Journal of Web Engineering, Vol. 13, No. 5&6 (2014) 361–377
© Rinton Press

TOWARDS AUTOMATIC CONSTRUCTION OF SKYLINE COMPOSITE SERVICES

SHITING WEN
Ningbo Institute of Technology, Zhejiang University, Ningbo, China
wensht@nit.zju.edu.cn

QING LI
City University of Hong Kong, Hong Kong, China
itqli@cityu.edu.hk

LIWEN HE[a]
Nanjing University of Posts and Telecommunications, Nanjing, China
helw@njupt.edu.cn

AN LIU
Soochow University, Suzhou, China
anliu@suda.edu.cn

JIANWEN TAO, LONGJIN LV
Ningbo Institute of Technology, Zhejiang University,Ningbo, China
tjw@zjbti.net.cn, magic316@163.com

Received September 6, 2013
Revised January 4, 2014

Due to the rapid increase of available web services over the Internet, service-oriented architecture has been regarded as one of the most promising web technologies. Moreover, enterprises are able to employ outsourcing software to build and publish their business applications as services, the latter can be accessible via the Web by other people or organizations. While there are a large number of web services available, often no single web service can satisfy a concrete user request, so one has to "*compose*" multiple basic services to fulfill a complex requirement. Web service composition enables dynamic and seamless integration of business applications on the Web. The traditional composition methods select the "*best*" composite service through defining a simple weight-additive method based on a utility function. But a service has multiple dimensions of non-functional properties, so how to assign weight for each QoS dimension is a non-trivial issue. In this article, we propose algorithms to compose skyline or top-k composite services for a given user request automatically. Experimental results show that our approach can find all skyline or a set of top-k composite services effectively and efficiently.

Keywords: Services; Skyline; Composition
Communicated by: G.-J. Houben & E.-P. Lim

1 Introduction

Web services are an emerging and promising technology in distribution applications. A service, identified by a URI in the Internet, is a software module whose interface and bindings are able to be defined, described, and discovered by XML artifacts [1], supporting direct interactions with other applications using XML-based messages via suitable Internet protocols. In this

[a]Corresponding Author

new computing paradigm, the functional granularity of services should be constrained from a reusability point of view [2]. Thus, multiple services need to be composed into a new value-added one, resulting in a composite service, to fulfill complex user requirements. Nowadays, this can be implemented in a quite flexible and efficient way through WS-BPEL [3], which has been approved as OASIS standard in 2007, or WS-CDL [4].

Naturally, the dynamicity and flexibility of services bring forth both opportunity and challenge to service composition, as we discuss below.

- **Opportunity**: The functional requirement of a user request may sometimes be so complex that none of existing services can fulfill it alone. With the help of service composition, a composite service can be constructed and invocated on-demand, which can fully exploit the power of web service technologies.

- **Challenge**: To construct such a composite service, not only functional but also non-functional requirement of user's request should be satisfied.

Web service composition is a methodology for building value-added applications by aggregating together several existing web services according to dynamic business requirements. As the web is populated with a large number of web services, multiple service providers often compete to offer the same functionality with different quality of services (e.g., response time, throughput, and price). Traditionally, if there are multiple services satisfying both functional and non-functional user requirement, a service is selected according to its value of some utility function. However, how to design or define the utility function to ensure its fairness to all the candidate services is a challenging issue. To the best of our knowledge, most utility functions [5, 6, 7, 8] use a simple weight-additive method to calculate a single numerical value and use this value as a guideline to select a service for invocation. The fairness of those methods is determined by the weight of each dimension of QoS attributes, but such weight is hard to be assigned objectively. Indeed, ensuring the fairness by means of a simple weigh-additive method in service composition is almost impossible. In order to overcome the shortcoming of simple weight-additive methods, skyline-based [9] methods can be utilized instead of assigning the weight to each QoS attribute dimension.

In this paper, we propose an efficient approach to automatically constructing composite services on-demand. The approach first obtains and recommends all skyline composite services to a user, and let the user select one to invoke according to his preferences. As the scale of skyline may get out of control, we also present a top-k dominating query technique as a replacement strategy when the skyline technique is unapplicable. The top-k dominating query can return k objects which dominate the highest number of objects in a dataset. Moreover, in order to control the scale of skyline candidate solutions, we further use cluster-based grouping method to reduce the skyline results. Its efficiency is demonstrated by extensive experiments on three comprehensive data sets.

The rest of the article is organized as follows. Section 2 gives some preliminaries to be used throughout the remainder of the paper. Section 3 present a motivating example. Section 4 presents an automatic composition framework and associated algorithms. The details of our experiments are given in section 5. Section 6 discusses some related works. Finally, section 7 concludes the paper and sheds light on future research.

2 Preliminaries

In this section, we briefly introduce the notion of skyline query first, and then address how to apply it in our approach.

Given a set of points in a d-dimensional space, a skyline query selects those points that are not dominated by any other point. A point x is said to dominate another point y, if x is better than or equal to y in all dimensions and strictly better in at least one dimension. Intuitively, a skyline query selects the "*best*" or most "*interesting*" objects with respect to all the dimensions. In this section, we define and exploit dominance relations between services based on their QoS attributes, which is used to identify and prune services that are dominated by other services in the same class.

Definition 1 *(*Dominance*) Consider a service class S, and two services $x, y (\in S)$ which are characterized by a set Q of QoS attributes. We term x dominates y, denoted as $x \succ y$, iff x is as good as or better than y in all parameters in Q and better in at least one parameter in Q, i.e., $\forall k \in [1, |Q|] : q_k(x) \leq q_k(y)$ and $\exists k \in [1, |Q|] : q_k(x) < q_k(y)$.*

Corollary 1 *(*Transitivity*) If a service x dominates another service y and y dominates z, then x dominates z. That is, if $x \succ y$ and $y \succ z$, then $x \succ z$.*

This corollary is obvious, so the proof is omitted. However, it is very powerful for pruning the search space.

Definition 2 *(*Skyline Services*) The skyline services of a service class S, denoted by SLs, comprise those services in S that are not dominated by any other service, i.e.,$SLs = \{x \in S | \neg\exists y \in S : y \succ x\}$.*

Definition 3 *(*Dominating Score*) Given a service $x \in S$, $\varphi(x)$ denotes the dominating score of the service x, i.e.,*

$$\varphi(x) = |\{y | x \succ y, x, y \in S\}| \tag{1}$$

In order to retrieve the top-k dominating services, we only need to obtain k services whose dominating score are large than those of the remaining services. The detail is shown as follows:

Corollary 2 *Given two services x and y, if x dominates y, then the dominating score of x is larger that that of y. That is:*

$$x \succ y \Longrightarrow \varphi(x) > \varphi(y) \tag{2}$$

Proof: According to Corollary 1, x will dominate any service which is dominated by y if x dominates y. In other words, the dominating score of x is at least larger than or equal to that of y plus one, since x dominates y meanwhile. Hence, $\varphi(x) > \varphi(y)$ $\square$.

Figure 1 shows an example of skyline and Top-k services for a given service class (e.g., hotel). Each service is described by two QoS parameters (e.g., price and distance). Thus, the services are represented as points in the 2-dimensional space, with the coordinates of each point corresponding to the values of the service in these two parameters. As shown in Figure 1, the skyline query may return $p1$, $p2$ and $p5$, because these are not dominated by any other services. On the other hand, the other points are not contained in the skyline, because they are dominated by the at least one of the skyline point(s). A key advantage of skyline query is that it does not require the use of a specific ranking function; its results only depend on the intrinsic characteristics of the data. However, the shortcoming of skyline query is that the

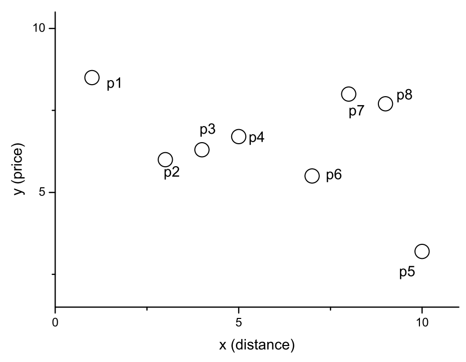


Fig. 1. An Example of Skyline and Top-k Services

size of skyline can not be controlled by the user and it can be as large as the data size in the worst case [10, 11, 12, 13]

Consequently the skyline services provide different trade-offs between the QoS parameters, hence are incomparable to each other when there is no pre-specified preference scheme regarding the relative importance of these parameters. Accordingly to the top-k dominating query returns k points with the highest score. For example, the top-2 dominating query on the data of Figure 1 retrieves $p2$ (with $\varphi(p2) = 4$) and $p3$ (with $\varphi(p3) = 3$). The results may indicate that user may consider price and distance as selection factors. We can see that the biggest advantage of top-k is that the output size can be easy to control when comparing to skyline.

2.1 Automatic Composition

Before defining automatic composition, we first present the definition of a web service s_i.

Definition 4 (Service) *A service s_i is a 3-tuple $s_i = (I_i, O_i, Q_i)$ where I_i is a set of input parameters, O_i is a set of output parameters, and Q_i is a set of QoS parameters.*

Note that, the above service definition applies to not only atomic services but also composite ones because both expose only their interface to users. Due to the limited capability of a single service, often a number of services need to work together to satisfy a user's request.

Formally, we define automatic composition as follows:

Definition 5 *(*Automatic Composition*) Given a request $r_k = (I_k, O_k)$, the process of automatic composition is to repeat condition (b) holds or no more services available in the registry.*

$$\begin{array}{ll} (a). & I_k \bigcup O_1 \bigcup O_2 ... \bigcup O_i \supseteq I_{i+1} \\ (b). & I_k \bigcup O_1 \bigcup O_2 ... \bigcup O_{i+1} \supseteq O_k \end{array} \tag{3}$$

The goal of automatic composition is thus to find a set of services that can work together to meet user requests. The structure of a composite service can be represented by a *DAG* (Directed Acyclic Graph). Note that, this is a simple approach because we do not consider any QoS of the component services when we compose a composite service. Next, we present the definition of a QoS-Aware automatic composition.

Definition 6 *(QoS-Aware Automatic Composition) Given a request $r_k = (I_k, O_k, Q_k^c)$, the process of QoS-aware automatic composition is to find a service s_i such that*

$$\begin{array}{ll} (a). & I_k \bigcup O_1 \bigcup O_2 ... \bigcup O_i \supseteq I_{i+1} \\ (b). & I_k \bigcup O_1 \bigcup O_2 ... \bigcup O_{i+1} \supseteq O_k \\ (c). & \forall k \in [1, |Q|],\ Q_k^{cs} \ \le \ Q_k^c,\ \textit{and utility}^{cs} \textit{ is optimal.} \end{array} \tag{4}$$

Here, Q_k^{cs} and Q_k^c indicate the k^{th} QoS parameter value and user's constraints on the k^{th} QoS parameter of the composite service cs, respectively. $utility^{cs}$ represents the utility value of the composite service cs.

2.2 Problem Statement

Traditionally, QoS-driven service composition is an optimization problem, where the user defines an abstract process that is specified as a collection of generic service tasks [5, 8], and the selection of the services is conducted by executing a given task of the process specification. This approach however does not take into account the other tasks involved in the composite service. In contrast, our QoS-aware automatic composition aims to construct a composite service according to user's input and desired output parameters over a group of component services without pre-formatting an abstract process.

More specifically, QoS-Aware automatic composition is a constraint optimization problem which aims at finding the composition that maximizes (or alternatively, minimizes) the overall QoS values, thereby satisfying all the global QoS constraints. In the rest of this article, we present a QoS-aware automatic composition approach to constructing all skyline composite services as candidate solutions.

Definition 7 *(*Skyline/Top-k-based Automatic Composition*) Skyline/Top-k-based automatic composition is QoS-aware through enumerating and comparing all possible combinations of candidate composite services, as opposed to just finding a composite service whose QoS value is the "* optimal*". All skyline or a set of Top-k composite services are returned as candidates per any given user request.*

In the following section, we address this problem by considering dominance relations when selecting and recommending skyline composite services as candidate solutions.

3 Motivating Example

As a motivating example, consider a scenario where a recommender system recommends translation services to users. Suppose six translation services are available, and their basic information is shown in Table 1. The "*Input*" and "*output*" columns describe the functionality of services. The "*Price*" and "*Time*" columns describe the quality of services. Now, consider the following two requests:

- Find a service that can translate a text from English to Chinese and its response time should be less than 500ms.
- Find a service that can translate a text from German to Japanese and its response time should be less than 1500ms.

For the first request, the recommender system can make a recommendation at once: both s_1 and s_2 provide translation function from English to Chinese, but only s_1 can return the translation result within 500ms, so it should be recommended.

Table 1. Translation Services

Name	Input	Output	Price	Time
s_1	English	Chinese	10	300
s_2	English	Chinese	2	800
s_3	German	English	5	200
s_4	German	English	3	300
s_5	English	Japanese	10	600
s_6	English	Japanese	15	500

For the second request, a service needs to take German as input and return Japanese. Unfortunately, none of the services can fulfill the request alone. It should however be noticed that s_3 and s_4 can translate German into English while s_5 and s_6 can translate English into Japanese. Therefore, we can select one service from $\{s_3, s_4\}$ and another from $\{s_5, s_6\}$, and compose them into a composite service to fulfill the required functionality. We have four possible combinations, $(s_3 + s_5)$, $(s_3 + s_6)$, $(s_4 + s_5)$, and $(s_4 + s_6)$, and their QoS ("*Price*" and "*Time*") values are (15, 800), (20, 700), (13, 900), and (18, 800), respectively. We can observe that the QoS value of $(s_3 + s_5)$ is better than $(s_4 + s_6)$ since the price of $(s_3 + s_5)$ is lower than that of $(s_4 + s_6)$ and the response time of $(s_3 + s_5)$ is less than that of $(s_4 + s_6)$. However, the other composition solutions can not dominate each other. Therefore, we have to return all remaining combination solutions $(s_3 + s_5)$, $(s_3 + s_6)$, and $(s_4 + s_5)$ to the user, which constitute a set of skyline composite services.

4 Constructing Skyline Composite Services

4.1 Overview

Figure 2 gives an overview of the automatic composition framework which involves four types of entities: Service, Service Registry, Customer, and Composer. A service publishes its basic information (e.g., input parameters, output parameters, and QoS parameters) in the Service Registry when it is ready for usage (Step (1)). The automatic composition begins when a user sends to the Composer a request that contains required functionality and personal preferences (Step (2)). After receiving the request, the Composer makes use of the search ability of Service Registry to find out all the possible candidates that can fulfill the user's requirements. A composition will be activated if no single service can fulfill the user's requirements. In this case, several services will be selected and composed as a temporary composite service and recommended to the user as a candidate if the composite service is not dominated by any other composite service (Step (3)). The user selects and invokes a composite service from the set of candidate composite services (Step (4)).

As shown in Figure 2, three components work together to realize automatic service composition. In particular, the WSDL parser is responsible for parsing service description files and automatically obtaining the input and output parameters of services; the Service Composer constructs candidate composite services by specifying which and how services can be composed to fulfill user requirements; Specially, the Service Selector takes charge of a set of candidate composite services by selecting one of them to invoke according to user preferences. The detailed functions of these components are to be discussed in the following sections.

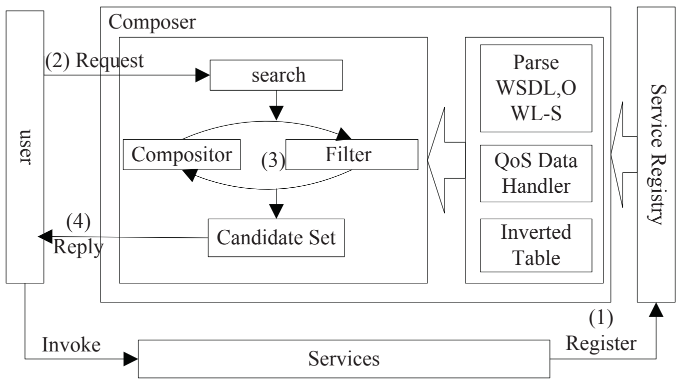


Fig. 2. Architecture of QoS-Aware Automatic Composition

Table 2. A Set of Services

Name	Input	Output	QoS	Name	Input	Output	QoS
s_1	a,b,c	d	9, 1	s_6	l,j	d	5, 7
s_2	a,b	e,f	1, 2	s_7	h	d	2, 3
s_3	c,e	h	2, 4	s_8	g	h	5, 2
s_4	c,f	g	5, 6	s_9	c	d	11, 2
s_5	k	h	6, 8	s_{10}	b	e	2, 1

We first introduce the data structure which will be used in our algorithms. As shown in Table 2, each component service has a set of input and output parameters, respectively. Meanwhile, it also has a vector of QoS attributes. For example, service s_1 has a set of input parameters a, b, and c, output parameter d, along with two QoS dimensions: 9 and 1. Moreover, all component services are indexed according to their input parameters, as shown in Table 3.

4.2 Constructing Skyline Composite Services

Because our goal is to construct all skyline composite services as candidate solutions, we first prune all non-skyline component services which are of the same functionality, so as to keep the set of candidate component services of each functionality as small as possible. In this way, we can speed up the skyline-based automatic composition process effectively.

Lemma 1 *Suppose $CS = \{s_1, s_2, ..., s_n\}$ is one of the skyline candidate solutions for a given request, i.e.,* CS *is a composite service that satisfies all the specified constraints and is not dominated by any other candidate solution. Then, each constituent service s_i of CS belongs to skyline of the corresponding class, i.e.$\forall s_i \in CS : s_i \in SLs_i$ where SL_i denotes the skyline service class of s_i.*

The above lemma has been proven in [13, 12], so it is omitted here. In the next section, we will exploit this lemma for pruning the search space.

We solve the problem of constructing skyline composite services in two steps. First, we construct an enabled list that consists of a sequence of services which may be composed into a composite service (i.e., they can be activated), and filter other services which can't

Table 3. Input Parameter Indexed Table

Input	Services	Input	Services
a	s_1 s_2	k	s_5
b	s_1 s_2 s_{10}	l	s_6
c	s_1 s_3 s_4 s_9	j	s_6
d	s_3	h	s_7
e	s_4	g	s_8

Algorithm 1: Constructing Composite Services

```
   input  : $R$:User's Request
   output: $PSC$:Providing Skyline Services
1  begin
2      $s_i \leftarrow R\ (s_i.input = \#,\ s_i.output = R.input)$;
3      $s_o \leftarrow R\ (s_o.input = R.output,\ s_o.output = \#)$;
4      $asList \leftarrow asList \cup s_i$;
5      foreach $p \in s_i.output$ do
6          $PSC(p) \leftarrow s_i$;
7      end
9      while $maxKeys < asList.size$ do
10         $maxKeys \leftarrow asList.size$;
11         $\delta \leftarrow \Omega(PSC.keys)$;
12         foreach $s\ in\ \delta$ do
13             foreach $p\ in\ s.output$ do
14                 $PSC(p) \leftarrow updatingPSC(p, s, PSC)$;
15             end
16             $asList \leftarrow asList \cup s$;
17         end
18     end
19     if $s_o \in asList$ then
20         $resultList \leftarrow PSC(s_o.output)$;
21         foreach $element\ in\ resultList$ do
22             $Print:\ element$;
23         end
24     else
25         $return:\ no\ result$;
26     end
27 end
```

be activated by the request. In the second step, we select a service in a stepwise fashion by constructing skyline composite services repeatedly and iteratively.

To address the first problem, we prune the services by building a list of activated services, called *asList*, which contains all activated services as per user input parameters. The aim of our algorithm is to generate all possible skyline composite services for each output parameter, and construct for every parameter p an inverted index $PSC(p)$ which is a set of skyline composite services whose output parameters contain p. Based on PSC, we return all skyline composite services according to the output parameters of the user request. Algorithm 1 shows the procedure of constructing skyline composite services, where s_i and s_o are two virtual services which are generated according to user request. In particular, s_i indicates the virtual input service and s_o indicates the virtual output service, respectively (Lines 02-03). Note that *asList* indicates a set of activated services. Clearly, its initial value is the virtual input service given in the user request. Our algorithm continues to explore newly activated services by building a loop, and stops when no new services can be activated (Lines 09-18). At each iteration, function Ω takes PSC's keys as input and returns a set of services (denoted as δ) whose input parameters are contained by the keys of PSC(Line 11). For each output parameter p of service s that is contained by δ and is not visited before, the algorithm first adds it into PSC and then adds s into *asList*.

The procedure of updating PSC is shown in Algorithm 2. Here, we use a set $skylineSet(p)$ to store all skyline composite services whose output parameters contain parameter p. For each input parameter p' of service s, we first search its $skylineSet(p')$ and generate all skyline composite services whose output can be used to activate the service s (Line 05). As the parameter p is one of the output parameters of service s, if it does not belong to the keys of *PSC*, we can insert *skylineSet*(s) to PSC directly (Lines 06-07). Otherwise, we need to compare *skylineSet*(s) with $PSC(p)$ to see whether they have some element not dominated by each other, and insert the elements in *skylineSet*(s) which are not dominated by $PSC(p)$ and discard the other elements of *skylineSet*(p) (Line 08-13).

With the help of Algorithms 1 and 2, all skyline composite services can be indexed in PSC ($s_o.output$). Take the set of services in Table 2 as an example. Suppose that we have a request whose input parameters are a, b, c, and output parameter is d, respectively. We first generate two virtue services s_i and s_o. The service s_i's input parameters and output parameters are #, and a, b, and c, respectively. The service s_o's input parameter and output parameter are d and #, respectively. Then, we can construct a set of skyline composite services accordingly. We obtain three composite services which are $s_i - s_1 - s_o$ whose QoS value is (9, 1), $s_i - s_2 - s_7 - s_o$ whose QoS value is (6, 8), and $s_i - s_3 - s_{10} - s_7 - s_o$ whose QoS value is (5, 9).

In view of the complexity of calculating skyline immediate composite services (c.f. Line 05 of Algorithm 2), we next propose a cluster-based grouping method to reduce the complexity for calculating the skyline composite candidate services.

4.3 Cluster-based Grouping

Clearly, when the set of skyline composite service is obtained, the size of this set matters a lot in practice. There have been some approaches such as [14] proposed by focusing on smaller number of representative items. Here, we investigate an approach for web service discovery by

Algorithm 2: Updating PSC to Building Skyline Composite Services

```
input  : $p$, $s$, $PSC$, $asList$
output: $PSC$
1  begin
2      foreach $p' \in s.input$ do
3          $skylineSet(p') \leftarrow PSC(p')$;
4      end
5      $skyline(s) \leftarrow \underset{\succ}{\bowtie} SkylineSet(p')$;
6      if $p \notin PSC.key$ then
7          $insert: \ PSC(p) \leftarrow skyline(s)$;
8      else if $\exists t \in skyline(s) \succ PSC(p)$ then
9          $insert: \ PSC(p) \leftarrow PSC(p) \cup skyline(s)$;
10     else
11         $discard: \ skyline(s)$;
12         $asList \leftarrow asList \cup skyline(s)$;
13     end
14 end
```

means of clustering the matched services. Our goal is to select a set of representative skyline services with different trade-offs for the various QoS parameters, and to use this reduced set as input for our algorithms in order to save the computation time by reducing the (intermediate) results.

The main idea of our grouping method is to divide the skyline services into k clusters. When the intermediate clusters are too large, we select one (or several) representative service(s) from each cluster. In this way, we select representative services with the best utility value (some other user preferences can also be applied in our methods, easily). At run-time, when a service composition request is processed, we start the search from the initial service classes by considering all services belonging to them. We use the popular k-means clustering algorithm [15] to build the representative services, as shown in Algorithm 3. The algorithm takes as input the skyline set SL of class S and returns a representative service set of class RSL. The size of RSL is equal to k if we select one service from each cluster, or multiples of k if we select several services from each cluster.

4.4 Constructing Top-k composite services

In this section, we present algorithms to retrieve top-k services, and then conduct extensive experiments comparing to the skyline-based approach. Identical to the skyline-based method, we first prune non-top-k component services to keep the set of candidate as small as possible. Lemma 2 is used to guide our pruning procedure.

Lemma 2 *Suppose $CS = \{s_1, s_2, ..., s_n\}$ be one of the top-k candidate solutions for a given request, i.e.,* CS *is a composite service that satisfies all the specified constraints and dominates highest number of other composite services. Then, each constituent service s_i of CS belongs to top-k of the corresponding class, i.e., $\forall s_i \in CS : s_i \in TKs_i$ where TK_i denotes the Top-k service class of s_i.*

Algorithm 3: Cluster-based Grouping Method

```
input  : a set of skyline services SL, k
output: a set of representative skyline service RSL
1 begin
2     $C_s \leftarrow KMeansCluster(SL, k)$;
3     $RSL \leftarrow \emptyset$;
4     foreach cluster $c \in C_s$ do
5         $s \leftarrow maxUtilityService(c)$;
6         $RSL \leftarrow RSL \cup s$;
7     end
8 end
```

Proof: Suppose s_i is a service which is part of the composite service CS and does not belong to the Top-k of its class S_i. Then, according to Definition 3 and Corollary 2, there exists another service s_i' which dominate more other services (at least one more) than s_i in all considered QoS parameters. Let $\overline{CS}$ is a composite service which equal to CS but without the service s_i or s_i'. Then, we assume that the service s_i and s_i' dominate n and $n+1$ other services in the same class S_i according to Definition 3 and Corollary 2. Since the QoS aggregation functionality is monotone (i.e. the higher the values, the better the quality), if $x \succ y\ : x, y \in S_i$, then $(\overline{CS}+x) \succ (\overline{CS}+y)$. That is , if $x \succ y$, we can conclude that $\varphi(\overline{CS}+x) > \varphi(\overline{CS}+y)$. Therefore, we can easily conclude that the composite service $(\overline{CS}+s'_i)$ dominate more other composite services than $(\overline{CS}+s_i)$ does (at least one more), i.e., $\varphi(\overline{CS}+s'_i) \geq \varphi(\overline{CS}+s_i)+1$. Hence, it contradicts with the fact that CS is one of the top-k candidate solutions$\square$.

After pruning the candidate component services, we also need to build a list of activated services, the process of which is also identical to the skyline-based method (c.f., Algorithm 1). The procedure of updating PSC is shown in Algorithm 4. As proposed in Section 2, the top-k based composite services is to retrieve k objects which dominate the highest number of composite services. In Algorithm 4, $Top-kSet(p)$ is maintained as a set of Top-k intermediate composite services whose output parameter contain parameter p (Line 03). To facilitate synchronous traversal join, the component services in a service class are pre-organized by an R-tree data structure [11] (Line 05). We also use rules presented in Section 2 (see Corollaries 1 and 2) to conduct an in-memory update of top-k composite services (Line 08-09). After performing line 09, for each composite service, we insert or discard intermediate top-k composite service to the set of PSC accordingly.

5 Experiments

In this section, we evaluate the effectiveness of our method by conducting extensive experiments. Given a service class, the size of the skyline services depends on the distribution of the QoS values and correlations between the different QoS parameters. Figure 3 shows three types of two dimensional datasets as an example: 1) the correlated dataset, in which a service that is good in one dimension is also good in the other dimensions; 2) the independent dataset, in which the values of the two QoS dimensions are independent of each other; 3)

Algorithm 4: Updating PSC to Building Top-k Composite Services

```
input  : $p$, $s$, $PSC$, $asList$
output: $PSC$
begin
    foreach $p' \in s.input$ do
        $Top - kSet(p') \leftarrow PSC(p')$;
    end
    $Top - k(s) \leftarrow Top - kSet(Max - k(\varphi(p')))$;
    if $p \notin PSC.key$ then
        $insert: PSC(p) \leftarrow Top - k(s)$;
    else if $\exists t \in Top - k(s), \varphi(t) > \varphi(PSC(p))$ then
        $insert: PSC(p) \leftarrow PSC(p) \cup Top - k(s)$;
    else
        $discard: Top - k(s)$;
        $asList \leftarrow asList \cup Top - k(s)$;
    end
end
```

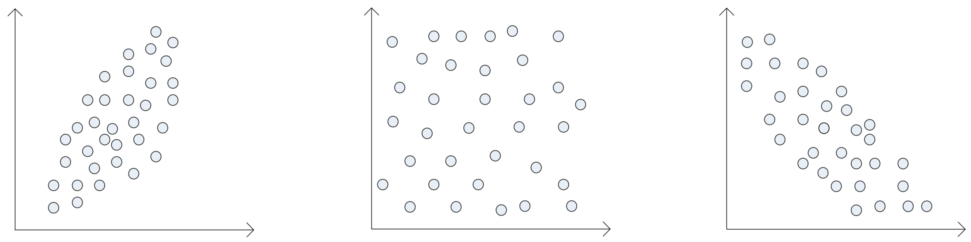

(a) Correlated (b) Independent (c) Anti-Correlated

Fig. 3. Different Skyline Data Types

the anti-correlated dataset, in which there is a clear trade-off between the two dimensions. The number of skyline services is relatively small in the correlated datasets, large in the anti-correlated and medium in the independent ones.

We perform experiments on a PC with 2.2GHz Intel Pentium Duo2 CPU, 2048M of RAM using Microsoft Windows 7 Operating Systems and J2SDK 1.7. To make the experimental result comparable, our experiments are conducted on a publicly available (free) dataset EEE05 [16]. However, this dataset only contains the WSDL files of services; the QoS information is not available. In order to test our approach with a larger number of services and different distributions, we use each WSDL file of EEE05 as a service class and use a publicly available synthetic generator [17] to obtain three different datasets processing the patterns of Figure 3). Each service's QoS is represented by a vector of five dimensions.

Figure 4 shows the ratio of skyline data points for the cases of correlated, independent, and anti-correlated. From Figure 4, we can see that the skyline ratio for the independent and anti-correlated is increasing along with the number of dimensions increasing. When the dimension is lower than ten, the skyline ratio increases very slowly. When the number of dimensions reaches to ten, the skyline ratio of the anti-correlated reaches to about 60 percent. Next, we

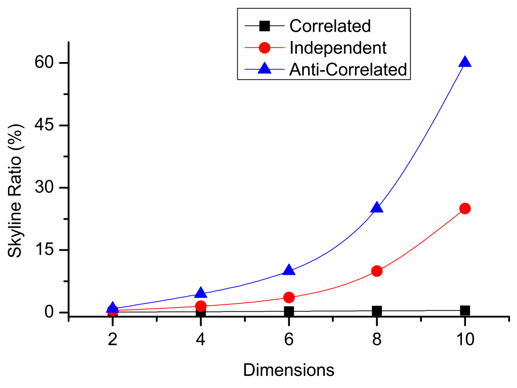


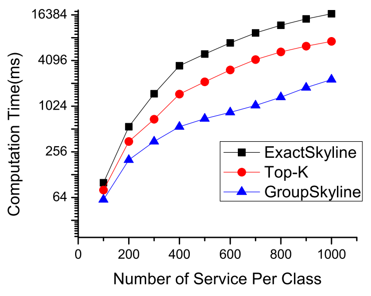
Fig. 4. Skyline Ratio of Different Data Sets

compare the efficiency of the following two automatic composition methods for constructing all skyline composite services:

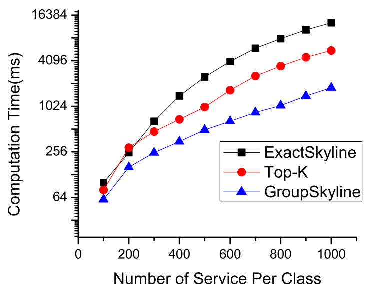
- *ExactSkyline*: this method uses Algorithms 1 and 2 only (that is, without using the grouping method) to generate the skyline composite services.
- *GroupSkyline*: this method uses not only Algorithms 1 and 2 but also adopts the grouping method (Algorithm 3) for saving the computation time and reducing the (intermediate results). In this method, we assume that the grouping algorithm is activated only when the intermediate or final results are ten times more than the initial number of services per class, in which case we set k to be the initial number of services per class.
- *Top-k*: this method uses Algorithms 1 and 4 to generate the top-k composite services. Here, we set k to 100, because the initial number of component services is also set to 100, and we use aRtree as in-memory index, so the influence on performance by the number of k can be omitted [11].

We first investigate the performance of our proposed methods through measuring the average computation time required by each of the aforementioned three methods. We vary the number of service candidates from 100 to 1000 services per class. The results of this experiment are shown in Figure 5. Comparing the performance of *ExactSkyline*, *GroupSkyline*, and *Top-k* methods, we can observe that a significant gain is achieved when the grouping method is applied. However, as expected, this gain in performance varies for the different datasets, depending on the size of the skyline services. Moreover, the top-k method always outperforms Exactskyline method, and its performance is always between ExactSkyline method and GroupSkyline method.

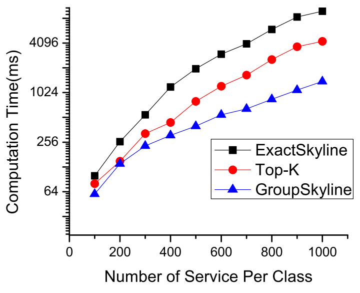
In the next set of experiments, we measure the success ratio of the three methods by varying the number of users' constraints from 1 to 5, i.e., the percentage of skyline composite services where a solution is found to satisfy users' requirements. As shown in Figure 6, the solutions founded by the *ExactSkyline* and *Top-k* can almost satisfy all of the users' requirements. Particular, the success ratio of these two methods always approach to 100



(a) Anti-Correlated



(b) Independent



(c) Correlated

Fig. 5. Computation Time vs. Number of Services per Class

percent when the number of QoS constraints equals to 1 or 2. Event with the five QoS constraints, the success ration of those two methods can still be larger than 90 percent. However, the success ratio of the *GroupSkyline* method degrades quickly as the number of QoS constraints increases. The reason for this behavior is that the *GroupSkyline* method uses grouping technique to filter skyline services. And the scale of the filtered skyline services may increase when the number of initial number of services varies from 100 to 1000. Here, we only consider the dataset of independent, yet the performance of *Top-k* methods in the other two datasets is similar, hence is omitted to keep the presentation succinct.

6 Related Work

In this section, we review from the literature some existing works related to our research. Web service as a key technique for implementing service-oriented architecture (SOA) has been attached with more and more importance. Earlier works have focused on how to select the desired service from all functionality-similar services. In [18], a framework to evaluate the QoS from a vast number of web services is constructed, with an aim to enable quality-driven web service selection. More recently, web service composition has been studied extensively, focusing on two general types of composition: *manual* and *automatic*. The former manually defines a process consisting of multiple tasks, and its objective is to bind these tasks to concrete services while satisfying user QoS constraints. Hence, it is also called QoS-aware service composition. In [8], the authors consider it as an integer programming (IP) problem in which the objective function is defined as a linear composition of multiple QoS parameters. In [19],

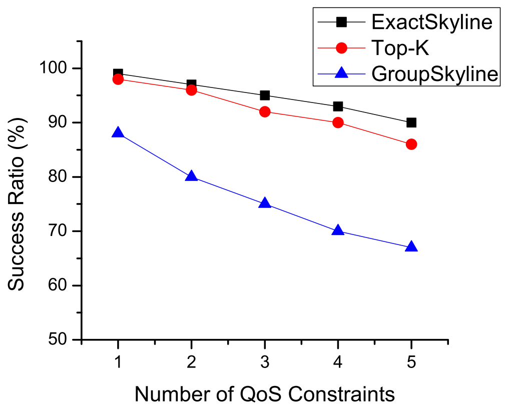


Fig. 6. Independent Dataset: Success Ratio vs. Number of QoS Constraints

the authors also adopt IP but they have a different method to eliminate loop constructs in the process of composite services. Yu et al. [5] propose two models: a combinatorial model and a graph model. The combinatorial model considers service composition as a multi-dimension multiple choice 0-1 knapsack problems. The graph model considers service composition as a multi-constraint optimal path problem. They propose a polynomial heuristic algorithm for the combinatorial model and an exponential heuristic algorithm for the graph model. Alrifai and Riss [6] decompose global QoS constraints into an optimal set of local QoS constraints by using IP technique. The satisfaction of local constraints guarantees the satisfaction of global constraints. Through the decomposition of global constraints, it is only necessary to conduct several local selections simultaneously, which significantly improves the performance of the composition process.

Automatic service composition, on the other hand, aims at dynamically generating the composite services. Current approaches for automatic service composition are mainly based on AI planning [20]. Oh et. al. propose a planning algorithm called *WSPR*[21]. It is essentially in accordance to Graph plan [22], but differs in that it adopts a heuristic to minimize the number of services in a solution while Graph plan aims at minimizing the number of time steps but not necessarily the number of actions. However, it ignores QoS constraints, so the composite service sometimes cannot fulfill the user's QoS requirements. While the authors of [23] do consider QoS, they only focus on one dimension, which is not applicable to the ubiquitous QoS model of Web services. Moreover the method presented in [23] is hard to be extended to multiple QoS dimensions. The authors of [12, 24] propose to employ skyline and top-k dominating techniques, respectively, so as to tackle the drawback of the traditional method which requires service users to assign weights to each QoS attribute.

With no exceptions, currently, existing service composition methods can only return one solution to the user since all QoS-aware composition is to determine which one is the best based on a utility function and return the optimal to the end customer. However, for a set of services which has multiple QoS parameters, it is hard to estimate which one is better, since a service could be better than others in one parameter but worse in another parameter. Skyline queries, which gather all special data objects that are not dominated by each other in a data set, have received significant attention over recent years. The skyline technique provides a unique perspective on the dominance relationship, and is totally dependent on the intrinsic characteristics of the data. Based on the skyline technique, our work not only dynamically generates the process, but also obtains and returns all skyline services as candidate solutions. Interestingly, the top-k dominating query has also played an increasingly significant role in contemporary applications such as multi-criteria decision making, data cleaning, and so on [10, 11].

7 Conclusion

In this paper, we have presented an approach to constructing skyline composite services as a set of candidate solutions per a user request. By identifying the skyline services in terms of their QoS values, our method can find all possible skyline solutions for a given request. In order to address the over size problem of skyline candidate solutions, we have devised a grouping method to accelerate the computation process and reduce the scale of candidate solutions. We have also presented the top-k based method which has good performance and success ratio, as replacement method to the skyline based method, especially for a large date set. The results of our experimental evaluation indicate a significant gain of this approach.

In our future studies, we plan to devise more sophisticated algorithms to further improve the efficiency of constructing skyline composite services. In addition, other issues such as service profiling are also relevant and can be useful to our aim, which will also be investigated subsequently.

Acknowledgment

This work has been supported by the Natural Science Foundation of Ningbo with the projects Grant No.2014A610066. And also supported by the Natural Science Foundation of China under Grant No. 11326230,61003044,61303019, and 61073061,and the State Key Program of National Natural Science Foundation of China with the project Grant No. 61232006.

References

1. M. S. Aktas, G. C. Fox, M. Pierce, and S. Oh, "Xml metadata services," *Concurrency and Computation: Practice and Experience*, vol. 20, no. 7, pp. 801–823, 2008.
2. R. Haesen, M. Snoeck, W. Lemahieu, and S. Poelmans, "On the definition of service granularity and its architectural impact," in *Advanced Information Systems Engineering.* Springer, 2008, pp. 375–389.
3. D. Jordan, J. Evdemon, A. Alves, A. Arkin, S. Askary, C. Barreto, B. Bloch, F. Curbera, M. Ford, Y. Goland *et al.*, "Web services business process execution language version 2.0," *OASIS Standard*, vol. 11, 2007.
4. Y. Xia, G. Dai, J. Li, T. Sun, and Q. Zhu, "A model-driven approach to predicting dependability of ws-cdl based service composition," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 10, pp. 1127–1145, 2011.
5. T. Yu, Y. Zhang, and K.-J. Lin, "Efficient algorithms for web services selection with end-to-end qos constraints," *ACM Transactions on the Web (TWEB)*, vol. 1, no. 1, p. 6, 2007.
6. 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.* ACM, 2009, pp. 881–890.
7. M. Serhani, R. Dssouli, A. Hafid, and H. Sahraoui, "A qos broker based architecture for efficient web services selection," in *proceedings of IEEE International Conference on Web Services*

(ICWS),. IEEE, 2005, pp. 113–120.
8. L. Zeng, B. Benatallah, A. H. H. Ngu, M. Dumas, J. Kalagnanam, and H. Chang, "Qos-aware middleware for web services composition," *IEEE Transactions on Software Engineering*, vol. 30, no. 5, pp. 311–327, 2004.
9. S. Borzsony, D. Kossmann, and K. Stocker, "The skyline operator," in *Proceedings 17th International Conference on Data Engineering*. IEEE, 2001, pp. 421–430.
10. M. L. Yiu and N. Mamoulis, "Efficient processing of top-k dominating queries on multi-dimensional data," in *Proceedings of the 33rd international conference on Very large data bases*. VLDB Endowment, 2007, pp. 483–494.
11. C. Tang, S. Wen, Q. Li, Y. Xiong, and A. Liu, "Probabilistic top-k dominating composite service selection," in *2012 IEEE Ninth International Conference on e-Business Engineering (ICEBE)*. IEEE, 2012, pp. 1–8.
12. Q. Yu and A. Bouguettaya, "Computing service skyline from uncertain qows," *IEEE Transactions on Services Computing*, vol. 3, no. 1, pp. 16–29, 2010.
13. 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*. ACM, 2010, pp. 11–20.
14. D. Skoutas, D. Sacharidis, A. Simitsis, and T. Sellis, "Ranking and clustering web services using multicriteria dominance relationships," *IEEE Transactions on Services Computing*, vol. 3, no. 3, pp. 163–177, 2010.
15. S. Lloyd, "Least squares quantization in pcm," *IEEE Transactions on Information Theory*, vol. 28, no. 2, pp. 129–137, 1982.
16. M. Blake, K. Tsui, and A. Wombacher, "The eee-05 challenge: A new web service discovery and composition competition," in *proceedings of IEEE International Conference on e-Technology, e-Commerce and e-Service,*. IEEE, 2005, pp. 780–783.
17. Random, "Random data gnerator," *http://randdataset/projects/postgresql/org/*, accessed in 5 May. 2013.
18. Y. Liu, A. H. Ngu, and L. Z. Zeng, "Qos computation and policing in dynamic web service selection," in *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*. ACM, 2004, pp. 66–73.
19. D. Ardagna and B. Pernici, "Adaptive service composition in flexible processes," *IEEE Transactions on Software Engineering*, vol. 33, no. 6, pp. 369–384, 2007.
20. J. Rao and X. Su, "A survey of automated web service composition methods," in *Semantic Web Services and Web Process Composition*. Springer, 2005, pp. 43–54.
21. S.-C. Oh, D. Lee, and S. R. Kumara, "Web service planner (wspr): An effective and scalable web service composition algorithm," *International Journal of Web Services Research (IJWSR)*, vol. 4, no. 1, pp. 1–22, 2007.
22. A. L. Blum and M. L. Furst, "Fast planning through planning graph analysis," *Artificial intelligence*, vol. 90, no. 1, pp. 281–300, 1997.
23. W. Jiang, C. Zhang, Z. Huang, M. Chen, S. Hu, and Z. Liu, "Qsynth: A tool for qos-aware automatic service composition," in *2010 IEEE International Conference on Web Services (ICWS)*. IEEE, 2010, pp. 42–49.
24. D. Skoutas, D. Sacharidis, A. Simitsis, V. Kantere, and T. Sellis, "Top-k dominant web services under multi-criteria matching," in *Proceedings of the 12th international conference on extending database technology: advances in database technology*. ACM, 2009, pp. 898–909.