UPPSALA
UNIVERSITET

Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 755

Querying Data Providing Web Services

MANIVASAKAN SABESAN

ACTA
UNIVERSITATIS
UPSALIENSIS
UPPSALA
2010

ISSN 1651-6214
ISBN 978-91-554-7852-0
urn:nbn:se:uu:diva-128928

Dissertation presented at Uppsala University to be publicly examined in Room 1211, Building 1, Polacksbacken, Lägerhyddsvägen 2, Uppsala, Friday, October 8, 2010 at 13:15 for the degree of Doctor of Philosophy. The examination will be conducted in English.

Abstract

Sabesan, M. 2010. Querying Data Providing Web Services. Acta Universitatis Upsaliensis. *Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology* 755. 37 pp. Uppsala. ISBN 978-91-554-7852-0.

Web services are often used for search computing where data is retrieved from servers providing information of different kinds. Such *data providing web services* return a set of objects for a given set of parameters without any side effects. There is need to enable general and scalable search capabilities of data from data providing web services, which is the topic of this Thesis.

The Web Service MEDiator (WSMED) system automatically provides relational views of any data providing web service operations by reading the WSDL documents describing them. These views can be queried with SQL. Without any knowledge of the costs of executing specific web service operations the WSMED query processor automatically and adaptively finds an optimized parallel execution plan calling queried data providing web services.

For scalable execution of queries to data providing web services, an algebra operator *PAP* adaptively parallelizes calls in execution plans to web service operations until no significant performance improvement is measured, based on monitoring the flow from web service operations without any cost knowledge or extensive memory usage.

To comply with the Everything as a Service (XaaS) paradigm WSMED itself is implemented as a web service that provides web service operations to query and combine data from data providing web services. A web based demonstration of the WSMED web service provides general SQL queries to any data providing web service operations from a browser.

WSMED assumes that all queried data sources are available as web services. To make any data providing system into a data providing web service WSMED includes a subsystem, the *web service generator,* which generates and deploys the web service operations to access a data source. The WSMED web service itself is generated by the web service generator.

Keywords: views of web service operations, web service queries, adaptive parallelization, query optimization

Manivasakan Sabesan, Department of Information Technology, Computing Science, Box 337, Uppsala University, SE-75105 Uppsala, Sweden

ISSN 1651-6214
ISBN 978-91-554-7852-0
urn:nbn:se:uu:diva-128928 (http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-128928)

என் பெற்றோருக்கு இந்நூல் சமர்ப்பணம்

To my parents

List of Papers

This Thesis is based on the following papers, which are referred to in the text by their Roman numerals.

I Manivasakan Sabesan, and Tore Risch, Web Service Mediation Through Multi-level Views, International Workshop on Web Information Systems Modeling (WISM 2007), *In Proc. Workshops and Doctoral Consortium*, tapir academic press , pp 755-766, 2007.

II Manivasakan Sabesan, and Tore Risch , Adaptive Parallelization of Queries over Dependent Web Service Calls, 1st IEEE Workshop on Information & Software as Services(WISS 2009), *In Proc. 25th International Conference on Data Engineering (ICDE2009)*,IEEE Computer Society, pp 1725-1732, 2009.

III Manivasakan Sabesan, and Tore Risch, Adaptive Parallelization of Queries to Data Providing Web Service Operations, *submitted for conference publication,* 2010.

IV Manivasakan Sabesan, Tore Risch, and Feng Luan, Automated Web Service Query Service, *accepted for publication in International Journal of Web and Grid Services (IJWGS),* Inderscience, Volume 6, Number 4, 2010.

Reprints of papers I, II, and IV were made with permission from the respective publishers.

Other Related Publications

V Manivasakan Sabesan, Tore Risch, and Gihan Wikramanayake, Querying Mediated Web Services , *In Proc. 8th International Information Technology Conference (IITC 2006)*, Infotel Lanka Society Ltd, pp 39-44, 2006.

VI Manivasakan Sabesan, Querying Mediated Web Services, *Thesis for the degree of Licentiate of Philosophy in Computer Science with specialization in Database Technology*, Department of Information Technology, Uppsala University, 2007.

VII Manivasakan Sabesan, and Tore Risch, Web Service Query Service, *In Proc. 11th International Conference on Information Integration and Web-based Applications & Services (iiWAS2009)*, ACM and Austrian Computer Society, pp 692-697, 2009.

VIII Manivasakan Sabesan, and Tore Risch, Adaptive Parallelization of Queries Calling Dependent Data Providing Web Services, *In Divyakant Agrawal, K. Selcuk Candan and Wen-Syan Li (Editors): New Frontiers in Information and Software as Service,* Lecture Notes in Business Information Processing (LNBIP) series, Springer-Verlag, 2010.

Contents

Abbreviations

AMOS	Active Mediators Object System
CDM	Common Data Model
DBMS	Data Base Management System
FTP	File Transfer Protocol
HTTP	Hypertext Transport Protocol
PAP	Parameterized Adaptive Parallelization
RDBMS	Relational Database Management System
SMTP	Simple Mail Transfer Protocol
SOAP	Simple Object Access Protocol
TCP	Transmission Control Protocol
UDDI	Universal Description Discovery and Integration
URL	Uniform Resource Locator
WQL	WSMED Query Language
WSDL	Web Services Description Language
WSMED	Web Service MEDiator
XaaS	Everything as a Service
XML	eXtensible Mark up Language

1. Introduction

The growth of the Internet and the emergence of XML for data interchange in a loosely coupled way have increased the importance of web services [7] incorporating standards such as SOAP [18], WSDL [9], and XML Schema [42]. Web services support an application infrastructure by defining a set of *operations* that can be invoked over the communication network. Web service operations are self contained using meta-data to describe data types of their arguments and results, i.e. their signatures, using the Web Service Description Language, WSDL. Thus web services provide a general infrastructure for remote calls to predefined operations.

Web services are often used for retrieving data from servers providing information of different kinds. A *data providing web service operation* returns collections of objects for a given set of arguments without any side effects. This is known as a form of search computing [8]. However, data providing web service operations don't provide general query language or view capabilities to search and join data from one or several data providing web services, which is the topic of this Thesis.

As an example, consider a query to find information about places in some of the US states along with their zip codes and weather forecasts. Four different data providing web service operations can be used for answering this query. First the *GetAllStates* operation from the web service *GeoPlaces* [10] is called to retrieve the desired states. The *GetInfoByState* operation by *USZip* [36] returns the zip codes for a given US State. The *GetPlacesInside* operation by *Zipcodes* [11] retrieves the places located within a given zip code area. The *GetCityForecastByZip* operation by *CYDNE* [12] returns weather forecast information for a given zip code.

A mediator [39] is a system that allows data from different data sources to be combined and queried. In our setting a mediator enables queries joining data from different data providing web service operations.

In this work it is investigated how to build a general system for scalable querying of data providing web service operations. The development of a web service based mediator prototype called *WSMED (Web Service MEDiator)* is expected to provide insights into a number of research questions:

1. To what extent can web service standards, such as WSDL and SOAP, be utilized by a mediator to query data providing web service operations efficiently and scalable?

2. How can views of data providing web service operations for a high level query language such as SQL be automatically generated based on WSDL descriptions?
3. How can query optimization and rewrite techniques be used to provide efficient and scalable search from different data providing web services?
4. How can the query optimizer speed up general queries calling web service operations without knowing their costs?
5. How can data sources that are not accessible via web services be simply transformed into data providing web service operations, making them queryable by a web service mediator?
6. How can the *Everything as a Service* (XaaS) paradigm [33] be used for querying data providing web services? That is, can a web service mediator be provided as a web service and be used in a browser without any additional software installations and hardware setups?

To answer the research questions we have developed and evaluated the *WSMED* prototype, which enables high level and scalable queries over any data providing web services.

WSMED can access dynamically any web service operation by retrieving its WSDL document. WSMED contains a generic web service database for representing descriptions of any WSDL document. This database is used to dynamically construct the web service operation calls required to process a query. This provides the answer to research question **one**.

A web service operation is presented by WSMED as an SQL view. SQL queries can be expressed in terms of these views. For a given web service WSMED automatically generates such views for all its web service operations based on its WSDL definition. The views are generated using the internal *WSMED query language* (WQL), which has support for the web service data types. The automatic generation of SQL views provides the answer to research question **two**.

Web service operations are usually parameterized where input parameters have to be bound before they are called. Two web service operation calls in a query are *dependent* if one of them requires as input an output from the other one, otherwise they are *independent*. In the above example, the web service operations *GetPlacesInside* and *GetCityForecastByZip* are dependent on *GetInfoByState* but independent of each other. A challenge here is to develop methods to optimize queries containing both dependent and independent web service calls. In general such optimization depends on some unknown web service properties. Those properties are not explicitly available and depend on the network and runtime environments when and where the queries are executed. In such scenarios it is very difficult to base execution strategies on a static cost model, as is done in relational databases.

To improve the response time without a cost model, WSMED uses an approach to automatically parallelize the web service calls at run time while

keeping the dependencies among them. For each web service operation call in a query the WSMED query optimizer generates a parameterized sub-plan, called a *plan function,* which encapsulates the web service operation call and makes data transformations such as nesting, flattening, filtering, data conversions, and calls to other plan functions. WSMED will decompose the query plan to guarantee that dependent web service operations are called with proper parameter bindings.

The query performance is often improved by setting up several parameterized web service calls in parallel rather than to call the operations in sequence for different parameters. In WSMED multi-level parallel execution plans are automatically generated as process trees where different plan functions are called in parallel in different processes, called *query processes*. For adaptive parallelization of queries with web service operation calls, the algebra operator *PAP* (Parameterized Adaptive Parallelization) is implemented. *PAP* dynamically modifies a parallel plan by local monitoring of plan function calls without any cost knowledge.

The adaptive parallelization of queries calling data providing web service operations provides the answer research questions **three** and **four**.

WSMED assumes that queried data sources are available as web services. To implement a new data providing web service for a data source requires development of software to access the data source from web service operations, defining a WSDL document to describe the interface, and deploying the interface code. To simplify the implementation of data providing web services WSMED includes a subsystem, the *web service generator,* which generates and deploys the web service operations to access a data source. The programmer first defines data source interface functions to access the data source as queries by developing a wrapper in the extensible wrapper/mediator system Amos II [32]. Once the interface functions are defined the WSMED web service generator automatically generates the corresponding web service operations and dynamically deploys them without restarting the web server. The signature of each so generated web service operation is defined in an automatically generated WSDL document based on the signatures of the interface functions. The WSDL document completely describes the web service interfaces of the deployed operations. Each operation calls the interface function and sends back the result as a collection. Interface functions have been defined for many different kinds of data sources [1], e.g. relational DBMSs, semantic web data, topic maps, and CAD servers.

Automatic generation and deployment of web services for wrapped data providing systems provides an answer to research question **five**.

WSMED itself is available as a general web service to process queries over other web services, known as the *WSMED web service*. It provides web service operations to handle user sessions, import WSDL documents for web services to query, user authentications for accessed web service operations,

inspecting the schema for the generated SQL views, and executing queries over the views. The WSMED web service is generated by the web service generator. The automatically generated WSDL document *wsmed.wsdl* [41] describes the interface of the WSMED web service operations. The functionality of WSMED is demonstrated through a publicly accessible web based demonstration [40]. A JavaScript program enables the user to query any data providing web service by calling the WSMED web service operations directly from a browser without downloading any software. This shows that the WSMED web service adheres to the XaaS paradigm and provides an answer to research question **six**.

The reminder of this Thesis is organized in the following way: Section two introduces the technical background on which the research work is based. Section three explains how the papers I-VIII contribute to answering the research questions. Finally, Section four concludes and indicates future directions.

2. Background

This chapter presents the technical background of the major enabling technologies for mediating and querying web services. It briefly covers database management systems and the core technologies involved with web services.

2.1. Database Management Systems

A software system that allows creating and manipulating huge amounts of data in a structured way is known as a *Database Management System (DBMS)* [14]. 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 specify its structures as a *database schema* through a *Data Definition Language (DDL)*.
- It permits the users to insert, delete, update and query data from a data base through a *Data Manipulation Language (DML)*.
- It provides a security system to support multilevel authentication control.
- It preserves the consistency of data through an integrity system.
- It provides transaction and recovery control to restore the database to a previous consistent state after hardware and software failures.

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. DBMSs use different kind of data models. The most common data model is the relational data model where data is represented as tables. Central in the relational data model is the provision of a *high level query language* for efficient database search using declarative queries. The most common relational query language is the *Structured Query Language (SQL)* [14]. SQL is used in this Thesis work for querying data providing web services rather than data stored in tables.

A relational *view* is virtual relation (i.e. table) defined through a query expression. A view is not physically stored in the database but can be queried as other relations. It is sometimes possible to modify views by an insertion, deletion, or update, so called *updatable views*. In this Thesis

relational views are defined that search data from data providing web service operations.

The *Entity-Relationship (ER) model* is a graphical data model for abstract representation of database schemas. During the database design process, the database schema is represented in the ER model and then converted to the data model of the DBMS, e.g. the relational model.

In a functional data model [34] data is represented using typed functions rather than tables. This Thesis work uses the functional DBMS Amos II [32] to internally represent web service meta-data and views over web service operations.

Query processing

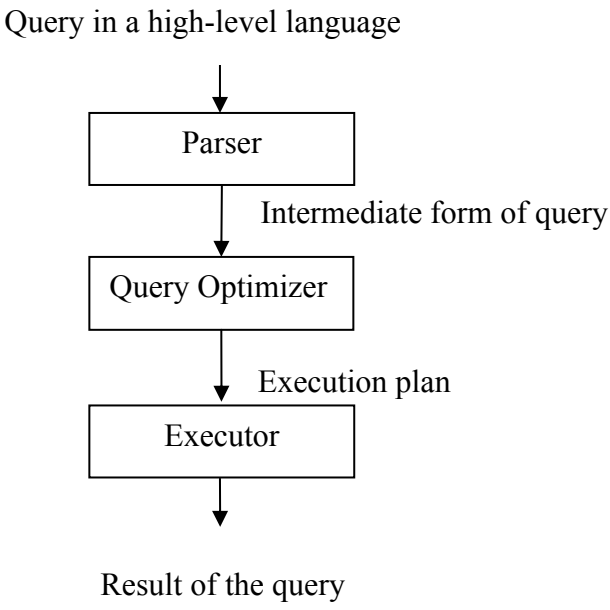


Figure 1 Query processor

Query processing (Figure 1) is the process of efficiently executing declarative queries over large databases. It transforms a declarative query into an *execution plan,* which is a program that specifies in details how the data is retrieved. The *query processor* is the group of components of a DBMS responsible for query processing. It has the following components:

- The *parser* ensures that the query syntax follows the grammar of the query language. It transforms the query into an internal intermediate form, usually a logical calculus expression.

- The *query optimizer* translates the parsed query into an execution plan, which is a program to retrieve data. The query execution plan is a 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 choosing an efficient plan is named *query optimization*, which is performed by the query optimizer. The traditional query optimization is based on cost-based optimization [17]. 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 optimization cost.
- The *executor* interprets the execution plan to produce the query result.

In this Thesis work query optimization techniques are developed for generating efficient execution plans that contain calls to web service operations.

Adaptive Query Processing

The traditional cost-based optimization strategies often expose limitations and have bad performance when the execution costs cannot be estimated precisely enough. In particular, it is not always possible to get the precise statistics about derived data collections. Furthermore, the statistics are sometimes unreliable due to dynamically changing data at runtime and work load characteristics. Therefore, adaptive query processing (AQP) techniques [13] have been developed for query optimization while the query is executing. AQP utilizes runtime feedback and modifies the query execution plan on the fly. To increase the opportunities of adaptation, special dynamic execution plan operators are introduced, such as Symmetric Hash Join [29] and Eddies [3].

In this Thesis work techniques are introduced for run time adaptive parallelization of execution plans that call expensive functions such as web service operation.

Distributed and Parallel databases

In distributed databases [30], data management is distributed over many processing nodes that are interconnected via a network. The data distribution is not visible to the end user. The database administrator provides data distribution hints to the distributed DBMS. Distributed DBMSs effectively manage distributed databases by query optimization and reliable data

management. Distributed query optimization is the process of generating an efficient execution plan for the processing of a query to a distributed database system. In this Thesis queries over distributed data providing web service operations are optimized.

Parallel DBMSs [30] is a kind of a distributed database system that runs on a cluster of processing nodes to achieve better performance through parallel execution of operators. In contrast to distributed database managements systems, data distribution is not visible to the database administrator in parallel DBMSs. Cost-based approaches, such as two-phase query optimization [19], is used in parallel database management systems to speed up queries. This Thesis work adaptively parallelizes queries calling distributed web service operations without any cost model.

2.2 Mediators

Mediators [39] are software modules used to query heterogeneous data sources. A mediator represents a virtual view or composition of views that integrate data from different data sources. Mediators don't store any data themselves and this contrasts mediation from the data warehouse [16] approach where all data is uploaded from data sources to a database. Instead, as shown in Figure 2, mediators make use of interfaces called *wrappers* to retrieve data dynamically from the data sources.

Views play a prominent role in mediation. 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. Further, the views must be able to join and convert conflicting and overlapping data from different data sources. The views are defined by means of a common data model (CDM).

The system interpreting the mediator modules is known as the *mediator engine*. The mediator engine interprets queries expressed in terms of the CDM. Performance and scalability over the amounts of data retrieved are important design aspects of mediator engines.

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 engine, it constructs the appropriate sub queries to send to the wrappers. A wrapper accepts queries from the mediator engine and translates them so they can be answered by the underlying data source. Then it returns back the result to the mediator engine. The mediator engine collects data from several wrapped data sources and post-processes them before sending back the result of the query to the user.

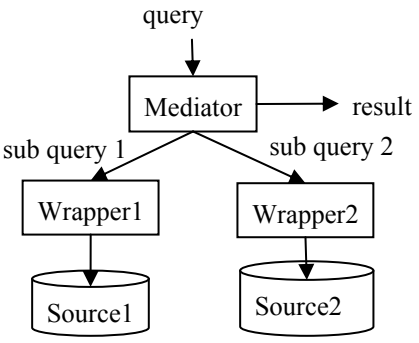


Figure 2 Mediation architecture

There are several systems such as Garlic [35], Information manifold [23], and TSIMMIS [15] using mediators for data integration from heterogeneous data sources.

This Thesis work extends the Amos II mediator engine [32] to process data from wrapped web service operations.

Capability based optimization in mediators

Wrapped data sources often limit certain attributes as inputs and produce values of other attributes as outputs, but have no general query capabilities. We say that such sources have *limited capabilities*. For example, web service operations can be seen as data sources with limited capabilities.

Capability-based query optimization [25] [43] is tailored to generate feasible plans accessing data sources with limited capabilities. 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 [35], Information Manifold [23], and TSIMMIS [24] to match the source capabilities. Source capabilities are represented as capability records [23] or by some special description language such as Relational Query Description Language (RQDL) [37]. 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* known as *binding patterns* [16]. Matching sources are selected by analyzing the binding patterns. For example, the web query optimization system [44] and Amos II [32] utilize binding patterns to represent source capabilities. Adornments are attached with each

attribute of a data source. It is represented by an alphabet with specific meaning:

I *f (free)* - the value of the attribute need not to be specified
II *b(bound)* - the value of the attribute must be specified
III *c[L] (choice from a list L)* - the value of the attribute must be specified from the values in the list L.
IV *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.

This Thesis work use binding patterns for defining capability limited view over web service operations.

Estimating cost metrics in the mediation environment is often difficult as the data sources are independent from the mediator. For example, with data accessible via web services the data retrieval time can vary 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 [20] and adaptive query processing strategies can alleviate this. This Thesis work uses adaptive parallelization to dynamically optimize queries calling web service without using cost metrics of web service operations.

2.3 Web Services

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 [18], WSDL [9], XML Schema [42], HTTP [21] and UDDI [6]. 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 used to transfer the SOAP message to invoke a web service and return the result back.

The layered web service architecture is illustrated in Figure 3. The *discovery* layer acts as a centralized repository of web services. By querying this repository one can find a required web service based on their

descriptions. The open standard technologies UDDI and WS-Inspection [5] is used at this layer for how to publish, categorize, and search for services based.

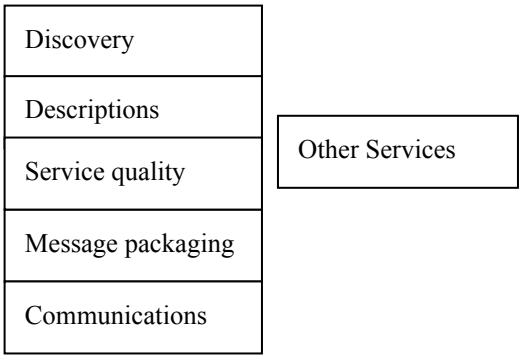


Figure 3 Web service architecture

The *descriptions* layer deals with how to represent service behavior, capabilities, and requirements in machine readable form. WSDL 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.

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 [26] and FTP [28].

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 [4] declares the service quality requirements and their capabilities to enable 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 [22]. The web service management task force [38] 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 various purposes such as composing services to create new applications. For example, BPEL4WS [2] provides a workflow oriented composition model well suited for business applications.

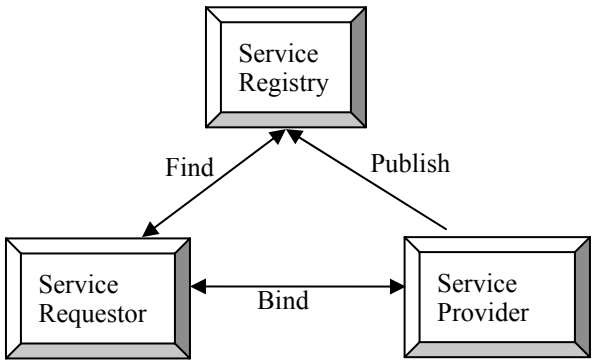


Figure 4 Service-oriented architecture

Figure 4 illustrates the interrelationship of SOAP, WSDL and UDDI in a service oriented environment. The *service provider* is responsible for generating and deploying a service. It publishes a service description using WSDL in a *service registry, UDDI*. The UDDI advertises the service and allows a *service requestor* to send queries to the registry to find a service either by name, category, identifier, or a 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 the WSDL document and sends it over the network to the service provider to use the service. The *bind* operation embodies the relationship between the service requestor and the service provider.

Web Services Description Language

The functional description of a web service is defined by the XML based Web Services Description Language (WSDL). A WSDL document 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 defines *services* as set of network endpoints, called *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* define abstract

descriptions of the data being exchanged. *Port types* are abstract collections of *operations*. An operation defines the description of an action supported by the service. A protocol such as SOAP, HTTP, and data type specifications for a particular port type represent a *binding* for a web service operation. A *port* is defined by associating a network address with a binding. XMLSchema is used to describe message formats. WSDL allows user defined type definitions known as *extensibility elements*.

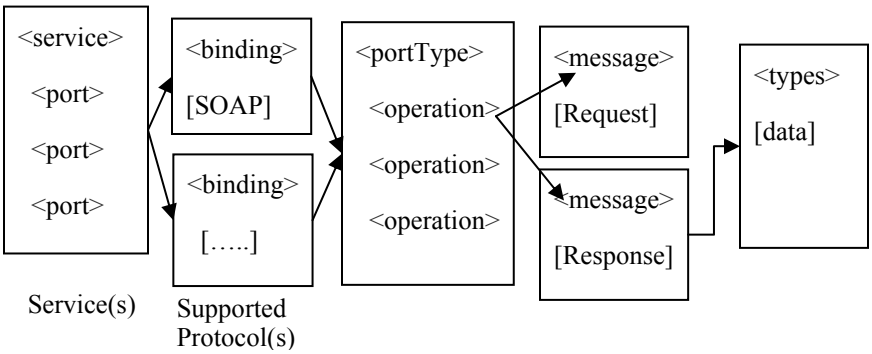


Figure 5 Document structure of WSDL

Figure 5 illustrates a simple WSDL document structure. Each service has several ports to define where it is located. In turn each port is attached to one or more bindings that describe how a web service is accessed. Each binding is attached to a *portType* having a set of operations to answer what a service is does. Request and response messages are associated with each operation to indicate the input and output of an operation.

In this Thesis work web service operations' meta-data are imported from the WSDL documents that describe the operations. Those meta-data are used to automatically define SQL views over web service operations.

SOAP

SOAP is an XML based lightweight, platform independent protocol for information exchange in a distributed environment. SOAP is used not only with HTTP but also used in combination with other protocols such as SMTP and TCP [27]. The simplicity and extensibility are the major design goals of SOAP.

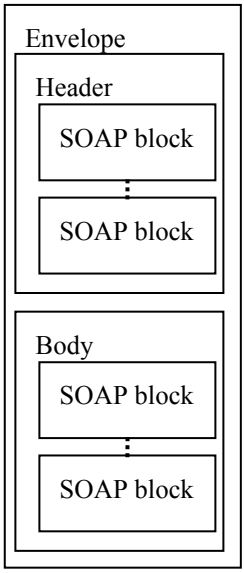


Figure 6 SOAP Message

A SOAP message (Figure 6) is made up of three elements:

1. The *SOAP Envelope* is a top element that encapsulates the other two elements representing the message.
2. The 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.
3. The *SOAP body* contains the actual message to be delivered and processed.

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.

In this Thesis work the query processor constructs SOAP calls to web service operations using the imported WSDL meta-data.

2.4 Active Mediators Object System (Amos II)

Our prototype system WSMED is based on the existing mediator engine Amos II [32]. Amos II has a functional data model as CDM. The functional query language, *AmosQL*, is the primary query language. Wrappers can be

defined to make heterogeneous data sources queryable. A wrapper performs [31] the following:

- *Schema importation* translates a sources' schema into a form compatible with the CDM of Amos II.
- *Query translation* converts AmosQL queries into API calls or query expressions executable by a source.
- *Statistics computation* estimates costs and selectivities for the calls to retrieve data from sources.
- *Proxy OID generation* constructs proxy object identifiers to describe the data from sources.

The basic concepts of the Amos II data model are *objects*, *types*, and *functions*. It is used as the CDM for the mediation and it is an extension of the Daplex [32] [34] functional data model.

Objects 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. 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 the notation *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. 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*. For example:

```
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 arguments values of a function if the expected result value is known. 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, similar to tables in a relational database.
- *Derived* functions are defined as queries in terms of other Amos II functions. They are side-effect free and they are precompiled and optimized as soon as they are defined. The queries are expressed in AmosQL, using has an SQL-like select statement for defining derived functions. Derived functions correspond to views in relational databases.
- *Foreign* functions enable low-level interfaces for wrapping external systems. For example, in this Thesis a general mechanism to call any web service operation is implemented as a foreign function named *cwo*.
- *Multi-directional functions* enable to associate several implementations of inverses for a given function. This defines functional views having different implementations depending on the actual binding pattern of its parameters. For example, a view over web services may be implemented using several web service operations as in Paper I where different operations are called depending on what parameters are known.

3. Summary of the Papers

This section summarizes how Paper I - VIII contribute to answering the research questions proposed. Paper I - IV are the main contributions.

3.1 Paper I

Paper I presents the overall architecture of WSMED and the general capabilities of WSMED for querying data accessible via web service operations. After the system has imported meta-data by reading WSDL documents for the operations to query, the user can manually define views that extract data from the results of web service operations calls. The views can be queried using SQL. In Paper I the views are manually specified as a set of declarative queries that access web service operations differently depending on what view attributes are known in a query. To enable semantic optimization of queries over the views based on automatic query transformations the user can specify key attributes of a view as a semantic enrichment. We evaluated the effectiveness of such enrichments over multi-level views of publicly available web service operations and showed that the key constraint enrichment substantially improves query performance. Paper I answers research question **one** and partially answers research questions **two, three,** and **four.** However, the optimization is based on semantic enrichments that have to be manually defined by the view definer.

3.2 Paper II

Paper II describes and evaluates strategies for adaptive parallelization of web service calls based on automatically generated SQL views of web service operations. Each generated view encapsulates a data providing web service operation for given parameters and emits the result as a flattened stream of tuples. SQL queries can be made over these views with the restriction that the input attributes must be known in the query. When joining such views it is often the case that in the execution plan the output of one web service call is the input for another, etc. The challenge addressed in Paper II is to develop methods to speed up such dependent calls by parallelization. Since web service calls incur high-latency and message set-up costs, a naïve

approach making the calls sequentially is time consuming and parallel invocations of the web service calls should improve the speed. Our approach automatically parallelizes the web service calls by starting separate query processes, each managing a plan function for different parameter values. For a given query, the query processes are automatically arranged in a multi-level process tree where plan functions are called in parallel. The parallel plan is defined in terms of an algebra operator, *First Finished Apply in Parallel* (FF_APPLYP), to ship in parallel to other query processes the same plan function for different parameters. By using FF_APPLYP we first investigated ways to set up different process trees manually. We concluded from our experiments that the best performing query execution plan is an almost balanced bushy tree. To automatically achieve the optimal process tree we modified FF_APPLYP to an operator *Adaptive First Finished Apply in Parallel* (AFF_APPLYP) that adapts the process tree locally in each query process until optimized performance is achieved. AFF_APPLYP starts with a binary process tree. During execution each query process in the tree makes local decisions to expand or shrink its process sub-tree by comparing the average time to process each incoming tuple. The query execution time obtained with AFF_APPLYP is shown to be close to the best time achieved by manually built query process trees. Paper II answered research questions **one** and **two** and partially answered research questions **three** and **four**.

3.3 Paper III

In general queries calling data providing web service operations may have both dependent and independent calls. Paper III generalizes the adaptive strategy presented in Paper II to handle both independent and dependent web service operation calls. The adaptive operator PAP speeds up queries with independent web service operation calls by calling in parallel the plan functions encapsulating each independent call. Dependent web service calls are handled by adaptive parallelization of sequences of PAP calls. This is shown to substantially improve the query performance without any cost knowledge or extensive memory usage compared to other strategies. Paper III answers the research questions **one, two, three,** and **four** by providing a generalized approach to query both dependent and independent data providing web service operations. The performance of PAP is evaluated using publicly available web services.

3.4 Paper IV

Paper IV describes the overall functionality of the WSMED system. This includes the WSMED query processor, the WSMED web service to query

any data providing web service operations, the web based demonstration of WSMED, and the web service generator.

The generation and deployment of web services for data providing systems answers research question **five**.

The web based demonstration of WSMED allows making SQL queries combining data from any data providing web services. This answers research question **six**.

3.5 Paper V

Paper V provides some preliminary work for Paper I. The WSMED architecture and a proposed method to manually define SQL views over web service operations are outlined.

3.6 Licentiate Thesis (Paper VI)

The Licentiate Thesis outlines some of the research questions, presents the technical background on which the research work is based, and proposes the WSMED architecture. Paper I and V are based on the Licentiate Thesis.

3.7 Paper VII

Paper VII describes the web based demonstration of WSMED that directly invokes WSMED web service operations from a web browser. This work is included and elaborated in Paper IV.

3.8 Book Chapter (Paper VIII)

The book chapter in Paper VIII is based on Paper I and II. It summarizes the WSMED architecture and the adaptive query processing strategies used.

4. Conclusions and Future Work

WSMED provides general database query capabilities over any data providing web service operations given their WSDL meta-data descriptions. For each data providing web service operation in a given WSDL document, WSMED automatically generates relational views by reading web service operations' WSDL descriptions. Such automatically generated relational views can be queried with SQL.

Without any cost knowledge the WSMED query processor automatically and adaptively finds an optimized parallel execution plan calling the queried data providing web service operations. The algebra operator *PAP* locally adapts the parallel plan until no significant performance improvement is measured, based on monitoring the flow from data providing web service operations. The operator handles queries where data providing web service operations are called both dependently and independently. A strategy using *PAP* is developed, which substantially improves the query performance without any cost knowledge or extensive memory usage compared to other strategies.

WSMED assumes that all queried data sources are available as web service operations. To make any data providing system into a web service WSMED includes a subsystem, the web service generator, which generates and deploys the web service operations to access a data source.

To comply with the XaaS paradigm WSMED itself is implemented as a web service that provides SQL query functionality to query and join any data providing web service operations. The WSMED web service is also generated by the web service generator. To enable search of any data providing web services from a browser without any need for installing software, the web based demonstration is written as a JavaScript program that directly calls the WSMED web service. In summary the contributions of the Thesis are:

1. The WSMED system architecture provides general SQL query capabilities over any data providing web services based on their WSDL documents.
2. To enable SQL queries to data providing web services, SQL views are automatically generated for any data providing web service operations by reading their WSDL documents.
3. To automatically parallelize queries to data providing web service, an algorithm is implemented to transform a non parallel plan into a parallel

plan by introducing the adaptive operator *PAP* that encapsulates plan functions calling data providing web service operations.

4. To automatically and adaptively optimize a parallel plan, the operator *PAP* adapts an initial parallel query process tree by locally monitoring result flows from each child query process until satisfactory performance is obtained. The adaptive query parallelization does not need any static cost model.
5. To generate data providing web service interfaces to any data providing system a web service generator automatically generates web service operations for wrapped data sources defined as interface functions. The generated web service operations are dynamically deployed without restarting a web server.
6. To comply with the XaaS paradigm, the WSMED web service is provided to query any data providing web services. It can be used directly from a browser without any software installations. The WSMED web service operations are generated by the web service generator.

All performance measurements were made with publicly available web service operations. A possible future work is to develop a benchmark to simulate the parallel web service calls for controlled experiments.

WSMED presently handle relational views that calls data providing web services operations without any side effects. Updatable relational views over web services is a subject for future work.

5. Summary in Swedish

Sökning bland datagenererande web services

Den kraftigt ökande tillgången till internetbaserade informationssystem har skapat ett behov att utveckla *web services* [7], dvs. system och standarder för att utbyta information mellan internetbaserade program. Medan s.k. *webbtjänster* gör det möjligt att utbyta information mellan människor och webbaserade program i vanliga webbläsare, tillhandahåller *web services* en infrastruktur för informationsutbyte mellan olika webbaserade program. För web services har man utvecklat ett antal standarder som SOAP[18], WSDL [9] och XML Schema [42]. Web services tillhandahåller verktyg för programutvecklare att definiera *operationer* (eng. operations) som är programmeringsgränssnitt för att anropa andra program via Internet. Dessa web service-operationer (WSO) är självbeskrivande i den meningen att information om hur de anropas och hur data som skall överföras skall se ut (s.k. meta-data) beskrivs för varje WSO m.h.a ett speciellt språk som heter *Web Service Description Language, WSDL*. WSDL-beskrivningarna läggs upp på Internet som maskinläsbara dokument. Genom att läsa WSDL-dokumentet för en web service har ett program all information som behövs för att kunna anropa de WSOer som beskrivs i dokumentet.

Web services används ofta för att hämta data från servrar som tillhandahåller information av olika slag. En *datagenerande WSO* returnerar datamängder för givna sökparametrar utan att ha sidoeffekter som ändrar data på servern. Sådana tjänster är en form av *sökbearbetning* (search computing) [8]. Andra typer av web services utför någon åtgärd, t.ex. gör en banktransaktion eller startar en maskin.

Ämnet för denna avhandling är att undersöka hur *frågespråk* kan göra det möjligt att effektivt söka bland olika datagenererande WSOer. Ett frågespråk är ett kraftfullt högnivåspråk för att söka bland data. T.ex. är frågespråket SQL standardspråk för sökning i konventionella databaser. I avhandlingen används SQL för att söka bland data från olika datagenererande WSOer i stället för från en konventionell databas. För att utföra motsvarande sökningar utan frågespråk programmerat i ett konventionellt programmeringsspråk måste man för varje fråga utveckla ett specialiserat program som implementerar en detaljerad strategi för hur sökningen bland datagenererande WSOer skall gå till.

Som ett exempel, antag att vi vill ställa en fråga som returnerar information om namngivna platser i några av USAs delstater, t.ex. deras postnummer och väderprognoser. Fyra olika datagenererande WSOer kan användas för att besvara frågan. Först kan operationen *GetAllStates* från web servicen *GeoPlaces* [10] anropas för att finna allmän information om delstater i USA. Sedan kan operationen *GetInfoByState* från web servicen *USZip* [36] anropas för att finna alla postnummer i en given delstat. Operationen *GetPlacesInside* från *Zipcodes* [11] returnerar alla platser inom ett postnummerområde. Slutligen kan operationen *GetCityForecastByZip* från *CYDNE* [12] anropas för att få väderprognosen för ett givet postnummer.

Ytterligare teknik som används i avhandlingsarbetet är mediatortekniken [39]. En mediator är ett system för att utföra frågor som kombinerar data från många olika datakällor. I detta arbete avses med en mediator ett system som gör det möjligt att m.h.a. ett frågespråk specificera frågor som kombinerar data från olika datagenererande WSOer.

I avhandlingen undersöks hur man kan bygga ett generellt system för skalbara frågor över datagenererande WSOer. Ansatsen är att utveckla ett prototypsystem med benämningen *WSMED (Web Service MEDiator)* för att ge svar på ett antal forskningshypoteser:

1. I vilken utsträckning kan standarder för web services som WSDL och SOAP utnyttjas av en web service mediator för att effektivt och skalbart utföra frågor till datagenererande WSOer?
2. Hur kan man, baserat på WSDL-beskrivningar automatiskt generera vyer över datagenererande WSOer för ett högnivåfrågespråk som SQL?
3. Hur kan optimerings- och transformationstekniker för databasfrågor användas för att tillhandahålla effektiv och skalbar sökning bland data från olika datagenererande WSOer?
4. Hur kan en frågeoptimerare snabba upp sökning från datagenererande WSOer utan att innehålla kunskap om hur kostsamma operationerna är?
5. Hur kan datakällor som inte är tillgängliga som web services på ett enkelt sätt transformeras till datagenererande WSOer för att göra det möjligt att ställa frågor till dem från en web service mediator?
6. Hur kan paradigmen ”*allt som en service*” (XaaS) [33] tillämpas för att ställa frågor mot datagenererande WSOer? Det vill säga, kan en web service mediator implementeras i form av en web service som anropas från en godtycklig webbläsare utan att kräva att användaren först installerar speciell programvara i sin dator?

För att besvara ovanstående forskningsfrågor har WSMED-prototypen utvecklats och utvärderats och har nu förmågan att skalbart utföra frågor över datagenererande WSOer.

WSMED kan dynamiskt anropa en godtycklig WSO genom att läsa dess WSDL-dokument. WSDL-dokumenten lagras i WSMED i en generell *web service databas* som kan representera beskrivningar av godtyckliga WSDL-dokument. Databasen används för att dynamiskt konstruera anrop till de WSOer som behövs för att utföra en fråga. Detta ger svar på forskningsfråga **ett**.

En WSO presenteras av WSMED som en tabell (vy) i SQL. SQL frågor kan ställas över dessa vyer. För en given web service genererar WSMED automatiskt SQL vyer för alla dess WSOer genom att läsa WSDL dokumentet. SQL vyn för en WSO definieras i termer av ett internt frågespråk som heter *WQL (WSMED Query Language)* och kan hantera de datatyper som behövs för att anropa WSOer. Den automatiska genereringen av SQL-vyer besvarar forskningsfråga **två**.

WSOer är normalt parametriserade i den meningen att de kräver att in-parametrar har kända värden för att de skall kunna anropas. Två WSO-anrop i en fråga är *beroende* om det ena kräver in-parametrar som produceras i resultatet av ett annat WSO-anrop, i annat fall är de *oberoende*. I exemplet ovan är *GetPlacesInside* and *GetCityForecastByZip* WSO-anrop som beror på *GetInfoByState* men som är oberoende av varandra. En utmaning är här att utveckla metoder att automatiskt optimera frågor som innehåller både beroende och oberoende WSO-anrop. Generellt är sådan optimering beroende av olika egenskaper hos WSO-anropen. Dessa egenskaper är i allmänhet inte tillgängliga och beror på olika nätverks- och datoregenskaper när och var frågorna körs. I sådana fall är det mycket svårt att basera optimeringen på en statisk kostnadsmodell av de olika ingående kostnaderna, vilket är den teknik för frågeoptimering som tillämpas i traditionella databaser.

För att optimera frågorna utan en kostnadsmodell av underliggande WSOer använder WSMED en ansats där WSO-anropen dynamiskt parallelliseras vid frågetillfället med hänsyn tagen till beroenden mellan olika WSO-anrop i en fråga. Ofta förbättras prestanda dramatiskt genom att systemet ser till att WSOer anropas parallellt i stället för att anropa dem efter varandra. WSMED genererar automatiskt parallella sökprogram, *exekveringsplaner*, som anropas i ett träd av kommunicerande processer, ett *processträd*, där olika exekveringsplaner anropas parallellt. Under körning optimeras och ändras processträdet dynamiskt genom att systemet mäter tiden att utföra delplaner utan kännedom om kostnaden att anropa underliggande WSOer. I avhandlingen visas att denna dynamiska frågeoptimering ger stora prestandaförbättringar och detta resultat besvarar forskningsfrågorna **tre** och **fyra**.

WSMED antar att de datakällor som anropas är definierade som WSOer. Att skapa en ny datagenerarande web service för en datakälla kräver normalt en del programmeringsarbete, t.ex. för att implementera WSOer, definiera WSDL-dokument och att driftsätta web servicen på nätet. För att på ett

enkelt sätt göra ett dataproducerande system tillgängligt som datagenererande WSOer innehåller WSMED en *web service-generator* som skapar och driftsätter WSOer. Programmeraren måste först definiera ett gränssnitt mot datakällan i mediatorsystemet Amos II [32]. Därefter generar systemet automatiskt motsvarande WSOer och gör dem omedelbart tillgängliga på nätet. Samtidigt genererar system ett WSDL-dokument som beskriver genererade WSOer. Denna automatiska generering och driftsättning av WSOer ger ett svar på forskningsfråga **fem**.

WSMED-systemet självt är tillgängligt som en web service som kan utföra frågor till andra datagenererande web services. Denna *WSMED web servic*e innehåller WSOer för att sätta upp sessioner, importera WSDL-dokument för de web services som man vill söka i, inspektera de SQL-vyer som generats, ställa frågor mot SQL-vyerna och autentisera användaren. WSMED web servicen har genererats automatiskt m.h.a. web service-generatorn. WSMEDs funktionalitet demonstreras genom ett webbaserat användargränssnitt som är tillgängligt från en godtycklig webbläsare. Ingen programvara behöver då installeras eftersom gränssnittet är implementerat som ett JavaScript-program som exekveras i webbläsaren och direkt anropar WSMED web servicen. Detta visar att WSMED uppfyller XaaS paradigmen vilket besvarar forskningsfråga **sex**.

6. 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 papers and 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 Professor G.N.Wikramanayake is supporting me by his constructive advices and guidance and I appreciate his assistance. Dr.S.Mahesan and Dr.S.Kanaganathan are my first Computer Science teachers and emboldened me as a research student in Computer Science. I would like to thank them for their rewarding guidance and assistance.

I also wish to thank all Sri Lankan Sida split PhD program management committee members and Sida coordinator for Uppsala University for their great support all the time.

I offer my sincere gratitude to the administrative authorities of Department of Computer Science and Faculty of Science, University of Jaffna for their enormous support.

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 to thank all my fellow Sri Lankans for their friendship and support.

Ulrika Andersson and all the others at the Department of Information Technology, Uppsala University who have helped me immensely need mentioning.

I'm grateful to my wife, Sutha and my daughters Sruthy and Sharana for their generous support and patience.

I have great pleasure to dedicate this Thesis to my parents, Manivasakan and Saroginidevi, who have always encouraged and supported me to study.

This work was supported by the Swedish International Development and Cooperation (Sida), and the Swedish Foundation for Strategic Research under contract RIT08-0041.

Bibliography

[1] AmosII wrappers, http://user.it.uu.se/~udbl/amos/wrappers.html

[2] T.Andrews et al., Business Process Execution Language for Web Services, Version 1.1, http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-bpel/ws-bpel.pdf, 2003

[3] R.Avnur and J.M.Hellerstein, Eddies: Continuously Adaptive Query Processing, *Proc. 2000 ACM SIGMOD International Conference on Management of Data*, pp 261-272, 2000

[4] S.Bajaj et al., Web Services Policy Framework (WSPolicy), *http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-polfram/ws-policy-2006-03-01.pdf*, 2006

[5] K.Ballinger, P.Brittenham, A.Malhotra, W.A. Nagy, and S.Pharies, Web Services Inspection Language (WS-Inspection), *ftp://www6.software.ibm.com/software/developer/library/ws-wsilspec.pdf*, 2001

[6] T.Bellwood et al, UDDI Version 3.0.2, UDDI Spec Technical Committee Draft, *http://uddi.org/pubs/uddi_v3.htm#_Toc85907967*, 2004

[7] D.Booth, H.Haas, F.McCabe, E.Newcomer, M.Champion, C.Ferris, and D.Orchard, Web Services Architecture,W3C Working Group Note, *http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/* , 2004

[8] S.Ceri, Search Computing. *Proc. International Conference on Data Engineering*, IEEE Computer Society, pp. 1- 3, 2009

[9] E.Christensen, F.Curbera, G.Meredith, and S. Weerawarana, Web services description language (WSDL) 1.1., W3C Recommendation, *http://www.w3.org/TR/wsdl*, 2001

[10] codeBump, GeoPlaces web service *http://codebump.com/services /PlaceLookup.asmx*

[11] codeBump, Zipcodes web service *http://codebump.com/services /ZipCodeLookup.asmx*

[12] CYDNE, *http://ws.cdyne.com/WeatherWS/Weather.asmx?WSDL*

[13] A.Deshpande, Z.G:Ives and V.Raman, Adaptive Query Processing, *Foundations and Trends in Databases*, 2007

[14] R.Elamasri, and S.M.Navathe, *Fundamentals of Database Systems*, 4th Edition, ISBN 0-321-20448-4, Pearson Education, pp 855-856, 2004

[15] 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, *Journal of Intelligent Information Systems*, 8(2), pp 117-132, 1997

[16] H.Garcia-Molina, J.D Ullman, and J.Widom, *Database Systems: The Complete Book,* ISBN 0-13-098043-9, Prentice Hall, pp 1047-1069, 2002

[17] G.Graefe, Query evaluation techniques for large databases**,** *ACM Computing Surveys (CSUR)***,** 25(2), pp 73-169, 1993

[18] M.Gudgin, M.Hadley, N.Mendelsohn, J.Moreau, and H.Frystyk Nielsen, SOAP Version 1.2 Part 1: Messaging Framework,W3C Recommendation, *http://www.w3.org/TR/soap12-part1/* ,2003
[19] W. Hasan, *Optimization of SQL queries for Parallel Machines*, Springer-Verlag,1997
[20] 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
[21] Hypertext Transfer Protocol, W3C Architecture domain, *http://www.w3.org/Protocols/*
[22] K.Lawrence, C.Kaler, A.Nadalin, M.Gudgin, A.Barbir, and H.Granqvist, WS-SecurityPolicy v1.0, OASIS Working Draft, *http://www.oasis-open.org/committees/download.php/15979/oasis-wssx-ws-securitypolicy-1.0.pdf*, 2005
[23] A.Y.Levy et al., Querying Heterogeneous Information Sources Using Source Descriptions, *Proc. of 22nd Very Large Data Bases Conference(VLDB 96)*, pp 251-262 , 1996
[24] C.Li et al., Capability Based Mediation in TSIMMIS, *Proc. 1998 ACM SIGMOD International Conference on Management of Data,* 1998, pp 564-566
[25] Y.Papakonstantinou, A.Gupta, and L.Haas, Capabilities-base query rewriting in mediator systems, *Proc. Conference on Parallel and Distributed Information Systems*, pp 170-183, 1996
[26] J.Postel, SIMPLE MAIL TRANSFER PROTOCOL, RFC 821, *http://www.ietf.org/rfc/rfc0821.txt*, 1982
[27] J.Postel, Transmission Control Protocol, *http://www.ietf.org/rfc/rfc793.txt,* 1981
[28] J. Postel, and J. Reynolds, FILE TRANSFER PROTOCOL (FTP), *http://tools.ietf.org/html/rfc959*, 1985
[29] L.Raschid and S.Y.W.Su, A Parallel Processing Strategy for Evaluating Recursive Queries, *Proc. 12th Very Large Data Bases Conference(VLDB '86)*, pp 412-419, 1986
[30] T.Risch, Distributed Architecture, in L.Liu and M.Tamer Özsu (eds.): *Encyclopedia of Database Systems*, 2(1), Springer, pp 875-879, 2009
[31] 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, pp 933-953, 2001
[32] 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,* Springer, pp 211-238, 2003
[33] S.Robison, The Next Wave: Everything as a Service, *http://www.hp.com/hpinfo/execteam/articles/robison/08eaas.html*
[34] D. Shipman, The Functional Data Model and the Data Language DAPLEX, *ACM Transactions on Database Systems*, 6(1), pp 140-173, 1981
[35] 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)*, pp 266-275, 1997
[36] USZip, *http://www.webservicex.net/uszip.asmx*
[37] V.Vassalos, and Y.Papakonstantinou, Describing and Using Query Capabilities of Heterogeneous Sources, *Proc. 23rd Very Large Data Bases Conference(VLDB 97)*, pp 256-265*,* 1997
[38] Web Services Management Work by the Web Services Architecture Working Group, *http://www.w3.org/2002/ws/arch/4/management/*

[39] G. Wiederhold, Mediators in the Architecture of Future Information Systems, *IEEE Computer*, 25(3), pp 38-49, 1992
[40] WSMED Demo, *http://udbl2.it.uu.se/WSMED/wsmed.html*
[41] WSMED WSDL, *http://udbl2.it.uu.se/WSMED/wsmed.wsdl*
[42] XML Schema, *http://www.w3.org/standards/xml/schema*
[43] R.Yerneni, C.Li, H.Garcia-Molina, and J.D:Ullman, Computing capabilities of mediators, *Proc. 1999 ACM SIGMOD International Conference on Management of Data*, pp 443-454, 1999
[44] V.Zadorozhny, L.Raschid, M.E.Vidal, T.Urban, and L.Bright, Efficient Evaluation of Queries in a Mediator for WebSources, *Proc. 2002 ACM SIGMOD International Conference on Management of Data,* pp 85-96, 2002

Acta Universitatis Upsaliensis

Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 755

Editor: The Dean of the Faculty of Science and Technology

A doctoral dissertation from the Faculty of Science and Technology, Uppsala University, is usually a summary of a number of papers. A few copies of the complete dissertation are kept at major Swedish research libraries, while the summary alone is distributed internationally through the series Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology. (Prior to January, 2005, the series was published under the title "Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology".)

Distribution: publications.uu.se
urn:nbn:se:uu:diva-128928

ACTA
UNIVERSITATIS
UPSALIENSIS
UPPSALA
2010

Paper I

The paper is reformatted for typographic consistency.

Web Service Mediation Through Multi-level Views

Manivasakan Sabesan and Tore Risch

Department of Information Technology, Uppsala University, Sweden
{msabesan, Tore.Risch}@it.uu.se

Abstract. The web Service MEDiator system (WSMED) provides general query capabilities over data accessible through web services by reading WSDL meta-data descriptions. Based on imported meta-data, the user can define views that extract data from the results of calls to web service operations. The views can be queried using SQL. The views are specified in terms of declarative queries that access different web service operations in different ways depending on what view attributes are known in a query. To enable efficient query execution over the views by automatic query transformations the user can provide semantic enrichments of the meta-data with key constraints. 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.

Keywords: web service views, query optimization, semantic enrichment

1. Introduction

Web services [4] provide an infrastructure for web applications by defining sets of operations that can be invoked over the web. Web service operations are described by meta-data descriptions of operation signatures, using the *Web Services Description Language* (WSDL) [5]. An important class of operations is to access data through web services, e.g. Google's web page search service [12] and the United States Department of Agriculture nutrition database of foods [27]. However, web services don't support general query or view capabilities; they define only operation signatures.

We have developed a system, WSMED – Web Service MEDiator, to facilitate efficient queries over web services. The 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 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 and flatten relevant data from a web service call is defined by the user as queries called *capability definitions* using and object-oriented query language, *WSMED query language* (WQL), which has support for web service data types.

An important semantic enrichment is to allow for the user to associate with a given WSMED view different capability definitions depending on what view attributes are known in a query, the *binding pattern* of the capability definition. The WSMED query optimizer automatically selects the optimal capability definition for a given query by analyzing its used binding patterns. These view definitions enrich the basic web service operations to support SQL data access queries.

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, called *key constraints*.

This paper is organized as follows: Section two describes the architecture of WSMED. Section three gives examples of WSMED view definitions using an existing web service and explains the capability definitions. Section four analyzes the performance of a sample query to verify the effectiveness of query transformations based on the semantic enrichments compared to conventional relational algebra transformations. Section five describes the strategies of the query processor. Section six discusses related work. Finally section seven summarizes the results and indicates future work.

2. The WSMED System

Figure 1a, illustrates WSMED's system components. Imported WSDL meta-data is stored in the *web service meta-database* using a generic *web service schema* that can represent any WSDL definition. The *WSDL Importer* populates the web service meta-database, given the URL of a WSDL document. It reads the WSDL document using the WSDL parser toolkits *WSDL4J* [24] and *Castor* [23]. The retrieved WSDL document is parsed and automatically converted into the format used by the web service meta-database. In addition to the general web service meta-database, WSMED also keeps additional user-provided *WSMED enrichments* in its local store.

The *query processor* exploits the web service descriptions and WSMED enrichments to process queries. The query processor calls the *web service manager* which invokes web service calls using *Simple Object Access Protocol* (SOAP) [13] through the toolkit *SAAJ* [19] to retrieve the result for the user query.

Figure 1b illustrates architectural details of the query processor. The *calculus generator* produces from an SQL query an internal calculus expression in a Datalog dialect [18]. This expression is passed to the *query rewriter* for further processing to produce an equivalent but simpler and more efficient calculus expression.

The query rewriter calls the *view processor* to translate SQL query fragments over the WSMED view into relevant capability 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 will show that such rewrites significantly improve the performance of queries to multi-level views of web services.

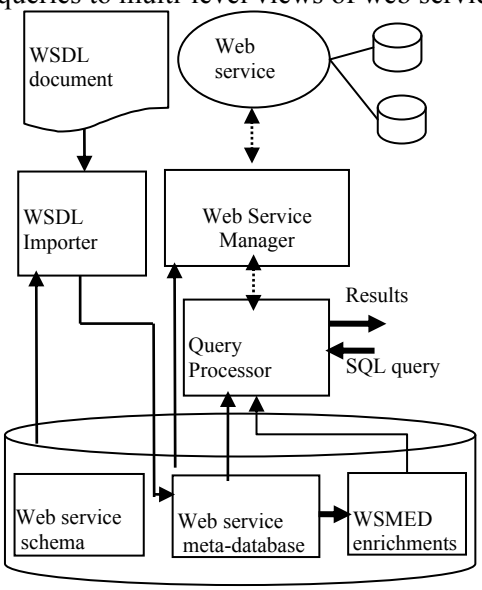


Figure 1a: WSMED components

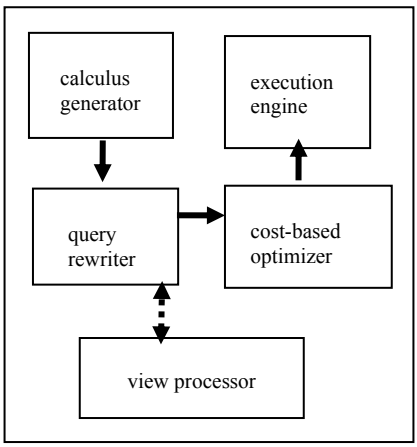


Figure 1b: Query Processor

The rewritten query is finally 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 generate a SOAP message when a web service operation is called.

3. WSMED Views

To illustrate and evaluate our approach we use a publicly available web service to access and search the National Nutrient Database for US Department of Agriculture [28]. 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 named *food* in Table 1 allows SQL queries over this web service operation.

Table 1. WSMED view *food*

ndb	keyword	descr	gpcode
19080	Sweet	Candies	1900
……	……	……	……

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 descr
from   food
where  gpcode='1900' and keyword ='Sweet';
```

The view *food* is defined as follows:

```
create SQLview food (Charstring ndb,
   Charstring keyword,Charstring descr, Charstring gpcode)
as multidirectional
  ("ffff" select ndb, "",descr, gpcode
          where foodDescr("","")= <ndb,descr,gpcode>)
  ("fffb" select ndb, "",descr
          where foodDescr("",gpcode)= <ndb,descr,gpcode>)
  ("fbff" select ndb,descr,gpcode
          where foodDescr(keyword, "")= <ndb,descr,gpcode>)
  ("fbfb" select ndb, descr
           where foodDescr(keyword,gpcode)
                 = <ndb,descr,gpcode>)
```

Figure 2: 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 declarative queries, called capability definitions, using an object oriented query language called WQL having special web service oriented data types. Each capability definition implements a different way of retrieving data through web service operations using WQL. Different capability definitions can be defined based on what view attributes are known or unknown in a query, called the capability binding patterns. The

query optimizer automatically chooses the most promising capability definitions for a given query to a WSMED view. Each capability definition provides a different way of using the web service operations to retrieve food items. The capability binding patterns of the view *food* are:

1. *ffff*- all the attributes of the view are free in the query. That is, the query does not specify any attribute selection value. In this case the capability definition specifies that all food items should be returned.
2. *fffb*- a value is specified only for fourth attribute *gpcode*. This means that the capability definition returns all food items for a given food group code.
3. *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.
4. *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 capability definitions are defined as declarative WQL queries that all call a function *foodDescr* in different ways. The function *foodDescr* is defined as a WQL query 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. 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 attributes are specified as a set of attribute names for a given view *(e.g. {"ndb"})*. Several keys can be specified by several calls to *declare_key*.

The query optimizer may also need to estimate the cost to invoke a capability and the estimated 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.

3.1 Capability definition function

The function *foodDescr,* used in the capability definitions in Figure 2, has the following definition:

```
1.create function foodDescr (Charstring fkw,
2.                           Charstring fgc)
3.          ->Bag of <Charstring ndb,Charstring descr,
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 generic 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 query variable *out* (line 8). It holds the output from the web service operation temporarily stored in WSMED's local database. The system automatically converts the input and output messages from the operation into records and sequences where records are used to represent complex XML elements [7] and sequences represent ordered elements. In our example, the argument list holds the parameters *Food-Keywords* 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 function *foodDescr* selects relevant parts of the result from the call to 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. Our example web service operation *SearchFoodByDescription* returns descriptions of all available food items when both attributes *foodkeywords* and *foodgroupcode* are empty strings. 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 capability definition function in Figure 2.

4. Impact of key constraints

To illustrate the impact of key constraints we define two 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, descr)
    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.descr
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 1b) into a Datalog expression:

```
Query(1) :- foodclasses(ndb,keyword,gpcode) AND
fooddescriptions (ndb,descr) AND descr=1 AND gpcode='1900'
```

The definitions of the views *foodclasses* and *fooddescriptions* are defined in Datalog as[1]:

```
foodclasses(ndb, keyword, gpcode) :- food(ndb, keyword, *,
            gpcode).
fooddescriptions(ndb,descry) :- food(ndb, *, descr, *).
```

Given these view definitions the Datalog expression is transformed by the view processor (Figure 1b) into:

```
Query(1) :- food(ndb,*,*,'1900') AND food(ndb,*,1,*).
```

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* [9]. If a key constraint that *ndb* is the key is specified, this is used by a query rewriter to combine the two calls to *food*:

```
Query(1) :- food(*,*,1,'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

[1] '*' means don't care.

expanded query. The simplification is very important to attain a scalable query execution performance as shown in Section 5.

The next step is to select the best capability definition for the query. The heuristics is that if more than one capability 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. The call to *food* then becomes:

```
Query(l) :- l=foodDescr("","1900").
```

Similar to relational database optimizers, given the definition of *foodDescr*, a cost based optimizer generates the algebra expression in Figure 3a, 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 [14]. By contrast, Figure 3b 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 capability definitions. An alternative possible better plan based on hash join (HJ) that materializes the inner web service call is shown in Section 5. In case no costing data is available about the capability 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 3a, which is optimal for our query.

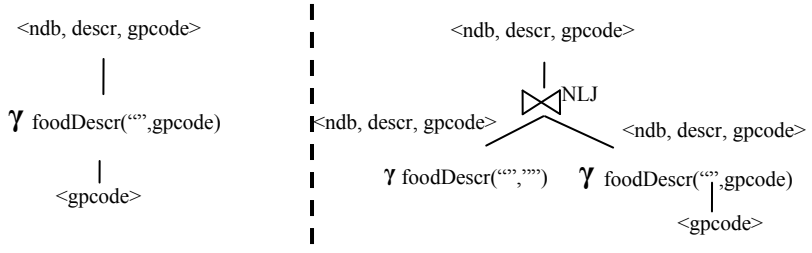


Figure 3a: Full semantic enrichment

Figure 3b: Naïve execution

5. Query Performance

To determine the impact of semantic enrichments on query processing strategies, 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 capability 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 3b 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 capability definition

where more variables are bound. Still there is no key specified. Figure 5b illustrates the plan using nested loop join.

3. Figure 5a 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 3a, shows the execution plan. It is clearly optimal.

As shown in Figure 4a, 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 4b 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 enrichment is critical for high performing queries over multi-level views of web services.

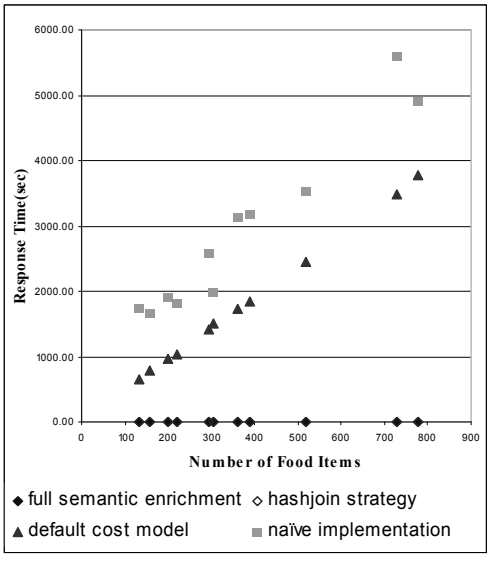


Figure 4a: Performance comparison of four query execution strategies

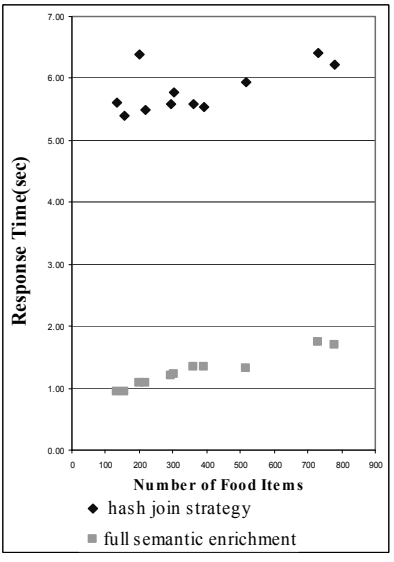


Figure 4b: Performance comparison of hash join and full semantic enrichment execution strategies

The diagrams are based on the experimental results in Table 2 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 *gpcode*, the number of food items (actual fanout), and the execution time in seconds for the four different 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 a large costly result containing all food items in the outer loop. This is clearly very slow.

Table 2. Experimental results

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

With the default cost model strategy the system assumes that queries over the view *food* produce larger results when the attribute *gpcode* is unknown than when it is known. Based on this the call to *foodDescr* with a known *gpcode* value is placed in the outer loop of a nested loop join. This clearly is a better strategy than the naïve implementation.

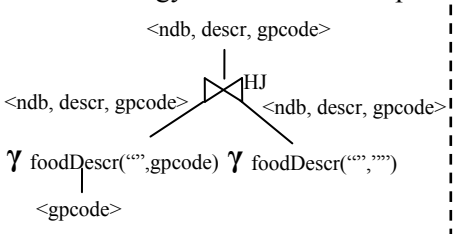


Figure 5a: Execution plan of hash join strategy

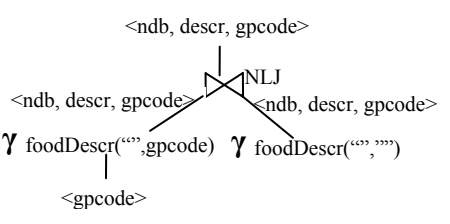


Figure 5b: Execution plan with default cost model

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.

6. Related Work

Preliminary results for our method of querying mediated web services were reported in [20].

SOAP [12] and WSDL [5] provide standardized basic interoperation protocols for web services but no query or view capabilities. The SQL 2003 standard [8][26] has facilitates to combine SQL with *XML Query language* (XQuery) [3] to access both ordinary SQL-data and XML documents stored in a relational database. By contrast, we optimize SQL queries to views over data returned by invoking web services and we use semantic query transformations to improve the performance.

The formal basis for using views to query heterogeneous data sources is reviewed in [10][15][25]. As some other information integration approaches, e.g. [11][29], we also use binding patterns as one of our semantic enrichments to access data sources with limited query capabilities. We define semantically enriched declarative views extracting data from the results of each web service operations in terms of an object-oriented query language. In [1] an approach is described for optimizing web service compositions by procedurally traversing ActiveXML documents to select embedded web service calls, without providing view capabilities.

WSMS [22] also provide queries to mediated web services. However, they concentrate on optimizing pipelined execution of web service queries while we utilize semantic enrichments for efficient query processing over multi-level views of web services. XLive [6] is a mediator for integrating heterogeneous sources including web service sources with specific wrappers based on XML standards. In contrast we deploy a generic wrapper that can call any web service.

In particular, unlike the other works, we show that key constraints significantly improve performance of queries to multi-level views of web services with different capabilities.

7. Conclusions and future work

We devised a general approach to query data accessible through web services by defining relational views of data extracted from the result SOAP messages returned by web service operations. Multi-level relational views of web service operations can be defined. The system allows SQL queries over these WSMED views. The view extractions are defined in terms of an object oriented query language. The query performance is heavily influenced by knowledge about the semantics of the specific web service operations invoked and all such information is not provided by standard web service descriptions. Therefore the user can complement a WSMED view with semantic enrichments for better query performance. Our experiments

showed that *binding patterns* combined with *key constraints* are essential for scalable performance when other views are defined in terms of WSMED views.

Strategies for parallel pipelined execution strategies of web service operation calls as in WSMS [22] should be investigated. The pruning of superfluous web service operation calls is crucial for performance. The adaptive approaches in [2][17] should be investigated where useless results are dynamically pruned in the early stage of query execution. Currently the semantic enrichments are added manually. Future work could investigate when it is possible to automate this and how to efficiently verify that 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 [16] based on monitoring the behavior of web service calls.

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 [21]. It should be investigated how mediate of web services based on such semantic web representations.

Acknowledgements

This work is supported by Sida.

References

[1] S. Abiteboul et al., Lazy query evaluation for active XML, *Proc. of the 2004 ACM SIGMOD Intl. Conf. on Managementof Data*, 227–238, 2004.

[2] R. Avnur, and J. M. Hellerstein, Eddies: Continuously adaptive query processing, *Proc. SIGMOD conference*, 2000.

[3] 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

[4] 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

[5] E.Christensen, F.Curbera, G.Meredith, and S. Weerawarana, *Web services description language (WSDL) 1.1*., W3C, http://www.w3.org/TR/wsdl, 2001.

[6] T.Dang Ngoc, C.Jamard, and N.Travers , XLive : An XML Light Integration Virtual Engine*, Proc. of BDA*, 2005

[7] 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

[8] A.Eisenberg, and J.Melton, SQL/XML is Making Good Progress, *ACM SIGMOD Record*, 31(2), June 2002

[9] G. Fahl, and T. Risch, Query Processing over Object Views of Relational Data, *The VLDB Journal* , 6(4), 261-281, 1997.

[10] 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

[11] H.Garcia-Molina, J.D Ullman, and J.Widom, *Database Systems: The Complete Book,* ISBN 0-13-098043-9, Prentice Hall, 1047-1069, 2002.

[12] Google SOAP Search API (Beta), *published online at http://code.google.com/apis /soapsearch/*

[13] 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

[14] L.M.Haas, D. Kossmann, E. Wimmers, and J .Yang, Optimizing queries across diverse data sources, *Proc. Very Large Database Conference(23rd VLDB),* 1997

[15] A.L.Halevy, Answering queries using views: A survey, *VLDB Journal,* 4(10), 270-294, 2001.

[16] 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), 812-853, 2005.

[17] Z.G.Ives, A.Y.Halvey, and D.S.Weld, Adapting to Source Properties in Processing Data Integration Queries, *Proc. SIGMOD conference*, 2004

[18] W. Litwin, and T. Risch, Main Memory Oriented Optimization of OO Queries using Typed Datalog with Foreign Predicates, *Proc. IEEE Transactions on Knowledge and Data Engineering,* 4(6), pp. 517-528, 1992

[19] SAAJ Project, *published online at https://saaj.dev.java.net/*

[20] M.Sabesan, T.Risch, and G.Wikramanayake, Querying Mediated Web Services**,** *Proc. 8th International Information Technology Conference (IITC 2006*), 2006

[21] Semantic Web Activity, W3C Technology and Society domain, *published online at http://www.w3.org/2001/sw/*

[22] U.Srivastava, J.Widom, K.Munagala, and R.Motwani, Query Optimization over Web Services, *Proc Very Large Database Conference(VLDB 2006),* 2006

[23] The Castor Project, *published online at http://www.castor.org/index.html*

[24] The Web Services Description Language for Java Tool kit(WSDL4J), *published online http://sourceforge.net/projects/wsdl4j*

[25] J.D.Ullman, Information Integration Using Logical Views, *Proc. 6th International Conference on Database Theory (ICDT '97)*, 19-40, 1997.

[26] XML-Related specifications (SQL/XML), *published online at http://www.sqlx.org/SQL-XML-documents/5FCD-14-XML-2004-07.pdf*, 2005

[27] Web Service USDAData, *published online http://ws.strikeiron.com/USDAData? DOC&page=proxy*

[28] WSDL document for USDAData web service, *published online http://ws.strikeiron.com/ USDAData?WSDL*

[29] 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.

Paper II

The paper is reformatted for typographic consistency.

Adaptive Parallelization of Queries over Dependent Web Service Calls

Manivasakan Sabesan and Tore Risch

Department of Information Technology, Uppsala University
Sweden
msabesan@it.uu.se
Tore.Risch@it.uu.se

Abstract— We have developed a system to process database queries over composed data providing web services. The queries are transformed into execution plans containing an operator that invokes any web service for given arguments. A common pattern in these query execution plans is that the output of one web service call is the input for another, etc. The challenge addressed in this paper is to develop methods to speed up such *dependent* calls in queries by parallelization. Since web service calls incur high-latency and message set-up costs, a naïve approach making the calls sequentially is time consuming and parallel invocations of the web service calls should improve the speed. Our approach automatically parallelizes the web service calls by starting separate *query processes,* each managing a parameterized sub-query, a *plan function*, for different parameter tuples. For a given query, the query processes are automatically arranged in a multi-level process tree where plan functions are called in parallel. The parallel plan is defined in terms of an algebra operator, *FF_APPLYP,* to ship in parallel to other query processes the same plan function for different parameters. By using *FF_APPLYP* we first investigated ways to set up different process trees manually. We concluded from our experiments that the best performing query execution plan is an almost balanced bushy tree. To automatically achieve the optimal process tree we modified *FF_APPLYP* to an operator *AFF_APPLYP* that adapts a parallel plan locally in each query process until an optimized performance is achieved. *AFF_APPLYP* starts with a binary process tree. During execution each query process in the tree makes local decisions to expand or shrink its process sub-tree by comparing the average time to process each incoming tuple. The query execution time obtained with *AFF_APPLYP* is shown to be close to the best time achieved by manually built query process trees.

I. INTRODUCTION

There is a common need to search information supplied by *data providing web services* that return a set of objects for a given set of parameters without any side effects. For example, consider a query to find *USAF Academy's Zip code* and the *State* where it is located. The three different data providing web service calls in this query are *GetAllStates* [3] to retrieve all the states, *GetInfoByState* [19] to get all the Zip codes within a given state, and *GetPlacesInside* [4] to provide all the places having a given Zip code. A naïve implementation of the example query makes 5000 calls sequentially and takes nearly 2400 seconds to execute. The reason is that each web service call incurs high latency and message set-up costs.

Queries calling data providing web services often have a similar pattern where the output (e.g. state) of one web service call is the input for another web service call (e.g. *GetInfoByState*), i.e. the second call is *dependent* on the first one, etc. A challenge here is to develop methods to speed up queries requiring such dependent web service calls.

In our approach a web service call is considered as an expensive function call where the result is a collection. It is likely that making parallel invocations of such calls will speed up the performance of queries with several dependent web service calls. To improve the response time, we present an approach to parallelize the web service calls while keeping the dependencies among them. With the approach separate *query processes* are started in parallel, each calling a parameterized sub query, called a *plan function,* for a stream of parameter tuples. Each plan function encapsulates a web service call.

The approach is implemented in the *Web Service MEDiator (WSMED)* system [15] that extends a main memory functional DBMS[14] with primitives to call web services. WSMED enables general query capabilities over data accessible through any data providing web service by reading the WSDL meta-data description. Queries are expressed in SQL. To enable simple queries to complex collections returned by web services, WSMED automatically generates flattened views of the result collections as tables.

For a given query the WSMED optimizer first produces a non-parallel plan where web service operations are called as functions. The query processor then automatically reformulates the non-parallel plan into a parallel one where web service operations are called in parallel while keeping the required dependency among the calls. The algebra operator, *FF_APPLYP* (First Finished Apply in Parallel), ships a plan function in parallel to other query processes and then calls the shipped plan function in parallel for a stream of parameter tuples.

Multi-level execution plans are generated with several layers of parallelism in different query processes. This forms the *process tree* for the query. Each child query process delivers back the result data from the shipped plan function to its parent process asynchronously. The number of children processes below a parent query process is called its *fanout*. During execution a coordinator query process first initiates the communication with its child query processes and then ships in parallel to the children their plan functions. Then a stream of different parameter tuples for the plan functions is shipped in parallel to the children. At any point in time every process in the tree executes one plan function for a specific parameter tuple. The results from the children are delivered to the parent in parallel as streams.

The performance is often improved by setting up several web service calls to the same operation in parallel rather than to call the operation in sequence for different parameters. Normally there is an optimal number of parallel calls for a given web service operation. It is therefore important to

figure out an optimized process tree for an execution plan by automatically arranging the available query processes for best performance. We first evaluated *FF_APPLYP* for different process trees by setting different fanouts manually. We tested flat and bushy process trees over existing real web services. Based on the experiments we concluded that a process tree rather close to a balanced tree performed best.

The exact properties of the composed web service operations and the computing environments involved in the calls are usually unknown. Therefore an optimal process tree is very difficult to produce using traditional query optimization assuming a cost-model describing these properties. WSMED therefore adaptively achieves an optimized process tree by run-time monitoring of the plan function calls. For the adaptation we modified *FF_APPLYP* to an operator *AFF_APPLYP* that dynamically modifies a parallel plan locally and greedily in each query process. We compared the operator *AFF_APPLYP* to the process tree with best effort manual process arrangement.

In summary the contributions of our work are:

- We define an algebra operator *FF_APPLYP* to distribute a plan function among child query processes for parallel calls with different parameter tuples.
- An algorithm is implemented to transform a central plan into a parallel plan by introducing *FF_APPLYP* operators calling plan functions encapsulating each web service call.
- Experiments with using *FF_APPLYP* showed that the best execution time for queries with dependent joins is achieved with a bushy tree rather close to a balanced one.
- To automatically optimize the parallel plan, we developed another algebra operator *AFF_APPLYP* that locally adjusts an initial balanced binary process tree adaptively until best performance is obtained.

The rest of this paper is organized as follows. In Section 2, we provide a motivating scenario used in experiments in terms of existing web services. Query process arrangements using *FF_APPLYP* are presented in Section 3. The query processing details are explained in Section 4. Experimental results and the *AFF_APPLYP* operator are presented in Section 5. Related work is analyzed in Section 6, and Section 7 summarizes and indicates future directions.

II. MOTIVATING SCENARIO

The class of queries we consider here is dependent-join [7] queries, which in their simplest form can be expressed as:

$$f(x-, y+) \wedge g(y-, z+)$$

The predicate *f* binds *y* for some input value *x* and passes each *y* to the predicate *g* that returns the bindings of *z* as result. Thus, *g* depends on the

output of *f*. The predicates *f* and *g* represent calls to parameterized sub queries, which in our case are execution plans encapsulating data providing web service operations. Inputs parameters are annotated with '-' and outputs with '+'.

We made experiments with two different queries calling different web service operations provided by different publicly available service providers.

A. Query1

In the first test case we used the SQL *Query1* in Fig . 1 that finds information about places located within 15 km from each city whose name starts with 'Atlanta' in all US states. In the query we utilize the web service operations *GetAllStates*[3], *GetPlacesWithin*[3], and *GetPlaceList* [17] . For a given web service WSMED automatically generates *operation wrapper functions* (OWFs) based on the WSDL definitions of the web service operations. Each OWFs encapsulates a data providing web service operation for given parameters and emits the result as a flattened stream of tuples. Each OWF defines an SQL view of a web service operation. SQL queries can be made over these views with the restriction that the input values of the OWFs must be known in the query. In Fig. 1 the three OWFs *GetAllStates, GetPlacesWithin,* and *GetPlaceList* define views encapsulating web service operations with the same names. The query returns a stream of 360 result tuples. A naïve central sequential execution plan invokes more than 300 web service calls.

```
Select   gl.placename,gl.state
From     GetAllStates gs, GetPlacesWithin gp,
         GetPlaceList gl
Where    gs.State=gp.state and gp.distance=15.0
         and gp.placeTypeToFind='City' and
         gp.place='Atlanta' and
         gl.placeName=gp.ToPlace+' ,'+gp.ToState
         and gl.MaxItems=100 and
         gl.imagePresence='true'
```

Fig . 1*Query 1* defined in SQL

The OWF *GetAllStates* presents information of US states as a set of tuples <*name*, *type*, *state*, *latDegrees*, *lonDegrees*, *latRadians*, *lonRadians*>. However, we are only interested in the values of the attribute *State*. The OWF *GetPlacesWithin* returns a set of tuples <To*City, ToState, GeoPlaceDistance_Distance*> for given place (*'Atlanta'*), state (*gs.State*), distance (*15.0*), and kind of place type to find (*'City'*). The OWF *GetPlaceList* retrieves a set of places <*placename, state, country, placeLon, placeLat, availableThemeMask, placeTypeId*, *population*> given a specification of a place (concatenate To*City+','+ToState*), the maximum number result tuples (*100*), and a flag indicating whether places having an associated map are returned.

Fig . 2 shows the automatically generated OWF *GetAllStates,* which flattens the result from the web service operation named *GetAllStates*. An OWF is generated based on the WSDL definition of a web service operation. Any web service operation can be invoked by the built-in function *cwo* (line 14). Its parameters are the URI of the WSDL document that describes the service, the name of the service, the operation name, and the input parameter list for the operation. The web service operation *GetAllStates* has no input parameters ({}).

```
1.  create function GetAllStates()-> Bag of
        <Charstring name, Charstring type,
         Charstring state, Real latDegrees,
         Real lonDegrees, Real latRadians,
         Real lonRadians> as
2.  select      GeoPlaceDetails['Name'],
3.              GeoPlaceDetails['Type'],
4.              GeoPlaceDetails['State'],
5.              GeoPlaceDetails['LatDegrees'],
6.              GeoPlaceDetails['LonDegrees'],
7.              GeoPlaceDetails['LatRadians'],
8.              GeoPlaceDetails['LonRadians']
9.  from        Sequence out,
10.             Record GetAllStatesResult ,
11.             Record GetAllStatesResult1,
12.             Sequence GetAllStateResult2,
13.             Record GeoPlaceDetails
14. where       out=cwo('http://codebump.com/servi
                ces/PlaceLookup.wsdl',
                'GeoPlaces', 'GetAllStates',
                {})and
15.             GetAllStatesResult1 in out and
16.             GetAllStatesResult2 =
                GetAllStatesResult1
                ['GetAllStatesResult']and
17.             GetAllStateResult in
                GetAllStatesResult2 and
18.             GeoPlaceDetails=GetAllStatesResult
                ['GeoPlaceDetails'];
```

Fig . 2 Automatically generated OWF *GetAllStates*

The result from *cwo* is bound to the query variable *out* (line 14). It holds an object representing the output from the web service operation temporarily materialized in WSMED's local store. The OWF converts the output XML structure from the web service operation call into records and sequences. The result *out* is here a sequence from which elements are extracted (line 15)

into the *GetAllStatesResult1* record structure using the *in* operator. The records have only one attribute named *GetAllStatesResult* whose values are assigned to another sequence structure *GetAllStatesResult2* (line 16). An attribute *a* of a record *r* is accessed using the notation *r[a]*. Each element record from the sequence *GetAllStatesResult2* is bound to the variable *GetAllStateResult* (line 17). The values of the attribute *GeoPlaceDetails* are assigned to the *GeoPlaceDetails* record with attributes *Name, Type, State, LatDegrees, LonDegrees, LatRadians,* and *LonRadians* (line 18). The OWFs *GetPlacesWithin* and *GetPlaceList* are automatically generated analogously.

B. Query2

The second case, *Query2* in Fig . 3, finds the zip code and state of the place 'USAF Academy'. A naïve sequential plan invokes more than 5000 web service calls. Here also three different dependent web services are involved. *GetAllStates* is the same as in *Query1*. *GetInfoByState* is provided by the USZip [19] web service to retrieve all zip codes for a given state as a single comma separated string (*gi.GetInfoByStateResult*). *getzipcode* is an helping function defined in WSMED that extracts the set of zip codes (*gc.zipcode*) given a string of zip codes (*gc.zipstr*). The OWF *GetPlacesInside* is supported by the Zipcodes [4] web service provider and returns for a given zip code a set of tuples <*ToPlace, ToState, Distance*> where *ToPlace* is a place located within the zip code area, *ToState* is the state of the place, and *Distance* is the distance from the place to the origin of the given zip code area.

```
select   gp.ToState, gp.zip
From     GetAllStates gs, GetInfoByState gi,
         getzipcode gc, GetPlacesInside gp
Where    gs.State=gi.USState and
         gi.GetInfoByStateResult=gc.zipstr and
         gc.zipcode=gp.zip and
         gp.ToPlace='USAFAcademy'
```

Fig . 3 *Query2* defined in SQL

III. WSMED PROCESS ARRANGEMENT

The web service metadata in a WSDL document is first imported and stored in the WSMED local database [15]. A query is processed by a coordinator process *q0.* Fig . 4 gives an example of a process tree generated by the WSMED query optimizer. Every query process on each level can be connected with a number of child processes and all the processes on the same level execute the same plan function but with different parameters.

In Fig . 4, *q1* is connected with *q3, q4,* and *q5*. The plan function in the coordinator *q0* encapsulates the OWF *GetAllStates*, while the plan functions of the processes in level one encapsulate the OWF *GetPlacesWithin* for different states. On level two the plan function calls the OWF *GetPlaceList* for different place specifications.

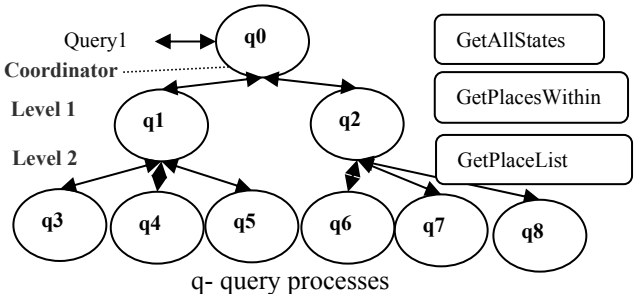


Fig . 4 Process tree

The coordinator *q0* first generates a central plan containing calls to the OWFs. It then automatically reformulates the central plan to incorporate parallel web service calls by inserting algebra operators *FF_APPLYP* in the execution plan whenever an OWF is encountered. For each OWF a plan function is generated that encapsulates a fragment of the central execution plan embodying the OWF call. When the algebra operator *FF_APPLYP* is executed in process *q0,* it first ships in parallel to its children in level one (*q1,q2*) the same plan function definition that encapsulates *GetPlacesWithin*. Then it ships in parallel different parameter tuples to the shipped plan function installed in the children processes ready for execution. Analogously, the *FF_APPLYP* operators executing in the level one processes send another plan function definition to the level two processes (*q3,q4,q5,q6,q7,q8*). Each query process initially receives its own plan function definition once before execution. When the level two processes receive data from the wrapped web service operation *GetPlaceList*, the results will be returned asynchronously as streams to the processes in level one, and finally the results are streamed to the coordinator process.

A. FF_APPLYP

The operator *FF_APPLYP* enables parallel invocation of a plan function for different parameter tuples delivered as an input stream to *FF_APPLYP*. *FF_APPLYP* has the signature:

FF_APPLYP(Function pf, Integer fo, Stream pstream) → Stream result

It ships in parallel to *fo* number of child query processes the definition of the same plan function *pf*. Then it ships one by one parameter tuples from *pstream* to each of the children. The result stream from a call to *pf* for a given parameter tuple is sent back to *FF_APPLYP* asynchronously as a stream of tuples, *result*.

In our first experiments the fanout *fo* is set manually for each level. This allows us to analyze different process trees. In Fig . 4 the fanout on level one is $fo_1=2$ and on level two $fo_2=3$. The coordinator *q0* at level zero first initializes the two child processes *q1* and *q2*. Then *q0* ships the plan function

encapsulating the web service operation *GetPlacesWithin* to the children (*q1, q2*). When all plan functions are shipped it starts picking parameter tuples one by one from *pstream*, to send down to the plan function started in the children. In *q0* the stream *pstream* is a stream of state names produced as the result of the plan function that encapsulates the web service operation *GetAllStates*. When the first round of parameter tuples are shipped to all children, *FF_APPLYP* will broadcast that it is ready to receive results. Whenever a result tuple is received from some child it is directly emitted as a result of *FF_APPLYP*. When a child completed the processing of a plan function for a given parameter tuple it sends an *end-of-call* message to *FF_APPLYP*. When the parent receives an end-of-call message from a child it will ship the next pending parameter tuple from *pstream* to the idle child process. When there are no pending parameter tuples in *pstream* and no pending results from the child processes, *FF_APPLYP* is finished.

IV. Query Parallelization in WSMED

Fig . 5 illustrates the query processor in WSMED [15]. The *calculus generator* produces from a given user query defined in SQL an internal calculus expression in a Datalog [13] dialect. The symbol '_' represents an anonymous result variable.

Query1 is transformed into the following calculus expression:

```
Query1(pl,st) :-
    GetAllStates()                    AND
    GetPlacesWithin('Atlanta',_,
                    15.0,'City')      AND
    GetPlaceList(_, 100,'true')
```

With naïve query optimization the calculus expression is translated by the central plan creator into the algebra expression in Fig . 6. The central plan creator uses a simple heuristic web service cost model based on the signatures of web service operations assuming that web service operations are expensive. The algebra expressions contains calls to the *apply* operator γ[6], which applies a plan function for a given parameter tuple. The naïve central query execution plan with γ can be directly interpreted but with very bad performance since many web service operations are applied in sequence.

The plan first executes the OWF *GetAllStates* returning a stream of tuples <*st1*>. Each of these tuples are fed to the next OWF *GetPlacesWithin* called by the apply operator with the given argument tuple *('Atlanta', st1, 15.0, 'City')* returning a stream of tuples <*city, st2*>. The built in function *concat* is then applied on each argument tuple *(city,',',st2)* producing a stream of strings *str*. Finally the OWF *GetPlaceList* is applied on each argument tuple *(str,100,'true')* returning a stream of tuples <*pl,st*>.

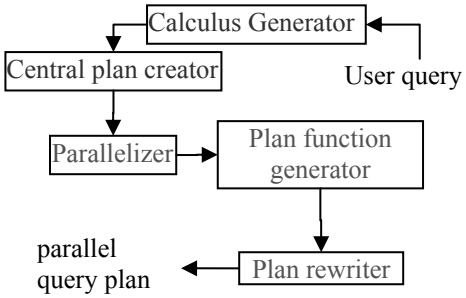


Fig . 5 Query Processor

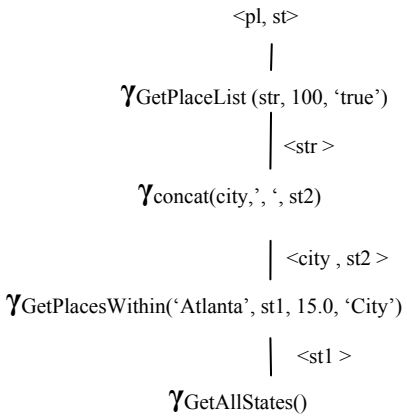


Fig . 6 Central query plan - *Query1*

The *parallelizer* in Fig . 5 takes as input a central plan (e.g. the one in Fig . 6) and identifies there the parallelizable OWFs. Since the parallelization is based on parameter streams, OWFs not having input parameters are not considered. For example, the plan in Fig . 6 can be parallelized for the OWFs *GetPlacesWithin* and *GetPlaceList*, but not for *GetAllStates*. The parallelizer splits the plan into one section for each parallelizable OWF starting from the bottom. The first section, flattening the result from the call to the web service operation *GetAllStates*, is executed in the coordinator. The next section contains the calls to *GetPlacesWithin* and *concat*. The final section contains only the call to *GetPlaceList*.

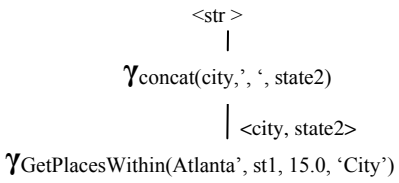


Fig . 7 Plan function *PF1* wrapping *GetPlacesWithin*

For each parallelizable section the *plan function generator* creates a plan function that encapsulates a parallelizable call to an OWF. For example, the plan function *PF1* in Fig . 7 encapsulates the OWF *GetPlacesWithin*. It has the signature *PF1(Charstring st1) → Stream of Charstring str*. Analogously *PF2* in Fig . 8 flattens the web service operation *GetPlaceList* to return a stream of tuples *<pl, st>* and has the signature *PF2(Charstring str) → Stream of <Charstring pl, Charstring st>*.

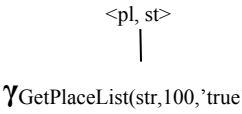


Fig . 8 Plan function *PF2* wrapping *GetPlaceList*

Finally, the *plan rewriter* transforms the central query by inserting the algebra operator *FF_APPLYP* for each generated plan function. Fig . 9 shows the final parallelized execution plan with two calls to *FF_APPLYP* (*FF_γ*)**.**

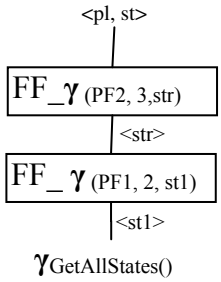


Fig . 9 Parallel execution plan-*Query1*

Analogously *Query2* is initially compiled into the central plan in Fig . 10. The central plan first executes the OWF *GetAllStates* to return a stream of

tuples <*st1*>. These outputs are fed to the next OWF *GetInfoByState* returning a stream of single comma separated strings *zstr*. For each *zstr* the γ operator applies the user defined helping function *getzipcode* to produce a stream of extracted zip codes *zc*. Then the OWF *GetPlacesInside* is applied for each *zc* returning a stream of tuples <*st, pl, zc*>. Finally the *equal* function is applied to check if *pl* is equal to '*USAF Academy*' and returns stream of valid tuples <*st ,zc*>.

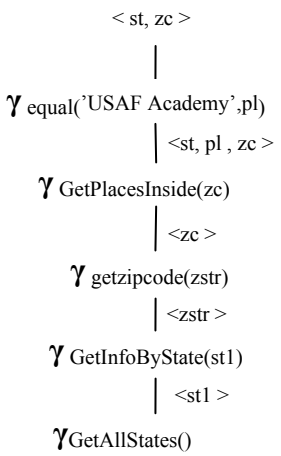


Fig . 10 Central query plan- *Query2*

The parallelizer splits the first parallelizable section (call to OWF *GetAllStates*) to execute in the coordinator. The next parallelizable section contains the calls to *GetInfoByState* and *getzipcode*. The final section contains only the call to *GetPlacesInside* and *equal*. Then the plan function generator creates plan functions to encapsulate the parallelizable OWFs. The plan function *PF3* in Fig . 11 encapsulates *GetInfoByState*. It has the signature:

PF3(Charstring st1) → Stream of Charstring zc.

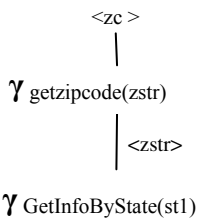


Fig . 11 Plan function *PF3* wrapping *GetInfoByState*

PF4 in Fig . 12 wraps the OWF *GetPlacesInside* and returns <*st,zc*>. It has the signature:

PF4(Charstring zc) → <*Charstring st, Charstring zc*>.

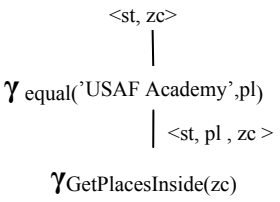


Fig . 12 Plan function *PF4* wrapping *GetPlacesInside*

Finally, the plan rewriter transforms the central query by inserting *FF_γ* for each generated plan function as illustrated in Fig . 13.

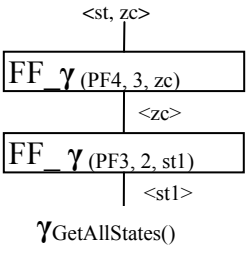


Fig . 13 Parallel execution plan-*Query2*

V. EXPERIMENTS

We compared the query execution times for *Query1* using the central execution plan in Fig . 6 with the parallel plan in Fig . 9 (for *Query2* we compare the plans in Fig . 10and Fig . 13). To analyze different process trees, we set manually a *fanout vector* with fanouts for the different process tree levels to evaluate the query execution times. The tests were run on a computer with a 3 GHz single processor Intel Pentium 4 with 2.5GB RAM. We evaluated the following process trees:

- *Flat tree* (Fig . 14): The fanout vector has fo_2=0 ($\{fo_1,0\}$) in which case both OWFs are combined into the same plan function executed at the same level.
- *Unbalanced tree* (Fig . 15): Fanout vector $\{fo_1,fo_2\}$, $fo_1 \neq fo_2$
- *Balanced tree*: the fanouts are equal, i.e. $fo_1 = fo_2$

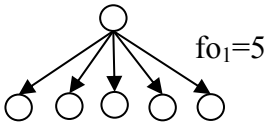


Fig . 14 Flat tree

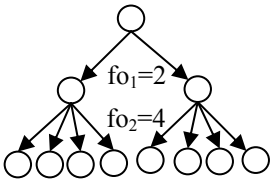


Fig . 15 Unbalanced tree

The total number of query processes *N* needed to execute the parallel queries is $N = fo_1 + fo_1 * fo_2$.

In general, there should be an optimum shape of the process tree based on properties of the web service calls, which are not known. The experiments investigate the optimum tree topology for up to 60 query processes.

Fig . 16 illustrates the execution times in seconds for *Query1* by varying the values of fo_1 and fo_2. It shows the lowest execution time region is achieved within the range 50 - 60 sec. The fastest execution time 56.4 sec for fanout vector *{5,4}* outperformed with speedup 4.3 the central plan (244.8 sec). Fig . 17 shows that the best execution time for *Query2* is achieved within the range of 1200-1400 sec. The best execution time 1243.89 sec for fanout vector *{4,3}* outperformed with speed up of nearly 2 the central plan (2412.95 sec).

We notice from the experiments that the best execution time for both queries is achieved close to, but not exactly for, balanced trees, (*Query1*: fo_1=5, fo_2=4 , *Query2*: fo_1=4, fo_2=3).

A. Adaptive apply, AFF_APPLYP

To automatically achieve an optimized process tree, we developed another algebra operator *AFF_APPLYP* (Adaptive First Finished Apply in Parallel) to replace *FF_APPLYP,* but requires no explicit fanout argument.

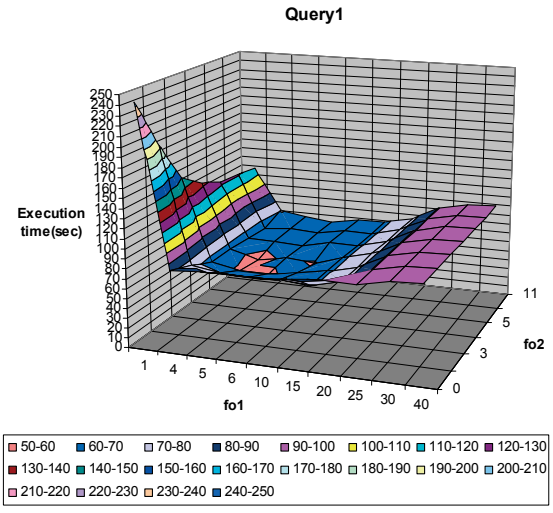


Fig . 16 Execution time for *Query1*

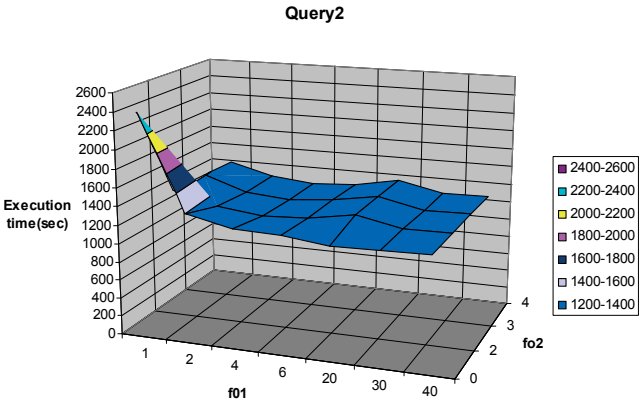


Fig . 17 Execution time for *Query2*

Based on the observation that the best parallelization is close to a balanced tree, *AFF_APPLYP* adapts the process plan at run time starting with a binary tree. Each node locally monitors the execution times of its children to dynamically modify its subtrees *AFF_APPLYP* does the following:

1. *AFF_APPLYP* initially forms a binary process tree (Fig . 18) by always setting fanout to 2, the *init stage*.
2. A *monitoring cycle* for a non-leaf query process is defined as when it has received the same number of end-of-call messages as its number of

children. After the first monitoring cycle *AFF_APPLYP* adds *p* new child processes. Adding new processes is called an *add stage*. In Fig . 19, *p=1* and therefore query process *q0* adds one new process *q7* at level 1, while *q1* and *q2* add *q10* and *q11* at level 2, respectively.

3. When an added node has several levels of children the init stages of the children's *AFF_APPLYs* will produce balanced binary sub–trees. That is, *q7* adds *q8* and *q9*.
4. *AFF_APPLYP* records per monitoring cycle *i* the average time t_i to produce an incoming tuple from the children. If t_i decreases more than a threshold (set to 25%) the add stage is rerun. If t_i increases we either stop or run a *drop stage* that drops one child and its children. In Fig . 20, *q2* adds *q12,* while *q0* drops *q7,* and *q7* drops *q8* and *q9*.

We experimented with different values of *p* and different change thresholds, with and without the drop stage. The results for 25% change are shown in Fig . 21. The fanout values are exact for *FF_APPLYP* while fo_1 and fo_2 for *AFF_APPLYP* are average fanouts. The measurements include the adaptation times.

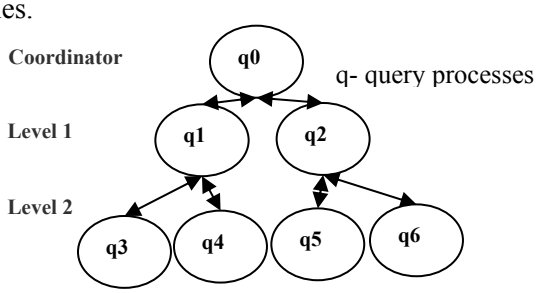


Fig . 18 Binary process tree

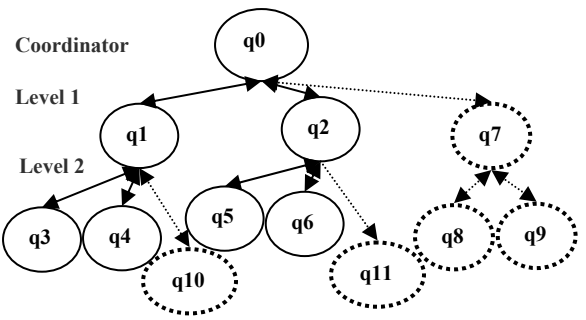


Fig . 19Adding processes

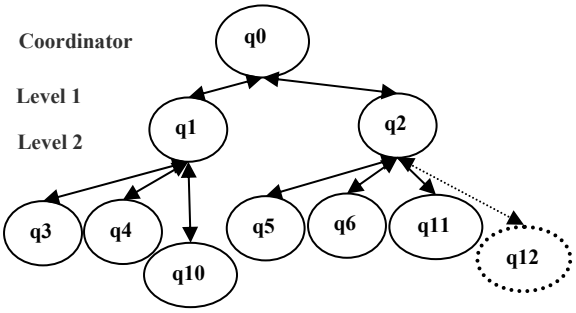
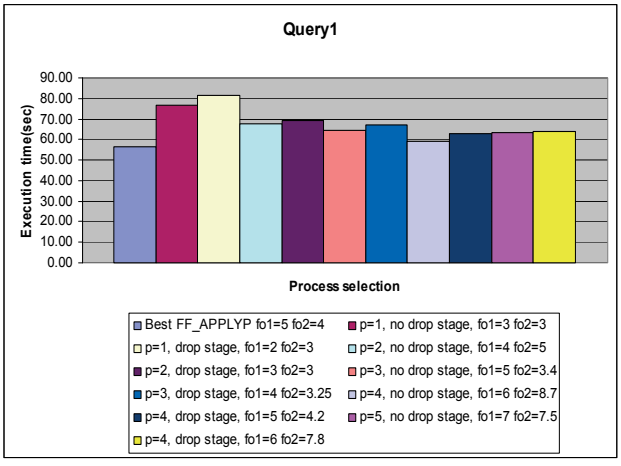


Fig . 20 Adding and removing processes



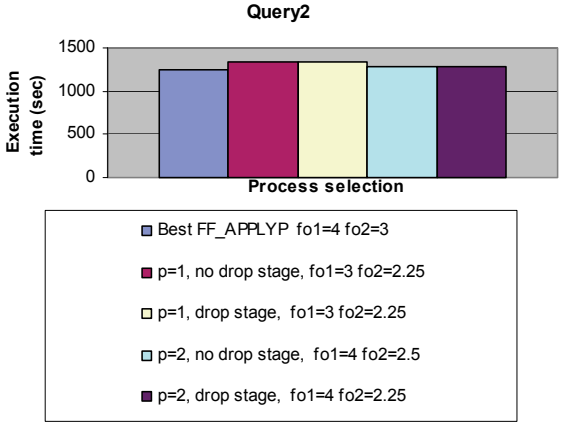


Fig . 21Execution time with AFF_APPLYP

We notice that for *Query1* the execution time with p=4 and no drop stage comes close to the execution time of the best manually specified process tree, while for *Query2* the execution with p=2 and no drop stage is the closest one.

We concluded in both cases that execution time with $p=2$ and no drop stage is close to the execution time of the best manually specified process tree (*Query1* 80%, *Query2* 96 %) and further dropping processes make insignificant changes in the execution time.

VI. Related work

BPEL [2] proposes workflow primitives to manually invoke parallel web service calls. It requires a lot of effort on the part of the programmer to manually identify sections of the code to run in parallel, and to specify dependencies among the calls. In contrast, WSMED automatically compiles a given query over composed data providing web services by generating an adaptive, parallel, and optimized workflow.

In [1] an approach is described for optimizing web service compositions by procedurally traversing ActiveXML documents to select embedded web service calls. It demonstrates the gain obtained by maximizing parallelism achieved by invoking calls to *independent* web services in a query. Conversely, WSMED adaptively parallelizes *dependent* web service calls.

WSQ/DSQ [9] handles high-latency calls to web search engines by launching asynchronous materialized dependent joins later joined in the execution plan using a special operator. In contrast, WSMED produces non-blocking multi-level parallel plans based on streams of parameter tuples passed to parallel sub plans without any materialization.

WSMS [16] proposed an approach for pipelined parallelism among dependent web services to minimize the query execution time. By contrast, we parallelize by partitioning parameter tuple streams. Furthermore, WSMS didn't propose any adaptive parallelization, lacked support for code shipping, and couldn't make parallel calls to the same web service. In contrast we propose a strategy to adaptively produce a parallelized plan where *AFF_APPLYP* invokes parameterized plans calling web services in parallel.

Like two-phase parallel query optimization [11] WSMED also generates a parallelized query execution plan from an initial central query plan. However, WSMED adaptively parallelizes dependent joins by generating plan functions that are called in parallel using the adaptive operator *AFF_APPLYP*, while [11] focused on static inter-operator parallelism in distributed databases based on a static cost model.

The plan function and parameter tuple shipping phase of *FF_APPLYP* is similar to the map phase of *MAPREDUCE*[5]. However, *MAPREDUCE* is more of a programming model than a query operator and is not dynamically rearranging query execution plans as *AFF_APPLYP*.

In [10] run time adaptation of buffer sizes in web service calls is investigated, not dealing with adaptive parallelism on web service calls at the client side.

The formal basis for using views to query heterogeneous data sources is reviewed in [8] [18]. *Chocolate* [12] extends the federated database capabilities of *DB2/UDB* by automatically creating views of web services from WSDL descriptions, similar to the OWF generation in WSMED. However, Chocolate does not deal with adaptive parallelization of the web service calls in a query as WSMED.

VII. Conclusions and Future work

We presented an approach to automatically parallelize queries with dependent web service calls. The algebra operator *FF_APPLYP* was first defined in order to parallelize calls to parameterized sub plans partitioned for different parameter tuples. We did experiments by manually arranging different process trees with different fanouts. From the experiments we concluded that the optimum process fanout is close to, but not exactly, a balanced tree. To adaptively find the best process tree we devised an algebra operator *AFF_APPLYP* that starts with a balanced binary process tree and then each non-leaf process locally adapts the process sub-trees by adding and removing children until an optimum is reached, based on monitoring the flow of result tuples from the children. The adaptive method obtained performance close to the best manually specified process tree.

Our algebra operators *FF_APPLYP* and *AFF_APPLYP* can handle parallel query plans for a query with any number of dependent joins. We would like to generalize the strategy for queries mixing both dependent and independent web service calls, as well bushy trees. Further we need to investigate different process arrangement strategies with the algebra operators.

Acknowledgments

This work is supported by Sida and the Swedish Foundation for Strategic Research under contract RIT08-0041.

References

[1] S. Abiteboul et al., Lazy query evaluation for active XML, *Proc. of the 2004 ACM SIGMOD*, 227–238, 2004.

[2] T. Andrews et al., Business Process Execution Language for Web Services version 1.1., *http://ifr.sap.com/bpel4ws/*, 2003

[3] codeBump, GeoPlaces web service, http://codebump. com /services /PlaceLookup.asmx

[4] codeBump, Zipcodes web service, http://codebump.com/services /ZipCodeLookup.asmx

[5] J.Dean, and S.Ghemawat, MAPREDUCE: Simplified Data Processing on Large Clusters*, Communications of the ACM*, 51(1), 107-113, 2008

[6] G. Fahl, and T. Risch, Query Processing over Object Views of Relational Data, *The VLDB Journal* , 6(4), 261-281, 1997
[7] D.Florescu, A.Levy, I.Manolescu and D.Suciu, Query Optimization in the Presence of Limited Access Patterns, *Proc. of ACM SIGMOD '99,* 311-322, 1999
[8] 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, *Journal of Intelligent Information Systems*, 8(2): 117-132, 1997
[9] R.Goldman, and J.Widom, WSQ/DSQ: a practical approach for combined querying of databases and the Web, *Proc. of 2000 ACM SIGMOD Intl. Conf. on Management of Data,* 285-296*,* 2000.
[10] A. Gounaris, et al., Robust runtime optimization of data transfer in queries over Web Services, *Proc. of ICDE 2008,* 2008
[11] W.Hasan, D.Florescu, and P.Valduriez, Open Issues in Parallel Query Optimization, *SIGMOD Record*, 25(3), 1996
[12] V.Josifovski, S.Massmann, and F.Naumann, Super-Fast XML Wrapper Generation in DB2: A Demonstration, *Proc. International Conference of Data Engineering, (ICDE'03)*, 756-758, 2003
[13] W. Litwin, and T. Risch, Main Memory Oriented Optimization of OO Queries using Typed Datalog with Foreign Predicates, *Proc. IEEE Trans. on Knowledge and Data Engineering,* 4(6), 517-528, 1992
[14] T.Risch, V.Josifovski, and T.Katchaounov, Functional Data Integration in a Distributed Mediator System*, Functional Approach to Data Management - Modeling, Analyzing and Integrating Heterogeneous Data*, Springer, 211-238, 2003
[15] M.Sabesan and T.Risch, Web Service Mediation Through Multi-level Views, *Proc. International Workshop on Web Information Systems Modeling (WISM 2007)*, 755-766, 2007
[16] U.Srivastava, J.Widom, K.Munagala, and R.Motwani, Query Optimization over Web Services, *Proc. Very Large Database Conference (VLDB 2006),* 2006
[17] TerraServer, TerraService, http://terraservice.net/webservices.aspx
[18] J.D.Ullman, Information Integration Using Logical Views, *Proc. 6th International Conference on Database Theory (ICDT '97)*, 19-40, 1997
[19] USZip, http://www.webservicex.net/uszip.asmx

Paper III

Adaptive Parallelization of Queries to Data Providing Web Service Operations

Manivasakan Sabesan and Tore Risch

Department of Information Technology, Uppsala University, Sweden
{msabesan,Tore.Risch }@it.uu.se

Abstract. A data providing web service operation returns a collection of objects for given parameters without any side effects. The Web Service MEDiator (WSMED) system automatically provides relational views of any data providing web service operations by reading their WSDL documents. These views can be queried with SQL. In an execution plan a call to a data providing web service operation may be dependent of the results from other web service operation calls. In other cases different web service calls are independent of each other and can be called in any order. In WSMED the adaptive operator *PAP* speeds up queries with both dependent and independent web service operation calls. It adaptively parallelizes calls to web service operations until no significant performance improvement is measured. The performance of *PAP* is evaluated using publicly available web services. The operator is shown to substantially improve the query performance without any cost knowledge or extensive memory usage compared to other strategies.

Keywords: Web service composition, Adaptive parallelization, Query optimization.

1 Introduction

Data providing web service operations are web service operations where data collections are retrieved from servers without side effects. The Web Service MEDiator (WSMED) system enables general query capabilities over any data providing web service operations without any further programming. WSMED automatically provides relational views of the operations by reading the WSDL documents. These views can be queried and joined with SQL. A web service operation is considered as a high latency function call where the result is a nested data collection. For a given SQL query, WSMED first generates an initial execution plan calling web service operations. At run time the initial execution plan is adaptively parallelized.

As an example, consider a query to find all the information of the places in some of the US states along with their zip codes and weather forecasts. Four different data providing web service operations can be used for answering this query. First the *GetAllStates* operation from the web service *GeoPlaces* [3] is called to retrieve the states. The *GetInfoByState* operation by *USZip* [13] returns the zip codes for a given US State. The *GetPlacesInside* operation by *Zipcodes* [4] retrieves the places located within a given zipcode. Finally, the *GetCityForecastByZip* operation by *CYDNE* [5] returns weather forecast information for a given zip code.

Two operation calls are dependent if one of them requires as input an output from the other one, otherwise they are independent. In the above example, the web service operations *GetPlacesInside* and *GetCityForecastByZip* are dependent on *GetInfoByState* but independent of each other. A challenge here is to develop methods to speed up queries requiring both dependent and independent web service calls. In general such speed-ups are based on some unknown web service properties. Those properties are not explicitly available and depend on the network and runtime environments when and where the queries are executed. In such scenarios it is very difficult to base execution strategies on a static cost model, as is done in relational databases.

To improve the response time without a cost model, WSMED uses an approach to automatically parallelize the web service calls at run time while keeping the dependencies among them. For each web service operation call the optimizer generates a *plan function* which encapsulates the web service operation call and makes data transformations such as nesting, flattening, filtering, data conversions, and calls to other plan functions.

Web service operations are usually parameterized where input parameters have to be bound to call them. WSMED will decompose the query plan to guarantee this. The performance is often improved by setting up several parameterized web service calls in parallel rather than to call the operation in sequence for different parameters. In WSMED such multi-level execution plans are automatically generated as several layers of parallelism where each parameterized plan function is executed in different query processes. This forms a process tree for the query.

In the initial execution plan the dependencies between dependent and independent plan functions calls are resolved so that collections of independent calls are grouped together.

For adaptive parallelization of queries with web service operation calls, the algebra operator, *PAP* (Parameterized Adaptive Parallelization) is implemented. It takes as arguments a set of independent plan functions along with a stream of parameter values to be processed by the plan functions. For each received parameter tuple it starts one process per plan function call. Different plan functions will select different elements from the input tuple. The results from the query processes are collected asynchronously and

delivered as a stream. The result tuples from *PAP* are formed by combining result tuples from each child. When a child process has delivered all result tuples in a call it is terminated and another child plan function call is started asynchronously.

A set of independent plan function calls is processed by a single call to *PAP*. By contrast, dependent plan function calls are processed as sequences of parallelized *PAP* calls.

For the adaptation *PAP* dynamically modifies the parallel plan by monitoring the performance of each plan function call. Based on the monitoring new children are started until no significant performance improvement is measured. Sequences of *PAP* calls will start sequences of process sub-trees which are locally adapted as well.

The *PAP* operator provides process tree adaptation without any central control or cost model. At any point in time every process in the tree executes one plan function for a specific parameter tuple.

In summary the contributions of our work are:

1. For a given SQL query, the system automatically generates a parallel execution plan calling *PAP* that adaptively parallelizes both dependent and independent web service operation calls.
2. *PAP* is shown to substantially improve the query performance without any cost knowledge or extensive memory usage compared to other strategies.

In Section 2, we provide a motivating scenario used in experiments in terms of real web services. Section 3 shows how the query plans are generated. In Section 4 adaptive parallelization using PAP and experimental results are presented. Related work is analyzed in Section 5, and finally Section 6 summarizes and indicates future directions.

2 Motivating Scenario

We consider the class of queries with both dependent and independent joins, which in their simplest form can be expressed as: $\{v, z \mid e(u-, v+) \wedge f(x-, y+) \wedge g(y-, z+)\}$

Input parameters are annotated with '-' and outputs with '+'. Given the input values *u* and *x* the query returns the tuple <*v,z*> where the predicate *e* binds *v* for the given *u*. The predicate *f* binds *y* for the given *x* and passes each *y* to the predicate *g* that returns *z*. Thus, predicate *g* depends on the output of *f* but *e* and *f* are independent. The predicates *e, f,* and *g* represent calls to plan functions encapsulating data providing web service operations. We made experiments with two different queries calling different web

service operations provided by the previously mentioned publicly available service providers.

2.1 Query1

The example SQL *Query1* in Fig. 1 has the above form. It finds all information about places in some of the US states, along with their zip codes and weather forecasts. The result set size is scaled by varying the number of selected states.

```
select gp.TOPLACE,gp.TOSTATE,gz.ZIPCODE,gc.DATE,gc.DESCRIPTION
from   GetAllStates gs, GetPlacesInside gp, GetInfoByState gi,
       GetCityForeCastByZip gc, getzipcode gz
where gs.State<'MD' and gi.USState=gs.State and
      gi.GetInfoByStateResult=gz.zipstr and
      gz.zipcode=gp.zip and gc.zip=gz.zipcode
```

Fig. 1. SQL *Query1*

For a given web service WSMED automatically generates *Operation Wrapper Functions (OWF)* [11] that represent SQL views of the web service operations based on the WSDL definitions. To provide relational views of web service operations returning complex objects, the OWFs flatten the result from the web service call. Analogously, web service operation arguments are constructed as a nested structure before an operation is called. For Q*uery1*, the views *GetAllStates, GetInfoByState, GetPlacesInside,* and *GetCityForeCastByZip* are defined as OWFs that encapsulate four different web service operations from four different service providers. The OWF *GetAllStates* presents information of US states as a set of tuples <*state*>. The OWF *GetInfoByState* retrieves all zip codes for a given state as a single comma separated string (*gi.GetInfoByStateResult*). *getzipcode* is a helping function defined in WSMED that extracts the set of zip codes (*gz.zipcode*) given a string of zip codes (*gz.zipstr*). The OWF *GetPlacesInside* returns for a given zip code a set of tuples <*ToPlace, ToState, Distance*> where *ToPlace* is a place located within the zip code area, *ToState* is the state of the place, and *Distance* is the distance from the place to the origin of the zip code area. The OWF *GetCityForeCastByZip* reports the weather forecast as a set of tuples <*Date, Description*> for a given zip code where *Date* is date of the forecast, and *Description* is the short description of the forecast. In the above query the OWF *GetInfoByState* depends on OWF *GetAllStates*. The OWFs *GetPlacesInside*, and *GetCityForeCastByZip* depend on the OWF *GetInfoByState*, while the OWFs *GetPlacesInside*, and *GetCityForeCastByZip* are independent on each other.

2.2 Query2

The SQL *Query2* in Fig. 2 has one more dependent OWF *GetPlaceDetails* than *Query1*. It finds all information about places in some of the US states, along with their zip codes, weather forecasts, and geographical positions. The result set size is scaled by varying the number of selected states and filtering city names.

```
select gd.Name,gd.LatDegrees,gd.LonDegrees,
       gz.ZIPCODE,gc.DATE,gc.DESCRIPTION
from   GetAllStates gs, GetPlacesInside gp, GetInfoByState gi,
    GetCityForeCastByZip gc, getzipcode gz, GetPlaceDetails gd
where gs.State<'MD' and gi.USState=gs.State and
      gi.GetInfoByStateResult=gz.zipstr and
      gz.zipcode=gp.zip and gc.zip=gz.zipcode and
      gd.Place like '[A-Z]*' and gd.Place=gp.TOPLACE and
      gd.State=gp.TOSTATE
```

Fig. 2. SQL *Query2*

For *Query2*, the views *GetAllStates, GetInfoByState, GetPlacesInside,* and *GetCityForeCastByZip* are the same as for *Query1*. The additional OWF *GetPlaceDetails* returns for a given city and state a set of tuples <*Name,LatDegrees, LonDegrees*> where *Name* is a place located within the city, and *LatDegrees* and *LonDegrees* represents latitude and longitude of the place in degrees, respectively. The OWF *GetPlaceDetails* depends on OWF *GetPlacesInside*. *Query2* filters the city name *Place* (*gd.Place like '[A-Z]*'*) since the web service operation *GetPlaceDetails* [3] doesn't support such filters.

3 Query Plans

The WSMED query processor first generates a central plan containing calls to the web service operations. It is a left-deep tree of executable predicates enumerated from 0 and up. The central plan contains calls to the *apply* operator γ, which applies a plan function for a given parameter tuple. The non parallel query execution plan with *γ* can be directly interpreted but with very bad performance, since the web service operations are applied in sequence.

In Fig. 3 the central *Plan1* for *Query1* first calls the plan function that encapsulates the web service operation *GetAllStates* returning a stream of tuples <*state*>, which is then selected by inequality (*pos=1*). Each of these tuples is fed to the next plan function encapsulating web service operation *GetInfoByState* parameterized by *state* returning a stream of comma separated strings *zipstr*. For each *zipstr* the γ operator applies the user defined helping function *getzipcode* to produce a stream of extracted zip

codes *zipcode*. Then the plan function encapsulating web service operation *GetPlacesInside* is applied on each argument tuple <*zipcode*> to produce a stream of tuples <*toplace, tostate, zipcode*>. Finally the plan function for *GetCityForeCastByZip* is applied on each argument tuple <*toplace, tostate, zipcode*> returning as the query result a stream of tuples <*toplace, tostate, zipcode, date, description*>.

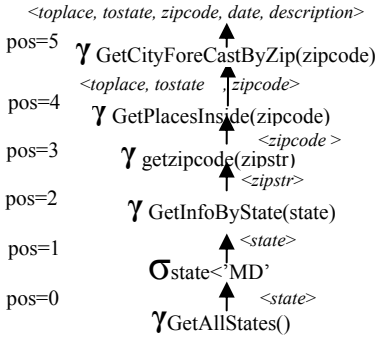


Fig. 3 Central *Plan1*

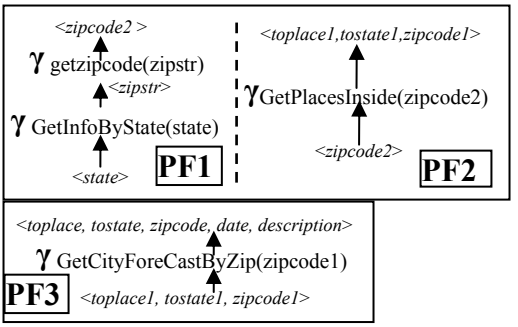


Fig. 4 Plan functions

For each web service operation call a plan function is generated that encapsulates a fragment of the non-parallel execution plan embodying the web service operation call. Each fragment is defined as a set of predicates from one web service operation call up to just before the next web service operation call in *Plan1*. The WSMED query processor then automatically reformulates *Plan1* to incorporate parallel web service calls by inserting *PAP* in the execution plan for each plan function call.

Fig. 4 shows the query plans of three different parallelizable fragment plan functions *PF1, PF2* and *PF3* generated by the WSMED query processor for *Query1*. *PF1* calls the web service operation *GetInfoByState*, and the foreign function *getzipcode*. *PF2* calls web service operation *GetPlacesInside* while *PF3* calls the web service operation *GetCityForeCastByZip*.

In the parallel plan Fig. 5 the *PAP* operator applies in parallel one plan function at the time. It is suboptimal since it assumes that all the web service operation calls are considered as dependent on each other and the *PAP* operators are therefore called in sequence. The *PAP* operator adaptively parallelizes the calls to the plan functions *PF1, PF2* and *PF3* so that they will be executed as a parallel pipeline. A better plan will be shown later.

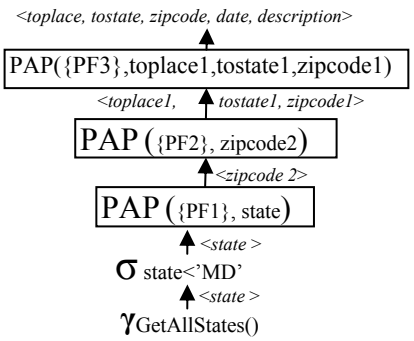


Fig. 5 Dependent adaptive parallel plan

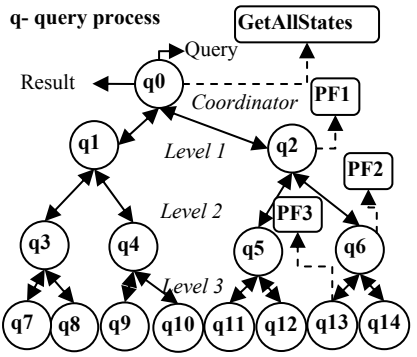


Fig. 6 Adaptive dependent parallel process tree

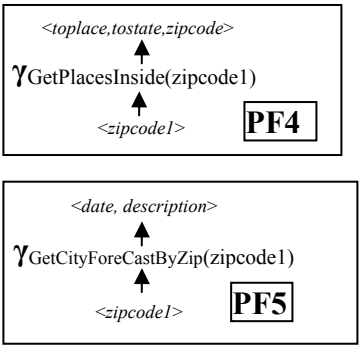


Fig. 7 Independent plan functions

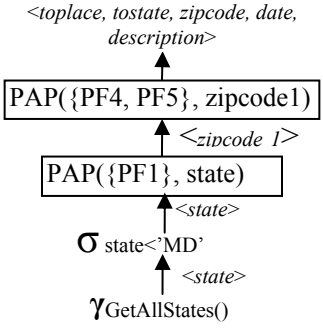


Fig. 8 Dependent and independent adaptive parallel execution plan

Once the parallel plan is started *PAP* will automatically start new parallel processes to dynamically form a process tree. Fig. 6 shows a process tree for the dependent parallel plan of *Query1* in Fig. 5. Every query process on each level is connected with several child processes. All processes on the same level execute the same set of plan functions for that level, but with different parameter tuples. On each level always one plan function is applied.

In Fig. 6, the coordinator *q0* is connected with *q1* and *q2*. The execution plan in *q0* calls the non-parameterized web service operation *GetAllStates*, while *PF1* executing in level one calls the web service operation *GetInfoByState* for different states. On level two *PF2* calls the web service operation *GetPlacesInside* for different zipcodes. Finally on level three *PF3* calls the web service operation *GetCityForecastByZip* for different zipcodes.

In this plan the web service operations *GetPlacesInside* and *GetCityForecastByZip* are regarded as dependent on each other. This makes the web service operation *GetCityForecastByZip* be called several times for the same zipcode. Since web service calls have high latency, these redundant

calls cause delays. Next it will be shown how such redundant calls are removed when the web service operations are independent.

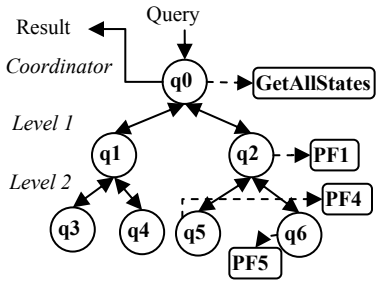


Fig. 9 Adaptive parallel process tree – dependent and independent

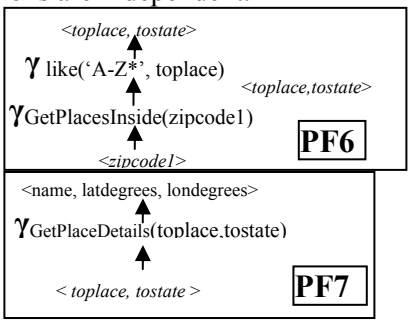


Fig. 10 Plan functions-*Query2*

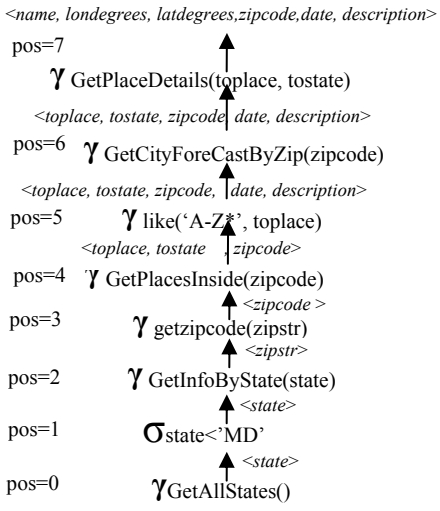


Fig. 11 Central *Plan2a*

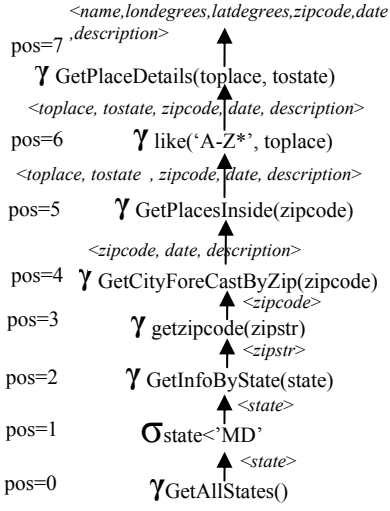


Fig. 12 Central *Plan2b*

Fig. 7 shows modified query fragments using the independent plan functions *PF4* and *PF5,* which both depend on the parallelized *PF1*. For a given zipcode *PF4* calls *GetPlacesInside* and *PF5* calls *GetCityForeCastByZip*. The modified adaptive parallel plan in Fig. 8 uses the *PAP* operator to parallelize the calls to the independent plan functions *PF4* and *PF5*. Fig. 9 shows the parallel process tree. In contrast to the process tree in Fig. 6**,** the web service operations *GetPlacesInside* and *GetCityForecastByZip* are called parallel at level two.

For the initial central plan the query processor uses a simple heuristic web service cost model based on the signatures of web service operations assuming that web service operations are expensive. One such possible

central execution plan for *Query2* is illustrated by *Plan2a* in Fig. 11. However, since the web service operations *GetPlacesInside* and *GetCityForeCastByZip* are independent, they can be called in any order. Fig. 12 shows the alternative execution *Plan2b* where, in contrast to *Plan2a* the calls to the web service operations *GetCityForeCastByZip* and *GetPlacesInside* have been swapped. Fig. 10 shows the two fragment plan functions *PF6* and *PF7* used in both plans. *PF6* calls the web service operation *GetPlacesInside*, and a filtering function *like* while *PF7* calls the web service operation *GetPlaceDetails*. Fig. 13 and Fig. 14 illustrate the two parallel plans.

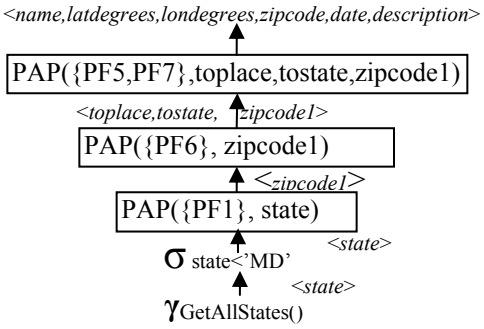


Fig. 13 Parallel *Plan2a*

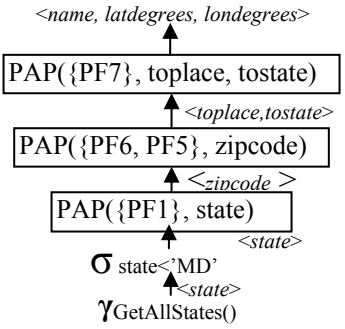


Fig. 14 Parallel *Plan2b*

4 Adaptive Parallelization

First the details of *PAP* are discussed. Then experiments with different dependent and independent strategies of using *PAP* are analyzed. The full pseudo code of *PAP* is shown in [10].

4.1 PAP Operator

The *PAP* operator calls one or several plan functions in parallel. It has the signature:

```
PAP(Vector of Function vpf, Stream pstream ,
    Vector argorder, Vector resorder)→Stream res
```

The arguments of the plan functions f_i in *vpf* to execute are provided through the input stream *pstream*. For each input tuple in *pstream PAP* starts processes executing all f_i in parallel in a round robin fashion. Each input tuple p in *pstream* provides arguments for all f_i. However, different f_i use different parameter values in p. The parameter *argorder* specifies for each f_i how to form the arguments of f_i from p. It is a vector of vector of argument

positions *{{a_{ij},...}...}*, that specifies per f_i the parameter positions *{a_{ij},...}* to pick from *p*. For example, in Fig. 8 the uppermost call to *PAP* has *argorder = {{1},{1}}* because both *PF4* and *PF5* take as argument the first element of the input tuple <*zipcode1*>.

Each result tuple *r* emitted from *PAP* consists of values r_k. The *PAP* parameter *resorder* specifies how to compute r_k from results of f_i. It is a vector of pairs *{{p_{km},c_{km}}...}*, that specifies per element position *k* in *r i)* the position p_{km} of the function f_m in *vpf* that computed r_k, and *ii)* which element c_{km} in the result from f_m to select as r_k. In Fig. 8 *resorder = {{1,1},{1,2},{1,3},{2,1},{2,2}}* specifying the result tuple <*toplace, tostate, zipcode, date, description*>.

A child result tuple is delivered back to the parent asynchronously as soon as its plan function f_i has produced a new value. *PAP* stores each received child result in an input buffer per child. When *PAP* has received at least one result ruple from every f_i for a given input tuple *p* the system will emit one or several result tuples based on the *resorder* and cartesian product of the result tuples received from each f_i. Once a child has no more result tuples to emit it terminates. When the parent receives a termination message from a child, it starts another child process for the plan function in *vpf* to be called next picking its parameter tuple from the current input tuple. When there are no pending parameter tuples in *pstream* and no still running children, *PAP* is finished.

In a process tree, the *fanout* is defined as the number of children processes below a parent query process. A process tree for the execution plan in Fig. 8 is shown in Fig. 9, where every node has fanout two. First the coordinator *q0* has started two children *q1* and *q2,* each executing the same plan function *PF1*. In the next call to *PAP* the plan functions *PF4* and *PF5* are independent. Therefore a call to *PAP* is executed in each of *q1* and *q2* with different plan functions *PF4* and *PF5,* respectively. *PAP* in *q1* has created a binary sub-tree with children *q3* and *q4,* while *q2* has the children *q5* and *q6*. The query processes *q3* and *q5* are started with *PF4* while *q4* and *q6* are started with *PF5*.

Once started *PAP* dynamically modifies the process tree at run time. The query process locally monitors the execution times of its children to locally add new children to improve performance until no more performance improvement is expected. *PAP* does the following:

1. It initially forms a process tree by having *fanout* set to the length of *vpf.* The fanout is minimally two to ensure parallelism when length of *vpf* is one, as in *q0*. This is called the *init* stage.
2. A *monitoring cycle* for a non-leaf query process is defined as when *PAP* has received end-of-call messages from all its children and the total number of received result tuples is at least one. After the first monitoring cycle *PAP* adds *p* new child processes, initially *p=2*. This is called the

add stage. In Fig. 15, the query process *q0* has added two new processes *q7* and *q10* at level 1 compared to Fig. 9.

3. When an added node has several levels of children the init stages of the children's *PAP*s are rerun. That is, *q7* adds *q8* and *q9* while *q10* adds *q11* and *q12*.
4. *PAP* records per monitoring cycle *i* the average computation time t_i to produce an incoming tuple from the children. This time is dominated by the latency of the encapsulated web service operations.
    a. If t_i decreases more than a threshold (set to 20%) the add stage is rerun.
    b. If t_i increases no more children are added.

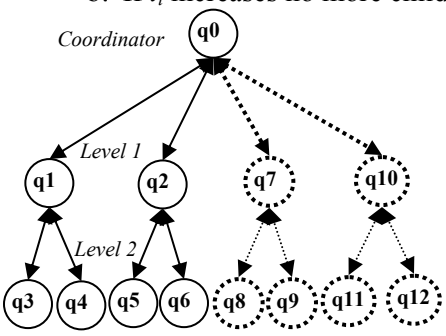


Fig. 15 Add stage

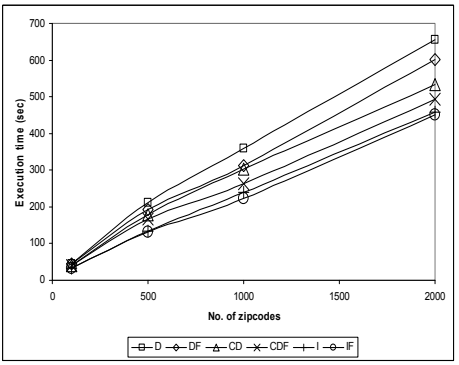


Fig. 16 Experiments with adaptive strategies

4.2 Experiments

Experiments were run under Windows XP on an HP Compaq 530 with a 3 GHz single processor Intel Pentium 4 and 2.5GB RAM. We compared the query execution times for Q*uery1* using six different strategies:

1. *Dependent (D)*: Strategy *D* is a naïve dependent strategy as in Fig. 5. This corresponds to the adaptive parallelization in [11]. All the web service operations in the query are considered as dependent calls, even the independent ones. A new sub-tree is always started with fanout two, which is increased by two by the adaptation.
2. *Dependent with varying initial fanout (DF):* Strategy *DF* measures the impact of varying initial fanout for a dependent strategy. This is as strategy *D*, but new sub-trees are started with the same fanout as the current adapted fanout of its siblings. The fanout of the first child of a level is two.
3. *Cached dependent (CD)*: Strategy *CD* measures impact of caching results from the web service operations for a dependent strategy. It

modifies strategy *D* by caching results of operation calls. For example, in Fig. 5 the result of calling the operation *GetCityForecastByZip* for a given zip code is cached in a main memory table. Whenever the operation *GetCityForecastByZip* is required to be called in the query, the cache table is checked to avoid redundant calls.

4. *CD with DF (CDF)*: The impact of caching combined with varying initial fanout is investigated for a dependent strategy.
5. *Independent (I)*: Strategy *I* measures naïve independent calls for the execution plan in Fig. 8. A new sub-tree is always started with fanout equal to the number of plan functions in *vpf* of the *PAP* call. The fanout is two if *vpf* has length one.
6. *Independent with varying initial fanout (IF):* This is as strategy *I,* but new sub-trees are started with the same fanout as the current adapted fanout of its siblings.

The experiments were made by scaling *Query1* by selecting an increasing number of states. This produces an increasing number of zipcodes and increases the cardinality of the result.

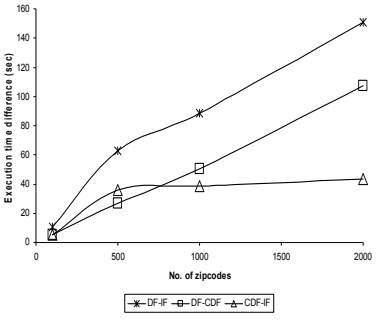


Fig. 17 Relative scalability

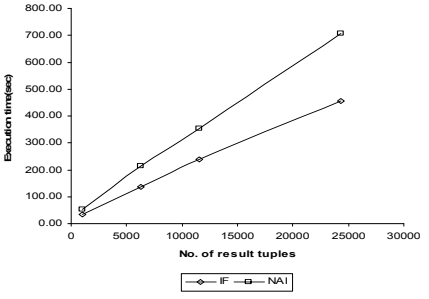


Fig. 18 Impact of adaptation

Fig. 16 shows that strategy *D* is slowest, and *DF* is somewhat faster. *CD* is even faster, showing that caching is favorable since web service calls incur high latency. *CDF* is even better as it combines caching and adaptive initial fanout. However, even the naïve *PAP* strategy *I* is faster than all variants of the dependent strategies. Strategy *IF* is best. Caching does not pay off for independent strategies, since no redundant calls are made; therefore the combination of caching with IF was not measured.

Fig. 17 shows the relative scalability comparing independent and dependent strategies and caching. *DF-IF* plots the performance difference between *DF* and *IF*. It shows that the independent strategy *IF* scales better. Analogously, *DF-CDF* shows for dependent strategies that caching improves scalability. *CDF-IF* shows that the best independent strategy *IF*

scales somewhat better that the best dependent strategy *CDF*. However, unlike *CDF*, *IF* requires no extensive memory for caching.

To investigate the impact of adaptation we devised another strategy *Non Adaptive Independent*(*NAI*). It is similar to *I*, but fanout is fixed to two in all levels of the process tree. Fig. 18 shows that *IF* outperformed *NAI*.

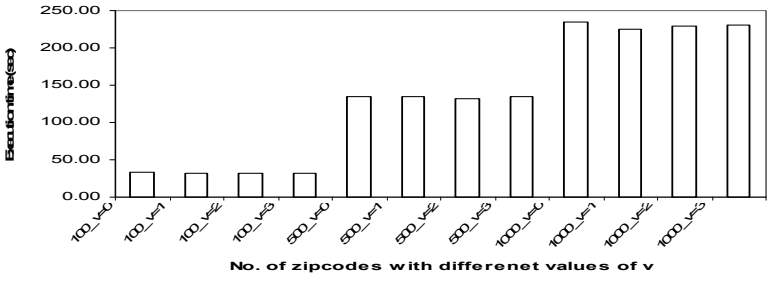


Fig. 19 Execution time with further increased fanouts

When the average computation time t_i is less than a specified threshold (set to 20%) *PAP* increases its number of children *cf* with a constant increment *i*. For the above experiments *i=2* which was measured optimal for adaptive dependent calls [11].

To investigate the impact of increasing *i* gradually we made experiments by increasing *i* as: $i = i + v$ *(v=0,1,2…)*

The value of *v* is incremented until no significant performance improvements are measured. Fig. 19 shows that incrementing with *v* does not make any significant performance improvement for *PAP*.

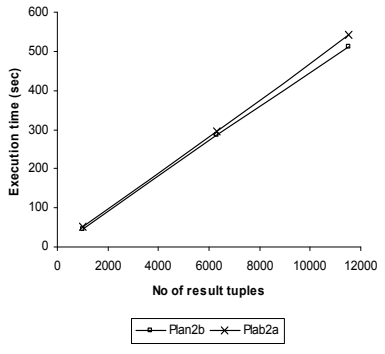


Fig. 20 Impact of central plan execution order

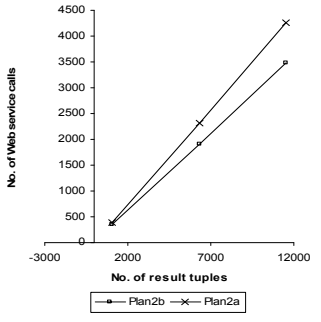


Fig. 21 Number of web service calls- Pla2a Vs Plan2b

To investigate the performance of *PAP* for different central plans, experiments were made for *Query2* scaled by selecting an increasing number of states and places. Fig. 20 shows that *Plan2b* (Fig. 13) outperforms parallel

Plan2a (Fig. 14). The reason is that *PF5* and *PF6* are independent, which is reflected in parallel *Plan2b* but not in parallel *Plan2a*. This causes parallel *Plan2a* to make redundant web service calls to *PF5* for each place returned by *PF6*.

The central optimizer will not know that *Plan2b* is better when parallelized. Therefore WSMED reorders the central preliminary plan by co-locating the predicates that directly depend on each. Fig. 21 compares the number of web service calls made by the two different parallel plans.

5. Related Work

PAP generalized *AFF_APPLYP* [11] by parallelizing both dependent and independent web service operation calls, while *AFF_APPLY* produced and parallelized pipelined plan of dependent calls.

WSQ/DSQ [7] handles high-latency calls to web search engines by launching asynchronous materialized sub-queries later joined in the execution plan using a special operator without any adaptation. In contrast, WSMED adaptively produces multi-level parallel plans based on streams of parameter tuples passed to parallel sub-plans.

WSMS [12] proposed a cost-based approach for pipelined parallelism among web service operation calls to minimize the query execution time. By contrast, we parallelize adaptively calls to web service operations without any cost model. Furthermore, *PAP* adaptively parallelizes the same web service operation by starting several query processes.

In [8] run time adaptation of buffer sizes in web service calls is investigated, not dealing with adaptive parallelism on web service calls at the client side.

In [1] an approach is described for optimizing web service compositions by traversing ActiveXML documents to select relevant embedded web service calls. It identifies only the independent sub-queries having web service operations and calls them in parallel. The parallelization required a static cost model. By contrast, *PAP* adaptively parallelizes plan functions with both dependent and independent web service calls without any cost model.

Eddies [2] dynamically reorder query processing operators by an n-ary tuple router interposed between *n* data sources and a set of query processing operators. Rather than routing, *PAP* adaptively parallelizes calls to parameterized operators (plan functions) for different parameter values. The purpose of eddies is to avoid dependencies between operators, while the purpose of *PAP* is to speed up calls to individual plan functions. *PAP* complements eddies.

Like two-phase query optimization in distributed databases [9] WSMED also parallelizes a query execution plan based on an initial central query plan. However, the strategy in [9] is based on a static cost model for distributed databases, while WSMED adaptively parallelizes dependent joins. Furthermore, WSMED reorders the preliminary plan for better parallel performance by co-locating adjacent dependent plan function calls.

Starting query processes with plan functions and the parameter tuple shipping phase of *PAP* has some similarity with the map phase of *MAPREDUCE* [6]. However, *MAPREDUCE* is not dynamically adapting query execution plans as *PAP* and is not streamed.

6. Conclusion

WSMED provides general relational query capabilities over any data providing web service operations given their WSDL meta-data descriptions. Queries are expressed in SQL over automatically generated relational views over the data providing web service operations.

Without any cost knowledge the WSMED query processor automatically and adaptively finds an optimized parallel execution plan calling the queried data providing web services. The algebra operator *PAP* locally adapts the parallel plan by adding children, until no significant performance improvement is measured, based on monitoring the flow between query processes. The operator handles queries where data providing web service operations are called both dependently and independently. *PAP* substantially improves the query performance without any cost knowledge or extensive memory usage compared to other strategies. The measurements are all made with publicly available web service operations.

To lower the number of web service operation calls WSMED includes a strategy to co-locate adjacent dependent plan functions.

The WSMED approach relies on calling side effect free data providing web service operations. The widely available WSDL language does not provide meta-data describing side effects in web service operations. When such a standard is available WSMED can utilize it to guarantee query correctness.

WSMED is accessible through a URL [14] from anywhere without installing any software.

Acknowledgments This work is supported by the Swedish Foundation for Strategic Research under contract RIT08-0041 and Sida.

References

1. S.Abiteboul, et al., Lazy query evaluation for active XML, In: International Conference on Management of Data, pp 227- -238, ACM Press, New York(2004)
2. Avnur, R., Hellerstein, J.M.:Eddies: Continuously adaptive query processing, In Proceedings of International Conference on Management of Data , pp 261- -272, ACM Press, New York, (2000)
3. codeBump, GeoPlaces web service, http://codebump.com/services /PlaceLookup.asmx
4. codeBump, Zipcodes web service, http://codebump.com/services/ZipCodeLookup.asmx
5. CYDNE, http://ws.cdyne.com/WeatherWS/Weather.asmx?WSDL
6. Dean, J., and Ghemawat, S. 2008.MAPREDUCE: Simplified Data Processing on Large Clusters, Communications of the ACM, pp 107- -113, ACM Press, New York, (2008)
7. Goldman, R. and Widom, J.:WSQ/DSQ: a practical approach for combined querying of databases and the Web. In: International Conference on Management of Data , pp 285- - 296,ACM Press, New York, (2000)
8. Gounaris, A., Yfoulis, C., Sakellariou, R., and Dikaiakos, M.D.:Robust Runtime Optimization of Data Transfer in Queries Over Web Services. In: International Conference on Data Engineering, pp 596- -605, IEEE, (2008)
9. Hasan, W. :Optimization of SQL queries for Parallel Machines, Springer-Verlag.(1997)
10. PAP Operator, http://user.it.uu.se/~msabesan/PAP/PAP.pdf
11. Sabesan, M. and Risch, T.:Adaptive Parallelization of Queries over Dependent Web Service Calls. In Proceedings of First IEEE Workshop on Information & Software as Services., pp 1725- -1732, IEEE computer society (2009)
12. Srivastava, U., Widom, J., Munagala, K., and Motwani, R.: Query Optimization over Web Services. In: Proceedings of Very Large Database Conference, VLDB Endowment, pp 355- -366, (2006)
13. USZip, http://www.webservicex.net/uszip.asmx
14. WSMED Demo, http://udbl2.it.uu.se/WSMED/wsmed.html

Paper IV

The paper is reformatted for typographic consistency.

Automated Web Service Query Service

Manivasakan Sabesan[1], Tore Risch[1], and Feng Luan[2]

[1]Uppsala Database Laboratory
Department of Information Technology
Uppsala University
Uppsala, Sweden

[2]Database Systems
Department of Computer and Information Science,
Norwegian University of Science and Technology,
Trondheim, Norway
[1]{Manivasakan.Sabesan, Tore.Risch}@it.uu.se, [2]luan@idi.ntnu.no

Abstract. A data providing web service returns a collection of objects for given parameters without any side effects. The Web Service MEDiator (WSMED) system automatically provides relational views of data providing web service operations by reading their WSDL documents. These views can be queried with SQL. A common pattern in queries over data providing web services is that the output of one web service call is the input for another. A challenge addressed by WSMED is to speed up such queries by parallelization. To automatically achieve the optimal parallel plan WSMED adapts an initial parallel plan locally in each node until optimized performance is achieved. To make any data providing system into a data providing web service WSMED includes a web service generator that automatically deploys the web service operations required to access a data source. Given that interface functions are implemented the web service generator automatically deploys corresponding web service operations and generates the WSDL document. The web service generator is used also for defining the web service interface to WSMED itself. The WSMED web service operations provide SQL query functionality to query and join any data providing web services. Search of any data providing web service from a browser can be done by a JavaScript program that directly calls the WSMED web service without any need for installing software.

1 Introduction

Web services are often used for search computing (Ceri, 2009) where data is retrieved from servers providing information of different kinds. Such *data providing web services* return collections of objects for a given set of parameters without any side effects. A System, *Web Service MEDiator (WSMED),* is built that provides a web service to query any data providing web service operations without any further programming. The search is completely specified by SQL queries that retrieved data from the data providing web services. WSMED adheres to the *Everything as a Service* (XaaS) paradigm[11] by providing a general web service to process queries over other web services, known as the WSMED web service.

WSMED can import any WSDL file and automatically generate relational views for the web service operations defined in the WSDL file. These views can be queried and joined with SQL. For a given SQL query, WSMED dynamically composes the web services, optimizes the web service calls, and adaptively parallelizes the execution plan.

As an example, consider a query to find information about places located within 15 km from each city whose name starts with 'Atlanta' in all US states. Three different data providing web services can be used for answering this query, using the operations *GetAllStates*[6] to retrieve all the states, *GetPlacesWithin*[6] to get all the places located within a given distance, and *GetPlaceList*[10] to provide all the places whose names start with 'Atlanta' for a given state.

WSMED assumes that all queried data sources are available as web services. The conventional way to define a new data providing web service for a data source which is *not* a web service is manual development of software to access the data source, defining a WSDL document to describe the interface, and deploying the interface code.

To facilitate the provision of a data providing system as a web service, WSMED includes a subsystem, the *web service generator,* which automatically generates the web service operations to access a data source. The programmer first defines data source interface functions to access the data source as queries using the extensible wrapper/mediator system Amos II (Risch et al., 2003). Once the interface functions are defined, the WSMED web service generator automatically generates the corresponding web service operations and dynamically deploys them without restarting the web server. The signature of each so generated web service operation is defined in an automatically generated WSDL document based on the signatures of the interface functions. The WSDL document completely describes the web service interfaces of the deployed operations. Each operation calls the interface function and sends back the result as a collection. Interface functions have been defined for many different kinds of data sources[3], e.g. relational DBMSs, semantic web data, topic maps, and CAD servers.

Even the WSMED web service itself is generated by the web service generator. An automatically generated WSDL document[14] describes the interface of the WSMED web service operations.

Queries calling web services often have a similar pattern where the output of one web service call (e.g. *GetAllStates*) is the input for another one (e.g. *GetPlacesWithin*), i.e. the second call is dependent on the first one, etc. A challenge here is to develop methods to speed up queries requiring such *dependent* data providing web service calls. In general such speed-ups are based on some unknown web service properties. Those properties are not explicitly available and depend on the network and runtime environments when and where the queries are executed. It is very difficult to base

execution strategies on a static cost model in such scenarios, as is done in relational databases.

In our approach a web service call is considered as an expensive function call where the result is a data collection. To improve the response time, WSMED uses an approach to parallelize the web service calls while keeping the dependencies among them. With the approach separate *query processes* are started automatically in parallel, each calling a parameterized sub query plan, called a *plan function,* for given parameters. Each plan function encapsulates one web service call and makes data transformations such as flattening nested results, filtering, and data conversions.

To provide a view query-able with SQL, the nested result collections are flattened. Conversely arguments of operation are nested. A common constraint is that input parameters have to be bound in operations and WSMED will decompose the query plan so guarantee this.

The performance is often improved by setting up several web service calls to the same operation in parallel rather than to call the operations in sequence for different parameters. The algebra operator, *AFF_APPLYP* (Adaptive First Finished Apply in Parallel), takes a stream of parameter values and, for each received parameter tuple in the stream, ships a plan function in parallel to other query processes and then asynchronously receives the results from the shipped plans in parallel.

Multi-level execution plans are automatically generated with several layers of parallelism in different query processes. This forms a *process tree* for the query. During execution *AFF_APPLYP* first initiates the communication with its child query processes and then ships the plan function to the children. Then the *AFF_APPLYP* operator starts shipping in parallel to the children the argument tuples from the parameter stream. At any point in time every process in the tree executes one plan function for a specific parameter. The results from the children are delivered to the parent in parallel as streams.

WSMED adaptively achieves an optimized process tree by local run-time monitoring of each plan function call. For the adaptation *AFF_APPLYP* dynamically modifies a parallel plan locally and greedily in each query process.

The functionality of WSMED is demonstrated through a publicly accessible web interface[13]. It enables the user to query any data providing web service. SQL views of the queried web services are automatically generated, given its WSDL URL. General SQL queries over the views can be specified. The schema of the generated views can be inspected. The demonstration is fully implemented as a JavaScript program calling the WSMED web service using SOAP, without any need to download or install any software.

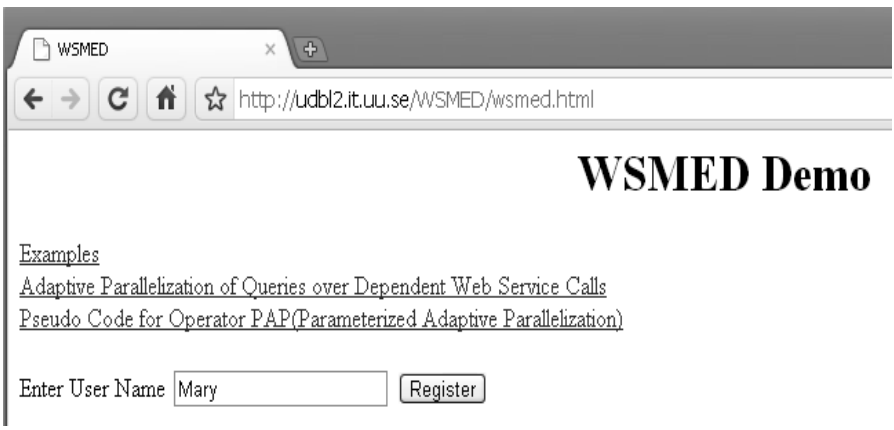
In summary the contributions of our work are:

1. The WSMED system provides general SQL query capabilities over any data providing web services based on their WSDL meta-data descriptions.
2. For a given SQL query, the system automatically and adaptively generates and optimizes a parallel execution plan calling the web services.
3. A web service generator automatically generates web service interfaces for data sources once they have been defined as interface functions.
4. The generated web service operations are dynamically deployed without restarting web server.

Section 2 describes the WSMED on-line demo. Section 3 overviews the WSMED system architecture. The WSMED service generator is described in Section 4. In Section 5 we show how WSMED processes in parallel SQL queries and adaptively parallelize their execution automatically. Related work is discussed in section 6. Finally Section 7 concludes our approach.

2 The WSMED demo

Figure. 1 User registration



The WSMED on-line demonstration illustrates the functionality of the WSMED web service. It demonstrates all web service operations provided by WSMED through a user interface that can be run in any web browser without software installation. The web page is written as an application program in JavaScript that directly calls the WSMED web service. The communication between the JavaScript program and the WSMED web service operations uses the SOAP protocol.

A user first starts a WSMED session (Figure. 1) by registering her name, for example *Mary*, and then clicks on the '*Register'* button. A WSMED session is closed with the '*Exit'* button (Figure. 2).

Before querying a web service she has to import its metadata by entering its WSDL URL, e.g. http://terraservice.net/TerraService2.asmx?WSDL in a text box with label '*Enter New WSDLURL'* (Figure. 2). Alternatively a predefined set of WSDL URLs for common services is provided by the pull-down menu labelled '*Available WSDL URL's*. The WSDL file is selected by pressing the '*ImportWSDL'* button.

Figure. 2 Enter WSDL URL

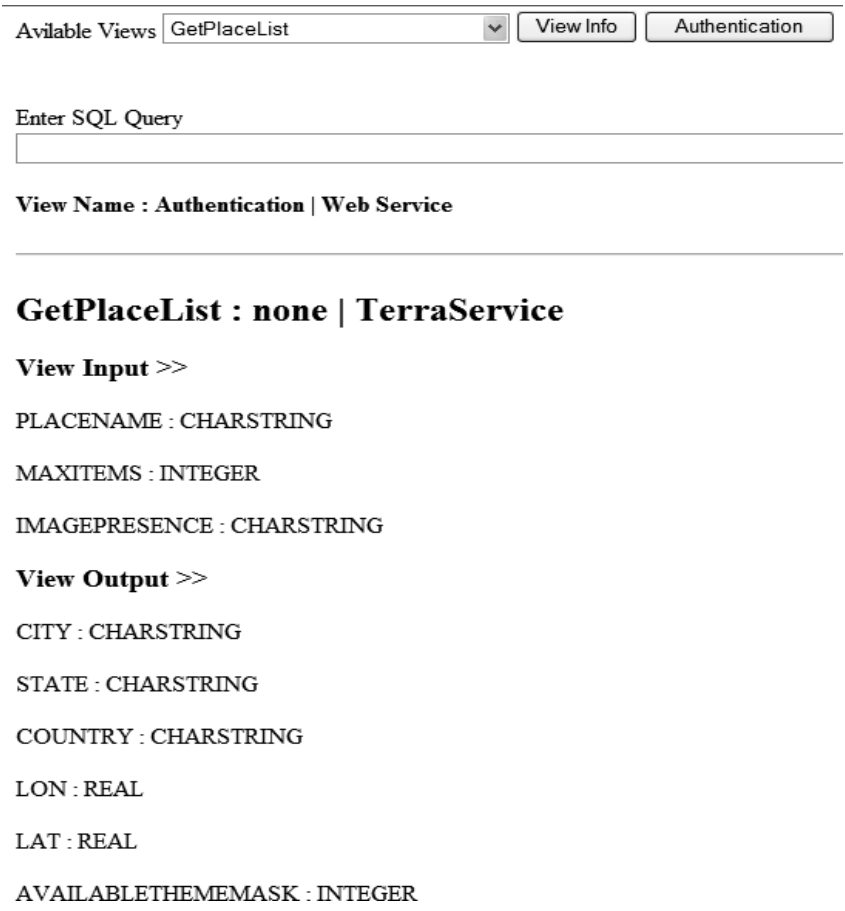
After meta-data of a WSDL URL is imported the system automatically generates SQL views of all web service operation specified by the WSDL file.

Figure. 3 Get generated SQL Views

Figure. 3 shows the imported SQL views of the web service *TerraService*. The names of the views are based on the names of imported web service operations. They are displayed in the format: '*View: Authentication* |

Service'. *View* is the name of a view. *Authentication* indicates whether authentication is required when querying the web service operation defining the view. It may be one of the strings *required, none,* or *builtin*. *Service* is the name of the web service that supports the operation over which the view is defined. The currently imported SQL views can be selected in a pull-down menu labeled '*Available Views'*. For example, the user can inspect the details of the generated view named *GetPlaceList* by selecting the view in the pull-down menu (Figure. 4) and clicking '*View Info*'.

Figure. 4 View information



As illustrated in Figure. 4 this will display the view name, its authentication status, the web service hosting the operation encapsulated by the view, the data types of its attributes, and what attribute values are required to be known in order for the view to be queried. That information is important for the user to express a correct SQL query. To inspect the authentication status of an available view, the user selects it from the pull-down menu and then presses the 'Authentication' button. A new authentication value (for example

345DERT56TY) can be entered in the text box labeled '*Enter New Authentication Value'* and stored by pressing the '*Enter'* button as shown in Figure. 5.

Figure. 5 New authentication value

As shown in Figure. 6 in the text box labeled '*Enter SQL Query*' an SQL query can be expressed in terms of the available views and executed by clicking the '*Execute'* button. Figure. 6 shows the result of an SQL query:

```
select gl.City ,gl.Lon
from   GetPlaceList gl
where  gl.placeName='Atlanta' and gl.MaxItems=100 and
       gl.imagePresence= 'true'
```

Figure. 6 Execute SQL Query

3 The WSMED system

Figure. 7 illustrates the WSMED architecture. It contains four subsystems: the *WSMED query processor*, the *WSMED coordinator,* the *WSMED web server*, and the *web service generator*. The WSMED query processor provides general SQL query capabilities to any data providing web service. It accepts SQL queries using its web service interface managed by the WSMED web server. The WSMED web server extends the lightweight standalone server Quick Server[9], to send and receive SOAP messages using the HTTP protocol. The WSMED coordinator manages user sessions starting a separate WSMED query processor for each user. The query processor is terminated when the user ends the session.

The purpose of the web service generator is to generate web service operations for a data source which is *not* implemented as a web service. First a programmer implements interface functions to access the data source. Then the web service generator automatically deploys the web service operations to call the interface functions. The basic functionality of WSMED itself is implemented as interface functions. Therefore all web functionality provided by WSMED is automatically deployed as web service operations using the web service generator. The automatically generated document *wsmed.wsdl* describes these operations.

Figure. 7 WSMED architecture

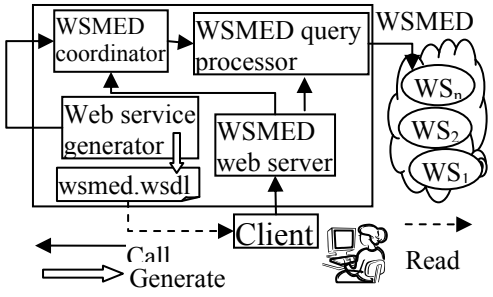


Figure. 8 Service oriented architecture of WSMED

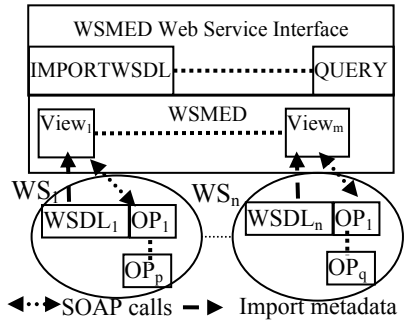


Figure. 8 illustrates the web service operations of the WSMED web service. The top box illustrates the supported web services operations *IMPORTWSDL*, *QUERY*, etc. They are all implemented as interface functions and automatically deployed by the web service generator. The following WSMED web service operations are defined:

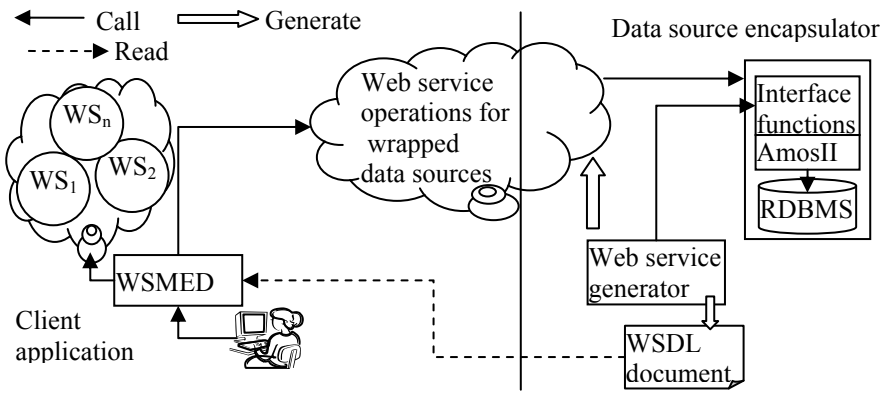
- The *INIT* operation registers a WSMED user session.
- For a given a URL the *IMPORTWSDL* operation imports WSDL meta-data information and automatically creates an SQL view $View_i$ for each operation OP_j provided by a web service WS_k described by an imported WSDL document $WSDL_k$.

- The *AUTHENTICATION* operation provides authentication information for web service operations that so require.
- The *VIEWINFO* operation provides information about the SQL view over a given web service operation. For example, it lists view attributes that must always be specified in the queries.
- The system accepts SQL queries to the generated views by the *QUERY* operation. The results from the operation is automatically flattened, optimized, and post processed by WSMED in order to deliver a proper SQL result as a collection of tuples.
- Finally, the operation *EXIT_S* terminates a user session.

4 Automated web service generation

In Figure. 9 the WSMED *web service generator* dynamically deploys web service operations for data. The web service generator calls a *data source encapsulator* to obtain the signatures of the interface functions to be provided as web service operations. Based on these signatures it deploys the web service operations for the functions. It also generates a *WSDL document* describing the deployed web service operations. The WSDL document can be read by WSMED for querying the encapsulated data source. In Figure. 9 a relational database is encapsulated.

Figure. 9 Automated web service generation



The *interface functions* are defined as parameterized queries to the mediator/wrapper system Amos II(Risch et al., 2003) Different kinds of data sources can be made queryable by Amos II by implementing *wrappers* that interface the Amos II kernel with the systems providing the data in a source. For example, wrappers have been built to query relational databases (Fahl et al., 1997), semantic web data (Petrini et al., 2007), topic maps (Stefanova et al., 2008), or CAD systems (Koparanova et al., 2002).

In addition, Amos II provides a built-in database that can be populated with source data as an alternative to defining the interfaces function by

wrapping an external data source system. For example, Figure. 10 illustrates how a text file[15] is represented in Amos II as a stored function (table) *getzipc*. The database is populated by reading the text file.

To deploy interface functions as web service operations by the web service generator the system function *deploy_wsdl* is called. It takes as arguments the names of the interface functions, the name of the deployed web service, and the name of the WSDL file describing the web service operations. In the example, the *deploy_wsdl* call creates a WSDL document *zip.wsdl* to describe the exported interface function *getzipc*.

Figure. 10 Deploying a data source

```
create function getzipc(Charstring state)→Charstring zipcode
 as stored;
deploy_wsdl({'getzipc'},'zipcode','zip.wsdl');
```

WSMED itself is also regarded as an encapsulated data source, which is automatically deployed by the web service generator. The web service interface to WSMED is defined using interface functions. For example Figure. 11 shows the signature of the interface function implementing WSMED's web service operation *QUERY*. The interface function *query* is defined in terms of many other functions and external programs to process an SQL query *sqlq* by a user identified by the parameter *userid*.

Figure. 11 Signature of interface function *query*

```
query (Integer userid, Charstring sqlq)-> Bag of Charstring
```

4.1 The Web service generator

The web service generator in Figure. 12 consists of four sub modules, the *AmosII* mediator/wrapper system, the *function analyzer*, the *WSDL creator*, and the *WSDL exporter*.

The *function analyzer* is called by *deploy_wsdl* and receives a set of exported functions. It then queries the meta-data of Amos II for the signature of each function to export and generates a data structure, *exported signatures,* that describes them. An exported signature consists of the names and types of a function's arguments and results. They are passed to the *WSDL generator*. The WSMED web server does not need to restart when exporting and publishing new functions as web service operations since it dynamically looks up the signatures of interface functions at run time when web service operation calls are received.

The signature of an interface function is automatically translated into a corresponding message structure in WSDL. To produce the WSDL document, the WSDL generator dynamically builds an *internal export description* as a DOM data structure in main memory using the WSDL4J[12] Java toolkit. The rules for transforming signatures to WSDL operation

descriptions will be discussed in Section 4.2. The *WSDL exporter* then transforms the DOM representation of the export description into a WSDL document that describes the exported function interfaces as web service operations.

Figure. 12 The web service generator

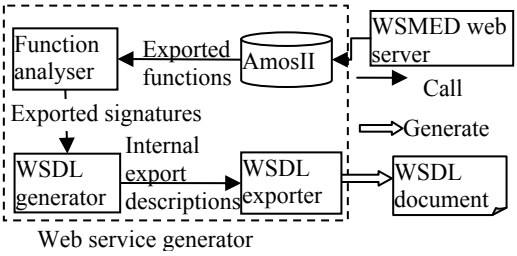
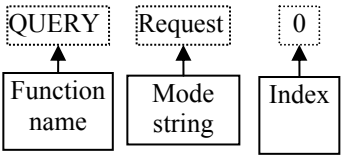


Figure. 13 The structure of the input element



4.2 Publishing a web service operation

Figure. 14 shows the WSDL document representing the interface function *query* (Figure. 11) as a web service operation named *QUERY*. In general, the web service operation is defined in the *portType* WSDL element. The operation contains an input element and an output element. The input element has an XML attribute *message* named, e.g. *QUERYRequest0*. Figure. 13 illustrates how the message name is constructed. It is a concatenation of the name of the web service operation and a *mode* string (‘*Request’* or *‘Response’*) indicating whether it is an input or and output message. The *index* number is appended to translate overloaded interface functions into uniquely named messages.

A request message has the same number of *part* elements as the number of arguments in the interface function. Our example interface function *query* has two arguments named *userid and sqlq* with types *int* and *string* respectively.

A response messages always has one part named *results* representing the result of an interface function. In the example query the result is a set (bag) of strings, which is represented in WSDL as a sequence of type *string*. The type of *results* is a concatenation of the web service operation name (here *QUERY)* and an index to handle overloaded interface functions.

Figure. 14 WSDL for interface function *query*

```
<wsdl:definitions ...>
  <wsdl:types>
    <xsd:schema>
      <xsd:complexType name="QUERY0">
       <xsd:sequence>
         <xsd:element name="R1" type="xsd:string">
         </xsd:element>
       </xsd:sequence>
```

```
      </xsd:complexType>
    </xsd:schema>
  </wsdl:types>

  <wsdl:message name="QUERYResponse0">
   <wsdl:part name="results" type="tns:QUERY0" />
  </wsdl:message>

  <wsdl:message name="QUERYRequest0">
    <wsdl:part name="USERID" type="xsd:int" />
    <wsdl:part name="SQLQ" type="xsd:string" />
  </wsdl:message>

 <wsdl:portType name="WSMEDPortType">
  <wsdl:operation name="QUERY"
                  parameterOrder="USERID SQLQ">
   <wsdl:input name="QUERYRequest0"
    message="tns:QUERYRequest0"/>
   <wsdl:output name="QUERYResponse0"
     message="tns:QUERYResponse0" />
  </wsdl:operation>
 </wsdl:portType>

<wsdl:binding name="WSMEDSoapBinding"
       type="tns:WSMEDPortType">
  <wsdlsoap:binding style="rpc"
   transport="http://schemas.xmlsoap.org/soap/http" />
 <wsdl:operation name="QUERY">
  <wsdlsoap:operation soapAction="" />
   <wsdl:input name="QUERYRequest0">
    <wsdlsoap:body use="encoded"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
     namespace="urn:WSAmos" />
   </wsdl:input>
   <wsdl:output name="QUERYResponse0">
     <wsdlsoap:body use="encoded"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
    namespace="urn:WSMED" />
   </wsdl:output>
  </wsdl:operation>
</wsdl:binding>

<wsdl:service name="WSMEDservice">
  <wsdl:port name="WSMEDPort"
          binding="tns:WSMEDSoapBinding">
     <wsdlsoap:address location=
"http://130.238.11.96:8082/wsmed/service/WSMEDServlet" />
  </wsdl:port>
 </wsdl:service>
</wsdl:definitions>
```

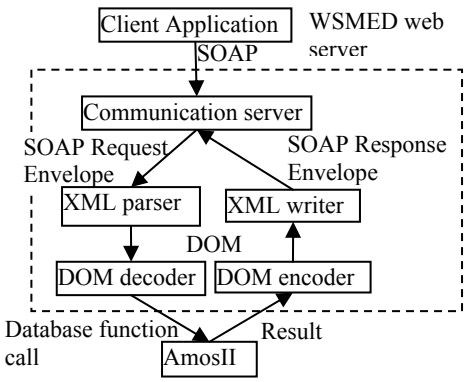
4.3 The WSMED web server

The WSMED web server is a server that uses the HTTP protocol to communicate SOAP messages. The WSMED web server immediately services the interface functions as web service operations once they are exported, without need for restarting the WSMED web server or deploying any additional server site code.

Figure. 15 illustrates the WSMED web server. It consists of a *communication server*, an *XML parser* and *writer*, a *DOM decoder*, and *an encoder*. The *communication server* first receives a remote call from the client application. The remote call is a RPC SOAP call via the HTTP protocol. The communication server extracts the message content and passes it to the XML parser. The *XML parser* uses the input SOAP envelope to generate a DOM data structure.

The *DOM decoder* converts the DOM data representation of a SOAP message into a call to an interface function. The XSD data types of a receiving message are converted from DOM to the format required by interface functions. Then the DOM decoder calls Amos II to execute the interface function. After receiving the results from the function, the DOM encoder uses the signature of the function and data type mappings between XML and Java to build a result DOM structure. The *XML writer* passes the result DOM structure to the communication server as a SOAP response message and the communication server sends back the SOAP message to the client application over the HTTP protocol. A modified *JSoapServer*[7] is used as the communication server. JSoapServer is a lightweight standalone SOAP web server using the *QuickServer* library for building web services.

Figure. 15 WSMED web server



5 The WSMED query processor

To improve the query performance the WSMED query processor automatically produces a parallel multi-level execution plan with several layers of parallelism and forms a *process tree* to execute the parallel plan. Each node in the process tree executes some of the web service calls. The

query parallelization is performed in two phases as illustrated in Figure. 17. In *phase1* a central query execution plan is created from the given SQL query, and in *phase 2* the central plan is automatically transformed into a parallel query plan by the *parallel plan creator*. In this section we will explain how the parallelization is done, illustrated with an example SQL query to be presented next.

5.1 Example SQL query

The example *Query* in Figure. 16 finds information about places located within 15 km from each city whose name starts with 'Atlanta' in all US states. In the query we utilize the web service operations *GetAllStates, GetPlacesWithin,* and *GetPlaceList.* For a given web service WSMED automatically generates *Operation Wrapper Functions (OWF)* (Sabesan et al., 2009) that represent *SQL views* of the web service operations based on the WSDL definitions. In Figure. 16 the three generated OWFs *GetAllStates, GetPlacesWithin,* and *GetPlaceList* are defined to encapsulate web service operations with the same names. The query returns a stream of 360 result tuples and invokes more than 300 web service calls.

Figure. 16 Example Query

```
select    gl.place,gl.state

From      GetAllStates gs, GetPlacesWithin gp, GetPlaceList gl

where       gs.State=gp.state and gp.distance=15.0 and
            gp.placeTypeToFind='City' and
            gp.place='Atlanta'
            and gl.placeName=gp.ToPlace+' ,'+gp.ToState and
            gl.MaxItems=100 and gl.imagePresence='true'
```

The OWF *GetAllStates* presents information of US states as a set of tuples <*Name*, *Type*, *State*, *LatDegrees*, *LonDegrees*, *LatRadians*, *LonRadians*>. However, we are only interested in the values of the attribute *State*. The OWF *GetPlacesWithin* returns a set of tuples <*ToCity, ToState, GeoPlaceDistance_Distance*> for given place (*'Atlanta'*), state (*gs.State*), distance (*15.0*), and kind of place type to find (*'City'*). The OWF *GetPlaceList* retrieves a set of places <*placename, state, country, placeLon, placeLat, availableThemeMask, placeTypeId*, *population*>, given a specification of a place (concatenation of *ToCity+','+ToState*), the maximum number result tuples (*100*), and a flag indicating whether places having an associated map are returned.

5.2 Parallelizing web service operation calls

Figure. 17 illustrates the query processor in WSMED. It parallelizes the queries in two phases. In *Phase1* a non parallel plan is created from the given SQL query. The parallel plan is produced in *Phase2* based on the non parallel plan. The calculus generator produces from the SQL query an internal calculus expression in a Datalog dialect (Litwin et al., 1992). For example, *Query* is transformed into the following calculus expression:

```
Query1(place,state) ← GetAllStates(_,_,st1,_,_,_,_) AND
   GetPlacesWithin("Atlanta",st1,15.0,"City",tp,ts,_) AND
 GetPlaceList(pn,100, "true", place, state,_,_,_,_,_,_) AND
 Concat(tp," ,",ts,pn)
```

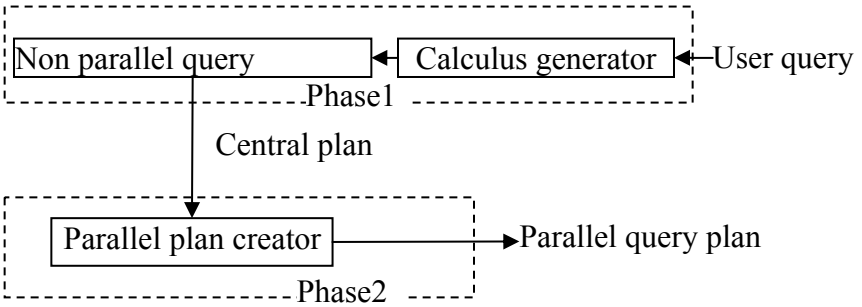
The symbol '_' represents an anonymous result variable. With non-parallel query optimization the calculus expression is translated by the non-parallel query optimizer into the algebra expression in Figure. 18. This is the central plan for the example query. It is a left-deep tree of executable predicates enumerated from 0 and up. The algebra expression contains calls to the *apply* operator γ, which applies a plan function for a given parameter tuple. The central query execution plan with γ can be directly interpreted but with bad performance, since the web service operations are applied in sequence. For the initial central plan the non-parallel plan optimizer uses a simple heuristic web service cost model based on the signatures of web service operations assuming that web service operations are expensive.

The plan first calls the operation *GetAllStates* returning a stream of tuples <*st1*>. Each of these tuples are fed to the next operation *GetPlacesWithin* called by the apply operator with the given argument tuple *('Atlanta', st1, 15.0, 'City')* returning a stream of tuples <*tp, ts*>. The built in function *Concat* is then applied on each argument tuple *(tp,',',ts)* producing a stream of strings *pn*. Finally the operation *GetPlaceList* is applied on each argument tuple *(pn,100,'true')* returning a stream of tuples <*place, state*>.

The *parallel plan creator* calls the *parallelize* algorithm in Figure. 20 to automatically transform the central plan *NP* into a parallel one. For example, it translates the central plan in Figure. 18 to the parallel one in Figure. 19.

The parallelization produces parallel web service calls by inserting an algebra operator *AFF_APPLYP* in the execution plan whenever a call to a web service operation is encountered. The *AFF_APPLYP* operator calls a plan function in parallel and adaptively modifies the process tree to improve performance. For each web service operation call a plan function is generated by the parallel plan creator to encapsulate a fragment of the non parallel query plan embodying the web service operation call. Section 5.3 explains the functionality of *AFF_APPLYP* in detail.

Figure. 17 WSMED query processor



The parallel plan preserves the dependent execution order of the encapsulated web service calls. First *pos* is identified as the split point where the first parallelizable web service operation call is found (*line 2*). The *AFF_APPLYP* operator requires as input a stream of parameters. Therefore the identified operation call must have at least one input parameter. In the example in Figure. 18 the first split point is identified as the predicate applying the web service operation *GetPlacesWithin* where *pos*=1. If there is no split point the central plan is not parallelized.

Figure. 18 Central query plan

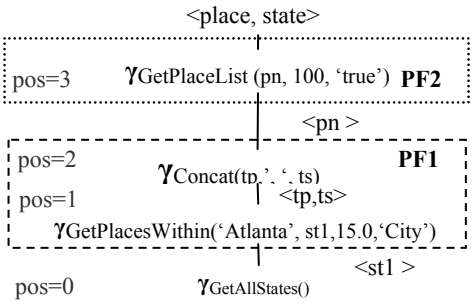


Figure. 19 Parallel query plan

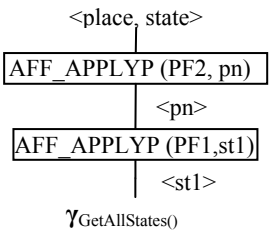


Figure. 20 Parallelize algorithm

parallelize(NP) → P
input: NP –central query plan
output: P– parallel query plan

1. *len* ← number of predicates in *NP*
2. *pos* ← Find the first position of a web service call in NP with number of inputs > 0
3. **if** *pos* exists **then**
  3.1 *subplan1* ← create a sub plan with all the predicates between the positions *0* and *(pos-1)* in *NP*.
  3.2 *temp-subplan* ← create another sub plan with all the predicates between positions *pos* and *len* in the *NP*.

3.3. *subplan2* ← parallelize(*temp-subplan*)
3.4. *P*←rewrite-plan(*subplan1*, *subplan2*)
4. **Else**
4.1 *P*←NP
5. **return** *P*

Then a plan function *subplan1* is created (*line 3.1*) with the predicates between the positions *0* and *(pos-1)*. Another plan function *temp-subplan* (*line 3.2*) is created with the rest of the predicates (the predicates between the positions *pos* and *len*) of *NP*. The example *subplan1* calls the web service operation *GetAllStates*. The *temp-subplan* contains the remaining calls to *GetPlacesWithin* , *concat*, and *GetPlaceList*.

To find the second parallelizable web service operation call, *parallelize* is recursively called with *temp-subplan* (*line 3.3*) as argument. The *rewrite-plan* (*line 3.4*) procedure modifies *subplan1* so that the result from the recursive call, *subplan2* is called in parallel for each result from *subplan1*. This is done by inserting an *AFF_APPLYP* operator to encapsulate *subplan2*.

In the example, a second recursive call of *parallelize* creates *subplan1=PF1* and *subplan2=PF2*. The final parallel plan *P* (Figure. 19) contains the *AFF_APPLYP* operators to adaptively parallelize the calls to *PF1* and *PF2*.

5.3 Adaptive Apply in Parallel - AFF_APPLYP

The algebra operator *AFF_APPLYP* (Adaptive First Finished Apply in Parallel) (Sabesan et al., 2009) has the signature:

```
AFF_APPLYP (Function pf, Stream pstream) → Stream result
```

The pseudo code for AFF_APPLYP is listed in Figure. 21.

Figure. 21 AFF_APPLYP algorithm

AFF_APPLYP(fn, pstream) → result
input: fn : plan function *pstream* : a stream of parameter values for *fn*
output: result : Stream of result tuples from children

fanout ← 2

number of query processes added after each monitoring cycle: *p* ← 1

number of query processes: *nq* ← *0*

number of tuples: *tc* ← *0*

number of end-of-call messages: *ack* ← *0*

time required to retrieve a tuple (time per tuple): *tupt* ← *0*

opt: flag indicates whether adaptive expansion of fanout is started (opt=true) or stopped (opt=false): *opt*←false

```
        stopping threshold, change in tupt per cycle: threshold_value← 0.25
        execution time to process fn in children processes per cycle: exet ← 0
        while  (pstream is not empty )
            Initialize a query process to execute the plan function fn with arguments
            taken from pstream
            nq ← nq+1
            while (nq = fanout)
               res ←  retrieve the result tuple from a child process
               if (res is a valid result)
                   tc ← tc+1
                   emit res as the result of AFF_APPLYP
10             else if (res is end-of-call message)
11                 nq ← nq -1
12                 ack ← ack+1
13                 exet ← exet + execution time of the child process to execute fn
14                 if (ack =fanout) and (tc >0)
15                   pre_tupt←tupt
16                   tupt ← (exet / tc)
17                   if (pre_tupt >0)
18                       relative_error← ((pre_tupt–tupt)/(pre_tupt))
19                   end if
20                   if (((threshold_value < relative_error) and (not opt)) or
                     (pre_tupt = 0))
21                       fanout ← fanout + p
22                   else if (threshold_value >= relative_error)
23                           opt ← true
24                       end if
25                   end if
26                end if
27                if (ack =fanout)
28                   ack ← 0;  exet ← 0;  tc ← 0;
29                end if
30             end if
```

```
31              end if
32            end while
33          end while
34          while (nq > 0)  /* some child process left to be finished * /
35           res ← retrieve the result tuple from a child process
36           if (res is a valid result)
37             emit res as the result of AFF_APPLYP
38           else if (res is end-of-call message)
39                 nq ← nq -1
40                 end if
41            end if
42          end while
```

The algebra operator *AFF_APPLYP* first starts *fanout* (initially 2, *line 1*) children processes (threads) with plan function *pf*. Then it starts picking parameter tuples (*line 3*) one by one from *pstream*, to send down to the children. When the all children have received one round of parameter tuples (*line 5*), *AFF_APPLYP* is ready to receive results. The result stream *result* from the children is delivered back to the parent asynchronously as soon as a child process has produced a new value. When a result tuple is received from some child it is directly emitted as a result of *AFF_APPLYP* (*line 9*). Once a child completed the processing of a plan function for a given parameter tuple in *pstream* it terminates. When the parent receives a termination message (*line 10*) from a child, it will start another new child process with the same plan function *pf* and passes the next pending parameter tuple (*line 3*) from *pstream* to the new child. When there are no pending parameter tuples in *pstream* (*line 2*) and no pending children (*line 34*), *AFF_APPLYP* is finished. The same procedure is repeated recursively in all child processes for each call to *AFF_APPLYP*.

The parallel execution plan is generated by the WSMED query processor (coordinator *q0*, Figure. 22*)*. It first generates a central plan (e.g. Figure. 18) containing calls to the web service operations. The *parallelize* algorithm is then called by the coordinator to produce a parallel query plan (e.g. Figure. 19) from the non-parallel one. Once the parallel plan is started the calls to *AFF_APPLYP* will automatically start new parallel processes to form a process tree (e.g. in Figure. 22). A query process can have an arbitrary number *(fanout)* of child processes. All the children on the same level execute the same plan function but with different parameters.

Figure. 22 gives an example of a process tree generated by the WSMED query processor for the example query in Figure. 16. The parallel plan for the example query contains two calls to *AFF_APPLYP*. First the coordinator *q0* calls *AFF_APPLYP* that generates a binary tree with two nodes *q1* and *q2*. A call to *AFF_APPLP* is executed in each of *q1* and *q2*. Thus *AFF_APPLYP* in *q1* creates a binary sub tree with children *q3* and *q4* while *q2* creates the children *q5* and *q6*.

The plan function in the coordinator *q0* encapsulates the web service operation call *GetAllStates*, while the plan function *PF1* of the processes in level one (*q1* and *q2*) encapsulates the web service operation call *GetPlacesWithin* for different states. On level two (*q3, q4, q5, q6*) the plan function *PF2* calls the web service operation call *GetPlaceList* for different place specifications.

Process *q0* starts the children *q1* and *q2* with the plan function *PF1*. Analogously, each *AFF_APPLYP* executing in level one processes starts the children *q3, q4, q5*, and *q6* with plan function *PF2*. The query processes in level two delivers a stream of tuples containing *placename* and *state* to the plan functions on level one that executes the web service operation *GetPlacesWithin* for each received tuple. The *AFF_APPLYP* operator in level one finally delivers the result stream to the coordinator process *q0*.

Figure. 22 Parallel process tree

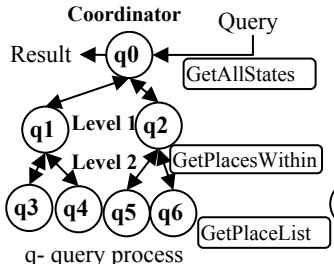


Figure. 23 Add stage

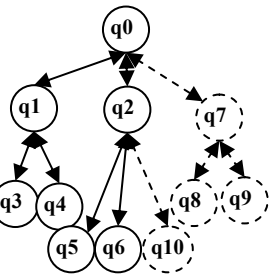
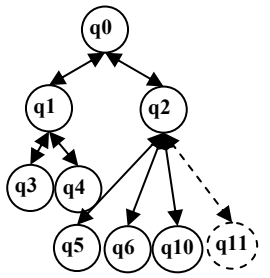


Figure. 24 Add and drop stage



Once started, *AFF_APPLYP* dynamically modifies the initially binary process tree at run time. The query process locally monitors the execution times of its children to locally add or delete children to improve performance until no more performance improvement is expected. Consider the binary process tree in Figure. 22.

AFF_APPLYP does the following:

1. It initially forms a binary process tree by initially having *fanout = 2*. This is called the *init stage*.
2. A *monitoring cycle* for a non-leaf query process is defined as when *AFF_APPLYP* has received end-of-call messages from all its children and the total number of received result tuples is at least 1 (*line 14*) . After the first monitoring cycle (*line 20* when *pre_tupt*=0)

AFF_APPLYP adds (*line 21*) *p* new child processes. Adding new processes is called an *add stage.* In Figure. 23, *p=1* and therefore query process *q0* adds one new process *q7* at level 1, while *q1* and *q2* add *q10* and *q11* at level 2, respectively.

3. When an added node has several levels of children the init stages of the children's *AFF_APPLY*s will produce binary sub–trees. That is, *q7* adds *q8* and *q9*.
4. AFF_APPLYP records per monitoring cycle *i* the average time t_i (*line 16*) to produce an incoming tuple from the children.
    a. If t_i decreases (*line 20*) more than a threshold (set to 25%) the add stage is rerun.
    b. If t_i increases (*line 22*) no more children are added. This is indicated as *opt=true* (*line 23*). As an option a *drop stage* is run that drops one child and its children (the drop stage is not shown in the algorithm).

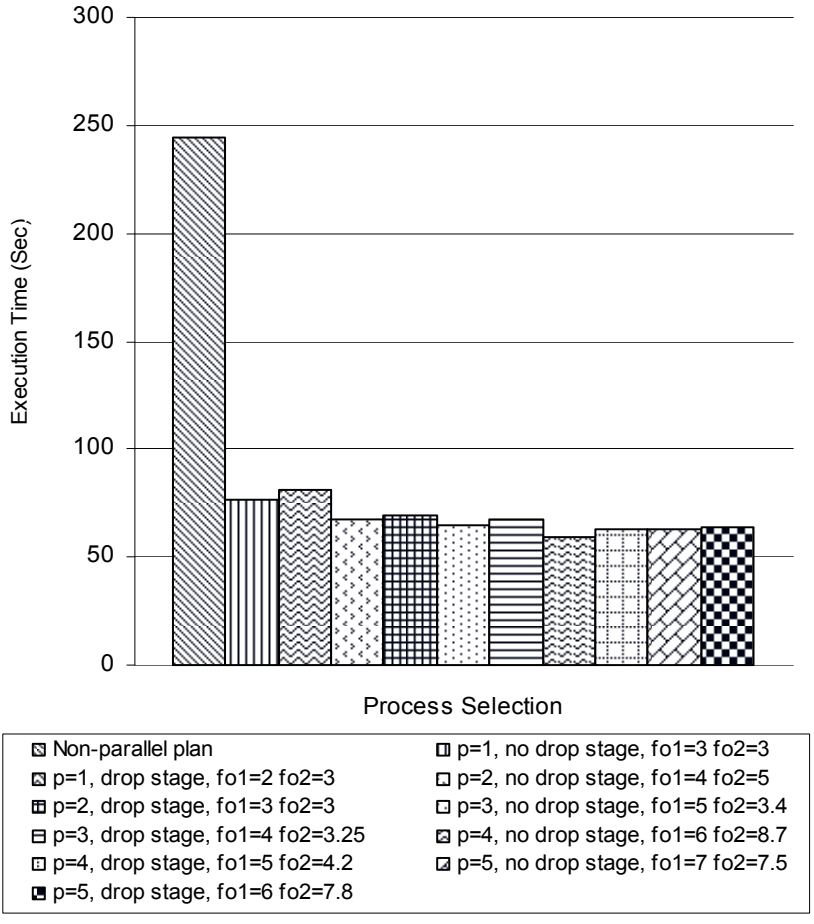
In Figure. 24, *q2* adds *q11,* while *q0* drops *q7,* and *q7* drops *q8* and *q9*.

5.4 Experimental results

For our example query, we experimented with different values of *p* (number of query processes added after each monitoring cycle) and different change thresholds, with and without the dropping query processes when an optimum point is reached. The average fanouts of the process trees are measured. The results for 25% change thresholds are shown in Figure. 25. We concluded that execution (59.07 sec) time *AFF_APPLYP* performed best (4 times faster) when comparing with the sequence web service invocation (244.394 sec). Further the execution time with *p=4* and no drop stage performed best and execution time with *p=2* and no drop stage also showed closer performance (88%) with the best execution time. Dropping processes make insignificant changes in the execution time.

In general the execution time of a web service operation is not known in prior. *AFF_APPLYP* is therefore a better approach to reach optimal execution time than having a traditional static cost model.

Figure. 25 Comparisons of naïve and adaptive approaches



6. Related work

WSQ/DSQ (Goldman et al., 2000) handles high-latency calls to web search engines by launching asynchronous materialized dependent joins later joined in the execution plan using a special operator. In contrast, WSMED produces non-blocking multi-level parallel plans based on streams of parameter tuples passed to parallel sub plans without any materialization.

WSMS (Srivastava et al., 2006) proposed an approach for pipelined parallelism among dependent web services to minimize the query execution time. By contrast, we parallelize by partitioning parameter tuple streams. Furthermore, WSMS didn't propose any adaptive parallelization, lacked support for code shipping, and couldn't make parallel calls to the same web service. In contrast we propose a strategy to adaptively produce a

parallelized plan where *AFF_APPLYP* invokes parameterized plans calling web services in parallel.

The plan function and parameter tuple shipping phase of *AFF_APPLYP* is similar to the map phase of *MAPREDUCE* (Dean et al., 2008). However, *MAPREDUCE* is not dynamically adapting query execution plans as *AFF_APPLYP* and is not streamed.

In (Gounaris et al., 2008) run time adaptation of buffer sizes in web service calls is investigated, not dealing with adaptive parallelism on web service calls at the client side.

A reference model for dynamic web service composition is described in the D-WSCS system (Eid et al., 2008). The monitoring module of D-WSCS is responsible for monitoring and showing the status of the composed services at runtime. When a composite service fails, D-WSCS is looping back to call the same service or find an alternative service. Similarly WSMED dynamically composes web service operations to answer an SQL query and call the operation again if it fails. Unlike D-WSCS, WSMED is handling adaptive parallelization of web service calls.

Parallel execution scheduling strategies that require static costs are discussed in Taniar et al. (1999), Taniar et al.(2003) and Taniar et al.(2008). In contrast WSMED is using adaptive parallelization that is independent of static costs of web service calls.

The formal basis for using views to query heterogeneous data sources is reviewed in (Ullman, 1997). *Chocolate* (Josifovski et al., 2003) extends the federated database capabilities of *DB2/UDB* by automatically creating views of web services from WSDL descriptions, similar to the OWF generation in WSMED. However, unlike WSMED, Chocolate does not deal with adaptive parallelization of the web service calls.

Query as a Web Service[8] allows users to create queries and publish them as web services similar to the WSMED web service generator. However, WSMED is more general by providing SQL query service to any data providing web services based on reading the web services' WSDL documents.

Apache Axis[4] supports *JAVA2WSDL* APIs to create WSDL documents for Java methods. Apache Axis can be plugged into web servers such as *Tomcat*[5] to access Java methods as web service operations. In contrast to WSMED's web service generator, Tomcat needs to be restarted and the servlet code recompiled every time a new web service operation is deployed. The WSMED web service generator automatically generates an interface function as a web service operation with a simple command.

Similar to WSMED's web service generator, Oracle (Das et al., 2009) supports access to databases as web services. Java proxy classes that correspond to database operations are first generated. The wrappers are compiled and deployed in an Oracle application server. In contrast to Oracle, WSMED's web service generator doesn't need any proxy classes and

dynamically deploys new web service operations based on interface function signature.

The Amazon Relational Database Service[1] web service provides relational databases in the cloud using the Amazon SimpleDB[2] that provides a subset of SQL. The WSMED web service generator can deploy web service operations for any RDBMS or other wrapped data sources.

7. Conclusion

WSMED provides a general relational query service over data providing web services given their WSDL meta-data descriptions. Queries are expressed in SQL to dynamically join data providing web services. WSMED is accessible through a URL (WSMED Demo) from anywhere without installing any software.

The WSMED query processor automatically and adaptively finds an optimized parallel execution plan calling the queries data providing web services. The algebra operator *AFF_APPLYP* locally adapts the parallel plan by adding and removing children until an optimum is reached, based on monitoring the flow between query processes. It is shown to improve query performance substantially compared with a central plan.

AFF_APPLYP can handle parallel query plans for a query with any number of dependent joins. We plan to generalize the strategy for queries mixing both dependent and independent web service calls. Further we need to investigate different process arrangement strategies with the algebra operator *AFF_APPLYP*.

The WSMED service generator provides web service operations for data sources once they have been wrapped as interface functions. The web service generator automatically generates the WSDL document to describe the interface functions. The generated web service operation is dynamically deployed without restarting the web server and without writing any server side code.

Acknowledgments This work is supported by the Swedish Foundation for Strategic Research under contract RIT08-0041 and Sida.

References

Ceri, S. (2009) ´Search Computing', *Proceedings of the International Conference on Data Engineering*, pp. 1-3.

Das, T., Maring, S., Sapir, R., and Wiesenberg, M. (2009) 'Oracle Database Web Services', *http://download.oracle.com/docs/cd/B28359_01/java.111/b31225.pdf*.

Dean, J. and Ghemawat, S. (2008) 'MAPREDUCE: Simplified Data Processing on Large Clusters', *Communications of the ACM,* Vol. 51, No. 1, pp 107-113.

Eid, M., Alamri, A. and El Saddik, A. (2008) 'A reference model for dynamic web service composition systems', *Int. J. Web and Grid Services*, Vol. 4, No. 2, pp.149–168.

Fahl, G. and Risch, T. (1997) 'Query Processing over Object Views of Relational Data', *VLDB Journal* , Vol. 6, No. 4 , pp 261-281.

Goldman, R. and Widom, J. (2000) 'WSQ/DSQ: a practical approach for combined querying of databases and the Web', *Proceedings of the International Conference on Management of Data* , pp. 285-296.

Gounaris, A., Yfoulis, C., Sakellario R.and Dikaiakos, M.D(2008) 'Robust Runtime Optimization of Data Transfer in Queries Over Web Services'. *Proceedings of the International Conference on Data Engineering*, pp. 596-605.

Josifovski, V., Massmann, S. and Naumann, F. (2003) 'Super-Fast XML Wrapper Generation in DB2: A Demonstration', *Proceedings of the International Conference of Data Engineering* (ICDE 2003), pp. 756 -758.

Koparanova, M. and Risch, T. (2002) 'Completing CAD Data Queries for Visualization', *Proceedings of the International Database Engineering and Applications Symposium (IDEAS 2002)*, pp 130 – 139.

Litwin, W. and Risch,T. (1992) 'Main Memory Oriented Optimization of OO Queries using Typed Datalog with Foreign Predicates', *IEEE Transactions on Knowledge and Data Engineering*, Vol. 4, No. 6, pp. 517-528.

Petrini, J. and Risch, T. (2007) 'SWARD: Semantic Web Abridged Relational Databases', *Proceedings of the 6th International Workshop on Web Semantics*, pp 455-459.

Risch, T., Josifovski, V. and Katchaounov, T. (2003) 'Functional Data Integration in a Distributed Mediator System', *Functional Approach to Data Management - Modeling, Analyzing and Integrating Heterogeneous Data*, pp. 211-238.

Sabesan, M. and Risch, T. (2009) 'Adaptive Parallelization of Queries over Dependent Web Service Calls'. *Proceedings of the International Conference on Data Engineering (ICDE2009)*, pp.1725-1732.

Stefanova, S. and Risch, T. (2008) 'Viewing and Querying Topic Maps in terms of RDF', *Proceedings of the SEMMA2008: First International Workshop on Semantic Metadata Management and Applications*.

Srivastava, U., Widom, J., Munagala, K. and Motwani, R. **(** 2006) 'Query Optimization over Web Services', *Proceedings of the Very Large Database Conference*, pp. 355- 366.

Taniar, D. and Leung, C.H.C. (1999) 'Query execution scheduling in parallel object-oriented databases', *Information & Software Technology*, Vol. 41, No. 3, pp 163-178

Taniar, D. and Leung, C.H.C. (2003) 'The impact of load balancing to object-oriented query execution scheduling in parallel machine environment', *Information Sciences*, pp. 33-71.

Taniar, D., Leung, C.H.C. and Wenny Rahayu, J., Goel, S. (2008) 'HighPerformance Parallel Database Processing and Grid Databases' *John Wiley& Sons.*

Ullman, J.D. (1997) 'Information Integration Using Logical Views'. *Proceedings of the*: *International Conference on Database Theory*, pp. 19- 40.

Notes

1. Amazon Relational Database Service, http://aws.amazon.com/rds/.
2. Amazon SimpleDB, http://aws.amazon.com/simpledb/.
3. AmosII wrappers, http://user.it.uu.se/~udbl/amos/wrappers.html.
4. Apache Axis, http://ws.apache.org/axis/.
5. Apache Tomcat, http://tomcat.apache.org/.

6. GeoPlaces, http://codebump.com /services /PlaceLookup.asmx.
7. JSoapServer, http://jsoapserver.sourceforge.net/.
8. Query as a Web Service, http://help.sap.com/businessobject/product_guides/boexir31/en/xi3-1_query_as_a_web_service_en.pdf.
9. QuickServer, http://www.quickserver.org/.
10. TerraService, http://msrmaps.com/TerraService2.asmx.
11. The Next Wave: Everything as a Service, http://www.hp.com/hpinfo/execteam/articles/robison/08eaas.html.
12. WSDL4J, http://sourceforge.net/projects/wsdl4j/.
13. WSMED Demo, http://udbl2.it.uu.se/WSMED/wsmed.html.
14. WSMED WSDL, http://udbl2.it.uu.se/WSMED/wsmed.wsdl.
15. ZCTAs (ZIP Code Tabulation Areas), http://www.census.gov/tiger/tms/gazetteer/zcta5.txt.