Benchmarking performance of web service operations

Shuai Zhang
Abstract

**Benchmarking performance of web service operations**

*Shuai Zhang*

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. In this project a web service benchmark (WSBENCH) is developed to simulate the performance of web service calls. Web service operations are specified as SQL statements. The function generator of WSBENCH converts user specified SQL queries into functions and automatically generates a web service. WSBENCH can automatically both generate and deploy web the service operations for exported functions. Furthermore WSBENCH supports controlled experiments, since users can control the characteristics of web service operations such as scalability of data and delay time. The database used in this project is generated by the Berlin Benchmark database generator.

A WSBENCH demo is built to demonstrate the functionality. The demo is implemented as a JavaScript program acting as a SOAP client, directly calls WSBENCH services from a web browser. Users can make a web service request by simply providing the web service operation’s name and parameter values list as the input. It makes the WSBENCH very simple to the use.
# Index

1. Introduction .......................................................................................... 2
2. Background ............................................................................................ 4  
   2.1 WSDL ............................................................................................... 4  
   2.2 SOAP ............................................................................................... 6  
   2.3 JavaScript ........................................................................................ 8  
   2.4 Amos II ............................................................................................. 8  
      2.4.1 Amos II function ........................................................................ 9  
      2.4.2 The Relational database wrapper ............................................ 9  
   2.5 WSMED ............................................................................................. 11  
      2.5.1 The web service generator ...................................................... 11  
      2.5.2 The WSMED web server ......................................................... 13  
   2.6 The Berlin SPARQL Benchmark (BSBM) Dataset ............................ 14  
3. The WSBENCH system .......................................................................... 16  
   3.1 Deploying WSBENCH operations .................................................... 16  
   3.2 Implementation ................................................................................ 19  
4. The WSBENCH demo ............................................................................ 21  
Conclusion & Future work ......................................................................... 28  
References .................................................................................................. 29  
Appendix A: WSDL of WSBENCH ............................................................. 31  
Appendix B: Web service specifications .................................................... 35  
Appendix C: Generated interface functions .............................................. 36  
Appendix D: Generated WSDL ................................................................. 37
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 [1] incorporating standards such as SOAP [3], WSDL [2], and XML Schema [9]. Web services provide a distributed application infrastructure by defining a set of operations that can be invoked over the internet. Web service operations are programs called using web services. They are self contained by meta-data that describe the data types of their arguments and results, i.e. their signatures, using the XML-based 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 is a web service operation that returns collections of objects for a given set of arguments without any side effects.

The purpose of this project is to develop a web service benchmark, WSBENCH, to simulate the performance of data providing web service operation calls. Using WSBENCH, user can defined their own data providing web service operations by just specifying parameterized SQL queries and a function name. The corresponding web service operations are automatically generated from the user provided SQL queries and immediately deployed.

The Berlin benchmark dataset is used in this project to implement the generated web services. The Berlin SPARQL Benchmark has an automatic database generator to generate relational databases of different sizes. The web services access such a relational database produced by the Berlin Benchmark. The relational database is currently managed by MySQL. Since WSBENCH is based on standard SQL other benchmarks, database generators, and DBMSs can be used as well.

The WSBENCH benchmark generator supports controlled experiments to analyze the different characteristics of web service operations such as scalability of data
provided through the web service operations. In general web service operations have different execution times. The execution time highly depends on when and where the web service operations are called. Therefore WSBENCH permits the user to specify desired minimum delay of a web service operation calls. Another parameter is the maximum size of result collection of a call. Experiments can be set up by specifying different SQL queries, different result sizes, and different execution delays.

WSBENCH is based on WSMED (Web Service MEDiator) [5], which is a system to enable SQL queries to query data providing web service operations. 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 [4]. 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. Each operation calls the interface function and sends back the result as a collection. The WSMED web service generator is used in WSBENCH to generate and deploy a web service operations specified as a parameterized SQL query.
2. Background

2.1 WSDL

WSDL (short for Web Services Description Language) [10] is an XML-based language used for describing Web services and how to access them. It provides a mechanism to specify information about services in a structured way, such as the data format, the concrete protocol, and the public interface. The web services in a WSDL document are defined as abstract collections of network endpoints or ports which can be reused. WSDL became a W3C Recommendation on 26, June, 2007 [11].

A WSDL document is an XML document that contains some definitions for describing web service(s). It has a root element: definitions and other seven major elements: types, message, portType, binding, port, operation, and service.

- Definitions is the root element of the WSDL document, it contains other WSDL elements.
- Types is a container for web service type definitions.
- Message is the definition(s) for the data element(s) of an operation.
- Operation is a description for an operation of a service.
- Port Type is a collection of operations, used for specifies the abstract operation(s) of a service.
- Binding describe concrete the communication protocol and the message format.
- Port defines the network address for a web service.
- Service is collection of ports.

A complete WSDL document contains two parts: abstract definitions and concrete definitions. Abstract definitions contain elements types, message and portType. They are independent of platform and programming language, which makes the service
reusable. *Concrete definitions* contain *binding* and *service*, which binds the service to specific case.

![WSDL document structure](image-url)

**Figure 2.1.1 WSDL document structure**

WSDL describes the web services in a structured format, which provides a standard way for other businesses to access those services.

In this project, client users can define their needed operations as SQL queries. WSBENCH will automatically deploy the operations into web services with corresponding WSDL documents.

WSDL documents are used to describe the predefined functionality of WSBENCH and user defined web service operations.
2.2 SOAP

SOAP stands for Simple Object Access Protocol. SOAP uses XML to define an extensible messaging framework providing a message construct that can be exchanged over a variety of underlying protocols [12]. It is language, platform independent, simple and extensible. As a communication protocol over HTTP, it provides a way to let the applications exchange structured information over the internet.

A SOAP message is composed by the following elements: Envelope element, Header element, Body element and Fault element. All these elements are declared in the default namespace for the SOAP envelope [13].

The Envelope element is the root of element of a SOAP message. The Header and Body elements are contained in it. The Envelope element defines the XML document as a SOAP message. The namespace is specified in Envelope.

The Header element is optional. It is used for providing additional information about the soap message, such as authentication, security, transaction information and etc.

The Body element is the mandatory part of the SOAP message. It contains the actual
information about the operations of the web service used for communicating such as: message name, reference to service operation, operation information, etc.

The *Fault* element is optional. It is used for the error messages and can only appear once in a SOAP message. The *Fault* element is composed by the following sub-elements: *faultcode*, *faultstring*, *faultactor*, and *detail*. *Faultcode* is an identifying code for predefined error. *Faultstring* is a readable explanation for understand the fault. *Faultactor* is used for giving the reason for the fault. *Detail* is used to carry application-specific error messages related to the *body* element. The detail element may have child elements, called detail entries.

SOAP is used as the transport protocol connecting the WSBENCH web service with a client application, such as a Java, C#, or JavaScript application.

The demo application is an application written in JavaScript, which communicates directly with WSBENCH using SOAP. The following is an example of a SOAP message from this project:

```xml
<soap:Envelope xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
    <soap:Body>
        <DNEW_FUNCTION xmlns="urn:WSAmos">
            <member0 xsi:type='xsd:string'>op1</member0>
            <member1 xsi:type='xsd:string'>select nr,product from offer where price&gt;tprice</member1>
        </DNEW_FUNCTION>
    </soap:Body>
</soap:Envelope>
```

Figure 2.3.2 SOAP message example

SOAP messaging can be written in two styles: *document* style and *RPC* (Remote Procedure Call) style. The *Document* style has no fixed structure, the *body* element simply contains an XML document whose format the sender and the receiver must agree upon. For the *RPC* style, a method or remote procedure of the service can be
invoked and the corresponding results can be got from the service. The RPC style is used in this project.

2.3 JavaScript

JavaScript [14] is a Netscape developed object-oriented scripting language. It is compatible with almost all the browsers such as Internet explorer, Chrome, Firefox, and so on. JavaScript is a lightweight and interpreted programming language. It is accessible and easy understandable. JavaScript can be regarded as a procedural or an object oriented programming language. You can directly embed it into HTML pages or you can define your JavaScript objects attach with methods and properties in separate “.js” files that can be invoked by HTML pages. JavaScript is primarily used in the client side of a web application within a web browser. JavaScript is very useful in designing dynamic and interactive web pages.

The WEBENCH demo is built using only JavaScript. The system uses the Functional Web Services Client (FWSC) [7], which allows any web service operation to be called as a JavaScript function. SOAP is used as communication protocol. FWSC dynamically builds, sends, and receives SOAP messages to communicate from JavaScript directly with web service operations.

2.4 Amos II

Amos II (Active Mediators Object System II) [4] is an extensible mediator system allowing different kinds of distributed data sources to be queried. The system is centered on an object-relational and functional query language called AmosQL. It can store data in its main-memory object store; wrappers can also be defined for different kind of data sources. In this project we use Amos II to make function calls to a wrapped beck-end relational database.
2.4.1 Amos II function

Functions in Amos II are mainly used to represent the properties of objects, describe the relationship among different objects, get views on object or create stored procedures. Kinds of functions in Amos II used in this project are [15]:

- A **derived function** is defined by a single AmosQL query. The syntax is:

  ```
  create function <function-name(type-name)> as select objects [from-clause] [where-clause];
  ```

- A **foreign function** is defined by some external programming languages. Amos II provides foreign function interfaces for some programming languages, such as C/C++, Java, and Lisp.

- A **procedural function** is defined by using procedural AmosQL statements. It makes AmosQL computationally complete.

In this project, user specified SQL queries are compiled into derived functions by a function generator. WSBENCH’s main module the function generator is defined as a foreign function and the web service generator is defined as a procedural function.

2.4.2 The Relational database wrapper

Amos II has a number of ways to access different external data sources by defining wrappers for each kind of external sources. A wrapper is a software module used for accessing external data source using AmosQL.

The **JDBC wrapper** represented by the type `Jdbc` implements wrappers for the relational databases.

In order to access a relational data source using JDBC, first we need to create an instance of type `Jdbc` which associates a JDBC driver with a database connection:

```
jdbc(Charstring database-name, Charstring driver);
```
For example:

```java
jdbc("ibds", "com.mysql.jdbc.Driver");
```

To actually establish the relational database connection this function is used: ‘

```java
connect(Jdbc j, Charstring db, Charstring username, Charstring password)
    -> Relational
```

Where \( j \) is a connection object, \( db \) is the identifier of the database for using \( username \) and \( password \) are the information needed to access the database. E.g.

```java
       "regress");
```

Now, we can send arbitrary SQL queries to the connected database for execution by:

```java
sql(Relational r, Charstring query, Vector params) -> Bag of Vector results
```

\( j \) is a connection object, \( query \) is the SQL statement to execute and vector \( params \) contains the value of the ‘?’ parameters of the SQL statement. The SQL result is returned as a bag of vectors. The details of other functions in the JDBC wrapper are explained in Amos II user’s manual [16].

In this project, using the Amos II Jdbc wrapper, WSBENCH server connects and queries the Berlin SPARQL Benchmark (BSBM) Dataset [6].
2.5 WSMED

WSMED stands for Web Service MEDiator. It provides a general SQL query web service to query any data providing web service operations without any further programming [5]. WSMED 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 over any data providing web services based on their WSDL meta-data descriptions. The WSMED coordinator automatically generates and optimizes a parallel execution plan calling the web services for a given SQL query. The web service generator generates web service interfaces for data sources once they are defined as the interface functions. The WSMED web server uses the HTTP protocol to receive, parse and send SOAP messages. The WSMED web server immediately deploys the interface functions as web service operations once they are exported without the need of restarting the WSMED web server.

In this project, the web service generator and the WSMED web server are utilized.

2.5.1 The web service generator

The web service generator is defined as an foreign Amos II function implemented in Java. The web service generator architecture is shown in Figure 2.5.5.1. It consists of four sub modules: the Amos II mediator/wrapper system, the function analyzer, the WSDL generator, and the WSDL exporter.
The function analyzer is an Amos II foreign function that defines the Amos II functions to be deployed as web service operations. It queries Amos II meta-data for the signature of the functions from which the exported web service operation signatures are generated. An exported signature consists of the names and types of a function’s arguments and results. They are passed to the WSDL generator.

The WSDL generator is used to produce the corresponding WSDL document for the exported functions. It dynamically builds an internal export description as a DOM data structure [21] in main memory using the WSDL4J [17] Java toolkit. The rules for how to transform signatures to WSDL operation descriptions is discussed in Section 4.2 of the publication for WSMED [5].

Finally the WSDL exporter transforms the main memory DOM representation of the export description into a WSDL document that describes the exported function interfaces as web service operations.
2.5.2 The WSMED web server

The WSMED web server is SOAP based server that uses the HTTP protocol to deal with SOAP messages communication. The purpose of the WSMED web server is to immediately deploy the interface functions as web service operations once they are exported without the need of restarting the WSMED web server or deploying any additional server site programming.

Figure 2.5.2.1 WSMED web server [5]

Figure 2.5.2.1 illustrates the structure of the WSMED web server. It consists five parts: the communication server, the XML parser, the XML writer, the DOM decoder, and the DOM encoder. The communication server is a modified JSoapServer [18]. JSoapServer is a lightweight standalone SOAP web server using the QuickServer [19] library for building web services. The communication server first receives a remote SOAP call from a client application via the HTTP protocol. Then the communication server extracts the information of the message and passes it to the XML parser as a SOAP request envelop. The XML parser transfers the request envelope into a DOM
data. Then through the DOM decoder the DOM data representation of a SOAP message is converted into a call to the corresponding interface function. The DOM decoder converts the data types of a receiving message to the format required by interface functions. Then it calls Amos II to execute the functions. The results are passed to the DOM encoder which uses the signature of the function and data type mappings between XML and Java to build a result DOM structure. The XML writer converts the DOM structure result to a SOAP response message and sends it back to the commutation server. Finally, the communication server sends back the SOAP message to the client application over the HTTP protocol.

2.6 The Berlin SPARQL Benchmark (BSBM) Dataset

The dataset used in this project is from the Berlin SPARQL Benchmark. The Berlin SPARQL Benchmark (BSBM) is a benchmark for measuring the performance of storage systems. The Berlin SPARQL Benchmark dataset is named BSBM. It is built around an e-commerce use case, where a set of products is offered by different vendors and different consumers have posted reviews about products [6]. The BSBM dataset is generated by a Java based data generator, which is open source and can be downloaded for free. The data generation is deterministic. The dataset is scalable to different sizes based on a user provided number of products. It has three data modes for the same semantics: RDF triple data mode, Named Graphs data mode and relational data mode. Only the relational data mode is used in this project. Using the data generator, the BSBM dataset is written into `.sql` script files using the following relational tables. (The words with red color and _ are the primary key attributes.)

- **ProductFeature** (nr, label, comment, publisher, publishDate)
- **ProductType** (nr, label, comment, parent, publisher, publishDate)
- **Producer** (nr, label, comment, homepage, country, publisher, publishDate)

- **Product** (nr, label, comment, producer, propertyNum1, propertyNum2,
15

The scale factor value is set to 2785, which means we have 2785 rows in the `product` table. Depending on the BSBM dataset population rules, we can specify the number of rows of each table as Figure 2.6.1

<table>
<thead>
<tr>
<th>Scale Factor (number of products)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of <code>ProductType</code></td>
<td>151</td>
</tr>
<tr>
<td>Number of <code>ProductFeature</code></td>
<td>4745</td>
</tr>
<tr>
<td>Number of <code>Producer</code></td>
<td>60</td>
</tr>
<tr>
<td>Number of <code>Vendor</code></td>
<td>34</td>
</tr>
<tr>
<td>Number of <code>Offer</code></td>
<td>55700</td>
</tr>
<tr>
<td>Number of <code>Review</code></td>
<td>26200</td>
</tr>
<tr>
<td>Number of <code>ProductTypeProduct</code></td>
<td>11140</td>
</tr>
<tr>
<td>Number of <code>ProductFeatureProduct</code></td>
<td>57163</td>
</tr>
<tr>
<td>Number of <code>Person</code></td>
<td>1432</td>
</tr>
</tbody>
</table>

Figure 2.6.1 The characteristics of BSBM dataset used in this project
3 The WSBENCH system

Figure 3.1 below illustrates the WSBENCH architecture.

The purpose of WSBENCH is to convert user provided SQL queries into corresponding web service operations, generate a WSDL document describing the operations, and finally deploying the generated operations.

3.1 Deploying WSBENCH operations

WSBENCH itself is a web service defined by three Amos II functions from which the corresponding three web service operations are generated and deployed by the WSMED web service generator. The automatically generated WSDL document is listed in Appendix A. The WSBENCH functionality is defined by the following three web service operations:
The web service operation `define_operation` defines a new web service operation by an SQL query.

The operation `deploy_service` deploys a number of defined web service operations as a web service.

The operation `execute_operation` calls any deployed operation. The call includes specification of a time delay per tuple in the query result and an upper limit of the size of the result.

The Amos II function `define_operation` has the signature:

```
define_operation(Charstring oname, Charstring query) -> Charstring
```

The corresponding web service operation is named ‘WSBENCH’ and described by the WSDL document in Appendix A. A new web service operation named `oname` is defined by a parameterized SQL query. The input parameters of the new web service operation are determined by prefixing input parameter names in the SQL query with ‘?’. The types of the output parameters are obtained by the system accessing the schema of the relational database. The result data types are determined by analyzing the projection in the query. For example:

```
define_operation("op1","select distinct ps.name,pr.nr from person ps, producer pr where ps.country=pr.country and ps.country=?coun and ps.publisher=?pub");

define_operation("op2","select product from offer where offer.deliveryDays=?day");
```

These will define two web service operations called `op1` and `op2` as derived Amos II functions with signatures:
The three parameters `coun`, `pub`, and `day` of `op1` and `op2` are defined by identifying the parameters `?coun`, `?pub`, and `?day` in the parameterized SQL query. Their data types are obtained by accessing the database schema to obtain the types of `ps.country`, `ps.publisher` and `offer.deliveryDays`, respectively. The system automatically adds the two additional parameters `limit` and `delay` to the function’s signature to allow run time specification of maximum result size and time delay per tuple, respectively. The result of the function `op1` is a set of tuples containing pairs of strings and the result of the function `op2` is a set of tuples containing integers.

The function `define_operation` is normally called several times to define all desired operations. Then `deploy_operation` is called to generate the WSDL document and dynamically deploy specified web service operations as a new web service. It has the signature:

```
deploy_service(Charstring ws, Charstring onames) -> Charstring
```

The name of the new web service in `ws` and `onames` specifies a comma separated list of the predefined web service operations to be deployed. A WSDL document with all necessary meta-data is generated in the file `ws.wsdl` in the server. For example:

```
deploy_service("service1", "op1,op2");
```
At this point the web service operations named \textit{op1, op2} are deployed as the web service named \textit{service} described by the WSDL file \textit{service.wsdl}. It can be called as any other web service. The WSDL file \textit{service.wsdl} has the contents listed in Appendix D.

The function \textit{execute_operation} allows to execute any deployed web service operation with different limits and delays. It has the signature:

\[
\text{execute\_operation(\text{Charstring oname, Vector params, Charstring delay, Integer limit})}\rightarrow\text{Bag of Vector}
\]

The web service operation named \textit{oname} is called with parameters \textit{params}, a minimum time \textit{delay}, and maximum result size \textit{limit}. For example:

\[
\text{execute\_operation(“op1”,{“US”,1}, 0.01,100);}
\]

\[
\text{execute\_operation(“op2”,{3}, 0.1,12);}
\]

The deployed web service operation \textit{op1} is called with parameters “US” and 1, a time delay of 0.01 seconds is added for each result tuple. At most 100 tuples are returned. These tuples are also returned as the result from \textit{execute\_operation}. The deployed web service operation \textit{op2} is called with parameters 4, a time delay of 0.1 seconds and at most 12 tuples are returned.

\textbf{3.2 Implementation}

When a new web service operation is defined by \textit{define\_operation}, the \textit{function generator} parses and compiles the user provided SQL query into a derived Amos II interface function. The operation \textit{deploy\_service} calls the WSMED web service generator to automatically deploy the web service operations for the exported
functions.

The function generator compiles the definitions of \textit{op1} and \textit{op2} into the following Amos II interface functions:

\begin{verbatim}
create function op1(Charstring coun, Integer pub, Integer t_number, Real delay) -> <Charstring a0, Integer a1>
as select a0, a1 from vector vec
where vec = sql(getJDBC(), "select distinct ps.name, pr.nr, sleep(?)
from person ps, producer pr
where ps.country = pr.country and
ps.country = ? and ps.publisher =?
limit ?;",
{delay, coun, pub, t_number})
and a0 = vec[0] and a1 = vec[1];
\end{verbatim}

\begin{verbatim}
create function op2(Integer day, Integer t_number, Real delay) -> <Integer a0>
as select a0 from vector vec
where vec = sql(getJDBC(), "select product, sleep(?)
from offer
where offer.deliveryDays = ? limit ?;",
{delay, day, t_number})
and a0 = vec[0];
\end{verbatim}

The \texttt{sleep()} call in MySQL delays each tuple with the specific time delay. The limit clause limits the size of the result. The function \texttt{getJDBC()} returns the current JDBC connection to the database.
Appendix C lists the definitions of the two Amos II interface functions \( \text{op1} \) and \( \text{op2} \).

### 4. The WSBENCH demo

The WSBENCH on-line demo illustrates the functionality of WSBENCH. It demonstrates all web service operations provided by WSBENCH through its user interface that can be run in any web browser without software installation. The web pages are written in JavaScript. FSC (Functional Web Services Client) [7] is used to create a JavaScript SOAP client which enables WSBENCH and user defined web applications can be directly called from a web browser. Figure 4.1 shows the architecture of the WSBENCH Demo.
FSC consists of three modules, the *public APIs*, the *request module*, and the *response module*. The *public APIs* contains the public functions which can be called by user scripts. The *request module* consists of three sub-modules. The *Load WSDL* module loads the WSDL information into a JavaScript global variable. The *Build SOAP envelope* module parses the global variable and generates the SOAP request envelope with the user input data. The *Send request* module sends the SOAP message to the web service by using *XMLHttpRequest* object. The response module analyzes the response message according to WSDL description that saved in the global variable and extracts result objects from the SOAP envelope.

Using FSC, users can make a web service request by simply providing the web service operation’s name and parameter values list as the input. By resolving the WSDL documents of the operation, automatically FSC builds the SOAP request envelope and sends the request SOAP in either an asynchronous or synchronous way.
execute web service operations. Users don’t need to create functions every time, if they have defined them previously. Whenever new functions are created, the user has to make sure that the functions have been deployed before executing them.

Figure 4.3 Create functions

Create Functions

An example:

<table>
<thead>
<tr>
<th>Name of Function</th>
<th>qpl</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL Query</td>
<td>select distinct pa.name, pr, rr from person pa, producer pr where pa.country=pr.country and pa.country=qpl and pa.publisher=pub</td>
</tr>
</tbody>
</table>

Enter the parameters for your own operation.

<table>
<thead>
<tr>
<th>Name of Function</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL Query</td>
<td></td>
</tr>
</tbody>
</table>

Create | Reset form | Close

In this page, users can create Amos II functions by entering “Name of operation” and “SQL query”. The button “Create” invokes the *define_operation* function of WSBENCH. The parameters passed in will be automatically converted into corresponding Amos II function arguments. Users still have to make sure that the SQL query is correct, otherwise an error message will be displayed as shown in Figure 4.4.

Figure 4.4 Error message for creating function

Please make sure you have input the correct SQL query!

With appropriate inputs, functions are created and stored in the file “user_defined.amosql” following a message (Figure 4.5). The functions can be created at anytime without any restrictions.
After having created all functions, it is the time to deploy them as web service operations. The button “Deploy” in figure 8.6 calls the web service generator. By pressing this button, the deploy_service operation of WSBENCH is invoked. It loads users defined Amos II functions into the WSBENCH server, then deploys the exported interface functions as web service operations, and finally generates web service interface descriptions (a WSDL document) for them. Figure 4.7 shows an example result if a user assigns test to web service name and “op1” to Amos II function name.
Users can deploy more than one operations of a web service at once. The WSMED web server immediately deploys the interface functions as web service operations once they are exported without the need of restarting the WSMED web server. So operations are usable directly after they are deployed.

Figure 4.8 Execute operations

**Execute existing operations.**

*Information about the existing operations.*

Please enter the necessary information:

<table>
<thead>
<tr>
<th>Name of the Operation:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation parameters:</td>
<td></td>
</tr>
<tr>
<td><strong>You dont need</strong> to fill this if an operation does not have parameters.</td>
<td></td>
</tr>
<tr>
<td>Time delays:</td>
<td></td>
</tr>
<tr>
<td>Number of Tuples:</td>
<td></td>
</tr>
</tbody>
</table>

---

**The result of operation is:**

When the operations are deployed they can be executed by pushing the button “Execute” to call the WSBENCH operation *execute_operation*. If an operation has more than one parameter, the parameters should be separated with comma like “US,1”. The user will receive the matched result with the execution time, as shown in Figure 4.9.
Two details should be noticed: In order to get results for an operation, the user has to make sure that a correct operation name is specified along with and correct value for operation’s parameter(s). If the user entered an operation that does not exist an error message will be shown as Figure 4.11. On the other hand, if the parameter value(s) entered do not match the operation’s signature, another error message is displayed, as in Figure 4.10. The link “Information about existing operations” displays the necessary information about all defined operations.
Figure 4.10 Error message for wrong parameter(s) value

<table>
<thead>
<tr>
<th>Name of the Operation:</th>
<th>op1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation parameters:</td>
<td></td>
</tr>
<tr>
<td>You don't need to fill this if an operation does not have parameters.</td>
<td></td>
</tr>
<tr>
<td>Time delays:</td>
<td>1</td>
</tr>
<tr>
<td>Number of Tuples:</td>
<td>12</td>
</tr>
</tbody>
</table>

The result of operation is: **Query execution time = 0.054 seconds**
The parameter(s) you inputed do not match with the function, please check the function definition and try again!

Figure 4.11 Error message for wrong operation name

<table>
<thead>
<tr>
<th>Name of the Operation:</th>
<th>op1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation parameters:</td>
<td></td>
</tr>
<tr>
<td>You don't need to fill this if an operation does not have parameters.</td>
<td></td>
</tr>
<tr>
<td>Time delays:</td>
<td>1</td>
</tr>
<tr>
<td>Number of Tuples:</td>
<td>12</td>
</tr>
</tbody>
</table>

The result of operation is: **Executing the Query ..**
There is no data for your operation
Conclusion & Future work

In conclusion, this project develops a web service benchmark (WSBENCH) used to simulate the performance of web service calls. WSBENCH adheres to the *Everything as a Service* (XaaS) paradigm [20]. It has three main modules: the WSBENCH function generator, the WSMED web service, and the WSMED web service generator. The WSBENCH function generator converts user provided parameterized SQL queries into corresponding Amos II functions. The WSMED web service generator and WSMED web server automatically generates and deploys web service operations for the exported Amos II functions without the need of restarting the web server or deploying any additional server site programming. In order to let WSBENCH support controlled experiments, WSBENCH lets the users control the characteristics of web service operations such as size of result data sets and delay time.

As a proof-of-concept application of WSBENCH, a WSBENCH demo is built. The functional web service client [7] is used to implement a JavaScript SOAP client which enables WSBENCH and user defined web service applications to be directly called from a web browser without the need of installing software. Users can make web service requests by just entering the web service operation’s name and parameter values list as inputs.

A future work is to implement WSBENCH in a cluster to simulate parallel web services. An important feature of WSMED is adaptive operator PAP (Parameterized Adaptive Parallelization) [8] where web service operations are called in parallel. This will require a parallel web service server for improved performance.

When the user defined web service operations is called with a large “number of tuples”, the whole data result is currently packed in a single SOAP message, which will not work for very large result data sets. To solve this, implementing stateful web service operations is needed by using a scan feature that gets a piece of the result for each call.
References


[7]. Di Jin, JavaScript based web service access to a functional DBMS, MSc Thesis IT 10 019, Department of Information Technology, Uppsala University, May 2010


[15]. Amos II Function introduction,
http://www.it.uu.se/research/group/udbl/amos/doc/amos_users_guide.html#function-definitions

[16]. Amos II relational database wrapper,
http://www.it.uu.se/research/group/udbl/amos/doc/amos_users_guide.html#relational


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

[21]. DOM, http://www.w3.org/DOM/
Appendix A: WSDL of WSBENCH

<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions name="WSbench" targetNamespace="urn:WSAmos"
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/encoding/">
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/
xmlns:tns="urn:WSAmos"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/">

<wsdl:types>

<xsd:schema targetNamespace="urn:WSAmos"
xmlns="http://www.w3.org/2001/XMLSchema">

<xsd:complexType name="VectorofanyType">
<xsd:sequence>
<xsd:element maxOccurs="unbounded" minOccurs="1" name="member"
type="xsd:anyType"/>
</xsd:sequence>
</xsd:complexType>

<xsd:complexType name="VectorofOID">
<xsd:sequence>
<xsd:element maxOccurs="unbounded" minOccurs="1"
name="member" type="xsd:string"/>
</xsd:sequence>
</xsd:complexType>

<xsd:complexType name="VectorofINTEGER">
<xsd:sequence>
<xsd:element maxOccurs="unbounded" minOccurs="1"
name="member" type="xsd:int"/>
</xsd:sequence>
</xsd:complexType>

<xsd:complexType name="VectorofREAL">
<xsd:sequence>
<xsd:element maxOccurs="unbounded" minOccurs="1"
name="member" type="xsd:double"/>
</xsd:sequence>
</xsd:complexType>

<xsd:complexType name="VectorofCHARSTRING">
<xsd:sequence>
<xsd:element maxOccurs="unbounded" minOccurs="1"
name="member" type="xsd:string"/>
</xsd:sequence>
</xsd:complexType>

</xsd:schema>

</wsdl:types>
</wsdl:definitions>
type="xsd:string"/>
  </xsd:sequence>
</xsd:complexType>
</xsd:schema>

</wsdl:types>

<wsdl:message name="DEFINE_OPERATIONResponse">
  <wsdl:part name="VWSMOS1" type="xsd:string"/>
</wsdl:message>

<wsdl:message name="DEPLOY_SERVICERequest">
  <wsdl:part name="ONAME" type="xsd:string"/>
  <wsdl:part name="WSNAME" type="xsd:string"/>
</wsdl:message>

<wsdl:message name="EXECUTE_OPERATIONRequest">
  <wsdl:part name="ONAME" type="xsd:string"/>
  <wsdl:part name="OPARA" type="xsd:string"/>
  <wsdl:part name="DELAY" type="xsd:string"/>
  <wsdl:part name="NOTUP" type="xsd:int"/>
</wsdl:message>

<wsdl:message name="DEFINE_OPERATIONRequest">
  <wsdl:part name="ONAME" type="xsd:string"/>
  <wsdl:part name="QUERY" type="xsd:string"/>
</wsdl:message>

<wsdl:message name="DEPLOY_SERVICEResponse">
  <wsdl:part name="VWSMOS1" type="xsd:string"/>
</wsdl:message>

<wsdl:message name="EXECUTE_OPERATIONResponse">
  <wsdl:part name="VWSMOS1" type="tns:VectorofanyType"/>
</wsdl:message>

<wsdl:portType name="WSbenchPortType">

  <wsdl:operation name="DEFINE_OPERATION" parameterOrder="ONAME QUERY">
    <wsdl:input name="DEFINE_OPERATIONRequest" message="tns:DEFINE_OPERATIONRequest"/>
    <wsdl:output name="DEFINE_OPERATIONResponse" message="tns:DEFINE_OPERATIONResponse"/>
  </wsdl:operation>

</wsdl:portType>
<wsdl:operation>
  <wsdl:input name="DEFINE_OPERATIONRequest">
    <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
      namespace="urn:WSAmos"/>
  </wsdl:input>
  <wsdl:output name="DEFINE_OPERATIONResponse">
    <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
      namespace="urn:WSAmos"/>
  </wsdl:output>
</wsdl:operation>

<wsdl:operation name="DEPLOY_SERVICE" parameterOrder="ONAME WSNNAME">
  <wsdl:input name="DEPLOY_SERVICERequest" message="tns:DEPLOY_SERVICERequest"/>
  <wsdl:output name="DEPLOY_SERVICEResponse" message="tns:DEPLOY_SERVICEResponse"/>
</wsdl:operation>

<wsdl:operation name="EXECUTE_OPERATION" parameterOrder="ONAME OPARA DELAY NOTUP">
  <wsdl:input name="EXECUTE_OPERATIONRequest" message="tns:EXECUTE_OPERATIONRequest"/>
  <wsdl:output name="EXECUTE_OPERATIONResponse" message="tns:EXECUTE_OPERATIONResponse"/>
</wsdl:operation>

</wsdl:portType>

<wsdl:binding name="WSbenchSoapBinding" type="tns:WSbenchPortType">
  <wsdlsoap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>

  <wsdl:operation name="DEFINE_OPERATION">
    <wsdlsoap:operation soapAction=""/>
    <wsdl:input name="DEFINE_OPERATIONRequest">
      <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
      namespace="urn:WSAmos"/>
    </wsdl:input>

    <wsdl:output name="DEFINE_OPERATIONResponse">
      <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
      namespace="urn:WSAmos"/>
    </wsdl:output>
  </wsdl:operation>

  <wsdl:operation name="DEPLOY_SERVICE">
    <wsdlsoap:operation soapAction=""/>
    <wsdl:input name="DEPLOY_SERVICERequest">
      <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
      namespace="urn:WSAmos"/>
    </wsdl:input>

    <wsdl:output name="DEPLOY_SERVICEResponse">
      <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
      namespace="urn:WSAmos"/>
    </wsdl:output>
  </wsdl:operation>

</wsdl:binding>
<wsdl:input name="DEPLOY_SERVICERequest">
  <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" namespace="urn:WSAmos"/>
</wsdl:input>

<wsdl:output name="DEPLOY_SERVICEResponse">
  <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" namespace="urn:WSAmos"/>
</wsdl:output>

<wsdl:operation name="EXECUTE_OPERATION">
  <wsdlsoap:operation soapAction=""/>
  <wsdl:input name="EXECUTE_OPERATIONRequest">
    <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" namespace="urn:WSAmos"/>
  </wsdl:input>
  <wsdl:output name="EXECUTE_OPERATIONResponse">
    <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" namespace="urn:WSAmos"/>
  </wsdl:output>
</wsdl:operation>
</wsdl:binding>

<wsdl:service name="WSbenchService">
  <wsdl:port name="WSbenchPort" binding="tns:WSbenchSoapBinding">
  </wsdl:port>
</wsdl:service>
</wsdl:definitions>
Appendix B: Web service specifications

The Amos II call to define web service operation ‘op1’:
define_operation("op1","select distinct ps.name,pr.nr from person ps, producer pr where ps.country=pr.country and ps.country=?coun and ps.publisher=?pub");

The Amos II call to define web service operation ‘op2’:
define_operation("op2","select product from offer where offer.deliveryDays=?day");

The Amos II call to deploy web service operation ‘op1’ and ‘op2’ as a web service operations for the web service named ‘service’:
deploy_service("service1", "op1,op2");
Appendix C: Generated interface functions

Amos II interface function ‘op1’:
create function op1(Charstring coun, Integer pub, Integer t_number, Real delay) ->
  <Charstring a0, Integer a1>
as select a0, a1
  from vector vec
    where vec = sql(getJDBC(), "select distinct ps.name, pr.nr, sleep(?)
        from person ps, producer pr
        where ps.country = pr.country and
        ps.country = ? and ps.publisher = ?
        limit ?;",
          {delay, coun, pub, t_number})
  and a0 = vec[0] and a1 = vec[1];

Amos II interface function ‘op2’:
create function op2(Integer day, Integer t_number, Real delay) -><Integer a0>
as select a0
  from vector vec
    where vec = sql(getJDBC(), "select product, sleep(?)
        from offer
        where offer.deliveryDays = ? limit ?;",
          {delay, day, t_number})
  and a0 = vec[0];
Appendix D: Generated WSDL

Generated WSDL for web service ‘service’:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions name="service" targetNamespace="urn:WSAmos"
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" xmlns:tns="urn:WSAmos"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/">

<wsdl:types>

    <xsd:schema targetNamespace="urn:WSAmos"
xmlns="http://www.w3.org/2001/XMLSchema">

        <xsd:complexType name="VectorofanyType">
            <xsd:sequence>
                <xsd:element maxOccurs="unbounded" minOccurs="1" name="member"
type="xsd:anyType"/>
            </xsd:sequence>
        </xsd:complexType>

        <xsd:complexType name="VectorofOID">
            <xsd:sequence>
                <xsd:element maxOccurs="unbounded" minOccurs="1" name="member"
type="xsd:string"/>
            </xsd:sequence>
        </xsd:complexType>

        <xsd:complexType name="VectorofINTEGER">
            <xsd:sequence>
                <xsd:element maxOccurs="unbounded" minOccurs="1" name="member"
type="xsd:int"/>
            </xsd:sequence>
        </xsd:complexType>

        <xsd:complexType name="VectorofREAL">
            <xsd:sequence>
                <xsd:element maxOccurs="unbounded" minOccurs="1" name="member"
type="xsd:double"/>
            </xsd:sequence>
        </xsd:complexType>

    </xsd:schema>

</wsdl:types>
```

<xsd:complexType name="VectorofCHARSTRING">
  <xsd:sequence>
    <xsd:element maxOccurs="unbounded" minOccurs="1" name="member" type="xsd:string"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="WSOP1">
  <xsd:sequence>
    <xsd:element maxOccurs="unbounded" minOccurs="0" name="row">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="COUN" type="xsd:string"/>
          <xsd:element name="COUN" type="xsd:int"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="WSOP2">
  <xsd:sequence>
    <xsd:element maxOccurs="unbounded" minOccurs="0" name="row">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="OUT0" type="xsd:int"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>

</xsd:schema>

<wsd1:types>

<wsd1:message name="WSOP2Response">
  <wsdl:part name="results" type="tns:WSOP2"/>
</wsdl:message>

<wsd1:message name="WSOP1Request">
  <wsdl:part name="COUN" type="xsd:string"/>
  <wsdl:part name="PUB" type="xsd:int"/>
  <wsdl:part name="TNUMBER" type="xsd:int"/>
  <wsdl:part name="DELAY" type="xsd:double"/>
</wsdl:message>
<wsdl:message name="WSOP1Response">
  <wsdl:part name="results" type="tns:WSOP1"/>
</wsdl:message>

<wsdl:message name="WSOP2Request">
  <wsdl:part name="DAY" type="xsd:int"/>
  <wsdl:part name="TNUMBER" type="xsd:int"/>
  <wsdl:part name="DELAY" type="xsd:double"/>
</wsdl:message>

<wsdl:portType name="servicePortType">
  <wsdl:operation name="WSOP1" parameterOrder="COUN PUB TNUMBER DELAY">
    <wsdl:input name="WSOP1Request" message="tns:WSOP1Request"/>
    <wsdl:output name="WSOP1Response" message="tns:WSOP1Response"/>
  </wsdl:operation>

  <wsdl:operation name="WSOP2" parameterOrder="DAY TNUMBER DELAY">
    <wsdl:input name="WSOP2Request" message="tns:WSOP2Request"/>
    <wsdl:output name="WSOP2Response" message="tns:WSOP2Response"/>
  </wsdl:operation>
</wsdl:portType>

<wsdl:binding name="serviceSoapBinding" type="tns:servicePortType">
  <wsdlsoap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>

  <wsdl:operation name="WSOP1">
    <wsdlsoap:operation soapAction=""/>
    <wsdl:input name="WSOP1Request">
      <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                   namespace="urn:WSAmos"/>
    </wsdl:input>

    <wsdl:output name="WSOP1Response">
      <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                    namespace="urn:WSAmos"/>
    </wsdl:output>
  </wsdl:operation>
</wsdl:binding>
<wsdl:operation name="WSOP2">
  <wsdl:input name="WSOP2Request">
    <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
      namespace="urn:WSAmos"/>
  </wsdl:input>

  <wsdl:output name="WSOP2Response">
    <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
      namespace="urn:WSAmos"/>
  </wsdl:output>
</wsdl:operation>

</wsdl:binding>

<wsdl:service name="serviceService">
  <wsdl:port name="servicePort" binding="tns:serviceSoapBinding">
    <wsdlsoap:address location="http://192.168.0.100:8082/wsmed/service/WSMEDServlet"/>
  </wsdl:port>
</wsdl:service>

</wsdl:definitions>