Scaling up Web Service Composition with the Skyline Operator

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

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

***Abstract*—Web service composition enables the provision of existing resources on the web without investing in new infrastructure. However, searching an optimal composition solution with both functional and non-functional requirements is a computationally demanding problem: the time and space requirements may be insufferable due to the high number of available services. To alleviate this problem, we propose the application of a skyline operation to reduce the search space and improve the scalability.**

We design a system to solve the composition problem with two separate processes. The Graphplan approach finds a solution in a short time, the database 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. Full Solution Indexing using Database (FSIDB) approach pre-computes all services combinations and store them as paths in a database. Partial pre-composing approach chooses "popular" paths generated by FSIDB approach and store them in a separate table. If the problem can be solved by these paths, there is no need to search the table with whole paths. We evaluate our approach with a web service challenge dataset.

***Keywords*-Skyline operator; QoS-aware service composition; database**

I. Introduction

Service-oriented architecture (SOA) offers a platform for supporting on-demand software systems. The core idea of automated service composition is to select a set of appropriate services and compose them to fulfill a complex business task. As web services with similar functionality increase, researchers consider non-functional properties of web services, e.g., cost, availability, response time and throughput, to better satisfy user's requirement. Many in-memory approaches based on different techniques have been successfully employed in solving service composition problem. These include Integer Linear programming [1], beam-stack search [2] and the planning model [3]–[5]. For each user request, in-memory approaches construct a search graph and search this graph for a solution. Thus, in-memory approaches may be computationally expensive and are limited by the search space.

To deal with this challenge, we pre-process services and find a set of candidate services referred as "skyline services" based on the Skyline operator [6], [7]. 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 and design a parallel system to solve service composition problem. Our approach allows us to solve large composition problems with little storage and increased speed.

Our system relies on two subsystems:

1) When an in-memory approach is practical, we use Graphplan. The Graphplan approach constructs a planning graph and returns a solution in a short time. However, the solution returned is not the optimal one and may contain redundant services (a service is redundant if all its outputs used by other services are also produced by other services [8]).
2) Finally, a database approach relies on pre-computed service combinations stored in a relational database. When a user request comes, a SQL query is generated to query the database for an optimal solution. A shortcoming with this approach is that the time spent to search a solution may be higher than the Graphplan approach if the database stores a large number of paths.
   Different users may have same requests. Thus we store popular paths by using a Partial pre-composing approach. If the user request can be solved by these paths, this Partial-Pre-composing-Approach system can return an optimal solution without constructing a search graph.

The rest of this paper is organized as follows. Section II describes preliminary knowledge and provides the background of this paper. The architecture of the proposed system and algorithms are given in Section III. We present the experiment results in Section IV. Section V reviews related work and the conclusion is drawn in Section VI.

II. Preliminary

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. We refer to the input and output types as *concepts*. OWL-S (Web Ontology Language for Web Services [9]) files are used to define relationships between services and concepts.
- Q is a finite set of quality-of-service (QoS) values of w. The criteria for QoS are determined from users' constraints and preferences.

For illustrative purposes, we consider two criteria: response time (R) and throughput (T). Response time is the interval between the arrival of the request and the beginning of delivery the response, it is a negative criterion, the higher the value, the lower the quality [10]. Throughput is the average rate of successful message delivered per time, it is a positive criterion, the higher the value, the higher the quality.

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_s 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 1: $w_1 \prec w_2 \Leftrightarrow Q_k(w_1) \geq Q_k(w_2) \quad \forall k \in \{1, |Q|\}$ and $\exists k \in \{1, |Q|\}$ such that $Q_k(w_1) > Q_k(w_2)$. Suppose 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 another other service. For example, Table I shows a list of services and their QoS values. Suppose they have the same output, according to Definition 1, we find a skyline set $\{w_1, w_2\}$, w_3 does not belong to the skyline set because it is dominated by w_2.

Table I

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

We say that we have a QoS-aware service composition problem when we must combine several web services to satisfy both the functional requirements and the QoS constraints. We must aggregate the criteria from the various services. Services can be invoked in sequence one after the other ($w_1; w_2$), especially when the output of the first web service is needed for the second one. We can also invoke the services in parallel ($w_1 || w_2$) [11], when there is no dependencies between them. Any actual web service combination can be described as a collection of sequential calls and parallel calls. The overall response time and throughput of the service composition process can be calculated as follows:

$$R(w_1; \cdots ; w_n) = \sum_{i=1}^{n} R(w_i), \tag{1}$$

$$R(w_1 || \cdots || w_n) = \max\{R(w_1), \ldots, R(w_n)\}, \tag{2}$$

$$T(w_1; \cdots ; w_n) = \min\{T(w_1) \ldots, T(w_n)\}, \tag{3}$$

$$T(w_1 || \cdots || w_n) = \min\{T(w_1) \ldots, T(w_n)\}. \tag{4}$$

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.

We use plug-in matching degree to match services: two services can be connected if the input of a service is a subset of the output of the other service. This semantic model, borrowed from [12], is consistent with many proposed web service composition approaches, e.g., [13]–[15].

III. Architecture and algorithm

In this section, we present the framework of proposed system and algorithms.

Figure 1 illustrates an architecture overview of our system. "Web Service Repository" is a searchable repository which contains all services information. We find skyline services provided by the service repository. For each concept, we find a skyline service set among its parent services, services in this set do not dominant by each other in terms of response time and throughput. The initialization process takes user request and skyline services as inputs of the system. We use two processes to find satisfying solutions separately, a chosen solution is sent back to the user.

The searching approach plays an important role in this system. The Graphplan and database approach search solutions separately.

The Graphplan approach first constructs a planning graph from the initial states (provided in the user request). Layers of the planning graph form an alternative sequence of proposition layers P_i and action layers A_i. P layers contain concepts and A layers contain services. If the inputs of service w are satisfied in P_i layer, w can be added in A_i layer and its outputs are added in P_{i+1} layer. The graph construction stops when the goals are contained in the graph or no more services can be added in action layers. If the goals are not contained in the planning graph, the problem can not be solved. If the goals are contained in this graph, a backward search phase is carried out to find a solution. The Graphplan approach is an in-memory approach and can only work when data fits in RAM. Besides, the planning graph is constructed based on the initial states and goal, to solve N different user requests, N planning graphs are constructed, this is time consuming.

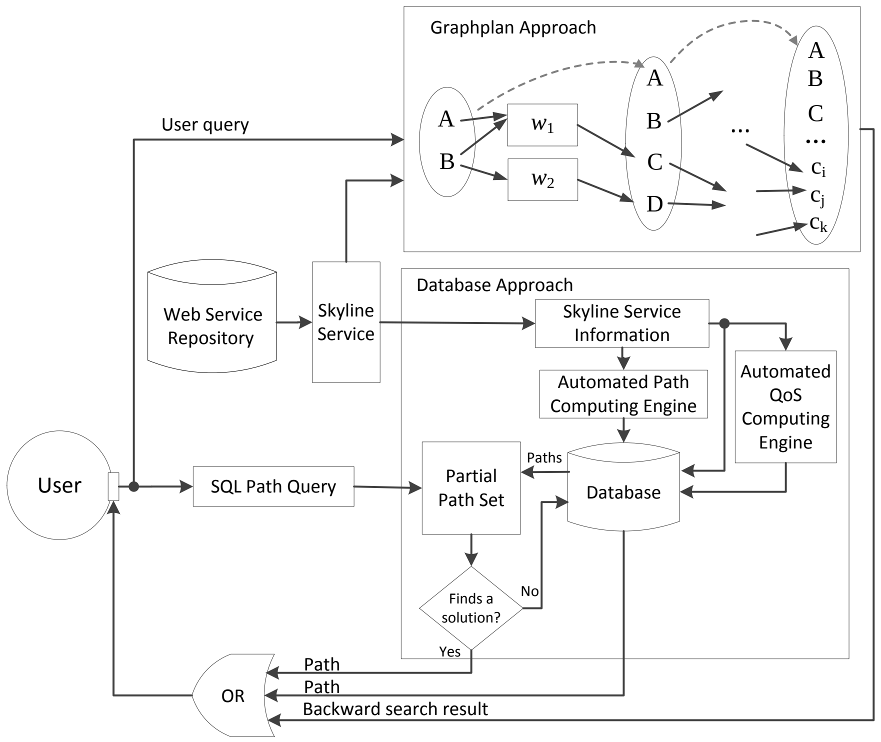

Figure 1. Architectural Overview.

The database approaches can find an optimal solution with fewer redundant services. With FSIDB approach, we generate all possible service combinations as paths and store them in a relational database. Then, when the user request comes, we compose SQL statements to query the database for paths which meet user's requirements. This approach may spend longer time in searching compared with the Graphplan approach. As different users may have same request, we user Partial pre-composing approach to pick popular paths and store them in a table (Partial Path Set). If the user request can be solved by these paths, this approach may return a solution in a very short time.

A. FSIDB approach

This approach is originally proposed in our previous work [16], in this paper, we pre-process services with skyline operators and alter the schema of the database, as a result, the searching speed increases. The information of services such as name, inputs, outputs and QoS values are stored in "Skyline Service Information" table. "Automated Path Computing Engine" computes all service combinations and store them as paths in "Path" table. Similarly, "Automated QoS Computing Engine" calculates the corresponding QoS value of each path and store this value in the database. Services used in each path are recorded with their layers in "UsedService" table.

Algorithm 1 ***PathsBuild*** repeatedly generates paths with multiple services (line 3), this process ends when no more paths can be generated (line 5).

Algorithm 1 *PathsBuild*

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

Algorithm 2 ***MulPathSetBuild*** creates new 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 3 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 5 (line 20).

Algorithm 3 ***CheckRedundant*** checks whether or not service ws can connect with 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 connect with $pathM$.

Algorithm 4 ***CreatePath*** creates a new path $path$ by connecting service ws and path $pathM$ together. The inputs of services in this new path can be provided by either

Algorithm 2 *MulPathSetBuild(i)*

```
Input: $SR, mulPathSets(i-1)$;
Output: $mulPathSets(i)$: a set of paths with $i$ services;
1: if $i = 2$ then
2:     $counter \leftarrow 1$
3:     for each service $srv$ in $SR$ do
4:         for each service $ws$ in $srv.skyline$ do
5:             $path \leftarrow createPath(ws, srv, counter, 1)$
6:             $writeService(ws, pathM, path, 1)$
7:             $mulPathSets(i) \leftarrow mulPathSets(i) \cup path$
8:             $counter \leftarrow counter + 1$
9:         end for
10:    end for
11: else
12:    $counter \leftarrow pathsSet.size$
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:                    $layer \leftarrow$ layer of $srv\_M$
18:                    $counter \leftarrow counter + 1$
19:                    $path \leftarrow$
                       $createPath(ws, pathM, counter, layer)$
20:                    $writeService(ws, pathM, path, layer)$
21:                    $mulPathSets(i) \leftarrow mulPathSets(i) \cup$
                       $path$
22:                end if
23:            end for
24:        end for
25:    end for
26: end if
27: return $mulPathSets(i)$
```

Algorithm 3 *CheckRedundant($ws, pathM$)*

```
Input: $ws, pathM$;
Output: $flag$;
1: $flag \leftarrow false$
2: if $ws \nsubseteq srv\_M.skyline$ then
3:     $flag \leftarrow true$
4: end if
5: if $ws \subset pathM.ws$ then
6:     $flag \leftarrow true$
7: end if
8: if $ws.out \not\subset pathM.in$ then
9:     $flag \leftarrow true$
10: end if
11: return $flag$
```

inputs of $path$ or outputs of services in preceding layers. The response time $resp$ and throughput thp of $path$ are calculated according to Equation (1)- Equation (3).The order of service layers of $path$ is decided by Algorithm 5.

Algorithm 4 *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:
10:    if $ws.resp > srv\_M.resp$ then
11:        $diffResp = ws.resp - srv\_M.resp$
12:    else
13:        $diffResp = 0$
14:    end if
15:    $resp = pathM.resp + diffResp$
16: end if
17: if $ws.thp > srv\_M.thp$ then
18:    $thp = srv\_M.thp$
19: else
20:    $thp = ws.thp$
21: end if
22: return $path$
```

Algorithm 5 ***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).

B. 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 solves the request and has best QoS value. We store these paths in table "Partial Path Set". Services which are not used by any of these paths are seen as non-candidate services and been pruned ($RemovedServiceSet$). To further decrease the number of used services, we find and remove services which are less used in "Partial Path Set". Also, we need to delete paths in "Partial Path Set" which use removed services ($RemovedPathSet$). For each path in $RemovedPathSet$,

Algorithm 5 *WriteService*

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

we search the database for alternative solutions, if no such solution exists, this query is removed from "Partial Path Set". If alternative solutions exist, we pick the one with best QoS value and add it into "Partial Path Set". This process is described in details in Algorithm 6. $inConcepts$ (resp. $outConcepts$) represents a set of input (resp. output) concept id, $PathSet$ represents a set of remaining paths.

Algorithm 6 *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 $PathSet$
```

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

C. Graphplan Approach

The Graphplan approach contains two stages: a forward expand stage constructs a planning graph and a backward search stage retrieves a solution. A planning graph contains two kinds of layers: the proposition (P) layers contain concepts and action (A) layers contain services. Layers of the planning graph form an alternative sequence of proposition layers and action layers. To construct this 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_0 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 action layer. If the goal can not be found in the planning graph, this means no solution exists in this composition problem, otherwise, a backward search phase is carried out to find a solution. The Backward search stage loops from the goal layer to the initial layer to extract 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.

IV. Experimental results

In this section, we give the experimental results performed to evaluate the performance of the proposed mechanism. We implement our algorithms using Java platform.

We run experiments on a computer with the following configuration: 1) CPU: Intel Core i5-2400 at 3.10 GHz, 2) Mainboard: Intel C206, 3) Memory: 8 GB DDR3 SDRAM PC3-10600, 4) Hard drive: WD2500AAKX 250GB 7200 RPM 16 MB 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. The relative standard error of the running time average is less than 5%.

A. Dataset

We use the test generator program from the WSC-2009 web-service challenge [17] to generate five datasets and evaluate our work. Each dataset contains a WSDL file which is the repository of web services. An OWL file defines the matching parameters by their semantics. WSLA file describes QoS values of services. The number of services varies from 1000 to 9000, and the number of concepts varies from 3000 to 28000 accordingly. Each web service has around 10 input and 30-40 output concepts.

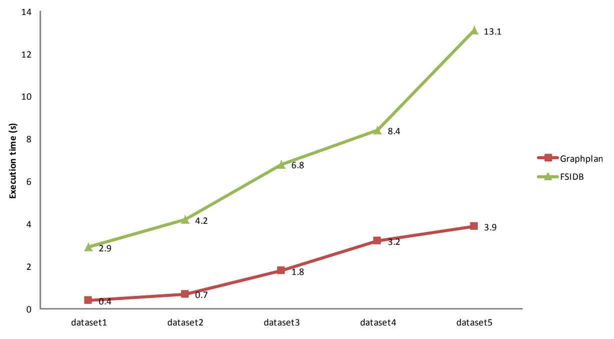

Figure 2. Time for web service composition search.

B. Performance analysis

We measure the average execution time spent to solve the composition problem, varying the number of services from 1000 to 9000. The experiment results are presented in Figure 2. Comparing the performance of Graphplan approach and FSIDB approach, we observe that Graphplan approach returns a solution in a shorter time than the FSIDB approach: on the largest dataset, it is three times as fast. Table II (resp. Table III) shows the results over solutions with optimal response time (resp. throughput). The row "#services" shows the number of services in the returned solution. Table II and table III show that, 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 database approach, paths with redundant services always have smaller competitive as they may lead to a worse QoS value. The Partial pre-composing approach returns a solution in less than 1s.

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 are used by these paths. Then, in each round, we delete services from "service_repository" which are least used by these queries. 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 3 (resp. Figure 4) show the relationship between services and queries while fetching for paths with minimum response time (resp. throughput).

V. Related work

There has been considerable volume of research on skyline analysis problem. Paper [6] 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 chose 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

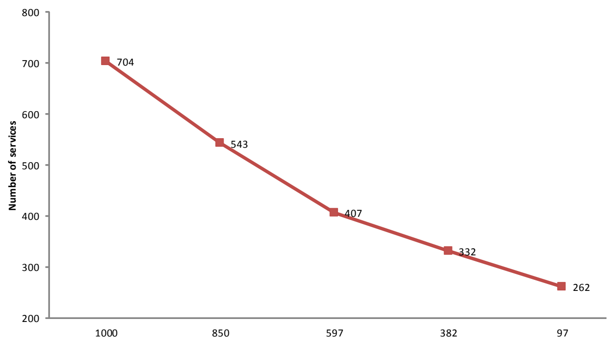

Figure 3. Remaining queries with optimal response time.

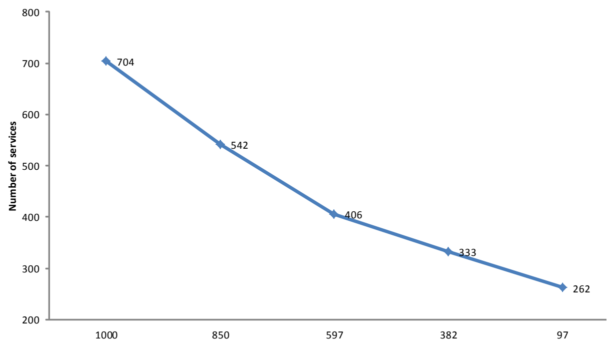

Figure 4. Remaining queries with optimal throughput.

the window. The goal of the divided and conquer algorithm is to divide the dataset into partitions so each of the partition fits into the main memory. First, the dataset is divided into m partitions by a M-way partitioning method, then, this algorithm computes the skyline of each partition using a main-memory algorithm. 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 [18]. This method also belongs to grid partitioning method, in which, they map multi-dimensional data space to a tree structured P2P network. The number of visited nodes is reduced with this technology.

Vlachou [19] *et al.* propose an angle-based space partitioning scheme that can be used in parallel skyline. The angle space partitioning technique first maps the cartesian coordinate space to hyperspherical space, then the space is partitioned 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 [7], each point is represented as a m bits vector, m is the number of points. This method may handle problems with multiple dimensions and is suitable to solve a problem with a small number of points. However, it is not suitable for dynamic datasets because a new bitmap is needed when a new point adds or

Table II
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 approach	**#services**	4	8	10	6	12
	response time[1]	760	1250	1080	600	1670
Graphplan approach	**#services**	4	9	12	10	15
	response time[1]	760	1270	1330	1010	1980

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

Table III
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 approach	**#services**	4	8	10	6	12
	throughput[1]	3000	5000	2000	6000	4000
Graphplan approach	**#services**	4	9	10	10	15
	throughput[1]	3000	4000	2000	5000	2000

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

disappears in the datasets. 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 [7].

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 [20]. They use a hierarchical clustering method to find skyline services, the basic 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 the 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. After that, they discuss how to increase services' potential so they can be included in composition applications. The one pass algorithm proposed in [21] enumerates and stores skyline service execution plans. However, non-candidates may be stored as 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 area. 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 [22]. Du *et al.* computes the composite service skyline in presence of QoS correlations [23]. 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.

VI. CONCLUSION

Taking advantage of the skyline operator, we design a system to solve QoS-aware service composition problem. Skyline analysis prunes less competitive services and reduces computational space requirement. When a user request comes, our system uses Graphplan and database approach to find solutions with two separate processes. We use two processes because we want to give user backup solutions. In the Graphplan approach, a forward expand stage constructs a search graph and a backward search stage retrieves a solution. In the database method, we pre-compute service combinations as paths and store them in a relational database. FSIDB approach and Partial pre-composing approach belong to database approaches. Partial pre-composing approach chooses popular paths generated by FSIDB approach and store them in a separate table. Suppose the problem can be solved by these chosen paths, there is no need to search the table with whole paths. Compared with the Graphplan approach, the solution returned by FSIDB approach may contain fewer services with a better QoS value, but longer time needed. In future work, we will explore the extension of our work for multiple QoS criteria.

REFERENCES

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

[2] H. Kil and W. Nam, "Efficient anytime algorithm for large-scale qos-aware web service composition," *Int. J. Web Grid Serv.*, vol. 9, no. 1, pp. 82–106, Mar. 2013.

[3] 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.

[4] 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.

[5] M. Kuzu and N. Cicekli, "Dynamic planning approach to automated web service composition," *Applied Intelligence*, vol. 36, no. 1, pp. 1–28, 2012.

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

[7] 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.

[8] 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.

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

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

[11] (2009) Web service challenge rules. [Online]. Available: http://ws-challenge.georgetown.edu/wsc09/downloads/WSC2009Rules-1.1.pdf

[12] S. Bleul, T. Weise, and K. Geihs, "The web service challenge - a review on semantic web service composition," *Electronic Communications of the EASST*, vol. 17, 2008.

[13] 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.

[14] 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*, 2015, pp. 403–410.

[15] P. Rodriguez-Mier, M. Mucientes, and M. Lama, "A dynamic qos-aware semantic web service composition algorithm," 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. 623–630.

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

[17] S. Kona, A. Bansal, M. Blake, S. Bleul, and T. Weise, "Wsc-2009: A quality of service-oriented web services challenge," in *Commerce and Enterprise Computing, 2009. CEC '09. IEEE Conference on*, July 2009, pp. 487–490.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.