FULL SOLUTION INDEXING AND EFFICIENT COMPRESSED GRAPH REPRESENTATION FOR WEB SERVICE COMPOSITION

JING LI

A THESIS

IN

THE DEPARTMENT

OF

COMPUTER SCIENCE AND SOFTWARE ENGINEERING

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY (COMPUTER SCIENCE)

CONCORDIA UNIVERSITY

MONTRÉAL, QUÉBEC, CANADA

OCTOBER 2016

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Jing Li**

Entitled: **Full Solution Indexing and Efficient Compressed Graph Representation for Web Service Composition**

and submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Computer Science)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the Final Examining Committee:

_____ Chair
Dr. Anjali Awasthi

_____ External Examiner
Dr. Chung-Horng Lung

_____ Examiner
Dr. Constantinos Constantinides

_____ Examiner
Dr. Todd Eavis

_____ Examiner
Dr. Wahab Hamou-Lhadj

_____ Supervisor
Dr. Yuhong Yan

_____ Co-supervisor
Dr. Daniel Lemire

Approved by _____
Dr. Sudhir Mudur, Chair
Department of Computer Science and Software Engineering

_____ 2016

_____
Amir Asif, Dean
Faculty of Engineering and Computer Science

Abstract

Full Solution Indexing and Efficient Compressed Graph Representation for Web Service Composition

Jing Li, Ph.D.

Concordia University, 2016

Service-oriented computing enhances business scalability and flexibility; providers who expect to benefit from it may bring explosive growth of web services. Searching an optimal composition solution with both functional and non-functional requirements is a computationally demanding problem: the time and space requirements may be infeasible due to the high number of available services. In this thesis, we study QoS-aware service composition problems which satisfy functional requirements as well as non-functional requirements. We use optimization algorithms to enhance accuracy of our searching algorithms.

In the first approach, we propose a database-based approach to search a service composition solution. Current in-memory methods are limited by expensive and volatile physical memory, to deal with this problem, we want to use the large space available in relational database on persistence disk. In our database-based approach, all possible service combinations are generated beforehand and stored in a relational database. When a user request comes, SQL queries are composed to search in the database and K best solutions are returned. We test the performance of the proposed approach with a service challenge data set; experiment results demonstrate that this approach can always successfully find top-K valid solutions.We offer three main contributions in this approach. First, we overcome the disadvantages of in-memory composition algorithms, such as volatile and expensive, and provide a solution suitable to cloud environments. Second, we fetch top-K solutions in case the optimal solution is not available as backup solutions to the user. Third, compared with other pre-computing composition methods, we use a single SQL query: there is no need to eliminate spurious services iteratively.

Then, we propose the application of a skyline operator to reduce the search space and improve the scalability. Skyline analysis returns all of the elements that are not dominated by another element. We use skyline analysis to find a set of candidate services referred to as "skyline services", therefore, less competitive services are reduced. This allows us to find a solution for a large composition problem with less storage and increased speed.

In reality, different users may have same requests, we are motivated to pick some popular requests and generate paths for fast delivery. These paths are stored in a separate table of the relational database. When a user request comes, we first search to find a nearly ready-made solution. Only as a last resort do we search the table with whole paths to find a solution.

Finally, to deal with the problem that the search space may explore, we apply a compressed data structure to represent the service composition graph. The goal is to allow algorithms running in in-memory over larger graphs. In this approach, we use compact K^2-trees to represent the service composition graph. When a user request comes, we search the K^2-tree for a satisfactory solution. We use an array to store values in the last level of the compact tree, which represents relationships between services and concepts. In our algorithms, we find services' inputs (resp. outputs) by locating elements in this array directly, therefore, decompressing the graph is unnecessary. To the best of our knowledge, our work is the first attempt to consider compact structure in solving web service composition problems. Experiment results demonstrate that this approach takes less space and has good scalability when handling a large number of web services.

We provide different approaches to search a solution for the user. If the user want to find an optimal solution with fewer services, he may use the database-based approach to search for a solution. If the user want to get a solution in a short time, he may choose the in-memory approach.

Acknowledgments

I would like to express my sincere gratitude to my supervisors Dr. Yuhong Yan and Dr. Daniel Lemire for their continuous support and guidance over my research work. I am very thankful to Dr. Yuhong Yan for leading me to the domain of automated reasoning. Dr. Daniel Lemire is a wonderful mentor who guides my research with a lot of patience.

I am also thankful to all the members of my examining committee including Dr. Constantinos Constantinides, Dr. Todd Eavis and Dr. Wahab Hamou-Lhadj for their precious time to review my research proposal, doctoral seminar, thesis and give me valuable feedback on my work.

Special thanks to my friends during my Ph.D. program for wonderful moments we shared and their encouragements especially during tough times. I greatly thank my colleague Min Chen, for the fruitful discussions.

I would like to take this opportunity to extend my gratitude to my parents for their unconditional support with their sacrifice. I express my sincere gratitude to my husband for his understanding and unwavering support. They are definitely the main source of my inspiration and motivation. I do not have enough words to express my eternal gratitude to them.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

To survive in the highly competitive corporate environment, enterprises may focus on their core services and find other services from other providers to fulfill a business goal via Internet. Service-oriented architecture (SOA) is an architectural pattern in which application components provide on-demand software systems to other components [1]. A fundamental objective of SOA is to compose existing web services to meet complex user requirements. Contrary to traditional methods which implement new systems from scratch, SOA helps lower costs of software development and management. As shown in Figure 1.1, SOA follows the find-bind-execute paradigm:

- Service provider owns web services and implements functionalities of web services. Service provider publishes services and describes interfaces of services as well as their operations in WSDL (Web Service Description Language) [5] documents. The service provider is also responsible for registering services information with service register.
- Service requestor is the client who searches service register for desired services, downloads WSDL through HTTP and binds services with SOAP (Simple Object Access Protocol) [6].
- Service register publishes services in a service repository.

Web services are loosely coupled software modules that are published, located and invoked on the web. The web services environment is basically defined by the following:

- WSDL is an XML-based specification schema, it describes interfaces of services and is the

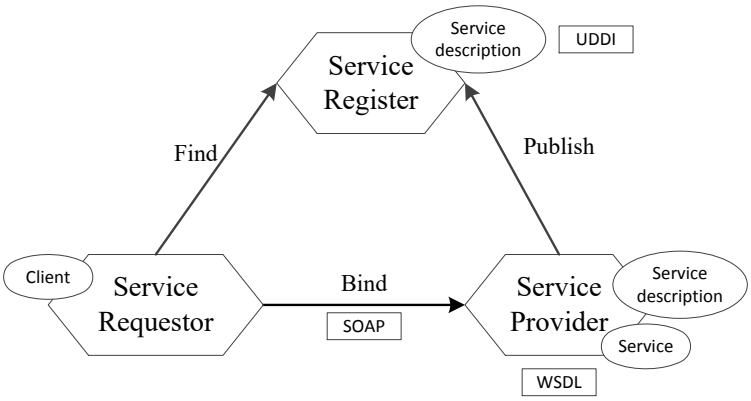
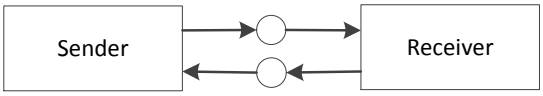


Figure 1.1: An overview of service-oriented architecture [1].

(a) One-way messaging



(b) Request/response messaging

Figure 1.2: SOAP messaging.

means to access services. WSDL specifies syntactic signature of services but does not specify non-functional properties.

- SOAP is a lightweight communication protocol which web services use to exchange messages. SOAP defines an extensible XML messaging framework which can send and receive HTTP packets and process XML messages. As illustrated in figure 1.2, web services can use *one-way messaging* to send messages from a sender to a receiver or *request/response messaging* to exchange messages.

- UDDI (Universal Description Discovery and Integration) is a registry standard for service description and discovery that supports publishing and discovery processes [7]. The core concept of UDDI is an XML document that describes a business entity and its web services.

There are two main problems in web service research area: web service selection and composition.

Web Service Selection

Finding a service on the web is referred to as web service selection problem. Quality of Service (QoS) refers to the non-functional properties of services such as availability, response time, throughput, price, execution duration, reputation and successful execution rate. Data mining [8, 9] and skyline techniques [10, 11] have been applied to prune the searching space of service selection. In considering that services are often created with similar purposes, Wagner *et al.* find clusters with functionally similar services [8]. Aznag *et al.* use a hierarchical cluster approach to extract clusters from service descriptions [9]. Alrifai *et al.* present a pair-wise comparison method to determine skyline services [10]. Yu and Bouguettaya enumerate service plans in an ascending order of QoS scores [11]. Cluster analysis divides data into meaningful groups, however, this method is always computationally difficult [12] and time consuming [13].

In addition, semantic matching is introduced to improve the accuracy of service discovery. The difficulty of picking a suitable service among a number of functionally similar web services increases if the non-functional requirements of users are not well explained or understood. Ontology linking and Latent Semantic Indexing are employed in [14] to help extend service selection from syntactical to semantic level. Semantic matching helps providers understand users' requirements better and enhances efficiency of service discovery.

Web Service Composition

Users may fulfill their complex business requirements by combining together different web services. We demonstrate a scenario of a travel plan to show the service composition motivation in Figure 1.3.

(1) A tourist wants to travel from Montreal to Chicago. First, he describes his requirements 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 find a solution by combining different services as a travel plan.

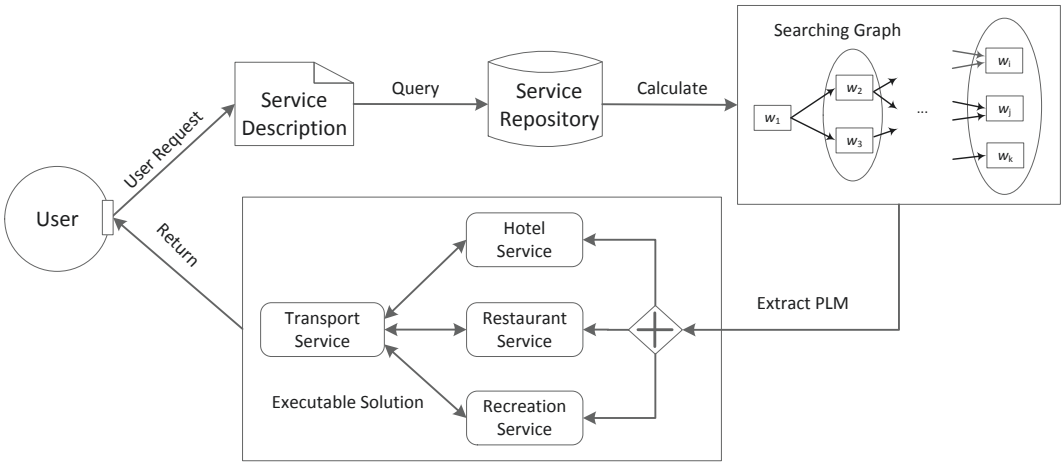


Figure 1.3: A web service composition example.

(2) We choose services in the "Service Repository", and construct a searching graph from initial states to the goal with chosen services.

(3) We extract a solution from the searching graph. Different services may have same functionality, in this situation, we select services based on the user's non-functional requirements.

(4) We combine chosen services as a new service, return this service as an executable solution to the user.

The aim of web service composition is to find a chain of connected services in which output parameters of a service are used as part of input parameters of another service (or a group of services). When the composition problem satisfies both the functional goals and optimization of QoS as well, it is called a QoS-aware service composition problem.

The number of web services on the web is enormous and users' requests can be complex. Therefore, efficient service composition is required. It is expected that algorithms should scale with the number of services and meet users' requirements as fast as possible. Many researchers have put great effort in studying service composition problem. Different heuristic search methods, such as A* method [15, 16], Multi-Optimization [17, 18], planning algorithms [19, 20] as well as Integer Liner Programming [21, 22] are applied to find a solution in this area. Searching an optimal composition solution is a computationally demanding problem: the time and space requirements may be infeasible due to the large number of available services. To alleviate this problem, optimization

algorithms are introduced to reduce the searching space, minimize the search time and improve accuracy of the obtained solution.

Most of current composition algorithms are in-memory algorithms, which can only work when the data fits in RAM. However, loading lots of services information into RAM is expensive (even today), and the searching 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. Based on this consideration, researchers are motivated to utilize relational database to not only store web services but also solve service composition problem [23, 24]. For a database-based composition method, all the service combinations are generated and stored in the database. We want to study, implement, and compare different in-memory and database-based service composition algorithms, the aim of this thesis is to propose novel and effective methods to find solutions for QoS-aware service composition problem.

1.1 Objectives

We are motivated to use a relational database to store possible service combinations on persistence disk instead of small volatile memory. A database makes it possible to find a solution for the problem which has large-scale services with complex operators. Current database-based service composition methods ignore users' preferences and do not support QoS optimization. Besides, experimental results show that the existence of redundant services (a service is redundant if all its outputs used by other services are also produced by other services [25]) is common in both in-memory and database-based service composition methods. In our first approach, we focus on solving service composition problem as well as QoS optimization problem with a database.

To find a solution with a database, we first evaluate the applicability of database-based service composition method and propose a QoS-aware service composition method. Current database-based service composition methods abstract services as single operators and find paths with single input/output. To solve a problem with multiple inputs and outputs, these methods find and return

all paths which meet part of requirements. This is easy to understand and implement, as a side effect, redundant services exist in these methods. We use services with multiple inputs and outputs to generate combinations with multiple inputs and outputs. It is challenging to store and query a combination with multiple inputs and outputs.

We are also motivated to use a compressed data structure to find a solution in RAM. Compressed graph representation allows graph algorithms running in RAM over large graphs. We propose to apply a compact tree to represent the search graph and save storage space.

QoS optimization is another important object in solving service composition problem. QoS-aware service composition fulfills users' functional goal and maximizes their satisfaction at the same time. In local QoS optimization methods, service candidates are grouped into multiple QoS levels, services in different levels are selected independently, as a result, the complexity of execution time is polynomial. Unfortunately, these methods can easily fall into "local maximum" problem. Global optimization requires calculation of all the possible combinations and has the drawback of exponentially increased computation complexity. This makes it difficult to find a global optimization solution.

Most of current service composition methods contain redundant services which waste execution time and resources. In this research, we study redundant services problem and try to reduce redundant services in the solution. Finally, we compare the efficiency of our system to other database-based service composition methods as well as in-memory service composition algorithms. The proposed method should overcome disadvantages of in-memory composition algorithms, such as volatile and expensive. Besides, this system should minimize the number of redundant services.

1.2 Thesis Contributions

The thesis aims at addressing the service composition problem and calculating globally optimized QoS values. The main contributions of this thesis are summarized as follows:

✓ **Full Solution Indexing for top-K Web Service Composition:**

We propose to use a relational database to find solutions for web service composition problem with a large number of web services and complex operators. The possible connections among

services 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.

✓ **Compressed Graph Representation for Service Composition:**

We propose a novel service composition method in which a compact tree is applied to represent the search graph, this tree representation takes advantage of the scarcity of the search graph, and thus saves storage space. The tree representation allows fast navigation without decompression.

1.3 Thesis Organization

This thesis is organized as follows:

- **Chapter 1:** overviews web service selection and composition problems. Afterwards, the main contributions of this thesis are described.

- **Chapter 2:** introduces the preliminary knowledge and reviews related work. This chapter introduces the background for rest of this thesis. In particular, we review popular web service composition algorithms and related techniques.

- **Chapter 3:** proposes a relational database approach for web service composition problem. This system composes SQL queries to search in the database and returns K best solutions. Experiment results demonstrate that this approach can always find top-K valid solutions.

- **Chapter 4:** presents the application of a skyline operator in the database approach to reduce the searching space and improve the scalability. We also present a partial pre-composing approach which stores popular paths for fast delivery. Only as a last resort do we search the table with whole paths to find a solution.

- **Chapter 5:** represents web service composition problem with a compact graph representation. Two methods are respectively proposed to find solutions for the problem with the

compact K^2-tree representation. There is no need to decompress the compact graph while searching for a composition solution. Experimental results show this method can find a solution with limited storage requirement.

- **Chapter 6:** summarizes the thesis by highlighting main contributions. We also outline the direction of our future work in this chapter.

1.4 Publications

(1) J. Li, Y. Yan, and D. Lemire, Full Solution Indexing for top-K Web Service Composition, accepted for publication in *IEEE Transactions on Services Computing* (based on work of Chapter 3).

(2) J. Li, Y. Yan, and D. Lemire, Scaling up web service composition with the skyline operator, in *International Conference on Web Services (ICWS), 2016 IEEE International Conference on* (based on work of Chapter 4).

(3) J. Li, Y. Yan, and D. Lemire, A web service composition method based on compact k2-trees, in *Services Computing (SCC), 2015 IEEE International Conference on* (based on work of Chapter 5).

(4) J. Li, Y. Yan, and D. Lemire. Full solution indexing using database for QoS-aware web service composition, in *Services Computing (SCC), 2014 IEEE International Conference on* (based on work of Chapter 3).

Chapter 2

Background and Related Work

2.1 Preliminary

In this section, we first formalize definitions of web services and web service composition. Then, we show how to calculate single and utility QoS values of web service compositions.

2.1.1 Web Service

Definition 1. *A rooted ontology tree is a tuple 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: $\forall c_1, c_2, c_3 \in C, (c_1 \rightarrow c_2) \wedge (c_2 \rightarrow c_3) \Rightarrow c_1 \rightarrow c_3$.

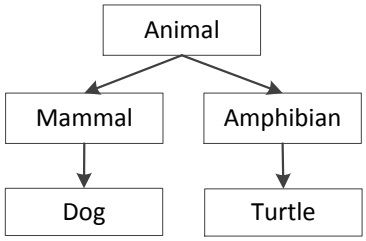


Figure 2.1: A rooted ontology tree example.

A rooted ontology tree is constructed according to similarities and differences among different entities, it is an essence to enable semantic interpretation. For example, in the rooted ontology tree of figure 2.1, we have five concepts: "Animal" "Mammal" "Amphibian" "Dog" and "Turtle". "Animal" is the root of this tree, "Dog" is a child of "Mammal".

Web services are loosely coupled reusable software modules that can be published, located and invoked on the web. Web services complete tasks, solve problems, and conduct transactions on behalf of a user [1].

Definition 2. *A web service w is defined as a tuple 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* 4 *and Definition* 5*.*
- Q *is a finite set of QoS criteria for* w*.*

Services are connected either in sequence or in flow control. Services in sequence are invoked one by one ($w_1; w_2; \ldots; w_n$). Services in a flow control are invoked in parallel ($w_1||w_2|| \ldots ||w_n$).

The used quality criteria, which determine the usability and utility of a service, are listed as below [1, 26]:

1) *Response time (R)*: the interval between the receipt of an inquiry message and the beginning of the transmission of a response message (unit:milliseconds).

$$R(w_1; w_2; \ldots; w_n) = \sum R(w_i) \tag{1}$$

$$R(w_1||w_2|| \ldots ||w_n) = \max R(w_i) \tag{2}$$

2) *Throughput (T)*: the average rate of successful message delivered per time over a communication channel (unit: requests/min).

$$T(w_1; w_2; \ldots; w_n) = \min T(w_i) \tag{3}$$

$$T(w_1||w_2|| \ldots ||w_n) = \min T(w_i) \tag{4}$$

3) *Cost (C)*: the amount of money paid to the service provider to use the service (unit: cents).

$$C(w_1; w_2; \ldots; w_n) = \sum C(w_i) \tag{5}$$

$$C(w_1||w_2|| \ldots ||w_n) = \sum C(w_i) \tag{6}$$

4) *Reliability (RB)*: the ability to serve correctly despite system or network failures. It is measured by the number of transactional successes per month (unit: successes/month).

$$RB(w_1; w_2; \ldots; w_n) = \prod RB(w_i) \tag{7}$$

$$RB(w_1||w_2|| \ldots ||w_n) = \prod RB(w_i) \tag{8}$$

5) *Availability (A)*: the probability a service is available, θ is a constant of time period.

$$A(w_1; w_2; \ldots; w_n) = \prod A(w_i) \tag{9}$$

$$A(w_1||w_2|| \ldots ||w_n) = \prod A(w_i) \tag{10}$$

2.1.2 Web Service Composition

Sometimes, an individual web service may fail to accomplish a business task, in such a situation, a number of existing services are composed to complete the task. Service composition is the generation of a business process which fulfills business tasks that cannot be finished by individual services.

Definition 3. *A web service composition problem can be represented by a tuple 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 4. *Two services are exact match if both their input parameters and output parameters are the same:* $w \equiv w' \leftrightarrow \forall i \in w_{\text{in}}, i' \in w'_{\text{in}}, i = i' \wedge o \in w_{\text{out}}, o' \in w'_{\text{out}}, o = o'$.

Definition 5. *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 thesis, we use plug-in matching degree to match services and achieve semantic service composition. Semantic matching overcomes limitations of syntactic matching and enhances efficiency of service discovery. An OWL file lists "concepts" (classes), "things" (instances), similarities and differences among them. In a rooted ontology tree, concepts and things are represented as nodes, their relationships are represented as edges. The rooted ontology tree is the foundation to identify semantic matching.

We first build an indexing table for input and output concepts of each service. Then, we extend this indexing table according to the ontology tree. For each web service in the service repository: for each of its output parameters, we index its directed concept (parent in the rooted ontology tree) as well as its parent concepts as effects of this service. For each of input parameters, we index its directed concept as the precondition of this service. To check whether two services can be connected or not, we search the indexing table for matching services, if the output of a service subsumes the input of another service, these two services can be connected. For example, in Table 2.1, service w_1 has an output parameter which is an instance of "Dog", according to the rooted ontology tree illustrated in Figure 2.1, the output of w_1 is extended to {"Dog", "Mammal", "Animal"}. Service w_2 has an input parameter which is an instance of "Animal", as the outputs of w_1 subsumes the input of w_2, w_1 is an input service of w_2.

Table 2.1: An example of indexing table

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

2.1.3 Utility Function

QoS criteria determine the usability and utility of a service. Some of these criteria are positive; the higher the value, the higher the quality. throughput and reliability both belong to this category. On the other hand, response time and cost are negative criteria, the higher the value, the lower the quality. We want to find a uniform way to compare these qualities, especially with multiple criteria. Besides single QoS criterion, we use a utility function to evaluate the overall, multiple QoS criteria of a given service. We adopt Multiple Criteria Decision Making (MCDM) [27] technique to aggregate QoS to obtain this utility value:

$$U_i(w_j) = \begin{cases} \dfrac{Q_i(w_j) - Q_i^{min}}{Q_i^{max} - Q_i^{min}} & \text{if} \quad Q_i^{max} - Q_i^{min} \neq 0 \\ 1 & \text{if} \quad Q_i^{max} - Q_i^{min} = 0 \end{cases} \tag{11}$$

$$U_i(w_j) = \begin{cases} \dfrac{Q_i^{max} - Q_i(w_j)}{Q_i^{max} - Q_i^{min}} & \text{if} \quad Q_i^{max} - Q_i^{min} \neq 0 \\ 1 & \text{if} \quad Q_i^{max} - Q_i^{min} = 0 \end{cases} \tag{12}$$

For a positive criteria (*i.e.,* throughput and reliability), this value is scaled with Equation (11), for a negative criteria (*i.e.,* response time and cost), this value is scaled with Equation (12). This aggregate function is consistent with other papers like [20] and [21]. The utility quality score of a service w_j is calculated in Equation (13), the higher the utility value, the higher the quality value.

$$U(w_j) = \sum_{i=1}^{n} U_i(w_j) \times W_i \tag{13}$$

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

In a QoS-aware service composition problem, we compare solutions with either single QoS criterion or multiple QoS criteria. For single criterion problem, the QoS value of a composition solution can be calculated with Equation (1) to Equation (10). Combine Equation (1) to Equation (13),

we get the overall QoS score of a service composition:

$$U(S) = \sum_{i=1}^{n} U_i(S) \times W_i \tag{14}$$

Here, S represents services composition.

2.2 Literature Review

A great deal of work has been done in the theory and practice of web services. In this section, we first introduce web service selection approach. Then, we review related work on redundant service after introducing the study of service composition. Finally, we discuss how to revise a broken service in the service composition.

2.2.1 Web Service Selection

To utilize web services on the web, developers must first locate them, finding a service on the web refers to web service selection problem. Web service selection can be achieved by two main approaches: web service search engines and Universal Description Discovery and Integration (UDDI). Service providers register service information on the web, therefore, 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 has been used to discover web services [28, 29], unfortunately, it is limited to keyword matching, as a result, the returned results 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 [30, 31]. Semantic service matchmaking uses OWL-S (Web Ontology Language for Web Services) description language [32] or XML Schema to define services and their relationships. OWL file expresses preconditions and effects of web services as "concepts" by using ontologies and defines relations among concepts. Ontology is a hierarchy subdivided according to similarities and differences among different entities, it is an essence to enable semantic interpretation. Seba *et al.* measure services similarities as a graph for

better accuracy [31]. Taking consideration of the high computational complexity, they decompose the calculation process in a 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 advertisements may have different criteria on service properties, a service property used by one service advertisement may not be explicitly used by another advertisement. For example, in Table 2.2, the property P_1 used by service advertisement S_1 does not appear in service advertisement S_2. When S_1 and S_2 are matched with a query using P_1, P_2 and P_3, P_1 is an uncertain property to S_2. Similarly, P_3 is an uncertain property to S_1. As a result, S_1 and S_2 may fail to find each other though they are relevant to the query. The above stated problem may stand in the way of communication between services. To solve this problem, Li *et al.* propose a ROSSE system to deal with uncertainty of service properties [4]. ROSSE employs the Rough sets theory to improve the precision of service matching.

Table 2.2: Service advertisements with uncertain service properties [4]

Service advertisements	property	property	property
S_1	P_1	P_2	
S_2		P_2	P_3

Users may have different preferences on QoS criteria. For example, to choose a flight provided by different airlines. Customer A may pick a flight with the cheapest price, while customer B may pick the one with the earliest departure time. QoS-aware service discovery is proposed to enhance users' satisfaction. Liu *et al.,* [33] propose a model which achieves dynamic and fair computation of QoS values based on users' feedbacks. This approach applies a two stages normalization on a matrix in which each row represents a service and each column represents a QoS criterion. Firstly, the QoS value of each service is divided by the average QoS value, in this process, a maximum value is introduced as an upper limit of the division result. Secondly, the authors consider 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.

Zeina *et al.* classify web services by means of Relational Concept Analysis (RCA) classification method [2]. FCA (Formal Concept Analysis is a principled way of deriving a concept hierarchy or

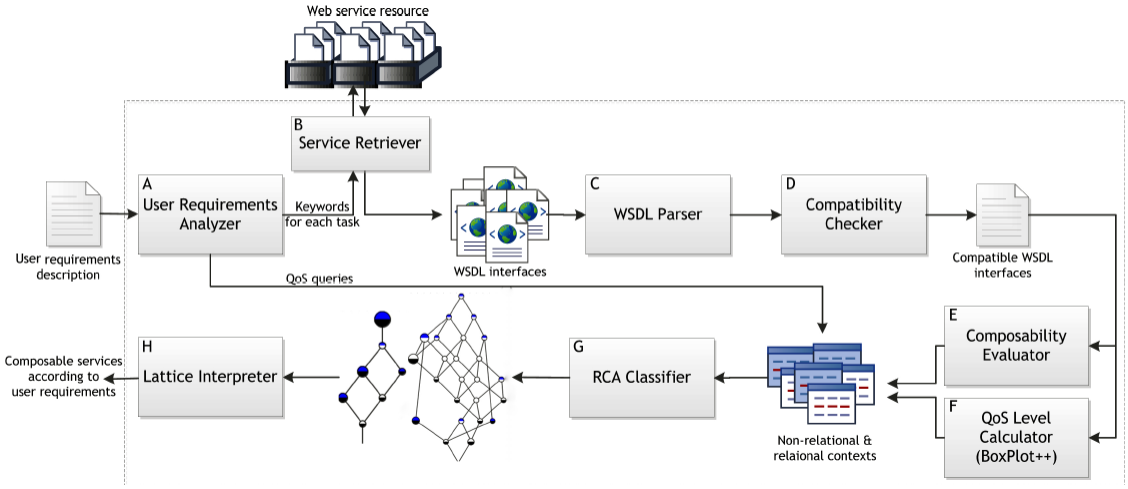


Figure 2.2: An overview of the selection approach [2].

formal ontology from a collection of objects and properties [34]) identifies groups of objects with common attributes by extracting the set of all the formal concepts. FCA reveals the inheritance relations among extracted concepts and organizes them into an ordered structure. RCA extends FCA by taking consideration of relations between objects. Figure 2.2 illustrates an overview of their service selection approach. This approach contains four main steps: service collection, service validation and compatibility filtration, QoS levels calculation, and RCA classification.

The initialization process takes users' requirements and web service resources as system inputs. The Analyzer extracts information from the requirements and WSDL Parser checks services' compatibility. Then, the Compatibility Checker checks whether or not available services provide part of the task, this is done by using the Jaro-Winkler string distance measure. This component also helps prune services which do not contribute to the task. QoS Levels Calculator clusters convergent values by applying BoxPlot++ technique which generates a non-relational context for each service set. Later, the context is exploited by RCA Classifier to classify services according to their QoS levels.

The RCA Classifier plays an important role in the system. The RCA classifier first integrates QoS specifications to non-relational context and adds composition specifications to the relational

context. Then, the RCA classifier generates services lattices and passes them to the final component. Finally, the Lattice Interpreter locates satisfying services at sub-concepts of the queries concepts. Services in the same group have common QoS and composition levels. The composable services are returned as the outputs of the system.

Wu *et al.* propose a hybrid service tag recommendation strategy (WSTRec) to retrieve relevant services [35]. Tagging, which provides more meaningful descriptions, is the act of adding keywords to services. To cluster web services, the WSTRec approach first crawls tags of web services, then, it sends tags to service clustering module where feature-level similarities and tag-level similarities are integrated into global similarities. K-means clustering approach is employed in service clustering.

Tag co-occurrence, tag mining and relevance measurement are employed in tag recommendation process. Specifically, tag co-occurrence suggests candidate tags by extracting rules among existing tags, tag mining digs initial tags from WSDL files. Relevance measurement is adopted to rank candidate tags. However, the authors do not mention how to handle noises caused by misspellings or unrelated tags. Besides, as they use K-means clustering approach, the number of clusters should be defined manually before clustering.

Due to the isolation of services and the lack of social relationships among services, service discovery and composition are not as successful as expected. Chen *et al.* [36] study service discovery problems based on a global social service network, mainly focusing on building services' sociability. They publish distributed services in service social network by following linked data principles. Then, these services are linked to related services with social links. Various requirements are considered while developing services' sociability, *e.g.,* QoS preference and the correlation between two services. Besides, past social interactions and popularity are also considered in this process. However, this approach may obtain a better efficiency if they involve users' feedbacks and social influences in the model.

2.2.2 QoS-aware Service Composition

In view of the large amount of web services offering similar functionalities, there has been considerable volume of research on QoS-aware service composition problem. This problem aims at selecting competitive services to optimize the whole QoS value of the composition. In this process,

optimization algorithms are introduced to reduce the searching space, minimize the searching time and enhance accuracy of obtained solutions.

In-memory Composition Methods

In-memory composition methods load service information into RAM and construct the search graph in RAM. A common strategy is to treat this problem as multiple objective optimization problem which can be solved by heuristic algorithms [17, 37] or Integer Linear Programming (ILP) [22]. Heuristic algorithms can help remove services that are not available or combine equivalent services to reduce the searching space. Planning graph, an AI (Artificial Intelligence) algorithm, is a powerful data structure based on Planning Graph Analysis for reaching a goal state. Recently, Graphplan [38], a planning technique which uses initial conditions, goals and information to reduce the amount of searches, is adopted in service composition to find a solution [20, 39]. Yan modifies the standard Graph Plan Algorithm to extract a composition solution, the planning graph can be built in polynomial time [39].

Definition 6. *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 7. *The planning graph is a directed layered graph in which the layers of vertices form an alternating sequence of literals and operators:*
$(P_0, A_1, P_1, A_2, P_2, A_3, P_3, \ldots, A_{\mathrm{k}}, P_{\mathrm{k}})$

In the planning graph, the initial layer P_0 denotes the initial state of the planning problem. P is a set of states and A is a set of actions. Each node in A_1 has incoming and outgoing arcs from P_0 to P_1 respectively. The multiple actions in a layer means they are possibly to be executed in parallel.

Definition 8. *A planning problem* [38] *is defined as a triple with the following components:*

- $S \in 2^L$ *is a set of states.*
- *A is a set of actions, an action* a *has a precondition and effect set where* $\mathrm{preconds}(a) \in L$ *and* $\mathrm{effects}(a) \in L$.

- γ *is a state transition function, for a state* s *where* $\text{preconds}(a) \in s, \gamma(s,a) = s \cup \text{effects}(a)$.
- $s_0 \in S$ *is the initial state,* $g \in L$ *is the goal.*

Layer 1 of the planning graph consists of an action layer A_1, and a proposition layer P_1. A_1 is a set of services whose input parameters are a proposition set of P_0:

A_1 = $\{a|$ where preconds$(a) \in P_0\}$.

P_1 is a union of P_0 and the sets of effects of actions in A_1:

P_1= $P_0 \cup$ $\{$effects$(a)|a \in A_1\}$.

The Web service composition problem can be mapped to the Planning Graph Model.

- $w \leftrightarrow a$
- $w_{\text{in}} \leftrightarrow \text{preconds}(a)$
- $w_{\text{out}} \leftrightarrow \text{effects}(a)$
- $C_{\text{in}} \leftrightarrow s_0$
- $C_{\text{out}} \leftrightarrow g$

The Graphplan approach contains two phases: Forward Expand and Backward Search. The Forward Expand phase builds the planning graph from the initial state. We loop over the service repository and add available services into the planning graph, this process ends when no more services can be added in the action layer of the planning graph (we call this layer fixed layer). Time complexity of this process is polynomial in the length of initial state, the number of services and concepts [40]. If the goal is contained in the fixed layer, we say a solution exists, thus, the service composition problem is converted into a reachable problem.

Yan *et al.,* QoS driven approach [16] is a three steps search, they hold a point that "the longer the sequence is, the longer the response time will be", based on this belief, they try to find the shortest path. This viewpoint is not correct, a short sequence can only guarantee a short execution time, but is irrelevant to the response time. We give an example to prove our points. Supposing we have a planning graph (Figure 2.3), in this graph, the input (resp. output) parameter set is $C_{\text{in}} = \{A, B, C\}$ (resp. $C_{\text{out}} = \{F\}$), the response time of each service is shown in Table 2.3.

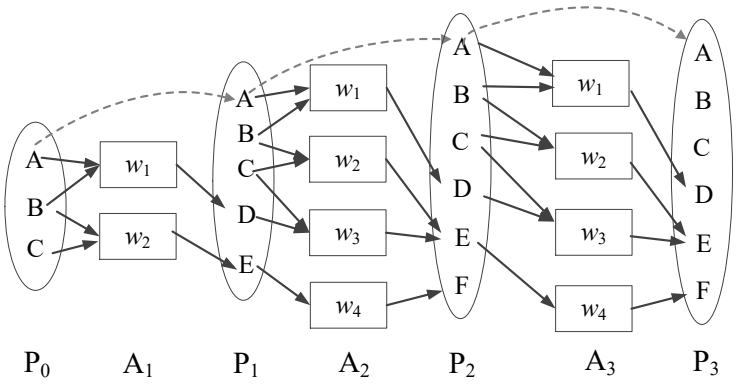


Figure 2.3: A planning graph example.

Table 2.3: Service Information

Service	Input	Output	Response time (ms)
w_1	A, B	D	100
w_2	B, C	E	200
w_3	C, D	E	30
w_4	E	F	80

To shortest solution path is: $w_2 \rightarrow w_4$, in this situation, the total response time is 280, we can also achieve the goal by $w_1 \rightarrow w_3 \rightarrow w_4$, in this situation, the total response time is 210. Therefore, we can conclude that the length of the sequence has no relation with the response time.

In the pruning algorithm [41], the authors calculate and record single QoS values in the Forward Expand phrase. For each concept, they record service names which produce this concept and supply best QoS value so far. Backward Search loops from the goal layer to the first layer and retrieves a solution.

Integer Liner programming (ILP) has been applied in finding an optimal solution in recent research [21, 42]. In an ILP problem, some or all of the variables are restricted to be integers. Zeng *et al.* consider multiple non-function criteria in service selection process and apply dynamic global optimization method in composition process [21]. Berbner *et al.* propose a MIP (Mixed Integer Programming) formulation which is more feasible in dynamic real-time environments [42].

ILP technique has the drawback of exponentially increased computation complexity and cost when the number of variables increases. Taking this into consideration, Alrifai and Risse combine

global optimization with local selection techniques to solve this problem [43]. They decompose each QoS constraint into a set of local constraints which serve as upper bounds, then, local selection is applied independently. Their method can find a close to optimal solution while reducing the computational time. Canfora *et al.* use Genetic Algorithms to handle QoS attributes which have non-liner aggregation functions [44]. Experimental results show this method is able to keep a constant timing performance.

Recently, researchers propose a similarity-based approach to deal with the situation that no feasible service composition returns. Among them, relaxation method is an outstanding one [45, 46]. Lin *et al.* propose a Relaxable QoS-based Service Selection Algorithm (RQSS) to find an approximate solution [45]. This algorithm relaxes the degree of constraints and recommends a similar solution in case no feasible solution satisfies the constraints. Yu *et al.* module service selection problem in both combinatorial and graph models, their heuristic algorithms can return near-optimal solutions in polynomial time [46]. Mabrouk *et al.* present a near-to-optimal method in which the whole composition task is divided into pieces [47]. In this method, they use K-means to group service candidates into multiple QoS levels, and select multiple services for each subtask. The main concern of this method is: it may fall into the "local maximum" problem (A local maximum is a maximum within some neighbors but not the global maximum [48]).

Database-based Composition Methods

In-memory composition methods work when data fits in RAM. However, the searching space is limited by the available physical memory. Researchers have been motivated to utilize relational database to solve service composition problem [3, 23, 49, 50].

Utkarsh *et al.* [3] present a Web Service Management System (WSMS) which transforms the service composition problem into a query optimization problem in database. In the WSMS system, they first build virtual tables for services' inputs and manage service interfaces, then, they use a multi-thread pipeline executive mechanism to improve the efficiency in searching services. The WSMS system contains three main components and is illustrated in Figure 2.4). The *Metadata* component registers new web services and maps their schemas to an integrated view. The *Query Processing and Optimization* component chooses and executes query plans whose operators may

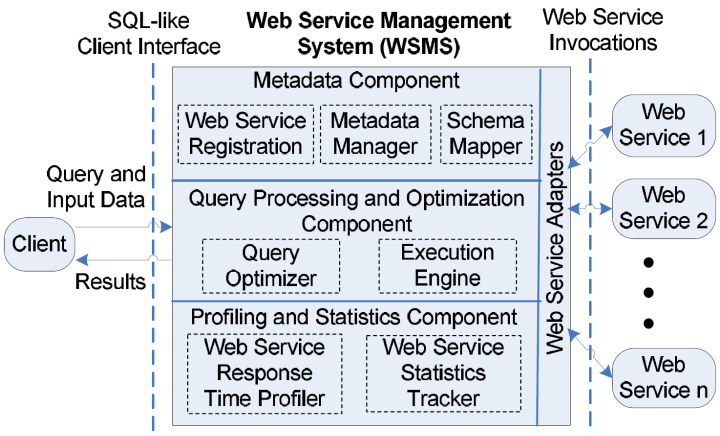


Figure 2.4: A web service management system (WSMS) [3].

invoke services. The *Profiling and Statistics* component profiles services for their response time and maintains these values.

Lakshmi and Mohanty in [49] use a relational database as a repository to store services in UDDI. They use "join" operator to find service composition by matching services. Zeng *et al.* [50] present a web service matching algorithm (SMA), which considers semantic similarities based on WordNet. Moreover, they put forward a Fast-EP service composition algorithm which can be applied in relational database. Although 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 inputs for another service. Another work [23] 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 a relational database. Searches are done by specifying SQL statements. In the PSR system, services are abstracted as single operators and paths have single inputs (outputs). To handle user query with multiple goals, the PSR system searches all paths whose outputs are a part of user query and combines them together. Our work is distinct from [23] 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 a 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.

Cloud Service Composition

Cloud services are available to users via the Internet from a cloud computing provider's servers, on the contrary, web services may be provided from a company's own on-premises servers. Cloud services are designed to provide easy, scalable access to applications, resources, and are fully managed by cloud services providers. Cloud service resources are: Software as a Service (SaaS), Platform as a Service (PaaS) and Infrastructure as a Service (IaaS). Compared with in-memory approaches, cloud service composition is long term based and more economical, some companies such as Amazon, Google and IBM are offering cloud composition solutions to the market.

Definition 9. *A cloud service base* $C = \{sf_k | 1 \leq k \leq m\}$ *is a set of service files and* sf_k *is a service file published by a provider.*

An overall cloud service composition structure is illustrated in Figure 2.5. In Cloud environment, service requester describes request and goal using service ontologies. Services are stored in the Cloud from which an appropriate service combination is selected. The cloud convertor transforms the composition request and the selected service combination into a composition domain [51].

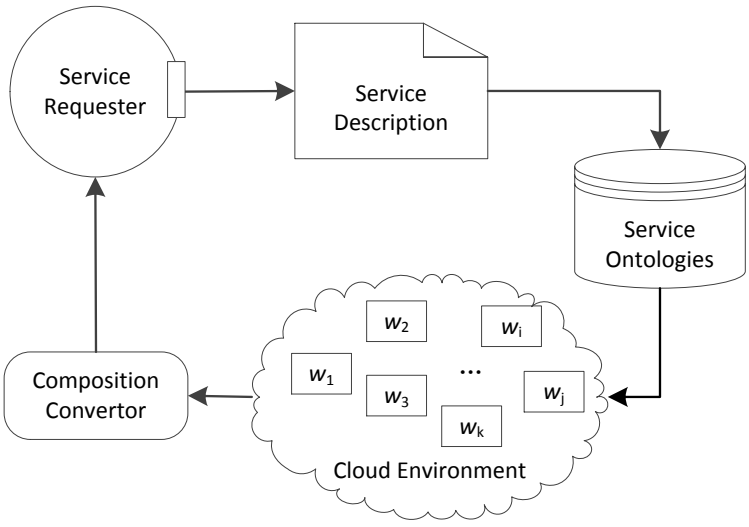


Figure 2.5: Service composition in Cloud environment.

Service composition approaches were firstly applied in cloud systems in 2009. Authors in [52]

firstly decompose and process main elements of WSDL documents of web services, so these elements can be extended to adapt to new service description standards. After using a database to store single and composite web services, a SMA algorithm is presented to search matched services in the database, SMA considers semantic similarity between inputs and outputs based on WordNet. At last, a Fast-EP algorithm is applied to find composition solutions. Though service composition has been extensively studied, cloud service composition is still a new topic and many challenging issues like real-time requirements, QoS model for cloud, network protocols are waiting to be addressed.

Zou *et al.* [51] study the service composition problem in multiple clouds base (MCB) environments and propose three planning combination methods to find composition plans. In the all clouds combination method, they first put all clouds in MCB as inputs for composition conversion. Then, they execute a composition planner to find a solution. This method can find a solution quickly yet at a high cost. The base cloud combination method finds a cloud combination with a minimum number of clouds after enumerating all possible cloud combinations, this results in high time complexity. To solve the shortcomings of those two methods, a smart cloud combination method is presented, which first models MCB as a tree and then finds a minimum cloud set by searching the tree.

Duan analyzes achievable performance of cloud service provisioning and proposes a cloud service provisioning system [53]. In his system, SOA is applied in network virtualization, therefore, networking and cloud computing system are integrated into a composite service provisioning system. Bao and Dou hold the point that, services in cloud environment are not segregated and irrelevant [54]. Based on this consideration, they adopt a Finite State Machine (FSM) to declare the order among services in the cloud. They also propose an improved tree-pruning-based algorithm to solve the service composition problem. Huang *et al,* address the QoS-aware composition across network and cloud services problem [55]. They formulate the composition problem as a Multi-Constrained Optimal Path (MCOP) problem. Theoretical analysis is given to show the efficiency of their algorithm.

Recent research on quality-based cloud services are mainly driven by service selection and composition, using a Bayesian [56] network or Markovian [57] decision process. In [56], an economic model is constructed for users to model their long term behaviors. They utilize Bayesian Network to represent users' models and present the composition problem with an Influence Diagram. Authors

in [57] present a "Smart Virtual Machine Provisioner" learning system that dynamically allocates resources in the cloud. More research on cloud service composition can be found in survey [58].

Table 2.4: A detailed classification of selected research prototypes

Category	Author	Year	Heuristic	QoS	Global	multiple QoS	Anytime	Semantic
in-memory	Huang et al. [41]	2009	planning	√	√			√
	Yan et al. [16]	2009	BFS	√				√
	Zeng et al. [21]	2003	ILP	√	√	√		
	Cui et al. [22]	2011	ILP	√	√	√		√
	Yan et al. [39]	2008	Graphplan	√				√
	Yan et al. [20]	2012	Graphplan	√			√	√
	Mabrouk et al. [47]	2009	clustering	√	near-to			√
	Lin et al. [45]	2011	relaxation	√	near-to			√
	Zeng et al. [50]	2010						√
database	Lee et al. [23]	2011						√

2.2.3 Redundant Services Discovery and Removal

Most web services have multiple input and output parameters, the reproduction of same output parameters by different services lead to redundant services. Current methods have the drawback of including redundant web services in the solutions. The existence of redundant services lead to undesired side-effects (*e.g.,* longer execution time, larger space requirement, and unintended actions). We study redundant service problems and find non-redundant solutions to avoid those undesired side-effects.

We define a parameter as a key parameter if it is used as an input parameter of a service.

Definition 10. *A service is redundant if all its key outputs are produced by other services.*

Example: Figure 2.6 shows a layered graph of services and parameters, w_2 has a key output F as this parameter is used by w_4. Similarly, w_3 has a key output E, but E is also produced by w_2. As the key output of w_3 is produced by other services, w_3 is a redundant service and can be removed.

Definition 11. *A service is removable if none of its outputs is a key parameter.*

Figure 2.7 shows a new layered graph which removes service w_3 from Figure 2.6. As we can see, the output parameter D of service w_1 is not used by any other service, therefore, w_1 is removable.

Zheng and Yan [59] propose four strategies to prune redundant services in the forward expand stage: avoid adding a service if its outputs are existing in previous proposition levels or are produced

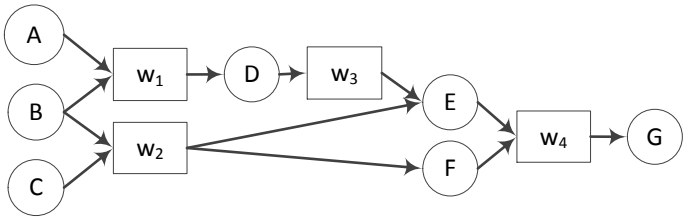


Figure 2.6: A layered graph example.

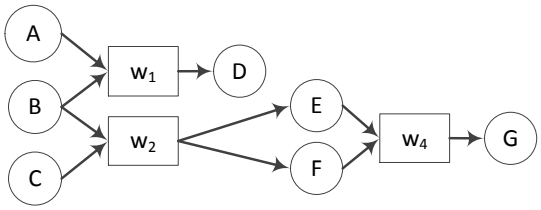


Figure 2.7: A layered graph after w_3 is removed from Figure 2.6.

by other existing services, delay adding a service whose outputs are not used in the next action layer, stop expanding the graph if the goals are found in a proposition layer. Lin *et al*. [19] propose a *service threshold* mechanism to reduce the number of services in the search stage, that is, fix the maximum number of services can be invoked in a solution, solutions with too many services or with more services but are similar to a shorter solution are removed. In the particle swarm optimisation composition technique [60], the authors use a greedy optimisation algorithm to extract non-redundant solutions from a graph showing all service connections.

To check whether or not a service is redundant, the simplest way is to remove that service and recompute the QoS value of the new plan. The redundant service removal method is suitable as a last step to optimize the composition solution [25]. Kwon and Lee [61] propose a two-phase algorithm based non-redundant composition system–NRC, in which, the forward phase finds candidate compositions and the backward phase decomposes compositions into several non-redundant solutions by using tokens.

2.2.4 Revise a Broken Service Composition

In an automated service composition, selected services may fail to work, a broken service should be discovered and replaced so the process is not interrupted. Replanning is unavoidable if Web

Service Level Agreement (WSLA) [62] violations are detected. Three kinds of technologies have been proposed to revise a broken service composition: replacement, recomposition and replanning.

First, **replacement**. When a service fails to work, instead of stopping the whole process, this technique replaces the broken service by a proper one. Based on the replacement path idea in [63], Yu and Lin raise a CSPR algorithm which uses backup services to re-construct the process [64]. The backup path is produced offline during the service composition. Unfortunately, their backup method is feasible when there is only one faulty service. The algorithm fails to find a replacement if two or more services fail. Replacement is preferred when a service is faulty or has a bad QoS. However, replacement cannot deal with the needs of adding or removing services.

A second solution is **recomposition**. In a recomposition process, service replacement and reconfiguration are necessary. To guarantee the service dependencies, the state of a service should be transferred to another one if it is replaced. This method deals with the possibility of adding and removing services and is applied in [65, 66, 67]. Lin *et al.* [65] present a reconfiguration solution to support multiple faulty services. An iterative algorithm is used to decide reconfiguration regions, this algorithm stops when a solution is found or the number of services exceeds expectation. Later, Lin *et al.* extend the work of [65] and implement the solution in Llama architecture [66]. They claim that, the region based algorithm reduces the recomposition complexity. However, recomposition is time-consuming and quite costly since a new business process should be computed. Zhai *et al.* [67] propose a services reconfiguration solution to handle multiple service failures in the business process.

As stated above, it is undesirable to recompose a process when a few services fail to work. In 2002, a **replanning** algorithm was introduced to determine which actions should be removed in a failed plan [68]. The term **plan repair** was first introduced from a theoretical perspective in the AI area [69]. Plan repair may be used to solve broken composition as it consists of both removing and adding actions. However, the relationship in the web services is different from operations in AI planning. In order to correct a broken service composition, plans should be extended into a new solution.

2.2.5 Other Related Research

QoS Predication

QoS measures how well a service serves the user. Server-side QoS property values (*e.g.,* cost) are always provided by service providers. Such QoS information is identical and static. In contrary, user-observed QoS property values (*e.g.,* response time, throughput, reliability) are influenced by Internet conditions and employ time. Amin *et al.* discover that, the QoS value of the same service may vary over different employing time [70]. Based on the analysis on real-world web services, Zheng *et al.* [71] demonstrate that the QoS values are not identical to different consumers even when calling the same service. Therefore, Menasce suggests providers monitor and check whether or not their services meet the agreed SLAs, while users check the quality of services they obtain [72]. Also, QoS predication approaches are developed under different environments [73, 74, 75].

Quality of Service Management (QoSM) is a generation of business event processing systems that supports QoS monitoring, collection and prediction. Employing Time series models to predict QoS values has become a popular research trend. Zeng *et al.* suggest that historical data can help predict future performance metrics [73]. Based on this assumption, they build an event-driven QoS predication system to predict performance metrics.

QoS predication helps prevent QoS violations. Shao *et al.* present a collaborative filtering approach to predict QoS values by referencing other customers who have similar historical experiences [74]. They assume that customers who have similar experiences on some services will have similar experiences on other services. These similarities include both inexplicit (*e.g.,* users' preferences) and explicit (*e.g.,* network environment) features. Similarly, the authors in [75] predict service's QoS values based on other similar services in the cloud. The predication is done by calculating similarity scores between cloud services and end users' internal features. By analyzing real datasets, Amin *et al.* point QoS attributes' behaviors are nonlinear [70]. They integrate ARIMA (AutoRegressive Integrated Moving Average) and SETARMA (Self Exciting Threshold ARMA) time series models to capture the dynamic QoS behavior.

Chapter 3

Full Solution Indexing for top-K Web Service Composition

3.1 Introduction

In this chapter, we study QoS-aware service composition problem, which selects the composite of services with the best QoS value. Many in-memory approaches based on different kinds of techniques have been proposed to find solutions. These include A* methods [15, 76], planning-graph model [19, 59] and Integer Linear programming (ILP) [22, 42]. Planning-graph is a powerful approach to solve the composition problem, this approach contains two stages: a forward expand stage constructs a search graph and a backward searching stage retrieves a solution. To find the solution with the best QoS value, this method checks all services' combinations in the backward searching stage, thus, the complexity is NP-complete. ILP may find an optimal solution, however, this algorithm leads 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 [77, 78]. In local selection methodology, services in different layers are selected independently, therefore, the problem can be solved in polynomial time, but local selection methodology is easily falling into the "local maximum" problem.

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 searching 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.

Attempts to resolve above mentioned problems have resulted in the utilization of relational database [3, 23, 49]. Utkarsh *et al.* [3] build virtual tables to manage service interfaces and present a Web Service Management System (WSMS). WSMS transforms the service composition problem into a query optimization problem in database. 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 [23]. 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. The system developed by Lakshmi and Mohanty [49] assumes all the inputs of a service can be provided by another service. However, in real world, normally a service only provides part inputs for another service.

In this chapter, 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).

The main contributions presented in this chapter can be summarized as follows:

- We address the quality of services and calculate globally optimized QoS values.
- We use a relational database to find solutions for web service composition problems 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 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.

The rest of this chapter is organized as follows. Section 3.2 gives the motivation to search the solution with a relational database. We introduce preliminary knowledge in Section 3.3. The system architecture and algorithms are given in Section 3.4, we use examples to explain how paths are generated and queried, we also discuss how to update the database in this section. A case study is demonstrated in Section 3.5 to explain our algorithms. We present the experimental results in Section 3.6. Finally, the conclusion is drawn in Section 3.7.

3.2 Motivation

In-memory algorithms can find a solution in a short time when the number of services is in small-scale. However, these algorithms have some shortcomings:

- When the number of services increases, the search space grows exponentially large and beyond the limitation of available physical memory.
- Loading lots of services information into RAM is expensive (even today).
- It makes little sense to lock up all the data in RAM just for composition and it is a waste of resources.

To find a solution with a database is feasible if we can map the problem to records in the database. Yet, we need to overcome several difficulties. First of all, we use services with multiple input and output parameters, and there are always more than one goals to be achieved. We define a service combination as a path, normally, a path has a start node and an end node, in this thesis, we define a path has multiple inputs and outputs. Second, normally a path only represents sequential connections, in our research, multiple services are invoked simultaneous, the path here should support both sequential and parallel connections. Third, we need to find an efficient way to query a path in the database with multiple inputs and outputs. Last but not least, how to update the database in case services are added or disappeared. We present our solutions in the following subsections.

3.3 Preliminary

Definition 12. *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.

We show a set of available services in Table 3.1, In which, we have four parameters (A, B, C, D) and two services (w_1, w_2). Figure 3.1 is an example of a layered graph which represents the connection of services in Table 3.1. We use circles to represent parameters and use rectangles to represent services. 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, therefore, we say w_1 is an input service of w_2.

Table 3.1: A set of available services

Service	Input	Output
w_1	A, B	C
w_2	B, C	D

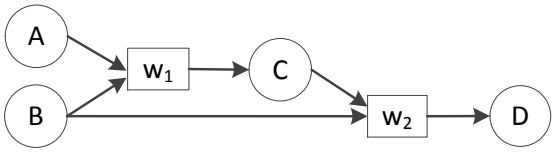


Figure 3.1: A Layered graph example of Table 3.1.

Definition 13. *A path is a layered graph defined as a tuple with the following components:*

- $SL = \{W_{\mathrm{k}} | k = 1 : l\}$ *is a set of service layers and l is the number of layers in the path.*
- $path_{\mathrm{in}} = \{\bigcup_{k=1}^{l}\{\{w_{i.in} | w_i \in W_k\} - \{\bigcup_{m=1}^{m<k}\{w_{j.out} | w_j \in W_m\}\}\}\}$ *is a finite set of typed input parameters. 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.*

Three paths can be found in Figure 3.1 and are listed in Table 3.2.

Table 3.2: Paths in Figure 3.1

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

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

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

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

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

Definition 14. *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 path satisfies the condition:* $\{C_{\text{in}} \supseteq path_{\text{in}}\} \wedge \{C_{\text{out}} \subseteq path_{\text{out}}\}$.

For example, the user specifies his requirement $(C_{\text{in}}, C_{\text{out}}) = (\{A, B\}, \{C\})$, we find two paths ID = 1 and ID = 3 (Table 3.2) meet the requirements (figure 3.1).

3.4 Architecture and Algorithm

Now we describe the framework of our proposed FSIDB system. Figure 3.2 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

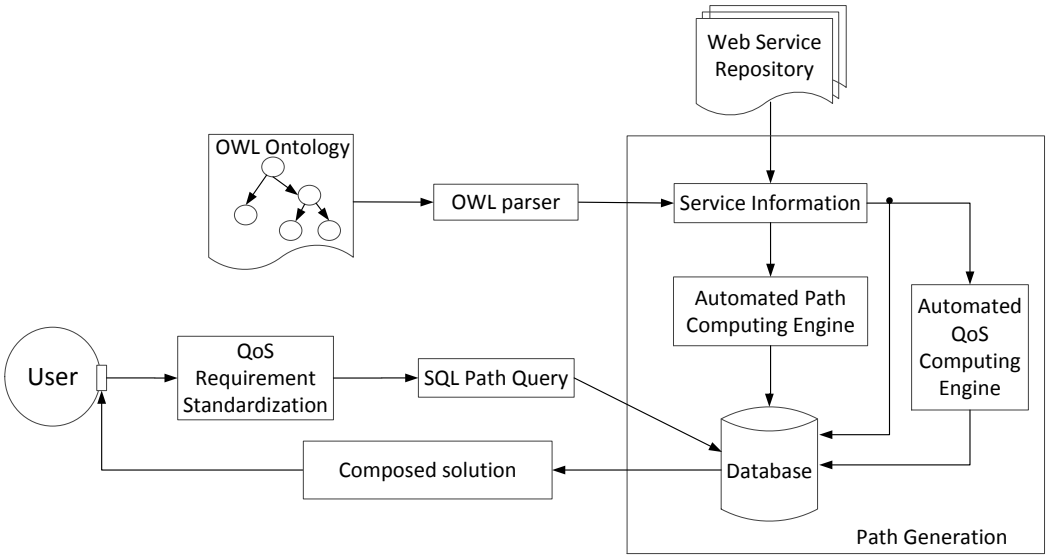


Figure 3.2: Architectural overview of FSIDB system.

an OWL parser. The OWL ontology is used to infer semantic relationships among services. The information of services such as the name, input, output concepts and 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 3.3. 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 layer. All the outputs of services in the path compose the outputs of this path. This procedure ends when no more paths can be generated. Similarly, "Automated QoS Computing Engine" calculates QoS values of paths according to algorithm 6 **CalculateQoS**.

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 Section 3.4.2). After that, a SQL path query is generated. We use this SQL statement

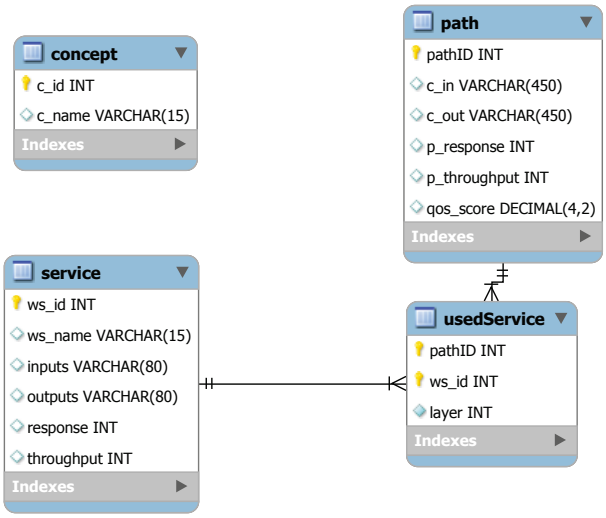


Figure 3.3: Relational schema of database.

to query the database, a list of QoS-aware solution paths are returned to the user. This procedure is described further in Section 3.4.2.

3.4.1 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 thesis, 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 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*** 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 input (resp.output) concepts of w are inputs (resp.outputs)

of $path$ (line 5-7).

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: create a new path $path$
4: $path.w \leftarrow w$
5: $path.pathID \leftarrow counter$
6: $path.in \leftarrow w.in$
7: $path.out \leftarrow w.out$
8: $path.resp \leftarrow w.resp$
9: $path.thp \leftarrow w.thp$
10: $path.cost \leftarrow w.cost$
11: $path.avail \leftarrow w.avail$
12: $path.relib \leftarrow w.relib$
13: $sPathSet \leftarrow sPathSet \cup path$
14: $counter \leftarrow counter + 1$
15: **end for**
16: **return** $sPathSet$

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 in $pathS$, we create a new path $path$ by connecting $path$ and $pathM$ (line 5-8). The order of service layers of $path$ is decided by Algorithm 4 **FindLayer**.

Algorithm 4 ***FindLayer*** checks in which layer $PathS$ can 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 5 ***AddLayer*** decides the order of services layers in the newly created path. If $pathS$ can be added in front of $pathM$ (line 2), the service in $pathS$ 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 3 *MulPathSetBuild(i)*

Input: $sPathSet, mulPathSets(i-1)$;
Output: $mulPathSets(i)$: a set of 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 $path$
      $path.w \leftarrow AddLayer(pathS, pathM, path)$
      $path.pathID \leftarrow counter$
      $path.in \leftarrow pathS.in \cup pathM.in$
      $path.in \leftarrow path.in \setminus pathS.out$
      $path.out \leftarrow pathS.out \cup pathM.out$
      CalculateQoS(pathS, pathM, path)
      $mulPathSets(i) \leftarrow mulPathSets(i) \cup path$
      $counter \leftarrow counter + 1$
    end if
  end for
end for
return $mulPathSets(i)$

Algorithm 4 *FindLayer*

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

$i \leftarrow 1$
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 *AddLayer*

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

```
1:  f ← FindLayer(pathS, pathM)
2:  if f = 1 then
3:     path.layer(1).w ← pathS.layer(1).w
4:     for each service layer i of pathM do
5:        path.layer(i + 1).w ← pathM.layer(i).w
6:     end for
7:  else
8:     for each service layer i of pathM do
9:        path.layer(i).w ← pathM.layer(i).w
10:       if i = f − 1 then
11:          path.layer(i).w ← path.layer(i).w ∪ pathS.w
12:       end if
13:    end for
14: end if
15: return path
```

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 (1)- Equation (10).

Table 3.3 gives an example to show how to calculate the response time of the new path. Row 1 and row 2 show two paths before connection, we discuss three possibilities for connection.

Case 1: $FindLayer = 1$, $pathS$ is added before $pathM$, according to Algorithm 6 **Calculate-QoS** (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 **CalculateQoS** (line 9), $path.resp = pathM.resp$.

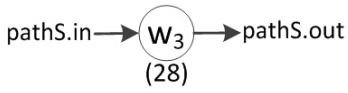
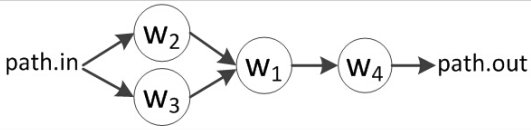
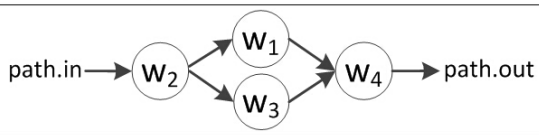
Case 3: $FindLayer = 3$, service of $pathS$ (w_3) is added in the second layer and executes in parallel with w_1, $pathS.resp > pathM.layer(2).resp$, according to Algorithm 6 **CalculateQoS** (line 7), $path.resp = pathS.resp + pathM.resp - pathM.layer(2).resp$.

Algorithm 6 *CalculateQoS*

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

```
f = FindLayer(pathS, pathM)
if f = 1 then
    path.resp ← pathS.resp + pathM.resp
else
    i = f − 1
    if pathS.resp > pathM.layer(i).resp then
        path.resp ← pathM.resp + pathS.resp − pathM.layer(i).resp
    else
        path.resp ← pathM.resp
    end if
end if
path.thp ← min{pathS.thp, pathM.thp}
path.cost ← pathS.cost + pathM.cost
path.avail ← pathS.avail × pathM.avail
path.relib ← pathS.relib × pathM.relib
return path
```

Table 3.3: Generate new paths

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

3.4.2 Path Query

After all paths are generated, we query the database for service composition solutions. First, we standardize user's QoS requirement, to make sure there is no confusion for the FSIDB system.

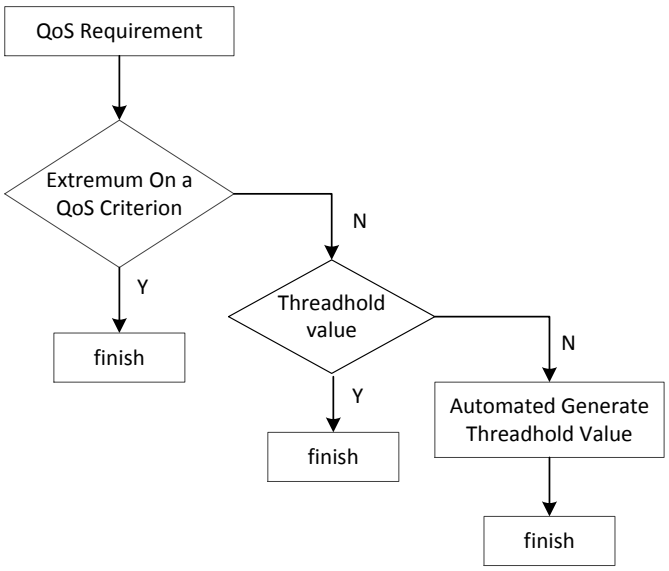


Figure 3.4: Standardizing QoS requirement.

The process of "QoS requirement standardization" is shown in Figure 3.4. 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 a threshold value for the QoS criteria, *e.g.,* "price less than $500", in this situation, a set of satisfying solutions are returned. If the requirement is dim, for example "a cheap price", it automatically picks a threshold value and returns satisfying solutions to the user.

Then, we generate SQL statements 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 user's QoS requirements . Finally, search in the "UsedService" and "service" table for services in the path. This procedure is detailed described in Algorithm 7 **Service composition queries**.

3.4.3 Database Update

Since services on the web 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 section, we shall discuss how to add and remove services in the database.

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** $solServices$

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 detail in Algorithm 8 **Delete service**.

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'

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

Service Addition

The basic process of adding a service in the database is as follows: create a new path $npath$ for the newly added services w, connect $npath$ 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:

Supposing 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 $O(m \times nlog(m \times n))$.

Service Update

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

3.5 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 3.4, which contains inputs, outputs, response time, throughput and cost.

Table 3.4: A set of available services

ws_id	Service	Input	Output	Response	Throughput	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

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 3.5. The sequences of services in paths are decided via Algorithm 4 **FindLayer** and Algorithm 5 **AddLayer** (Table 3.6).

Table 3.5: Path table

pathID	c_in	c_out
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

Table 3.6: 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
...	...	...

QoS values of paths are calculated in Algorithm 6 **CalculateQoS** and stored in the "path" table (Table 3.7). The single QoS value of a path is calculated via Equation 1 to Equation 10. The utility QoS value of a path is obtained via Equation 14.

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

Example 1. *The user wants to find a path with the minimum response time. This process contains two phases, first, find the path that meet user's functional constrains with minimum response time. Then, search for services in this path. The searching 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:

Table 3.7: 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

SELECT ws_name FROM service WHERE ws_id IN (

SELECT ws_id FROM UsedService WHERE pathID=8);

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

Example 2. *User sets his QoS constraints as "response time $<$ 110 and throughput $\geq$ 2000". In this case, the searching 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.

Example 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}.

Example 4. *To calculate the utility value of service w_1: We use Equation 11 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 12 to scale their utility value. thus, we have: $U_{resp}(w_1) = 0.5, U_{thp}(w_1) = 1, U_{cost}(w_1) = 0.3$*

Supposing each QoS criterion has a same weight, with Equation (13), the overall utility score of service w_1 is $U(w_1)$=0.6

Example 5. *To get the QoS score of a path, combine Equation (1) to Equation (13), we have:*

$$U(path) = \sum_{i=1}^{n} U_i(path) \times W_i = \frac{T(path) - T^{min}}{T^{max} - T^{min}} \times W_{thp} + \frac{l \times R^{max} - R(path)}{l \times (R^{max} - R^{min})} \times W_{resp} + \frac{m \times C^{max} - C(path)}{m \times (C^{max} - C^{min})} \times W_{cost} \quad (15)$$

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

According to Equation 15, U(path8)=0.41

3.6 Empirical 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 wall-clock execution time.

3.6.1 Data Set

We use TestsetGenerator2009 [79] 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 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.

3.6.2 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 [25]. 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 longer response time or smaller throughput, so it is quite possible that these paths are eliminated in this stage.

Execution Time for Service Composition Search

Figure 3.5 to Figure 3.7 show the execution time of the first stage in path query, which find paths that meet user's functional and non-functional requirements. Case 1–case 3 represent three possibilities of QoS constraints as shown in Section 3.5. 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.

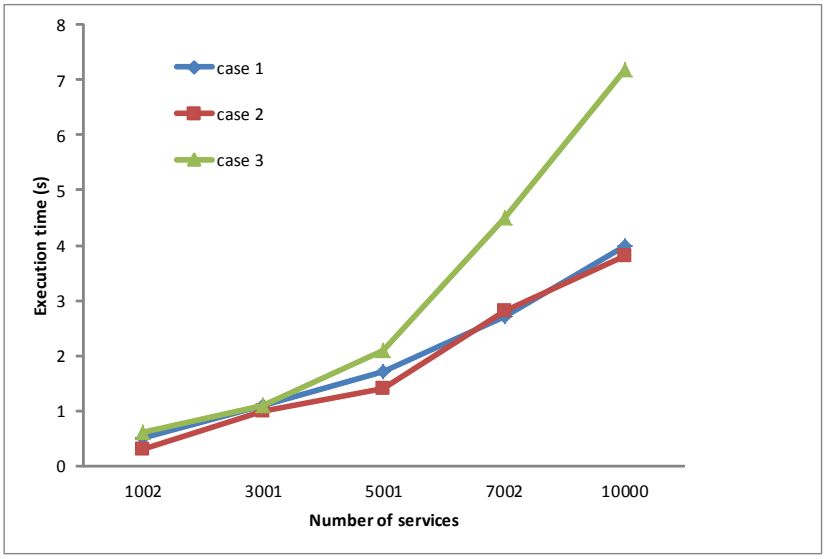


Figure 3.5: Time for searching solutions with optimal response time.

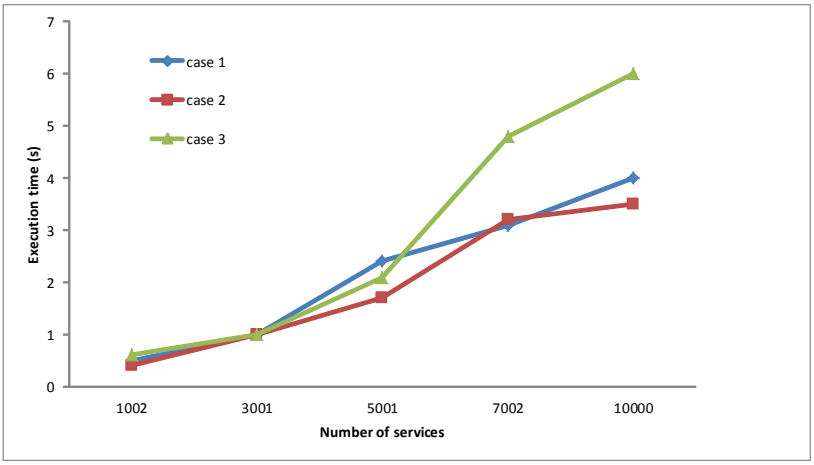


Figure 3.6: Time for searching solutions with optimal throughput.

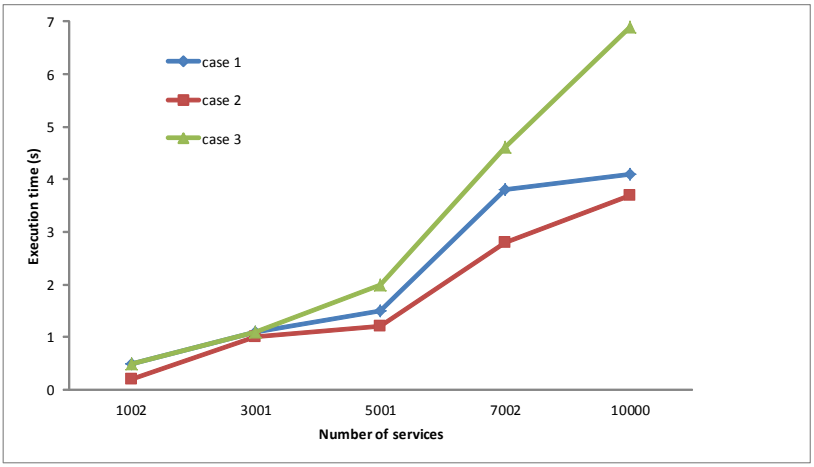


Figure 3.7: Time for searching solutions with optimal utility score.

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 3.4.3. We build index on services' names, and compare the performance with and without index in terms of execution time. Figure 3.8 shows the results. As expected, it takes more time to find and delete records when there is no index on services' name.

Time for Service Addition

In the experiment, we measure the time spent on adding a new service in the database. The process of service addition is described in detail in Section 3.4.3. From figure 3.9, we can see as the number of original services in the database increased, the execution time increased dramatically.

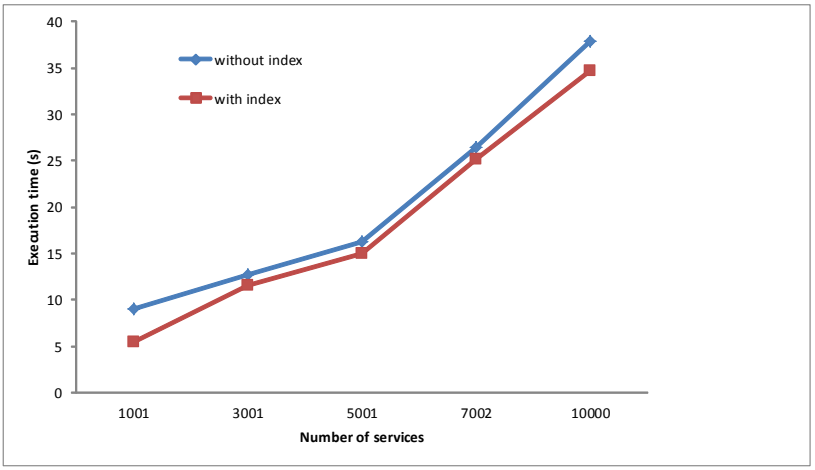


Figure 3.8: Time for service disappearance.

This is because, when the number of web services increases, the number of newly generated paths increases linearly.

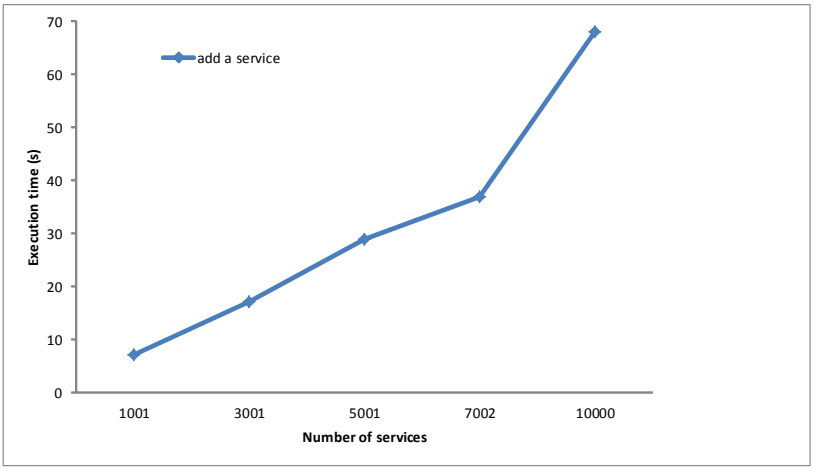


Figure 3.9: Time for service addition.

3.7 Summary

In this chapter, we find top-K solutions for 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. We fetch solutions by

converting 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 fuzzy query on multiple QoS criteria. We present algorithms to update the database *e.g.,* service disappearance, addition and update, and analyse time complexity of these operations. Compared with in-memory methods, the service combinations are stored in the database and are reusable. Besides, our system takes advantage of the large space available on persistent disk. Finally, we find and rank top-K paths according to the comparison of their QoS values; these paths provide backup solutions.

Chapter 4

Scaling up Web Service Composition with the Skyline Operator

4.1 Introduction

Searching an optimal composition solution with both functional and non-functional requirements is a computationally demanding problem: the time and space requirements may be infeasible due to the high number of available services. To alleviate this problem, in this chapter, we propose the application of a skyline operation to reduce the search space and improve the scalability.

we pre-process services and find a set of candidate services referred to as "skyline services" based on the Skyline operator [80, 81]. Generally, the skyline operator returns all of the elements that are not dominated by another element: an element dominates another one if it is at least as good in every respect, and better in some way. Intuitively, for every element not in the skyline, there is a better element in the skyline, not matter what your criteria. We find skyline services, this allows us to find solutions for large composition problems with less storage and increased speed.

There has been considerable volume of research on skyline analysis problem. Borzsony *et al.,* [80] firstly proposes to extend the database by a "skyline" operation. Two algorithms are proposed in this paper. The block-nested-loops algorithm compares each tuple with chosen tuples and uses a *window* to store incomparable tuples. However, this method is time consuming since each tuple is compared with all the tuples in the window. The goal of the divide and conquer algorithm

is to divide the dataset into partitions, so that each of the partition fits into the main memory. First, the dataset is divided into m partitions by a MapReduce method. Then, this algorithm computes the skyline of each partition using an in-memory method. The final skylines are obtained by a merging algorithm. Wang *et al.* propose a Skyline Space Partitioning (SSP) method to provide efficient processing of unconstrained skyline queries [82]. This method also belongs to grid partitioning method, in which, it maps multi-dimensional data space to a tree structured P2P network. The number of visited nodes is reduced with this technology.

Vlachou [83] *et al.* propose an angle-based space partitioning scheme that can be used in parallel skyline. The angle space partitioning technique firstly maps the cartesian coordinate space to hyperspherical space, then partitions the space into N parts with an angular coordinate. The authors claim: the number of returned local skylines is declined by applying this technology, so the amount of work in the merging stage is decreased.

In the Bitmap-based algorithm proposed in [81], each point is represented as a m bits vector, m is the number of points. This method may handle problems with multiple dimensions or with a small number of points. However, it is not suitable for dynamic datasets, because a new bitmap is needed when a new point is added or disappears. The index approach maps high dimensional points into single dimensional space by using a B^+-tree structure, this approach may find skyline points in batches [81].

Recently, researchers have applied skyline methods in solving web service composition problem to prune less competitive services and reduce space requirement. Alrifai *et al.* leverage skyline as a pre-process step before service composition to remove non-interesting candidates [10]. They use a hierarchical clustering method to find skyline services, the idea is to cluster services into k (k = 2,4,8,16...) clusters and select one service from each cluster. A tree was built to represent dominant relationships. When a composition request comes, they first consider only top service of each class, if the problem can not be solved, they proceed to the next level, repeat the process until a solution is found or the whole tree is searched. The authors also discuss how to increase services' potential so they can be included in composition applications. The one pass algorithm proposed in [11] enumerates and stores skyline service execution plans. However, non-competitive plans may be stored because the enumeration order is not restricted in this method. To avoid this

problem, the authors further propose a dual progressive algorithm in which execution plans are enumerated according to their scores. Wu *et al.* use skyline technology in service selection [84]. First, an angle partitioning method is applied to partition the dataset and compute local skylines, then a merge method is applied to obtain global skylines. Du *et al.* compute the composite service skyline in presence of QoS correlations [85]. In this paper, the authors combine pruning criteria with a min-heap to select skyline services. The efficiency of their approach is proved by experimental results.

In this chapter, we propose to use skyline operators to reduce the searching space and improve the scalability. Also, we present a partial pre-composing approach which chooses popular paths for fast delivery. The main contributions presented in this chapter can be summarized as follows:

- We utilize skyline analysis in solving service composition problem, skyline operators help prune less competitive services and reduce computational space requirement.
- A partial pre-composing approach which chooses popular paths is proposed for fast delivery. When a user request comes, we first search the database to see whether we can find a nearly ready-made solution. Only as a last resort do we search the table with whole paths to find a solution.

The rest of this chapter is organized as follows. Section 4.2 gives the motivation of applying skyline analysis to find a solution. We define skyline services in Section 4.3. The system architecture and algorithms are given in Section 4.4, also, we present a partial pre-composing approach in this section. Experimental results are given in Section 4.5. Finally, the conclusion is drawn in Section 4.6.

4.2 Motivation

To reduce the searching space and improve the scalability of algorithms, we are motivated to search a solution with limited storage requirement. Solving a composition problem with skyline analysis is feasible if we can answer the following listed questions. First, a concept's parent services may have other output concepts, therefore, we need to extend the definition of skyline service, to explain in which situation a service dominates another service. Second, as many services are pruned,

we need to find the tradeoff between the space and solutions.

In reality, different users may have same requests, we are motivated to pick some popular user requests and generate paths for fast delivery. When a user request comes, we first check whether we can find a nearly ready-made solution. Supposing we can find a solution among these chosen paths, there is no need to search the table with whole paths. We present the partial selecting method in the following sections.

4.3 Skyline Services

By convention, irrespective of whether it is a positive or negative criterion, we write $Q_k(w_1) \geq Q_k(w_2)$ (resp. $Q_k(w_1) > Q_k(w_2)$) if w_1 is better or equal (resp. better) than w_2 according to criterion Q_k.

We say w_1 dominates w_2 on concept c denotes as $w_1 \prec w_2$ if and only if w_1 and w_2 have same output concept c, w_1 is as good or better in all criteria in Q and better in at least one criterion in Q.

Definition 15. $w_1 \prec w_2 \Leftrightarrow Q_k(w_1) \geq Q_k(w_2) \quad \forall k \in \{1, |Q|\}$ *and* $\exists k \in \{1, |Q|\}$ *such that* $Q_k(w_1) > Q_k(w_2)$.

Supposing W is a finite set of services, w_1 and w_2 are two services in W, $|Q|$ is the number of criteria.

Given a set of web services, the skyline services are the services that are not dominated by any other services. For example, Table 4.1 shows a list of services and their QoS values, each service is described by two QoS parameters, namely response time and throughput. Response time is a negative criterion, the higher the value, the lower the quality. On the contrary, throughput is a positive criterion, the higher the value, the higher the quality. We represent these services as points in 2-dimensional space, x-coordinate represents their response time, and y-coordinate represents their throughput (Figure 4.1). Supposing these services have the same output concepts, according to Definition 15, we find a skyline set $\{w_1, w_2\}$, *i.e.,* there is no other service that offers both shorter response time and higher throughput than w_1. The same holds for services w_2, which are also on the skyline. On the other hand, w_3 does not belong to the skyline set because it is dominated by w_2.

Please note that services in the skyline set provide different trade-offs between the QoS criteria, and hence are incomparable to each other.

Table 4.1: Example of skyline services

service	response	throughput
w_1	60	4000
w_2	280	16000
w_3	340	6000

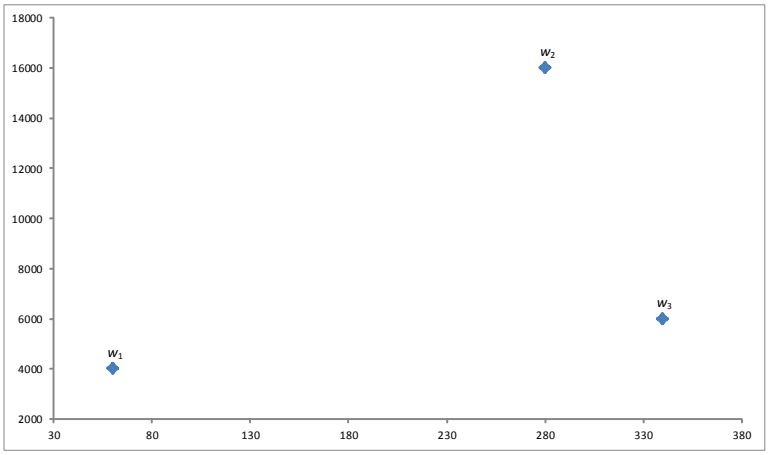


Figure 4.1: Example of skyline services of Table 4.1.

4.4 Architecture and Algorithm

In this section, we present the framework of proposed system and algorithms. Figure 4.2 illustrates an architectural overview of this system.

"Web Service Repository" is a searchable repository which contains information of all services. We find skyline services among a set of web services stored in this repository. For each concept, we find a skyline service set among its parent services, services in this set do not get dominated by each other in terms of response time and throughput. The initialization process takes user's request and skyline services as inputs of the system. We generate all possible service combinations as paths by using FSIDB approach, and store these paths in a relational database. Then, when the user request comes, we compose SQL statements to query the database for paths which meet users' requirements. As different users may have the same request, we use partial pre-composing approach

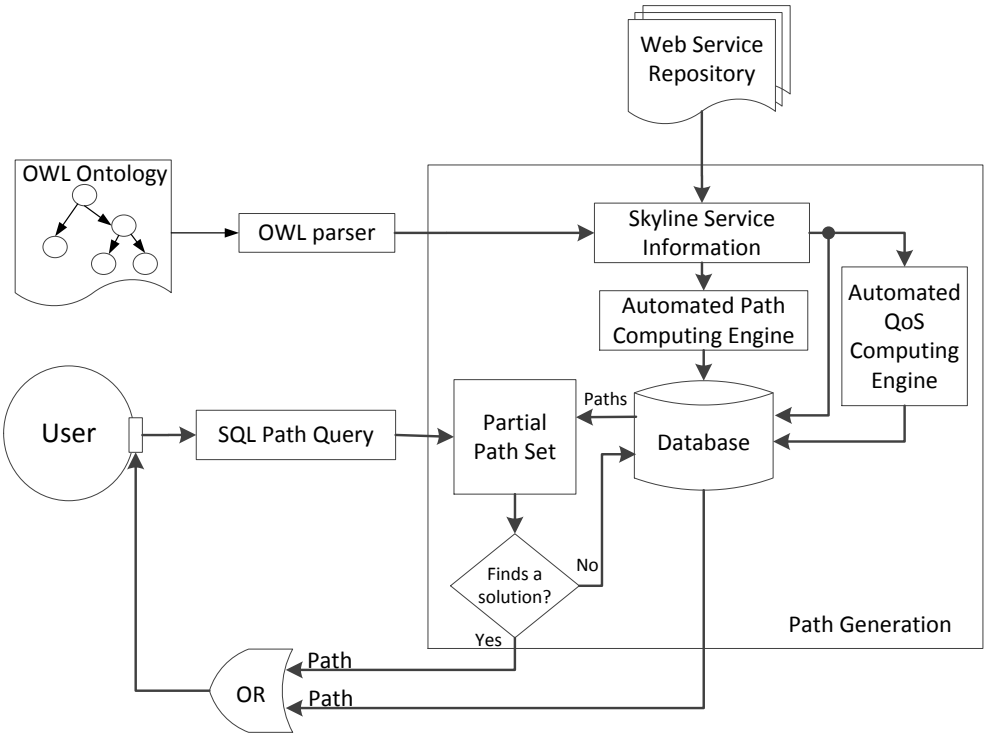


Figure 4.2: Architectural overview.

to pick popular paths and store them in a table (Partial Path Set). If the user request can be answered by these paths, this approach may return a solution in a very short time. We extend algorithms proposed in Chapter 3 and present extended algorithms.

Algorithm 9 ***MulPathSetBuild*** creates paths with multiple services. The number of generated paths decides the path ID of the first created path in this set (line 12). For each service ws in skyline service repository SR and service srv_M in path $pathM$, if Algorithm 10 **CheckRedundant** returns false (line 16), we create a new path $path$ by connecting them together (line 19). The order of services in $path$ is calculated in Algorithm 12 **WriteService**(line 20).

Algorithm 10 ***CheckRedundant*** checks whether or not service ws can be added in path $pathM$. If ws is a skyline service of $pathM$ and provides part of inputs for $pathM$, and ws is not contained in $pathM$, ws can be added in $pathM$.

Algorithm 11 ***CreatePath*** creates a new path $path$ by connecting service ws and path $pathM$ together. The inputs of services in $path$ can be provided by either inputs of $path$ or outputs of services in preceding layers. The response time $resp$ and throughput thp of $path$ are calculated

Algorithm 9 *MulPathSetBuild(i)*

Input: $SR, mulPathSets(i-1)$;
Output: $mulPathSets(i)$: a set of paths with i services;

```
1: if i = 2 then
2:    counter ← 1
3:    for each service srv in SR do
4:       for each service ws in srv.skyline do
5:          path ← createPath(ws, srv, counter, 1)
6:          writeService(ws, pathM, path, 1)
7:          mulPathSets(i) ← mulPathSets(i) ∪ path
8:          counter ← counter + 1
9:       end for
10:   end for
11: else
12:   counter ← pathsSet.size
13:   for each service ws in SR do
14:      for each path pathM in mulPathSets(i − 1) do
15:         for each service srv_M in PathM.ws do
16:            if !checkRedundant(ws, pathM) then
17:               layer ← layer of srv_M
18:               counter ← counter + 1
19:               path ← createPath(ws, pathM, counter, layer)
20:               writeService(ws, pathM, path, layer)
21:               mulPathSets(i) ← mulPathSets(i) ∪ path
22:            end if
23:         end for
24:      end for
25:   end for
26: end if
27: return mulPathSets(i)
```

Algorithm 10 *CheckRedundant*

Input: $ws, pathM$;
Output: $flag$;

```
1: flag ← false
2: if ws ⊈ srv_M.skyline then
3:    flag ← true
4: end if
5: if ws ⊂ pathM.ws then
6:    flag ← true
7: end if
8: if ws.out ⊄ pathM.in then
9:    flag ← true
10: end if
11: return flag
```

according to Equation (1)- Equation (4).The order of service layers of $path$ is decided by Algorithm 12.

Algorithm 11 *CreatePath*

Input: $ws, pathM, counter, layer$;
Output: $path$: a new created path;

1: create a new path $path$
2: $path.pathID \leftarrow counter$
3: $path.in \leftarrow ws.in \cup pathM.in$
4: $path.in \leftarrow path.in \setminus ws.out$
5: $path.out \leftarrow ws.out \cup pathM.out$
6: **if** $layer = 1$ **then**
7: $resp = pathM.resp + ws.resp$
8: **else**
9: **if** $ws.resp > srv\_M.resp$ **then**
10: $diffResp = ws.resp - srv\_M.resp$
11: **else**
12: $diffResp = 0$
13: **end if**
14: $resp = pathM.resp + diffResp$
15: **end if**
16: **if** $ws.thp > srv\_M.thp$ **then**
17: $thp = srv\_M.thp$
18: **else**
19: $thp = ws.thp$
20: **end if**
21: **return** $path$

Algorithm 12 ***WriteService*** decides the order of services in the newly created path $path$. If service ws is added in front of $pathM$ (line 1), ws is added as the first layer of $path$ (line 2), service layers (from 1 to k) of $pathM$ are added as layers (from 2 to $k+1$) of $path$ (line 3-4). If ws is not added in front of $pathM$, layers from 1 to k of $pathM$ are added into $path$ as layers from 1 to k (line 7-8), then we check in which layer service ws should be added and add it into $path$ (line 9-10).

4.4.1 Partial Pre-composing Approach

In reality, different users may have same requests, in this approach, firstly, we pick N popular user requests, then, for each request, we find the path (generated and stored by FSIDB approach) which answers the request and has best QoS value. We store these paths in a separate table (Partial

Algorithm 12 *WriteService*

Input: $ws, pathM, path, layer$;
Output: $path.ws$;

```
1: if layer = 1 then
2:    path.layer(1).ws ← ws
3:    for each service layer i of pathM do
4:       path.layer(i + 1).ws ← pathM.layer(i).ws
5:    end for
6: else
7:    for each service layer i of pathM do
8:       path.layer(i).ws ← pathM.layer(i).ws
9:       if i = layer − 1 then
10:         path.layer(i).ws ← path.layer(i).ws ∪ ws
11:      end if
12:   end for
13: end if
```

Path Set). Services which are not used by any of these paths are seen as non-candidate services and have been pruned ($RemovedServiceSet$). To further decrease the number of used services, we find and remove services which are less used by paths in "Partial Path Set". Then, we need to delete paths in "Partial Path Set" which use removed services ($RemovedPathSet$). For each path in $RemovedPathSet$, we search the database for alternative solutions, if no such solution exists, this query is removed from "Partial Path Set". If alternative paths exist, we pick the one with best QoS value and add it into "Partial Path Set". This process is described in detail in Algorithm 13 **FindAlternativePaths**. $inConcepts$ (resp. $outConcepts$) represents a set of input (resp. output) concept ID, $PathSet$ represents a set of remaining paths.

When a user request comes in, if we can find a path from "Partial Path Set" to find a solution, there is no need to search the whole database.

4.4.2 Graphplan Approach

Planning algorithms are frequently used to find a composition solution, they search the problem space to find a path from initial states to the goal. A planning graph contains two kinds of layers: the proposition (P) layers contain concepts and action (A) layers containing services. Layers of the planning graph form an alternative sequence of proposition layers and action layers. Our in-memory method is based on Graph plan (a planning algorithm [40]) and is called Graphplan approach. The

Algorithm 13 *FindAlternativePaths*

Input: $PartialPathSet, RemovedServiceSet$
Output: $PathSet$;

1: $RemovedPathSet \leftarrow$ Index scan on "Partial Path Set", "usedservice" and "service"
SELECT * FROM $PartialPathSet$ WHERE pathid IN(SELECT pathid FROM usedservice WHERE ws_id IN(SELECT ws_id FROM service WHERE ws_name IN ('$RemovedServiceSet$'))AND pathid IN (
SELECT pathid FROM '$PartialPathSet$'))
2: $PathSet \leftarrow PartialPathSet \setminus RemovedPathSet$
3: **for** each path in $RemovedPathSet$ **do**
4: $AlternativePath \leftarrow$ index scan on table "path", "concept" and "usedservice"
SELECT * FROM path WHERE c_in IN ('$inConcepts$') AND c_out like '$\%outConcepts\%$' AND pathid NOT IN(
SELECT pathid FROM usedservice WHERE ws_id IN(SELECT ws_id FROM service WHERE ws_name IN ('$RemovedServiceSet$')))
ORDER BY QoS ASC LIMIT 1;
5: **if** $AlternativePath \neq \phi$ **then**
6: $PathSet \leftarrow PathSet \cup AlternativePath$
7: **end if**
8: **end for**
9: **return** $PathSet$

Graphplan approach contains two stages: a forward expand stage constructs a planning graph and a backward search stage retrieves a solution. If it is proved that the goal set do exist in the planning graph by the forward expand, a Backward search stage loops from the last layer of the graph for a solution. To find a solution with the optimal QoS value, the Backward search stage needs to check all possible services' combinations, the complexity of this process is NP-complete. If the goal set can not be found in the planning graph, it means the solution does not exist, the composition fails.

To construct a planning graph, first, we add user's initial states to P_0 layer, then, search the service repository for services whose input concepts are all contained in P_0 layer. These services are seen as available services and added into A_1 layer. Add all concepts in P_0 layer and outputs of services in A_0 into P_1 layer, so P_1 is a superset of P_0 layer. We loop the service repository and extend the planning graph layer by layer, this process ends when no more services can be added in the action layer or the goal is reached.

Algorithm 14 ***Graphplan*** is the main algorithm of Graphplan approach. First, algorithm 15 **ForwardExpand** is performed to generate a planning graph (line 1). If the goal contains in the planning graph, algorithm 17 **BackwardSearch** is called to find an executable solution (line 3).

Otherwise, there is no solution for the problem.

Algorithm 14 *Graphplan*

Input: $SR, request, goal$;

1: $pg \leftarrow ForwardExpand(SR, request, goal)$
2: **if** $goal \subseteq pg$ **then**
3: **print** $Backwardsearch(pg, request, goal)$
4: **else**
5: **print** $\emptyset$
6: **end if**

Algorithm 15 ***ForwardExpand*** performs the forward expands to check whether or not a solution exists. The input parameters are SR, $request$, and $goal$, where SR represents the service repository, $request$ is the user request which is taken as input parameters, $goal$ is the goal. We first add $request$ in P_0 layer of the planning graph pg ($pg =< P_0, A_1, P_1, ...A_i, P_i >$). For each action layer in pg, we loop over SR, for each web service srv in SR, if its inputs are in the concept pool $conceptPool$, we add srv in the current action layer $aCurr$ (line 7). For each output concept c of srv (line 8), if c is not in $conceptPool$, add c in $conceptPool$ and update its optimal QoS value (line 10-14). If c exists in $conceptPool$, we compare the optimal QoS value of c and srv, if the latter is better, we update the optimal value of c (line 16-20). After all services in SR are checked, add $aCurr$ into the planning graph pg (line 26). Repeat this process until $goal$ is contained in pg or a fixed layer is reached. The complexity of this process is polynomial in the length of request, the number of services and concepts [40].

Algorithm 16 ***FixedPoint*** checks whether or not a fixed point layer is reached. If the current proposition layer P_i is the same with its previous layer P_{i-1} (line 1), which means no more concepts can be added in the proposition layer of the planning graph pg, the algorithm returns true, otherwise, it returns false.

Algorithm 17 ***BackwardSearch*** performs a backward search to find a suitable solution for the composition problem. The input parameters are pg, $request$ and $goal$, sol represents the solution and $tempSrv$ is a set of candidate services. We loop from the last layer to the first layer of pg (line 2). In each layer, we add services that may generate the best QoS value for the goal as a candidate service (line 4-7). For each service srv in $tempSrv$, add its input concepts into $goal$ and remove its output concepts from $goal$ (line 11-12).

Algorithm 15 *ForwardExpand*

Input: $SR, request, goal$;
Output: pg;

```
1: conceptPool ← request
2: aCurr ← ∅
3: pg.addPLayer(request)
4: while goal ⊈ conceptPool ∪ !Fixedpoint(G) do
5:    for srv ∈ SR do
6:       if srv_in ⊆ conceptPool then
7:          aCurr ← aCurr ∪ srv
8:          for each c in srv_out do
9:             if c ⊈ conceptPool||srv.getOptimalQoS < c.getOptimalQoS then
10:               conceptPool ← conceptPool ∪ c
11:               c.setOptimalQoS(srv.getOptimalQoS)
12:               c.setOptimalService(srv)
13:               for each s in c.getParentSrv do
14:                  s.setOptimalQoS(srv.getOptimalQoS)
15:               end for
16:            end if
17:         end for
18:      end if
19:   end for
20:   pg.addALayer(aCurr)
21:   aCurr ← ∅
22: end while
23: return pg
```

Algorithm 16 *FixedPoint*

Input: pg;

```
1: if P_i = P_{i-1} then
2:    return true;
3: else
4:    return false;
5: end if
```

Algorithm 17 *BackwardSearch*

Input: $pg, request, goal$;
Output: sol;

```
1:  sol ← φ
2:  for i = |pg| to 1 do
3:      tempSrv ← φ
4:      for each g in goal do
5:          if g.getOptimalService ≠ φ then
6:              tempSrv ← tempSrv ∪ g.getOptimalService
7:          end if
8:      end for
9:      sol ← sol ∪ tempSrv
10:     for each service srv in tempSrv do
11:         goal ← goal ∪ srv_in
12:         goal ← goal \ srv_out
13:     end for
14:     goal ← goal \ request
15:     if goal = φ then
16:         break;
17:     end if
18: end for
19: return sol
```

Traditional exhaustive algorithms search all possible combinations of candidate services in each action layer. As a result, for a layer of n services, it loops $2^n - 1$ times to find an optimal solution. While keeping a track of services which provide best QoS values for the goal concepts, there is no need to check all possible combinations and thus reduces searching time.

Example 6. *Table 2.3 shows a set of available services with their input and output parameters. The composition problem is* $C_{\text{in}} = \{A, B, C\}$ *and* $C_{\text{out}} = \{F\}$*. According to algorithm 15* ***ForwardExpand****, a planning graph is constructed and shown as figure 2.3. Please note that, the planning graph reaches the goal at layer 2, and reaches the fixed point at layer 3. If we use an exhaustive search algorithm, two solutions are found for the composition problem:* $w_2 \rightarrow w_4$ *and* $w_1 \rightarrow w_3 \rightarrow w_4$*.*

4.5 Empirical Results

We use TestsetGenerator2009 [79] to generate five data sets and evaluate our work. We generate random queries over the dataset as user requests and search for solutions with optimal QoS value.

We compare the performance of FSIDB method with the Graphplan method. In the worst situation, the Graphplan method can be six times faster to find a solution than the FSIDB method. Table 4.2 (resp. Table 4.3) shows the results over solutions with optimal response time (resp. throughput). The row "#services" shows the number of services in the returned solution. Table 4.2 and Table 4.3 show that, compared with the Graphplan approach, the FSIDB approach may return a solution with a better QoS value or fewer web services. In real system, redundant services in a solution cost more and spend time. In the path query stage of FSIDB approach, paths with redundant services are removed because these paths always have worse QoS values. The partial pre-composing approach returns a solution in less than 1s.

Table 4.2: Experiment results for solutions with optimal response time

		Testset1	**Testset2**	**Testset3**	**Testset4**	**Testset5**
	service	1020	3026	5045	7028	9052
	concept	3100	9400	16000	22000	28000
FSIDB	**#services**	4	8	10	6	12
	response time[1]	760	1250	1080	600	1670
Graphplan	**#services**	4	9	12	10	15
	response time[1]	760	1270	1330	1010	1980

[1] response time (ms) as a QoS metric

Table 4.3: Experiment results for solutions with optimal throughput

		Testset1	**Testset2**	**Testset3**	**Testset4**	**Testset5**
	service	1020	3026	5045	7028	9052
	concept	3100	9400	16000	22000	28000
FSIDB	**#services**	4	8	10	6	12
	throughput[1]	3000	5000	2000	6000	4000
Graphplan	**#services**	4	9	10	10	15
	throughput[1]	3000	4000	2000	5000	2000

[1] throughput (invocations per minute) as a QoS metric

In the partial pre-composing approach, we firstly pick 1000 queries from dataset 1 and find paths with either minimum response time or maximum throughput. We count how many times each service appears in these paths. Then, in each round, we delete services from "service_repository" which are least used by these paths. If the solution of a query is deleted due to services removed from "service_repository", we search the "path" table for alternative solutions. If no such solution exists, this query is removed. Figures 4.3 (resp. Figure 4.4) shows the relationship between services and queries while fetching paths with minimum response time (resp. throughput). When the number

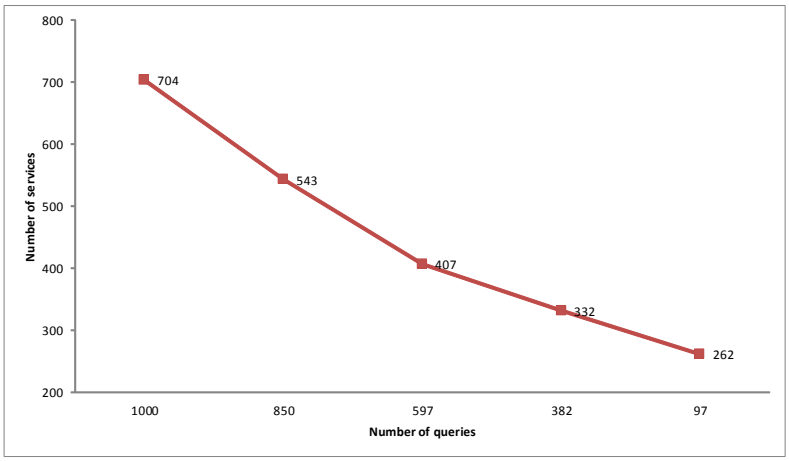


Figure 4.3: Remaining queries with optimal response time.

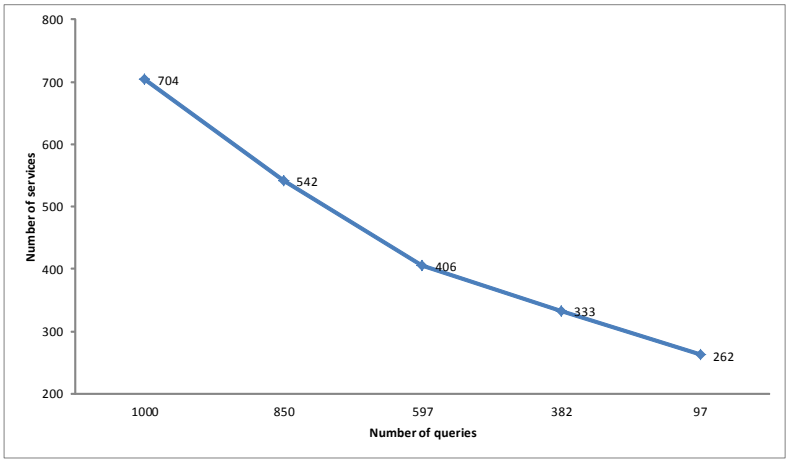


Figure 4.4: Remaining queries with optimal throughput.

of available services decreases, the number of resolvable queries decreases accordingly.

4.6 Summary

Taking advantage of skyline operator, we apply skyline analysis in solving QoS-aware service composition problem. Skyline analysis prunes less competitive services and reduces computational space requirement. We implement both FSIDB approach and Graphplan approach to compare their performance. In the Graphplan approach, a forward expand stage constructs a search graph and a backward search stage retrieves a solution. In the FSIDB approach, we pre-compute service combinations as paths and store them in a relational database. Compared with the Graphplan approach, the solution returned by FSIDB approach may contain fewer services with a better QoS value, but

more time is needed. The Graphplan approach is not global optimal and does not guarantee the solution has the best QoS value. The FSIDB approach finds all possible services connections and in the last step of query, paths are ranked according to their QoS values. As a result, paths with worse QoS values are pruned. We also propose a partial pre-composing approach which chooses popular paths generated by FSIDB approach and stores them in a separate table. When a user request comes, we first search this table to see whether we can find a nearly ready-made solution or not. Supposing the problem can be answered by these chosen paths, there is no need to search the table with whole paths. In future work, we will explore the extension of our work for multiple QoS criteria.

Chapter 5

Efficient Compressed Graph Representation for Service Composition

5.1 Introduction

In this chapter, we find a solution for QoS-aware service composition problem with a compact graph representation. Providers who expect to benefit from the SOA bring a large number of web services with similar functionalities. This poses a pressing challenge to service discovery and composition. To deal with the problem that the search space may encounter, we apply compressed graph representation to find a solution.

In a Web graph, each URL has a node, a hyperlink between two pages is represented as a directed arc. Compact graph stores a Web graph in memory with limited space, so it is easier for users to manipulate large graphs. Compressed graph representation aims at allowing graph algorithms running in in-memory over larger graphs. Data compressing has been widely studied [86, 87, 88, 89, 90].

The Connectivity Server [86] and Link Database [87] are among the first attempts to tackle the compression problem. The Connectivity Server uses lexicographical ordering and delivers linkage information for pages retrieved and indexed by the AltaVista search engine [86]. The Link Database encodes similar adjacency lists with reference compression methods [87]. As there are clusters of pages with similar adjacency lists, the Link Database studies this similarity and achieves space

savings. Properties of the web graph are extensively studied and applied in compression methods [87, 88]. Among them, a notable work is the WebGraph Framework which allows fast extraction of neighbors of a given page [88]. In their method, the graph is represented by a sequence of adjacency lists, each list is described as a sequence of numbers, then, they encode the numbers with self-delimiting bit-encoding method. Brisaboa *et al.* use a compact tree structure to represent the adjacency matrix of the Web graph [89]. Their method takes advantage of the sparseness and clustering of the adjacency matrix. We apply this data structure to represent the service composition problem.

In this chapter, we propose a novel service composition method in which a compact tree is applied to represent the search graph. The main contributions presented in this chapter can be summarized as follows:

- We propose a novel service composition method in which a compact tree is applied to represent the search graph, this tree representation takes advantage of the sparseness of the graph, and thus saves storage space.
- The tree representation allows fast navigation without decompression.
- We address the quality of services and calculate the overall QoS values of web service compositions.

The rest of this chapter is organized as follows. Section 5.2 shows how to represent a service composition problem with compact K^2-trees. The overall architecture and algorithms are given in Section 5.3. We give experiment results in Section 5.4 and draw the conclusion in Section 5.5

5.2 Compact Graph Representation

A web service composition problem can be represented by a directed graph where services and parameters are seen as nodes and their relationships are represented as directed edges. Figure 5.1 is a simple example of a web service composition graph, in which, we have two services and four parameters. We use circles to represent parameters and rectangles to represent services. If a parameter p is an input or output of a service w, there is an edge between p and w, we use arrows to represent

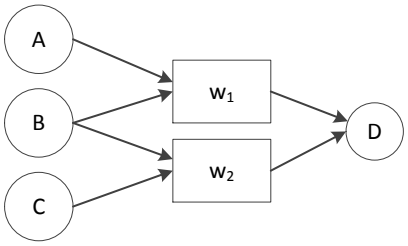


Figure 5.1: A web service composition representation.

Table 5.1: An adjacency matrix

	w_1	w_2
A	1	0
B	1	1
C	2	1
D	0	2

Table 5.2: Extended matrix of Table 5.1

	w_1	w_2	X	X
A	1	0	0	0
B	1	1	0	0
C	2	1	0	0
D	0	2	0	0

edges. For example, in this figure, A is an input parameter of w and w has an output parameter D. No cycle or parallel edges exist in our composition graph.

We use an adjacency matrix to represent the relationships between services and parameters in a web service network. In the adjacency matrix, each row represents a parameter and each column represents a service. The cell $\mathrm{adjMatrix}[i][j]$ represents the edge between node i and j. We have $\mathrm{adjMatrix}[i][j]$ = 1 if parameter i is an input parameter of service j (j is an output service of parameter i), $\mathrm{adjMatrix}[i][j]$ = 2 if parameter i is an output parameter of service j (j is an input service of parameter i), otherwise, $\mathrm{adjMatrix}[i][j]$ = 0. Figure 5.1 can be represented by the adjacency matrix shown in Table 5.1. For example, the edge from A to w_1 is represented as $\mathrm{adjMatrix}[1][1] = 1$, the edge from service w_1 to D is represented as $\mathrm{adjMatrix}[4][1] = 2$.

In real scenario, there can be thousands of services, but each service is only expected to have, at most, dozens of input and output concepts. Thus the adjacency is necessarily sparse. We use a compressed data structure to efficiently represent such a sparse matrix. For this purpose, let us briefly present the key ideas from Brisaboa *et al.* [91]. They rely on the notion of K-ary tree.

Definition 16. *A K-ary tree is a rooted tree in which each node has no more than k children.*

Definition 17. *A full K-ary tree is a K-ary tree where in each level every node has either 0 or k children.*

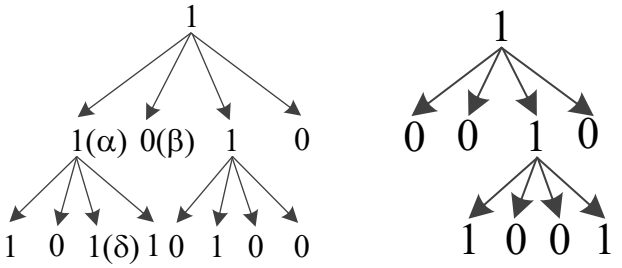
In this chapter, we present two methods for the composition problem. The first method, which is referred to as **two trees method**, builds two compact K^2-trees to represent the adjacency matrix of services and parameters. One for edges from input parameters to services (*inTree*), the other for edges from services to output parameters (*outTree*). The second method, which is referred to as **combined tree method**, combines *(*inTree) and *(*outTree) and represents the adjacency matrix with only one K^2-tree. When a user query comes, both methods check the compact K^2-tree, and return a composition solution or fail if no solution exists.

Now we use *inTree* as an example to show how to build the compact K^2-tree. In this process, matrix cells with value 2 are treated as 0 because they only matter with *outTree* building. Assume temporarily that the adjacency matrix is square and its size is a power of k. The whole matrix is represented as the root of tree values 1. Then, the matrix is divided from left to right, up to bottom into k^2 distinct sub-matrices, each with size n^2/k^2. Each sub-matrix is a child of the root node, so the root has exactly k^2 children. The child node is a leaf node with value 0 if the corresponding sub-matrix is filled with all zeroes, otherwise, it is an internal node with value 1. For each internal node, we recursively divide the sub-matrix it represents into k^2 sub-matrices. This process ends when the size of the sub-matrix is 1 or the value of node is 0. As sub-matrices filled with zeroes can be stored as a single value (0), the K^2-tree representation guarantees good compression. If the size of the matrix is not a power of k, we first extend the matrix to right and bottom, filling new cells with 0. Thus, the extended matrix is a square matrix and the length n' is a power of k. This will not bring significant changes because large area of 0 can be stored as a single value (0).

The *ourTree* building process is similar to that of *inTree*. In this process, the matrix cells with value 1 are treated as 0. For each child node in the tree, it is an internal node with value 1 if the corresponding sub-matrix contains 2. Otherwise, it is a leaf node with value 0.

Supposing we take $k = 2$, to represent the matrix (Table 5.1) with compact K^2-trees, first, we extend the matrix to right and bottom with 0 so the length of new matrix is a power of 2. The extended new matrix is shown in Table 5.2.

Then, we use two compact K^2-trees to represent the above extended matrix. One for edges from input parameters to services (*inTree*), the other for edges from services to output parameters (*outTree*). These two trees are shown in figure 5.2.



(a) inTree (b) outTree

Figure 5.2: K^2-tree representation of Table 5.2.

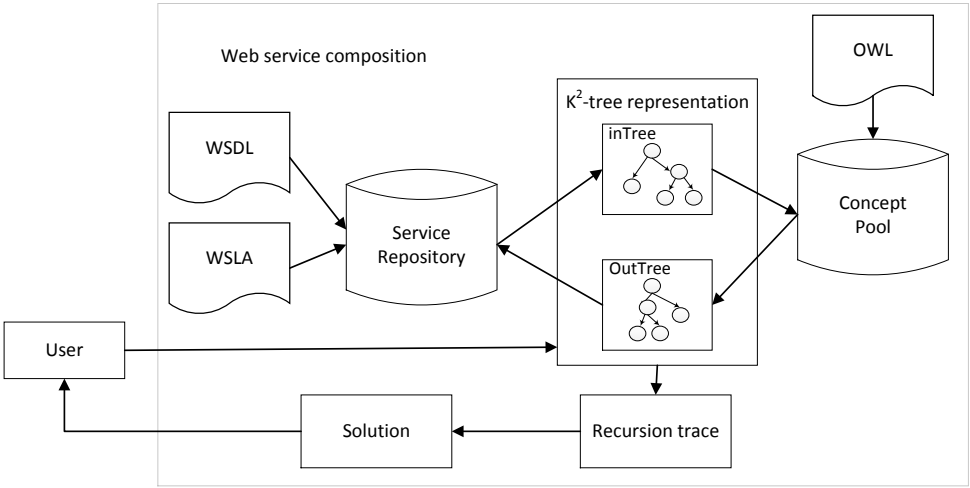


Figure 5.3: Architectural Overview.

5.3 Architectural Overview

In this section, we present the framework of proposed system and algorithms. Figure 5.3 shows an architecture overview of **two trees method**.

Preparation. Service providers publish web services with WSDL interfaces. The quality of services are published in WSLA files. Services' information is extracted and stored in the "Service Repository". Meanwhile, we obtain concepts and their tree relationships from the OWL file. These concepts are stored in a "Concept Pool". We build compact K^2-trees to represent the adjacency matrix of concepts and services.

User Query. When a user query comes with the initial states and expected goal, a forward propagation process is executed to check the K^2-trees. If a solution exists in the compact representation,

a backtracking algorithm is called to return a satisfying solution.

5.3.1 Two Trees Method

In this method, we use two compact K^2-trees to represent the adjacency matrix. One for edges from input parameters to services, the other for edges from services to output parameters.

Algorithm 18 ***TreeBuild*** builds K^2-tree to represent the adjacency matrix $matrix$. We first create a root node with value 1 and put it in a linked list pos. This linked list stores nodes waiting to be added in the tree. Then, we extend the tree level by level by adding k^2 children of each node in pos, if the child node values 1, put the child in pos. This algorithm stops when the size of sub-matrix is 1.

Theorem 1. *To represent the adjacency matrix, the space required for the compact tree is:* $k^2(\lceil \log_k n \rceil (e_{\text{in}} + e_{\text{out}}) - (e_{\text{in}} \log_{k^2} e_{\text{in}} + e_{\text{out}} \log_{k^2} e_{\text{out}}))$.

Proof. Supposing we have n concepts, e_{in} edges from input concept to services, and e_{out} output concepts from services. We review the worst possibility of inTree. The height of the tree is $h = \lceil \log_k n \rceil$, if $e_{\text{in}} = 1$, the space required is $k^2 \lceil \log_k n \rceil$; if $e_{\text{in}} = 2$, and in the worst case, a new internal node and its corresponding children are created, the space required is $k^2 \lceil \log_k n \rceil + k^2(\lceil \log_k n \rceil - 1)$, thus, at level l, the total space required is $k^2 e_{\text{in}}(\lceil \log_k n \rceil - l)$. There are at most $(k^2)^l$ nodes, and the internal nodes of the first $l-1$ levels are 1, thus $l = \log_{k^2} m$. The total space required for ***inTree*** is $k^2 e_{\text{in}}(\lceil \log_k n \rceil - \log_{k^2} e_{\text{in}})$.

Similarly, the total space for outTree is $k^2 e_{\text{out}}(\lceil \log_k n \rceil - \log_{k^2} e_{\text{out}})$. The total space required is $k^2(\lceil \log_k n \rceil (e_{\text{in}} + e_{\text{out}}) - (e_{\text{in}} \log_{k^2} e_{\text{in}} + e_{\text{out}} \log_{k^2} e_{\text{out}})$. □

After the compact K^2-tree is built, we use two arrays $ParentList$ and $LeafList$ to represent the compressed tree by a level order traversal. $ParentList$ stores values of nodes except the last level. $LeafList$ stores values of nodes in the last level. For example, to represent ***inTree*** of Figure 5.2, we have $ParentList$ = 1010, $LeafList$ = 10110100.

Algorithm 19 ***CalRow*** calculates the starting row of a sub-matrix (node $tNode$ in the K^2-tree), we recursively calculate the row offset of the sub-matrix to its parent matrix.

Algorithm 18 *TreeBuild*

Input: $matrix$: the adjacency matrix;
Output: $tree$: the compact K^2-tree;

create a root with value 1 for the tree;
pos $\leftarrow root$;
tempos $\leftarrow \phi$
for $length \leftarrow n/k; length \geq 1; length \leftarrow \frac{length}{k}$ **do**
  while pos $\neq \phi$ **do**
    $parent \leftarrow$ pos.$remove$(pos.$first$)
    if $parent = 1$ **then**
      $startRow \leftarrow calRow(tree, parent, n)$
      $startColumn \leftarrow calColumn(tree, parent, n)$
      create k^2 children nodes $child$ for parent, the sub-matrix has size $length^2$, the offset of the sub-matrix relative to the $matrix$ is represented by $startRow, startColumn$ and $length$
      if $child = 1$ **then**
        tempos $\leftarrow$ tempos $\cup$ $child$
      end if
    end if
  end while
  repeat
    pos $\leftarrow$ pos $\cup$ tempos.$remove$(tempos.$first$)
  until tempos $= \phi$
end for
return $tree$

Supposing the length of the matrix is n, the length of sub-matrix in depth d is

$$l = n/k^{d-1}. \tag{16}$$

If $tNode$ is the first k children of its parent, the row offset to its parent matrix is 0, otherwise, the row offset is calculated as

$$offset = l/k. \tag{17}$$

Combine Equation (16) and Equation (17), we get

$$offset = n/k^{d}. \tag{18}$$

Algorithm 19 *CalRow*

Input: $tree, tNode, n$;
Output: $startRow$: the offset of sub-matrix;

```
1:  startRow ← 0
2:  temp ← tNode
3:  while temp ≠ root do
4:     if temp is the first k children of parent then
5:        startRow ← startRow
6:     else
7:        set ← tree.depth(temp)
8:        startRow ← startRow + n/k^set
9:     end if
10:    temp ← tree.parent(temp)
11: end while
12: return startRow
```

Example 7. *We want to find the row of δ (its depth is 2) in figure 5.2. This node is not the first or the second child of its parent, according to equation 18, $startRow = 1$ (Algorithm 19 **CalRow** line 8). In the second round, α is δ's parent, as it is the first child of its parent, the offset between α and its parent is 0, so $startRow = 1$. In the third round, $temp = root$, the algorithm stops. The row of δ is 1.*

Algorithm 20 ***CalColumn*** calculates the starting column of the sub-matrix, the column offset of the sub-matrix (node $tNode$) is calculated according to Equation (18).

Algorithm 20 *CalColumn*

Input: $tree, tNode, n$
Output: $startColumn$: the offset of sub-matrix;

```
1: startColumn ← 0
2: temp ← tNode
3: while temp ≠ root do
4:    if temp is an odd child of parent then
5:       startColumn ← startColumn
6:    else
7:       set ← tree.depth(temp)
8:       startColumn ← startColumn + n/k^set
9:    end if
10:   temp ← tree.parent(temp)
11: end while
12: return startCol
```

To find the output services of a given parameter p, we need to locate cells which contain 1 in row p and obtain their column numbers. This is done by a level order traversal of *inTree*. Similarly, to obtain the input services of a given parameter p, we need to find cells contain 2 in row p of the matrix, which is done by a traversal of *outTree*. As each internal node has k^2 children, we need to check k children nodes in each iteration. We call the related columns of given rows as direct nodes. Similarly, the related rows of given columns are called as reverse nodes.

Example 8. *We want to find the column of δ in inTree of Figure 5.2. This node is the third child of its parent, according to equation 18, $startColumn = 0$ (Algorithm 20 **CalColumn** line 5). In the second round, α is δ's parent, as it is the first child of root, the offset is 0, $startColumn = 0$. In the third round, $temp = root$, the algorithm stops. The column of δ is 0.*

Algorithm 21 ***Direct*** describes how to retrieve direct nodes. In this algorithm, P represents $ParentList$ and L represents $LeafList$, $rank(P, i)$ counts how many 1 appear in P until position i, $rank(P, -1) = 0$. If $P[x] = 1$, the children of x is at position $rank(P, x) \times k^2$ to $rank(P, x) \times k^2 + (k^2 - 1)$ [89]. $size$ denotes the length of current sub-matrix, row and col represent the row and column of current sub-matrix. $index$ represents the position in P and L. If $index$ moves to L and values 1, this means there is an edge between row and col, we add the current column col into the solution (lines 2–4). If the size of sub-matrix is not 0, and this is the first round ($index = -1$) or the node value is 1, which means the sub-matrix is not all 0, we check the children nodes of the

current node (lines 10–11).

Algorithm 21 *Direct*

Input: $P, L, size, row, col, index$
Output: $colSol$:a set of services;

```
1:  temp ← 0
2:  if index ≥ P.length then
3:      if L[index − P.length] = 1 then
4:          colSol ← colSol ∪ col
5:      end if
6:  else if size ≥ 1 then
7:      if index = −1||P[index] = 1 then
8:          temp ← k^2 × rank(P, index) + k × ⌊k × row/size⌋
9:      end if
10:     for j ← 0; j < k; j + + do
11:         direct(P, L, size/k, row%(size/k), col + j × (size/k), temp + j)
12:     end for
13: end if
14: return colSol
```

Example 9. *we want to find the output services of parameter A in Table 5.2. We need to find cells containing 1 in row 1 by using algorithm 21* ***Direct****. We search the inTree and the search process is shown in figure 5.4, the output refers to service w_1 in column 0.*

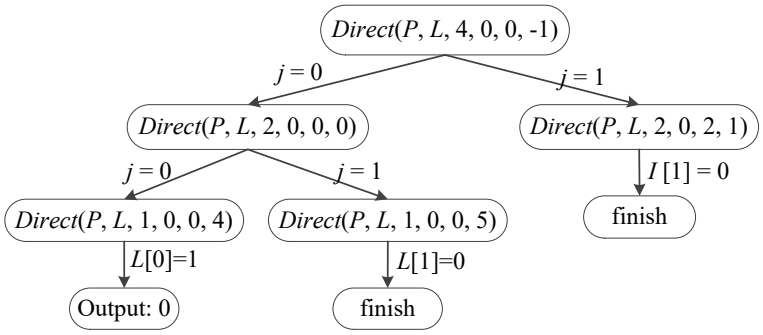


Figure 5.4: Finding output service of A with two trees method.

Algorithm 22 ***Reverse*** describes how to retrieve reverse nodes. If $index$ moves to array $LeafList$ and values 1, this means there is an edge between row and col, add the current row row into the solution (lines 2–4). If the size of sub-matrix is not 0, and this is the first round or the node value is 1 (line 7), check the children nodes of the current node (lines 10–11).

Algorithm 22 *Reverse*

Input: $P, L, size, col, row, index$
Output: $rowSol$:a set of parameters;

```
1: temp ← 0
2: if index ≥ P.length then
3:    if L[index − P.length] = 1 then
4:       rowSol ← rowSol ∪ row
5:    end if
6: else if size ≥ 1 then
7:    if index = −1||P[index] = 1 then
8:       temp ← k^2 × rank(P, index) + ⌊k × col/size⌋
9:    end if
10:   for j ← 0; j < k; j + + do
11:      reverse(P, L, size/k, col%(size/k), row + j × (size/k), temp + k × j)
12:   end for
13: end if
14: return rowSol
```

We use algorithm 22 ***Reverse*** to find the input parameters of a give service ws by a level order traversal of *inTree* and the output parameters of ws in *outTree*. For the first case, we need to locate the 1 in column ws of the matrix and find their corresponding row numbers. For the second case, we need to locate the 2 in column ws of the matrix and find the row numbers.

Example 10. *we want to find the output parameters of service w_1 in Table 5.2. We need to find cells containing 2 in column 1 by using algorithm 22* ***Reverse****. We search the outTree and the search process is shown in figure 5.5, the output refers to parameter C in column 3.*

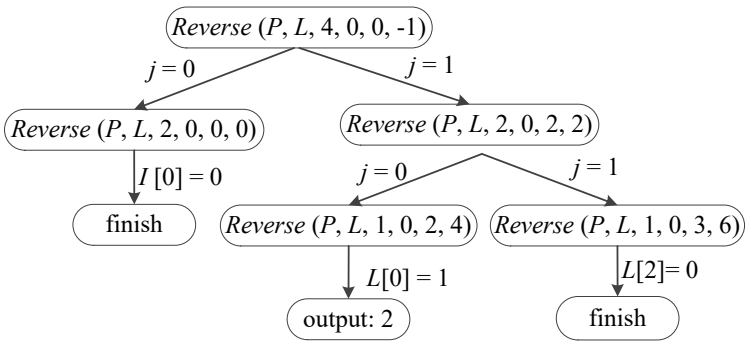


Figure 5.5: Finding output parameter of w_1 with two trees method.

Algorithm 23 ***ForwardPropagation*** finds whether or not the composition problem can be solved. In this algorithm, $ConceptPool$ stores all the evocable concepts. We use $inTreeP$ (resp. $inTreeL$)

to represent $ParentList$ (resp. $LeafList$) of $inTree$, similarly, $outTreeP$ (resp. $outTreeL$) represents $ParentList$ (resp. $LeafList$) of $outTree$. In each iteration, for each newly added concept, we find its output services by calling algorithm 21 (line 5), for each service found by algorithm 21 **Direct**, we use algorithm 22 **Reverse** to find all its input concepts (line 6), if all its input concepts exist in the $conceptPool$, this service can be invoked and added in the $addedService$ set (line 7). We add the output concepts of services in $addedService$ into the $conceptPool$ (line 15). To carry out a global optimization, we record the QoS values of services and corresponding concepts in the process of forward expand. For each service s, the total QoS value along the way will be calculated and represented as $s.curQoS$, for each concept c, the optimal QoS value is represented as $c.optQoS$, the input service who provides the best QoS value is represented as $c.optSrv$. This algorithm ends when no more concepts can be added into the $conceptPool$. If the goals are contained in $conceptPool$, the algorithm returns true, otherwise, it returns false.

Theorem 2. *Consider a service composition problem with n concepts, m services, e_{in} edges from concepts to services and e_{out} edges from services to concepts, the time spent in the search algorithm is polynomial in n, m, e_{in} and e_{out}.*

Proof. The navigation time for finding direct and reverse edges of a given node in the inTree (resp. outTree) is $O(\sqrt{e_{\text{in}}})$ (resp. $O(\sqrt{e_{\text{out}}})$) [92]. Support in round i, the number of newly added concept is C_{add}, and newly added service is S_{add}, the time spent in round i is $C_{\text{add}}\sqrt{e_{\text{in}}} + S_{\text{add}}(\sqrt{e_{\text{in}}} + \sqrt{e_{\text{out}}})$. As the total number of applied concepts and services are the total number of applied concepts and services are fixed and no new concept or service will be created in the algorithm, the algorithm will end when the goal is reached or there are no more concepts to be added. Thus, the time spent is polynomial in the number of concepts n, the number of service m, e_{in} and e_{out}. □

Algorithm 24 ***Backtracking*** finds a suitable solution for the composition problem. In each iteration, we add services who provide optimal QoS values for goals as solution services (line 3), then we search and add input parameters of these service to $goal$ (line 6). This algorithm stops when the goal is a subset of the initial states.

Algorithm 23 *ForwardPropagation*

Input: $initial, goal$
Output: Boolean:

1: $conceptPool \leftarrow initial$
2: $addedConcept \leftarrow initial$
3: **while** $addedConcept \neq \varphi$ **do**
4: **for** each concept c in $addedConcept$ **do**
5: $tempSrv \leftarrow Direct(\text{inTreeP}, \text{inTreeL}, n, c, 0, -1)$
6: **if** $(Reverse(\text{inTreeP}, \text{inTreeL}, n, tempSrv, 0, -1) \setminus conceptPool) = \varphi$ **then**
7: $addedService \leftarrow adddedService \cup tempSrv$
8: **end if**
9: **for** each service s in $addedService$ **do**
10: $s.curQoS \leftarrow (c.optQoS + s.QoS)$
11: **end for**
12: **end for**
13: $addedConcept \leftarrow \varphi$
14: **for** each service s in $addedService$ **do**
15: $addedConcept \leftarrow addedConcept \cup Reverse(\text{outTreeP}, \text{outTreeL}, n, s, 0, -1)$
16: **for** each concept c in $addedConcept$ **do**
17: **if** $c.optQoS < s.curQoS$ **then**
18: $c.optQoS \leftarrow s.curQoS$
19: $c.optSrv \leftarrow s$
20: **end if**
21: **end for**
22: **end for**
23: $conceptPool \leftarrow conceptPool \cup addedConcept$
24: $addedService \leftarrow \varphi$
25: **end while**
26: **if** $goal \subseteq conceptPool$ **then**
27: **return** true
28: **else**
29: **return** false
30: **end if**

Algorithm 24 *Backtracking*

Input: $initial, goal$
Output: $solution$: a set of solution services;

```
1: while (goal \ initial) ≠ φ do
2:    for each concept c in the goal do
3:       cover ← cover ∪ c.optSrv
4:    end for
5:    for each service s in cover do
6:       goal ←
         goal ∪ Reverse(inTreeP, inTreeL, n, s, 0, −1)
7:       goal ←
         goal \ Reverse(outTreeP, outTreeL, n, s, 0, −1)
8:    end for
9:    solution ← solution ∪ cover
10:   cover ← φ
11: end while
12: return solution
```

5.3.2 Combined Tree Method

In this method, we represent the adjacency matrix with only one K^2-tree. The tree building process is similar with the $inTree$ and $outTree$ building process. The only difference is: for each node in the tree, its value is 3 if both 1 and 2 are contained in the corresponding sub-matrix; if there are only 1 and 0 exist in the corresponding sub-matrix, the value of this internal node is 1. If the sub-matrix contains only 2 and 0, the value of the internal node is 2. Otherwise, it is an external node with value 0. The rank algorithm in this method counts how many none 0 appear in $ParentList$ until position i, $rank(P, -1)$= 0. In figure 5.6, we use a compact K^2-tree to represent the extended matrix in Table 5.2. In this method, $ParentList$ stores values of nodes except the last level. $LeafList$ stores values of nodes in the last level. In this example, $ParentList$ = 1030, $LeafList$ = 10112102.

Algorithm 25 ***CombinedDirect*** describes how to retrieve direct nodes. We add a variable $flag$, if $flag$=1, we want to find output services of a given parameter; if $flag$=2, we want to find input services of a given parameter. If $index$ moved to array L and its value is the same with $flag$, add col into the solution set $colSol$ (line 2–4). If this is the first round ($index$ = -1), or the node value is 3 (both 1 and 2 are contained in the corresponding sub-matrix) or $flag$ (line 7), go on checking the children nodes (line 10–11).

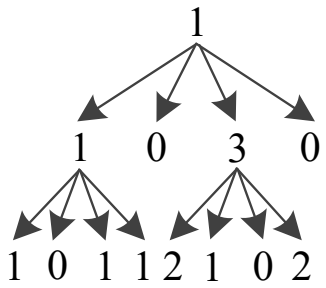


Figure 5.6: The combined tree representation.

Algorithm 25 *CombinedDirect*

Input: $P, L, size, row, col, index, flag$
Output: $colSol$:a set of services;

```
1: temp ← 0
2: if index ≥ P.length then
3:    if L[index − P.length] = flag then
4:       colSol ← colSol ∪ col
5:    end if
6: else if size ≥ 1 then
7:    if index = −1||P[index] = 3||P[index] = flag then
8:       temp ← k^2 × rank(P, index) + k × ⌊k × row/size⌋
9:    end if
10:   for j ← 0; j < k; j++ do
11:      direct(P, L, size/k, row%(size/k), col + j × (size/k), temp + j)
12:   end for
13: end if
14: return colSol
```

Algorithm 26 ***CombinedReverse*** describes how to retrieve reverse nodes. If $index$ moves to array $LeafList$ and its value equals $flag$, this means there is an edge between row and col, add the current row row into the solution set $rowSol$ (lines 2–4). If this is the first round or the node value is 3 or $flag$ (line 7), check the children nodes of the current node (lines 10–11).

Example 11. *we want to find the output services of parameter A in Table 5.2 by using the combined tree method. We find cells containing 1 in row 1 by using algorithm 25* ***CombinedDirect****. Figure 5.7 shows the new recursion trace in the combined tree, the output refers to service* w_1 *in column 0.*

Example 12. *we want to find the output parameter of* w_1 *by using the combined tree method. We find cells containing 2 in column 1 by using algorithm 25* ***CombinedReverse****. Figure 5.8 shows the new recursion trace in the combined tree, the output refers to parameter* C *in row 3.*

Algorithm 26 *CombinedReverse*

Input: $P, L, size, col, row, index, flag$
Output: $rowSol$:a set of parameters;

```
temp ← 0
if index ≥ P.length then
    if L[index − P.length] = flag then
        rowSol ← rowSol ∪ row
    end if
else if size ≥ 1 then
    if index = −1||P[index] = 3||P[index] = flag then
        temp ← k^2 × rank(P, index) + ⌊k × col/size⌋
    end if
    for j ← 0; j < k; j++ do
        reverse(P, L, size/k, col%(size/k), row + j × (size/k), temp + k × j)
    end for
end if
return rowSol
```

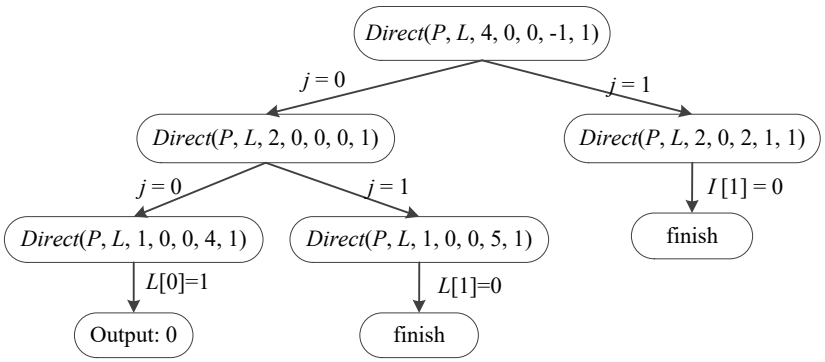


Figure 5.7: Finding output service of A with combined tree method.

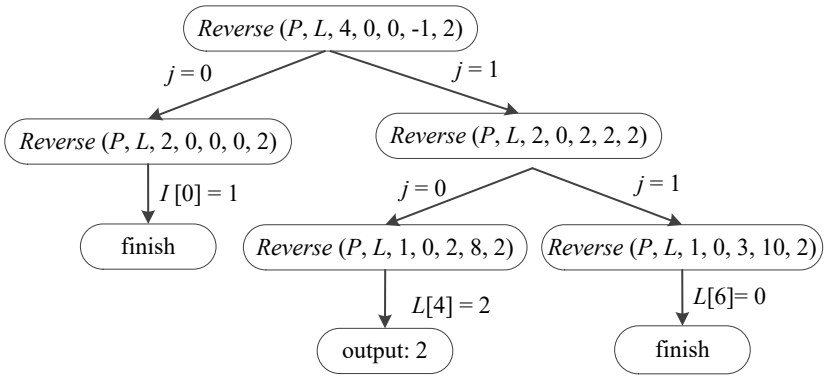


Figure 5.8: Finding output parameter of w_1 with combined tree method.

5.4 Empirical Results

Supposing response time and throughput have same weight, we use the following equation to calculate the overall QoS value of a web service composition solution (wsc):

$$U(wsc) = \frac{T(wsc) - T^{min}}{T^{max} - T^{min}} \times W_{thp} + \frac{l \times R^{max} - R(wsc)}{l \times (R^{max} - R^{min})} \times W_{resp} \quad (19)$$

There are l layers in the solution.

The experimental results are shown in Table 5.3. The service (resp. concept) row represents the number of services (resp. concepts) in the dataset. #service row represents the number of services in the returned solution. The build-time is the time spent on building the compact tree and the search graph, the search-time represents time spent to retrieve a solution. The compression rate is the ratio between the number of cells after compression and the original number of matrix cells used in the adjacency matrix:

$$compression\ rate = \frac{compact\ structure}{services \times concepts}.$$

Table 5.4 and Table 5.5 show the QoS values of solutions with optimal response time, throughput or the utility value which is calculated according to Equation (19).

Figure 5.9 and figure 5.10 show the percentage of $ParentList$ and $LeafList$ in the compressed data structure. The proportion of $ParentList$ increases when the number of services increases. When the number of services is fixed, the proportion of $ParentList$ decreases when k is larger. Because the corresponding tree becomes shorter and wider when k is larger, the proportion of $LeafList$ is bigger in the combined tree method compared with the two trees method.

Figure 5.11 shows the comparison of compression rate, we observe that, this compact representation method has a good compression ratio, which helps reduce the storage requirement. When k is fixed, the compression rate decreases as the number of services increases, this shows our methods have good scalibility. For a same testset, the compression ratio is worse when k is greater, because more 0 are stored in $ParentList$ and $LeafList$ and the proportion of $LeafList$ in compressed

Table 5.3: Experimental results

			Testset1	Testset2	Testset3	Testset4	Testset5
		service	1020	3026	5045	7028	9052
		concept	3100	9400	16000	22000	28000
		#service	10	13	13	13	21
Two trees	**k=2**	**matrix length**	2^{12}	2^{14}	2^{14}	2^{15}	2^{15}
		build (s)	3.5	42	127	206.3	485.7
		search (s)	0.3	3.7	9.6	20.9	42
	k=3	**matrix length**	3^8	3^9	3^9	3^8	3^{10}
		build (s)	2.2	22.5	78	128	398.8
		search (s)	0.2	2.3	5.8	14.7	36.2
	k=4	**matrix length**	4^6	4^7	4^7	4^8	4^8
		build (s)	1.4	15.8	51.5	104.4	323.7
		search (s)	0.1	1.5	4.3	9.6	26
Combined tree	**k=2**	**build** (s)	6.3	83.8	299.5	441	1811.8
		search (s)	0.4	5.7	14.9	32.6	86.4
	k=3	**build** (s)	3.6	42.3	192.2	241.5	797.1
		search (s)	0.3	3.4	9.5	21.8	54.6
	k=4	**build** (s)	2.2	27	114.5	181	602.4
		search (s)	0.2	2.3	6.4	15	38.4

Table 5.4: Results for solution with optimal response time

	Testset1	Testset2	Testset3	Testset4	Testset5
R[1]	2310	1910	2560	2730	4030
T[2]	2000	1000	1000	1000	1000
U[3]	0.3	0.36	0.34	0.3	0.31

[1] R: response time (ms) as a QoS metric
[2] T: throughput (invocations per minute) as a QoS metric
[3] U: the overall utility value

representation increases. The combined-tree method can provide better compression rate compared with the two-trees method.

Figure 5.12 shows the test results of search time. From Algorithm 21 **Direct** and Algorithm 22 **Reverse**, we conclude that the search time is related to the height of the tree. The K^2-tree is shorter and wider when k is greater, as a result, the search time is shorter. We observe that, with the increase of services, the search time increases slower when k becomes bigger. Although the combine tree method builds only a tree to represent the adjacency matrix, this method takes almost 1.5 times search time compared with two trees method. This is because, when we combine *inTree* and *outTree* to one tree, the tree becomes bigger and more complicated, so it takes longer time to return a solution with the combined tree method.

Table 5.5: Results for solution with optimal throughput

	Testset1	Testset2	Testset3	Testset4	Testset5
R[1]	2310	2970	4070	2730	5530
T[2]	2000	4000	9000	1000	2000
U[3]	0.3	0.36	0.44	0.3	0.29

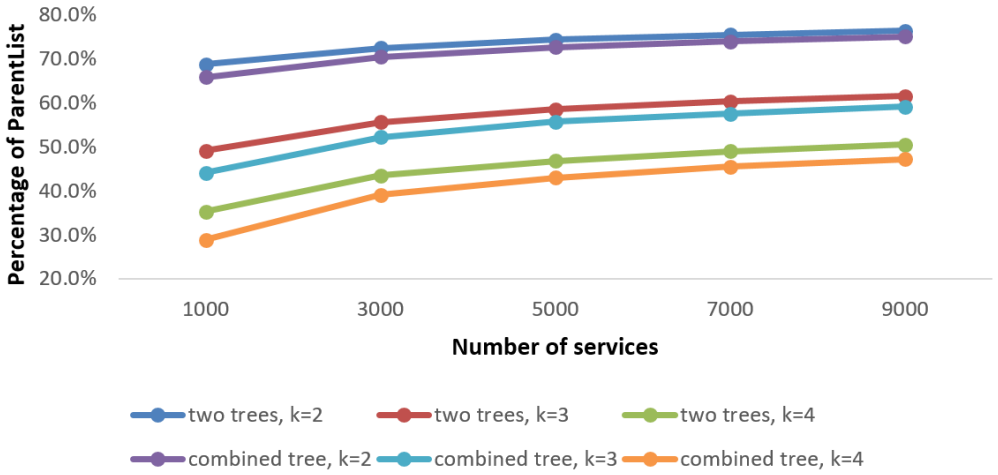


Figure 5.9: Percentage of $ParentList$.

5.5 Summary

In this chapter, we address web service composition challenges and present two compact methods. In the **two trees method**, we use two compact K^2-trees to represent the adjacency matrix of services and concepts: one for services and input concepts, the other for services and output concepts. In the **combined tree method**, we combine the two compact trees and represent the adjacency matrix with only one K^2-tree. When a user request comes, both methods search the K^2-tree for a satisfactory solution. Decompression is unnecessary in the search process. In summary, the main goal and contribution of this chapter is to find a solution for the composition problem with limited storage requirement. As we can notice, the proposed algorithms may run on large data sets with lesser space requirements.

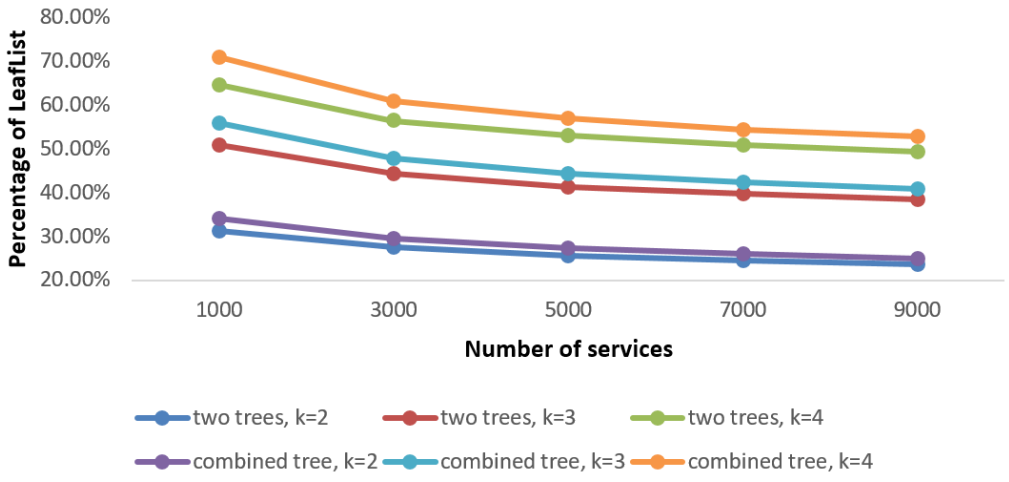


Figure 5.10: Percentage of $LeafList$.

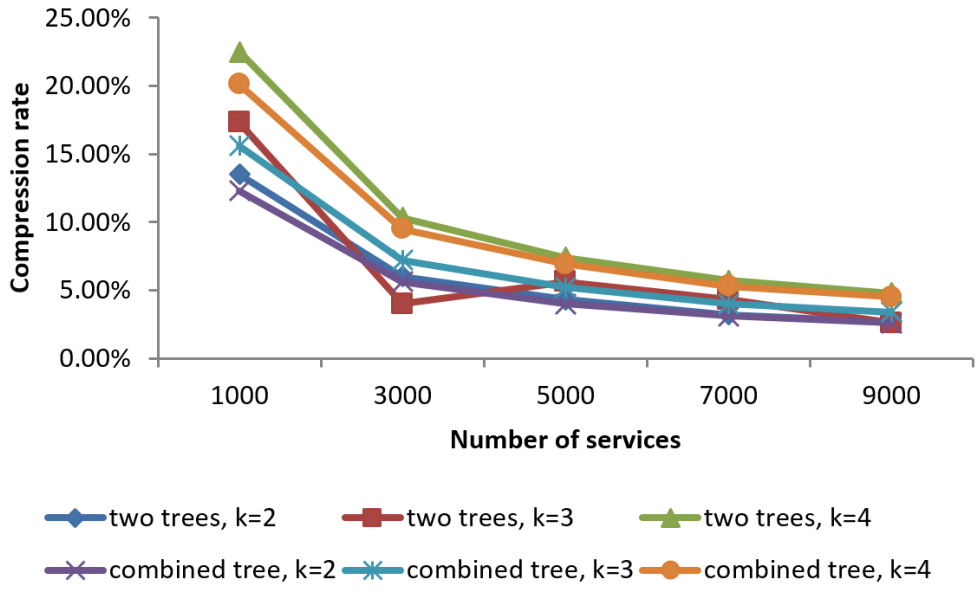


Figure 5.11: Comparison of compression rate.

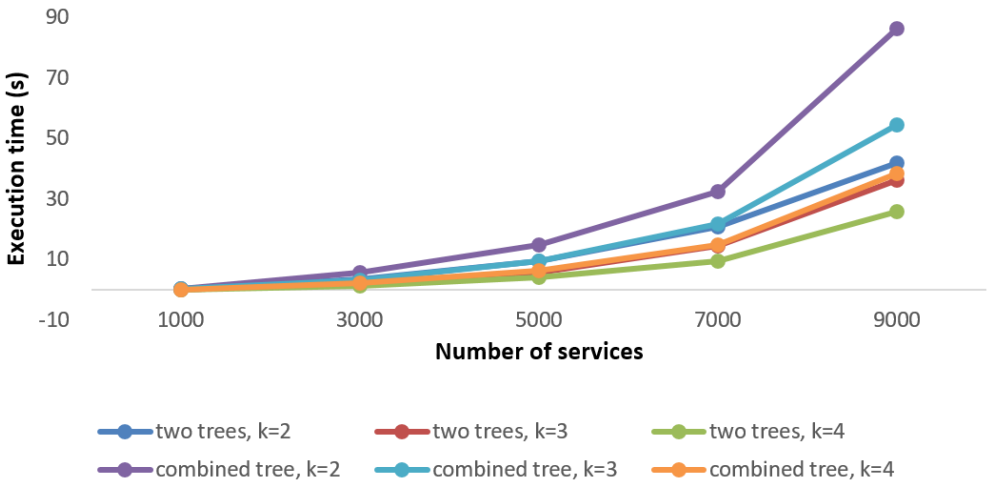


Figure 5.12: Comparison of search time.

Chapter 6

Conclusion

6.1 Summary

In this thesis, we address web service composition problem that achieves both functional and non-functional requirements. As the huge storage required in searching a composition solution may be infeasible, we propose two methods to overcome this difficulty: by using a relational database or compressed data structure.

In the database-based approach, we propose a FSIDB method to retrieve top-K solutions. First, we generate all possible service combinations as paths and store them in a database. Then, when a user query comes, 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 top-K solutions as backup solutions in case of service disappearance or failure. Since services on the network always change, *e.g.,* new services being added to the network, old services fail to work or disappear, we discuss the scalability and update cost of our method. Compared with other database-based approaches, our system supports services with multiple input and output parameters, and parallel structures of services are supported by the paths. In this thesis, we also study redundant service problem. A service is redundant if all its outputs used by other services are also produced by other services. In real system, redundant services in a solution cost more and spend time. Paths with redundant services always have less competition because they may lead to a worse QoS value.

We are motivated to employ skyline analysis to prune less competitive services and reduce

computational space requirement. For each concept, we find a skyline service set among its parent services, services in this set do not get dominanted by each other in terms of response time and throughput. To find a composition solution, we consider skyline services and implement Graphplan and FSIDB method. The Graphplan approach is an in-memory approach containing two stages: a forward expand stage constructs a planning graph and a backward search stage retrieves a solution. Compared with FSIDB method, the Graphplan approach finds a solution in a short time, the FSIDB approach may take longer time to find a solution, but the solution returned by this approach always has fewer redundant services with a better QoS value. We also present a partial pre-composing approach which chooses popular paths generated by FSIDB approach. These paths are kept in a separate table of the database for fast delivery. When a user request comes, we first search in this separate table to see whether or not we can find a nearly ready-made solution. Only as a last resort do we search the table with whole paths to find a solution.

In the second approach, we use a compressed data structure to represent the search graph. We present two compact methods to find the solution. In the "two trees method", we use two compact K^2-trees to represent the adjacency matrix of web services and concepts: one for services and their input concepts, the other for services and corresponding output concepts. In the "combined tree method", we combine the two compact trees and represent the adjacency matrix with only one K^2-tree. When a user request comes, both methods search the K^2-tree for a satisfying solution. Decompression is unnecessary in our method. The main goal and contribution of this approach is to find a composition solution with limited storage requirement. As we can notice, this approach may run on large data sets with lesser space requirements.

6.2 Future work

In our future research, we plan to improve the scalability of our FSIDB method. If a query takes too much memory or time, we may return a non-optimal solution instead of an optimal one. Also, we will study cloud service composition problem and extend our work. We want to study cloud service composition problem to utilize the existing resources on the Internet. Yet, we need to overcome

several difficulties. First of all, compared with local environment, cloud computing has higher real-time requirements. Second, relational database is no longer suitable to cloud environment, we need to extend our proposed method to cloud environment. NoSQL database is suitable for using as a cloud data management system. Compared with relational database, an attractive feature of NoSQL database is: we may add records in NoSQL on the fly. We need to design a pattern so that our algorithms work in NoSQL database.

Bibliography

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

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

[3] U. Srivastava, K. Munagala, J. Widom, and R. Motwani, "Query optimization over web services," in *Proceedings of the 32Nd International Conference on Very Large Data Bases*, ser. VLDB '06. VLDB Endowment, 2006, pp. 355–366.

[4] 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.

[5] R. Chinnici, J.-J. Moreau, A. Ryman, and S. Weerawarana. (2007) Web Services Description Language (WSDL) version 2.0. W3c. [Online]. Available: http://www.w3.org/TR/wsdl20/

[6] M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, H. Nielsen, A. Karmarkar, and Y. Lafon. (2007) SOAP Version 1.2 Part 1: Messaging Frame-work (Second Edition). W3c. [Online]. Available: http://www.w3.org/TR/{SOA}p12-part1/

[7] OASIS (2007) UDDI version 2.04 API specification. [Online]. Available: http://uddi.org/pubs/ProgrammersAPI-V2.04-Published-20020719.htm

[8] 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.

[9] M. Aznag, M. Quafafou, and Z. Jarir, "Leveraging formal concept analysis with topic correlation for service clustering and discovery," in *Web Services (ICWS), 2014 IEEE International Conference on*, 2014, pp. 153–160.

[10] M. Alrifai, D. Skoutas, and T. Risse, "Selecting skyline services for QoS-based web service composition," in *Proceedings of the 19th International Conference on World Wide Web*, ser. WWW '10. New York, NY, USA: ACM, 2010, pp. 11–20.

[11] Q. Yu and A. Bouguettaya, "Computing service skylines over sets of services," in *Web Services (ICWS), 2010 IEEE International Conference on*, July 2010, pp. 481–488.

[12] D. Papadias, Y. Tao, G. Fu, and B. Seeger, "Progressive skyline computation in database systems," *ACM Trans. Database Syst.*, vol. 30, no. 1, pp. 41–82, 2005.

[13] M. Steinbach, L. Ertoz, and V. Kumar, "The challenges of clustering high dimensional data," in *New Directions in Statistical Physics*, L. Wille, Ed. Springer Berlin Heidelberg, 2004, pp. 273–309.

[14] A. Paliwal, B. Shafiq, J. Vaidya, H. Xiong, and N. Adam, "Semantics-based automated service discovery," *Services Computing, IEEE Transactions on*, vol. 5, no. 2, pp. 260–275, 2012.

[15] 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.

[16] Y. Yan, B. Xu, Z. Gu, and S. Luo, "A QoS-driven approach for semantic service composition," in *Commerce and Enterprise Computing, 2009. CEC '09. IEEE Conference on*, July 2009, pp. 523–526.

[17] Q. Fang, X. Peng, Q. Liu, and Y. Hu, "A global QoS optimizing web services selection algorithm based on moaco for dynamic web service composition," in *Information Technology and Applications, 2009. IFITA '09. International Forum on*, vol. 1, May 2009, pp. 37–42.

[18] L. Wang and Y. He, "A web service composition algorithm based on global QoS optimizing with MOCACO," in *Proceedings of the 2011, International Conference on Informatics, Cybernetics, and Computer Engineering (ICCE2011)*, vol. 111, November 2011, pp. 79–86.

[19] 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.

[20] 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.

[21] L. Zeng, B. Benatallah, M. Dumas, J. Kalagnanam, and Q. 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.

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

[23] 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.

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

[25] 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.

[26] (2009) Web service challenge rules. [Online]. Available: http://ws-challenge.georgetown.edu/wsc09/downloads/WSC2009Rules-1.1.pdf

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

[28] E. Al-Masri and Q. H. Mahmoud, "QoS-based discovery and ranking of web services," in *Computer Communications and Networks, 2007. ICCCN 2007. Proceedings of 16th International Conference on*, Aug 2007, pp. 529–534.

[29] E. A. Masri and Q. H. Mahmoud, "Discovering the best web service," in *Proceedings of the 16th International Conference on World Wide Web*. ACM, 2007, pp. 1257–1258.

[30] 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.

[31] 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.

[32] (2004) Web ontology language for web services. [Online]. Available: http://www.w3.org/submission/owl-s/

[33] 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 &Amp; Posters*, ser. WWW Alt. '04. New York, NY, USA: ACM, 2004, pp. 66–73.

[34] B. Ganter and R. Wille, "Applied lattice theory: Formal concept analysis," in *In General Lattice Theory, G. Grätzer editor, Birkhäuser*. Preprints, 1997.

[35] J. Wu, L. Chen, Z. Zheng, M. R. Lyu, and Z. Wu, "Clustering web services to facilitate service discovery," *Knowledge and Information Systems*, vol. 38, no. 1, pp. 207–229, 2013.

[36] W. Chen, I. Paik, and P. Hung, "Constructing a global social service network for better quality of web service discovery," *Services Computing, IEEE Transactions on*, vol. 8, no. 2, pp. 284–298, 2015.

[37] A. Dastjerdi and R. Buyya, "Compatibility-aware cloud service composition under fuzzy preferences of users," *Cloud Computing, IEEE Transactions on*, vol. 2, no. 1, pp. 1–13, 2014.

[38] M. Ghallab, D. Nau, and P. Traverso, *Automated planning: theory and practice*. Morgan Kaufmann Publishers, 2004.

[39] Y. Yan and X. Zheng, "A planning graph based algorithm for semantic web service composition," *2008 10th IEEE Conference on E-Commerce Technology and the Fifth IEEE Conference on Enterprise Computing, E-Commerce and E-Services*, vol. 0, pp. 339–342, 2008.

[40] A. L. Blum and M. L. Furst, "Fast planning through planning graph analysis," *Artificial Intelligence*, vol. 90, no. 1-2, pp. 281–300, 1997.

[41] Z. Huang, W. Jiang, S. Hu, and Z. Liu, "Effective pruning algorithm for QoS-aware service composition," in *Commerce and Enterprise Computing, 2009. CEC '09. IEEE Conference on*, July 2009, pp. 519–522.

[42] 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.

[43] 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.

[44] G. Canfora, M. Di Penta, R. Esposito, and M. Villani, "An approach for QoS-aware service composition based on genetic algorithms," in *Proceedings of the 2005 Conference on Genetic and Evolutionary Computation*, ser. GECCO '05. ACM, 2005, pp. 1069–1075.

[45] 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.

[46] 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.

[47] N. Ben Mabrouk, S. Beauche, E. Kuznetsova, N. Georgantas, and V. Issarny, "QoS-aware service composition in dynamic service oriented environments," in *Middleware 2009*, ser. Lecture Notes in Computer Science, J. Bacon and B. Cooper, Eds. Springer Berlin Heidelberg, 2009, vol. 5896, pp. 123–142.

[48] (2015, September) Maxima and minima. [Online]. Available: https://en.wikipedia.org/wiki/Maxima_and_minima/

[49] 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.

[50] 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.

[51] G. Zou, Y. Chen, Y. Xiang, R. Huang, and Y. Xu, "AI planning and combinatorial optimization for web service composition in cloud computing," in *Proceedings of the International Conference on Cloud Computing and Virtualization*, ser. CCV Conference 2010, 2010, pp. 28–35.

[52] C. Zeng, X. Guo, W. Ou, and D. Han, "Cloud computing service composition and search based on semantic," in *Cloud Computing*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, vol. 5931, pp. 290–300.

[53] Q. Duan, "Modeling and performance analysis on network virtualization for composite network-cloud service provisioning," in *Proceedings of the 2011 IEEE World Congress on Services*, ser. SERVICES '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 548–555.

[54] H. Bao and W. Dou, "A QoS-aware service selection method for cloud service composition," in *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*, May 2012, pp. 2254–2261.

[55] J. Huang, G. Liu, Q. Duan, and Y. Yan, "QoS-aware service composition for converged network-cloud service provisioning," in *Services Computing (SCC), 2014 IEEE International Conference on*, June 2014, pp. 67–74.

[56] Z. Ye, A. Bouguettaya, and X. Zhou, "QoS-aware cloud service composition based on economic models," in *Service-Oriented Computing*, ser. Lecture Notes in Computer Science, C. Liu, H. Ludwig, F. Toumani, and Q. Yu, Eds. Springer Berlin Heidelberg, 2012, vol. 7636, pp. 111–126.

[57] H. Qavami, S. Jamali, M. K. Akbari, and B. Javadi, "Dynamic resource provisioning in cloud computing: A heuristic markovian approach." Cloud Computing: 4th International Conference, CloudComp 2013, 2014, pp. 102–111.

[58] D. Kourtesis, J. M. Alvarez-Rodríguez, and I. Paraskakis, "Semantic-based QoS management in cloud systems: Current status and future challenges," *Future Generation Computer Systems*, vol. 32, no. 0, pp. 307–323, 2014.

[59] 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.

[60] A. S. da Silva, H. Ma, and M. Zhang, "A graph-based particle swarm optimisation approach to QoS-aware web service composition and selection," in *Evolutionary Computation (CEC), 2014 IEEE Congress on*, July 2014, pp. 3127–3134.

[61] J. Kwon and D. Lee, "Non-redundant web services composition based on a two-phase algorithm," *Data Knowl. Eng.*, vol. 71, no. 1, pp. 69–91, Jan. 2012.

[62] H. Ludwig, A. Keller, A. Dan, R. P. King, and R. Franck. (2003) Web service level agreement (WSLA) language specification. [Online]. Available: http://www.research.ibm.com/wsla/WSLASpecV1-20030128.pdf

[63] J. Hershberger and S. Suri, "Vickrey prices and shortest paths: what is an edge worth?" in

Foundations of Computer Science, 2001. Proceedings. 42nd IEEE Symposium on, Oct 2001, pp. 252–259.

[64] T. Yu and K.-J. Lin, "Adaptive algorithms for finding replacement services in autonomic distributed business processes," in *Autonomous Decentralized Systems, 2005. ISADS 2005. Proceedings*, April 2005, pp. 427–434.

[65] K.-J. Lin, J. Zhang, and Y. Zhai, "An efficient approach for service process reconfiguration in SOA with end-to-end QoS constraints," in *Commerce and Enterprise Computing, 2009. CEC '09. IEEE Conference on*, July 2009, pp. 146–153.

[66] K.-J. Lin, J. Zhang, Y. Zhai, and B. Xu, "The design and implementation of service process reconfiguration with end-to-end qos constraints in soa," *Service Oriented Computing and Applications*, vol. 4, no. 3, pp. 157–168, 2010.

[67] Y. Zhai, J. Zhang, and K.-J. Lin, "Soa middleware support for service process reconfiguration with end-to-end qos constraints," in *Web Services, 2009. ICWS 2009. IEEE International Conference on*, July 2009, pp. 815–822.

[68] G. Boella and R. Damiano, "A replanning algorithm for a reactive agent architecture," in *Artificial Intelligence: Methodology, Systems, and Applications*, ser. Lecture Notes in Computer Science, D. Scott, Ed. Springer Berlin Heidelberg, 2002, vol. 2443, pp. 183–192.

[69] R. van der Krogt and M. de Weerdt, "Plan repair as an extension of planning," in *In Proceedings of the 15th International Conference on Automated Planning and Scheduling*, 2005.

[70] A. Amin, L. Grunske, and A. Colman, "An automated approach to forecasting QoS attributes based on linear and non-linear time series modeling," in *Automated Software Engineering (ASE), 2012 Proceedings of the 27th IEEE/ACM International Conference on*, Sept 2012, pp. 130–139.

[71] Z. Zheng, Y. Zhang, and M. Lyu, "Investigating QoS of real-world web services," *Services Computing, IEEE Transactions on*, vol. 7, no. 1, pp. 32–39, Jan 2014.

[72] D. Menasce, "QoS issues in web services," *Internet Computing, IEEE*, vol. 6, no. 6, pp. 72–75, Nov 2002.

[73] L. Zeng, C. Lingenfelder, H. Lei, and H. Chang, "Event-driven quality of service prediction," in *Service-Oriented Computing-ICSOC 2008*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, vol. 5364, pp. 147–161.

[74] L. Shao, J. Zhang, Y. Wei, J. Zhao, B. Xie, and H. Mei, "Personalized QoS prediction for web services via collaborative filtering," in *Web Services, 2007. ICWS 2007. IEEE International Conference on*, 2007, pp. 439–446.

[75] R. Karim, C. Ding, and A. Miri, "End-to-end QoS prediction of vertical service composition in the cloud," in *Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on*, June 2015, pp. 229–236.

[76] 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.

[77] 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.

[78] 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.

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

[80] S. Borzsony and K. Kossmann, D. andStocker, "The skyline operator," in *Data Engineering, 2001. Proceedings. 17th International Conference on*, 2001, pp. 421–430.

[81] K.-L. Tan, P.-K. Eng, and B. C. Ooi, "Efficient progressive skyline computation," in *Proceedings of the 27th International Conference on Very Large Data Bases*, ser. VLDB '01. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001, pp. 301–310.

[82] S. Wang, B. C. Ooi, A. Tung, and L. Xu, "Efficient skyline query processing on peer-to-peer networks," in *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, April 2007, pp. 1126–1135.

[83] A. Vlachou, C. Doulkeridis, and Y. Kotidis, "Angle-based space partitioning for efficient parallel skyline computation," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '08. New York, NY, USA: ACM, 2008, pp. 227–238.

[84] J. Wu, L. Chen, Q. Yu, L. Kuang, Y. Wang, and Z. Wu, "Selecting skyline services for QoS-aware composition by upgrading mapreduce paradigm," *Cluster Computing*, vol. 16, no. 4, pp. 693–706, 2013.

[85] Y. Du, H. Hu, W. Song, J. Ding, and J. Lu, "Efficient computing composite service skyline with QoS correlations," in *Services Computing (SCC), 2015 IEEE International Conference on*, 2015, pp. 41–48.

[86] K. Bharat, A. Broder, M. Henzinger, P. Kumar, and S. Venkatasubramanian, "The connectivity server: fast access to linkage information on the web," *Computer Networks and {ISDN} Systems*, vol. 30, no. 1-7, pp. 469 – 477, 1998, proceedings of the Seventh International World Wide Web Conference.

[87] K. Randall, R. Stata, R. Wickremesinghe, and J. Wiener, "The link database: fast access to graphs of the web," in *Data Compression Conference, 2002. Proceedings. DCC 2002*, 2002, pp. 122–131.

[88] P. Boldi and S. Vigna, "The webgraph framework I: Compression techniques," in *Proceedings of the 13th International Conference on World Wide Web*, ser. WWW '04. ACM, 2004, pp. 595–602.

[89] N. Brisaboa, S. Ladra, and G. Navarro, "K^2-trees for compact web graph representation," in *String Processing and Information Retrieval*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, vol. 5721, pp. 18–30.

[90] D. J. Watts and S. H. Strogatz, "Collective dynamics of 'small-world' networks," *Nature*, vol. 393, pp. 440–442, 1998.

[91] (2014) K-ary tree. [Online]. Available: http://en.wikipedia.org/wiki/K-ary_tree

[92] D.-H. Shin, K.-H. Lee, and T. Suda, "Automated generation of composite web services based on functional semantics," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 7, no. 4, pp. 332–343, 2009.