IT Licentiate theses
2007-001

Querying Mediated Web Services

MANIVASAKAN SABESAN

UPPSALA UNIVERSITY
Department of Information Technology

UPPSALA
UNIVERSITET

Querying Mediated Web Services

BY
MANIVASAKAN SABESAN

February 2007

COMPUTING SCIENCE DIVISION
DEPARTMENT OF INFORMATION TECHNOLOGY
UPPSALA UNIVERSITY
UPPSALA
SWEDEN

Dissertation for the degree of Licentiate of Philosophy in Computer Science with specialization in Database Technology
at Uppsala University 2007

Querying Mediated Web Services

Manivasakan Sabesan
msabesan@it.uu.se

Computing Science Division
Department of Information Technology
Uppsala University
Box 337
SE-751 05 Uppsala
Sweden

http://www.it.uu.se/

ISSN 1404-5117
Printed by the Department of Information Technology, Uppsala University, Sweden

Abstract

Web services provide a framework for data interchange between applications by incorporating standards such as XMLSchema, WSDL SOAP, HTTP etc. They define operations to be invoked over a network to perform the actions. These operations are described publicly in a WSDL document with the data types of their argument and result. Searching data accessible via web services is essential in many applications. However, web services don't provide any general query language or view capabilities. Current web services applications to access the data must be developed using a regular programming language such Java, or C#.

The thesis provides an approach to simplify querying web services data and proposes efficient processing of database queries to views of wrapped web services. To show the effectiveness of the approach, a prototype, *web Service MEDiator system (WSMED*), is developed.

WSMED provides general view and query capabilities over data accessible through web services by automatically extracting basic meta-data from WSDL descriptions. Based on imported meta-data, the user can then define views that extract data from the results of calls to web service operations. The views can be queried using SQL. A given view can access many different web service operations in different ways depending on what view attributes are known. The views can be specified in terms of several declarative queries to be applied by the query processor. In addition, the user can provide semantic enrichments of the meta-data with key constraints to enable efficient query execution over the views by automatic query transformations. We evaluated the effectiveness of our approach over multi-level views of existing web services and show that the key constraint enrichments substantially improve query performance.

Acknowledgements

First and foremost I would like to thank my supervisor Professor Tore Risch for supervising me. I'm deeply appreciating his willingness to assist me in writing the thesis by providing valuable suggestions and fruitful comments. I am very grateful to him to sharing his precious knowledge with me and being always ready to discuss the new directions and the research problems. My second supervisor Dr.G.N.Wikramanayake is supporting me by his constructive advices and guidance and I appreciate his assistance. I'm grateful to Sida for providing financial assistance to continue my research work and I also wish to thank all Sri Lankan Sida split PhD program management committee members and Sida coordinator for Uppsala University, Richard Wait, for their great support all the time.

Dr.S.Mahesan and Dr.S.Kanaganathan are my first Computer Science teachers and embolden me as a research student in Computer Science. I would like to thank them for their rewarding guidance and assistance. Especially I'm forever grateful to Dr.Mahesan for his encouragement and directing me towards the precise opportunities for my triumphant carrier.

I am in debt to all present and past UDBL group members for helping and sharing with me difficulties and happiness. I am also like thank all my fellow Sida split PhD candidates for their friendship and support.

I'm grateful to my wife, Sutha and my kids for their generous support even I'm away from them.

Finally, I would like to dedicate this thesis to my parents, who have always encouraged me to study.

Contents

Abbreviations

AMOS	Active Mediator Object System
BPEL4WS	Business Process Execution Language for Web Services
CDM	Common Data Model
FTP	File Transfer Protocol
HTTP	Hypertext Transport Protocol
JDBC	Java Database Connectivity
OASIS	Organization for the Advancement of Structured Information Standards.
OQL	Object Query Language
RDBMS	Relational Database Management System
RDF	Resource Description Framework
RPC	Remote Procedure Call
SAAJ	SOAP with Attachments API for Java
SMTP	Simple Mail Transfer Protocol
SOAP	Simple Object Access Protocol
TCP	Transmission Control Protocol
UDDI	Universal Description, Discovery, and Integration
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
W3C	World Wide Web Consortium
WQL	WSMED Query Language
WSDL	Web Services Description Language
WSMED	Web Service MEDiator
XML	eXtensible Mark up Language

1. Introduction

The growth of the Internet and the emergence of XML for data interchange have increased the importance of web services [8] incorporating standards such as SOAP[25], WSDL [11] and XML Schema [19].Web services support an infrastructure for the applications by defining a set of *operations* that can be invoked over the communication network. Web service operations are self contained using metadata to describe data types of their argument and result, i.e. their signatures, using WSDL. An important class of operations is to search data accessible through web services. However, web services don't support any general query language or view capabilities.

As the applications query data from different web services, there is need for a system to efficiently integrate data from heterogeneous data sources accessible over the web services. *Mediators* are software to enable queries to different kinds of data sources. In this work we investigate methods to build such mediators for querying data provided through web services. The development of a web service based mediator prototype is expected to provide insights into a number of research questions:

1. To what extent can the web service standards, such as WDSL, SOAP, and XML-Schema, be automatically utilized by a mediator engine to query the sources efficiently and scalable?
2. How can views in a high level query language such as SQL be defined in terms of imported web service descriptions?
3. How can the modern query optimization and rewrite techniques be used to provide efficient and scalable access that optimally utilizes the limited data access and update capabilities of different web services?
4. What minimal set of extra semantic enrichment is needed in addition to the current web service standards in order to provide scalable access through the views?
5. How can the semantic enrichments be automatically detected and verified?

We have developed a system called *WSMED – Web Service MEDiator* - to enable high level and scalable queries over data retrieved through web services. WSMED can access dynamically any web service by retrieving the meta-data of a WSDL document describing service interfaces and then invoking the web service operations. WSMED uses a generic web service schema for representing any web service description by a WSDL document

that conforms to an XML schema, such as operation signatures and other properties. The meta-data are used to construct arguments in calls from WSMED to web service operations and to convert the result of a call to the format used in WSMED. Further it exploits the SOAP protocol to pack messages to invoke web service operations. The prototype makes use of HTTP [61] for transmission of message and is using WSDL, SOAP and XML Schema to wrap the sources accessible through the web services. This addresses research question one.

SQL view definitions called *WSMED views* are defined in terms of imported WSDL descriptions of web service operations. Furthermore, multi-level WSMED views can be defined in terms of other WSMED views. Web services often return nested XML structures (i.e. records and collections), which have to be flattened into relational views before they can be queried with SQL. The knowledge how to extract relevant data from a given web service is added by the user as queries called *search definitions.* For each search definition, the flattening is specified as an object-oriented query using the *WSMED query language* (WQL) that has support for web service data types. The result of a web service operation invocation is translated into data structures that are queried by the search definitions. Alternatively, XQuery [7] can also be used for the flattening but it requires more complicated conversion of each web service result into a temporary XML document.

By creating views and querying these views through SQL, we partially answered research questions two and three. The analysis of the update capabilities is subject to the future directions. Modern Query optimizations need to be investigated deeper in the future.

An important semantic enrichment is to allow for the user to associate with a given WSMED view different search definitions depending on what view attributes are known in a query. This is called the *binding pattern* of a search definition. The WSMED query optimizer automatically selects the optimal search definition for a given query by analyzing its used binding patterns.

To further improve query execution performance, the user can add *key constraints* when defining WSMED views. A WSDL operation signature description does not provide any information about which parts of the signature is a key to the data accessed through the operation. As we show, this information is critical for efficient query execution of multi-level WSMED views. Therefore, we allow the user to declare to the system all (compound) keys of a given WSMED view. To answer the research questions four and five regarding semantic enrichments we will need to study further.

2. Background

This chapter introduces a literature study of the background knowledge about the major enabling technologies for mediating web services. It briefly covers data base management systems, information integration, web services, and the core technologies involved with web services such as XML, WSDL, SOAP, and semantic web representations.

2.1. Database Management Systems

A software system that allows creating and manipulating the huge amount of data in a structured way is known as a *Database Management System (DBMS)*. A *database* is defined as the group of data managed by a DBMS. A DBMS facilitates the following:

- It allows the users to create a database and specifies its data types and structures known as a *database schema* through a *Data Definition Language (DDL).*
- It permits the users to insert, delete, update and query data from data bases through a *Data Manipulation Language (DML)*
- It provides a security system to support multilevel authentication control for the users
- It preserves the consistency of data through an integrity system
- It provides a recovery control system to restore the database to a previous consistent state after hardware and software failures, called transaction and recovery control.
- It provides a user-accessible catalogue, called the *schema* that contains meta-data of the data in the database.

To describe the data requirements of an organization in a readily understandable way by the users, a higher-level description language for schemas is required: that is known as the *data model* for the DBMS. DMBSs use different kind of data models. The evolution of DBMS follows development of new data models.

In the late 1960s the first commercial DBMSs was developed utilizing *hierarchical* and *network* data models. These data models highly focused on the physical data arrangement and storage of data, and they didn't support any high-level query languages. Navigation through a graph or tree of data elements was the only possible way for data retrieval. Therefore the users had to have detailed knowledge about the physical data arrangement.

The *relational* data model was introduced by Codd [13] at the beginning of the 1970s. It relaxes the users' burden of how to access data and *Relational Database Management Systems (RDBMS)* started to evolve. This model is based on mathematical relations that present the data to the users in two dimensional tables. Each cell of the table contains a data value with different atomic types such as string, character and numbers. Even though it resembles the traditional tabular data representations, internal storage structures are very complex in order to provide efficient data manipulation. Further, it supports a *high level query language* for efficient data base programming. The functional *relational algebra* and declarative *relational calculus* are the major primitives to specify a query.

There are number of query languages emerged based on these formalisms, among them the *Structured Query Language (SQL)* became the de facto language. The SQL query processing modules transfer the declarative queries into an *execution plan*, which is a program to specify in details how the data is retrieved. Further it allows creating *views*: they resemble virtual relations defined through a query expression, but do not exist physically and can be queried as they exist physically. It is sometimes possible to modify views by an insertion, deletion, or update.

The relational model provides *data independence* by separating the high-level query language from the low level implementations details. There are two kinds of data independence: *physical* and *logical*. Physical data independence means the capability of changing the physical structure of data without affecting the applications, while logical data independence refers to the immunity of the conceptual changes to the application programs.

The applications from the new areas such as computer aided engineering, geographic information systems, and multimedia require complex data representations exposed the limitations of the relational model and they demanded for a new generation DBMSs: *Object Oriented Database Management Systems (OODBMS)* based on *object–oriented (OO)* data model. The *objects* are classified in *classes*. A class consists of a *type* and *methods* that can be executed on objects of the class. A powerful type system is represented with primitive *atomic* types, *record structures, collection types (sets, bags, arrays)* and *reference types (pointers)*. Also complex types could be defined by repeatedly apply record-structure and collection operators. Each object is uniquely identified by an *object identity (OID).* Classes are arranged according to a class hierarchy. That is, each class can be defined as a *sub class* of another and *inherits* all properties from some other classes with *overloading* and *overriding* characteristics.

The OODBMSs are implemented by extending object-oriented languages such as C++ or Java with database capabilities such as persistence, concurrency control, and recovery. The object–oriented model enriches the database with features to become more powerful in modeling real world objects for the new applications. Early OODBMSs could not support any

declarative querying facilities. Queries were specified through navigating the graph structures where arcs are defined by OIDs stored as attribute values of other objects. The ODMG (Object Data Management Group) [69] developed a standard query language for OODBMSs that consists O*bject Definition Language (ODL)* and *Object Query Language (OQL)*. The OODMS are well suited for the applications that process complex data with less significant query requirements such as computer-aided design packages.

By combining the declarative power of RDBMSs with the modeling power of OODBMSs another innovative kind of DBMS, *Object Relational Database Management System (ORDBMS)*, was introduced and it is now broadly used in commercial products. The object-relational model incorporates extensions of the relational model with the following features:

- Extensible base type system by *User Defined Types (UDT)* that can be introduced along with user defined functions, operators, and aggregates operating on the values of these types;
- Complex type support via type constructors for rows (records), collections (sets, bags, lists, and arrays), and pointer types;
- Special operations, methods, can be defined for, and applied to, values of user-defined types;
- Unique OIDs identify each object and its data values.
- User defined query optimization rules gives cost information about user-defined functions.
- User defined index structures provide a generic template index structure, e.g. Generalized Search Trees [30].

The major advantages of extending the relational model is come from reuse and sharing. To enable object-relational features these extended features are implemented SQL:1999 [14]. Further, ORDBMS is the appropriate choice of applications that process complex data and have complex querying requirements.

2.1.1. Entity-Relationship Data Model

The *Entity-Relationship (ER) model* is a data model for abstract representation of database schemas. During the database design process, initially the database schema is represented in the ER model and then converted to the data model of the DBMS, e.g. the relational model. An *ER diagram* is the graphical representation of an ER schema with boxes and arrows representing the data elements and their relationships. It represents:

- *Entity*: represents real-world data objects.
- *Entity Type*: represents a group of objects with the same properties.
- *Attribute*: denotes the property of an entity type.
- *Relationship*: is a meaningful association between entity types.

In an ER diagram, rectangles represent the entity types, ovals interpret the attributes, and diamonds denote the relationships. The lines interconnect the respective attributes of an entity type and the other entity types involved in a relationship.

Based on the number of entity types participating in a relationship, it can be characterized as *binary, n-ary* etc. For example, when two entity types participate in a relationship it is called a *binary relationship* and if *n* entities participate it is an *n-ary relationship*. *Cardinality constraint* specifies the number of entity occurrence take part in a relationship. In a binary relationship there can be the following common cardinality constraints:

- One-to-one: Each occurrence of one entity is associated with one of the other entity occurrence as in *Figure 1*.

Figure 1. one-to-one relationship

- One-to-many: Each occurrence of one entity is associated with many of the other entity occurrences as in *Figure 2*:

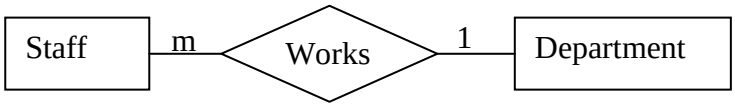


Figure 2. one-to-many relationship

- Many-to-many: Each occurrence of first entity is associated with multiple occurrences of the second entity and every occurrence of second entity has association with many of the occurrence of the first entity as in *Figure 3*.

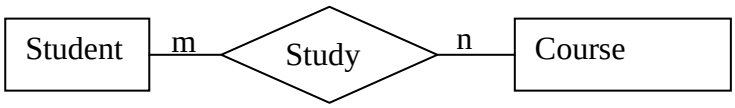


Figure 3. many-to-many relationship

2.1.2. Query processing

Query processing involves the essential activities to retrieve required data from a database. The *query processor* (*Figure 4*) is the group of components of a DBMS responsible for query processing.

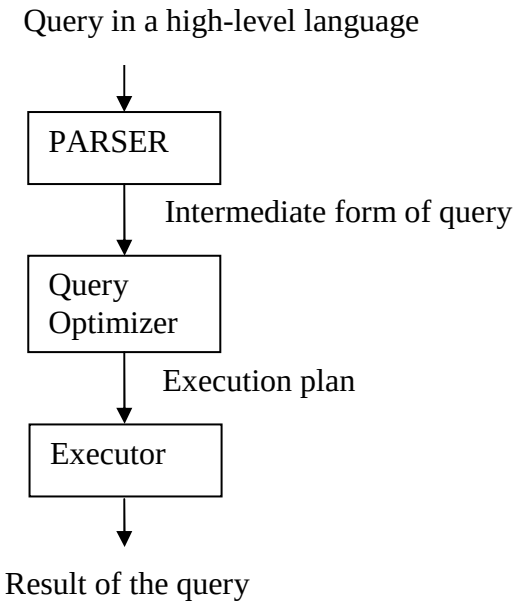


Figure 4. Query processing steps.

The *parser* ensures the query syntax follows the allowed grammar of the query language. The parser transforms the query into an internal intermediate form. The *query optimizer* translated the parsed query into an execution plan, which is a program to retrieve data. The query execution plan is functional program with DBMS-specific evaluation primitives such as scan operators, selection operators, various index scan operators, several join algorithms, sort operators, and a duplicate elimination operator. A query typically has many feasible execution plans, and the choosing the efficient plan is named *query optimization*, and is performed by the query optimizer. The traditional query optimization based on cost-based optimization [24]. It considers all likely execution plans and estimates the cost of each of the plans based on the number of disk blocks read, central processing unit (CPU) usage, and communication cost. Meta-data provides cost metrics.

Based on this the cheapest execution plan is chosen. Typically heuristics are applied to transform the execution plan to reduce the cost. The *executor* interprets the execution plan to produce the query result.

2.2. Information Integration

Producing, storing and transporting information in large scale are no longer momentous problems in the world. One significant issue in the Information era is the information integration: find any particular piece of information and combine this information with the existing information. Modern database systems are evolving in the direction towards information integration to emphasize an approach for data collection from multiple heterogeneous data sources. This is a key application in the daily operation of business, government, and academic organizations. The principal approaches for data integration are [21]: federation, warehousing, and mediation. There are subtle issues during the information integration:

1. Format differences: It covers the differences in data type, domain, precision, and item combination. For example, a part number is represented as an integer in one data source and represented as a string in another.
2. Value differences: The concept could be represented in different ways. E.g. one source could represent the value of state as 'Georgia' while other will represent as 'GA'.
3. Semantic differences: The same term could be interpreted differently in diverse sources. A university database keeps master degree students under the undergraduate section while another university database maintains it with the postgraduate portion.
4. Missing values: Some data sources may not keep some information that other sources provide. For example, a database administrator of a university keeps the initials of students' names while the students office database does not keep them.

The above inconsistency issues are solved in number of ways by different information integration systems.

2.2.1. Federated databases

A *federated database* resembles a class of heterogeneous databases. A common phenomenon is that the information sources are independent, but one source is able to communicate with the others to retrieve information. A wide range of solutions are proposed in the literature with different terms

such as distributed databases, federated databases, multi-databases, and interpretable systems[31].

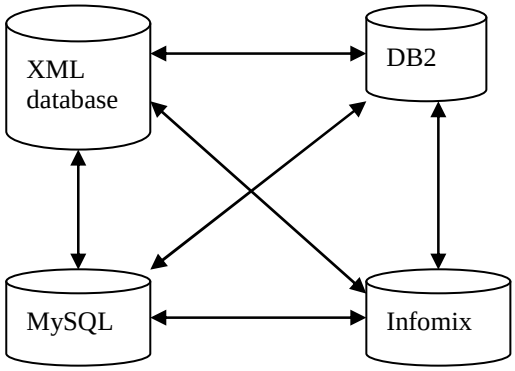


Figure 5: Federated databases

Figure 5 illustrates four different databases in a federation where 12 pieces of code is needed to translate the query from one another. In general, *n* databases in federation need *n (n-1)* translations to support queries between each other.

Each database involved in a federation maintains a local *import* and *export* schema. The export schema describes the information of the local database shared with the other databases in federation, while import schema is a description of the information can be retrieved from the other databases.

2.2.2. Data warehouses

When the data from data sources of diverse locations are stored in a single central database it is known as *data warehouse (Figure 6).* It requires a *global schema*. Further, data from the heterogeneous data sources are pre-processed, e.g. by filtering and aggregating, prior to storing the processed data in the data warehouse. Users query directly the warehouse instead of particular data sources.

For consistency, direct user updates to a data warehouse should be avoided. Generally in a data warehouse, data is constructed in three different ways:

- *Reconstruction*: Data warehouses are periodically reconstructed from the currently available source data. During the reconstruction the system is closed for queries. The major drawbacks are the time consuming reconstruction process and that data for applications that require data from the warehouse is unavailable.

- *Periodical update*: The data warehouse is periodically updated based on the changes that have been made to the sources since last modifications to the warehouse. This kind of update reduces amount of the time and data. But the process for calculating the changes, *incremental updates*, is very complex.
- *Immediate update:* Each change or small set of changes occurred in the sources are immediately reflected in the warehouse. As this approach incurs much communication, it is best suited for a warehouse that contains data sources changing slowly.

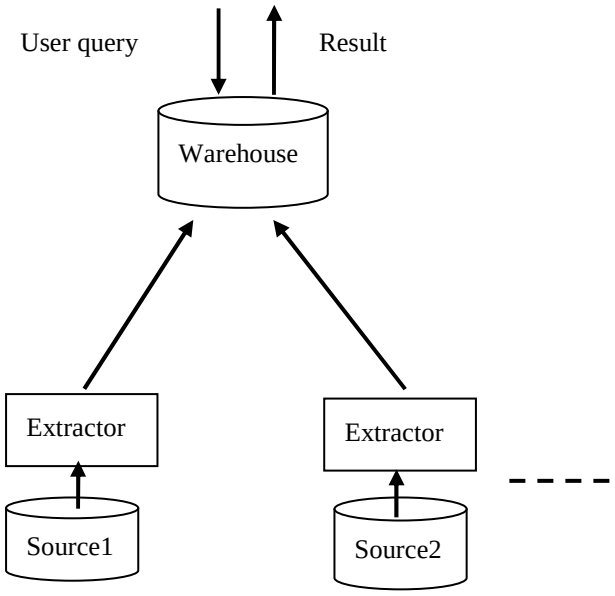


Figure 6 :Data ware house architecture

2.2.3. Mediators

Mediators are software modules used to query heterogeneous wrapped data sources and applications. In [57] a *mediator* is defined as:

> A mediator is a software module that exploits encoded knowledge about some sets or subsets of data to create information for a higher layer of applications.

A mediator represents a virtual view or composition of views that integrate several heterogeneous data sources. Mediators don't store any data themselves and this contrasts mediation from the data warehouse approach.

Instead as shown in *Figure 7*, it makes use of *wrappers* to retrieve data from heterogeneous data sources.

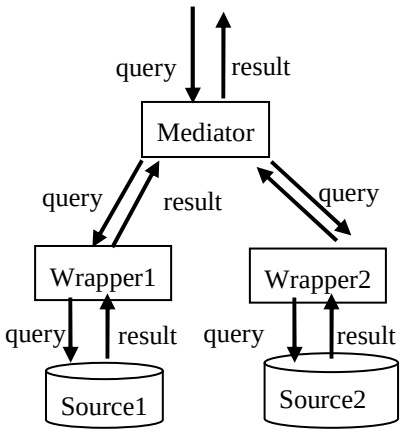


Figure 7. Mediation architecture

A wrapper is a software module that facilitates query processing and translation of data from a particular external data source. When a query is given to the mediator, it could then construct the appropriate sub queries and send them to the wrappers. A wrapper accepts queries from a mediator and translates them so they can be answered by the underlying data source. Then it returns back the result to the mediator and in turn the mediator collects data from several wrapped data sources and post-processes them before sending back the result of the user query. Mediators deploy a common data model (CDM) to map schemas of heterogeneous sources. Mediation addresses data integration in a more dynamic way than federation by using extraction, transformation, and integration processes, while a federation represents a static approach by utilizing agreed couplings to allow view creation.

There are several systems such as Garlic [55], Information manifold [38], and TSIMMIS [20] using mediators for data integration from heterogeneous data sources.

2.3. XML

XML [9] has evolved as a de facto standard for representing structured data and semi-structured data that have a structure changing rapidly Simplicity,

open standard, platform or vendor independence, extensibility, and reusability are some important aspects of XML.

An XML document consists of tagged data structures. An *element* is a technical name for the pairing of a start tag and end tag in an XML document. Each element contains zero or more attributes. An *attribute* is specified by name-value pair.

Grammatical constraints on the structure of XML documents are imposed by a *Document Type Definition (DTD)* [9]. A valid XML document must contain a valid *Document Type Declaration* that conforms to the DTD. A document's type declaration can be declared inline in an XML document, or as an external reference. With a document type declaration, independent groups of XML documents can agree to use a common document type declaration for exchanging data. Further an application can use a standard document type declaration to verify that data that is received from the outside world is valid and can also use a document type declaration to verify its own data. The DTD is used to verify the *wellformedness* and *validity* of an XML document. Wellformedness ensures the XML document is syntax error free while validity makes sure the elements and attributes in the XML document conforms to a predefined grammar.

XML schema [19] provides a much more powerful means by which to define the XML document structure and limitations. XML Schemas are themselves XML documents. A schema can be associated with an XML document by specifying the schema location via a *namespace*. An XML namespace is composed of a URI and a local name. The XML schema definition itself has its own DTD. XML schema provides a set of basic data types [19], called *Simple Types*. The users can define their own simple types by adding constraints to the basic data types. Another kind of user defined data types known as *Complex Types* which allow user defined data structure definitions containing elements and attributes. Simple types cannot have elements or attributes. These types are much wider ranging than the basic PCDATA and CDATA of DTDs. Further it specifies constraints on the attributes, supports some sophisticated structures [54] such as definition derived by extending or restricting other definitions, and a name space mechanism allowing the combination of different schemas.

Two types of XML documents emerge from applications: *data-centric* (Figure 8) and *document-centric* (Figure 9). Data-centric XML is characterized by a regular structure. It occurs in the context of structured data exchange and representation of semi structured data. Document centric XML has a much more irregular structure, is often characterized by the ubiquitous nature of mixed mark-up in it, and is often encountered as the means of encoding information about documents. There are XML documents that follow both data-centric and document-centric structures, namely hybrid XML documents.

```
< Notice >
  <Location>room 1345</Location>
  <MeetingTime>15:15 PM</MeetingTime>
  <Purpose>discuss future directions</Purpose>
</ Notice
```

Figure 8. Data-centric

```
<Notice>
Lab meeting will be held at the
<Location>room 1345</Location>by
<MeetingTime>15:15 PM</MeetingTime>
to<Purpose>discuss the future
directions</Purpose>
</Notice>
```

Figure 9. Document-centric:

2.3.1. XML databases

Various approaches [16] are followed to organize XML documents to facilitate querying and data retrieval. With the first approach [50], an RDBMS or ORDBMS can be used to store whole XML documents as text fields within a DBMS. A special document processing component is deployed to handle the XML documents and the approach is well suited for schema-less and document-centric XML documents. The second approach [36, 60] is utilizing an existing RDBMS to translate into a relational schema XML documents that follow a specific XML DTD or XML schema. A mapping algorithm manages to derive a database schema compatible with the XML DTD or schema. The third approach [43] creates a new type of DBMS for storing XML documents, which includes specialized querying and indexing facilities, and compression mechanisms to reduce the size of the documents.

2.3.2. XML querying

Several systems have been built to query XML in general, e.g. [17, 22, 23, 28, 42]. The Lore system [23] has its own XML based data model and a query language Lorel to allow navigation of both attributes and sub-elements. XPath [12] is a declarative query language for XML and collections of elements can be retrieved by defining a directory-like path along some conditions placed on the path. XPath considers an XML document as a tree with nodes for each element, attribute, text and namespace. Further, the XML Query Working Group [71] introduced a data

model for XML containing query operators such as projection, selection, iteration, join, sorting, aggregation, and a XML query language known as XQuery [7]. XQuery is a functional language in which a query is represented as expressions: path expressions and FLWR expressions. The path expressions make use of abbreviated XPath syntax, extended with a *dereference* operator and a *range* predicate. A set of XML documents is accessed like a database. The FLWR (*Figure 10*) expression is constructed from *FOR, LET, WHERE* and *RETURN* clauses. The FOR clause is used whenever an iteration is needed with a specified variable while LET clause is used to binds variables to paths before the iteration is performed. The WHERE clause defines the conditions and the RETURN clause generates the output of the FLWR expression.

```
FOR $B IN DISTINCT(document(“staff.xml”)//@branchNo)
LET $S:=document(“staff.xml”)/STAFF/[@branchNo=$B]
WHERE count($S)>20
RETURN $B
```

Figure 10. FLWR expression

The SQL 2003 standard [15, 64] facilitates to combine SQL with XQuery to access both ordinary SQL-data and XML documents stored in a relational database.

2.4. Web Services

A web service is defined by W3C [8] as:

> A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-process able 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.

Web services provide a message exchanging framework for applications by defining a set of *operations* that can be invoked over the communication network. Each web service operation defines a specific action performed. Web services incorporate standards such as SOAP [25], WSDL [11], XML Schema [19], HTTP [61] and UDDI [5]. A web service is described using the WSDL language. A WSDL description uses XML-Schema to describe data types of the arguments and results of operations. WSDL descriptions are published in a UDDI directory, which is a central place that holds set of web service descriptions. Any one can find required web service descriptions by querying the UDDI directory. A SOAP message is used to

invoke a web service operation call by packing all the necessary details in a standard format. HTTP may be deployed to transfer the SOAP message to invoke a web service and return the result back.

The web service architecture can be illustrated with layered technologies as shown in *Figure 11*. The *discovery* layer acts as a centralized repository of web services and by querying this repository one could find the required web service. The open standard technologies UDDI [5] and WS-Inspection [3] can be deployed at this layer for how to publish, categorize, and search for services based on the their service descriptions. The *descriptions* layer deals with how to represent service behavior, capabilities, and requirements in machine readable form.

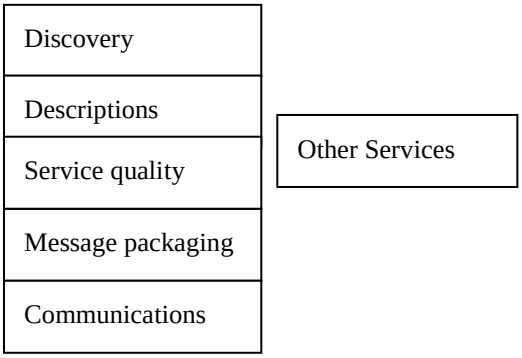


Figure 11. Web service architecture

WSDL [11] is used to define the functional capabilities of a service in terms of operations, service interfaces, and message types. Also it supplements deployment information such as network addresses, transport protocols, and encoding formats of the message transmission.

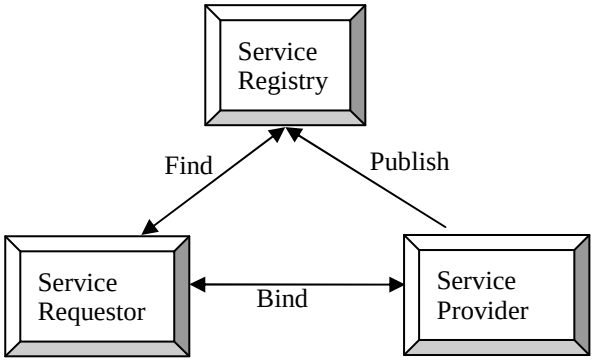


Figure 12: Service-oriented architecture

The *communications* layer carries the data over the network for the application. Data is converted into an internal format by the *message packaging* layer. SOAP provides a standard way for such message packaging. Then the packed message will be transported by the communications layer using internet technologies including HTTP, SMTP [46] and FTP [47].

The *service quality* layer addresses protocols that ensure the quality of the service such as security, reliable messaging, transactions, management etc. The WS-policy framework [2] declare the service quality requirements and capabilities that enables service quality policies of web services to be attached to the different parts of a WSDL definition. Security policies for authentication, data integrity, and data confidentiality are standardized by OASIS as WS-Security policy [37]. The web service management task force [70] is tailoring the standards for web service management that involves with monitoring, controlling, and reporting of service qualities and usage.

Other service layers represent the protocols used for some different purposes such as composing services to create new applications. For example, BPEL4WS [1] provides a workflow oriented composition model well suited for business applications.

Figure 12 illustrates the interrelationship of SOAP, WSDL and UDDI in a service oriented environment. The *service provider* is responsible for creating a service description using WSDL, and publishes service in a *service registry, UDDI*. The UDDI advertises the service and allows *service requestor* queries to the registry to find a service either by name, category, identifier, or supported specification. Once the service is found, the service requestor receives the information about the location of its WSDL document. Then the service requestor creates a SOAP message in accordance with service descriptions of WSDL document and sends it over the network to the service provider to apply the service. The *bind* operation embodies the relationship between the service requestor and the service provider.

2.5. Web Services Description Language

The functional description of a web service is defined by web services description language (WSDL) [11] that conforms to the XML grammar. A WSDL document defines *services* as set of network endpoints, or *ports***.** In WSDL, the abstract definition of endpoints and messages is separated from their concrete network deployment or data format bindings. This allows the reuse of abstract definitions: *messages*, which are abstract descriptions of the

data being exchanged, and *port types* which are abstract collections of *operations*. An operation defines the description of an action supported by the service. The concrete protocol such as SOAP, HTTP, and data type specifications for a particular port type represents a reusable *binding*. A port is defined by associating a network address with a binding. Different type definitions other than XMLSchema can be used to describe all message formats present and future, WSDL allows using other type definitions via extensibility, known as *extensibility elements*. Through this structure WSDL describes:

1. *What a service does*: The operations provided by the service and the data needed to invoke them.
2. *How a service is accessed*: Details of the data formats and protocols necessary to access the service's operations.
3. *Where a service is located*: Details of the protocol-specific network address, such as a URL.

A WSDL document can be described as a set of *definitions*. A grammar that contains a definition element at the root denotes the structure of a WSDL document as in Appendix A.

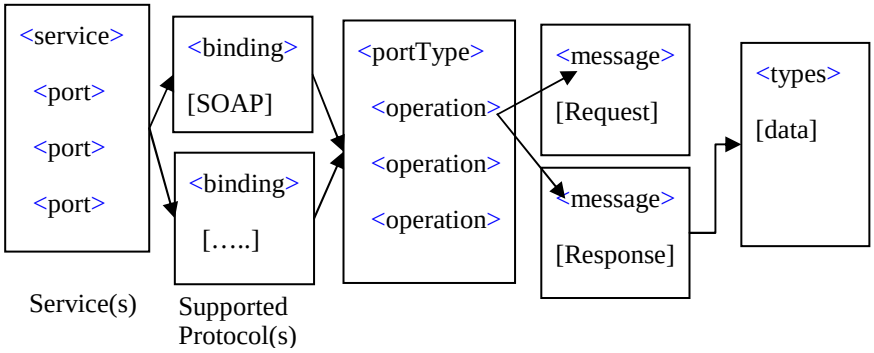


Figure 13. Document structure of WSDL

A simple document structure is illustrated by *Figure 13*. Each service has several ports to define where it is located. In turn each port is attached with one or more bindings that describe how a web service is accessed. Each binding is attached with a *portType* having a set of operations to answer what a service is does. Request and response messages are attached with each operation to indicate the input and output of an operation.

Definitions

A definition contains the elements *name*, *documentation*, *import*, *types*, *message*, *portType*, *binding*, and *service*. The element *definitions* contains

the attribute *name* (usually the name of the web service) for only documentation purpose. The attribute *targetNamespace* stores the namespace URI for the entire WSDL file. That attribute is used to form *QNames* (Qualified names) of *portTypes*, *bindings*, and so on, and how to combine WSDL descriptions that span multiple files. The usual XML namespace (*xmlns*) declarations are also part of *definitions*. The *import* element permits the separation of the different elements of a service definition into independent documents by associating a *namespace* with a document *location*.

Types

Types denote the data types for the exchanged messages and adopt the types supported XML-Schema.

Messages

A message consists one or more *parts*. Each part is associated with a type and element attribute. The *name* provides a unique name among all other parts and messages. Generally a message definition is considered as an abstract definition. A message binding describes the mapping between the abstract and concrete definitions.

Parts

A part supports a mechanism for depicting the abstract content of a message. A binding is specified with reference to the name of a part for binding-specific information. Multiple part elements are defined with messages to specify multiple logical units.

Port types

A *port type* defines a set of abstract operations and the abstract messages. The *name* attribute provides a unique name. A port type is defined as:

```
<wsdl:definitions .... >
   <wsdl:portType name="nt">
      <wsdl:operation name="nt" .... /> *[1]
   </wsdl:portType>
</wsdl:definitions>
```

Operations

An operation defines a method on a web service, including the name of the method and input parameters and the output parameters of the method. All the operation names within a single port type are different.

1. One-way: The endpoint only receives the message.

```
<wsdl:definitions .... >
```

[1] ***** - zero or more

```
    <wsdl:portType .... > *
       <wsdl:operation name="nt">
         <wsdl:input name="nt"? message="qname"/>
       </wsdl:operation>
    </wsdl:portType >
</wsdl:definitions>
```

2. Request-response: The endpoint receives a message and sends a response.

```
<wsdl:definitions .... >
 <wsdl:portType .... > *
  <wsdl:operation name="nt" parameterOrder="nts[2]">
   <wsdl:input name="nt"? message="qname"/>
   <wsdl:output name="nt"? message="qname"/>
   <wsdl:fault name="nt" message="qname"/>*
  </wsdl:operation>
 </wsdl:portType >
</wsdl:definitions>
```

3. Solicit-response: The endpoint sends a message, and receives a response.

```
<wsdl:definitions .... >
 <wsdl:portType .... > *
  <wsdl:operation name="nt"
         parameterOrder="nts">
    <wsdl:output name="nt"? message="qname"/>
    <wsdl:input name="nt"? message="qname"/>
    <wsdl:fault name="nt" message="qname"/>*
  </wsdl:operation>
 </wsdl:portType >
</wsdl:definitions>
```

4. Notification: The endpoint only receives a message.

```
<wsdl:definitions .... >
 <wsdl:portType .... > *
   <wsdl:operation name="nt">
     <wsdl:output name="nt"? message="qname"/>
   </wsdl:operation>
 </wsdl:portType >
</wsdl:definitions>
```

Bindings

Message formats and protocol information for each operation and message defined under a port type is defined by a *binding*. A given port type may have any number of bindings. The attribute *name* provides a unique name for a binding. A binding have to specify exactly one protocol and must not

[2] nts- nmtokens [9]

specify any address information. The referenced port type is depicted by attribute *type*. A concrete grammar for input, output and fault messages is specified by the binding elements. The following exemplifies a binding conforming to the above:

```
<wsdl:binding name="TerraServiceSoap"
type="tns:TerraServiceSoap">
 <soap:binding
 transport="http://schemas.xmlsoap.org/soap/http"
 style="document" />
 <wsdl:operation name="ConvertLonLatPtToNearestPlace">
   <soap:operation soapAction="http://terraservice-
   usa.com/ConvertLonLatPtToNearestPlace"
   style="document" />
    <wsdl:input>
       <soap:body use="literal" />
    </wsdl:input>
    <wsdl:output>
       <soap:body use="literal" />
    </wsdl:output>
 </wsdl:operation>
</wsdl:binding>
```

Further a binding element for a given port type can denote a transport protocol such as SOAP over HTTP, SOAP over SMTP, HTTP POST operation, etc.

Binding extensions

WSDL supports three extensibility conventions that allow the binding to be extended with elements from different XML namespaces to describe bindings to any number of transport protocols such as SOAP, HTTP.

SOAP Binding extension

Extension elements are used to bind a WSDL description to the SOAP protocol. The *Soap:binding* is an obligatory element when using the SOAP binding. The *style* attributes shows subsequent operations following one the two alternatives: *document* or *rpc (Remote Procedure Call)*. The option *rpc* declares that the messages have parameters and return values while the option *document* indicates that the messages contain documents. Further, the *document* option specifies how the body of the SOAP message will be interpreted in straight XML, while the *rpc* option indicates that the binding uses RPC conventions for SOAP body specifications. The style for the binding can be overridden by the style attributes in the child operation elements. The *soapAction* defines the name of the action (method) to be invoked by the service. It is placed in the *SOAPAction* HTTP header as the part of an HTTP message. The *soap:body* declares the structure of the contents of the message. The attribute *parts* specifies which parts will be

used during the SOAP message creation process. The *soap:fault* element shows the contents of the SOAP faults details.

Ports

For each binding the attribute *port* defines a single address endpoint. An extensibility element specifies the address information of a port and more than one address can't be specified for a port.

Services

A service groups is a set of related ports. An example that defines service element is:

```
<wsdl:service name="TerraService">
 <documentation
 xmlns="http://schemas.xmlsoap.org/wsdl/">TerraServer
 Web Service</documentation>
  <wsdl:port name="TerraServiceSoap"
  binding="tns:TerraServiceSoap">
     <soap:address location="http://terraservice.net
     /TerraService2.asmx" />
  </wsdl:port>
</wsdl:service>
```

The *name* attribute specifies a unique service name.

The structure of WSDL described above is based on WSDL 1.1 of W3C recommendation. Most of the existing WSDL documents are based this version. W3C now concentrates on WSDL 2.0, and it is defined with three specifications:

- Part I, the core Language
- Part II, Message Patterns
- Part III, Bindings

Some new features are added, some removed and some of them are modified for unambiguity, better naming, and simplifications.

2.6. SOAP

SOAP is an XML based lightweight, platform independent protocol for information exchange in a distributed environment. SOAP is not only used with HTTP but also potentially used in combination with other protocols such as SMTP, TCP [63]. The simplicity and extensibility are the major design goals of SOAP.

Structure of a SOAP message

A SOAP message (*Figure 14*) is made up of three elements:

- A *SOAP Envelope* is a top element that encapsulates the other two elements representing the message.
- An optional *SOAP header* provides a generic mechanism for adding additional features to the message such as routing and delivery setting, authentication assertions, and transaction contexts.
- A *SOAP body* contains the actual message to be delivered and processed.

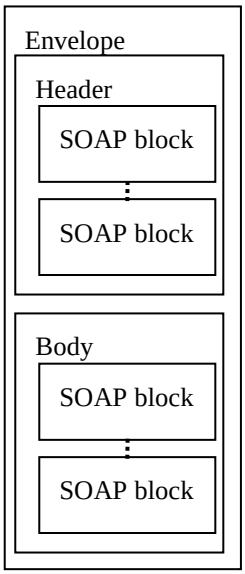


Figure 14. SOAP Message

In addition to the above components a *fault* block could appear with in the body whenever there is an error to be reported to the sender of the SOAP message. The SOAP block denotes a single computational unit of data by the processor of a message.

Example of a SOAP message:

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle=
              "http://schemas.xmlsoap.org/soap/encoding/">
   <SOAP-ENV:Body xmlns="http://terraservice-usa.com/">
    <GetPlaceList>
         <placeName>Atlanta</placeName>
```

```
        <MaxItems>100</MaxItems>
        <imagePresence>true</imagePresence>
     </GetPlaceList>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The current SOAP specification is version 1.2 [25] released by W3C in June 2003. SOAP message transmission is basically one-way from a sender to a receiver. To exploit the unique characteristics of the network protocols used for a transmission, SOAP implementations can be optimized. Generally messages are routed along a *message path* which contains one or more intermediate nodes in addition to the eventual destination. Further, the *actor* attribute is used to indicate the intended participants of various parts of a SOAP message and used to direct a SOAP message through a sequence of intermediaries, with each one processing its portion of the message and forwarding the remainder.

In addition to pure messaging semantics, SOAP defines a mechanism for RPC by placing some constraints such as how the root element of SOAP body is to be named and how the data could be encoded.

2.7 Semantic Web

The semantic web [62] is an emerging framework that aims at machine-processible information for information sharing. It defines standards not only for syntactic form of documents, but also for the semantic contents. Further, it enables intelligent services such as information brokers, search agents and information filters to offer more functionality and interoperability than current technologies. The prominent W3C standardization efforts are XML/XML schema and RDF [35]/RDFSchema [10] to facilitate semantic interoperability.An *ontology* defines a hierarchy of concepts within a domain and describes each concept's crucial properties through an attribute-value means Ontologies play a vital role in the semantic web for processing, sharing and reusing metadata between applications. The OWL [26] layers *OWL Lite*, *OWL DL* and *OWL Full* along with RDF are the commonly used as ontology languages in the semantic web.

RDF facilitates a common framework for expressing information about a web source that needs to be processed by applications so the information can be exchanged between applications without loss of meaning. What sets RDF apart from XML is that RDF is designed to represent knowledge in a distributed world while XML-Schema is purely syntactic/structural to encode an application-specific data interface. RDF provides a method to decompose any knowledge into small segments, called *triples* also known as

statements, with some rules about the semantics (meaning) of the statements. Each triple contains *subject*, *object* and *predicate*. The *subject* represents the entity described by the piece of knowledge. The *predicate* is an identifier for some property of the subject. The *object* denotes the value of the property. Consider the following knowledge represented by a sentence "*http://www.it.uu.se/edu/course/kurs start/spring.html web page maintained by Department of information technology*". An equivalent RDF representation can be stated with the subject, *http://www.it.uu.se/edu/course kursstart/spring.html* and the predicate, *maintained* with the object *Department of information technology*. Further, RDF permits the object to be lists, bags, and sequences.

RDF can be used in resource discovery for enhancing search engine capabilities, and cataloguing for content description of web sources and interrelationship of contents. W3C [35] standardize the concepts and syntax of RDF to achieve:

- Simple data model: easy to handle by the applications.
- Formal semantics and inference:
- An extensible URI-based vocabulary
- An XML-based syntax and support of XML schema data types

The knowledge is represented by the RDF statements as a labeled, directed graph, the *RDF Graph* (*Figure 15*). An RDF graph represents a set of RDF triples. The subject is what's at the start node of the edge, the predicate is the type of edge (its label), and the object is what's at the end node of the edge.

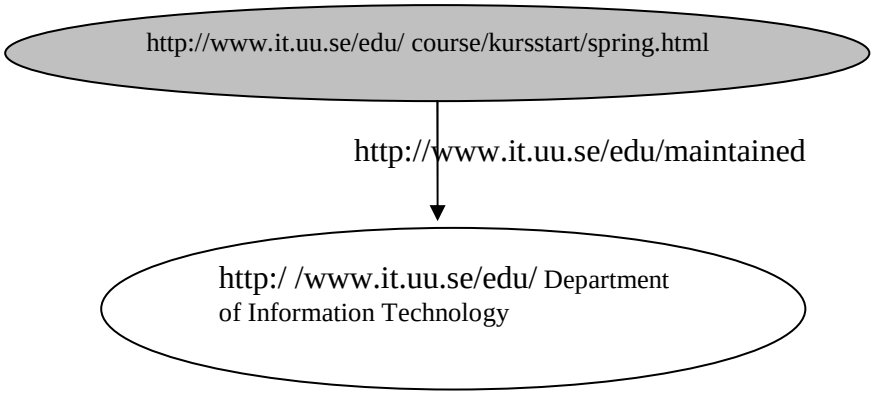


Figure 15. An RDF graph

In practice, name of predicate and object prefixed with a URI for the global identification. *Blank nodes* represent objects where the name of the object is unknown. RDF specifies how the triples are used to represent

knowledge, i.e. and abstract model. The triples are normally encoded in XML. Most of the abstract model of RDF comes down to these simple rules:

- A fact is expressed as a Subject-Predicate-Object triple.
- Subjects, predicates, and objects identify entities, whether concrete or abstract, in the real world.
- Names are URIs, which are global in scope, always referring to the same entity in any RDF document in which they appear.
- Objects can also be given as text values, called *literal* values, which may or may not be typed using XML Schema data types.

The W3C specifications define an XML format to encode RDF triples, RDF/XML [4].

RDF Schema [10] defines the vocabulary used in RDF models such as specifying which attributes apply to which kinds of objects and what values they can take, and describing the interrelationships between objects. Users that specify RDF documents are free to define their own terminology in RDF Schema. An elementary block in RDF schema is a *class* to group resources where each member shares some same properties. The other important elementary statement is the inheritance relation between classes: *subClassOf*. While XML Schema constraints the structure of XML documents based on syntax, an RDF Schema defines the vocabulary for the semantics used in RDF documents.

RDQL [51] and SPARQL [45] are two standard query languages for semantic web data. An RDQL query consists of a graph pattern, expressed as a list of triples and each triple pattern is comprised of named variables and RDF values (URIs and literals). An RDQL query (*Figure 16*) can support a set of constraints on the values of those variables and set of variables those produce answer set.

```
select ?a
where (?a, <http://www.type.com/syntax-ns#type>,<http://fin
d.com/someType>)
```

Figure 16. RDQL query

SPARQL has all the features of RDQL and more:

- ability to add optional information to query results
- disjunction of graph patterns
- more expression testing E.g. date-time support
- named graphs
- sorting

3 Querying data sources with Mediators

Performance and scalability over the amounts of data retrieved are important design aspects of mediators. Further, a mediator module is able to handle semantic integration of underlying sources. The system interpreting these mediator modules are known as the *mediator engine*. This chapter addresses schema representations in mediators, and query processing techniques used to handle limited capability sources. The WSMED using for web service mediation is developed by extending the AMOS II mediator engine [48, 49]. Further this chapter overviews the AMOS II mediator system including its data model and other functionalities.

3.1. Schema representations in mediators

Many systems [20, 38, 55] have been developed using the *central mediator* approach where the mediator is a central component with many wrappers. Other kinds of mediators are called *composable mediators* where mediators may wrap other mediators [34]. A mediator can have a *global* schema that includes all data schemas from the external sources. The global schema definition is difficult, in particular when there are many heterogeneous sources.

Mediators are providing a common data model CDM to represent the data integrated from different sources. The mediator engine interprets queries in terms of the CDM. Views play the prominent role in the meditation and defined by means of the CDM. Since the diverse sources represent the same information differently from the mediator schema, a mediator must include view definitions describing how to map the source schema into the mediator's schema. The most common methods in practice are:

- Global as view [20, 27]: With this strategy, the mediator schema is defined in terms of a number of views that map wrapped sources. This global mediator view is defined by matching and transforming data from the source schemas. Whenever new sources are inserted the view definitions need to be extended accordingly.
- Local as view [38]: It contains a fixed mediator schema. Whenever a new source is inserted, the view definitions have to define how to map data from the mediator schema to the new source’s schema without any further alternation in the mediator schema. This

approach simplifies the insertion of new sources. However, there are some issues of how to resolve differences when there are conflicts and overlaps between sources. Usually a default reconciliation based on accessing best source that covers the best data is needed for a user query. That is, the local as view approach is ill suited for reconciliation of the differences and similarities between different sources' data.

3.2. Capability based optimization in mediators

Data retrieval through web sources is a common practice in the industry. Mediators can be defined to integrate such data sources. They allow certain attributes as inputs and produce outputs with certain attributes, but have to real query capabilities. We say that these sources have *limited capabilities.* In addition, there are some other reasons for limited source capability:

- *Legacy sources:* Data is kept in some outdated format and it is impossible to convert the data format into a modern DBMS. Legacy sources only allow certain queries with specified inputs.
- *Security:* To ensure the privacy of data, such as defense information, sources permit only limited queries.
- *Limitation by indexes:* Indexing the data is a common mechanism to speedup the queries and is widely adopted in DBMSs. User queries to the attributes that are not indexed are not supported by the data sources as those queries examine millions of tuples.

The traditional cost based optimization is inadequate for web sources as queries to sources with limited capabilities are not only based on cost metrics but also depends on what query capabilities the sources provide. The optimization strategy *capability-based optimization* [44, 58] is tailored to consider the feasible plans on the basis whether the plans can execute at all using the limited capabilities of a data source. Cost measures can be used to choose among the feasible plans. Source capabilities are represented and examined during the query optimization mainly in two ways:

- *Rule-based checking*: This approach is implemented in mediator systems such as *Garlic* [20], *Information Manifold* [38], and TSIMMIS [39] to match the source capabilities. Source capabilities are represented as capability records [38] or by some special description languages such as Relational Query Description Language (RQDL) [56]. Complex rules are applied to find the suitable sources. During the query optimization phase rewrite rules are applied for efficient query execution.
- *Binding patterns:* Source capabilities are represented by a set of *adornments* [21] known as *binding patterns*. Matching sources are selected by analyzing the binding patterns. Information systems

such as the web query optimization system [59] utilize binding patterns to represent the source capabilities.

Estimating cost metrics in the mediation environment is quite difficult as the data sources are independent from the mediator. For example, with data accessible via web services the data retrieval time can be very slow due to congestion on the communication network or that the server providing service is highly loaded by several requests for data. Long-term observation or continuous monitoring of services will help for accurate cost estimation [29].

3.2.1. Representation of Source capabilities with binding patterns

The capability specifications of a data source are described as a set of *adornments* [21]. One adornment is attached with each attribute of the data source. It is represented by an alphabet with specific meaning:

- *f (free)* - the value of the attribute need not to be specified
- *b(bound)* - the value of the attribute must be specified
- *c[L](choice from a list L)*- the value of the attribute must be specified from the values in the list L.
- *o[L](optional, from the list L)*- the value of the attribute is optional, and if a value is specified it could be chosen from the list L.

f, *b*, and *c[L]* are the common adornments used to address the capabilities of sources that can be accessible via web services. *o[L]* is common when accessing web forms.

3.3. Active Mediator Object System (Amos II)

We have developed the prototype WSMED based on the existing mediator engine Amos II [49]. Amos II is an extensible main-memory oriented system that mediates distributed data sources. An object-oriented query language, *AmosQL*, is the primary query language. The system can support several wrappers to make heterogeneous data sources query able. A wrapper perform [48] the following:

- *Schema importation:* Translate the sources' schema into a form compatible with Amos II CDM.
- *Query translation:* translate AmosQL queries into API calls, web service calls or query expressions executable by the sources.
- *Statistics computation:* estimate costs and selectivities for the calls to retrieve data from sources.

- *Proxy OID generation:* constructs proxy object identifiers to describe the data from sources.

3.3.1. Amos II data model

The primitive concepts *objects*, *types*, and *functions* represent the Amos II data model (*Figure 17*). It is used as the CDM for the mediation and it is an extension of the Daplex [49, 52] functional data model.

Objects: They model all the entities in the database. Amos II has *system objects* and *user-defined objects*. Objects are represented in two ways, as *literal* or *surrogates*. Surrogates represent the real world entities such as vehicles, persons, etc; and have associated OIDs. They can be explicitly created and deleted by the users and the OIDs are maintained by the system. Literal objects are self-described system-maintained objects and do not have any explicit OIDs. For example numbers and strings. There are also *collections* of other objects: *bags*, *vectors*, and *records*. A *bag* represents unordered sets with duplicates while *vectors* denote the order-preserved collections. Vectors are accessed by *v[i]* where *v* is a variable holding a vector, and *i* is the index of an element in a vector. Records are useful to manage data retrieved through web services as they often handle nested structures. Records access uses the notation *s[k]*, where *s* is a variable holding a record, and *k* is the name of an attribute in a record. Thus records are indexed by arbitrary keys while vectors are indexed by numbers only.

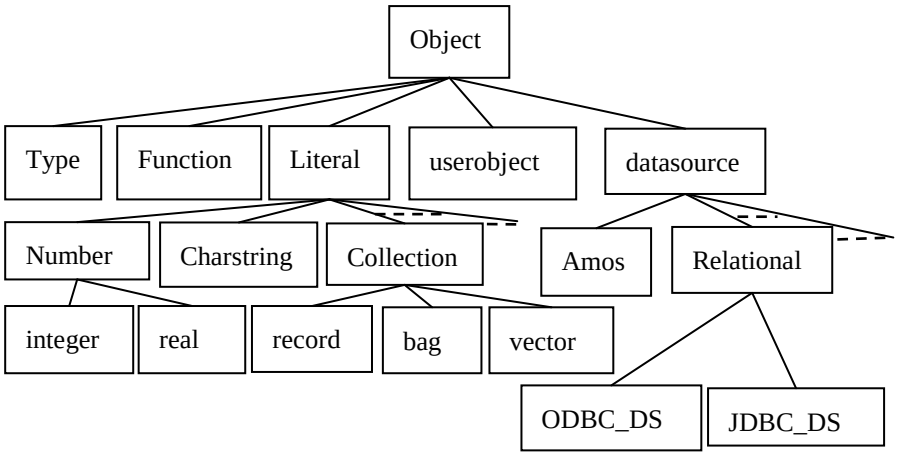


Figure 17. Amos II data model

Literals are automatically deleted by a garbage collector when they are no longer referenced.

Types: Objects are classified into *types* and each object is an *instance* of one or more types. The *extent* of a type represents the set of all instances of

the type. Types are ordered into a multiple inheritances type hierarchy. A type is defined and stored in the internal database of the system with system function *create type*. E.g.

```
create type Vehicle;
create type Truck under Vehicle;
```

Functions represent properties of objects, computations over objects, relationships between objects, and are used as primitives in queries and views. A function contains two parts: a *signature* and an *implementation*. The signature defines the types and names of the arguments and the result of a function. For example, the signature modeling the attribute *color* of the type *Vehicle* would have the signature:

```
colour(Vehicle) → Charstring
```

The *implementation* defines the mapping of a function to compute results for given arguments. Further, Amos II can inversely compute one or several arguments values of a function if the expected result value is known; this is known as the *multi-directional* feature of a function. The inverse usage of functions is crucial to specify general queries with function calls over the database. For example:

```
select vehichlenumber(v)
from   Vehicle v
where  colour(v)='blue';
```

Functions can be classified according to their implementations as:

- *Stored functions* are used to represent the properties of objects stored in an Amos II database.
- *Derived functions* are defined in terms of other Amos II functions as queries. They are side-effect free and they are precompiled and optimized as soon as they defined. The queries are expressed in AmosQL, using has an SQL-like *select* statement for defining derived functions.
- *Foreign functions* support low-level interfaces for wrapping external systems. They can update the external sources. However, foreign functions to be used in queries must be side-effect free. Further, it is possible to associate several implementations of inverses for a given foreign function, *multi-directional foreign functions*, which informs the query optimizer that there are several access paths implemented for the function. Users can help the query processor by associating *cost* and *selectivity* estimates for each access path implementation. Multi-directional foreign functions are defined using binding patterns. For example:

```
create function food(Charstring keyword,
                     Charstring groupcode)

                  →(Charstring ndb, Charstring descr)
as multidirectional
("ffff" foreign “JAVA: webservicewrapper/foodDescr”
  cost{100,1})
("fbff" foreign “JAVA: webservicewrapper/gp_foodDescr”
  cost{200,4})
("bfff" foreign “JAVA: webservicewrapper/kw_foodDescr”
  cost{150,3})
("bbff" foreign “JAVA: webservicewrapper/gp_kw_foodDescr”
  cost{400,6})
```

Here, the Java methods *foodDescr*, *gp_foodDescr*, *kw_foodDescr* and *gp_kw_foodDescr* are defined to retrieve some food data with different binding patterns. The *foodDescr* method will deliver data when none of the arguments of function food are known. The function *gp_foodDescr* retrieves data when the value for *groupcode* is known. Similarly *kw_foodDescr* returns values when *keyword* is known. In the case of both values for *keyword* and *groupcode* are specified, the *gp_kw_foodDescr* method will be used. The cost specifications estimate both execution costs in internal cost units and result sizes (fanouts) [40] for a given method invocation. In a web service mediation scenario, commonly many web service operations from diverse web services are involved. This common practice defines database views with multiple capabilities enabled with different binding patterns. Multi-directional foreign functions implement these kinds of views with various capabilities.

Data source: Diverse data sources are represented explicitly through the system type *Datasource* and its sub-types. Some of the sub-types embody generic kinds of data sources that share common properties. For example, the type *Relational* represents the common properties of all RDBMs. Other subtypes represent specific kinds of sources such as type *JDBC_DS* represents the JDBC drivers. Instances of these types represent individual data sources. Each data source type instance has a unique name and set of imported types.

3.4. Web service mediation

Our mediator engine for web services, WSMED (*Figure 18*), provides web service meditation by extending the Amos II mediator system. One common web service wrapper is deployed to wrap any web service. SQL user queries can be issued to WSMED. When an SQL query is received the required web service calls are passed to the web service wrapper to be invoked. The result of the web service call is normally a nested XML structure. It is post processed by the mediator to answer the user query. More detailed web service mediation with multi-level views is addressed in the chapter 5.

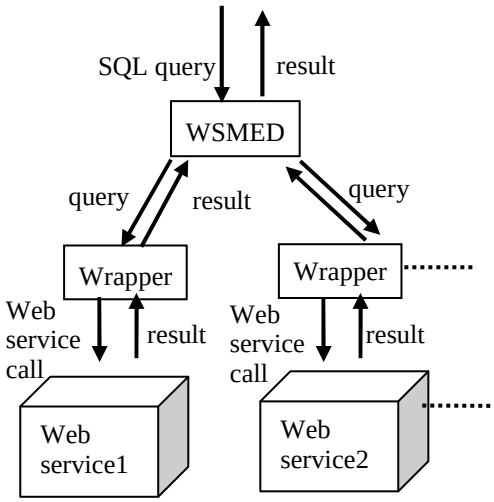


Figure 18. Mediation of web services

4. The WSMED system

This chapter gives an overview of the WSMED system. The web service schema subsection describes WSMED's internal representations of web service descriptions defined by WSDL documents and user provided semantic enrichments. The system components sub-section describes functionalities of WSMED system modules.

4.1. Web Service Schema

A WSDL description of a web service describes interfaces of its operations and the XML Schema data types are used. *Figure 19* shows an ER-diagram of WSMED's *web service schema* that represents WSDL descriptions. The *web service descriptions* store the WSDL core elements *service, operation,* and *element.*

A *service* describes a particular web service and supports a set of *operations,* the *Service* entity. Each web service has a *name*, and a namespace URI *nu* is a URI to identify the web service. The *ports* relationship represents the association between a service and its operations. Each *operation* named *na* represents a procedure that can be invoked through the web service. The *style, st*, indicates whether the operation is RPC-oriented or document-oriented. The *encoding style, es,* is a URI that indicates the encoding rules for data in the SOAP messages. The *target URL, tu,* determines the address of the SOAP message. The *SOAPactionURI, su,* identify the task of the SOAP Message.

Each operation has a number of *input* and *output* elements. An *element* is an abstract definition of the data being transmitted and is associated with a type definition using XML Schema. The input and output elements define the *signature* of the operation. Complex data elements may consist of other *sub-elements* where each sub-element has a data type, along with a name and the number of maximum occurrences within the super element. The WSMED uses a *conversion table* (Table 1) for type conversion from/to a XML Schema data type to/from the corresponding data type in WSMED.

The right part of *Figure 19* describes some semantic enrichments provided through WSMED in order to improve query execution efficiency. A *WSMED view* definition may reference several web service operations, as indicated by the *view_of* relationship. It is defined in terms of a number of

attributes. Each *attribute* has a *name*, a data *type*, and a flag (*is_key*) to indicate whether it is the primary key of the WSMED view. To simplify the schema, we here ignore representation of secondary keys.

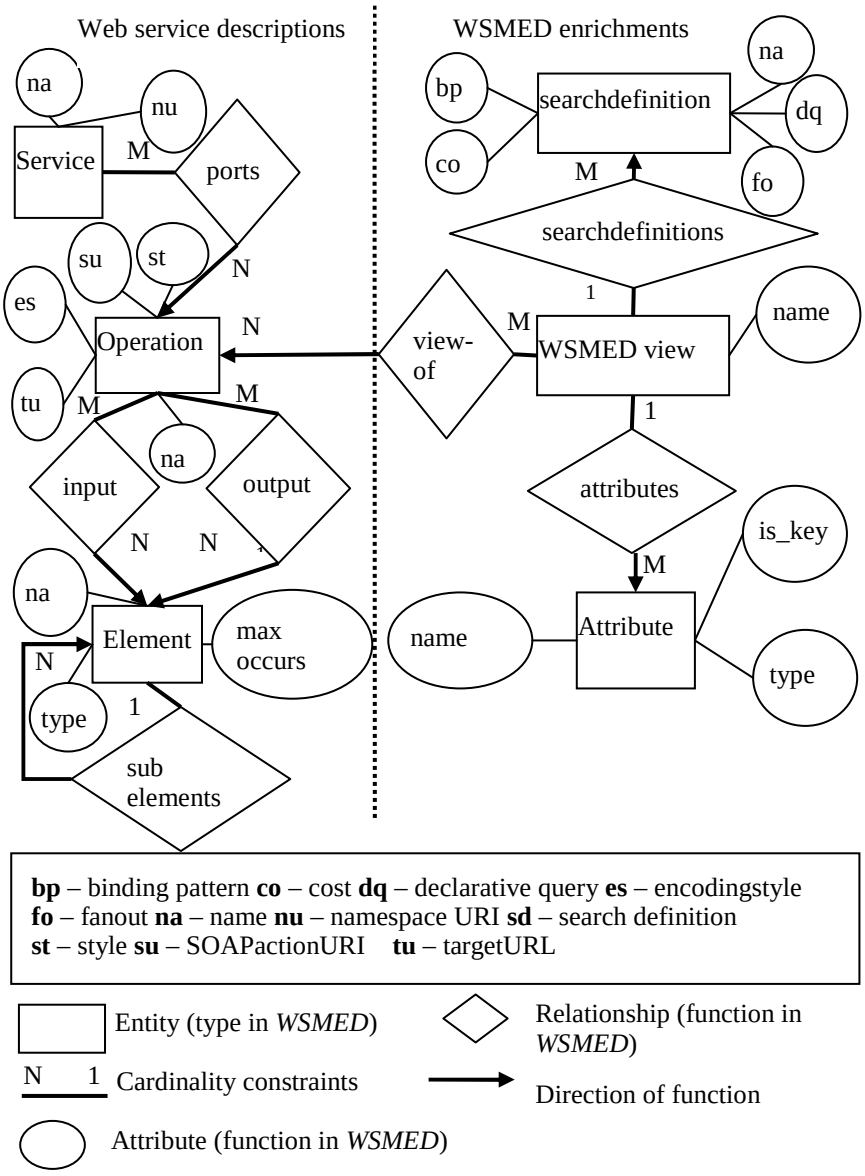


Figure 19. Web Service Schema

A WSMED view is defined in terms of a set of different search strategies, called *search definitions*. A search definition has an associated *binding pattern*, *bp*, indicating the attribute bindings when the search definition is applicable.

The *declarative query*, *dq*, of a search definition specifies for a particular binding pattern a query that computes the view in terms of the structure returned from web service operations. For query optimization each search definition also has some optional statistics, like the estimated cost, *co*, to execute the search definition and its estimated result size, *fo* (fanout). The user can explicitly specify *co* and *fo* by profiling the search definition[3]. In the search definition (*Figure 20*), *fbfb* is the binding pattern, *select foodDescr(fgc,fd)* is the declarative query of the search definition, and the numbers associated with the keyword *cost* represents *co* and *fo*, respectively. Chapter 5 explains more details of the view *foodDescr*..

```
("fbfb" select foodDescr(fgc,fd) cost {1000,100});
```

Figure 20. Search definition

In case the user cannot specify the costs, a *default cost model* is used to approximate the execution cost of web services. The default cost model is defined in Chapter 7.

Figure 21 shows how the web service schema is represented in WSMED using the object-oriented query language AmosQL [48, 49]. An entity is represented as a *type*, a relationship as a *function*, and an attribute as a *property* of a type.

```
create type Service
   properties (name Charstring,
               namespaceuri Charstring,
               wsdluri Charstring);

create type Operation
  properties(name Charstring,
             soapactionuri Charstring,
             style Charstring,
             encodingstyle Charstring,
             targeturl Charstring);

create function port(Service) -> Bag of Operation;

create type Element
    properties(name Charstring,
               mappedtype Charstring,
               maxoccurs Integer);
```

[3] Automatic computation of *co* and *fo* is future work.

```
create function input(Operation)
               -> vector of Element;

create function output(Operation)
               -> Vector of Element;

create function subelements(Element)
               ->Vector of Element;

create type WSMEDView
 properties (name Charstring);

create function view_of(WSMEDView)
               ->Bag of Operation;
 create type Attribute
 properties (name Charstring,
             type Charstring,
             is_key Boolean);

create function attributes(WSMEDView)
               ->Bag of Attribute;

create type Seachdefinition
 properties (name Charstring,
             co Integer,
             fanout Integer,
             dq Charstring,
             bindingpattern Charstring);

create function searchdefinitions(WSMEDView)
               ->Bag of Searchdefinition;
```

Figure 21. WSMED representation of the web service schema

WSDL data type	**WSMED data type**	**WSDL data type**	**WSMED data type**
anyURI	Charstring	Integer	Real
baseBinary	Charstring	Language	Charstring
Boolean	Boolean	Long	Integer
Byte	Integer	Name	Charstring
Date	Date	NCName	Charstring
dateTime	Charstring	Negative Integer	Real

Decimal	Real	NMTOKEN	charstring
Double	Real	NMTOKENS	charstring
Duration	Charstring	Nonnegative Integer	Real
ENTITIES	Charstring	nonPositive Integer	real
ENTITY	Charstring	Normalized String	charstring
Float	Real	NOTATION	charstring
gDay	Charstring	positiveInteger	real
gMonth	Charstring	QName	charstring
gMonthDay	Charstring	Short	integer
gYear	Charstring	String	charstring
gYearMonth	Charstring	Time	Time
hexBinary	Charstring	Token	charstring
ID	XS_ID	unsignedByte	integer
IDREF	XML	unsignedInt	integer
IDREFS	XML	unsignedLong	integer
Int	Integer	unsignedShort	Integer

Table 1. *Mappings between WSDL and WSMED data types*

An important semantic enrichment is information about the *key* of the data returned by a WSMED view, the attribute *is_key*. This enrichment is important to detect common sub-expressions in queries, as will be shown in the forthcoming chapters.

4.2. System Components

Figure 22 illustrates WSMED's system components. WSMED represents WSDL meta-data in the *web service meta-database* using the *web service schema* (*Figure 19*, left part).

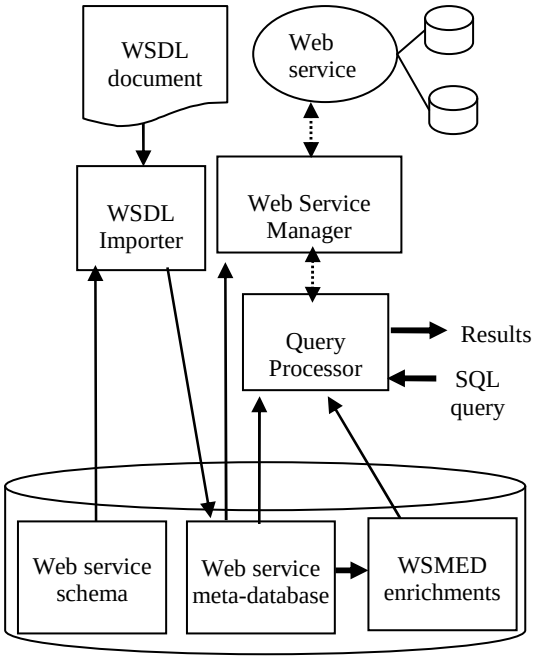


Figure 22. WSMED system components

The *WSDL Importer* can populate the web service descriptions by, given the URL of a WSDL document, reading the WSDL document using the Java tool kits *WSDL4J* [66] and *Castor* [68]. It parses the retrieved WSDL document, converts it to the format used by the web service schema, and stores the extracted meta-data in the web service meta-database. In addition to the web service descriptions, WSMED also keeps additional *WSMED enrichments* (*Figure 19*, right part) in its local store.

The *query processo*r exploits the web service descriptions and WSMED enrichments to process queries. It utilizes an existing mediator engine Amos II [48,49]. The query processor calls the *web service manager* component, which is implemented using the APIs *SAAJ* [65]. The web service manager is accountable for invoking web service calls using SOAP in order to retrieve the result for the user query.

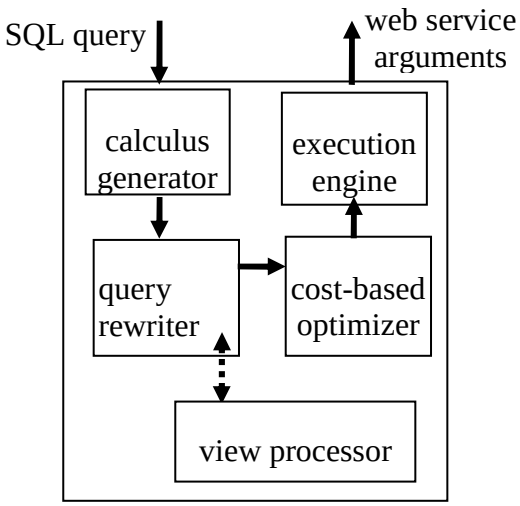
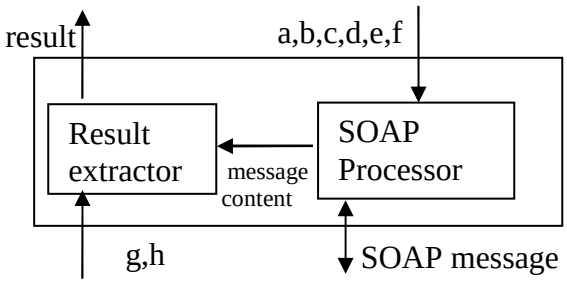


Figure 23. Query Processor

Figure 23 illustrates architectural details of the query processor. The *calculus generator* produces a domain calculus expression from an SQL query. This expression is passed to the *query rewriter* for further processing to produce an equivalent but simpler domain calculus expression. The query rewriter calls the *view processor* to translate SQL query fragments over the WSMED view into relevant search definitions that call web service operations. An important task for the query rewriter is to identify overlaps between different sub-queries and views calling the same web service operation. This requires knowledge about the key constraints. We show in Chapter 6 that such rewrites significantly improve the performance of queries to multi-level views of web services.

The rewritten query is translated into an algebra expression by a *cost-based optimizer* that uses a generic web service cost model as default. The algebra has operators to invoke web services and to apply external functions implemented in WSDL (e.g. for extraction of data from web service results). The algebra expression is finally interpreted by the *execution engine*. It uses the web service meta-database to convert between the WSMED data representation and a SOAP message when a web service operation is called.

A call to the web service manager is specified by *web service properties* such as *SOAPactionURI, style, encodingstyle, namespaceURI, and targetURL* (*Figure 19*). Furthermore, it contains the actual parameters of the operation, called the *input elements*. As shown by *Figure 24*, the web service manager uses two sub components to create a SOAP message: The *Result extractor* and the *SOAP Processor*. The result extractor and the SOAP processor are using SAAJ APIs.



a - input elements b - SOAPactionURI
c - style d -encodingstyle
e - namespaceURI f - targetURL
g - type h-maxoccurs

Figure 24. Web Service Manager

The SOAP processor creates a request SOAP message with a SOAP body (*Figure 25*). The SOAP processor requires additional information given web service properties to complete the SOAP message creation.

```
<SearchFoodByDescription>
   <FoodKeywords>Sweet</FoodKeywords>
   <FoodGroupCode>1900</FoodGroupCode>
</SearchFoodByDescription>
```

Figure 25. The content of request SOAP body

Finally the SOAP message is sent over the network to invoke the web service operation call. The response from the remote web service call is also received as a SOAP message. The contents of the SOAP message is extracted by the SOAP processor and sent it to the *result extractor*.

The result extractor extracts data from the SOAP message content(*Figure 26*). It requires the properties of the output elements (*Figure 19*) from the web service operation call, such as *type* and *maxoccurs,* to constructs the result data of the web service call. The result extractor retrieves the values for *type* and *maxoccurs* from the web service meta-database. The type of the operation's output elements is used by the result extractor for converting the XML data format into the data format used by WSMED. The attribute *maxoccurs* is used to construct the result object structure. Finally the result is sent back to the execution engine.

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap
.org/soap/envelope/" xmlns:xsi="http://www.w3.org
/2001/XMLSchema-instance" xmlns:xsd="http://www.w
3.org/2001/XML Schema">
```

```
  <soap:Body>
   <SearchFoodByDescriptionResponse xmlns="http:
//www.strikeiron.com/">
  <SearchFoodByDescriptionResult>
    <SearchByKeywordsOutput>
          <NDBNumber>02044</NDBNumber>
           <LongDescription>Basil, fresh
              </LongDescription>
          <FoodGroupCode>0200</FoodGroupCode>
        </SearchByKeywordsOutput>
      </SearchFoodByDescriptionResult>
      <ResponseStatus>
          <response_code>0</response_code>
          <response_string>Success
            </response_string>
      </ResponseStatus>
    </SearchFoodByDescriptionResponse>
</soap:Body>
</soap:Envelope>
```

Figure 26. The response SOAP message

The execution engine performs further post processing specified by the execution plan such as filtering and data transformation before the query result is delivered to the user.

5. WSMED Views

To illustrate and evaluate WSMED views we use a publicly available web service to access and search to the National Nutrient Database for US Department of Agriculture [67]. The database contains information about the nutrient content of over 6000 food items. It contains five different operations: *SearchFoodByDescriptions*, *CalculateNutrientValues*, *GetAllFoodGroupCodes*, *GetWeightMethods* and *GetRemainingHits*. We illustrate WSMED by the operation *SeachFoodByDescriptions* to search foods given a *FoodKeywords* or a *FoodGroupCode*. The operation returns *NdbNumber*, *LongDescription* and *FoodGroupCode* as the results. The WSMED view *food* in Table 2 allows SQL queries over this web service operation.

food:

ndb	keyword	descry	gpcode
19080	Sweet	Candies	1900
……	……	……	…….

Table 2. W*SMED view food*

For example, the following SQL query to the view *food* retrieves the description of foods that have food group code equal to 1900 and keyword ‘Sweet’:

```
select descry
from   food
where  gpcode = ’1900’
  and  keyword = ’Sweet’;
```

The WSMED view food is defined as follows:

```
create SQLview food (Charstring ndb ,Charstring keyword,
                     Charstring descry, Charstring gpcode)
as multidirectional
  (“ffff” select ndb, “”,descry, gpcode
          where foodDescr(“”,“”,)= <ndb,descry,gpcode>)
```

```
("fffb" select ndb, "",descry
        where foodDescr("",gpcode)= <ndb,descry,gpcode>)
("fbff" select ndb,descry,gpcode
        where foodDescr(keyword, "")= <ndb,descry,gpcode>)
("fbfb" select ndb, descry
      where foodDescr(keyword,gpcode)= <ndb,descry,
                                        gpcode>)
```

Figure 27. WSMED view definition

A given WSMED view can access many different web service operations in different ways. When the user defines a WSMED view he can specify the view by several different search definitions as declarative queries. They each implement a different way of retrieving data through web service operations. Different search definitions can be defined based on what view attributes are known or unknown in a query, the *view binding patterns*. The query optimizer automatically chooses the most promising search definitions for a given query to a WSMED view. Each search definition provides a different way of using the web service operations to retrieve food items. The binding patterns are:

- *ffff*- all the attributes of the view *food* are free in the query. That is, it does not specify any attribute selection value. In this case the search definition specifies that all food items should be returned.
- *fffb*- a value is specified only for fourth attribute *gpcode*. This means that the search definition returns all food items for a given food group code.
- *fbff*- a value is specified in the query only for the second attribute *keyword*, i.e. all food items associated with the given keyword are retrieved.
- *fbfb*- both the values *keyword* and *gpcode* are specified in the query, finding the relevant food items.

In our example query the binding pattern is *fbfb*. The search definitions are defined as queries that all call a function *foodDescr* in different ways. The function *foodDescr* is also defined as a declarative query (section 5.1) that wraps the web service operation *SearchFoodByDescription* given two parameters *FoodKeywords* and *FoodGroupCode*. It selects relevant pieces of a call to the operation *SearchFoodByDescription* to extract the data from the data structure returned by the operation.

To simplify sub-queries and provide heuristics for estimating selectivities, it is important for the system to know what attributes in the view are (compound) keys [18]. Therefore, the user can specify *key constraints* for a given view and set of attributes by a system function *declare_key*, e.g.:

```
declare_key(“food”, {”ndb”});
```

Key constraints are not part of WSDL and require knowledge about the semantics of the web service. In our example web service the attribute *ndb* is the key. The (compound) key attributes are specified as a set of attribute names for a given view *(e.g. {“ndb”}).* Multiple keys can be specified by several calls to *declare_key*.

The query optimizer may also need to estimate the cost to invoke the query, and an estimate of the size of its result, i.e. its fanout. Costs and fanouts can be specified explicitly by the user if such information is available. However, normally explicit cost information is not available and the cost is then estimated by a *default cost model* that uses available semantic information such as signatures, keys, and binding patterns to roughly estimate costs and fanouts.

Key constraints will be shown to be the most important semantic enrichment in our example, and additional costing information is not needed.

5.1 Search definitions

For defining search definitions WSMED uses AmosQL [48, 49] with special web service oriented data types. For example, the function *foodDescr* in *Figure 27*, has the following definition:

```
1.create function foodDescr (Charstring fkw,
2.                           Charstring fgc)
3.         ->Bag of <Charstring ndb,Charstring descry,
4.                   Charstring gpcode>
5. as select re[“NDBNumber”],re[“LongDescription”],
6.           re[“FoodGroupCode”]
7.    from  Record out, Record re
8.    where out =
9.      cwo(“http://ws.strikeiron.com/USDAData?WSDL”,
10.        “USDAData”,
11.        “SearchFoodByDescription”,
12.       {fkw, fgc}))
13.    and re in out[“SearchFoodByDescriptionResult”];
```

Given a food keyword, *fkw*, and a group code, *fgc*, the function *foodDescr* returns a bag of result rows extracted from the result of calling the web service operation named *SearchFoodByDescription*. Any web service operation can be called by the built-in function *cwo* (line 9). Its arguments are the URI of WSDL document that describes the service (line 9), the name of the service (line 10), an operation name (line 11), and the input argument list for the operation (line 12). The result from *cwo* is bound to the variable *out* (line 8). It holds the output from the web service operation temporarily

stored in WSMEDs local database. The system automatically converts the input and output messages from the operation into records, sequences, and other data structures. In our example, the argument list holds the parameters *FoodKeywords* and *FoodGroupCode* (line 12). The result *out* is a record structure from which only the attribute *SearchFoodByDescriptionResult* is extracted (line 13). Extractions are specified using the notation *s[k]*, where *s* is a variable holding a record, and *k* is the name of an attribute.

The search definition selects relevant parts of the result from calling the operation. In our example, the relevant attributes are *NDBNumber*, *LongDescription*, and *FoodGroupCode*, which are all attributes of a record stored in the attribute *SearchFoodByDescriptionResult* of the result record.

In our example it turns out that, when both *foodkeywords* and *foodgroupcode* are empty strings, the operation *SearchFoodByDescription* returns descriptions of all available food. On the other hand, if *foodkeywords* is empty but *foodgroupcode* is known, the web service operation will return all food with that group code. Similarly, if *foodgroupcode* is empty but *foodkeywords* is known, the web service operation will return all food with that keyword. If both *foodkeywords* and *foodgroupcode* are non-empty, the operation will return descriptions of all food items of the group code with matching keywords. This knowledge about the semantic of the web service operation *SearchFoodByDescription* is used to define the search definitions in *Figure 27*.

6. Impact of key constraints

To illustrate the impact of key constraints we define two other views in terms of the WSMED view *food*. The view *foodclasses* is used to classify food items while *fooddescriptions* describes each food item:

```
create view foodclasses(ndb, keyword, gpcode)
       as select ndb,keyword,gpcode from food;

create view fooddescriptions(ndb, descry)
       as select ndb, descr from food;
```

This scenario is natural for our example web service that treats *foodclasses* different from *fooddescriptions*. The following SQL query accesses these views.

```
select fd.descry
from   foodclasses fc, fooddescriptions fd
where  fc.ndb=fd.ndb and fc.gpcode='1900';
```

First the example query is translated by the calculus generator (*Figure 23*) into a domain calculus expression[4]:

```
{l | foodclasses(ndb,keyword,gpcode) ∧
     fooddescriptions (ndb,descry)∧
     descry=l ∧
      gpcode='1900'}
```

The definitions of the views *foodclasses* and *fooddescriptions* are defined in domain calculus as[5]:

```
foodclasses:{ndb, keyword, gpcode| food(ndb, keyword,
*, gpcode)}

fooddescriptions:{ndb,descry | food(ndb, *, descry,
*)}
```

[4] The variables are implicitly quantified.
[5] '*' means don't care.

Given these view definitions the calculus expression is transformed by the view expander (*Figure 23*) into:

```
{l |food(ndb,*,*,'1900') ∧
    food(ndb,*,l,*)}
```

Here the predicate *food* represents our WSMED view. At this point the added semantics that *ndb* is the key of the view play its vital part. Two predicates *p(k,a)* and *p(k,b)* are equal if *k* is a key and it is then inferred that the other attributes are also equal, i.e. *b=a* [18]. If a key constraint that *ndb* is the key is specified, this is used by the query rewriter to combine the two calls to *food*:

```
{l | food(*,*,l,'1900')}
```

Without knowing that *ndb* is the key the transformation would not apply and the system would have to join the two references to the view *food* in the expanded query. The simplification is very important to attain a scalable query execution performance as shown in Chapter 7.

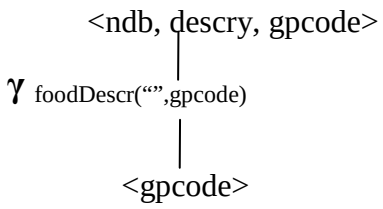


Figure 28. Execution plan with full semantic enrichment

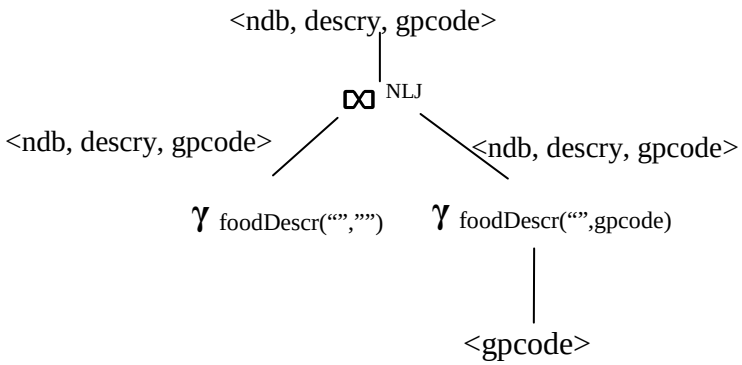


Figure 29. Naïve execution plan

The next step is to select the best search definition for the query. The heuristics is that if more than one search definition is applicable, the system chooses the one with the most variables bound. Since *l* is the query output and *gpcode* is given, the binding patterns *ffff* and *fffb* both apply, and the system chooses *fffb* because it is considered cheaper by default. The call to food then becomes:

```
{l | l=foodDescr(“”,”1900”)}
```

Similar to relational database optimizers, given the definition of *foodDescr*, a cost based optimizer generates the algebra expression in *Figure 28*, which is interpreted by the execution engine. The apply operator (γ) calls a function producing one or several result tuples for a given input tuple and bound arguments [27]. By contrast, *Figure 29* shows an execution plan for the non-transformed expression where the system does not know that *ndb* is key. It is using a nested loop join (NLJ) to join the search definitions. An alternative possible better plan based on hash join (HJ) that materializes the inner web service call is shown in Chapter 7.

In case no costing data is available about the search definitions (which is the case here), the system uses built in heuristics, i.e. a default cost model. In our case the cost based optimizer produces the plan in *Figure 28* which is optimal for our query.

7. Query Performance

To determine the impact of semantic enrichment on query performance, we have experimented with four different kinds of query execution strategies. They are:

1. The *naïve implementation* does not use any semantic enrichment at all and no binding pattern heuristics. That is, no *key* is specified for the *food* view definition and no default cost model was used. This makes the search definition be regarded as a black box called iteratively in a nested loop join since the system does not know that *foodDescr* returns a large result set when both arguments are empty. The execution plan in *Figure 29* shows the naïve plan.
2. With the *default cost model* the system assumes that the view *food* is substantially more expensive to use when attribute *gpcode* is not known than when it is known, i.e. it is cheaper to execute a search definition where more variables are bound. Still there is no key specified. *Figure 32* illustrates the plan.
3. *Figure 33* shows the execution plan with the default cost model and a *hash join strategy* where the results from web service operation calls are materialized by using hash join to avoid unnecessary web service calls. This can be done only when the smaller join operand can be materialized in main memory.
4. With *full semantic enrichment* the *key* of the view is specified. *Figure 28* shows the execution plan.

As shown in *Figure 30* the naïve strategy was the slowest one, somewhat faster than using the default cost model with nested loop join. The default cost model with a hash join strategy scaled significantly better, but requires enough main memory to hold the inner call to *foodDescr*. *Figure 31* compares the default cost model with hash join with the performance of full semantic enrichments. The hash join strategy was around five times slower. This clearly shows that semantic enrichments are critical for high performing queries over web services. The diagrams are based on the experimental results in Table 3 and the experiment was made by using the real values to actually retrieve the results through web service operations. VG, NF, S1, S2, S3, and S4 denote the value used for parameter *groupcode*, the number of food items (actual fanout), and the execution time in seconds for the four different strategies.

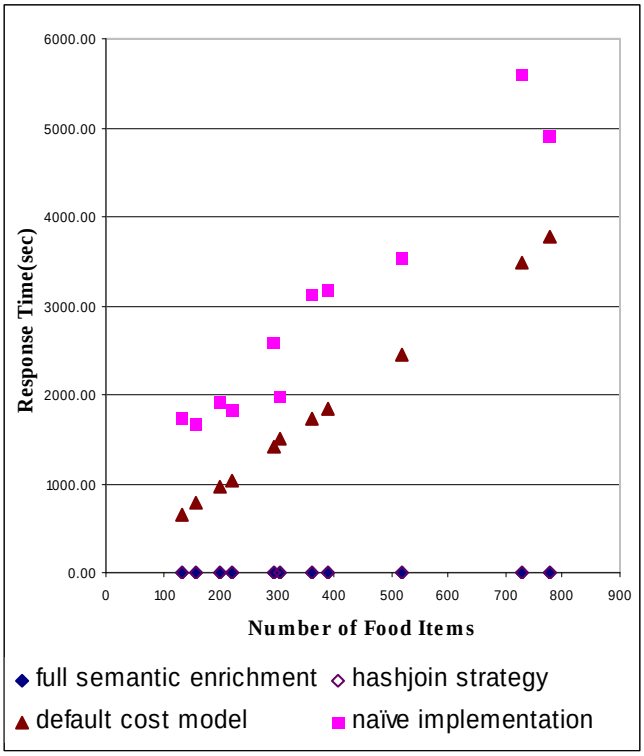


Figure 30. Performance comparison of the four query execution strategies

VG	NF	S1	S2	S3	S4
0900	303	1985.14	1512.74	5.77	1.22
0600	390	3177.28	1848.28	5.55	1.33
1400	219	1831.05	1041.74	5.50	1.08
1100	779	4891.13	3785.30	6.22	1.69
2000	157	1655.48	777.31	5.41	0.94
0800	359	3114.28	1723.28	5.59	1.35
0400	201	1914.23	955.38	6.38	1.08
1800	517	3524.34	2452.22	5.93	1.33

2200	132	1741.51	645.03	5.62	0.93
1900	293	2595.22	1415.98	5.58	1.19
1300	729	5596.38	3478.72	6.40	1.74

Table 3. *Experimental results*

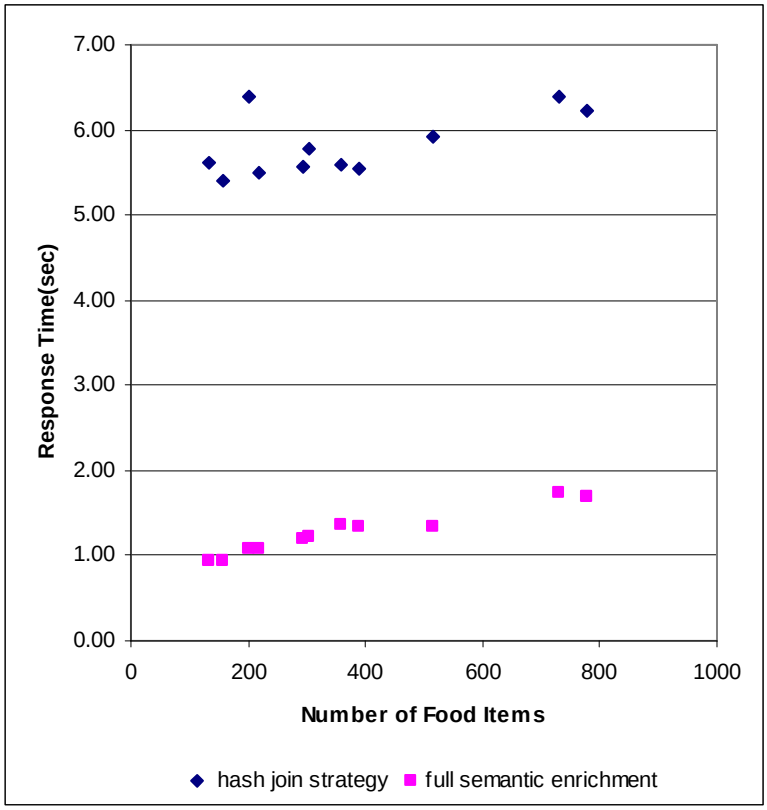


Figure 31. Performance comparison hash join and full semantic query execution strategies

With the naive strategy the system does not use any binding pattern heuristics and will call *foodDescr* with empty strings ($\gamma_{foodDescr(\text{“”},\text{“”})}$) which produces large costly results containing all food items in the outer loop. This is clearly very slow.

With the default cost model strategy the system assumes that queries over the view *food* produce larger results when the attribute *groupcode* is unknown than when it is known. Based on this the call to *foodDescr* with a

known *groupcode* value is placed in the outer loop of a nested loop join. This clearly is a better strategy than the naïve implementation.

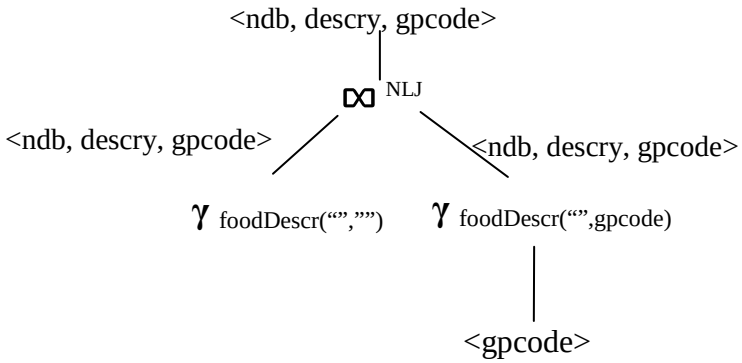


Figure 32. Execution plan for default cost model query execution strategy

Finally by utilizing key constraints in the WSMED view definition the system will know that the two applications of *foodDescr* can be combined into one call. With this *full enrichment strategy* only one web service operation call is required for execution of the query and no hash join is needed. We notice that this is the fastest and most scalable plan and that it needs no costing knowledge.

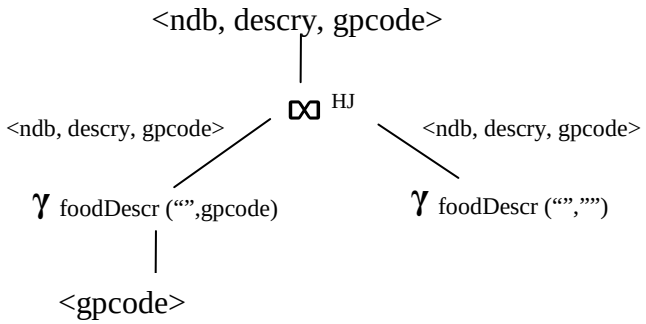


Figure 33. Execution plan for hash-join strategy

8. Related work

This chapter presents the overviews of the systems using the mediator approach to integrate data from heterogeneous data sources and the ontologies used to represent web services semantics. The important contributions and main functionalities of these systems briefly analyzed and compared with the WSMED system.

Web Service Management System (WSMS)

Figure 34 illustrates the WSMS system [53]. It provides DBMS-like capabilities when data sources are web services and enables queries against multiple web services. It consists of three major components. The *metadata component* manages metadata, registration of new web services, and mapping their schema to an integrated view provided to the client

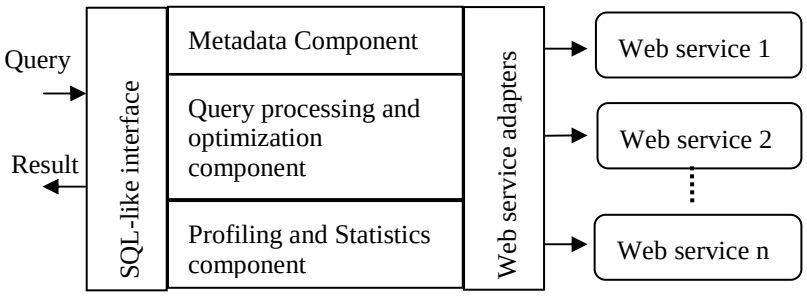


Figure 34. WSMS

A client can query the system with a given integrated schema. WSMED also resemble this feature with the support of the WSDL importer component. It automatically extracts the meta-data from the given WSDL document to represent it using the web service schema. In WSMS optimization and execution of declarative queries, as well as invoking relevant web services are managed by the *query processing and optimization component*. The *profiling and statistics component* profiles the response time of web services and maintains relevant statistics of data returned through web services. This component supports query optimization decision makings.

Precedence constraints exist when querying multiple web services. For example, to retrieve data from one web service *w1* it may expect output of another web service *w2*. Therefore *w2* will be queried before *w1* is accessed. Due to the restricted web service interfaces query processing over web services are considered by WSMS as work flow or pipeline processing. Initially some input data is fed to the WSMS and consequently this data is processed through a sequence of web service calls. Query processing in this scenario is sped up by pipelined parallelism because web services that are independent of precedent constraints can be executed in parallel.

The major contributions of WSMS are:

- When multiple web services are queried and some of them are executed in parallel the execution time is influenced by the slow response of web services. This is kind of bottleneck cost metric is formalized.
- Algorithms are developed for arranging web services in the pipeline to maximize the throughput in the presence of precedence constraints.
- When data sent to web services in chunks, the system estimates the optimal chunk size.

WSMED allows SQL queries to the wrapped web services as WSMS. WSMS currently concentrate on optimizing pipelined execution of web service queries. In contrast WSMED utilizes semantic enrichments for efficient query processing over multi-level views of web services. The parallel execution of web services is planned as the future work of WSMED.

Garlic

Garlic [55] supports the mediator approach to provide an integrated view of a variety of legacy data sources. Each data source is associated with a smart wrapper. In addition Garlic supports its own repository for *Garlic complex objects* that users can create to bind together existing objects from the data sources. Garlic's data model and programming interface are based upon the Object Database Management Group (ODMG) [69] standard.

Garlic objects are can be accessed both through the C++ programming interface and Garlic’s query language which is the extension of SQL to support path expressions and nested collections. Similarly, WSMED is using mediator-wrapper approach and SQL query interfaces. The global meta-data of Garlic describes a unified schema of the wrapped data sources and it doesn’t contain any a priori knowledge about the capabilities of the sources. By contrast, WSMED enriches the basic meta-data with user given binding patterns and key constraints to represent the sources’ capabilities

In Garlic wrappers model the contents of the underlying data sources as Garlic objects and then invoke the methods on the objects and retrieve the attributes. Other functionalities of Garlic wrappers are participating in query

planning and execution. Further, wrappers represent restricted declarative knowledge of source capabilities as they don't have any capability specification languages. Instead wrappers represent the sources' querying capabilities as methods. Each wrapper determines on a case-by-case basis the portion of the original user query its underlying source could answer.

By contrast, WSMED supports user provided semantic enrichments such as binding patterns to identify data sources' capabilities. Based on this knowledge, the query processor can invoke the appropriate web service operation call. Garlic's query processor didn't use any knowledge such as the key constraints to simplify the sub queries, unlike WSMED that utilizes key constraints to efficiently querying the sources. Further, WSMED optimizes multi-level views by using key constraints to integrate the different web service operations from different of web services.

TSIMMIS

TSIMMIS [20] also uses the mediation approach for data integration from multiple heterogeneous sources to provide users with integrated views of data. It transforms a user query for the integrated views into a collection of queries to sources and the results from the source queries are post-processed to answer the user's query. Wrappers are defined with the Wrapper Specification Language (WSL) to query the underlying data source. TSIMMIS define data source descriptions and query capabilities by rules.

WSMED also represents capabilities of sources accessible through web services but the capability specification is based on binding patterns which are simpler than the general rule based constraints of TSIMMIS. TSIMMIS has a logic-based object-oriented language *Mediation Specification Language (MSL)* used to specify the mediators. Mediators and wrappers are automatically produced by wrapper and mediator generators from the descriptions of their functionalities. By contrast WSMED uses a built-in common web service wrapper to access any kind of web services and allows users to create multi-level views and SQL queries over the views to mediate the web service operations. Furthermore the views are enriched with the user given semantics such as binding patterns and key constraints.

TSIMMIS uses a lightweight Object Exchange Model (OEM) to transport information among the components mediators, wrappers, and sources. The query language Lightweight Object Repository Language (LOREL) is used for user queries. Using all those components TSIMMIS build a mediator network which contains mediator-wrapper, wrapper-data source, and mediator-mediator interactions for information integration.

In TSIMMIS, query execution is performed in three phases. The logical plan generated by the *view expander* module is passed to the plan generator module. All the source queries that can process parts of a logical plan are identified during the initial step of the plan generation process. The capabilities of the sources are also taken into account. The second step is to

find the feasible execution sequences of source queries based on the binding requirements. During the final optimization phase the optimizer chooses the best feasible plan by applying standard query optimization techniques. However, TSIMMIS does not use any key constraints to simplify the queries for efficient query processing.

Information Manifold

Information Manifold [38] provides a uniform access to a set of heterogeneous information sources accessed through the internet and supports a mechanism for declarative description of contents and query capabilities of information sources. There is a clear distinction between a declarative source description and the real details for interaction with the information sources. The capabilities of sources are described using *capability records* that describe properties such as the number of attributes that can be retrieved as an output, the maximum and minimum number of inputs allowed, and the possible outputs from the source.

By contrast, WSMED uses binding patterns that are simpler compared with capability records. In Information Manifold the source description are used to prune the collection of information sources for a given user query and to generate executable query plans. It uses a relational model augmented with certain object-oriented features for describing and reasoning about the contents of information sources and keeps an integrated view of sources known as *world view* as a collection of virtual relations and classes.

Instead WSMED supports multi-level views with user given semantic enrichments. Information Manifold uses different interface programs to wrap different data sources. It devices the semantically correct query plan based on the ordering of sub goals of a given query in such a way that plan will be executable by adhering sources' capabilities. Unlike WSMED, there are no simplifications made based on the key constraints of the mediator view.

Web Query Optimizer System

The architecture of the mediator and wrapper in *Figure 35* [59] is proposed for Internet accessible web sources with limited query capabilities. Each call to a source defined as *WebSource Implementation* (WSI) that associates both capability and cost. The limited query capabilities of a source are defined by an *input-output relationship ior: Input→Output* where *Input* is a set of attributes that must be bound and *Output* is the set of projected output attributes.

Capability based rewriting of the query is processed by the *CBR Tool*. Another important contribution is the two-phased query optimizer. The first phase known as pre-optimization phase the *web query optimizer (wqo)* selects one or more WSIs. By using cost-based heuristics *wqo* evaluates the selection of WSI and chooses a good pre-plan. Then the *relational optimizer* devices a best plan.

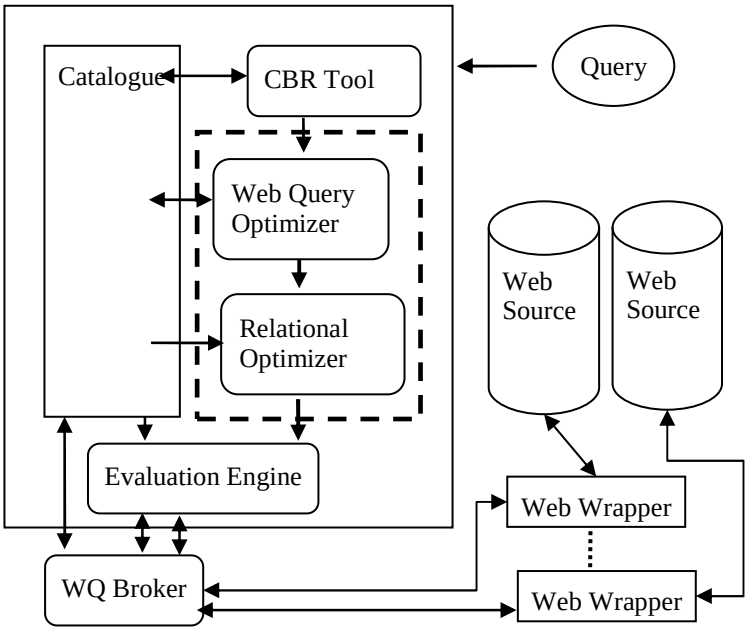


Figure 35. Mediator Architecture for web sources

The web query optimizer also follows a similar approach as WSMED to associate the source capabilities with binding patterns. In particular WSMED extends multi-directional foreign functions [40] to define semantically enriched views extracting data from the results of web service operations using an object-oriented query language [48, 49]. Furthermore, WSMED utilizes semantic enrichments of key constraints to optimize the multi-level views of web services with different capabilities while the web query optimizer only uses binding patterns to device query plan.

OWL-S

OWL-S [41] is an extension of the semantic web ontology language OWL to define web service ontologies. It provides a set of structures for describing the properties and capabilities the web services in unambiguous, computer-interpretable form. OWL-S enables:

- *Automatic Web service discovery:* is an automated process to locate web services that provide a certain class of service capabilities, while holding user specified constraints.
- *Automatic Web service invocation:* is the automatic invocation of a web service by a software component, given only a description of that service, in contrast to when that software component has to be pre-programmed to call that particular service. That is, OWL-S

provides an application programming interface that includes the semantics of the arguments of web service calls, and the semantics of the messages that are returned when the services succeed or fail. A software component can interpret this mark-up to understand what input is necessary to invoke the service, and what information will be returned.

- *Automatic Web service composition and interoperation*: involves the automatic selection, composition, and interoperation of web services to perform some complex task, given a high-level description of a user objective.

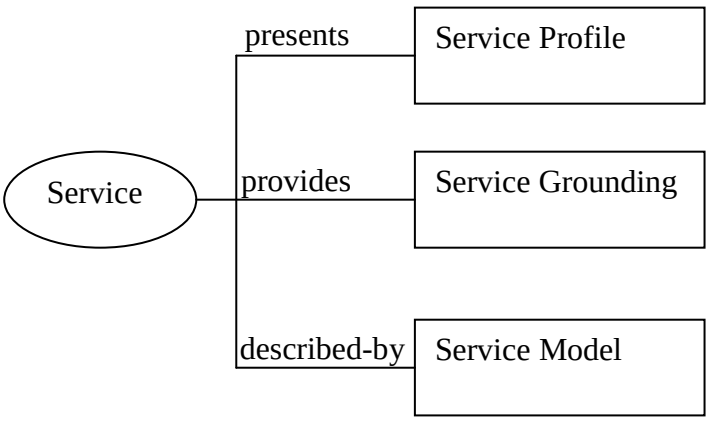


Figure 36. Service ontology

OWL-S supports a service ontology (*Figure 36*), where a *service profile* describes what a service does in a way understandable by a service seeking agent. The *service model* describes how to use the service, by detailing the semantic content of requests, the conditions under which particular output will occur, and, where necessary, the step by step processes leading to those output. The *service grounding* states the details of how a service can be accessed by specifying communication protocols, message formats, and other service-specific details such as port numbers used in contacting the service. In addition, the service grounding must specify, for each semantic type of input or output specified in the service model, an unambiguous way of exchanging data elements of that type with the service, i.e. serialization techniques. A service can be described by at most one service model, and a service grounding must be associated with exactly one service.

WSMED supports automatic web service invocation by providing web service description for any web services with a built-in function cwo (section 5.1). Service discovery and composition need to be analyzed further in future within the semantic web context.

9. Conclusions and future work

We devised a general approach to query data accessible through web services by defining multi-level views of data returned from web service operations and allowing SQL queries over these views. Given the URI of a WSDL description of a web service, WSMED automatically imports the basic meta-data from the WSDL file and represents them as a database schema. In terms of the database schema representation, a user can define multi-level views of web service operations using WSMED's query language. They can then be queried using SQL to search result structures from SOAP messages being the response of a web service operation calls. WSMED exploits the SOAP protocol to marshal messages to invoke a web service operation and makes use of HTTP for transmission of messages. We addressed the research question one in the Chapter 1 by deploying WSDL, SOAP and XML Schema with WSMED to wrap the data sources accessible through the web services. Further, WSMED allows the user to associate different search definitions with a given WSMED view, depending on the binding pattern of a query to the view, i.e. what view attributes are known. A WSDL operation signature description does not provide any information about which parts of the signature is a key to the data accessed through the operation. Instead the user can add key constraints when defining WSMED views.

The performance of queries to multi-level WSMED views varied very substantially depending on what query processing strategy is used. We evaluated four different query processing strategies using WSMED and existing web services. Our experiments showed that binding patterns and key constraints are essential for scalable performance when multi-level views are defined.

We gave an answer to research question two by defining multi-level views and showing that those views can be queried with SQL. The query optimizer automatically select the best search definition based on the heuristics of the provided binding patterns and simplifies the web service calls by identifying overlaps between different sub-queries and views calling the same web service operation. Normally explicit cost information is not available for call to a web service operation and the cost is then estimated by a *default cost model* that uses available semantic information such as keys, and binding patterns to roughly estimate costs and fanouts. By incorporating these features, WSMED partially answered research question three.

Generally web service mediation involves more than one operation from different web services. The common queries in this scenario have joins of views and those views are created in terms of the different operations from the diverse web services. Some web service operation calls need inputs from some of the other operations' outputs, namely *precedence constraints*. To optimize these kinds of web service operation calls in WSMED, we have to investigate synergies of pipelined execution strategies of web service operation calls as in WSMS [53]

Generally users have to pay to access commercial web services. Reducing the number of redundant web service calls is a decisive benefit from an economic perspective. To gain this kind of performance benefit, the pruning of superfluous web service operation calls is crucial especially those calls embedded with the join queries. *Adaptive data partitioning* (ADP) [32], which is based on the idea of dividing the source data into regions, each executed by different, complementary plans, is also a useful approach. Some prominent approaches like, passing adaptive information to prune useless results in the early stage of query execution without interrupting the query plan need to be studied. These kinds of adaptive query processing techniques need to be investigated further to improve the query optimization capability of the WSMED. Incorporating *partial evaluation*, a program transformation technique [33], during the query optimization is another interesting approach to investigate in this context. The partial evaluation reduces queries before the cost-based optimization by simplifying the query by iteratively evaluating some predicates at compile time until a fix-point is reached. These are some future directions to provide further answers to research question three.

Currently the semantic enrichments are added manually. Future work will investigate when it is possible to automate this and how to efficiently verify that an enrichment is valid. For example, determination of key constraints is currently added manually, and this could be automated by querying the source. Another issue is how to minimize the required semantic enrichments by self tuning cost modeling techniques [29] based on monitoring the behavior of web service calls.

Currently we assume all web service operations used in queries are side effect free. Another issue is semantic enrichments to allow SQL updates of web service data views.

The semantic web is an emerging prominent approach for the future data representations where WSDL working groups are proposing standards to incorporate semantic web representations [62]. We will next investigate the mediation of web services based on semantic web representations like RDF[35] and RDF-Schema [10].

We summarize that we answered research questions one and two by developing WSMED. Research question three is partially answered and

further investigation is needed for a complete answer. Research questions four and five is going to be answered in the ongoing work.

Appendix A: WSDL document structure

```
<wsdl:definitions name="nt[6]"? targetNamespace="uri"?>
 <import namespace="uri" location="uri"/>*[7]
  <wsdl:documentation .... /> ?[8]

 <wsdl:types> ?
  <wsdl:documentation .... />?
  <xsd:schema .... />*
  <-- extensibility element --> *
 </wsdl:types>

 <wsdl:message name="nt"> *
  <wsdl:documentation .... />?
  <part name="nt" element="qname[9]"? type="qname"?/> *
 </wsdl:message>

 <wsdl:portType name="nt">*
  <wsdl:documentation .... />?
  <wsdl:operation name="nt">*
   <wsdl:documentation .... /> ?
   <wsdl:input name="nt"? message="qname">?
    <wsdl:documentation .... /> ?
   </wsdl:input>
   <wsdl:output name="nt"? message="qname">?
     <wsdl:documentation .... /> ?
    </wsdl:output>
   <wsdl:fault name="nt" message="qname"> *
    <wsdl:documentation .... /> ?
   </wsdl:fault>
  </wsdl:operation>
 </wsdl:portType>

 <wsdl:binding name="nt" type="qname">*
  <wsdl:documentation .... />?
  <-- extensibility element --> *
  <wsdl:operation name="nt">*
   <wsdl:documentation .... /> ?
```

[6] nt – nmtoken[9]
[7] ***** - zero or more
[8] ? - zero or one
[9] XML qualified name [6]

```
   <-- extensibility element --> *
   <wsdl:input> ?
    <wsdl:documentation .... /> ?
    <-- extensibility element -->
   </wsdl:input>
   <wsdl:output> ?
    <wsdl:documentation .... /> ?
    <-- extensibility element --> *
   </wsdl:output>
   <wsdl:fault name="nt"> *
    <wsdl:documentation .... /> ?
     <-- extensibility element --> *
    </wsdl:fault>
  </wsdl:operation>
 </wsdl:binding>

 <wsdl:service name="nt"> *
  <wsdl:documentation .... />?
  <wsdl:port name="nt" binding="qname"> *
   <wsdl:documentation .... /> ?
   <-- extensibility element -->
  </wsdl:port>
  <-- extensibility element -->
  </wsdl:service>
   <-- extensibility element --> *
</wsdl:definitions>
```

References

1. T.Andrews, Business Process Execution Language for Web Services, Version 1.1, *published online at ftp://www6.software.ibm.com/software/developer /library/ws-bpel.pdf*, 2003
2. S.Bajaj et al, Web Services Policy Framework (WSPolicy), *published online at http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-polfram/ws-policy-2006-03-01.pdf*, 2006
3. K.Ballinger, P.Brittenham, A.Malhotra, W.A. Nagy, and S.Pharies, Web Services Inspection Language (WS-Inspection), *published online at ftp://www6.software.ibm.com/software/developer/library/ws-wsilspec.pdf*, 2001
4. D. Beckett, RDF/XML Syntax Specification (Revised), W3C Recommendation, *published online at http://www.w3.org/TR/rdf-syntax-grammar/*, 2004
5. T.Bellwood et al, UDDI Version 3.0.2, UDDI Spec Technical Committee Draft, *published online at http://uddi.org/pubs/uddi_v3.htm#_Toc85907967*, 2004
6. P.V.Biron, K.Permanente, and A.Malhotra, XML Schema Part 2: Datatypes Second Edition, W3C Recommendation, *published online at http://www.w3.org/TR/xmlschema-2/#Qname*, 2004
7. S.Boag, D.Chamberlin, M.F. Fernández, D.Florescu, J.Robie, and J.Siméon, XQuery 1.0: An XML Query LanguageW3C Candidate Recommendation, *published online at http://www.w3.org/TR/xquery/*, 2006
8. D.Booth, H.Haas, F.McCabe, E.Newcomer, M.Champion, C.Ferris, and D.Orchard, Web Services Architecture,W3C Working Group Note, *published online at http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/* , 2004
9. T.Bray, J.Paoli, C. M. Sperberg-McQueen, E.Maler, and F.Yergeau, Extensible Markup Language (XML) 1.0 (Third Edition),W3C Recommendation, *published online at http://www.w3.org/TR/2004/REC-xml-20040204/* , 2004
10. D. Brickley and R. V. Guha, RDF Vocabulary Description Language 1.0: RDF-Schema, *published online at http://www.w3.org/TR/rdf-schema/*, 2004
11. E.Christensen, F.Curbera, G.Meredith, and S. Weerawara na, Web services description language (WSDL) 1.1., W3C Recommendation, *published online at http://www.w3.org/TR/wsdl*, 2001
12. J.Clark and S.DeRose, XML Path Language (XPath) Version 1.0, W3C Recommendation, *published online at http://www.w3.org/TR/xpath*, 1999
13. E.F.Codd, A relational model of data for large shared data banks, *Communications of the ACM*, 13(6),1970,pp 377-387.
14. A.Eisenberg, and J.Melton, SQL:1999, formerly known as SQL3, *published online at http://www.sigmod.org/record/issues/9903/standards.pdf.gz*
15. A.Eisenberg, and J.Melton, SQL/XML is Making Good Progress, *ACM SIGMOD Record*, 31(2), 2002
16. R.Elamasri, and S.M.Navathe, *Fundamentals of Database Systems*, 4th Edition, ISBN 0-321-20448-4, Pearson Education, 2004, pp 855-856

17. L.Ennser, C.Delporte, M.Oba, and K.Sunil, Integrating XML with DB2 XML Extender and DB2 Text Extender, *published online at http://www.redbooks.ibm.com/redbooks/pdfs/sg246130.pdf , IBM Corp*., 2001
18. G. Fahl, and T. Risch, Query Processing over Object Views of Relational Data, *The VLDB Journal* , 6(4), 1997, pp 261-281.
19. D.C. Fallside, and P.Walmsley, XML Schema Part 0: Primer Second EditionW3C Recommendation, *published online at http://www.w3.org/TR/xmlschema-0/*, 2004
20. H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A.Rajaraman, Y. Sagiv, J.D. Ullman, V. Vassalos, and J.Widom, The TSIMMIS Approach to Mediation: Data Models and Languages, *In Journal of Intelligent Information Systems*, 8(2): 117-132, 1997.
21. H.Garcia-Molina, J.D Ullman, and J.Widom, *Database Systems: The Complete Book*, ISBN 0-13-098043-9, Prentice Hall, 2002, pp 1047-1069
22. G.Gardarin, A.Mensch, and A.Tomasic, An Introduction to the e-XML Data Integration Suite, *Proc. 8th International Conference on Extending Database Technology (EDBT '02)*, 2002, pp. 297–306.
23. R.Goldman, J.McHugh, and J.Widom, From Semistructured Data to XML: Migrating the Lore data Model and Query Language, *Proc. 2nd International workshop on the Web and Databases*, 1999
24. G.Graefe, Query evaluation techniques for large databases, *ACM Computing Surveys (CSUR)*, 25(2), 1993, pp 73-169
25. M.Gudgin, M.Hadley, N.Mendelsohn, J.Moreau, and H.Frystyk Nielsen, SOAP Version 1.2 Part 1: Messaging Framework,W3C Recommendation, *published online at http://www.w3.org/TR/soap12-part1/* ,2003
26. D.L.M.Guinness and F.v.Harmelen, OWL Web Ontology Language Overview, W3C Recommendation, *published online at http://www.w3.org/TR/owl-features/*, 2004
27. L.M.Haas, D. Kossmann, E. Wimmers, and J .Yang, Optimizing Queries across Diverse Data Sources, *Proc. of the 23rd Very Large Data Bases Conference(VLDB 1997)*, 1997
28. A. Halverson, V.Josifovski, G.Lohman, H.Pirahesh, and M. Mörschel, ROX: Relational Over XML, *Proc. 30th Very Large Data Bases Conference (VLDB 2004)*, 2004, pp 264-275
29. Z.He, B.S.Lee, and R.Snapp, Self-Tuning Cost Modeling of User-Defined Functions in an Object-Relational DBMS, *ACM Transactions on Database Systems*, 30(3), pp 812-853, 2005.
30. J.M.Hellerstein, J.F.Naughton, and A:Pfeffer, Generalized Search Trees for Database Systems, *Proc. 21st International Conference on Very Large Data Bases Conference(VLDB 95)*, 1995, pp 562-573
31. A.R.Hurson, M.W.Bright, and S.H.Pakzad, Multidatabase Systems: An Advanced Solution for Global Information Sharing, *IEEE Computer Society Press*, 1994
32. Z.G.Ives, A.Y.Halvey, and D.S.Weld, Adapting to Source Properties in Processing Data Integration Queries, *Proc. SIGMOD conference*, 2004
33. N. D. Jones, An Introduction to Partial Evaluation, *ACM Computing Surveys*, 28(3), 1996
34. V.Josifovski, T.Katchaounov, and T.Risch, Optimizing queries in distributed and composable mediators, *Proc. 4th International. Conference on Cooperative Information Systems (CoopIS'99)*, 1999.

35. G. Klyne and J. J. Carroll, Resource Description Framework (RDF): Concepts and Abstract Syntax, *published online at http://www.w3.org/TR/rdf-concepts/*, 2004.
36. M.Kristjánsson, Building with Oracle XML Database, *published online at http://www.oracle.com/technology/oramag/oracle/04-sep/o54xml.html*, 2004
37. K.Lawrence, C.Kaler, A.Nadalin, M.Gudgin, A.Barbir, and H.Granqvist, WS-SecurityPolicy v1.0, OASIS Working Draft, *published online at http://www.oasis-open.org/committees/download.php/15979/oasis-wssx-ws-securitypolicy-1.0.pdf*, 2005
38. A.Y.Levy et al., Querying Heterogeneous Information Sources Using Source Descriptions, *Proc. of 22nd Very Large Data Bases Conference(VLDB 96)*, 1996
39. C.Li et al., Capability Based Mediation in TSIMMIS, *Proc. of the 1998 ACM SIGMOD international conference on Management of data*, 564-566,1998.
40. W. Litwin, and T. Risch, Main Memory Oriented Optimization of OO Queries using Typed Datalog with Foreign Predicates, *IEEE Transactions on Knowledge and Data Engineering* , 4(6), 517-528, 1992.
41. D. Martin et al., OWL-S: Semantic Markup for Web Services, *published online at http://www.ai.sri.com/daml/services/owl-s/1.2/overview/*
42. J.Naughton, et al. , The Niagara Internet Query System, *IEEE Data Engineering bulletin*, 24(2) , 2001, pp. 27-33
43. M.Nicola, B. Linden, Native XML Support in DB2 Universal Database, *Proc. of the 31st Very Large Data Bases Conference(VLDB 2005)*, 2005
44. Y.Papakonstantinou, A.Gupta, and L.Haas, Capabilities-base query rewriting in mediator systems, *Proc. of Conference on Parallel and Distributed Information Systems*, 1996
45. E.Prud'hommeaux, and A,Seaborne, SPARQL Query Language for RDF,W3C Working Draft, *published online at http://www.w3.org/TR/rdf-sparql-query/*, 2006
46. J.Postel, SIMPLE MAIL TRANSFER PROTOCOL, RFC 821, *published online at http://www.ietf.org/rfc/rfc0821.txt*, 1982
47. J. Postel, and J. Reynolds, FILE TRANSFER PROTOCOL (FTP), *published online at http://tools.ietf.org/html/rfc959*,1985
48. T.Risch and V.Josifovski, Distributed Data Integration by Object-Oriented Mediator Servers, *Concurrency and Computation: Practice and Experience J.*, 13(11), John Wiley & Sons, 2001, pp 933-953.
49. T.Risch, V.Josifovski, and T.Katchaounov, Functional Data Integration in a Distributed Mediator System, in P.Gray, L.Kerschberg, P.King, and A.Poulovassilis (eds.): *Functional Approach to Data Management - Modeling, Analyzing and Integrating Heterogeneous Data*, ISBN 3-540-00375-4, Springer, 2003, pp 211-238.
50. M.Scardina, XML Storage Models: One Size Does Not Fit All, *published online at http://www.oracle.com/technology/oramag/webcolumns/2003/techarticles/scardina_xmldb.html*, 2001
51. A.Seaborne, RDQL - A Query Language for RDF, W3C Member Submission, *published online at http://www.w3.org/Submission/RDQL/*, 2004
52. D. Shipman, The Functional Data Model and the Data Language DAPLEX, *ACM Transactions on Database Systems*, 6(1), 140-173, 1981.
53. U.Srivastava, J.Widom, K.Munagala, and R.Motwani, Query Optimization over Web Services, *Proc Very Large Database Conference(VLDB 2006)*, 2006
54. H.S.Thompson, D.Beech, M.Maloney, and N.Mendelsohn, XML Schema Part 1: Structures second edition, W3C Recommendation, *published online at http://www.w3.org/TR/xmlschema-1/*, 2004

55. M. Tork-Roth, and P. Schwarz, Don't Scrap It, Wrap It! A Wrapper Architecture for Legacy Data Sources, *Proc. 23rd Very Large Data Bases Conference(VLDB 1997)*, 1997.
56. V.Vassalos, and Y.Papakonstantinou, Describing and Using Query Capabilities of Heterogeneous Sources, *Proc. 23rd Very Large Data Bases Conference(VLDB 97)*, 1997
57. G. Wiederhold, Mediators in the Architecture of Future Information Systems, *IEEE Computer*, 25(3), 1992, pp 38-49.
58. R.Yerneni, C.Li, H.Garcia-Molina, and J.D:Ullman, Computing capabilities of mediators, *Proc. International conference on Management of Data* , 1999, pp 443-454.
59. V.Zadorozhny, L.Raschid, M.E.Vidal, T.Urban, and L.Bright, Efficient Evaluation of Queries in a Mediator for WebSources, *Proc. of the 2002 ACM SIGMOD international conference on Management of data*, 85-96, 2002.
60. DB2 XML Extender, *published online at http://www-4.ibm.com/software/data /db2/extenders/xmlext*
61. HTTP - Hypertext Transfer Protocol, W3C Architecture domain, *published online at http://www.w3.org/Protocols/*
62. Semantic Web Activity, W3C Technology and Society domain, *published online at http://www.w3.org/2001/sw/*
63. TRANSMISSION CONTROL PROTOCOL, *published online at http://www.ietf.org/rfc/rfc793.txt*, 1981
64. XML-Related specifications (SQL/XML), *published online at http://www.sqlx. org/SQL-XML-documents/5FCD-14-XML-2004-07.pdf*, 2005
65. https://saaj.dev.java.net/
66. http://sourceforge.net/projects/wsdl4j
67. http://ws.strikeiron.com/USDAData?WSDL
68. http://www.castor.org/index.html
69. http://www.odmg.org/
70. http://www.w3.org/2002/ws/arch/4/management/
71. http://www.w3.org/XML/Query/

Recent licentiate theses from the Department of Information Technology

2006-012 Stefan Blomkvist: *User-Centred Design and Agile Development of IT Systems*

2006-011 Åsa Cajander: *Values and Perspectives Affecting IT Systems Development and Usability Work*

2006-010 Henrik Johansson: *Performance Characterization and Evaluation of Parallel PDE Solvers*

2006-009 Eddie Wadbro: *Topology Optimization for Acoustic Wave Propagation Problems*

2006-008 Agnes Rensfelt: *Nonparametric Identification of Viscoelastic Materials*

2006-007 Stefan Engblom: *Numerical Methods for the Chemical Master Equation*

2006-006 Anna Eckerdal: *Novice Students' Learning of Object-Oriented Programming*

2006-005 Arvid Kauppi: *A Human-Computer Interaction Approach to Train Traffic Control*

2006-004 Mikael Erlandsson: *Usability in Transportation -- Improving the Analysis of Cognitive Work Tasks*

2006-003 Therese Berg: *Regular Inference for Reactive Systems*

2006-002 Anders Hessel: *Model-Based Test Case Selection and Generation for Real-Time Systems*

2006-001 Linda Brus: *Recursive Black-box Identification of Nonlinear State-space ODE Models*

2005-011 Björn Holmberg: *Towards Markerless Analysis of Human Motion*

2005-010 Paul Sjöberg: *Numerical Solution of the Fokker-Planck Approximation of the Chemical Master Equation*

2005-009 Magnus Evestedt: *Parameter and State Estimation using Audio and Video Signals*

2005-008 Niklas Johansson: *Usable IT Systems for Mobile Work*

2005-007 Mei Hong: *On Two Methods for Identifying Dynamic Errors-in-Variables Systems*

2005-006 Erik Bängtsson: *Robust Preconditioned Iterative Solution Methods for Large-Scale Nonsymmetric Problems*

2005-005 Peter Nauclér: *Modeling and Control of Vibration in Mechanical Structures*

UPPSALA
UNIVERSITET

Department of Information Technology, Uppsala University, Sweden