Full Solution Indexing Using Database for QoS-aware Web Service Composition

Jing Li[a], Yuhong Yan[b]
Dept. of Computer Science and Software Engineering
Concordia University
Montreal, Canada
Email: jing.li.hnu@gmail.com[a]
Email: yuhong@encs.concordia.ca[b]

Daniel Lemire
LICEF Research Center, TELUQ
Université du Québec
Montreal, Canada
Email: lemire@gmail.com

***Abstract*—Automated service composition can fulfill user request by composing services automatically when no individual services meet the goal. Unfortunately, most of current automated service composition methods are in-memory methods, which are limited by expensive and volatile physical memory. In this work, we develop a relational-database approach for automatic service composition. Possible service combinations are stored in a relational database on persistence disk instead of volatile memory, and for any composition requests, solutions can be obtained by simple SQL queries. We offer three main contributions in this paper. First, pursuing earlier work, we overcome the disadvantages of in-memory composition algorithms, such as volatile and expensive, and provide a solution suitable to cloud environments. Second, compared with other pre-computing composition methods, we use a single SQL query: there is no need to eliminate spurious services iteratively. Third, we address the quality of services to maximize user's satisfaction in our system. An experimental validation is done, which shows the performance benefits of our system and proves that this system can find a valid composition solution with fewer services to maximize user satisfaction.**

***Keywords*-QoS based service composition; semantic match; database;**

I. Introduction

Cloud computing era has arrived and it raises the importance of web services. Web services are self-describing software modules that can be published, located and invoked on the web. Web services can be composed together when user requests cannot be fulfilled by any of them and this is the Web Service Composition (WSC) problem. Nowadays, enterprises may only focus on their core functions and find web services in the cloud to fulfill their tasks.

Many researchers have put great efforts in studying service composition problem and most of them are in-memory based algorithms. For each query, the composition algorithm is executed to find a solution from scratch. The in-memory based algorithms can only work when data fits in RAM. However, loading lots of services information into RAM is expensive (even today), and the search space is limited by the available physical memory. Besides, pure in-memory Java applications often assume that the machine will be doing just this one thing—composing web services. In the real world, from the web service data, users may need to support more than composition, out of the same servers and data. It makes little sense to lock up all the data in RAM just for composition and it is a waste of resources.

The above mentioned shortcomings have motivated researchers to utilize relational database to not only store web services but also solve service composition problem [1]–[3], among which, Lee *et al.* [2] is more outstanding, a PSR (Pre-computing Solutions in RDBMS) system is presented where paths between two services are computed beforehand using joins and indices and stored in relational database. In the PSR system, services are abstracted as single operators and paths have single input/output. To handle user query with multiple inputs and outputs, this system searches all the paths whose output is part of user query and combines them together. However, experimental results show that a number of redundant services exist in the solution, which increase user's execution cost as a result. (A service is redundant if all its outputs used by other services are also produced by other services [4].) Moreover, the PSR system does not support Quality of Service (QoS) optimization. These shortcomings motivate us to look for a more extensible method.

In this paper, we study web service composition problem in database and present a system called FSIDB (Full Solution Indexing using Database). This system takes advantage of large space available in relational database on persistence disk instead of small volatile memory to store all the possible service combinations. When a user query comes, we compose SQL queries to search in the database for a solution with optimized QoS.

The rest of this paper is organized as follows: Section II introduces the preliminary knowledge and system architecture. Data structure and algorithms are given in Section III. Experimental results of our proposed algorithms are shown in Section IV. Section V reviews related work and the conclusion is drawn in Section VI.

II. Preliminary and System Architecture

In this section, we first introduce preliminary knowledge, then, we provide an architecture overview of the FSIDB system and briefly describe this system.

A. QoS-aware service composition problem

Definition 1: A web service w is defined as a tuple (w_{in}, w_{out}, P, E, Q) with the following components:

- w_{in} is a finite set of typed input parameters of w. A web service is invoked only when all its input parameters are satisfied.
- w_{out} is a finite set of typed output parameters of w.
- P and E are sets of preconditions and effects respectively. P is the availability of the inputs and E is the availability of the outputs. The conditions are further discussed in Definition 6 and Definition 7.
- Q is a finite set of QoS criteria for w. In this paper, QoS of a service is expressed by the non-functional criteria on response time and throughput.

In addition to functional requirements, to meet the constraints in the composition problem and users' specific preference, recent research takes consider of non-functional requirements. QoS refers to the non-functional properties of services which determine service usability and utility such as availability, response time, throughput, price, execution duration, reputation and successful execution rate. When the service composition problem satisfies both users' functional goals and maximizes users' satisfaction at the same time, it is called QoS-aware service composition problem.

Response time in a data system is the interval between the request and the beginning of delivery the response. Throughput is the average rate of successful message delivery over a communication channel. Throughput is a positive criterion, the higher the value, the higher the quality. Meanwhile response time is a negative criterion, the higher the value, the lower the quality. In this paper, the quality of the solution is evaluated by either the response time or throughput.

Definition 2: A web service composition problem can be represented by a tuple (S, C_{in}, C_{out}, Q) with the following components:

- S is a finite set of services.
- C_{in} is a finite set of typed input parameters.
- C_{out} is a finite set of typed output parameters.
- Q is a finite set of quality criteria.

Definition 3: A layered graph is a type of directed graph in which the vertices are partitioned into a sequence of layers and edges generally directed backwards.

Definition 4: A service layer W=$\{w_i|i$ =1:$m\}$ is a finite set of services and m is the number of services.

Definition 5: A path is a layered graph defined as a tuple (SL, $path_{\text{in}}$, $path_{\text{out}}$, Q) with the following components:

- $SL = \{W_k|k$=1:$l\}$ is a set of service layers and l is the number of layers in the path. For each service in a layer, the input parameters are provided by either the input of path or the outputs of preceding layers.
- $path_{\text{in}} = \{\bigcup_{k=1}^{l}\{w_{i.in}|w_i \in W_k\}\} - \{\bigcup_{k=1}^{l}\{w_{j.out}|w_j \in W_k\}\}$ is a finite set of typed input parameters.

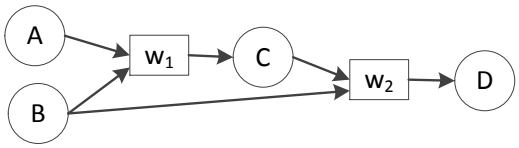

Figure 1. Service Composition Representation

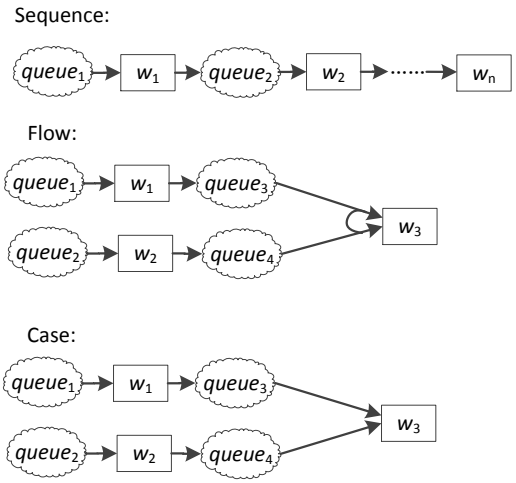

Figure 2. Services Composition Process Model

- $path_{\text{out}} = \{\bigcup_{k=1}^{l}\{w_{i.out}|w_i \in W_k\}\}$ is a finite set of typed output parameters.
- Q is a finite set of quality criteria.

The web service composition problem can be mapped to a path. To solve the web service composition problem, we can find a path between user request and goal.

- $S \leftrightarrow SL$
- $C_{\text{in}} \leftrightarrow path_{\text{in}}$
- $C_{\text{out}} \leftrightarrow path_{\text{out}}$
- $Q \leftrightarrow Q$

Figure 1 is an example of a path. $W = \{\{w_1\}, \{w_2\}\}$ is a set of service layers, $path_{\text{in}} = \{A, B\}$ is the input and $path_{\text{out}} = \{C, D\}$ is the output.

As shown in Figure 2, there are three kinds of control structures in the composition problem: sequence, flow and case. Services in the sequence control structure are invoked one by one. Services in the flow control are invoked in parallel. For example, the flow control of Figure 2, w_1 and w_2 are invoked simultaneously, after that, w_3 is invoked. In case control structure, if either a branch of the current statement executes, the control flow passes to the next statement. For example, the case control of Figure 2, w_3 will be invoked if either w_1 or w_2 is invoked, if both w_1 and w_2 are invoked, we choose either one branch or compare QoS value of each branch to decide which branch invokes w_3.

Suppose services are represented by $w_1, \ldots, w_n$, the response time (R) and throughput (T) of a service composition are calculated via Equation (1) to Equation (6):

$$R(sequence) = \sum_{n=1}^{N} R(w_n), \quad (1)$$

$$R(flow) = \max\{R(w_1), \ldots, R(w_n)\}, \quad (2)$$

$$R(case) = \min\{R(w_1), \ldots, R(w_n)\} \quad (3)$$

$$T(sequence) = \min\{T(w_1), \ldots, T(w_n)\}, \quad (4)$$

$$T(flow) = \min\{T(w_1), \ldots, T(w_n)\}, \quad (5)$$

$$T(case) = \max\{T(w_1), \ldots, T(w_n)\}. \quad (6)$$

With Equation (1) to Equation (6), we get the overall response time and throughput of the whole service composition process, and compare their QoS values by either one of these criteria.

OWL (Web Ontology Language) [5] file expresses preconditions and effects of web services as "concepts" by using ontologies and defines the relations among concepts. In this paper, we use the Web Service Challenge data set [6] to do the experiments. In this data set, input and output parameters of web services are also instances of "concepts", so "concepts" are preconditions and effects of services. Generally, matching degree of services is defined as exact, plug-in, subsume and fail. In this paper, we use exact and plug-in matching degree to define service relations.

Definition 6: Exact match is the most restrictive match, two services are said to be exact match if there is an isomorphism between them, such that: $w \equiv w' \leftrightarrow \forall i \in w_{\text{in}}, i' \in w'_{\text{in}}, i = i' \wedge o \in w_{\text{out}}, o' \in w'_{\text{out}}, o = o'$.

Definition 7: Plug-in match is a relation such that: $w \subseteq w' \leftrightarrow \forall i \in w_{\text{in}}, i' \in w'_{\text{in}}, i' \subseteq i \wedge o \in w_{\text{out}}, o' \in w'_{\text{out}}, o = o' \vee o \subseteq o' \wedge i = i'$.

In this paper, we work on semantic service composition.

Definition 8: Semantic service composition uses OWL ontologies or XML Schema to define services and their relationships to each other [7].

In semantic service composition, we use plug-in matching degree to match services. For a web service in the service repository: for each of its output parameters, we index its directed concept as well as concepts which are parents of this concept as the effects of this service. For each of the service input parameters, we index its directed concept as the precondition of this service. For example, we know "Dog" is a kind of "Mammal", and "Mammal" is a kind of "Animal". If the input parameter of a service is an instance of "Dog", we say the input concept of this service is "Dog". If the output parameter of a service is an instance of "Dog", we say the output concepts of this service are {"Dog", "Mammal", "Animal"}.

B. System architecture

The high level module of the FSIDB system is shown in Figure 3. This system has two main modules: path generation module and path query module.

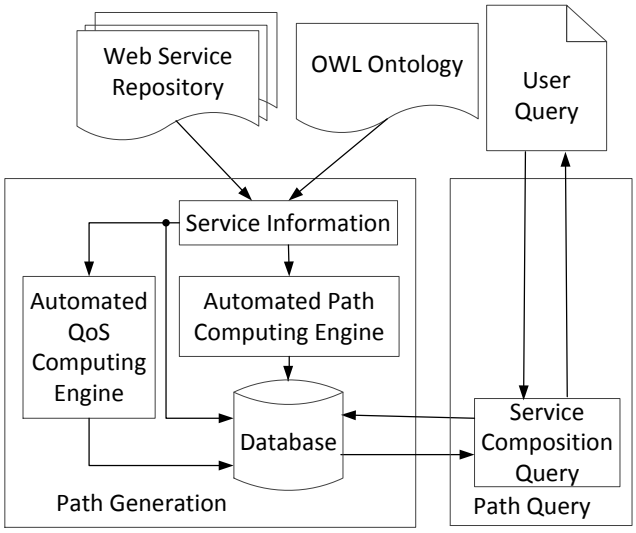

Figure 3. FSIDB system architecture

The path generation module loads WSDL files [8] from the "Web Service Repository" and the ontology definitions from a OWL file, and computes all the possible connection paths among services. The results are stored in a database, the schema of which is shown in Figure 4. More specifically, "Service Information" module extracts the service information, including the input, output concepts, and the QoS value of services. The relations between concepts extracted from the OWL file are stored and used for computing paths. "Automated Path Computing Engine" computes all the possible paths among services. If a service w_1 produces all or a part of the input parameters of another service w_2, w_1 is an input service of w_2 and there is a path that connects them. We would also avoid adding a service to a path which already contains that service. All the inputs (resp. outputs) of services in the newly created path compose the inputs (resp. outputs) of this path. This procedure ends when there is no path with i services. The detailed algorithms in the engine are further explained in Section III. "Automated QoS Computing Engine" calculates the QoS values of each path and the results are stored in the database.

When a user request comes, the path query module queries the database using a SQL statement and return a QoS-aware solution path to the user. This procedure is described further in Algorithm 6.

III. Data structure and algorithm

In this section, we first describe how to generate service combination paths and corresponding QoS values. Then, we show how to search a solution path with the user query, a detailed example will also be given.

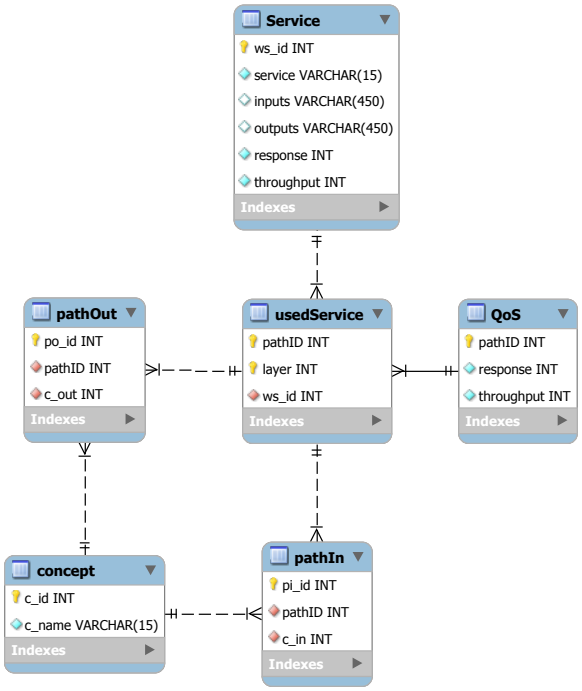

Figure 4. Relational Schema of Database

Example 1: Consider the service composition graph in Figure 1. We create a "ServiceInfo" table as shown in Table I, this table has six columns: service id, service name, input parameters, output parameters, response time and throughput.

Table I
SERVICEINFO TABLE OF FIGURE 1

ws_id	service	inputs	outputs	response	throughput
1	w_1	A, B	C	40	6000
2	w_2	B, C	D	20	4000

A. Find all service combination paths

Algorithm 1 *FindPaths* is the main algorithm to find the all paths ($pathsSet$). Initially, we have a service repository *SR* with all the available services. Algorithm 2 *SPathSetBuild* (line 1) generates paths with one service, then, we repeatedly create paths with i services (where $i > 1$) by joining paths with one service and paths with i-1 services (Algorithm 3 *MulPathSetBuild*), this process ends when there are no paths with i services. Algorithm 4 *AddServiceLayer* and Algorithm 5 *FindLayer* are called to add services in the new path. The QoS value of the new path is calculated in Algorithm 4 via Equation (1) to Equation (6). After all the paths are generated, we store them into the database.

Algorithm 2 *SPathSetBuild* generates paths with one service and this is the first step to create paths. For each service

Algorithm 1 *FindPaths(SR)*

Input: SR: service repository;
Output: $path$: service combinations;
1: $pathsSet \leftarrow SPathSetBuild(SR)$;
2: $i \leftarrow 2$;
3: **while** $(MulPathSetBuild(i-1) \neq \phi)$ **do**
4: $\quad pathsSet \leftarrow pathsSet \cup MulPathSetBuild(i)$;
5: $\quad i \leftarrow i + 1$;
6: **end while**

w in the service repository *SR*, create a new path of w, allocate a unique id to the path, the input (resp. output) concepts of w are inputs (resp. outputs) of the path.

Algorithm 2 *SPathSetBuild(SR)*

Input: SR: service repository;
Output: $sPathSet$: paths with one services;
1: $counter \leftarrow 1$ // unique path id
2: **for** each service w in SR **do**
3: $\quad$ create a new path $path$
4: $\quad path.pathID \leftarrow counter$
5: $\quad path.in \leftarrow w.in$
6: $\quad path.out \leftarrow w.out$
7: $\quad path.w \leftarrow w$
8: $\quad sPathSet \leftarrow sPathSet \cup path$
9: $\quad counter \leftarrow counter + 1$
10: **end for**
11: **return** $sPathSet$

Algorithm 3 *MulPathSetBuild* shows how to generate paths with multiple services; $mulPathSets(i)$ denotes set of paths with i (where $i > 1$, see Algorithm 1) services. For each path $pathS$ with one service in the path set *sPathSet* and each path $pathM$ with i-1 services in path set *mulPathSets(i-1)*, if there is overlap between the outputs of $pathS$ and the inputs of $pathM$, and the service in $pathS$ is not contained in $pathM$, we create a new path by join $pathS$ and $pathM$. We obtain the number of paths in the path set $pathsSet$, this decides the id of the first created path in *mulPathSets(i)*.

Example 2: To generate all the paths in Figure 1, we first create paths with one service (Algorithm 2), in this example, we find two paths with services w_1 and w_2 respectively. Then, we go on to generate paths with two services by join paths with one service (Algorithm 3). The output parameter C of path with pathID=1 is one of the input parameters of path with pathID=2, thus, these two paths can be joined together. As there is no path with three services, Algorithm 1 stops. The PathIn (resp. PathOut) table stores the identification id, path id and input (resp. output) of the path, we show these two tables in Table II and Table III.

Algorithm 4 *AddServiceLayer* adds service to the new path: $pathS, pathM$ denote two paths will be joined and

Algorithm 3 *MulPathSetBuild(i)*

Input: $sPathSet, mulPathSets(i-1)$;
Output: $mulPathSets(i)$: paths with i services;

if $i = 2$ **then**
  $mulPathSets(1) \leftarrow sPathSet$
end if
$counter \leftarrow pathsSet.size + 1$
for each $pathS$ in $sPathSet$ **do**
  for each $pathM$ in $mulPathSets(i-1)$ **do**
    if $pathS.out \cap pathM.in \neq \phi$ and $pathS.w \notin pathM.w$ **then**
      create a new path $path$
      $path.pathID \leftarrow counter$
      $path.in \leftarrow pathS.in \cup pathM.in$
      $path.in \leftarrow pathCur.in/pathS.out$
      $path.out \leftarrow pathS.out \cup pathM.out$
      $path.w \leftarrow AddServiceLayer(path, pathS, pathM)$
      $mulPathSets(i) \leftarrow mulPathSets(i) \cup path$
      $counter \leftarrow counter + 1$
    end if
  end for
end for
return $mulPathSets(i)$

Table II
PATHIN TABLE

pinID	pathID	C_{in}
1	1	A
2	1	B
3	2	B
4	2	C
5	3	A
6	3	B

Table III
PATHOUT TABLE

poutID	pathID	C_{out}
1	1	C
2	2	D
3	3	C
4	3	D

Table IV
USEDSERVICE TABLE

pathID	layer	ws_id
1	1	1
2	1	2
3	1	1
3	2	2

Table V
QOS TABLE

pathID	response	throughput
1	40	6000
2	20	4000
3	60	4000

$path$ denotes the newly created path. We have that $pathM$ has multiple layers of services execute in sequence, each layer has one service or several services be invoked in parallel, as shown in Figure 2. If $pathS$ is added in front of $pathM$, the service in $pathS$ will be the first layer in the new path $path$, and layers (from 1 to k) in $pathM$ will be added as layers (from 2 to $k+1$) in $path$. If not, we first add service layers (from 1 to k) in $pathM$ as layers (from 1 to k) in $path$, then, we check in which layer ($index$) the service of $pathS$ should be added and add this service in the $index$ layer of $path$. The check procedure is done by Algorithm 5. The response time ($path.resp$) and throughput ($path.thp$) of $path$ are also calculated in Algorithm 4 according to Equation (1)-Equation (6).

Algorithm 4 *AddServiceLayer(pathS, pathM, path)*

Input: $pathS, pathM, path$;
Output: $path$;

if $FindLayer(pathS, pathM) = 1$ **then**
  $path.layer(1) \leftarrow pathS.layer(1)$
  for each service layer i of $pathM$ **do**
    $pathCur.layer(i+1) \leftarrow pathM.layer(i)$
  end for
  $path.resp \leftarrow pathS.resp + pathM.resp$
else
  for each service layer i of $pathM$ **do**
    $path.layer(i) \leftarrow pathM.layer(i)$
    if $i = FindLayer(pathS, pathM) - 1$ **then**
      $path.layer(i) \leftarrow path.layer(i) \cup pathS.w$
      if $pathS.resp > pathM.resp$ **then**
        $path.resp \leftarrow pathM.resp + pathS.resp - pathM.layer(i).resp$
      else
        $path.resp \leftarrow pathM.resp$
      end if
    end if
  end for
end if
$path.thp \leftarrow min\{pathS.thp, pathM.thp\}$
return $path$

Algorithm 5 *FindLayer(pathS, pathM)*

Input: $pathS, pathM$;
Output: $index$: the index of the service layer;

$index \leftarrow 1$
for each service layer i of $pathM$ **do**
  if $pathS.out \cap pathM.layer(i).in \neq \phi$ **then**
    $index \leftarrow i$ break
  end if
end for
return $index$

Algorithm 5 *FindLayer* checks in which layer the service of $pathS$ should be added in the new path $path$. We check from the first to the last layer of $pathM$, if there is overlap between the outputs of $pathS$ and the inputs of services in current checked layer of $pathM$, the algorithm stops and returns the index of this service layer.

Example 3: The sequence of services in the newly created path is decided via Algorithm 4 and Algorithm 5 and stored in UsedService table (Table IV), the 3rd and 4th rows in this table denote that the path with pathID=3 has two services: w_1 and w_2. The response time and throughput of this path are calculated in Algorithm 4 and stored in the *QoS table*

(Table V), the 3rd row of this table denotes the path with pathID=3 has the response time of 60 and throughput of 4000.

Suppose n is the number of services, the time complexity of *MulPathSetBuild* is $O(n \times n)$ (line 1, line 2), so the complexity of *FindPaths* is $O(n^3)$.

B. Services composition queries

When a user request comes, we query in the database to find a satisfied composition path. This procedure is done as follows: firstly, find the "PathID" of the path which meet the inputs and outputs requirements of the user query. Then, we rank and filter the returned paths with QoS value. Finally, search in the "UsedService" table for a sequence of services in the path. This procedure is described in Algorithm 6.

Algorithm 6 *Service composition queries*

Input: $inConcepts, outConcept$:user query;
Output: $solServices$:a solution of web services;
1: $pathIDSet \leftarrow$ Index scan on table "pathIn", "pathOut" using user query;
SELECT $pathID$ FROM pathIn WHERE C_{in} in $inConcepts$ AND $pathID$ IN
(SELECT $pathID$ FROM pathOut WHERE $C_{\text{out}} = outConcept$);
2: $solPathID \leftarrow$ Index scan on the QoS table using $pathIDSet$;
SELECT $pathID, MIN(response)$ FROM QoS WHERE $pathID$ IN ($pathIDSet$)
3: $solServices \leftarrow$ Index scan on table "ServiceInfo", "UsedService" using $solPathID$;
SELECT $service$ FROM ServiceInfo WHERE ws_id IN(SELECT ws_id FROM UsedService WHERE $pathID = solPathID$
4: **return** $solServices$

Example 4: Assume the user wants to find service composition with input "A, B" and output "D". The search process of Algorithm 6 is illustrated as follows:

```
SELECT pathID, MIN(response) FROM QoS
WHERE pathID IN(
SELECT pathID FROM pathIn WHERE C_in='A'
AND pathID IN(
SELECT pathID FROM pathIn WHERE C_in='B'
AND pathID IN(
SELECT pathID FROM pathOut WHERE C_out='D'
)));
```

We find the path with minimum response time, then we search services in this path:

```
SELECT service FROM ServiceInfo
WHERE ws_id IN (
SELECT ws_id FROM UsedService
WHERE pathID=3);
```

The final service solution is $\{w_1, w_2\}$.

Lee *et al*. [2] adapt in-memory web service composition algorithm to compute service compositions by joining tables. This method is straightforward. However, intermediate steps are still needed to filter unavailable services in the search process. In contrast, our method only need one query to find a composition solution.

IV. EXPERIMENTAL RESULTS

We run our experiments on a PC platform running 64-bit Windows 7 Operating System with CPU Intel Core i5 480M Processor at 2.93 GHz, and 6 GB RAM.

A. Data set

We use the web service challenge data set [6] to evaluate our work. This data set has 572 services and 5000 concepts. Each web service has around 10 input and 30–40 output concepts.

1) Taxonomy.owl file defines the matching parameters by their semantics.
2) Services.wsdl is the repository of the web services which contains all the available services.
3) Challenge.wsdl is the WSDL contains the request and goal that should be satisfied by the selected solution of composition.

We use MySQL 5.6 as the database. For the initial states and goal, we used the given one by WS-Challenge as Test 1. We also generate random initial states and goal to work as Test 2 and Test 3.

B. Performance analysis

We reproduce the work of Lee *et al*. [2] and compare our experimental result with theirs. In Table VI, we show the experimental results of exact match method, semantic match method of Lee *et al*. [2] and our methods. From Table VI we can see that in Test 1 and Test 2, the exact match cannot return a path with user queries, which means it fails to solve the service composition problem. In Test 1, the semantic match method finds a solution with 48 services, however, we meet the same user query with only 4 services, this shows that the FSIDB system can find a solution with fewer services. In the real world, services may be produced by similar purposes and be used to achieve same or similar functionality, that is, they are redundant [4]. If not handled properly, redundancy wastes time and resources. Besides, our approach can optimize QoS (response and throughput in Table VI), which is ignored by Lee *et al*. [2].

In the composition search process of PSR system [2], paths whose start node is one of the initial states and end node is one of the goals are picked out first. Then, they calculate "no value input parameters" (parameters which are neither provided by the input nor produced by services in the solution, refer to unavailable concepts in Figure 5). After that, they get a set of unavailable services (services whose input parameters are contained in "no value input

Table VI
COMPOSITION SEARCH RESULT

		Test 1	Test 2	Test 3
Exact match [2]	**#path**	0	0	8
	#service	0	0	2
Semantic match [2]	**#path**	4756	5	6
	#service	48	5	2
Our method	**#path**	1	1	1
	#service	4	2	2
	Response[1]	2040	460	620
	Throughput[2]	15000	17000	8000

[1] Response: response time (ms) as a QoS metric
[2] Throughput (invocations per minute) as a QoS metric

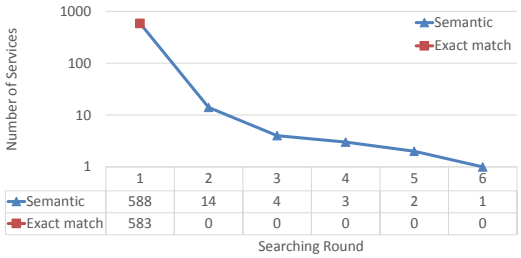

Figure 5. Unavailable input concepts found in each search round of [2]

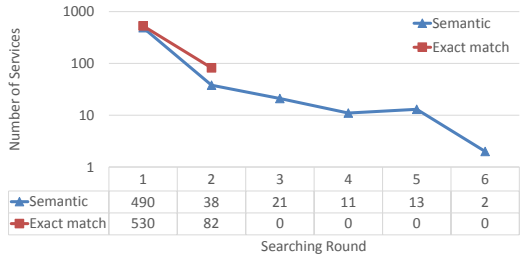

Figure 6. Unavailable web services found in each search round of [2]

parameters"). Finally, find and remove paths which contain unavailable services. The above steps are one round in their search procedure. Check again, if the "no value input parameters" still exists, a new round begins to eliminate unavailable services. This process ends when "no value input parameters" does not exist, services in the remaining paths are returned as the solution. We take Test 1 as an example, Figure 5 and Figure 6 shows the number of "no value input parameters" and unavailable services found in each search round. From these two figures, we can see, though all paths are stored in tables, on average, five to six rounds of checking are still needed to filter services whose inputs are not available in found paths. In contrast, our method only need one query to find a solution.

From the experimental results, we conclude that, the F-SIDB system leads to better performance with fewer services in finding a user satisfying solution.

V. RELATED WORK

Much of work has been done in the realm of QoS-based web services selection [9], [10] and composition [11]–[14].

Zeina *et al.* studies the service discover problem and proposes a four steps service selection approach, in this approach, QoS is classified by means of Formal Concept Analysis (FCA) classification method. The efficiency of their approach is proved by doing experiments on real-world web services [9]. Liu [10] proposes a two stages normalization on a service and QoS criteria matrix. Firstly, the criteria value of each service is divided by the average criteria value, in this process, a maximum value is introduced to guarantee that the division result does not exceed this value. Secondly, the author confiders the relationship between services and quality of group criteria value. The summation of each normalized criteria value defines the final QoS value of a service.

QoS-based service composition problem can be solved in two ways: local optimization and global optimization [11]–[14]. Jiang [11] observes that QoS criteria can help prune the search space in the service composition problem and based on this observation, a QSynth tool is proposed and implemented. Services which fail to provide optimal QoS value and with worse QoS value are pruned in the forward search stage; a backward search is executed to generate the solution path. In local QoS optimization methods, services of differently layers are selected independently and the time complexity is polynomial as a result. However, local optimization may fall into the "local maxima" problem and fail to find the best optimization solution.

Yu *et al.* modules the service selection problem in both the combinatorial and graph model ways, their heuristic algorithms can return near-optimal solutions in polynomial time [12]. Zeng considers multiple criteria in service selection and applies dynamic global optimization method in composition process [13]. The work of Zeng *et al.* [13] and Cui *et al.* [15] use linear integer programming technique to find the optimization solutions. This technique has the drawback of exponentially increased computation complexity and cost with the growing number of web services. Under this consideration, Alrifai and Risse combine global optimization with local selection techniques to solve the problem. They decompose each QoS constraint into a set of local constraints which serve as upper bounds, then, local selection is applied independently. The authors state that their method can find a close to optimal solution while improving the computational time [14]. Recently, researchers propose a similarity-based solution to deal with the situation that no feasible service composition can be found. Among them, relaxation method is an outstanding method [16], [17]. Lin *et al.* proposes a relaxable QoS-based service selection algorithm (RQSS) to find an approximate solution [17]. When used to find a solution, this algorithm relaxes the degree of QoS constraints

and recommends a similar solution in case there is no feasible solution satisfies the constraints of the user.

VI. Conclusions

In this paper, we study the QoS-aware services composition problem and propose a system called FSIDB. In the F-SIDB system, possible service combinations are generated as paths and stored in a relational database, the non-functional criteria-QoS values of the combinations are also calculated and stored before search process. It is the first time that QoS is considered in database based services composition methods. Besides, we support services with multiple input and output parameters, and parallel structures of services are supported by the paths. An elegant feature of our solution is that we only query the database once to find a solution path, no repeated elimination work is needed in the search process, and this might greatly shorten the execution time. The experimental results show that our system can always find a valid services composition solution and maximize user satisfaction.

References

[1] S. Das, E. I. Chong, G. Eadon, and J. Srinivasan, "Supporting ontology- based semantic matching in RDBMS," in *Proceedings of the Thirtieth International Conference on Very Large Data Bases* - Volume 30, 2004, pp. 1054–1065.

[2] D. Lee, J. Kwon, S. Lee, S. Park, and B. Hong, "Scalable and efficient web services composition based on a relational database," *Journal of Systems and Software*, vol. 84, no. 12, pp. 2139–2155, 2011.

[3] D. Chenthati, H. Mohanty, and A. Damodaram, "A scalable relational database approach for web service matchmaking," *IJSSST*, vol. 12, pp. 14–21,2011.

[4] M. Chen and Y. Yan, "Redundant Service Removal in QoS-Aware Service Composition," in *Web Services (ICWS), 2012 IEEE 19th International Conference on*, June 2012, pp. 431–439.

[5] W3C. Owl web ontology language overview. [Online]. Available: http://www.w3.org/TR/owl-features/, 2004.

[6] S. Bleul. (2009) Web service challenge rules. [Online]. Available: http://ws-challenge.georgetown.edu/wsc09/downloads/WSC2009Rules-1.1.pdf (last checked Feb. 2014)

[7] S-C Oh, J-W Yoo, H Kil, D. Lee and S. R T Kumara, "Semantic Web-Service Discovery and Composition Using Flexible Parameter Matching", in *E-Commerce Technology and the 4th IEEE International Conference on Enterprise Computing, E-Commerce, and E-Services, 2007. CEC/EEE 2007. The 9th IEEE International Conference on*, 2007, pp. 533C542

[8] W3c. (2007) web services description language (wsdl) version 2.0. [Online]. Available: http://www.w3.org/TR/wsdl20/.

[9] Z. Azmeh, M. Driss, F. Hamoui, M. Huchard, N. Moha, and C. Tiber- macine, "Selection of composable web services driven by user require- ments," in *Web Services (ICWS), 2011 IEEE International Conference on*, July 2011, pp. 395–402.

[10] Y. Liu, A. H. Ngu, and L. Z. Zeng, "Qos computation and policing in dynamic web service selection," in *Proceedings of the 13th International World Wide Web Conference on Alternate Track Papers; Posters*, ser. WWW Alt. 04. New York, NY, USA: ACM, 2004, pp. 66–73.

[11] W. Jiang, C. Zhang, Z. Huang, M. Chen, S. Hu, and Z. Liu, "Qsynth: A tool for qos-aware automatic service composition," in *Web Services (ICWS), 2010 IEEE International Conference on*, July 2010, pp. 42–49.

[12] T. Yu, Y. Zhang, and K.-J. Lin, "Efficient algorithms for web services selection with end-to-end qos constraints," *ACM Trans. Web*, vol. 1, no. 1, May 2007.

[13] L. Zeng, B. Benatallah, M. Dumas, J. Kalagnanam, and Q. Z. Sheng, "Quality driven web services composition," in *Proceedings of the 12th International Conference on World Wide Web*, New York, NY, USA: ACM, 2003, pp. 411–421.

[14] M. Alrifai and T. Risse, "Combining global optimization with local selection for efficient qos-aware service composition," in *Proceedings of the 18th International Conference on World Wide Web*, New York, NY, USA: ACM, 2009, pp. 881–890.

[15] L. Cui, S. Kumara, and D. Lee, "Scenario analysis of web service composition based on multi-criteria mathematical goal programming," *Service Science*, vol. 3, no. 4, pp. 280–303, December 2011.

[16] R.-K. Sheu, W.-T. Lo, C.-F. Lin, and S.-M. Yuan, "Design and implementation of a relaxable web service composition system," in *Cyber- Enabled Distributed Computing and Knowledge Discovery (CyberC), 2010 International Conference on*, Oct 2010, pp. 448–455.

[17] C.-F. Lin, R.-K. Sheu, Y.-S. Chang, and S.-M. Yuan, "A relaxable service selection algorithm for qos-based web service composition," *Information and Software Technology*, vol. 53, no. 12, pp. 1370–1381, 2011.