# Web Service Query Service

Manivasakan Sabesan
Uppsala DataBase Laboratory
Dept. of Information Technology
Uppsala University
Sweden
msabesan@it.uu.se

Tore Risch
Uppsala DataBase Laboratory
Dept. of Information Technology
Uppsala University
Sweden
Tore.Risch@it.uu.se

## ABSTRACT

A data providing web service returns a set of objects for a given set of parameters without any side effects. We demonstrate a system, *WSMED*, which provides a web service that can process SQL queries over any data providing web services. A challenge addressed by WSMED is to develop methods to speed up such queries by parallelization. WSMED automatically generates a distributed execution plan that calls web services in parallel. A common pattern in queries over web services is that the output of one web service call is the input for another. To speed-up such queries, WSMED automatically parallelizes the web service calls by starting separate *query processes,* each managing a parameterized sub-query, a *plan function*, for different parameters. To automatically achieve the optimal parallel process tree WSMED adapts an initial parallel plan locally in each query process until an optimized performance is achieved. The demonstration is a web interface to a WSMED server. It allow to make SQL queries joining any data providing web services, thus demonstrating that WSMED provides general search of composed web services.

## Keywords

Adaptive parallelization, Web service query service, Search computing.

## 1. INTRODUCTION

Web services are often used for search computing [1] 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. A System, *Web Service MEDiator (WSMED),* is built that provides a web service to compose any data providing web service operations. It automatically provides relational views of any data providing web service operation. 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.



*iiWAS2009*, December 14–16, 2009, Kuala Lumpur, Malaysia.



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* from the web service *GeoPlaces*[2] to retrieve all the states, *GetPlacesWithin* from *GeoPlaces*[2] to get all the places located within a given distance, and *GetPlaceList* from *TerraService*[8] to provide all the places starts with 'Atlanta' for a given state.

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 nested 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 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.

The WSMED can import any WSDL file and automatically generate relational views for each web service operation defined in the WSDL file. To provide a view query-able with SQL, the result collections are flattened. Similarly operation arguments are also flattened but have to be bound in queries and WSMED will try to decompose the query plan so that this is the case.

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. 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 the *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 children. Then AFF_APPLYP starts shipping in parallel to the children the argument tuples from the input 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 URL [10]. It enables the user to access any data provided web service. The schema of the generated views can inspected and the query can execute general SQL queries over the views. The demonstration is fully implemented as a JavaScript calling a WSMED server 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 data providing web services by reading WSDL meta-data description.
2. For a given SQL query, the system automatically and adaptively generates and optimizes a parallel execution plan calling the web services.

Section 2 explains the WSMED demo. Section 3 overviews the WSMED system architecture. In Section 4 we present an example of an SQL query with dependent web service calls, for which WSMED automatically generates an optimized process tree. Related work is compared in Section 5. Finally Section 6 concludes our approach

## 2. The WSMED Demo

Figure 1 illustrates the WSMED demo. It demonstrates all web service operations provided by WSMED through a user interface that can be run in any web browser. The JavaScript completely implementing the user interface can be inspected from the browser. The communication between the JavaScript program and WSMED is completely based on SOAP.

A user needs to start a WSMED session by registering any user name, for example *me*(1), and click on the *Register* button(2). Then she can import the metadata of the web services to query by selecting a WSDL URL of a web service from the drop-down-list(3) with WSDL URLs known so far and pressing the *ImportWSDL* button(4). Also she can make a new WSDL URL known by selecting *Enter New WSDL* option from the drop-down-list(3) and enter a new WSDL URL in text box(5) and finally clicking the *Enter*(6)button. This will add the new WSDL URL to the drop-down-list(3).

When meta-data of a WSDL URL is imported, the SQL views are automatically created for all web service operation specified by the WSDL file. The imported SQL views as shown in the *Result Display*(14). The names of the views are based on the names of imported web service operations. They are displayed in the format:

*Table Name --- Authentication<<< Web Service.*

*Table Name* is the name of a view. *Authentication* refers the whether authentication is needed to invoke a web service operation and it may be *required* or *none* or *builtin*. *Web Service* is the name of the web service operation over which the view is defined. All currently available SQL views are present in the drop-down-list(7). A user can inspect the schema of a given view in the Result Display area(14) by selecting a view name from the drop-down-list(7) and then pressing the *Table Info*(8) button. 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 attributes are required to be known to query the view. That information is vital when a user is expressing an SQL query. To view the authentication status of an available table, a user selects an available table from the drop-down-list(7) and then presses the *Authentication* button(9). A new authentication value can be entered in the text box(10) and stored by pressing the *Enter* button(11). Any SQL query can be expressed with the available tables in the text box(12) and executed by clicking the *Execute* button(13). The result of a query is showed in result display area(14). A WSMED session quitted with the *Exit* button(15).

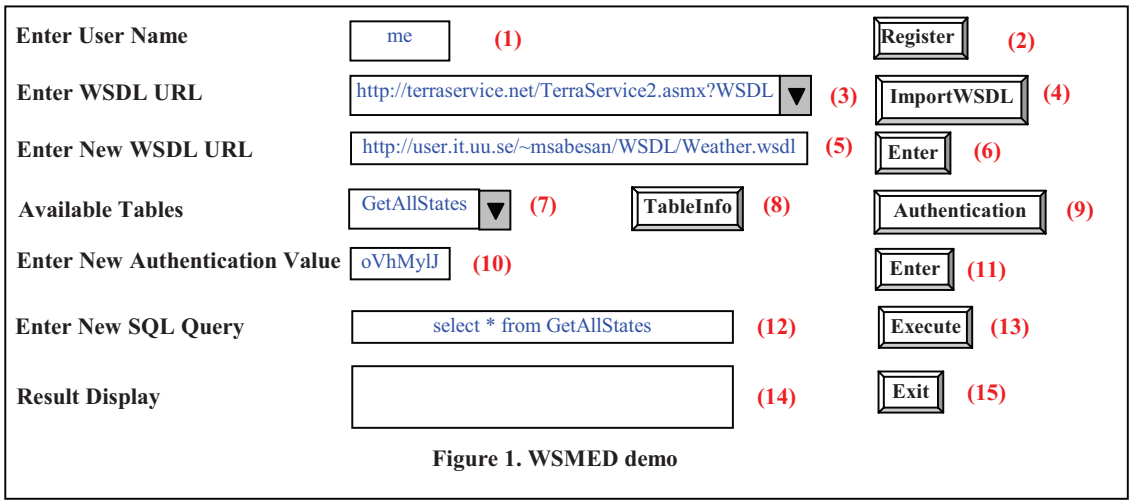


**Figure 1. WSMED demo**

## 3. The WSMED System

Figure 2 illustrates the service oriented architecture of WSMED. Following the *Everything as a Service* (XaaS) paradigm [9], WSMED is providing a web service to query arbitrary data providing web services.

- The *AUTHENTICATION* operation provides authentication information for web service operations that so require.
- The system accepts SQL queries to the generated views by the *QUERY* operation. The results from the operation is automatically flattened and post processed by WSMED in order to deliver a proper SQL result.

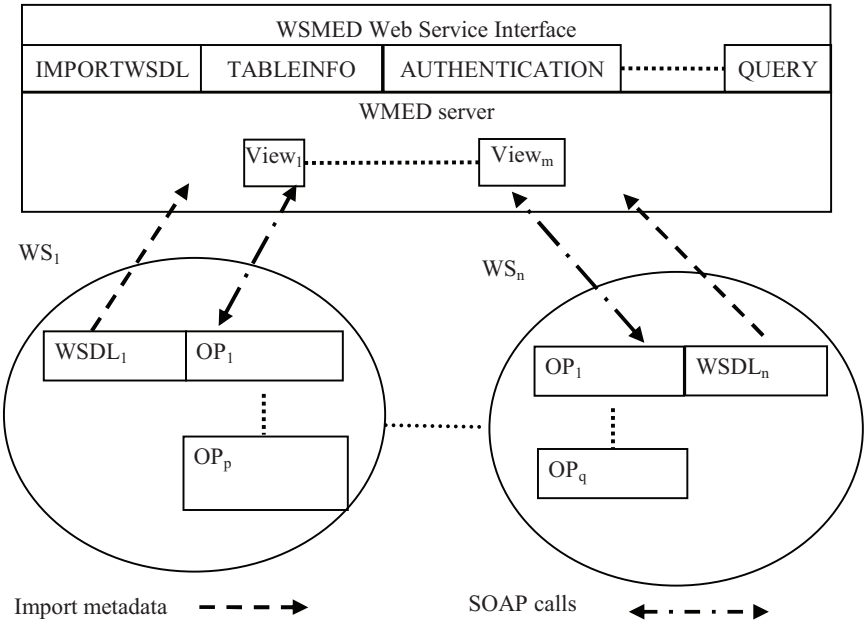


**Figure 2. Service oriented architecture of WSMED**

WSDL file system exports several web service operations[11]:

- For a given a URL the *IMPORTWSDL* operation imports WSDL meta-data information and automatically creates SQL views $View_i$ for every operation $OP_j$ provided by a web service $WS_k$ described by an imported WSDL document $WSDL_l$.
- The *TABLEINFO* operation provides information about the SQL view over a given web service operation. For example some table attributes must always be known when querying the service, since they provide the arguments for the underlying web service operation.
- There *INIT* operation registers a WSMED user session.
- Finally, the operation *EXIT_S* terminates a user session.

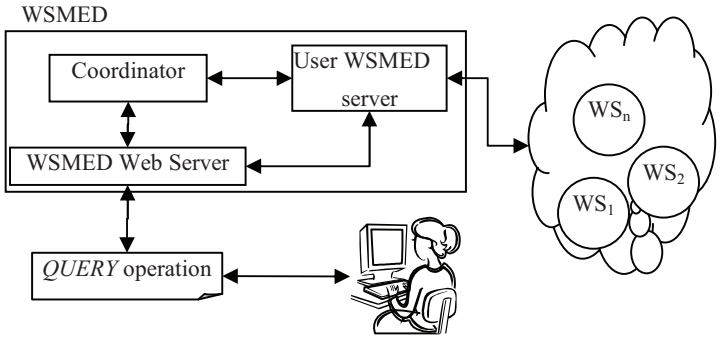


**Figure 3. Web service query service**

### 3.1 Web Service Architecture

Figure 3 shows the processes involved when using the WSMED web service. The *WSMED Web Server* is a lightweight standalone

SOAP web server using the library, *Quick Server* [5], to send and receive SOAP messages using the HTTP protocol.

The *coordinator* implements the WSMED operations *INIT* and *EXIT_S* described by the document *wsmed.wsdl* [11] to manage the user WSMED servers. Each user is assigned a private *user WSMED server* to manage the session of the data providing web services she is querying.

## 4. Queries over Dependent Web Service Calls

For a given web service WSMED automatically generates *operation wrapper functions* (OWFs)[6] based on the WSDL definitions of the web service operations. An OWF defines an SQL view of a web service operation. For example, *Query1* in Figure 4 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*[2], *GetPlacesWithin*[2], and *GetPlaceList*[8]. In Figure 4 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'
```

**Figure 4. Query1 defined in SQL**

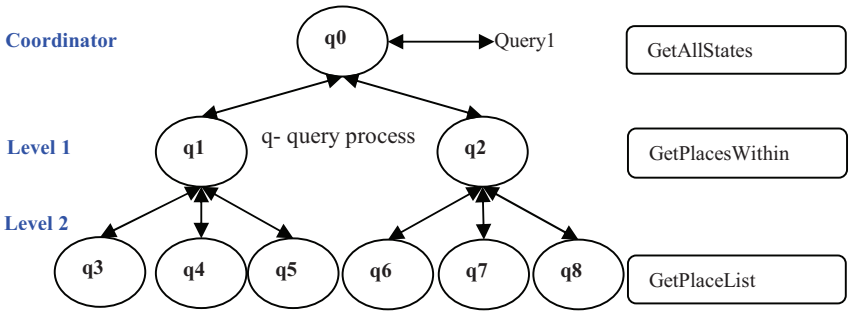


**Figure 5. Process tree**

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

### 4.1 WSMED Process Tree

The web service metadata in a WSDL file describing web service operations to query is stored in the WSMED's web service meta-store by the *IMPORTWSDL* operation. Figure 5 gives an example of a process tree generated by the WSMED query optimizer. The query is processed by the coordinator process *q0.* Any query process 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.

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. 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 an algebra operator *AFF_APPLYP* in the execution plan whenever an OWF is encountered.

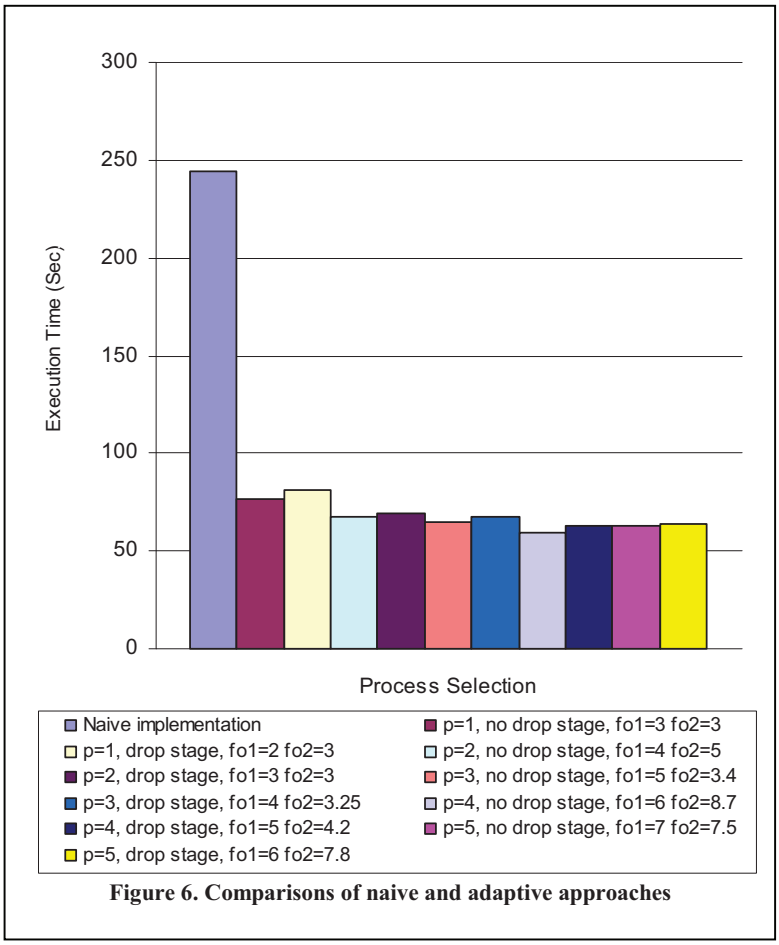


**Figure 6. Comparisons of naive and adaptive approaches**

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 *AFF_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 a stream of parameter tuples to the shipped plan function installed in the children processes ready for execution. Analogously, each *AFF_APPLYP* 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 when initialized. 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.

## 4.2 Adaptive First Finished Apply in Parallel

The algebra operator *AFF_APPLYP* (Adaptive First Finished Apply in Parallel) [6] automatically achieves an optimized process tree:

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

It ships in parallel to 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 AFF_APPLYP asynchronously as a stream of tuples, result. *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 sub-trees until it is not expected any more performance improvement.

We experimented [6] with different values of *p* (number of query processes added each time) and different change thresholds, with and without the dropping query processes when an optimum point is reached. We concluded (Figure 6) 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.

## 5. Related Work

WSQ/DSQ [4] 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 [7] 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* [3]. However, *MAPREDUCE* is more of a programming model than a query operator and is not dynamically rearranging query execution plans as *AFF_APPLYP*.

## 6. Conclusion

WSMED provides a general relational query service over any data providing web services for given WSDL meta-data descriptions. Queries are expressed in SQL to dynamically compose data providing web services. WSMED is accessible through a URL [10] from anywhere without installing any software.
WSMED automatically and adaptively find an optimized parallel execution plan calling 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. The adaptive method is shown to be efficient.

## 7. ACKNOWLEDGMENTS

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

## 8. REFERENCES

[1] Ceri, S. 2009.Search Computing. In Proceedings of International Conference on Data Engineering (The Shanghai, The China, March 29 – April 02, 2009), IEEE computer society (2009), 1-3.

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

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

[4] Goldman, R., Widom, J. 2000. WSQ/DSQ: a practical approach for combined querying of databases and the Web. In Proceedings of ACM SIGMOD International Conference on Management of Data. ACM, New York(2000), 285-296

[5] Quick Server, http://www.quickserver.org/

[6] Sabesan, M., Risch, T. 2009. In Proceedings of First IEEE Workshop on Information & Software as Services. Adaptive Parallelization of Queries over Dependent Web Service Calls. . IEEE computer society (2009), 1725-1732.

[7] Srivastava, U., Widom, J., Munagala, K., Motwani, R. 2006. In Proceedings of Very Large Database Conference. Query Optimization over Web Services. VLDB Endowment (2006), 355- 366.

[8] TerraServer, TerraService, http://terraservice.net/webservices.aspx

[9] The Next Wave: Everything as a Service, http://www.hp.com/hpinfo/execteam/articles/robison/08eaas.html

[10] WSQS Demo, http://udbl2.it.uu.se/WSMED/wsmed.html

[11] WSMED WSDL, http://udbl2.it.uu.se/WSMED/wsmed.wsdl