' day
        AND
      lcrecord_id =
      (
        SELECT ID FROM LCRECORD
        WHERE NAME LIKE 'Daily Closed Tasks'
      )
        AND
      org_id = $orgID
    )
      AND
    lcr_attr_id =
    (
      SELECT ID FROM LCRECORDATTRIBUTE
      WHERE NAME LIKE 'tid'
        AND
      LCRECORD_ID =
      (
        SELECT ID FROM LCRECORD
        WHERE NAME LIKE 'Daily Closed Tasks'
      )
    )
```

```
  )
)
, count2 AS
(
  SELECT count (*) c2 FROM
  (
    SELECT distinct value FROM OrgRecordVal
    WHERE org_record_id IN
    (
      SELECT id FROM OrgRecord
      WHERE time < sysdate
        AND
      time > sysdate  - interval '30' day
        AND
      lcrecord_id =
      (
        SELECT ID FROM LCRECORD
        WHERE NAME LIKE 'Daily New Tasks'
      )
        AND
      org_id = $orgID
    )
      AND
    lcr_attr_id =
    (
      SELECT ID FROM LCRECORDATTRIBUTE
      WHERE NAME LIKE 'tid'
        AND
      LCRECORD_ID =
      (
        SELECT ID FROM LCRECORD
        WHERE NAME LIKE 'Daily New Tasks'
      )
    )
  )
)
 SELECT c1/c2*100 FROM count1,count2
```

18. **Total Monthly New Campaigns**: This performance indicator is defined using the record definition C.51 (Daily New Campaigns). The indicator makes use of the organization records corresponding to the new campaigns created daily for the last 30 days. It selects the values for the attribute *cmid*, campaign identifier and computes their total count.

```
SELECT count (value) FROM OrgRecordVal
WHERE org_record_id IN
(
  SELECT id FROM OrgRecord
  WHERE time < sysdate
    AND
  time > sysdate  - interval '30' day
    AND
  lcrecord_id =
  (
```

```
    SELECT ID FROM LCRECORD
    WHERE NAME LIKE 'Daily New Campaigns'
  )
    AND
  org_id = $orgID
)
  AND
lcr_attr_id =
(
  SELECT ID FROM LCRECORDATTRIBUTE
  WHERE NAME LIKE 'cmid'
    AND
  LCRECORD_ID =
  (
    SELECT ID FROM LCRECORD
    WHERE NAME LIKE 'Daily New Campaigns'
  )
)
```

19. **Monthly Click Throughs of Campaign**: This performance indicator is defined using the record definition C.52 (Daily Campaign Statistics). The indicator makes use of the organization records corresponding to the daily registered campaign statistics for the last day. It selects the values for the attributes *cmid*, campaign identifier and *cmctr*, the campaign click count from these records.

```
WITH c AS
(
  SELECT * FROM OrgRecordVal
  WHERE org_record_id IN
  (
    SELECT id FROM OrgRecord
    WHERE time < sysdate
      AND
    time > sysdate  - interval '1' day
      AND
    lcrecord_id =
    (
      SELECT ID FROM LCRECORD
      WHERE NAME LIKE 'Daily Campaign Statistics'
    )
      AND
    org_id = $orgID
  )
    AND
  lcr_attr_id =
  (
    SELECT ID FROM LCRECORDATTRIBUTE
    WHERE NAME LIKE 'cmname' AND LCRECORD_ID =
    (
      SELECT ID FROM LCRECORD
      WHERE NAME LIKE 'Daily Campaign Statistics'
    )
  )
),
```

```
d AS
(
  SELECT * FROM OrgRecordVal
  WHERE org_record_id IN
  (
    SELECT id FROM OrgRecord
    WHERE time < sysdate
      AND
    time > sysdate  - interval '1' day
      AND
    lcrecord_id =
    (
      SELECT ID FROM LCRECORD
      WHERE NAME LIKE 'Daily Campaign Statistics'
    )
      AND
    org_id = $orgID
  )
    AND
  lcr_attr_id =
  (
    SELECT ID FROM LCRECORDATTRIBUTE
    WHERE NAME LIKE 'cmctr'
      AND
    LCRECORD_ID =
    (
      SELECT ID FROM LCRECORD
      WHERE NAME LIKE 'Daily Campaign Statistics'
    )
  )
)
SELECT c.value AS cmid ,d.value AS clicks FROM c,d
WHERE c.org_record_id = d.org_record_id AND c.idx = d.idx
```

20. **Monthly Forwards of Campaign**: This performance indicator is defined using the record definition C.52 (Daily Campaign Statistics). The indicator makes use of the organization records corresponding to the daily registered campaign statistics for the last day. It selects the values for the attributes *cmid*, campaign identifier and *cmfr*, the campaign forwards count from these records.

```
WITH c AS
(
  SELECT * FROM OrgRecordVal
  WHERE org_record_id IN
  (
    SELECT id FROM OrgRecord
    WHERE time < sysdate
      AND
    time > sysdate  - interval '1' day
      AND
    lcrecord_id =
    (
      SELECT ID FROM LCRECORD
      WHERE NAME LIKE 'Daily Campaign Statistics'
```

```
    )
      AND
    org_id = $orgID
  )
    AND
  lcr_attr_id =
  (
    SELECT ID FROM LCRECORDATTRIBUTE
    WHERE NAME LIKE 'cmname'
      AND
    LCRECORD_ID =
    (
      SELECT ID FROM LCRECORD
      WHERE NAME LIKE 'Daily Campaign Statistics'
    )
  )
),
d AS
(
  SELECT * FROM OrgRecordVal
  WHERE org_record_id IN
  (
    SELECT id FROM OrgRecord
    WHERE time < sysdate
      AND
    time > sysdate  - interval '1' day
      AND
    lcrecord_id =
    (
      SELECT ID FROM LCRECORD
      WHERE NAME LIKE 'Daily Campaign Statistics'
    )
      AND
    org_id = $orgID
  )
    AND
  lcr_attr_id =
  (
    SELECT ID FROM LCRECORDATTRIBUTE
    WHERE NAME LIKE 'cmfr'
      AND
    LCRECORD_ID =
    (
      SELECT ID FROM LCRECORD
      WHERE NAME LIKE 'Daily Campaign Statistics'
    )
  )
)
SELECT c.value AS cmid ,d.value AS forwards FROM c,d
WHERE c.org_record_id = d.org_record_id AND c.idx = d.idx
```

C.5 Test Data for Web Services

For every web service, we present the various test data that we manually entered using their forms (using browsers). The test data can also be added to the respective web services by making use of their APIs. The following test data were used for testing DaWeS.

Basecamp

1. Create two projects
   (a) Design: Then create two Todo-Lists within the Project 'Design'
      i. Database Design: Then create the following todos
         A. Relations for storing the record definitions (and mark it as done)
         B. Relations for storing the performance indicators (and mark it as done)
         C. Relations for storing the web service descriptions
         D. Relations related to organization and their data
      ii. Modules Design: Then create the following todos
         A. Design Answer Builder
         B. Design Generic Wrapper
         C. Scheduler
         D. Performance Indicator Computation
   (b) Development: Then create one Todo-List within the Project 'Development'
      i. Modules Development: Then create the following todos
         A. Develop Answer Builder
         B. Develop Generic Wrapper
         C. Scheduler
         D. Performance Indicator Computation

Liquid Planner

1. Create two projects
   (a) Unit Testing: Then create the following tasks
      i. Answer Builder
      ii. Generic Wrapper
      iii. Performance Indicator Computation

(b) Integrated Testing: Then create the following tasks
   i. Testing the calibration
   ii. Scheduler

Teamworkpm

1. Create two projects
   (a) Documentation: Then create two Todo-Lists
      i. Database Documentation: Then create the following todos
         A. Describe the relations for storing the record definitions (and mark it as done)
         B. Describe the relations for storing the performance indicators (and mark it as done)
         C. Describe the relations for storing the web service descriptions
         D. Describe the relations related to organization and their data
      ii. Modules Documentation: Then create the following todos
         A. Document Answer Builder
         B. Document Generic Wrapper (Mark it as done)
         C. Scheduler
         D. Performance Indicator Computation
   (b) System Requirement and Analysis (and mark this project as archived)

Zoho Projects Create a Project 'Client communication'

MailChimp: Create three campaigns

1. Alpha Release Campaign
2. Beta Release Campaign
3. Product Release Mailchimp Campaign (Mark it as draft)

CampaignMonitor: Create three campaigns

1. Alpha Release Campaign
2. Beta Release Campaign
3. Product Release CampaignMonitor Campaign (Mark it as draft)

iContact: Create two campaigns

1. Alpha Release Campaign (Mark it as draft)
2. Product Release iContact Campaign (Mark it as draft)

Zendesk

1. Create four tickets
   (a) Add support for a new web service (Normal Priority)
   (b) Add support for a new performance indicator (High Priority)
   (c) Add support for a new record definition (Normal Priority)
   (d) Response transformation not working for the given example (High Priority and mark it as fixed)

Desk: Create three tickets

1. Make a request to open an account (Priority: 9)
2. Make a call to the web service provider (Priority: 1)
3. Request the web service provider for a feature request (Priority: 2)

Zoho Support: Create four tickets

1. Request for a new feature (Low Priority)
2. Add support for a new web service (Normal Priority)
3. Add support for a new performance indicator (High Priority)
4. Add support for a new record definition (Low Priority)

Uservoice: Create three tickets

1. Add a new test case for the response transformation (mark it as Closed/done)
2. Study the new web service feature request
3. Add a new domain of web services

FreshDesk

1. Create three tickets
   (a) Correct typos in the documentation (Medium Priority)
   (b) Add a new section called Experiments in the documentation (High Priority)

   (c) Add new subsections Setup and Results in the section Experiments (High Priority)

2. Create two forums
   (a) Product Release Announcements and Create three new topics
      i. Alpha Release
      ii. Beta Release
      iii. Product version 1.0 Release
   (b) Request for Features and Create three new topics
      i. Add support for new Performance Indicator
      ii. Add support for new Record Definitions
      iii. Add support for new Web service

Appendix D

DaWeS: Manual

In this chapter we first take a look at the syntax for the datalog query used in DaWeS (section D.1). Then we take a detailed look at the various relations (SQL Tables in section D.2) used to store the web service definitions, record definitions, performance indicator queries and the enterprise data. DaWeS command line options are discussed in section D.3. Section D.4 presents the java interface for various DaWeS options.

D.1 Syntax for writing Datalog query

We use IRIS [IRIS, 2008] as the datalog engine for DaWeS. We use IRIS syntax to specify the LAV Mapping between local schema relations and the global schema relations and to define the datalog query for the record definitions. Let's first see the syntax for the specifying atoms and facts in IRIS. A literal (an atom or a fact) specified by the name followed by the list of terms in brackets. If a term is a constant, it is specified in quotes('), else if it is a variable, the variable name is prefixed with the question mark(?). If it is a fact, it must end with a period(.)

Example D.1.1. *Example facts*

```
Project('1', 'Documentation', 'Active', '2013-12-11').
Project('2', 'Development', 'Active', '2013-12-12').
```

Example of an atom

```
Project(?identifier, ?name, ?status, ?creation_date)
```

■

For a conjunctive query, the head of the conjunctive query (a literal) is followed by the symbols (:-) and finally followed by comma separated literals in the body. This is finally followed by a period (.).

Example D.1.2. *Following is a conjunctive query to get the identifiers and names of active projects.*

```
q(?identifier,?name) :- Project(?identifier, ?name, 'Active', ?date).
```

■

Recall that we use conjunctive query to specify the LAV mapping.

For a datalog query that consists of one or more conjunctive queries, every conjuntive query is specified as mentioned below. In order to distinguish between the query predicate and other predicates, the query predicate name is preceded by symbols (?-) and terminated by (.)

Example D.1.3. *Following is a datalog query (precisely, union of conjunctive query) to get the identifiers and names of active and archived projects. Note that we specify also the facts and the query predicate.*

```
Project('1', 'Documentation', 'Active', '2013-12-11').
Project('2', 'Development', 'Active', '2013-12-12').
q(?identifier,?name) :- Project(?identifier, ?name, 'Active', ?date).
q(?identifier,?name) :- Project(?identifier, ?name, 'Archived', ?date).
?-q(?identifier,?name).
```

On giving the above input to the IRIS datalog engine, it gives the following output

```
('1', 'Documentation')
('2', 'Development')
```

■

D.2 Relations

DaWeS Database is used for storing web service description, global (mediated) schema, record schema, performance indicators queries, enterprise authentication parameters, enterprise records and performance and indicators. They are shown in the Figures 5.5, 5.6, 5.7, 5.8, 5.9, 5.10, D.1 and D.2. In all these tables ID is the primary key. Figure 5.5 shows the information related to the web service. For every web service, we collect the

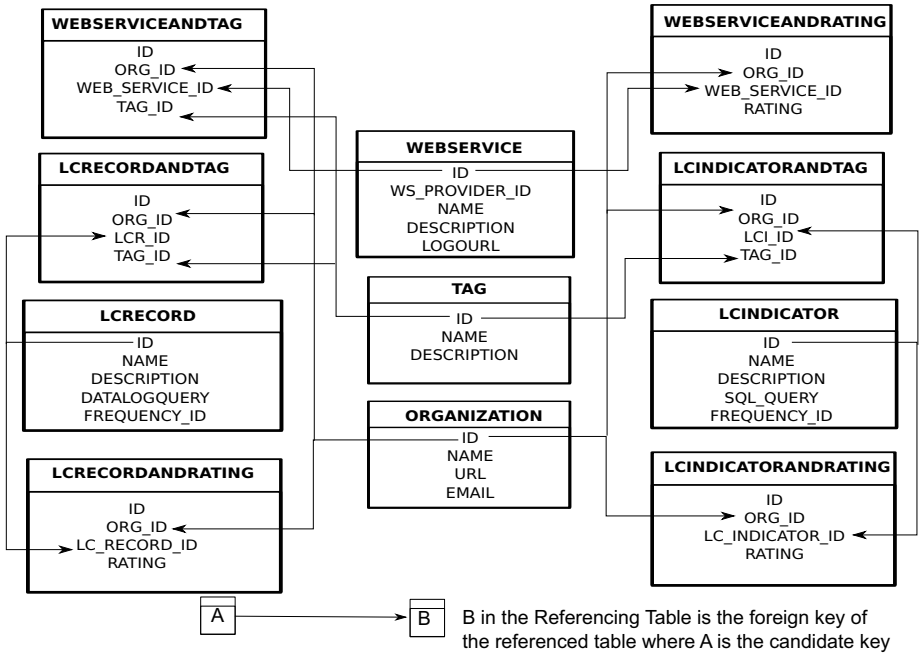

Figure D.1: Organization Tags and Ratings

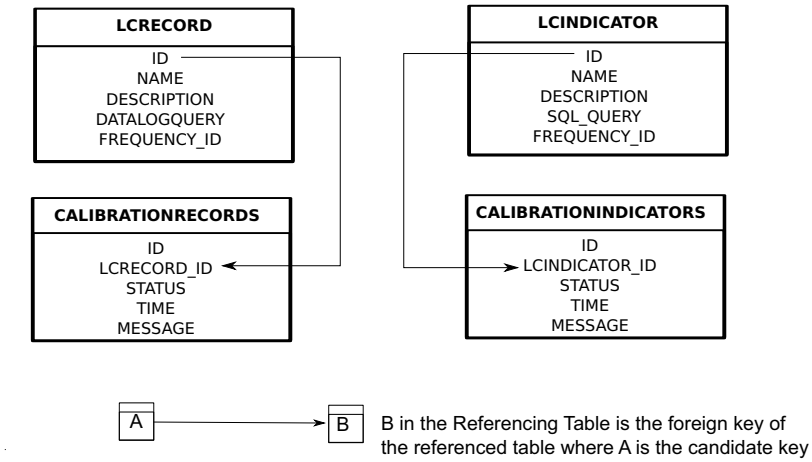

Figure D.2: Organization Tags and Ratings

information regarding the various categories (or domains) it belongs to. Examples of categories include *Project Management*, *Email Marketing* etc. The administrator also registers the various API of the web services and the details of the service providers. Figure 5.6 describes the various information that is essential to describe the web service API. For every API, the administrator must collect the information related to the message formats, state (current, deprecated or active), authentication parameters required from the enterprises and from DaWeS Administrator (for OAuth 1.0). It also shows the various details captured for every API operation like the expected response schema (XSD), the desired transformation (using XSLT) and HTTP details of request. Every API operation has an associated local schema relation as shown in Figure 5.7. Local schema relations are described using the global schema relations using LAV mapping (conjunctive query). Both local schema and global schema relations have their attributes and the corresponding data types described. Figure 5.8 shows how records (datalog queries) and performance indicator queries are stored. Records definitions are (recursive) datalog queries. Every record definition and every performance indicator has an associated frequency of execution that tells the scheduler (section 5.2.2.2) how often they must be computed. They also have associated calibration test data (section 5.2.2.3) to ensure their proper computation. Figure 5.9 captures every information required from the enterprise (or organization), i.e., organization details, authentication parameters for the web services, the interested records and performance indicators. Figure 5.10 shows how enterprise records and performance indicators are stored. In some cases, record or performance indicator may result in failures. This information is also captured. Figure D.1 shows how organization can tag and rate the web services, the records and performance indicators. Figure D.2 shows how the current calibration (section 5.2.2.3) status and performance indicators are handled.

We now take a detailed look at the various relations used in DaWeS along with their attributes. In the following figures, P denotes primary key F denotes the foreign key, U denotes whether unique attributes and $*$ denotes that the value cannot be NULL. Arrows signify the foreign key attributes.

D.2.1 Web Service

To define a web service, one requires to know its domain(category), the service provider name. A web service provider may provide one or more web services. And a web service may belong to one or more different categories. Refer the Figure D.3 to see the tables involved to define the details of a web service.

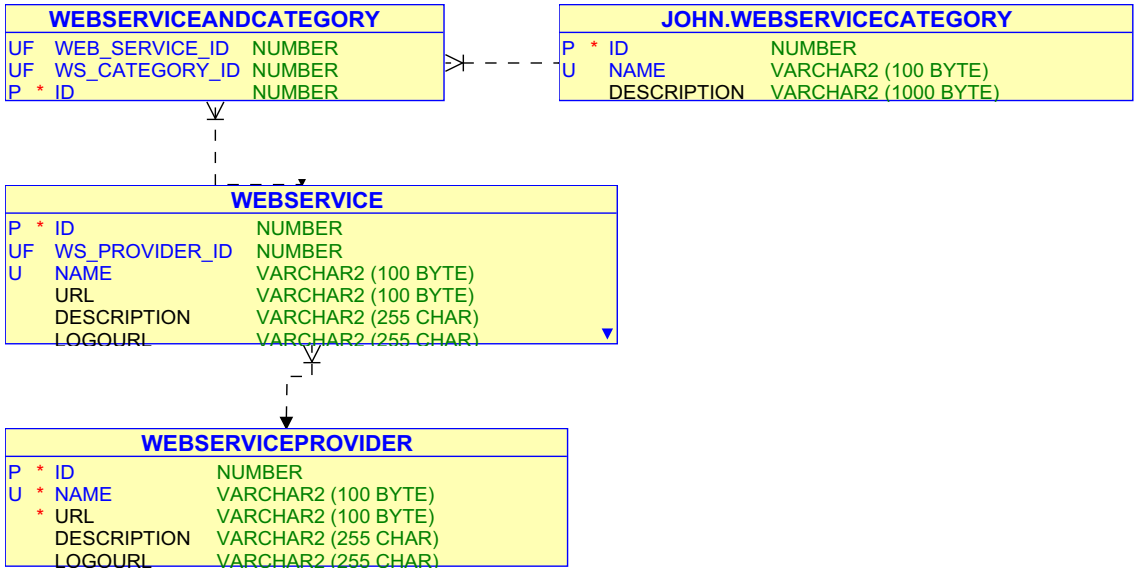

Figure D.3: Details of SQL Tables Related to Web Service

We now discuss the various tables in detail.

- **WEBSERVICECATEGORY**: To define various possible web service domains or categories. Every web service explicitly mentions its domain in its web site. Examples include project management, email marketing. The attributes of this relation include identifier, name and description.
- **WEBSERVICEPROVIDER**: A web service provider can provide one or more web services. This table captures the information of the web service provider like name and its URL. Considering our example, Zoho web service provider offers various services including Zoho Support and Zoho Projects. The attributes of this relation include identifier, name and description.
- **WEBSERVICE**: This table captures the information related to a web service, such as its name, URL, service provider and a brief description of its service as described on their website.
- **WEBSERVICEANDCATEGORY**: A web service can have one or more category. The various web service categories defined above can be referred for this purpose. The attributes of this relation include identifier, a reference identifier to the web service category and a reference to the web service.

D.2.2 Web Service API

Now we discuss about the tables used to store the details of a web service API and all the relevant operations. Refer the figure D.4 to see the tables involved.

Described below is the description of various tables in detail.

- **WEBSERVICEAPI**: It is used to store the details of a web service API. A web service can have more than one API, some of them may be in deprecated state. This table captures the information about its URL, any common http header or body information to be sent. To specify any variable use $ along with the name. Take for example, in a web service API operation, if domain is variable, specify it as *$domain.example.com*.
- **WEBSERVICEAPISTATE**: A web service can be in various states like active or deprecated. This information is captured in this table. The attributes include the identifier, a reference to the concerned web service and the state.
- **MESSAGEFORMAT**: Web services API use different formats to communicate. Popular message formats include XML and JSON. This table is used to store the details of various message formats. The attributes include identifier, name and description.
- **HTTPMETHOD**: This tables is used to store the HTTP method details. The attributes include identifier, name and description. Examples of HTTP method include GET, POST, PUT and DELETE.
- **WSAPIMESSAGEFORMAT**: This table is used to store the message format used for the communication between the web service and the API service users. The attributes include identifier, reference to the web service API and reference to the message format.
- **WEBSERVICEAPIOPERATION**: This table captures the information about the various web service API operations, the URL, the HTTP header, the HTTP body contents. To specify any variable use $ along with the name. Take for example, in a web service API operation, if an identifier is obtained from some other web service API operation(s), specify it as $var_name. Note that the URL is relative to the URL specified in the WEBSERVICEAPI table.
- **WSAPIOPSCHEMA**: This table contains the schema of the Web service Operation response. Note that we use the XML Schema to store this information. Therefore it must be remembered that any json response will be internally transformed to XML. The root element of a transformed JSON (to XML) is <json>
- **WSAPIOPTRANSFORM**: To make sure that every web service operation

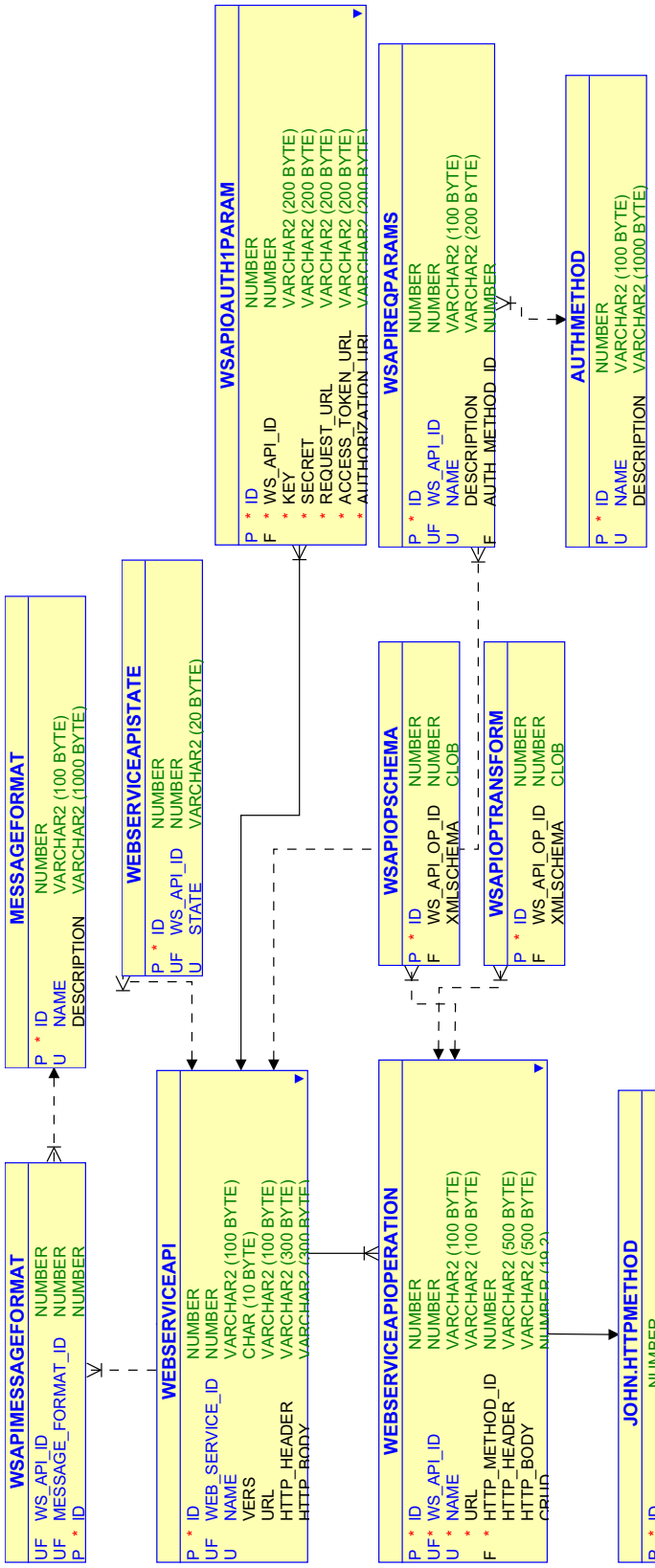

Figure D.4: Details of SQL Tables Related to Web Service API

is properly transformed to the required information, we make use of the XSLT transformation. As mentioned before, internally we transform the data to XML, therefore this fact must be considered while working with json format message responses. The root element of a transformed JSON (to XML) is <json>

- **WSAPIOAUTH1PARAM**: If a web service API uses OAuth 1.0 authentication, this stores the API key (of the administrator), API secret (of the administrator) and various authentication end points (URL) of the web service for OAuth 1.0.
- **WSAPIREQPARAMS**: This table consists of the information that must be obtained from the user (enterprise) in order to make the API calls on its behalf. Examples include authentication parameters like username, password.

D.2.3 Global and Local Schema Relations

Figure D.5 explains the tables used to store local and global schema relations and their associated data constraints. It also shows the attribute used for the LAVMapping.

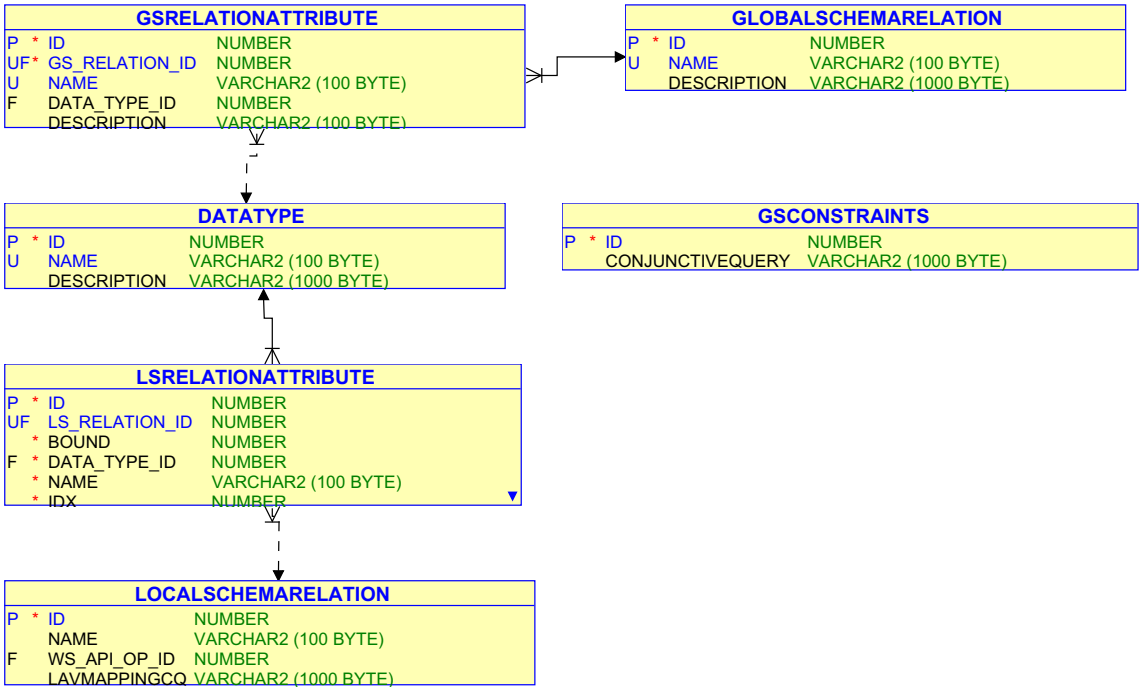

Figure D.5: Details of SQL Tables Related to Local and Global Schema

We now discuss the various tables in detail.

- **DATATYPE**: This table is used to define different data types. Examples include integer, string, float, project identifier, basecamp project identifier. The attributes include identifier, name and description.
- **GLOBALSCHEMARELATION**: Global schema relations are used to define new record definitions. Examples include Project, Campaign, Task. The attributes include identifier, name and description.
- **GSRELATIONATTRIBUTE**: This is used in conjunction with table GLOBALSCHEMARELATION. This contains the details of every attribute used in the global schema relation, its index (note that index starts from 1). The attributes include identifier, name, description, reference to the concerned global schema relation and reference to the data type.
- **GSCONSTRAINTS**: This is used to specify any constraints on the global schema. Currently it is used to support full and functional dependeny. The attributes include identifier, and conjunctive_query. The attribute conjunctive_query in the table has the following format (for the functional and full dependencies)

```
E(x...z) :- GSRel1(...),GSRelN(...),EQUAL(..)....EQUAL(..).
```

  Note there is a period(.) at the end.
- **LOCALSCHEMARELATION**: This is the transformed output of a web service response expressed like a relation. Therefore it contains the details of the relevant web service API operation. The attributes include the identifier, reference to the concerned web service API operation, name and the LAV Mapping between the current local schema relation and the global schema relations. Note that the LAV mapping is a conjunctive query in DaWeS.
- **LSRELATIONATTRIBUTE**: This is used in conjunction with the table LOCALSCHEMARELATION to specify the attributes and their index (note that index starts from 1). The attribute includes the identifier, reference to the respective local schema relation, name, bound status(whether the attribute in the relation is bound/input or not; hence it takes the values 1 when the concerned attribute is bound and 0, when it is not), index and reference to the data type.

D.2.4 Record Definitions and Performance Indicator Queries

The global schema relations are used to store the record definitions (datalog query formulated over the global schema relations). These record definitions are in turn

used to define the performance indicator queries. Both the record definitions and the performance indicator queries must be periodically calibrated in order to make sure that they perform the computation correctly. Figure D.6 shows the tables related to store the record definitions and performance indicator queries. It also shows the tables related to perform the calibration of the record definitions and the performance indicator queries.

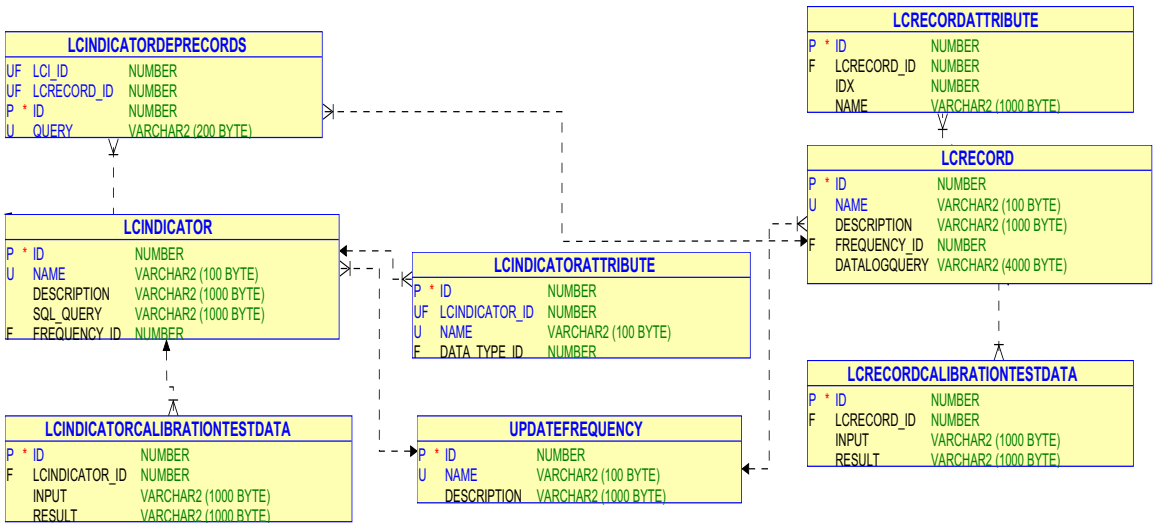

Figure D.6: Details of SQL Tables Related to Record Definitions and Performance Indicators

We now discuss the various tables in detail.

- **UPDATEFREQUENCY**: It is used to store different frequencies of computation for the performance indicator and the record definitions. The examples include daily, weekly. The attributes include identifier, name and description.
- **LCRECORD**: A record definition is a (recursive) datalog query defined over the global schema relations. This table is used to store the record definition. The attributes include identifier, name, description, datalog query and reference to the frequency of computation.
- **LCRECORDATTRIBUTE**: This is used in conjunction with the LCRECORD table to specify the attributes and their index (note that index starts from 1). The attributes include identifier, reference to the concerned record definition, name and the index. Note that the data type of every term obtained after the query computation is a string. Hence we don't have any specific reference to the data type.

- **LCRECORDCALIBRATIONTESTDATA** Computation of a Record must be in accordance with what it is defined to do. The calibration test data has an associated input and desired result. The calibration input test data is fed to an record computing module and it's verified whether the obtained relation is the same as that expected. The attributes include the identifier, name, input test data and the desired result. Note that when the input test data is empty, it corresponds to the situation when the record computation is made over the data obtained from the web service(s).The calibration test data for a record will be a set of input relations with the name of dependent global schema relation names. Example Project('1','Project A', 'Open','2013-11-11'). The resultant data is also expressed as a relation q(...), where q can be replaced by any word used as the query predicate name.
- **LCINDICATOR**: An indicator makes use of the records. It contains the SQL query describing the computation of the indicator using the record definitions. The attributes include identifier, name, description, sql query using the record definition names and reference to the frequency of computation.
- **LCINDICATORATTRIBUTE**: The attributes of the indicator are stored in this table. This table is used in conjunction with LCINDICATOR table. The index of the attributes start from 1. The attributes of this table include identifier, reference to the concerned performance indicator query, name and reference to the data type.
- **LCINDICATORCALIBRATIONTESTDATA**: The calibration test data for a indicator will be a set of input relations with the name of dependent LCRECORD with spaces removed and the resultant data is expressed as a relation q(...). The attributes include the identifier, name, input test data and the desired result.
- **LCINDICATORDEPRECORDS**: This is to specify the dependent records for a performance indicator computation. The attributes include the identifier, reference to the concerned performance indicator, reference to the required performance indicator and a (SQL) query to perform various checks on the dependent records of the organizations (whether they are error-free or any other checks).

D.2.5 Enterprises, Enterprise Records and Enterprise Performance Indicators

The users of DaWeS are the various organizations(enterprises). Not every organization is interested in integrating with every web service available with DaWeS. Similarly there are a large number of performance indicators and record definitions. Therefore, every organization needs to specify the authentication parameters of every interested web service and the desired record definitions as well as the performance indicator queries. Figure D.7 shows the tables related to an organization. Figure D.8 shows the tables related to the organization data (result obtained from the record definition evaluation and performance indicator query evaluation).

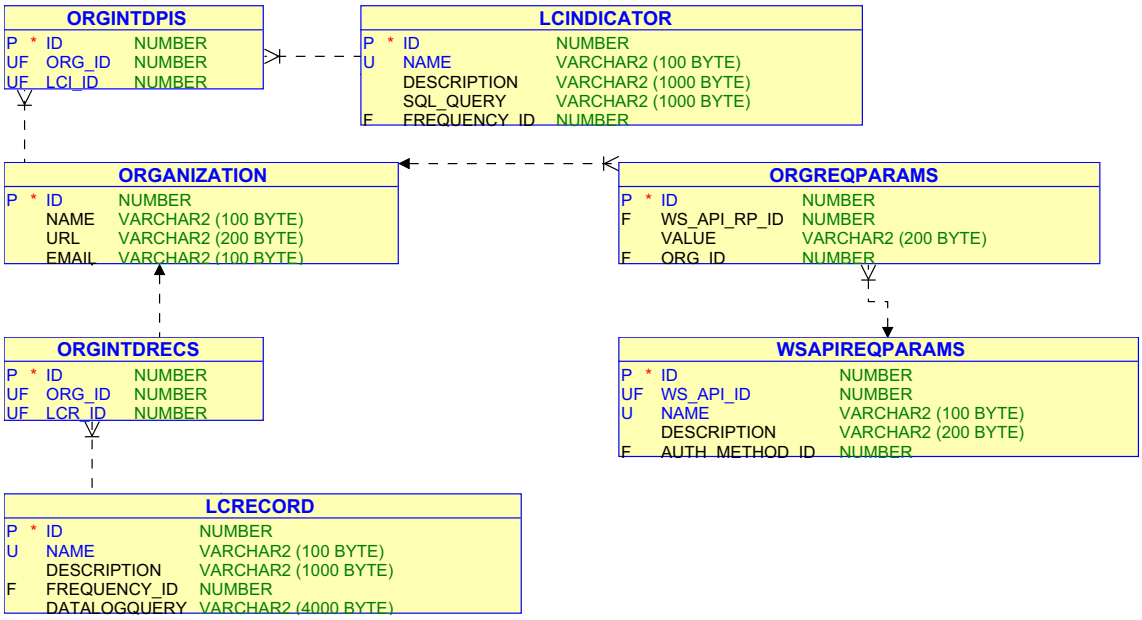

Figure D.7: Details of SQL Tables Related to Organization, its authentication params and interested Record Definitions and Performance Indicator

We now discuss the various tables in detail.

- **ORGANIZATION**: It is used to store the details of organizations (DaWeS end users/enterprises). The attributes of this table include identifier, url, name and email.
- **ERROR**: It is used to store the error encountered during the computation of a record or a performance indicator query. The attributes include identifier, error

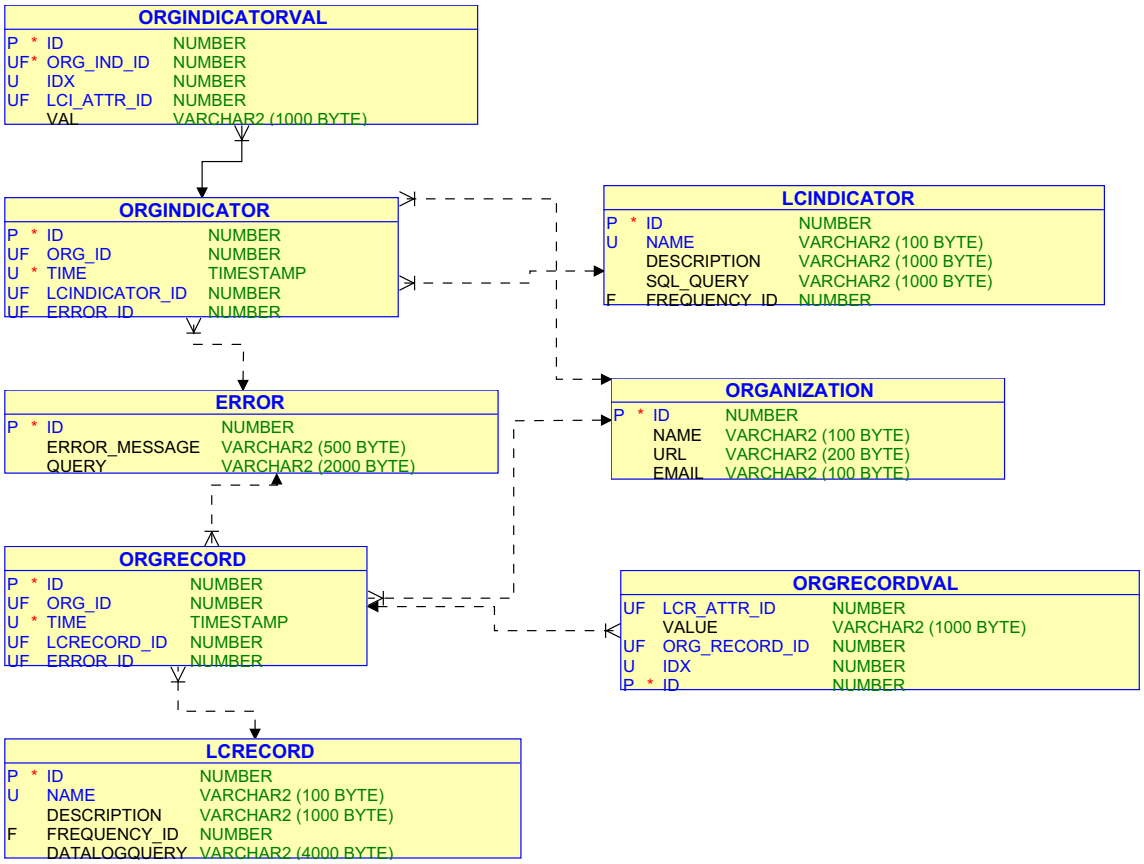

Figure D.8: Details of SQL Tables Related to Organization Data

message and the query that failed.

- **ORGREQPARAMS**: It is used to store the authentication parameters of the organizations for various web services. The attributes include the identifier, reference to the organization, reference to web service API required parameter and its value (Currently these values are plain text, but they can be hashed for security reasons).
- **ORGINTDRECS**: An organization may only be interested in some records. This table stores this information. Therefore the attributes include identified, reference to the organization and reference to the interested record definition.
- **ORGINTDPIS**: An organization may only be interested in some performance indicators. This table stores this information. Therefore the attributes include identified, reference to the organization and reference to the interested performance indicator query.
- **ORGRECORD**: The results obtained after the evaluation of the record defini-

tion are stored in this table along with the timestamp. The attributes include the identifier, reference to the concerned record definition, reference to the organization, time of evaluation and finally in case of error, a reference to the error occured.

- **ORGRECORDVAL**:It is used in conjunction with ORGRECORD table and contains the query results. The results obtained after the query evaluation consists of multiple tuples. Every tuple is identified by an index (index starts from 1). For every term in the tuple, there is an associated attribute (LCRECORDATTRIBUTE). Therefore the attributes of this table include identifier, reference to the record (computed), index of the tuple, reference to the attribute of the term and the obtained term value.
- **ORGINDICATOR**: The results obtained after the evaluation of the performance indicator query is stored in this table along with the timestamp. The attributes include the identifier, reference to the concerned performance indicator query, reference to the organization, time of evaluation and finally in case of error, a reference to the error occured (otherwise it is null).
- **ORGINDICATORVAL** It is used to store the results of organization indicator and is used in conjunction with ORGINDICATOR. The results obtained after the query evaluation consists of multiple tuples. Every tuple is identified by an index (index starts from 1). For every term in the tuple, there is an associated attribute (LCINDICATORATTRIBUTE). Therefore the attributes of this table include identifier, reference to the performance indicator (computed), index of the tuple, reference to the attribute of the term and the obtained term value.

D.2.6 Tags and Ratings

Different organizations have been given the option to tag the web services, record definitions and performance indicator queries of interest. They can also rate all of them with values 1 to 10 (with 10 being the highest rating). Figure D.9 shows the tables used to store the ratings and tags given by an organization.

We now discuss the various tables in detail.

- **TAG**: Tags for web services, records and indicators. The attributes include identifier, name and description (optional).
- **WEBSERVICEANDRATING**: Every user of the platform can specify a rating to the service. The rating ranges from 1-10, with 10 being the highest. The

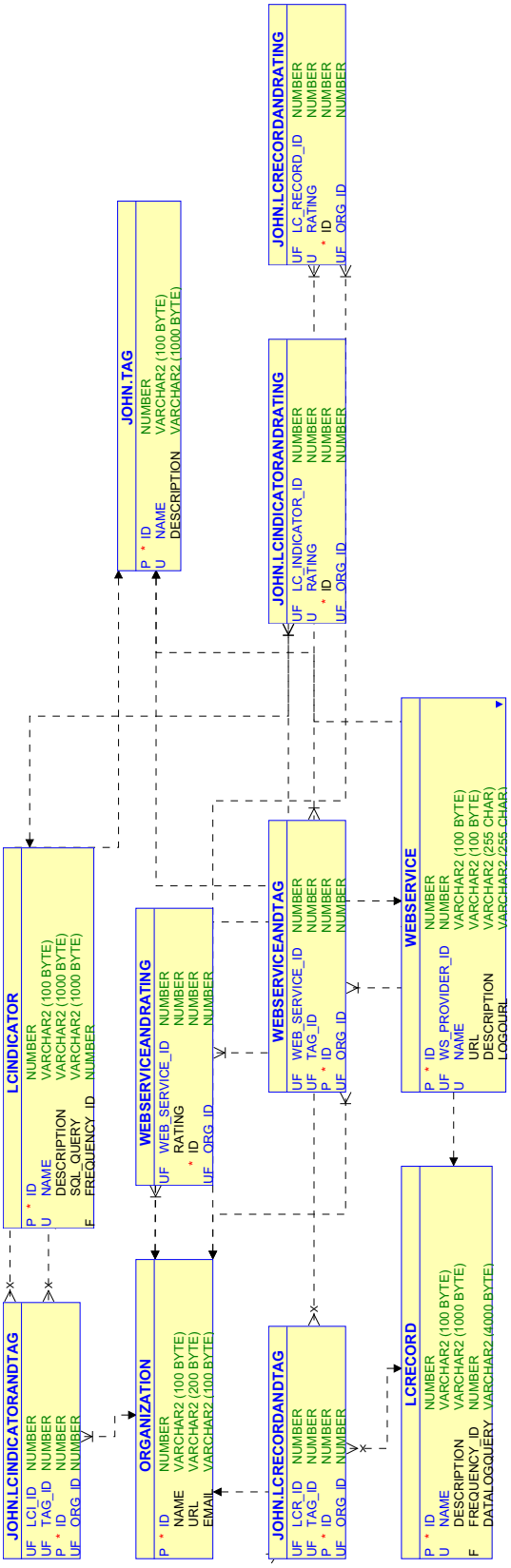

Figure D.9: Details of SQL Tables Related to Organization Tags and Ratings

attributes include the identifier, reference to the organization, reference to the web service and the rating.

- **WEBSERVICEANDTAG**: An enterprise can tag a web service to specify its purpose of usage. This is useful to search a web service based on various purposes of a service. The attributes include the identifier, reference to the organization, reference to the web service and reference to the tag.
- **LCRECORDANDTAG**: An organization can define a tag for a LCRecord. It can be any string of choice. The attributes include the identifier, reference to the organization, reference to the record definition and reference to the tag.
- **LCRECORDANDRATING**: An organization can rate a record a rating between 1 and 10, 10 being the highest. The attributes include the identifier, reference to the organization, reference to the record definition and the rating.
- **LCINDICATORANDTAG**: An organization can define tags for the indicator. It can be a string of any choice. The attributes include the identifier, reference to the organization, reference to the performance indicator query and reference to the tag.
- **LCINDICATORANDRATING**: An organization can rate an indicator a rating between 1 and 10, 10 being the highest. The attributes include the identifier, reference to the organization, reference to the performance indicator query and the rating.

D.2.7 Calibration Status and Error Details

We saw before the various calibration test data for the record definitions and performance indicator queries. Figure D.10 shows how the results of the calibration tests are stored.

- **CALIBRATIONRECORDS** It is used to store the calibration status of record definition computation. The attributes include identifier, reference to the concerned record definition, the time of calibration, status of calibration (whether passed or failed; if passed, the value is 1, else 0) and any remark after the calibration is performed (useful for diagnosis, useful after failure).
- **CALIBRATIONINDICATORS** It is used to store the calibration status of performance indicators computation. The attributes include identifier, reference to the concerned performance indicator, the time of calibration, status of calibration (whether passed or failed; if passed, the value is 1, else 0) and any remark after the calibration is performed (useful for diagnosis, useful after failure).

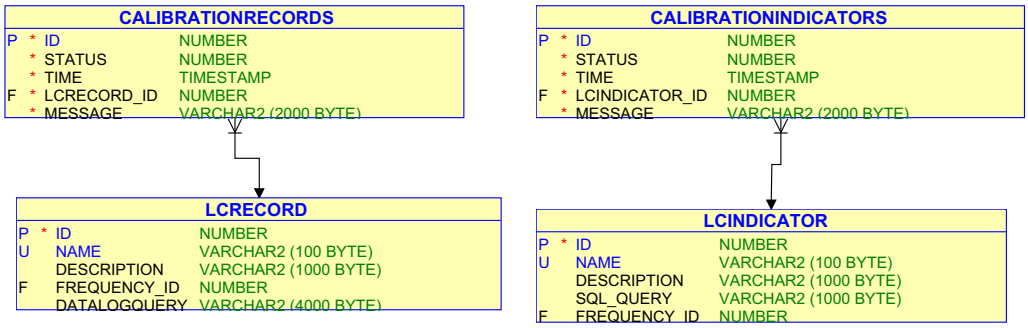

Figure D.10: Details of SQL Tables Related to Calibration

D.3 DaWeS: Command Line Manual

D.3.1 Name

dawes - Web Service fed Multi-Enterprise Data Warehouse

D.3.2 Synopsis

dawes [-irhv] [-csl r|ws|pi] [-o organization] [-p string] [-n Identifier]

D.3.3 Description

DaWeS is a Web Service fed multi-enterprise data warehouse. Using this application, users can add new web services to be integrated to the system, define new records and performance indicators. Users can also run the various unit tests and integration tests that comes alongwith the application. Thus the user can cali- brate the system to make sure that the system is working as it is expected. There's also a scheduler that runs to fetch the records and the performance indicators based on their expected frequency.

D.3.4 Options

- **-i or –initialize**: This option is used to initialize the system(to experiment the system with some default web ser- vices, records and performance indicators)

- **-c or –compute r|pi**: It is used to fetch or compute record or performance indicator respectively. Therefore it takes one of the values from r|pi option with the organization identifier as the argument. By default, if no specific identifier corresponding to the record or performance indicator is specified, all the records or performance indicators of the organization is computed. In order to compute a particular record or performance indicator, specify the identifier using the -n|–number option with the correspond- ing identifier as the argument.
- **-s or –search ws|r|pi**: It is used to search a new web service(ws), record(r) or performance indicator(pi). Therefore it takes one of the values from ws|r|pi -p|–pattern option. It can be a name or any desired string.
- **-l or –calibrate all|r|pi**: Run the unit tests and integration tests that comes along with the application It is used to calibrate a record, performance indicator or a web service. Therefore it takes one of the values from ws|r|pi By default, if no specific identifier corresponding to the web service or record or performance indi- cator is specified, all the web services, records or performance indicators are calibrated. In order to calibrate a particular web service, record or performance indicator, specify the identifier using the -n|–number identifier option with the corresponding identifier as the argument.
- **-r or –scheduler**: It is used to run the scheduler. The scheduler fetches the records from the web services and com- putes the performance indicators based on their associated frequency. Once the computation or fetching is complete, it sleeps till the next run. -t or –test
- **-h or –help**: Get the usage of tha application
- **-v or –version**: Get the version of the application
- **-o or –organization identifier**: It is used to specify the organization. The identifier of an organization is specified as an argument to this option. This option is used with the -d|–define and -c|–compute options.
- **-p or –pattern string**: This is used to specify a pattern to search along with the -s|–search option. The desired pattern is the argument to this option.
- **-n or –number identifier**: This is used to specify the identifier of a web service, record or performance indicator. The identi- fier is the argument to this option. This option is used with the -l|–calibrate and -c|–compute options.

D.3.5 Examples

- dawes -i
  It is used to initialize the system with initial set of data for running the unit tests

and calibration tests.

- dawes -s ws -p "project"
  It is used to search a web service with names matching project, or belonging to category '*project*' or having the tag '*project*'.
- dawes -s r -p "task"
  It is used to search a record having a name or tag '*task*'
- dawes -s pi -p "task"
  It is used to search a performance indicator having a name or tag '*task*'.
- dawes -l r
  It performs calibration for all the records
- dawes -l pi
  It performs calibration for all the performance indicators
- dawes -l r -n 23
  It performs the calibration of the record identified by the number 23
- dawes -l pi -n 23
  It performs the calibration of the performance indicator identified by the number 23
- dawes -c r -o 12 -n 15
  It fetches the record 15 of the the organization 12.
- dawes -c pi -o 12 -n 15
  It computes the performance indicator with the 15 of the the organization 12.
- dawes -r
  It runs the scheduler

D.3.6 Files

- config/application.ini
  It is used to configure the application. It is used to specify the username, password and the database schema.
- config/ecache.xml
  It is used to configure the internal caching mechanism.
- config/database.sql
  It contains the sql file for the initial setup
- config/log4j.properties
  It can be configured to configure the logging mechanisms.
- resource/
  It contains the files related to the unit tests. In general, this file is important to set

up an initial number of web services, their schema and response transformation files. This file doesn't require any modifications for the initial setup.

- man/
  It contains the installation manual.
- dawes.jar
  It is an executable jar
- dawes
  The script to run the application
- config/log4j.properties
  It can be configured to configure the logging mechanisms.
- resource/
  It contains the files related to the unit tests. In general, this file is important to set up an initial number of web services, their schema and response transformation files. This file doesn't require any modifications for the initial setup.
- man/
  It contains the installation manual.
- dawes.jar
  It is an executable jar
- dawes
  The script to run the application

D.4 DaWeS: Java Interfaces for Developers

DaWeS developers can refer the following files to add support for new web services, define new performance indicators and records.

- *com.littlecrowd.dataintegration.relations*: All the (relational) tables used in DaWeS.
- *com.littlecrowd.dataintegration.init*: It consists of a set of examples from the three domain of web services under consideration. These examples can be used to understand how new web services are added to the system, how local and global schema relations are defined, how the LAV mapping is done and finally how records and performance indicator queries are defined.

D.4.1 Interfaces

We take a look at various interfaces in DaWeS.

D.4.1.1 Adding a new domain in the Global Schema

When a new domain is added to the global schema, the following interface is used. It is used to add new global schema relations pertaining to the new domain.

```
public interface IGlobalSchema {
  /**
   * Creating or Updating new Global Schema relations of a new domain
   */
  public void createOrUpdateRelations() throws Exception;
}
```

D.4.1.2 Adding a new web service to DaWeS

In order to add a new web service to DaWeS, make sure that the associated web service provider and the API operations are added using the following interface:

```
public interface IWebServiceCreator {
  /**
   * Creating a new Web Service Provider
   */
  public void createWebServiceProvider() throws Exception;
  /**
   * Creating a new Web Service
   */
  public void createWebService() throws Exception;
  /**
   * Creating a new Web Service API
   */
  public void createWebServiceAPI() throws Exception;
  /**
   * Add the Web Service API operations
   */
  public void createWebServiceAPIOperations() throws Exception;
}
```

D.4.1.3 Adding a new organization

When a new organization is added to DaWeS, some default performance indicators and record definitions can also be added. In addition to this, the interface can also be used to add the authentication parameters for different web services.

```
public interface IOrganizationCreator {
  /**
   * Add a new organization
```

```
   */
  public void createOrganization() throws Exception;
  /**
   * Add the performance indicators that an organization is interested to
   * compute
   */
  public void createOrganizationInterestedPis()  throws Exception;
  /**
   * Add the records that an organization is interested to
   * compute
   */
  public void createOrganizationInterestedRecs()  throws Exception;
  /**
   * Add the authentication parameters for the web services that an
   * organization is intended to integrate with
   */
  public void createOrganizationReqParams()  throws Exception;
}
```

D.4.1.4 Adding new Record Definitions

The following interface can be used to add new record definitions related to a new domain (or domains):

```
public interface ILCRecordCreator {
  /**
   * Add the record definitionss related to a new domain (or domains)
   */
  public void createRecords() throws Exception;
}
```

D.4.1.5 Adding new Performance Indicator Queries

The following interface can be used to add new performance indicator queries related to a new domain (or domains):

```
public interface ILCIndicatorCreator {
  /**
   * Add the performance indicator queries related to a new domain (or domains)
   */
  public void createIndicators() throws Exception;
}
```

D.4.2 Options

We saw in section D.3 various command line options of DaWeS. We now discuss the java interfaces for those options. The options we mainly discuss here are the following:

1. *initialize* DaWeS with various default web services, global schema relations, record definitions and performance indicator queries.
2. *compute* the records and performance indicators of an organization
3. *search* a web service, record definition and performance indicator query
4. *calibrate* the record and performance indicator computation
5. *scheduler* to periodically perform the record and performance indicator computation

D.4.2.1 initialize

Given below is the interface to initialize DaWeS:

```
public class InitialData {
  public static void initialize() {
  }
}
```

D.4.2.2 compute

Given below are the interfaces to compute the records and performance indicators:

```
public class LittleCrowdRecord {
  /**
   * @param lcrID: Record Definition identifier
   * @param orgID: Organization identifier
   * @param print: whether to print the results on to the screen
   * Computes the given record of the organization
   */
  public static void compute(BigDecimal lcrID, BigDecimal orgID,
    boolean print) throws Exception {
  }
  /**
   * @param orgID: Organization identifier
   * Computes all the (interested) records of the organization
   */
  public static void compute(BigDecimal orgID) throws Exception {
  }
}
```

```
public class LittleCrowdPerformanceIndicator {
  /**
   * @param lcpID: Performance Indicator Query identifier
   * @param orgID: Organization identifier
   * @param print: whether to print the results on to the screen
   * Computes the given performance indicator of the organization
   */
  public static void compute(BigDecimal lcpID, BigDecimal orgID,
    boolean print) throws Exception {
  }
  /**
   * @param orgID: Organization identifier
   * Computes all the (interested) performance indicators of the organization
   */
  public static void compute(BigDecimal orgID) {
  }
}
```

D.4.2.3 search

Following is the interface to search a web service, record definition and performance indicator query recognized by the type ws, r and pi respectively:

```
public class Search {
  /**
   * @param type: Possible values include ws, r and pi
   * @param pattern: Pattern to search for
   * returns the search results
   */
  public static ISearchResult search(String type, String pattern)
    throws Exception {
  }
}
public abstract interface ISearchResult {
  public void addRow(ISearchResultRow row);
  public List<ISearchResultRow> get();
  public void print();
}

public abstract interface ISearchResultRow {
  public List<String> get();
}
```

D.4.2.4 calibrate

Following are the interfaces to calibrate the record and performance indicator computation:

```
public class CalibrationResult {
  /**
   * @param passed: Whether the calibration test passed
   * @param remark: Remark for calibration test success/failure
   * Constructor to set the calibration result
   */
  public CalibrationResult(boolean passed, String remark) {
  }
  /**
   * Returns whether the calibration passed or failed
   */
  public boolean isPassed() {
  }

  /**
   * Returns the remark related to the calibration test
   */
  public String getRemark() {
  }
}

public class LCRecordCalibration {
  /**
   * performs the calibration for all the record computation
   */
  public static void calibrate() {
  /**
   * @param lcrID: Record Definition Identifier
   * returns the calibration status
   * performs the calibration of a record definition
   */
  public static CalibrationResult calibrate(BigDecimal lcrID) {
}

public class LCPerformanceIndicatorCalibration {
  /**
   * performs the calibration of all performance indicator queries
   */
  public static void calibrate() {
  /**
   * @param lcpiID: Performance Indicator Query identifier Identifier
   * returns the calibration status
   * performs the calibration of the given performance indicator
   */
  public static CalibrationResult calibrate(BigDecimal lcpiID)
    throws Exception {
}
```

D.4.2.5 scheduler

Following is the interface to run the scheduler to perform the periodic computation of records and indicators.

```
public class SchedulerService {
  public SchedulerService() {
  }
}
```