Journal of Quantum Computing
DOI:10.32604/jqc.2020.015402
Article

A Database-Driven Algorithm for Building Top-k Service-Based Systems

Dandan Peng and Le Sun[*]

Jiangsu Collaborative Innovation Center of Atmospheric Environment and Equipment Technology (CICAEET), Nanjing University of Information Science & Technology, Nanjing, 210044, China

[*]Corresponding Author: Le Sun. Email: sunle2009@gmail.com

Received: 03 November 2020; Accepted: 05 December 2020

Abstract: The purpose of this work aims is to automatically build top-k (the number of suggested results) light weight service based systems (LitSBSs) on the basis of user-given keywords. Compared with our previous work, we use a score (oscore) to evaluate the keyword matching degree and QoS performance of a service so that we could find top-k LitSBSs with both high keyword matching degree and great QoS performance at the same time. In addition, to guarantee the quality of found top-k LitSBSs and improve the time efficiency, we redesign the database-driven algorithm (LitDB). We add the step of referential services selecting into the process of the LitDB, which could prioritize services with high quality (high keyword matching degree and great QoS performance). We design comprehensive experiments to demonstrate the great time performance of LitDB.

Keywords: Top-k LitSBSs; user-given keywords; database-driven algorithm

1 Introduction

Service composition is a technology of developing service-based systems (SBSs) by composing certain existing services [1]. Traditional service composition methods are too complex for non-expert users. More and more works [2,3] have been devoted to simplifying the service composition process. Meanwhile the researches [4,5] on the service composition algorithm based on keyword query came into being. These algorithms could build SBSs which reflects uses' preferences automatically, thus the non-expert users could build complex SBSs easily just provide a few keywords.

Some researches [6–8] develop several keyword-query based algorithms to find light weight SBSs (LitSBSs). Compared with other SBSs that satisfy certain query keywords, LitSBS has the least number of component services. In addition, compared with non-LitSBS, LitSBS is easy to managed, executed, monitored, debugged, deployed and scaled. There are some keyword-based algorithms using relational databases (DB) to store services since the query techniques of DB are mature and robust. In addition, using DB could guarantee the time efficiency of query functions.

Although many works use DB store services, there are also some problems. For example, in the works [9,10], both solutions of them store all possible SBSs in the database in advance requiring a lot of time and storage space. About the above issue, we proposed a database driven algorithm to build LitSBSs in our previously work [11]. We first evaluate keyword matching degree for each candidate service, and use utility function to calculate keyword matching score (*kscore*) for them. After that we design service composition algorithm to find top-k LitSBSs. Finally, we resort these top-k LitSBSs by calculating their QoS quality score (*qscore*). However, there is a problem in our previous work is that we cannot find top-k LitSBSs that meets both keyword matching degree and QoS quality at the same time, in other words, we have to split it into two separate processes: (1) Finding top-k LitSBSs with highest *kscore*s. (2) And resorting these top-k LitSBSs by their *qscore*s.

In this work, we integrate *qscore* and *kscore* to a new evaluating score *oscore*, therefore the step of QoS performance ranking could be canceled. In addition, in order to find top-k high-quality service compositions that meet users' needs, we propose the concept of preferential services. In the step of matched keyword table generating, we select the preferential services which with high *oscore*s to ensure that they can be composed first.

The particular contributions we do in this work are as follows:

- *Oscore is used* to integrate *kscore* and *qscore*, so that guarantee the QoS and keyword matching degree of the founded LitSBSs.
- We add the preferential services selecting in the process of matched service table generating, the aim is to give priority to the advantageous services.
- Extensive experiments are conduced to illustrate the great time performance of the fast service composition algorithm.

The structure of this paper is as follows: Section 2 clearly defines the issue we are willing to solve; Section 3 details the LitDB algorithm; Section 4 designs the experiments and illustrates the results; and Section 5 summarizes the full text.

2 Problem Definition

2.1 Service Database

We design a service database to represent service library. We call the database service database *L,* which includes 4 tables with different style: The service table (*TS*), the input table (*TI*), the output table (*TO*) and the parameter table (*TP*). All the details could be found in our previous work [11].

2.2 The Keyword Matching Score

We set keywords matching score *kscore* to measure the matching degree between certain query keywords and service.

$$\mathrm{kscore}(\mathrm{s},\mathrm{E_s},\mathrm{q_r}) = \sum_{\forall \mathrm{e_i}\in \mathrm{E_s}} \mathrm{kscore}(\mathrm{s},\mathrm{e_i},\mathrm{q_r}) \tag{1}$$

where $\mathrm{kscore}(\mathrm{s},\mathrm{e_i},\mathrm{q_r})$ is defined by Formula 2.

$$k\mathrm{score}(\mathrm{s},\mathrm{e_i},\mathrm{q_r}) = \frac{1+\ln(1+\ln \mathrm{tf})}{(1-\sigma)+\sigma\frac{\mathrm{dl_i}}{\mathrm{avdl}}}\cdot \ln\frac{\mathrm{N}+1}{\mathrm{df}} \tag{2}$$

where, tf is the frequency of $\mathrm{q_r}$ in $\mathrm{e_i}$; df is the number of tuples in $\mathrm{E^i}$ containing keyword $\mathrm{q_r}$ and $\mathrm{e_i}$ is the value union of the ith attribute of services in TS; $\mathrm{dl_i}$ is the character number in $\mathrm{e_i}$; $\mathrm{avdl} = (\mathrm{dl_1} + \cdots + \mathrm{dl_m})/\mathrm{m}$; N is the number of services in TS; and σ is a constant (usually 0.2).

2.3 The Quality Score of a Service

The following basic formula is a score function evaluating the overall QoS performance of a service.

$$\mathrm{qscore}(\mathrm{s}) = \mathrm{U}(q_1(\mathrm{s}),\dots,q_l(\mathrm{s})) = \sum_{k=1}^{l} \mathrm{U}(q_k(\mathrm{s}))\cdot \omega_k \tag{3}$$

where, U is a multi-objective value function, and $\omega_k \in [0,1], \forall k \in [1,..,l], \omega_1 + \cdots + \omega_l = 1$.

For a positive QoS attribute of service s which belongs to a matched keyword table (We will discuss it detailly in Section 4.1) $\mathrm{K_j}$, the value is calculated by Formula 4.

$$\mathrm{U}(q_k(\mathrm{s})) = \frac{q_k(s)-Q_{\min(j,k)}}{Q_{\max(j,k)}-Q_{\min(j,k)}} \tag{4}$$

where, for a negative QoS attribute of service s which belongs to MT $\mathrm{K_j}$, the value is calculated by Formula 5.

$$\mathrm{U}(q_k(\mathrm{s})) = \frac{Q_{max}(j,k)-ql(s)}{Q_{\max(j,k)}-Q_{\min(j,k)}} \tag{5}$$

2.4 Integrated Score

oscore measures the overall performance of a service by considering both its keyword matching degree and QoS performance.

$$\mathrm{oscore} = \mathrm{f}(\mathrm{kscore}, \mathrm{qscore}) \tag{6}$$

where, f is an aggregation function of *kscore* and *qscore*. For example, we can use the weighted average to aggregate *kscore* and *qscore*, or define f as: $\mathrm{oscore} = \mathrm{kscore} + \mathrm{qscore}$.

2.4 Definition of Top-k LitSBS

Definition 1 *Given a set of query keywords Q, and a service table TS, a lightweight SBS (LitSBS, represented by $\tilde{S}$) matching Q is an SBS that (1) $\tilde{S} \subset TS$; (2) the attribute values of $\tilde{S}$ contains Q; and (3) for any non-lightweight SBS S with kscore(S,Q) = kscore($\tilde{S}$,Q), the number of component services in S (represented by $|\mathrm{S}|$) is larger than $|\tilde{S}|$.*

Definition 1 defines the light weight SBS.

Definition 2 *Assume each service in a service database $\mathrm{L} = \{\mathrm{TS}, \mathrm{TI}, \mathrm{TO}, \mathrm{TP}\}$ has l QoS parameters $\{p_1, \ldots, p_l\}$. Given a set of keywords Q and a set of QoS constrained conditions $\mathrm{C} = \{c_1, \ldots, c_l\}$, the **top-k LitSBSs** are k services (in L) with the highest kscores matching to Q, and satisfy all QoS constraints C.*

Definition 2 defines the top-k LitSBSs.

To get top-k LitSBSs, we design an algorithm automatically querying service database and then recommending top-k LitSBSs to users, which is based on the input query keywords and QoS constraints.

3 LitDB: An Algorithm for Building Top-k LitSBSs

A database-driven algorithm (called LitDB) is proposed to efficiently build top-k LitSBSs according to user-given keywords. Fig. 1 shows the process of this algorithm. The LitDB includes three main stages: (1) The step of keyword matching is to search for services containing certain query keywords. (2) Matched service table generating builds a service matched table (MT) for each query keyword, which means all found services are related to a certain query keyword will be put into its MT. In addition, in each MT, those preferential services will be selected and sorted in descending order according to *oscores* firstly. Then those non-preferential services will be sorted behind those preferential services. (3) Service composition algorithm will find the top-k LitSBSs quickly and efficiently.

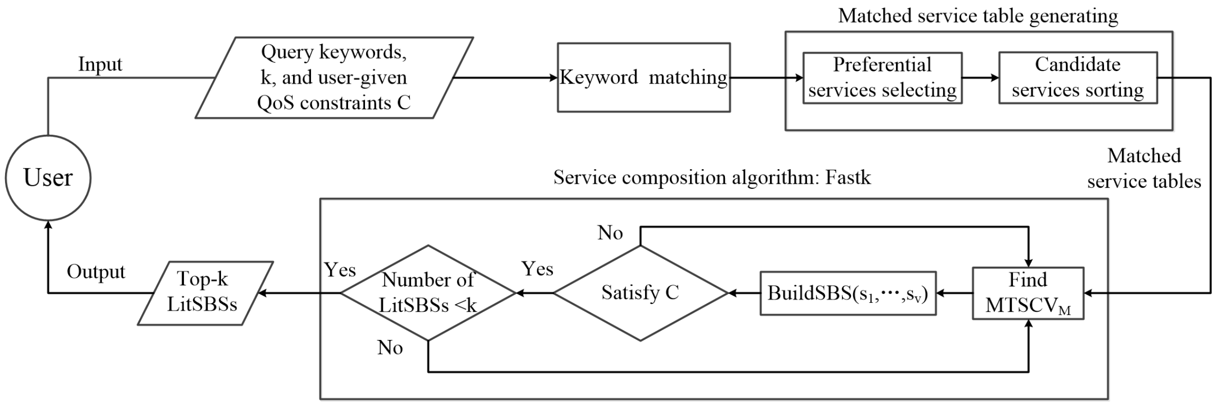
Matched service table generating
Input
Query keywords, k, and user-given QoS constraints C
Keyword matching
Preferential services selecting
Candidate services sorting
User
Matched service tables
Service composition algorithm: Fastk
No
Output
Top-k LitSBSs
Yes
Number of LitSBSs <k
Yes
Satisfy C
BuildSBS(s1,⋯,sv)
Find MTSCVM
No

Figure 1: The whole steps of LitDB

3.1 Matched Service Table Generating

According to the *oscores*, several matched keyword tables (MT) are generated for certain query keyword. For instance, Tab. 1 shows the MTs for certain query keywords: Car hire, Flight and Insurance quote. Services in a MT are ranked in descending order according to *oscores*.

Table1: Matched keyword tables for query keywords

K_1: Car hire		K_2:Flight		K_3:Insurance quote	
SID	*oscore*	SID	*oscore*	SID	*oscore*
S_1	6	S_5	3.7	S_3	7.6
S_2	4.1	S_4	0.2	S_7	3.4
S_6	1.8				

3.1.1 Preferential Services

qscore is design to measure the QoS performance of a service and *kscore* is to evaluate the matching degree between a query keyword and a service. The higher the *kscore* and *qscore* of a service which means the better this service matches the query keywords as well as the better its QoS performance. As is shown in Fig. 2, each service is presented as point in 2-dimensional space. We can see that service a is a preferential service, because there is no other service that has both higher *qscore* and *kscore* than a. Similarly, Service b, c, d has this condition too, therefore b, c, d are also preferential services. Since preferential services have better performance than other services, to save time, we should guarantee that they will be selected firstly to be composed.

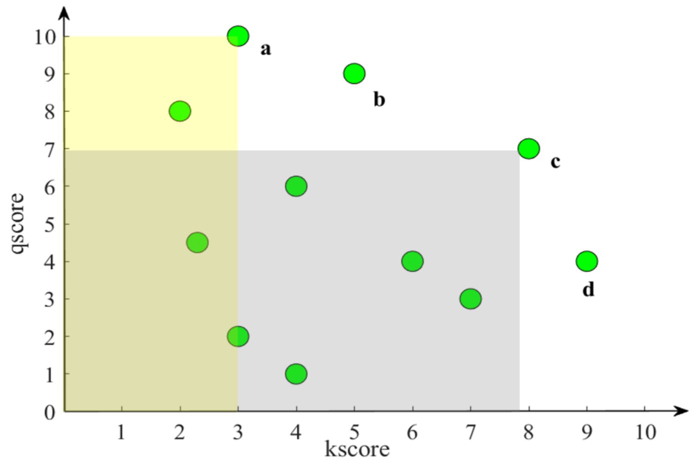

Figure 2: Example of preferential services

Before generating a MT for each keyword, we should select preferential services first, then these preferential services in the MT will be sorted in descending order according to *oscore*s. Finally other services will be ranked behind the preferential services also in descending order according to *oscore*s.

3.2 Service Composition Algorithm

We develop three service composition algorithms searching for top-k LitSBSs that matching Q among the MTs we mentioned before. We first introduce the most basic algorithm called Intuitive. Then we introduce other algorithm called Enhanced with better time performance than Intuitive by adding the pruning strategy. Finally, the most useful one called Fastk will be introduced, which has better time performance than the former two.

3.2.1 Intuitive Algorithm

Intuitive is the most basic algorithm, which is on the basis of exhaustive search. Algorithm 1 shows the process of Intuitive. We put combination of services obtained by searching the MTs into *BuildSBS* function in order to check whether the combination could be composed or not. If a combination of services can be composed, we put the service composition in R. When the search finished, we rank all LitSBSs in R in descending order by calculating their *oscore*s. Finally, we get the top-k LitSBSs in R with highest *oscore*s.

```
Algorithm 1 Intuitive algorithm
Input: $Q, k, K_1, ..., K_v, C$
Output: top-k LitSBSs
1: $R$: queue for possible top-k LitSBSs. LitSBSs in $R$ are always sorted in descending
   order in terms of the $oscore$.
2: for each combination $\{s_1, ..., s_v\}$ ,where $\forall s_i \in K_i$ do
3:     $\widetilde{S} = BuildSBS\ (s_1, ..., s_v)$
4:     if $\widetilde{S}$ satisfies $C$ then
5:         Add $\widetilde{S}$ to $R$
6:     end if
7: end for
8: Calculate $oscores$ of all LitSBSs in $R$
9: return top-k LitSBSs in $R$ with the highest $oscores$
```

Function *BuildSBS* (Algorithm 2) is used to check whether a set of services can be composed or not. We use Γ to limit the number of the component services in an SBS in order to avoid wasting too much time on searching for an SBS with very huge size. In this algorithm, an expansion rule is design to expand service composition: If a service s_i in TS could be combined with a service in a service composition SC, then SC will be expanded. We could use SQL query to search a service s_j. In this function, the result we get must with the smallest size of all the results.

```
Algorithm 2 $BuildSBS(s_1, ..., s_v)$
Input: $\{s_1, ..., s_v\}$, $\Gamma$, $TS$, $TI$, $TO$, $TP$
Output: the most lightweight SBS (a LitSBS)
1: $LS$: a list stores service compositions, service compositions in $LS$ are always sorted
   in ascending order of $\Gamma$
2: Add $s_1$ to $LS$
3: while $|LS| > 0$ do
4:     $SC$=$LS$.pop(0)
5:     if $SC$ contains $\{s_1, ..., s_v\}$ then
6:         return $SC$
7:     end if
8:     if $|SC| < \Gamma$ then
9:         for each service $s_i$ in $TS$ can be combined with a service $s_j$ in $SC$ to form a
           $SJP$ do
10:            $\widetilde{SC}$=$SC$.add($SJP$) and move $\widetilde{SC}$ to $LS$
11:        end for
12:    end if
13: end while
14: return null //(There is no LitSBS contains services $\{s_1, ..., s_v\}$)
```

3.2.1 Enhanced Algorithm

Enhanced algorithm is proposed to improve the time efficiency of Intuitive algorithm. Compared with Intuitive algorithm, Enhanced algorithm use an upper bound to improve time efficiency. We call the upper bound the most top service composition value (MTSCV) defined by the definition 3. Alg. 3 shows the whole process of the Enhanced. It first finds k LitSBSs, it then calculates $MTSCV_M$ for the remaining tuples in K_i. If the current $MTSCV_M$ is no larger than the lowest *oscore* of the found k LitSBSs, the algorithm will stop and return the k LitSBS; or else, it will continue searching the LitSBSs based on the remaining tuples.

Algorithm 3 Enhanced algorithm

```
Input: $Q, k, K_1, ..., K_v, C$
Output: top-k LitSBSs
1: $R$: possible top-k LitSBSs. LitSBSs in $R$ are always sorted in descending order of
   $oscore$.
2: while $|R| \le k$ do
3:   for each combination $\{s_1, ..., s_v\}$ ,where $\forall s_i \in K_i$ do
4:     $\widetilde{S} = BuildSBS\ (s_1, ..., s_v)$
5:     if $\widetilde{S}$ satisfies $C$ then
6:       Add $\widetilde{S}$ to $R$
7:     end if
8:   end for
9: end while
10: for each combination $\{s_1, ..., s_v\}$ ,where $\forall s_i \in K_i$ do
11:   $MTSCV_M$=max $MTSCV_i$
12:   if $MTSCV_M > R[R.size() - 1].oscore$ then
13:     $W = BuildSBS\ (s_1, ..., s_v)$
14:     if $W$ satisfies $C$ then
15:       Add $W$ to $R$
16:     end if
17:     if $W.oscore > R[R.size() - 1].oscore$ then
18:       $R[R.size() - 1] = W$
19:     end if
20:   end if
21: end for
22: return $R$
```

Definition 3 *Assume each query keyword in* $\mathrm{Q} = \{q_1, \dots, q_v\}$ *has a MT in* $\mathrm{K} = K_1, \dots, K_i, \dots, K_v$*, and the k LitSBSs (may not be top-k) are already found by joining the tuples in* $K_1, \dots, K_i, \dots, K_v$*. For each* K_i*, the remaining tuples (not contained in the k LitSBSs) are sorted in a queue* R_i *in the descending order according to their oscores. The* $MTSCV_i$ *for an MT* K_i $(\forall i \in 1, \dots, v)$ *is defined as follows:*

$$MTSCV_i = K_1.top.oscore + \cdots + h(K_i).oscore + \cdots + K_v.top.oscore \quad (7)$$

where $\mathrm{K_i.top}$ represents the tuple with highest oscore in $\mathrm{K_i}$. $\mathrm{h(K_i)}$ represents the top unprocessed tuple in $\mathrm{K_i}$. The maximum $\mathrm{MTSCV_M}$ is defined by Formula 8.

$$MTSCV_M = \max_{i=1}^{v} MTSCV_i \quad (8)$$

The $MTSCV_M$ represents the upper bound of the *oscore*s of all possible LitSBSs in the remaining tuples (i.e., excluding the tuples in the found k LitSBSs). The inputs of Algorithm 3 are query keywords Q, k, a set of MTs $(K_i, \dots, K_v)$. R keeps the possible LitSBSs, which are always sorted in descending order according to *oscore*. It first finds k LitSBSs (lines 2–9 in Algorithm 3). Then for those services waiting to be composed, we calculate their boundary $MTSCV_M$. If their $MTSCV_M$ exceed the lowest *oscore* of all the found k LitSBSs in R, the *BuildSBS* function is used to check whether they could be composed or not (lines 10–16 in Algorithm 3). After that, if combination's *oscore* is higher than the one of the found k LitSBSs, the LitSBS with the lowest *oscore* will be replaced (lines 17–18 in Algorithm 3). At last, we get top-k LitSBS in R.

3.2.1 Fastk Algorithm

Finally, we design Fastk has better time performance than Enhanced. Algorithm 4 shows the whole process of Fastk algorithm. The inputs of Algorithm 4 are query keywords Q, k, and a set of MTs $(K_1, \dots, K_V)$. We set a stack $\mathrm{N}(K_i)$ for each K_i to store the processed tuples of K_i. R is established to store the possible top-k LitSBSs, which are always sorted in descending order according to LitSBSs' *oscore*s. P is set to keep the final top-k LitSBSs, besides the top unprocessed tuple of K_i is stored in $\mathrm{h}(K_i)$. After that, we check whether the top tuple of each K_i could be composed or not. If they could be composed, a LitSBS will be built and put into R (line 4 in Algorithm 4). And then we calculate $MTSCV_i$ of each service in $\mathrm{h}(K_i)$, $\forall \mathrm{i} \in 1, \dots, \mathrm{v}$, and move the service with the highest MTSCV to $\mathrm{N}(K_M)$ (lines 9–10 in Algorithm 4). Then for each service combination, we use *BuildSBS* function to check whether they can be composed or not. (lines 11–12 in Algorithm 4). Finally, if those LitSBSs with *oscore*s ≥ the current $MTSCV_M$ in R will be moved to P. (line 14 in Algorithm 4). What is more, the previous operations in lines 9–14 in Algorithm 4 will be repeated until P contains k LitSBSs.

Algorithm 4 Fastk algorithm

```
Input: $Q, k, K_1, ..., K_v, C$
Output: top-k LitSBSs
1: $N(K_i) = \emptyset$: processed tuples of $K_i$
2: $f(K_i) = \emptyset$: top unprocessed tuple of $K_i$
3: $R = \emptyset$: possible top-k LitSBSs. LitSBSs in $R$ are always sorted in descending order
   of $oscore$.
4: Add LitSBS = $BuildSBS\ (f(K_1), ..., f(K_v))$ to $R$
5: move $f(K_i)$ to $N(K_i)$ for $\forall i \in \{1, ..., v\}$
6: while $|P| \leq k$ do
7:   Get tuple $s = f(K_M)$, where $MTSCV_M = max_{i=1}^{v} MTSCV_i$ (Calculate
     $MTSCV_i$ by using Formula 7.
8:   Add $s$ to $N(K_M)$
9:   for each group $(s_1, ..., s_{M-1}, s_{M+1}, ..., s_v)$ do
10:    $\widetilde{S} = BuildSBS\ (s_1, ..., s_{M-1}, s, s_{M+1}, ..., s_v)$
11:    if $\widetilde{S}$ satisfies $C$ then
12:      Add $\widetilde{S}$ to $R$
13:    end if
14:  end for
15:  Move the LitSBSs with $oscore \geq MTSCV_M$ in $R$ to $P$
16: end while
17: return $P$
```

4 Experiment

Performance testing of both Fastk and Enhanced is conducted in this section on the basis of WSC-2009-web challenge datasets [9]. The device we use to run certain experiments is a server with a core CPU at 16 GB RAM and 2.60 GHZ, running Windows10 x64 Enterprise. We perform each experiment five times to obtain the average execution time.

4.1 Dataset

The time efficiency is evaluated pertaining to the value of k (from 1 to 10) and the number of services (from 1000 to 9000 in five datasets). Each test is conduced five times to get the average performance of these two algorithms. The 5 datasets we use are from WSC-2009-web challenge datasets [12]. What is more, each service contains several information including service ID, four or five output and input parameters, service name and two QoS properties including response time and throughput. We will use all the information into our experiments. Since our main purpose is to compare the time efficiency of Fastk with that of Enhanced, we simulate *oscore*s of each query keywords for each dataset by randomly setting these *oscore*s in $\{1,2,3,\ldots,10\}$ instead of practically calculating them. Based on these *oscore*s, we create matched service tables (MTs) for each query keyword.

4.2 Time Efficiency of Fastk

4.2.1 Effect of Ns (Numbers of Services)

We set k = 1 and the number of query keywords (Nq) = 2 to compare the time performance of the Fastk with that of the Enhanced when Ns varies from 1000 to 9000. The average time is shown in Fig. 3. The average execution time of both algorithms rises as the Ns increases. When Ns rises from 1000 to 7000, the average execution time of Fastk gradually becomes faster than that of the Enhanced. In addition, during this interval, both average execution time increase slowly. While when Ns ranges from 7000 to 9000, both two average execution time rise dramatically. When Ns = 9000, the average execution time of the Enhanced is almost 3 times as much as that of the Fastk.

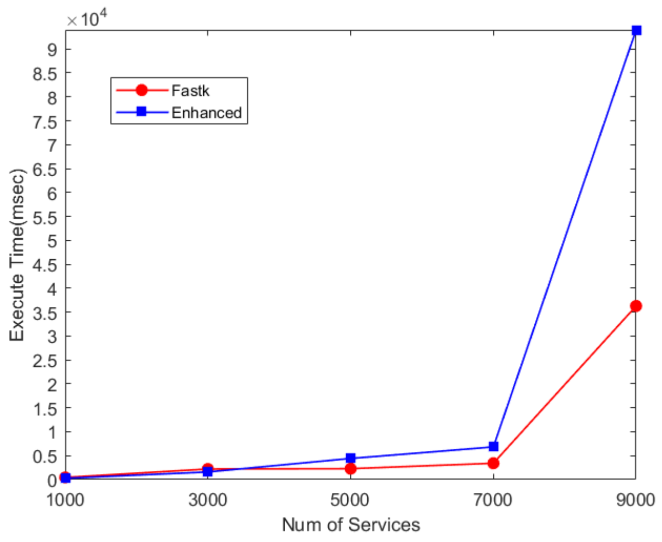

Figure 3: Number of Services

4.2.2 Effect of k

We fix Ns = 1000, Nq = 2 to test the time performance of the Fastk and the Enhanced as k varies from 1 to 15. The average execution time is shown in Fig. 4. When k increases from 1 to 9, the average execution time of both two algorithm change dramatically. While when k rangess from 10 to 15, the average execution time of both two algorithm change slightly. On the whole, the trend of average execution time is increasing.

From the above result we can see, compared with Enhanced, Fastk has greater time efficiency performance with the rising of number of services. Fig. 3 illustrates that Fastk has greater time performance than Enhanced when the number services rises. While when k changes, the difference of time performance between Fastk and Enhanced is not clear.

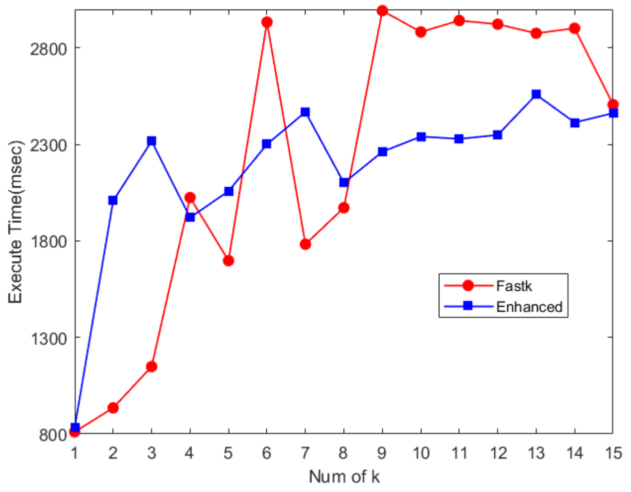

Figure 4: Value of k

5 Conclusion

We integrate the *kscore* which is used to evaluate the keyword matching degree and *qscore* evaluating the QoS of a service to a new score *oscore*. Thus we could find the top-k LitSBSs with both high keyword matching degree and great QoS performance at the same time. In addition, we redesign the database-driven algorithm by adding the step of preferential services selecting. This step guarantees the the services which with high quality will be considered preferentially in the whole process of the algorithm. The experimental results show the effectiveness of our algorithm.

Acknowledgement: We are grateful to the peoples for the support and encouragement.

Funding Statement: The author(s) received no specific funding for this study.

Conflicts of Interest: The authors declare that they have no conflicts of interest to report regarding the present study.

References

[1] Q. He, R. Zhou, X. Zhang, Y. Wang, D. Ye *et al.,* "Keyword search for building service-based systems," *IEEE Transactions on Software Engineering*, vol. 43, no. 7, pp. 658–674, 2016.

[2] V. Hristidis, Y. Papakonstantinou and L. Gravano, "Efficient IR-style keyword search over relational databases," in *Proc. of the 29th Int. Conf. on Very Large Data Bases*, Berlin, Germany, pp. 850–861, 2003.

[3] L. Sun, J. Ma, H. Wang, Y. Zhang and J. Yong, "Cloud service description model: An extension of USDL for cloud services," *IEEE Transactions on Services Computing*, vol. 11, no. 2, pp. 354–368, 2018.

[4] W. Yao, J. He, H. Wang, Y. Zhang and J. Cao, "Collaborative topic ranking: leveraging item meta-data for sparsity reduction," in *Twenty-Ninth AAAI Conf. on Artificial Intelligence*, Austin, Texas, USA, pp. 374–380, 2015.

[5] D. Yu, L. Zhang, C. Liu, R. Zhou and D. Xu, "Automatic web service composition driven by keyword query," *World Wide Web*, vol. 1, no. 1, pp. 1–28, 2020.

[6] S. Wang, A. Zhou, R. Bao, W. Chou and S. S. Yau, "Towards green service composition approach in the cloud," *IEEE Transactions on Services Computing*, vol. 1, no. 1, pp. 1–10, 2018.

[7] P. Rodriguez-Mier, M. Mucientes and M. Lama, "A dynamic QoS-aware semantic web service composition algorithm," in *Int. Conf. on Service-Oriented Computing*, Shanghai, China, pp. 623–630, 2012.

[8] M. Chen and Y. Yan, "Redundant service removal in qos-aware service composition," in *Int. Conf. on Web Services 2012*, Honolulu, HI, pp. 431–439, 2012.

[9] S. Kona, A. Bansal, M. B. Blake, S. Bleul and T. Weise, "WSC-2009: A quality of service-oriented web services challenge" in *2009 IEEE Conf. on Commerce and Enterprise Computing*, Vienna, Austria, pp. 487–490, 2009.

[10] 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.

[11] D. Peng, L. Sun and R. Zhou, "Fast build top-k lightweight service-based systems," in *Int. Conf. on Web Information Systems Engineering*, Amsterdam, Netherlands, pp. 516–529, 2020.

[12] P. Rodriguez-Mier, M. Mucientes and M. Lama, "Hybrid optimization algorithm for large-scale QoS-aware service composition," *IEEE Transactions on Services Computing*, vol. 10, no. 4, pp. 547–559, 2015.