Full Solution Indexing for top-K Web Service Composition

Jing Li, Yuhong Yan, *Member, IEEE,* and Daniel Lemire, *Member, IEEE,*

Abstract—Automated service composition fulfills complex tasks by combining different existing web services together. Unfortunately, optimizing service composition is still a challenging area that needs to be addressed. In this article, we propose a novel relational database approach for automated service composition. All possible service combinations are generated beforehand and stored in a relational database. When a user request comes, our system composes SQL queries to search in the database and return the best Quality of Service (QoS) solutions. We test the performance of the proposed system with a web service challenge data set. Our experimental results demonstrate that this system can always find top-K valid solutions to satisfy user's functional and non-functional requirements.

Index Terms—QoS-aware service composition; top-K; database;

1 Introduction

To survive in the competing business environment, enterprises may focus on their core services and rush to find other services via Internet. Service-oriented architecture (SOA) combines existing services on the Internet as on-demand software systems [1]. Compared with traditional methods which program new systems from scratch to satisfy customers, SOA helps lower costs of software development and management.

In general, given a user request and goal, the web service composition (WSC) problem is to combine together different web services to fulfill user's complex business requirements. We demonstrate a real scenario of a travel plan to show our motivation (Figure 1).

1) A tourist wants to travel from Montreal to Chicago. First, he describes his requirement in a "Service Description" file. The tourist needs a round trip flight between these two cities, then, he needs a hotel room while staying in Chicago, he wants a list of recreation e.g., movie theatre, museum, shopping mall. As no individual web service provides all the information he wants, we solve the problem by combining different services as a travel plan.
2) We choose services in the "Service Repository", construct a search graph from initial states to the goal with chosen services.
3) We extract a solution from the search graph. Different services may have same functionality, in this situation, we select services based on user's non-functional requirements.
4) We combine chosen services as a new service, return this service as an executable solution to the user.

- *J. Li and Y.Yan are with the Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada. E-mail: jing.li.hnu@gmail.com, yuhong@encs.concordia.ca*
- *D. Lemire is with LICEF Research Center, TELUQ, Université du Québec. E-mail:lemire@gmail.com*

In addition, recent research considers non-functional requirements of services: Quality of Service (QoS), such as availability, response time, throughput, reliability and so on. QoS measures how well a service serves the user. The selection of the composite service with the best QoS value is called QoS-aware service composition problem. Integer Linear programming (ILP) has been applied to solve this problem [2], [3], [4]. However, ILP may lead to exponentially increased computation complexity and cost when the number of variables increases. Considering unexpected long delay is not allowable in real e-business scenario, many researchers solve the problem with local selection strategy as a compromise [5], [6]. In local selection methodology, services in different layers are selected independently, this guarantees the problem is solved in polynomial time, but it is easily falling into the "local maximum" problem. (A local maximum is a maximum within some neighbor but not the global maximum [7].)

Many in-memory approaches based on different kinds of techniques are used to solve the composition problem. These include A* methods [8], [9], [10], beam-stack search [11] and planning-graph model [12], [13]. A planning-graph approach is a powerful method to solve the composition problem. This approach includes two stages: a forward expand stage constructs a search graph and a backward searching stage retrieves a solution. We show a set of available services in table 1, the planning search graph of this table is shown in figure 2. This graph contains two kinds of layers: the proposition (P) layers contain parameters and action (A) layers contain services. To construct a search graph, first, we add user's initial states to P_0 layer, then, search the service repository for services whose input parameters are satisfied in P_0 layer. These services are named as available services and added into A_0 layer. All parameters in P_0 layer and available services' outputs are added into P_1 layer, so P_1 layer is a superset of P_0 layer. We extend the search graph layer by layer, this process ends when the graph reaches a goal layer or no more services can be added into the graph. If the goal states can not be found in the search graph, the

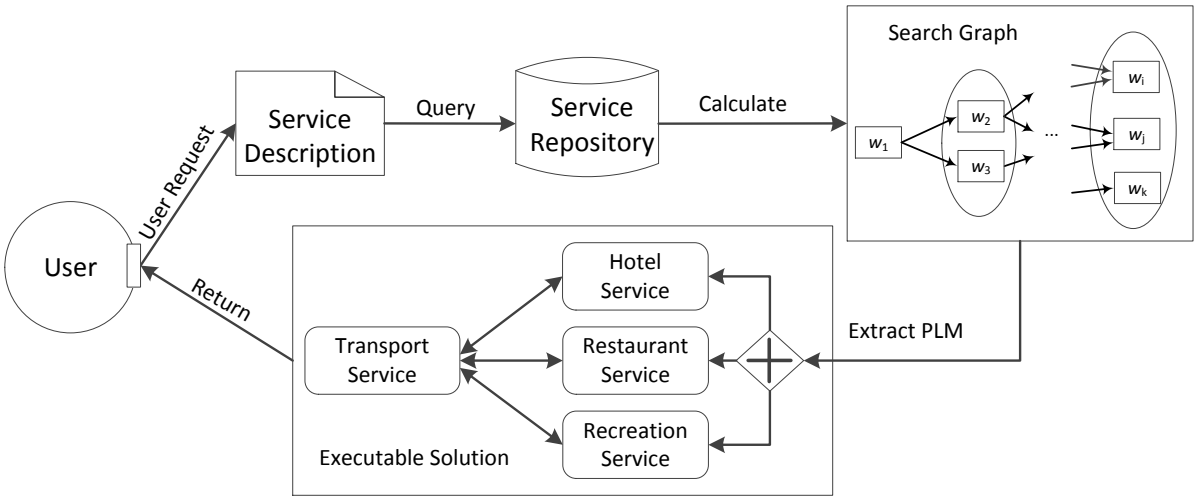


Fig. 1. A Web Service Composition Example.

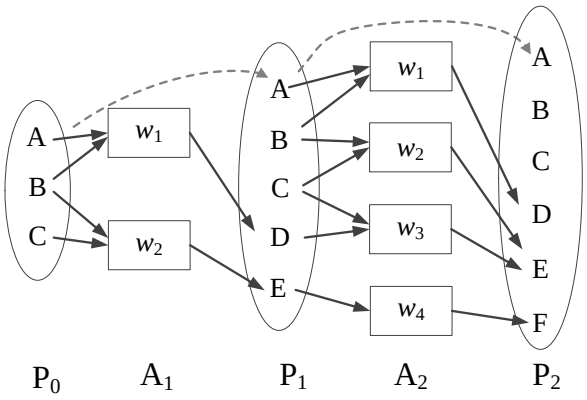


Fig. 2. A Planning Graph.

problem can not be solved, otherwise, the problem can be solved. A backward searching stage from the goal layer to the initial layer is applied to retrieve a solution. To find the solution with the best QoS value, we need to check all services' combinations, the complexity of the backward searching is NP-complete.

TABLE 1
A set of available services

Service	Input	Output
w_1	A, B	D
w_2	B, C	E
w_3	C, D	E
w_4	E	F

For each user request, in-memory approaches construct a necessary graph of service connections and search the graph to find a solution. To solve N different user requests, N graphs are constructed. This is time consuming. Besides, in-memory approaches can only work when data fits in RAM, the search space is limited by the size of physical memory. This makes in-memory approaches non-scalable. Last but not least, if a service in the solution is broken, the solution is no longer available. The search graph is reconstructed for an alternative solution.

The authors in [14] model a relational database as a graph in which tuples and foreign keys are represented as nodes and edges respectively. A parameterized solution is proposed to find the minimum cost connected tree. Unfortunately, this method does not hold for our problem. As in our problem, instead of finding the minimum cost or shortest path, we need to find a path whose output is a superset of the goal.

Ding *et al.* study the time-dependent shortest-path problem [15]. To solve this problem, they design a DIJKSTRA-based algorithm to calculate the earliest arrival time of each node. A linear-time algorithm is designed to compute the optimal path from originate to the terminate. Yang *et al.* discuss the shortest-path problem with time constraint in time-dependent graphs [16]. A TWO-STEP-SEARCH algorithm is proposed to solve this problem. First, they compute the minimum cost from the starting point to each other vertex by using the single-source shortest path algorithm. After that, they calculate the optimal waiting time for each vertex as the second step. These two methods aim at finding the shortest path and are not suitable in our situation.

Attempts to resolve the service composition problems have resulted in the utilization of relational database [17], [18], [19]. They use "join" operators to connect matched services, these services are seen as new available services. The PSR (Pre-computing Solutions in RDBMS) system developed by Lee uses joins and indices to connect services as paths and stores paths in a database [17]. The PSR system abstracts services as single operators, while handling user queries with multiple inputs and outputs, this system returns all paths which meet part of user requests. Redundant services may contain in the returned solution and increase user's cost (A service is redundant if all its output parameters used by other services can be provided by other services [18]). The system developed by [19] assumes all the inputs of a service can be provided by another service. However, in real world, normally a service only provides part inputs of another service.

In this paper, we propose a novel approach for the QoS-aware service composition problem, which is called FSIDB (Full Solution Indexing using Database). That is, all possible service combinations are generated beforehand and stored in a relational database. When a user request comes, the FSIDB system composes SQL queries to search in the database and find K best QoS solutions (top-K solutions). Compared with in-memory methods, the service combinations are stored in the database and are therefore reusable. Besides, our system takes advantage of large space available on persistent disk. Last but not least, we find and rank top-K paths according to the comparison of their QoS values, these paths provide backup solutions. For example, when searching a flight from Montreal to Chicago, we may get a list of flight numbers provided by different companies. We pick 5 flights as top-5 solutions ranked by prices from low to high. If the flight with the lowest price is cancelled, the user may book the flight with second cheapest price.

The key contributions of our work are as follows:

- We address the quality of services and calculate globally optimized QoS values for web service compositions.
- We use a relational database to solve web service composition problem with a large number of web services and complex operators. The services' combinations stored in the database are reusable. We present algorithms to update the database e.g., service disappearance and addition, and analyse time complexity of these operations.
- We fetch solutions by converting the services composition requests into SQL statements, our system supports several ways of searching. In case the optimal solution is not available, we fetch top-K solutions to provide backup solutions to the user. Also, we support threshold query on multiple QoS criteria.

The rest of this paper is organized as follows: In Section 2, we introduce the preliminary knowledge. The system architecture and algorithms are given in Section 3. We demonstrate a case study in Section 4 to further explain our algorithms. Experimental results are shown in Section 5. Related work is reviewed in Section 6 and the conclusion is drawn in Section 7.

This article is an extended version of a conference paper [20], this version presents upgraded algorithms and experiments. We optimized the database schema, using fewer tables and join operations, and as a result, the query times have improved.

2 PRELIMINARY

In this section, we formalize definitions of web services and web service composition.

Definition 1. An ontology rooted tree is a tuple (C, R) with the following components:

- C is a finite set of concepts represented as nodes in the rooted tree.
- R is a set of direct inheritance relationships represented as edges in the rooted tree.

$\forall c_1, c_2 \in C, c_1 \rightarrow c_2 \Leftrightarrow c_2$ is a child or descendant of c_1.
Transparency: $\forall c_1, c_2, c_3 \in C, c_1 \rightarrow c_2 \wedge c_2 \rightarrow c_3 \Rightarrow c_1 \rightarrow c_3$.

An Ontology rooted tree is constructed according to the similarities and differences among different entities, it is an essence to enable semantic interpretation.

Definition 2. a web service w is 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 7.
- Q is a finite set of non-functional criteria for w.

Normally the description of a service has two components: functional and non-functional characteristics. Non-functional requirements—QoS criteria determine usability and utility of a service [1]. We consider the following generic QoS criteria of services.

1) *Availability*: the availability (A) of a service is the probability a service is available, it is calculated as $A(w) = T(w)/\theta$, T is the time period in which the service w is available, θ is a constant of time period [21].
2) *Performance*: it is measured with response time (R) or throughput (T). Response time in a data system is the interval between the arrival of the request and the beginning of delivery the response (unit: milliseconds). Throughput is the average rate of successful message delivered per time over a communication channel (unit: requests/min).
3) *Reliability*: the reliability (RB) of a service is the ability to serve correctly despite system or network failures. It is measured by the number of transactional successes per month (unit: successes/month).
4) *Cost*: the cost (C) of a service is the amount of money paid to the service provider to use the service (unit: cents).

Among these criteria: availability, throughput, and reliability are positive criteria, the higher the value, the higher the quality. Meanwhile, response time and cost are negative criteria, the higher the value, the lower the quality.

Besides single QoS criterion, we would like to use a utility value to describe multiple QoS criteria as a whole. In this paper, we apply the Multiple Criteria Decision Making (MCDM) approach [22] to obtain this utility QoS value. First, we scale the value of a QoS criterion i for a service w_j. For a positive criteria, this value is scaled with Equation (1), for a negative criteria, this value is scaled with Equation (2).

$$U_i(w_j) = \begin{cases} \dfrac{Q_i(w_j) - Q_i^{min}}{Q_i^{max} - Q_i^{min}} & \text{if} \quad Q_i^{max} - Q_i^{min} \neq 0 \\ 1 & \text{if} \quad Q_i^{max} - Q_i^{min} = 0 \end{cases} \quad (1)$$

$$U_i(w_j) = \begin{cases} \dfrac{Q_i^{max} - Q_i(w_j)}{Q_i^{max} - Q_i^{min}} & \text{if} \quad Q_i^{max} - Q_i^{min} \neq 0 \\ 1 & \text{if} \quad Q_i^{max} - Q_i^{min} = 0 \end{cases} \tag{2}$$

The utility quality score of a service w_j is calculated in Equation (3).

$$U(w_j) = \sum_{i=1}^{n} U_i(w_j) \times W_i \tag{3}$$

Here, W_i represents the weight of QoS criteria i, $W_i \in [0, 1]$ and $\sum_{i=1}^{n} W_i = 1$.

Definition 3. 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.

Now we explain how to connect services, two kinds of control structures are applied in this paper: sequence and flow. Suppose services are represented by $w_1, w_2, \ldots, w_n$. Services in sequence control structure are invoked one by one ($w_1; w_2; \ldots; w_n$). Services in flow control are invoked in parallel ($w_1||w_2||\ldots||w_n$). We may use single QoS dimension to express the performance of a composition and compare QoS values of different compositions by either one of the criteria.

$$A(sequence) = A(flow) = \prod_{n=1}^{N} A(w_n). \tag{4}$$

$$R(sequence) = \sum_{n=1}^{N} R(w_n), \tag{5}$$

$$R(flow) = \max\{R(w_1), \ldots, R(w_n)\}. \tag{6}$$

$$T(sequence) = T(flow) = \min\{T(w_1) \ldots, T(w_n)\}. \tag{7}$$

$$RB(sequence) = RB(flow) = \prod_{n=1}^{N} RB(w_n). \tag{8}$$

$$C(sequence) = C(flow) = \sum_{n=1}^{N} C(w_n). \tag{9}$$

The single QoS criterion can be calculated with Equation (4) to Equation (9). Combine Equation (3) to Equation (9), we can get the overall QoS score of a service composition:

$$U(S) = \sum_{i=1}^{n} U_i(S) \times W_i \tag{10}$$

Definition 4. A graph is represented as $G = (W, E)$, where W is the vertex set and E is the edge set. The vertex set of graph $W = \{W_1, W_2, ..., W_k\}$ is partitioned into k subsets, each set W_i is called a layer, such that: for every edge $(u, v) \in E$ with $u \in W_i$ and $v \in W_j$ implies $\mid i - j \mid \leq 1$.

A graph with a layer is a layered graph. A layer is proper if all edges are between vertices in adjacent layers. Please note that services do not provide inputs to other services in the same service layer.

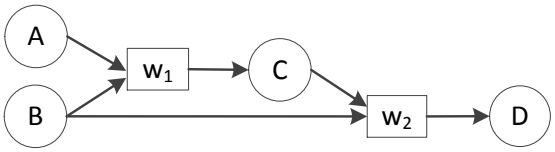


Fig. 3. A Layered Graph Example.

Figure 3 is an example of a layered graph represents the connection of services. We use circles to represent parameters and use rectangles to represent services. In this figure, we have four parameters (A, B, C, D) and two services (w_1, w_2). If a parameter p is one of the inputs or the outputs of a service w, there is an edge between p and w. The arrows represent the input and the output relations between parameters and services. In this figure, the output parameter C of service w_1 is an input parameter of w_2, thus, we say w_1 is an input service of w_2.

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

- $path_{\text{in}} = \{\bigcup_{k=1}^{l}\{w_{i.in} | w_i \in W_k\}\} - \{\bigcup_{k=1}^{l}\{w_{j.out} | w_j \in W_k\}\}$ is a finite set of typed input parameters.
- $SL = \{W_k | k = 1 : l\}$ 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 inputs of the path or the outputs of preceding layers.
- $path_{\text{out}} = \{\bigcup_{k=1}^{l}\{w_{i.out} | w_i \in W_k\}\}$ is a finite set of typed output parameters.
- Q is a finite set of quality criteria.

A web service combination C can be mapped to a path. To solve the web service composition problem, we need to find a path between user request and goal, *i.e.,* satisfying the following two conditions:

- $C_{\text{in}} \supseteq path_{\text{in}}$
- $C_{\text{out}} \subseteq path_{\text{out}}$

Three paths can be found in Figure 3 and are listed in table 2.

TABLE 2
Paths in Figure 3

ID	$path_{\text{in}}$	SL	$path_{\text{out}}$
1	A, B	$\{w_1\}$	C
2	B, C	$\{w_2\}$	D
3	A, B	$\{\{w_1\}, \{w_2\}\}$	C, D

When there are multiple solutions, we want to rank the solutions by their QoS values and return K best solutions.

Definition 6. Top-K best solutions: while solving a web service composition problem, we return the ranked top-K paths based on user's QoS constraints. Each of the path satisfies the condition: $\{C_{\text{in}} \supseteq path_{\text{in}}\} \wedge \{C_{\text{out}} \subseteq path_{\text{out}}\}$.

For example, suppose in figure 3 the user specifies his requirement $(C_{\text{in}}, C_{\text{out}}) = (\{A, B\}, \{C\})$, we find two paths ID = 1 and ID = 3 (table 2) meet the requirements.

Syntactic matching without considering semantics may lead to unsuitable solutions returned. Imaging a user is looking for a travel service, the system takes destination, duration as input, and returns a recommended travel package as output. Since the service is requested without specifying its precondition and effect, the system may compose a conference searching service, which returns the information of conference e.g., held place of the conference, duration and registration fee. Unfortunately, this does not carry out user's intention.

To clearly understand users' requirements and improve correctness of the solution, researchers introduce semantic feathers in service composition area. Semantic web service matchmaking overcomes limitations of syntactic matching and can be used to further enhance efficiency of service discovery. This is achieved by OWL-S (Web Ontology Language for Web Services) description language [23]. OWL-S allows logic inference while matching web services. We use an OWL ontologies file to define relationships among services in the repository. An OWL file lists "concepts" (Classes), "things" (Instances)and the relationship between them, it is one of the most important languages for specifying ontologies. Ontologies are used to define relations among concepts in an OWL file.

The use of semantic information allows the implementation of semantic service composition. In this paper, we use the Web Service Challenge data set [24] 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, the matching degrees of services are defined as exact, plug-in, subsume and fail. Two services are said to be exact match if their inputs and outputs are exactly the same, it is the most restrictive match. Plug-in match means the input set of a service (w) is a subset of another service's (w') input set or the output set of w' is a subset of output set of w.

Definition 7. Plug-in match can be represented as follows: $w \subseteq w' \leftrightarrow \forall i \in w_{\text{in}}, i' \in w'_{\text{in}}, o \in w_{\text{out}}, o' \in w'_{\text{out}}, i' \subseteq i \wedge o = o' \vee o \subseteq o' \wedge i = i'$.

In this paper, we use plug-in matching degree to match services. We use an example to show how to match services with plug-in match. There are three concepts in our example :"Dog", "Mammal" and "Animal". "Dog" is a kind of "Mammal", and "Mammal" is a kind of "Animal". We build an ontology tree to represent relationships of these concepts. Service w_1 has an output parameter which is an instance of "Dog". We extend the output set of w_1, so it contains ancestors of "Dog" in the ontology rooted tree, thus, the output of w_1 is extended to {"Dog", "Mammal", "Animal"}. Web service w_2 has an input parameter which is an instance of "Animal", we say the input concept of w_2 is "Animal". Two services w_1 and w_2 can be connected if they are exact or plug-in match. Because the input of w_2 is a subset of the output of w_1, w_1 and w_2 can be connected. To avoid checking the semantic relationships every now and then, we build an indexing table as shown in Table 3.

TABLE 3
An example of indexing table

Service	Input Concepts	Output Concepts
w_1	...	Dog, Mammal, Animal
w_2	Animal	...

3 Architecture and Algorithm

In this section, we describe the framework of our proposed FSIDB system. Figure 4 shows an architecture overview of our system.

Preparation. Service provider publishes web services with WSDL interfaces. These services are loaded into the "Web Service Repository" file, in which the functional properties and QoS values of services are registered. The OWL ontology file defines service ontologies and can be parsed by an OWL parser. The OWL ontology is used to infer semantic relationships among matching services. The information of services such as the name, input, output concepts and the QoS values are stored in the "Service Information" module.

Path Generation. The "Path Generation" module computes and stores all the possible connected paths and their corresponding QoS values. The generated paths are stored in a relational database, the schema of which is shown in Figure 5. More specifically, "Automated Path Computing Engine" computes all the possible paths. If the output of a service w_1 is an input of another service w_2, w_1 is an input service of w_2 and there is a path connects them. We avoid adding a service to a path which already contains that service. The inputs of a service in a newly created path can be provided by either the inputs of the path or the outputs of a service in a proceeding service layer. All the outputs of services in the path compose the outputs of this path. This procedure ends when there is no more paths can be generated. Similarly, "Automated QoS Computing Engine" calculates QoS values of paths according to algorithm 6.

Path Query. Up to now, all the service combinations and their relevant QoS values are stored in the relational database. When the user specifies service composition requirement, "QoS Requirement Standardization" module analyzes user's QoS requirement (the details are explained with an example in Subsection 3.2). After that, a SQL path query is generated. We use this SQL statement to query the database, a list of QoS-aware solution paths are returned to the user. This procedure is described further in Subsection 3.2.

3.1 Path Generation

Algorithm 1 **PathsBuild** is the main algorithm. First, we generate paths with only one service (line 1). Then, we repeatedly generate paths with multiple services (line 4). This process ends when no more paths can be generated (line 6). In this paper, we suppose there is no loop among services. To make it more clear, if service w_1 is an ancestor service of w_2, w_2 cannot be an ancestor of w_1.

Algorithm 2 **SPathSetBuild** generates paths with one service. Originally, all available services are stored in a service repository SR. For each service w in SR, we create a new path $path$ (line 2-4). A unique id is allocated to $path$, the

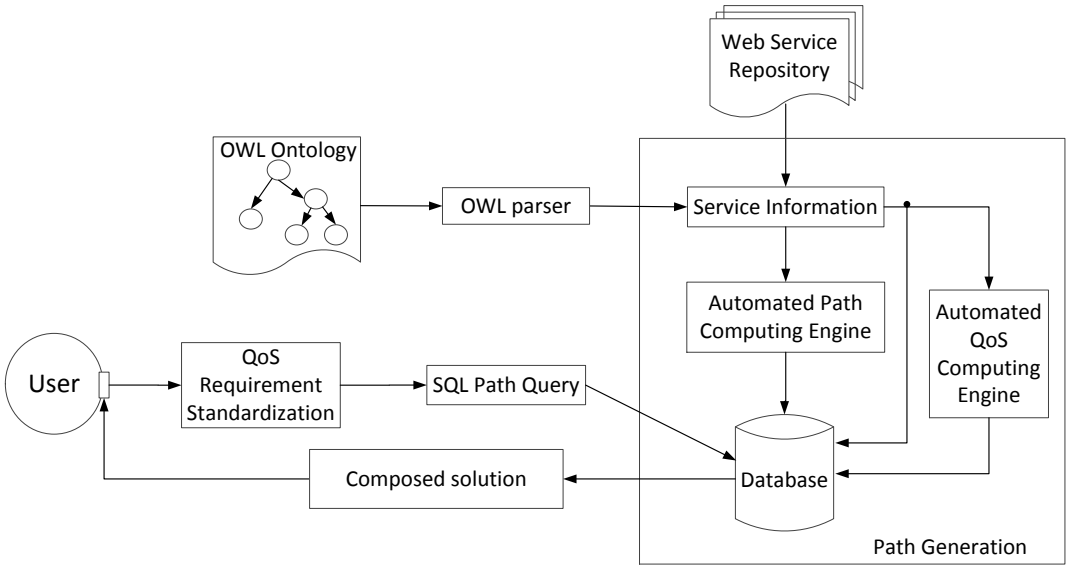


Fig. 4. Architectural Overview.

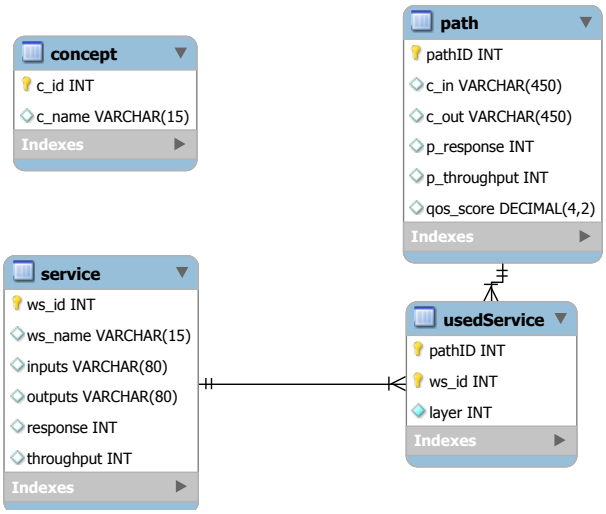


Fig. 5. Schema of Relational Database.

input (resp.output) concepts of w are inputs (resp.outputs) of $path$ (line 5-7).

Algorithm 3 **MulPathSetBuild** generates paths with multiple services. $mulPathSets(i)$ denotes a set of paths with i ($i \geq 2$) services. The number of generated paths decides the unique id of the first created path in this set (line 4). For each path $pathS$ with one service and $pathM$ with $i-1$ services, if the outputs of $pathS$ and inputs of $pathM$ have overlaps, and $pathM$ does not contain the service of $pathS$, we create a new path $path$ by connecting them together (line 5-8). The order of service layers of $path$ is decided by Algorithm 5.

Algorithm 4 **FindLayer** checks in which layer $PathS$ can

Algorithm 1 *PathsBuild*

Input: SR: service repository;
Output: $pathsSet$: a set of paths;
1: $pathsSet \leftarrow SPathSetBuild(SR)$;
2: $i \leftarrow 2$;
3: **repeat**
4: $\quad pathsSet \leftarrow pathsSet \cup MulPathSetBuild(i)$;
5: $\quad i \leftarrow i+1$;
6: **until** $(MulPathSetBuild(i) = \phi)$;

Algorithm 2 *SPathSetBuild*

Input: SR: service repository;
Output: $sPathSet$: a set of paths with one service;
1: $counter \leftarrow 1$ // unique path id
2: **for** each service w in SR **do**
3: $\quad$ create a new path $path$
4: $\quad path.w \leftarrow w$
5: $\quad path.pathID \leftarrow counter$
6: $\quad path.in \leftarrow w.in$
7: $\quad path.out \leftarrow w.out$
8: $\quad path.resp \leftarrow w.resp$
9: $\quad path.thp \leftarrow w.thp$
10: $\quad path.cost \leftarrow w.cost$
11: $\quad path.avail \leftarrow w.avail$
12: $\quad path.relib \leftarrow w.relib$
13: $\quad sPathSet \leftarrow sPathSet \cup path$
14: $\quad counter \leftarrow counter + 1$
15: **end for**
16: **return** $sPathSet$

Algorithm 3 *MulPathSetBuild(i)*

Input: $sPathSet, mulPathSets(i-1)$;
Output: $mulPathSets(i)$: a set of paths with i services;
1: **if** $i = 2$ **then**
2: $mulPathSets(1) \leftarrow sPathSet$
3: **end if**
4: $counter \leftarrow pathsSet.size + 1$
5: **for** each $pathS$ in $sPathSet$ **do**
6: **for** each $pathM$ in $mulPathSets(i-1)$ **do**
7: **if** $pathS.out \cap pathM.in \neq \phi$ and $pathS.w \notin pathM.w$ **then**
8: create a new path $path$
9: $path.w \leftarrow AddLayer(pathS, pathM, path)$
10: $path.pathID \leftarrow counter$
11: $path.in \leftarrow pathS.in \cup pathM.in$
12: $path.in \leftarrow path.in \setminus pathS.out$
13: $path.out \leftarrow pathS.out \cup pathM.out$
14: *CalculateQoS(pathS, pathM, path)*
15: $mulPathSets(i) \leftarrow mulPathSets(i) \cup path$
16: $counter \leftarrow counter + 1$
17: **end if**
18: **end for**
19: **end for**
20: **return** $mulPathSets(i)$

be added into $pathM$. We check from the first service layer of $pahM$ (line 1), if there is overlap between the outputs of $pathS$ and the inputs of current service layer i of $pathM$ (line 3) , the algorithm stops and returns the index of current layer (line 8).

Algorithm 4 *FindLayer*

Input: $pathS, pathM$;
Output: $index$: the index of the service layer;
1: $i \leftarrow 1$
2: **for** each service layer i of $pathM$ **do**
3: **if** $(pathS.out \cap pathM.layer(i).in) \neq \phi$ **then**
4: $index \leftarrow i$
5: break
6: **end if**
7: **end for**
8: **return** $index$

Algorithm 5 **AddLayer** decides the order of services layers in the newly created path. If $pathS$ can be added in front of $pathM$ (line 2), service of $pathS$ is added as the first layer of $path$ (line 3), service layers (from 1 to k) of $pathM$ are added as layers (from 2 to $k+1$) of $path$ (line 4-5). If not, layers from 1 to k of $pathM$ are added into $path$ as layers from 1 to k (line 8-9), then we check in which layer the service of $pathS$ should be added and add it into $path$ (line 10-11).

Algorithm 6 **CalculateQoS** calculates QoS values of the new path $path$. The response time ($path.resp$), throughput ($path.thp$), cost ($path.cost$), availability ($path.avail$) and reliability ($path.relib$) of $path$ are calculated according to Equation (4)- Equation (9).

Table 4 gives an example to show how to calculate the response time of the new path. Row 1 and row 2 show two

Algorithm 5 *AddLayer*

Input: $pathS, pathM, path$;
Output: $path.w$;
1: $f \leftarrow FindLayer(pathS, pathM)$
2: **if** $f = 1$ **then**
3: $path.layer(1).w \leftarrow pathS.layer(1).w$
4: **for** each service layer i of $pathM$ **do**
5: $path.layer(i+1).w \leftarrow pathM.layer(i).w$
6: **end for**
7: **else**
8: **for** each service layer i of $pathM$ **do**
9: $path.layer(i).w \leftarrow pathM.layer(i).w$
10: **if** $i = f - 1$ **then**
11: $path.layer(i).w \leftarrow path.layer(i).w \cup pathS.w$
12: **end if**
13: **end for**
14: **end if**
15: **return** $path$

Algorithm 6 *CalculateQoS*

Input: $pathS, pathM, path$;
Output: $path$;
1: $f = FindLayer(pathS, pathM)$
2: **if** $f = 1$ **then**
3: $path.resp \leftarrow pathS.resp + pathM.resp$
4: **else**
5: $i = f - 1$
6: **if** $pathS.resp > pathM.layer(i).resp$ **then**
7: $path.resp \leftarrow pathM.resp + pathS.resp - pathM.layer(i).resp$
8: **else**
9: $path.resp \leftarrow pathM.resp$
10: **end if**
11: **end if**
12: $path.thp \leftarrow min\{pathS.thp, pathM.thp\}$
13: $path.cost \leftarrow pathS.cost + pathM.cost$
14: $path.avail \leftarrow pathS.avail \times pathM.avail$
15: $path.relib \leftarrow pathS.relib \times pathM.relib$
16: **return** $path$

paths before connection, we discuss three possibilities for connection.

Case 1: $FindLayer = 1$, $pathS$ is added before $pathM$, according to Algorithm 6 (line 3), $path.resp = pathS.resp + pathM.resp$.

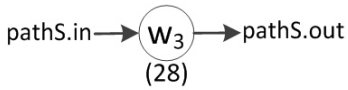
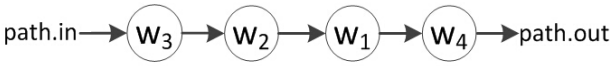
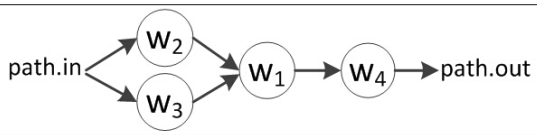
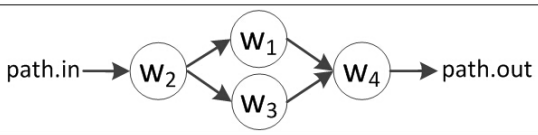
Case 2: $FindLayer = 2$, service of $pathS$ (w_3) is added in the first layer and occurs in parallel with w_2, $pathS.resp < pathM.layer(1).resp$, according to Algorithm 6 (line 9), $path.resp = pathM.resp$.

Case 3: $FindLayer = 3$, service of $pathS$ (w_3) is added in the second layer and executed in parallel with w_1, $pathS.resp > pathM.layer(2).resp$, according to Algorithm 6 (line 7), $path.resp = pathS.resp + pathM.resp - pathM.layer(2).resp$.

3.2 Path Query

After all paths are generated, we query the database for service composition solutions. First, we standardize user's

TABLE 4
Generate new paths.

Before connection		*pathS.resp*=28
		*pathM.resp*=90
After connection		*path.resp*=118
		*path.resp*=90
		*path.resp*=93

QoS requirement, to make sure there is no confusion for the FSIDB system.

The process of "QoS requirement standardization" is shown in Figure 6. When the user's QoS requirement comes, we first check whether or not it is an extremum QoS criterion. e.g. "The user wants a solution with the cheapest price", we return a solution with the minimum price. Otherwise, we check whether or not the user gives his threshold value for the QoS criteria, e.g. "price less than $500", in this situation, a set of satisfied solutions are returned. If the requirement is dim, for example "a cheap price", we automatically pick a threshold value for the user and return satisfied solution to the user.

Then, we generate SQL statement to query the database. The query procedure is done as follows: Firstly, find a set of "PathID" of paths which meet the inputs and outputs requirements of the user. Then, rank and filter the returned paths with their QoS values according to the given threshold of the user. Finally, search in the "UsedService" and "service" table for services in the path. This procedure is detailed described in Algorithm 7.

Algorithm 7 *Service composition queries*

Input: $inConcepts, outConcepts, constraints$: user query and QoS constraints;
Output: $solServices$: top-K solutions of web services;
1: $solPathIDs \leftarrow$ Index scan on table "path" and "concept" using user query;
SELECT $pathID$ FROM path WHERE c_in IN(
SELECT c_id FROM concept WHERE c_name IN '$inConcepts$') AND c_out IN(
SELECT c_id FROM concept WHERE c_name LIKE '$\%outConcepts\%$')
ORDER BY $constraints$ ASC LIMIT K
2: $solServices \leftarrow$ For each path in $solPathIDs$, index scan on table "service", "usedService";
SELECT ws_name FROM service WHERE ws_id IN(
SELECT ws_id FROM usedService WHERE $pathID = solPathID$)
3: **return** $solServices$

3.3 Database Update

Since services on the network always change, e.g., new services being added to the network, old services fail to work or disappear, database updating is also important. In this subsection, we shall discuss how to add and remove services in the database.

3.3.1 Service Disappearance

When a service disappears, we find and delete paths which contain this service. That is, to delete relative records in table "path" "usedService" and "service". This procedure is described in details by Algorithm 8.

Algorithm 8 *Delete service*

Input: w: disappeared service's name;
1: delete paths which contain w ;
DELETE FROM path WHERE pathID IN(
SELECT pathID FROM usedService WHERE ws_id IN(
SELECT ws_id FROM service WHERE ws_name = 'w'))
2: delete records in "usedService" which contain w;
DELETE FROM usedService WHERE ws_id IN(
SELECT ws_id FROM service WHERE ws_name = 'w')
3: delete records with w from table "service";
DELETE FROM service WHERE ws_name = 'w'

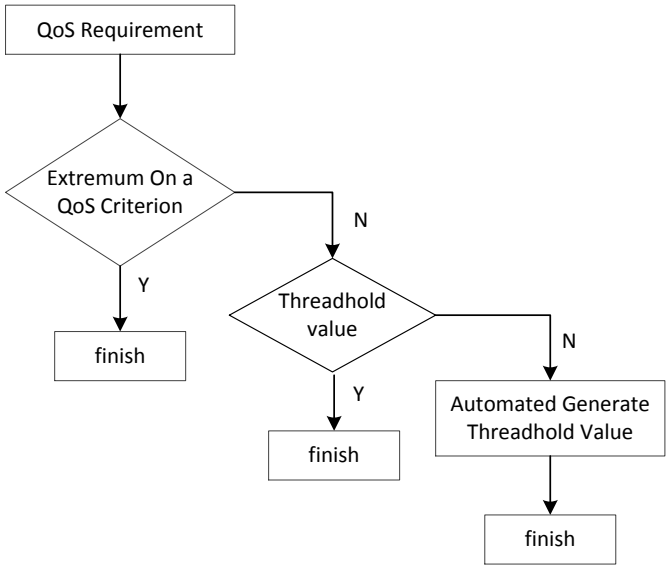


Fig. 6. Standardizing QoS Requirement.

Time complexity: with an index a SELECT is probably $O(log(n))$, so the time complexity is $O(log(n))$.

Case 4: To find paths which contain service B in Figure 3, the SQL statement is:

SELECT pathID FROM usedService WHERE ws_id IN (SELECT ws_id FROM service WHERE ws_name ='B')

3.3.2 Service Addition

The basic process when adding a service is as follows: Create a new path for the newly added services w, connect w with $path$ if there is overlap between w_{in} and $path_{out}$ or between w_{out} and $path_{in}$. In this process, Algorithm 3 - Algorithm 6 are used to compose new paths. After all paths are generated, insert them into "path" and "usedService" table, and w is inserted into "service" table.

Time complexity: Suppose the number of records in table path is n and m is the number of rows in table "newService". We scan "newService" and look up values in "path", the time requirement is $O(mlogn)$. To continue, we need to fetch the data for the table used the index. Fetching data is $O(n)$, so far the time estimation is $O(mlogn + n)$. We may have 0 - $n \times m$ matches to join, so the aggregation estimation is $O(m \times nlog(m \times n))$ as there is a sort. Thus, the total time complexity is $O(m \times nlog(m \times n))$.

3.3.3 Service Update

The idea of service updating is: first remove records contain this service (treat it as a disappeared service). Then add this service into database as a new service and recompute paths contain this service (treat it as a newly added service). Time complexity of service update is the same as service addition.

4 Case Study

In this section, we give a simple but meaningful example to explain how our algorithms work. In this example, the ontology hierarchy contains ten concepts and the service repository contains seven services. Service information is shown in Table 5, which contains input, output concepts, response time, throughput and cost.

TABLE 6
PATH table

pathID	inConcepts	outConcepts
1	A, B, C	J
2	B, C	E, F
3	C, E	H
4	C, F	G
5	K	H
6	H	D
7	G	H
8	B, C	E, F, H
9	B, C	E, F, G
10	C, E	D, H
11	C, F	G, H
12	K	D, H
13	G	D, H
14	B, C	D, E, F, H
15	B, C	E, F, G, H
16	C, F	D, H
17	B, C	D, E, F, H

We generate paths according to Algorithm 1 to Algorithm 3. We firstly generate paths with only one service. The input and output concepts of the service are inputs and outputs of newly generated path. Then, we generate paths with multiple services by connecting paths with one service to those with multiple services. This process ends when no more paths can be generated. The generated paths are shown in Table 6. The sequences of services in paths are decided via Algorithm 4 and Algorithm 5 and stored in "usedService" table (Table 7).

QoS values of paths are calculated in Algorithm 6 and stored in the "path" table (Table 8). The single QoS value is calculated via Equation 4 to Equation 9. The utility Qos value is obtained via Equation 10.

For example, assume the user wants to find service composition with input "B, C" and output "H". We discuss three possibilities of QoS constraints.

Case 1: The user wants to find a path with the minimum response time. This process contains two phases, first, find the path meets user's functional constrains with minimum

TABLE 7
UsedService table

pathID	layer	ws_id
1	1	1
2	1	2
3	1	3
4	1	4
5	1	5
6	1	6
7	1	7
8	1	2
8	2	3
9	1	2
9	2	4
10	1	3
10	2	6
11	1	4
11	2	7
12	1	5
...	...	...

TABLE 5
A set of available services

Ws_id	Service	Input	Output	Response	Thp	Cost
1	w_1	A, B, C	J	25	6000	420
2	w_2	B, C	E, F	30	4000	360
3	w_3	C, E	H	28	3000	330
4	w_4	C, F	G	35	5000	400
5	w_5	K	H	20	2500	290
6	w_7	H	D	15	4000	480
7	w_8	G	H	35	2000	280

Response: response time (ms) as a QoS metric
Throughput: (invocations per minute) as a QoS metric
Cost: (cents) as a QoS metric

TABLE 8
QoS table

pathID	Response	Throughput	Cost
1	25	6000	420
2	30	4000	360
3	28	3000	330
4	35	5000	400
5	20	2500	290
6	15	4000	480
7	35	2000	260
8	58	3000	690
9	65	4000	760
10	43	3000	810
11	70	2000	660
12	35	2500	770
13	50	2000	740
14	73	3000	1170
15	100	2000	1020
16	85	2000	1140
17	115	2000	1500

response time. Then, search services in this path. The search process is illustrated as follows:

SELECT $pathID$, MIN($response$) FROM path WHERE c_in IN
(SELECT c_id FROM concept WHERE c_name in ('B','C')) AND c_out IN
(SELECT c_id FROM concept WHERE c_name='H');

We find path 8 meets the requirement. Then, we search for services in the path:
SELECT ws_name FROM service WHERE ws_id IN (
SELECT ws_id FROM UsedService WHERE pathID=8);

The services in the path are $\{w_2, w_3\}$.

Case 2: User sets his QoS constraints as "response time $<$ 110 and throughput $\geq$ 2000". In this case, the search process is illustrated as follows:
SELECT $pathID$ FROM path
WHERE $response < 110$ AND $throughput >= 2000$ AND c_in IN
(SELECT c_id FROM concept WHERE c_name in ('B','C')) AND c_out IN
(SELECT c_id FROM concept WHERE c_name='H');

We find three paths {8,14,15} meet user's requirements.

Case 3: The user wants to find top-2 paths with the cheapest prices. We filter and rank the returned paths, and the search process is illustrated as follows:
SELECT $pathID$ FROM path WHERE c_in IN
(SELECT c_id FROM concept WHERE c_name in ('B','C')) AND c_out IN
(SELECT c_id FROM concept WHERE c_name='H')
order by $cost$ asc limit 2;

The returned top-2 paths are {8,14}.

Case 4: To calculate the utility value of service w_1:
We use Equation 1 to scale the utility value of throughput (represented as U_{thp}). Response time (U_{resp}) and cost (U_{cost}) are negative criteria, so we use Equation 2 to scale their utility value. thus, we have: $U_{resp}(w_1) = 0.5, U_{thp}(w_1) = 1, U_{cost}(w_1) = 0.3$.

Suppose each QoS criterion has a same weight, with Equation (3), the overall utility score of service w_1 is $U(w_1)$=0.6.

Case 5: To get the QoS score of a path, combine Equation (3) to Equation (9), we have:

$$U(path) = \sum_{i=1}^{n} U_i(path) \times W_i = \frac{T(path) - T^{min}}{T^{max} - T^{min}} \times W_{thp} + \frac{l \times R^{max} - R(path)}{l \times (R^{max} - R^{min})} \times W_{resp} + \frac{m \times C^{max} - C(path)}{m \times (C^{max} - C^{min})} \times W_{cost} \quad (11)$$

Here, we have n QoS criteria, m services and l layers.

According to Equation 11, U(path8)=0.41.

5 Experimental results

We run our experiments on a computer with the following configuration. 1) CPU: Intel Core i5-2400 at 3.10 GHz, 2) Mainboard: Intel C206, 3) Memory: 8GB DDR3 SDRAM PC3-10600, 4) Harddisk: WD2500AAKX 250GB 7200 RPM 16MB cache SATA, and 5) Operating system: Windows 7 professional 64-bit. We use MySQL 5.6 as the database. We run each experiment ten times to get the average execution time.

5.1 Data set

We use TestsetGenerator2009 [24] to generate five data sets and evaluate our work. Each data set contains a WSDL file which is the repository of web services. An OWL file lists

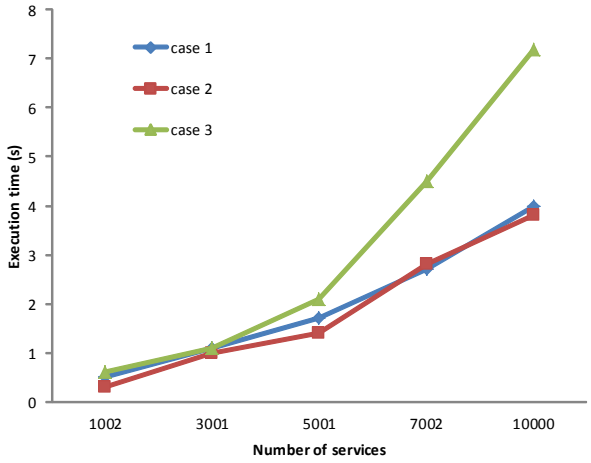


Fig. 7. Time for Searching Solutions with Optimal Response Time.

the relationship between "concepts" and "things". WSLA file describes QoS values of services. The number of services varies from 1000 to 10000, and the number of concepts varies from 3000 to 25000 accordingly. Each web service has around 10 input and 20 output concepts.

5.2 Performance analysis

We generate random queries over the dataset as the user requests and search for solutions with optimal QoS value. In real system, very similar services may be produced, which can be redundant [18]. If not handled properly, redundant services waste time and resources. In the path generating stage, it is unavoidable that paths are generated with redundant services, however, in the path query stage, we use QoS constraints to filter solutions. Paths with redundant services may lead to a longer response time or smaller throughput, so it is quite possible that these paths are eliminated in this stage.

5.2.1 Time for service composition search

Figure 7 to Figure 9 show the execution time of the first stage in path query, which find paths meet user's functional and non-functional requirements. Case 1 – case 3 represent queries shown in Section 4. The execution time of retrieving suitable paths increases as the number of web services increases. After this stage, we fetch top-K solutions with QoS constraints. The fetch time is approximately 15 millisecond.

5.2.2 Time for service disappearance

In this experiment, we randomly delete a service, then find and delete records related to this service in the database. This process is detailed described in Section 3.3.1. We build index on services' name, and compare the performance with and without index in terms of execution time. Figure 10 shows the results. As expected, it takes more time to find and delete records when there is no index on services' name.

5.2.3 Time for service addition

In this experiment, we add a new service in the database, then obtain the time of generating new paths. The process

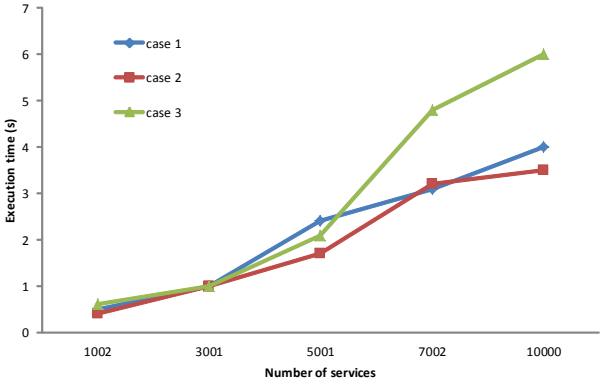


Fig. 8. Time for Searching Solutions with Optimal Throughput.

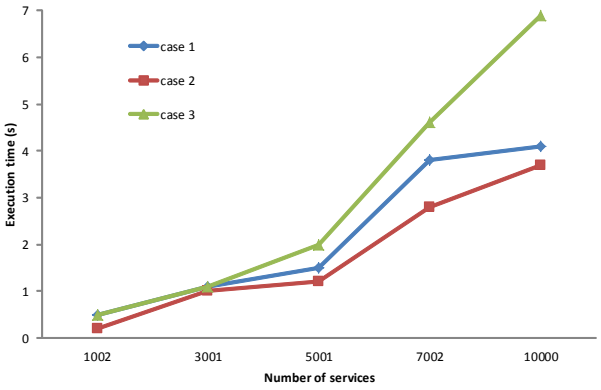


Fig. 9. Time for Searching Solutions with Optimal Utility Score.

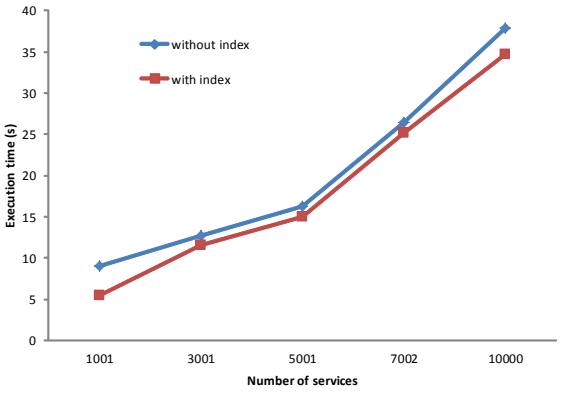


Fig. 10. Time for Service Disappearance.

of service addition is detailed described in Section 3.3.2. From figure 11, we can see as the number of original services in the database increases, the execution time increased dramatically. This is because as the number of web services increases, the number of newly generated paths increases linearly.

6 Related Work

A great deal of work has been done in the theory and practice of web services. In this section, we first discuss web service discovery approach. Then, we review related work on web service composition.

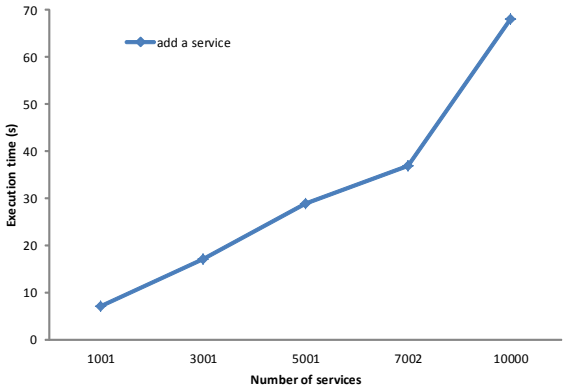


Fig. 11. Time for Service Addition.

6.1 Web service discovery

To utilize web services on the web, developers must first locate them, finding a service on the web refers to web service discovery (WSD) problem. Service providers register service information on the web, thus, services can be searched by syntactic matching, that is, specifying business name, service name, category and tModels (Technical Models, they support specification of additional attributes). UDDI (Universal Description, Discovery and Integration) [25] is an industry standard developed to solve service discovery problem. However, this kind of matching is limited to keyword matching and the way it can be queried. The results produced by syntactic matching are coarse and lack of accuracy. To overcome the above mentioned limitations, semantic web service matchmaking is proposed to further enhance efficiency of service discovery. This is achieved by OWL-S [26] description language. Approaches using OWL-S to match web services can be found in [27], [28], [29]. The authors in [27] store OWL-S descriptions in an UDDI registry to combine the OWL-S file and UDDI data model. Seba *et al.* measure services similarities as a graph for better accuracy [29]. Taking consideration of the high computational complexity, they decompose the calculation process in the graph into smaller subgraphs. Service similarities are represented as similarities of decomposed subgraphs. As a result, the time complexity of service matching is reduced.

Different service providers have different criteria on service properties, a service property used by one service may not appear in another service though both of them may belong to the same service category. As a result, two services may fail to find each other though they are relevant to the query, and this may stand in the way of communication between services. To solve this problem, researchers have demonstrated that service matchmaking should be able to deal with uncertainty in service properties. Li *et al*. propose a ROSSE system on the Rough sets theory to deal with uncertainty of service properties [30], [31]. ROSSE improves the precision of service matching.

6.2 Web service composition

6.2.1 In-memory composition methods

In-memory composition methods load service information into RAM and construct the search graph in RAM. In the area of QoS-aware web service composition, Planning graph model and beam-stack search are popular methods applied to solve this problem [32], [33], [34]. Wanger *et al.* search functionally similar services and store them in clusters, to compose services, they only consider root nodes of clusters [32]. Services in the same cluster are seen as backup services for registered services. However, it is not an easy work to allocate clusters, besides, services have different QoS criteria, it is hard to choose a root service of a cluster. More recently, Kil *et al.* [35] present a novel anytime algorithm approach to solve the QoS-aware service composition problem. They use dynamic beam which dynamically adjust the beam widths to solve the composition problem. Similarly, Yan and Chen present an anytime algorithm which may return better solution if the algorithm keeps running [33]. An extended Dijkstra's algorithm is applied to handle multiple inputs and outputs and search the solution on a Plan Graph. They also discuss how to remove redundant services in a plan to reduce execution cost. They assume that if unsatisfied goals or worse QoS value is caused by removing a service in a solution, this solution does not contain redundant services. Based on this assumption, they propose an algorithm to remove redundant services after they find the solution for the problem. More research on QoS-aware service composition can be found in surveys [36], [37].

6.2.2 Database-based composition methods

The aforementioned research focus on in-memory methods to solve the problem, these algorithms can only work when data fits in RAM. However, loading lots of services information into RAM is expensive, and the search space is limited by the available physical memory. These shortcomings have motivated researchers to utilize relational database to solve service composition problem [17], [19], [38].

The authors in [19] use relational database as a repository to store services in UDDI. They use "join" operator to find service composition by matching services. Zeng *et al.* in [38] present a web service matching algorithm (SMA), this algorithm considers semantic similarity based on WordNet. Moreover, they put forward a Fast-EP service composition algorithm which can be applied in relational database. Though they consider multiple inputs and outputs in their algorithm, they raise an assumption that, two services are connected when one service provides all inputs of the other service. On the contrary, in our approach, two services can be connected if a service provides part of the inputs of another service.

Another work [17] deals with the problem we address here. The authors put forward a PSR system, in which service compositions are computed in advance and stored in tables of relational database. Searches are done by specifying SQL statements. In the PSR system, services are abstracted as single operator and paths have single input (output). To handle user query with multiple goals, the PSR system searches all paths whose outputs are part of user query and combines them together. Our work is distinct from [17] by generating paths as multiple inputs and outputs. We consider services with multiple inputs and outputs, while matching, we connect two services if the outputs of a service can provide part of inputs of another service. Also, we consider QoS constraints which are missing in their system.

We demonstrate that our method can find solutions with fewer redundant services.

7 Conclusions and Future Work

The objective of this paper is to solve the QoS-aware service composition problem with a relational database, to this end, we propose a FSIDB system to retrieve the top-K solutions and return them to the user. First, we generate all possible service combinations as paths and store them in a relational database. Then, we compose SQL statements to query the database for solutions which meet user's functional requirements as well as non-functional requirements. Finally, we find the top-K solutions as backup solutions in case of service disappearance or failure. Experimental results show that our system can always find valid solutions to solve the problem and satisfy user requirements.

As future work, we plan to improve the scalability. Firstly, if a query takes too much memory or time, we might return a non-optimal answer instead of an optimal answer. Secondly, instead of composing all service combinations, we might focus on dominant services (i.e., services that are more frequently queried or have better QoS value) and pre-compose paths among these services, these paths are kept in the database for fast delivery. When a user request comes, we first search in the database to see whether we can find a nearly ready-made solution in the database. Only as a last resort do we construct a search graph to solve the problem.

References

[1] M. Papazoglou, *Web Services: Principles and Technology*. Prentice Hall, 2011.

[2] 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*, ser. WWW '03. New York, NY, USA: ACM, 2003, pp. 411–421.

[3] R. Berbner, M. Spahn, N. Repp, O. Heckmann, and R. Steinmetz, "Heuristics for qos-aware web service composition," in *Web Services, 2006. ICWS '06. International Conference on*, Sept 2006, pp. 72–82.

[4] 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.

[5] B. Benatallah, M. Dumas, Q. Sheng, and A. H. H. Ngu, "Declarative composition and peer-to-peer provisioning of dynamic web services," in *Data Engineering, 2002. Proceedings. 18th International Conference on*, 2002, pp. 297–308.

[6] 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.

[7] (2015, September) Maxima and minima. [Online]. Available: https://en.wikipedia.org/wiki/Maxima_and_minima/

[8] S.-C. Oh, J.-Y. Lee, S.-H. Cheong, S.-M. Lim, M.-W. Kim, S.-S. Lee, J.-B. Park, S.-D. Noh, and M. Sohn, "Wspr*: Web-service planner augmented with a* algorithm," in *Commerce and Enterprise Computing, 2009. CEC '09. IEEE Conference on*, July 2009, pp. 515–518.

[9] S.-C. Oh, B.-W. On, E. Larson, and D. Lee, "Bf*: Web services discovery and composition as graph search problem," in *e-Technology, e-Commerce and e-Service, 2005. EEE '05. Proceedings. The 2005 IEEE International Conference on*, March 2005, pp. 784–786.

[10] P. Rodriguez-Mier, M. Mucientes, and M. Lama, "Automatic web service composition with a heuristic-based search algorithm," in *Web Services (ICWS), 2011 IEEE International Conference on*, July 2011, pp. 81–88.

[11] H. Kil and W. Nam, "Anytime algorithm for qos web service composition," in *Proceedings of the 20th International Conference Companion on World Wide Web*. ACM, 2011, pp. 71–72.

[12] X. Zheng and Y. Yan, "An efficient syntactic web service composition algorithm based on the planning graph model," in *Web Services, 2008. ICWS '08. IEEE International Conference on*, Sept 2008, pp. 691–699.

[13] S.-Y. Lin, G.-T. Lin, K.-M. Chao, and C.-C. Lo, "A cost-effective planning graph approach for large-scale web service composition," in *Mathematical Problems in Engineering*, vol. 2012, 2012, pp. 1–21.

[14] B. Ding, J. Xu Yu, S. Wang, L. Qin, X. Zhang, and X. Lin, "Finding top-k min-cost connected trees in databases," in *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, April 2007, pp. 836–845.

[15] B. Ding, J. X. Yu, and L. Qin, "Finding time-dependent shortest paths over large graphs," in *Proceedings of the 11th International Conference on Extending Database Technology: Advances in Database Technology*, ser. EDBT '08. New York, NY, USA: ACM, 2008, pp. 205–216.

[16] Y. Yang, H. Gao, J. X. Yu, and J. Li, "Finding the cost-optimal path with time constraint over time-dependent graphs," *Proc. VLDB Endow.*, vol. 7, no. 9, pp. 673–684, may 2014.

[17] 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.

[18] 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.

[19] H. Lakshmi and H. Mohanty, "Rdbms for service repository and composition," in *Advanced Computing (ICoAC), 2012 Fourth International Conference on*, Dec 2012, pp. 1–8.

[20] J. Li, Y. Yan, and D. Lemire, "Full solution indexing using database for qos-aware web service composition," in *Services Computing (SCC), 2014 IEEE 11th International Conference on*, June 2014, pp. 99–106.

[21] L. Zeng, B. Benatallah, A. Ngu, M. Dumas, J. Kalagnanam, and H. Chang, "Qos-aware middleware for web services composition," *Software Engineering, IEEE Transactions on*, vol. 30, no. 5, pp. 311–327, May 2004.

[22] E. Triantaphyllou., *Multi-Criteria Decision Making: A Comparative Study*. Springer Science & Business Media, 2013.

[23] W3C OWL Working Group. (2012) OWL 2 Web Ontology Language Document Overview (Second Edition). [Online]. Available: http://www.w3.org/TR/owl2-overview/

[24] WS-Challenge. (2010) Testsetgenerator2009. [Online]. Available: https://code.google.com/p/wsc-pku-tcs/downloads/list

[25] Oasis.(2007) uddi version 2.04 api specification. [Online]. Available: http://uddi.org/pubs/ProgrammersAPI-V2.04-Published-20020719.htm

[26] Web ontology language for web services. [Online]. Available: http://www.w3.org/submission/owl-s/

[27] N. Srinivasan, M. Paolucci, and K. Sycara, "An efficient algorithm for owl-s based semantic search in uddi," in *Semantic Web Services and Web Process Composition*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2005, vol. 3387, pp. 96–110.

[28] S. Lagraa, H. Seba, and H. Kheddouci, "Matchmaking owl-s processes: An approach based on path signatures," in *Proceedings of the International Conference on Management of Emergent Digital EcoSystems*, ser. MEDES '11. ACM, 2011, pp. 169–176.

[29] H. Seba, S. Lagraa, and H. Kheddouci, "Web service matchmaking by subgraph matching," in *Web Information Systems and Technologies*, ser. Lecture Notes in Business Information Processing, J. Filipe and J. Cordeiro, Eds. Springer Berlin Heidelberg, 2012, vol. 101, pp. 43–56.

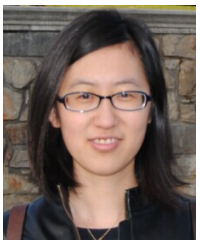
[30] M. Li, B. Yu, V. Sahota, and M. Qi, "Web services discovery with rough sets," *International Journal of Web Services Research*, vol. 6, no. 1, pp. 69–86, 2009.

[31] M. Li, B. Yu, O. Rana, and Z. Wang, "Grid service discovery with rough sets," *IEEE Transactions on Knowledge and Data Engineering*, vol. 20, no. 6, pp. 851–862, 2008.

[32] F. Wagner, F. Ishikawa, and S. Honiden, "Qos-aware automatic service composition by applying functional clustering," in *Web Services (ICWS), 2011 IEEE International Conference on*, July 2011, pp. 89–96.

[33] Y. Yan, M. Chen, and Y. Yang, "Anytime QoS optimization over the PlanGraph for web service composition," in *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, ser. SAC '12. ACM, 2012, pp. 1968–1975.

[34] 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*, ser. WWW '09. New York, NY, USA: ACM, 2009, pp. 881–890.



Jing Li is a PhD student in Computer Science at Concordia University, Montreal. She received her Masters degree in Computer Science from Hunan University, China in 2011. Her main research interests are automated web service composition and database-based service composition.

[35] H. Kil and W. Nam, "Efficient anytime algorithm for large-scale qos-aware web service composition," *International Journal of Web and Grid Services*, vol. 9, no. 1, pp. 82–10, 2013.

Dr. Yuhong Yan is an associate professor at the Department of Computer Science and Software Engineering at Concordia University, Montreal since June 2008. Before joining Concordia, she was a Researcher Officer in the Institute for Information Technology (IIT) in the Canadian National Research Councils (NRC) since Feb. 2003. Her current research focuses on Service Computing, Mobile Computing and Cloud Computing. She is exploring the domain of formal modeling, composition, monitoring, fault diagnosis, reparation, and adaptation of Web service processes. She has authored over 40 articles and papers. She is one of the organizers of IEEE ICWS and SCC in recent years.

[36] Q. Z. Sheng, X. Qiao, A. V. Vasilakos, C. Szabo, S. Bourne, and X. Xu, "Web services composition: A decades overview," *Information Sciences*, vol. 280, no. 0, pp. 218 – 238, 2014.

Dr. Daniel Lemire is a computer science professor at LICEF Research Center, TELUQ, Universit du Qubec. He has also been a research officer at the National Research Council of Canada and an entrepreneur. He has written over 45 peer-reviewed publications, including more than 25 journal articles. He has held competitive research grants for the last 15 years. He has served as program committee member on leading computer science conferences (e.g., ACM CIKM, ACM WSDM, ACM RecSys). His open source software has been used by major corporations such as Google and Facebook. His research interests include databases, information retrieval and high performance programming.

[37] Y. Syu, S.-P. Ma, J.-Y. Kuo, and Y.-Y. FanJiang, "A survey on automated service composition methods and related techniques," in *Services Computing (SCC), 2012 IEEE Ninth International Conference on*, June 2012, pp. 290–297.

[38] C. Zeng, W. Ou, Y. Zheng, and D. Han, "Efficient web service composition and intelligent search based on relational database," in *Information Science and Applications (ICISA), 2010 International Conference on*, April 2010, pp. 1–8.