A Service Composition Approach Based on Pre-joined Service Network in Graph Database

Abstract—We solve the service composition problem with plug-in semantic matching in a graph database. We present a Pre-joined Service Network (PJSN) approach which firstly constructs and stores a service composition network with all services and compositions in a graph database. Then, this approach fetches a satisfying solution by converting the composition request into Cypher and querying the graph database. We evaluate the performance of the proposed PJSN approach by conducting experiments and comparing with that of the Pre-joined Semantic Indexing Graph (PJSIG) method. The experiment results show that compared with the PJSIG method, the proposed approach can always find a solution and lead to higher user's satisfaction.

Index Terms—Graph Database; Service Composition; QoS

I. INTRODUCTION

Nowadays, more and more users turn to the Internet for ready-to-use web services. Service publishers release services on the Internet, users find services in the service repository and follow the pay-per-use model to use services. The service composition problem is to combine different web services to meet users' complex business requirements which cannot be finished by a single service. As more and more services with similar functions are available on the Internet, solving a service composition problem becomes a NP hard problem.

Researchers introduce artificial intelligence methods such as Integer Linear programming (ILP), A*, genetic algorithms and so on to solve composition problems.

At present, most of the existing service composition approaches are based on memory. That is, the process of searching solutions is mainly performed in memory of a computer. However, with the rapid development of cloud computing and internet technologies, a vast amount of services are published on the Internet, which brings the challenge of memory space explosion. To calculate a solution, each time, all services' information is loaded into memory first, and then build a service composition model in the memory. When the number of services grows tremendously, the demand of memory space for calculating solutions grows explosively.

To solve this challenge, researchers solve the composition problem with a relational database with larger storage space. Lee et al. use joins and indices to connect services and handle services as single input/output (PSR). To solve a problem with multiple inputs/outputs, they find and return all paths which satisfy part of the request. Li et al. [1] propose a full solution indexing database-based method (FSIDB) to compose services. In this method, a relational database is used to generate, store, and search composition solutions. However, both PSR and FSIDB use cross-table join operations, which slows down the processing speed.

Graph database is a type of NoSQL database designed to handle big data [2]. Silva et al. add relationships into graph database to generate compositions and optimize QoS using genetic programming [3]. By using edges to replace join operations, composing services in a graph database would be more efficient and faster than that is in a relational database [3]. However, this approach only optimizes a specific given task in advance. Li et al. propose a graph database-based model called pre-joined semantic indexing graph (PJSIG) which uses the shortest bidirectional breadth-first and Dijkstra algorithms to find composition solutions [4]. But, this approach fails to support the plug-in matching to compose services.

In this paper, motivated by research [4], we propose a novel service composition model named pre-joined service network (PJSN) in graph database. Specifically, The key contributions of our work are as follows:

- We use a graph database to solve service composition problem, services are composed with plug-in matching.
- We fetch a satisfying solution by converting the composition request into a graph database query.

The rest of this paper is organized as follows: Section II introduces related work. We review background knowledge in Section III. The framework of our proposed PJSN approach is given in Section IV. To illustrate the framework, we give a simple but meaningful example in Section V. Section VI provides the structure and algorithms. Section VII shows the experimental results. The conclusion is drawn in Section VIII.

II. RELATED WORK

In this part, we firstly discuss current work in graph-based in-memory approaches. Then, we describe relational and graph database approaches of services composition.

A. Graph-based In-memory approaches

Graph-based approaches generate an entire composition graph by selecting and combining relevant services. Zheng and Yan study and solve the service matching problem with a planning graph [5]. Experimental results show their approach can solve a problem in polynomial time, but with redundant services. They also propose strategies to prune redundant services, including avoid adding a service whose outputs exist in the planning graph, and so on. Lin et al. present a goal-driven mechanism to find and recommend optimal solutions [6]. Service Threshold is introduced to reduce redundant solutions and increase the search speed. Paper [7] presents
a graph-based Particle Swarm Optimisation approach which generates composition solutions and performs service selection simultaneously. Paper [8] proposes a GraphEvol approach which uses natural Directed Acyclic Graph (DAG) to represent and evolve composition solutions.

A recent and interesting graph-based approach is proposed in [9]. In this paper, the authors present a hybrid algorithm which generates a composition graph and reduces the search space by identifying dominated services. Another interesting graph-based approach is presented in [10]. This paper presents a parallel algorithm to solve maximal bicliques from a graph of composition. Takada proposes to search for BPEL fragments instead of individual web services [11]. Time spent on searching is reduced by matching activities in the query and the BPEL document. As a result, BPEL documents are combined into a graph and stored in a graph database.

B. Database-based approaches

Zeng et al. firstly present a matching algorithm (SMA) to match services on WordNet, and then propose a composition algorithm named Fast-EP [12]. Finally, a Web services search engine called WSIS is developed. Paper [1] propose a system algorithm named Fast-EP [12]. Finally, a Web services search engine called WSIS is developed. As a result, BPEL documents are combined instead of individual web services [11]. Time spent on search-space by identifying dominated services. Another interesting graph-based approach is proposed in [8] to solve the composition problem, the join operator can be avoided and as a result, the search speed increases.

III. PRELIMINARY

In this section, we give preliminary knowledge of our paper.

Definition 1. A web service \( w \) is a tuple \((w_{in}, w_{out}, Q)\) with the following components:

- \( w_{in} \) is a finite set of typed input parameters of \( w \). A web service is invoked only when all its input parameters are satisfied.
- \( w_{out} \) is a finite set of typed output parameters of \( w \).
- \( Q \) is a finite set of non-functional criteria for \( w \).

Usually, service descriptions include functional and non-functional features, in which non-functional features—QoS criteria determine the usability and utility of services. In this paper, we consider cost \((C)\), response time \((R)\) and throughput \((T)\) as QoS criteria, which are described in Table I.

Definition 2. A web service composition problem can be represented by a tuple \((S, C_{in}, C_{out}, Q)\) as follows:

- \( S \) 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.

According to [13], In this paper, we mainly focus on two basic models in service composition executive path: sequence and parallel. Given services \( w_1, w_2, \ldots, w_n \), services in a sequence control structure are invoked one by one \((w_1; w_2; \ldots; w_n)\), and services in a flow control structure are invoked in parallel \((w_1 || w_2 || \ldots || w_n)\).

We may use a single QoS dimension to express the performance of a composition and compare QoS values of different compositions by either one of the criteria, the cost \((C)\), response time \((R)\) and throughput \((T)\) of a service composition are calculated as shown in Table II.

In this research, we use semantic matching based on Web Ontology Language (OWL) to analyze relations among concepts of services [13]. Input and output parameters of web services are instances of concepts. In this paper, we use plugin matching to match services as shown in Definition 3.

Definition 3. Plug-in: If an output parameter \( out_w \subseteq w_{out} \) of service \( w \) is sub-concept of an input parameter \( in_{w'} \subseteq w_{in} \) of service \( w' \); formally, \( T \models out_w \subseteq in_{w'} \).

Definition 4. A service network graph is a directed graph represented as \( SNG = \langle C, S, E \rangle \). \( C \) is the set of input or output parameters of all services, each \( c \in C \) is a node, and \( \forall c \in C \models \exists w \in w_{in} \lor \exists c \in w_{out} \); \( S \) is the set of all services, each \( w \in S \) is a node containing properties of services, and \( \forall w \in S \{ \forall \in \in w_{in} \land \forall out \in w_{out} \land \forall \in \in \in \in \} \}; \( E \) is the set of directed edges, each of which is from a parameter to a service \( \forall (c, w) \in E \{ \exists c \in w_{in} \} \), or from a service to a parameter \( \forall (w, c) \in E \{ \exists w \in w_{in} \}. \)

Definition 5. A path of a service network graph \( SNG = \langle C, S, E \rangle \) is a sequence \( \langle \text{Start}, w_1, c_1, w_2, \ldots, w_k, c_k, w_{k+1}, \ldots, w_{n-1}, c_{n-1}, w_n, \text{End} \rangle \), where \( \text{Start} \in C \) is an input parameter of the request, and \( \text{End} \in C \) is an output of the goal, \( 1 \leq k \leq n - 1 \), \( w_k \in S \), \( c_k \in C \), \( (w_k, c_k) \in E \), \( (c_k, w_{k+1}) \in E \).

IV. ARCHITECTURE

In this section, we illustrate the framework of our proposed approach PJSN in Figure 1. It consists of three major modules: Preparation, Service Network Generation, and Path Query.

Preparation: In this module, web service information and OWL ontologies published by the service providers are analyzed. Initially, Services with functional properties and QoS values are loaded into the “Service Repository” file. Service ontologies and semantic relationships are stored into the OWL ontology file. Then, the information of services necessary for composition is parsed, associated, and stored in the “Service Information” component.

Service Network Generation: In this module, a service composition network that includes all services and compositions of services based on semantic matching are constructed, stored in the Neo4j graph database. “Preprocessing” component follows algorithms 1, 2 to construct and store the
service composition network into the graph database. Specifically, this component constructs service nodes, concept nodes, and connections between them. For services with semantic matching relationships, this component creates corresponding connections between services and concepts.

Path Query: In this module, an optimal solution of service composition that meets user’s functional and QoS requirements is calculated. The “Searching” component searches for composition solutions, which consists of two components: “Service Composition Planning” calculates possible solutions satisfying user’s functional requirements, and “Service Composition Selection” selects an optimal solution out of all possible solutions. Specifically, the “Service Composition Planning” component follows algorithms 3–6 to find paths meeting part of initial states and goals, then remove services and concepts that cannot be invoked, and finally generate information of all possible composition solutions. The “Service Composition Selection” component follows algorithm 7 to calculate an optimal solution in terms of user’s requirements.

V. CASE STUDY

In this section, we give a simple but meaningful example to illustrate our proposed PJSN approach, and compare it with PJSIG approach [4]. In this example, the service repository contains five services and the ontology hierarchy contains eight concepts. The services’ information is shown in Table III, which contains inputs, outputs, response time, throughput and cost.

### TABLE III
**Service QoS Information**

<table>
<thead>
<tr>
<th>Name</th>
<th>Input</th>
<th>Output</th>
<th>Response</th>
<th>Throughput</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>$w_1$</td>
<td>$B, C$</td>
<td>$A, F$</td>
<td>30</td>
<td>4000</td>
<td>360</td>
</tr>
<tr>
<td>$w_2$</td>
<td>$C$</td>
<td>$G$</td>
<td>28</td>
<td>6000</td>
<td>400</td>
</tr>
<tr>
<td>$w_3$</td>
<td>$F, G$</td>
<td>$H$</td>
<td>35</td>
<td>3000</td>
<td>330</td>
</tr>
<tr>
<td>$w_4$</td>
<td>$E, K$</td>
<td>$D$</td>
<td>15</td>
<td>5000</td>
<td>150</td>
</tr>
<tr>
<td>$w_5$</td>
<td>$E, H$</td>
<td>$D$</td>
<td>40</td>
<td>2000</td>
<td>200</td>
</tr>
</tbody>
</table>

Supposing for the output parameter $A$ of $w_1$, and for the input $E$ of $w_4$ and $w_5$, $A \sqsubseteq E$ exists in the ontology tree.

A. PJSN Approach

1) Preprocessing: In the preprocessing stage, we add services into the graph database and generate the service network according to Algorithm 1 and Algorithm 2 in section VI. Specifically, as $A \sqsubseteq E$, a connection from $w_1$ to $E$ is created. The generated service network is shown as Figure 2.

![Fig. 2. The Initial Service Network](image)

2) Service Composition Planning: After the service network is generated in graph database, we can search the database for possible services in solution. Supposing the proposed initial states set is $\{B, C\}$, and the goal is $\{D\}$. To solve this problem, Algorithm 6 is used in “Service Composition Planning.”
Planning” component to find all possible paths that satisfy part of functional requirement.

Firstly, every path that starts from one concept in \{B, C\} and ends with one concept in \{D\} is calculated by Algorithm 3 and Algorithm 4. The Results are listed in Table IV.

<table>
<thead>
<tr>
<th>TABLE IV</th>
<th>RETURNED PATHS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Path 1:</td>
<td>B \rightarrow w_1 \rightarrow E \rightarrow w_4 \rightarrow D</td>
</tr>
<tr>
<td>Path 2:</td>
<td>B \rightarrow w_1 \rightarrow E \rightarrow w_5 \rightarrow D</td>
</tr>
<tr>
<td>Path 3:</td>
<td>B \rightarrow w_1 \rightarrow F \rightarrow w_3 \rightarrow H \rightarrow w_5 \rightarrow D</td>
</tr>
<tr>
<td>Path 4:</td>
<td>C \rightarrow w_1 \rightarrow E \rightarrow w_4 \rightarrow D</td>
</tr>
<tr>
<td>Path 5:</td>
<td>C \rightarrow w_1 \rightarrow E \rightarrow w_5 \rightarrow D</td>
</tr>
<tr>
<td>Path 6:</td>
<td>C \rightarrow w_1 \rightarrow F \rightarrow w_3 \rightarrow H \rightarrow w_5 \rightarrow D</td>
</tr>
<tr>
<td>Path 7:</td>
<td>C \rightarrow w_2 \rightarrow G \rightarrow w_3 \rightarrow H \rightarrow w_5 \rightarrow D</td>
</tr>
</tbody>
</table>

Then, a service set \(S_{rvs} = \{w_1, w_2, w_3, w_4, w_5\}\) and a concept set \(C_{pts} = \{B, C, D, E, F, G, H\}\) can be extracted from all the paths returned. For each service in \(S_{rvs}\), if one of its input concepts is not contained in \(C_{pts}\), this service is not invoked and removed from \(S_{rvs}\), then we check if its outputs can be generated by another service in \(S_{rvs}\). If its output concept can be generated by another service, this concept is kept in \(C_{pts}\). Otherwise, the concept is removed. According to Algorithm 5, services and concepts are checked repeatedly until no more services and concepts can be removed. In this example, \(w_2\) is removed first, as its input concept \(K\) is not in \(C_{pts}\). For the output concept \(D\) of \(w_4\), as \(D\) is an output of \(w_5\), \(D\) is kept in \(C_{pts}\). So far, \(S_{rvs}\) is \{\(w_1, w_2, w_3, w_5\}\}, \(C_{pts}\) is \{\(B, C, D, E, F, G, H\)\}, and no services and concepts can be removed any more. Also, connections linking from and to removed services and concepts are removed accordingly. After this step, the service network is shown in Figure 3.

3) Service Composition Selection: Once all possible paths are found, we search for an optimal solution according to Algorithm 7. When a user specifies a QoS requirement, such as minimal cost or maximal throughput, the system finds a solution by using a greedy strategy to prune services with higher cost or lower throughput respectively. For example, to find a path with minimal cost, services \(\{w_1, w_2, w_3, w_5\}\) are sorted according to cost from high to low. We try to prune \(w_2\) with the highest cost. However, \(w_2\) cannot be removed. Because, if \(w_2\) were removed, \(w_3\) could not be invoked, so does \(w_5\). Then, we try to remove \(w_1\) with second-highest cost. But \(w_1\) cannot be removed either, as it is a necessary service for generating the solution. We keep checking services in the array, until all services are checked. In this example, no service can be removed, and the composition solution is shown in Figure 4.

B. PJSIG Approach

According to PJSIG approach in [4], it preprocesses services and constructs a service composition graph in a graph database, then uses either shortest bidirectional breadth-first algorithm or Dijkstra searching algorithm to find a solution.

1) Preprocessing: In PJSIG, all input concepts of a service together are created as an input concept node, all output concepts of the service together are created as an output concept node, and the service itself is created as a node. A connection is created from the input concept node to the service, and another connection is created from the service to the output node. For each service \(s\) and its input \(s_{in}\), if there is an output \(w_{out}\) of service \(w\), such that \(\forall p \in s_{in}, \exists q \in w_{out}, q \subseteq p\), a connection is created from \(w\) to \(s_{in}\). However, no such relation exists in this example. As \(A \subseteq E\), the output set of \(w_1\) is extended. The service composition graph is shown in Figure 5.

2) Finding Solutions: According to Figure 5, there is no path from initial states \{\(B, C\)\} to goal \{\(D\)\}. Compared with PJSN approach, the PJSIG fails to solve the problem.

VI. ALGORITHMS

In this section, algorithms supporting the proposed framework are explained in details.

Algorithm 1 \(AddSrv\) adds a service along with its properties, input and output concepts into the graph database. Firstly, this algorithm creates a service and adds its QoS properties (lines 1-6). Secondly, the algorithm searches if each input concept of the service already exists in the database. If not, this algorithm creates the input concept and connects it to the service (lines 7-17). Thirdly, for each output concept of the service, the algorithm searches if concepts that have plug-in
semantic matching relation with the output concept exist in the database. If not, the algorithm creates corresponding concepts and connects the service to them (lines 18-28).

Algorithm 1: AddSrv

Input: srv A Service
1 SrvQuery ← CREATE (s : Service {name:srv.name
2 for each property in properties of srv do
3 | SrvQuery ← SrvQuery + property.name : property.value
4 end
5 SrvQuery ← SrvQuery + );
6 runQuery(SrvQuery)
7 for each icpt in input concepts of srv do
8 SrvQuery ← MATCH(c:Concept) WHERE c = icpt
9 return = runQuery(SrvQuery)
10 if return = ∅ then
11 | runQuery( CREATE (c : Concept
12 | {name:icpt.name}) )
13 end
14 SrvQuery ← MATCH (s:Service), (c:Concept)
15 WHERE s.name=srv.name AND c=icpt
16 CREATE (c) : [CONNECT TYPE: INPUT ]->(s)
17 runQuery(SrvQuery)
18 end
19 for each ocpt in output concepts of srv do
20 SrvQuery ← MATCH(picpt:Concept) WHERE
21 ocpt ⊆ picpt RETURN s
22 return = runQuery(SrvQuery)
23 if return ⊆ ∅ then
24 | runQuery( CREATE (ocpt : Concept
25 | {name:ocpt.name}) )
26 end
27 SrvQuery ← MATCH (s:Service), (c:Concept)
28 WHERE s.name=srv.name AND c=picpt
29 CREATE (s) : [CONNECT TYPE: OUTPUT ]->(c)
30 runQuery(SrvQuery)
31 end

Algorithm 2 Preprocessing is used to construct and store the service composition network into the graph database. For each service srv obtained from “Service Information” component in “Preparation” module, the algorithm adds it into the graph database and connects it with other services by using algorithm 1 AddSrv.

Algorithm 2: Preprocessing

Input: SRDB: Services to be Stored in Database
1 for each srv in SRDB do
2 | AddSrv(srv)
3 end

Algorithm 3 SrvOnPath finds all services on paths. Firstly, it searches the graph database (line 3) to find all paths from each input s in the input concepts set IC to an output ocpt in the output concepts set OC (lines 4-6). Then, it extracts services in paths and stores them into service set Srvs (lines 7-10). For each service srv in Srvs, it is added into SR (lines 11-15). Finally, the algorithm returns SR a result.

Algorithm 3: SrvOnPath

Input: IC: Input Concepts; OC: Output Concepts
Output: SR Services on Paths
1 SR ← ∅
2 for each ocpt in OC do
3 | Srvs= runQuery(
4 | MATCH (s:Concept), (e:Concept)
5 | MATCH path=(s)\-[CONNECT*]\->(e)
6 | WHERE s in IC and e = ocpt
7 | WITH nodes(path) as services
8 | MATCH (s:Service)
9 | WHERE s in services
10 | RETURN DISTINCT s)
11 for each srv in Srvs do
12 | if srv /∈ SR then
13 | | SR ← SR ∪ {srv}
14 | end
15 end
16 end

Algorithm 4 CptOnPath finds all concepts on paths. Firstly, this algorithm finds paths from each input concept s in the input set IC to an output ocpt in the output concepts set OC (lines 4-6). Then, it extracts concepts from paths and stores them into Cpts (lines 7-10). For each concept cpt in Cpts, it is added into CP (lines 11-15). Finally, CP is returned as a result.

Algorithm 4: CptOnPath

Input: IC: Input Concepts; OC: Output Concepts
Output: CP Concepts on Paths
1 CP ← ∅
2 for each ocpt in OC do
3 | Cpts= runQuery(
4 | MATCH (s:Concept), (e:Concept)
5 | MATCH path=(s)\-[CONNECT*]\->(e)
6 | WHERE s in IC and e = ocpt
7 | WITH nodes(path) as concepts
8 | MATCH (c:Concept)
9 | WHERE c in concepts
10 | RETURN DISTINCT c)
11 for each cpt ∈ Cpts do
12 | if cpt /∈ CP then
13 | | CP ← CP ∪ {cpt}
14 | end
15 end
16 end
Algorithm 5 RmvInfeaSrv removes services and concepts that cannot be invoked or produced on paths respectively. A service cannot be invoked if one or more of its inputs are not satisfied. For each service \( srv \) in \( SR \), it checks if \( srv \) can be removed or not. If an input concept \( icpt \) of \( srv \) is not in the concepts set of paths \( CP \) (lines 5-6), \( srv \) is removed from \( SR \) and its output concepts are put into \( OC \) (lines 7-8). For each concept \( ocp \) in \( OC \), if it is not an input concept or it is generated by other services on paths, it is added into \( keepCpts \) that is not removed (lines 10-16). When concepts in \( OC \) are checked, \( srv \) is added into \( RSR \), and concepts that are not in \( keepCpts \) but in \( OC \) are added into \( RCP \) (lines 18-19).

**Algorithm 5: RmvInfeaSrv**

**Input:** \( SR \): Services on Paths; \( CP \): Concepts on Paths

**Output:** \( RSR \): Removed Services; \( RCP \): Removed Concepts

1. \( RSR, RCP \leftarrow \emptyset \)
2. for each \( srv \) in \( SR \) do
   3. if \( \exists icpt \in srv_{in} \) and \( icpt \notin CP \) then
      4. \( SR \leftarrow SR \setminus srv \)
      5. \( keepCpts \leftarrow \emptyset \)
      6. for each \( ocp \) in \( srv_{out} \) do
         7. if \( ocp \) is the concept provided as user’s input then
            8. \( keepCpts \leftarrow keepCpts \cup \{ocpt\} \)
         9. end
      10. if \( ocp \) is the output concept of another service in \( SR \) then
           11. \( keepCpts \leftarrow keepCpts \cup \{ocpt\} \)
      12. end
      13. \( RCP \leftarrow RCP \cup \{srv_{out} \setminus keepCpts\} \)
      14. \( RSR \leftarrow RSR \cup \{srv\} \)
6. end

Algorithm 6 CompositionPlanning is used to find possible service compositions that satisfy user’s functional requirements. Firstly, the algorithm finds all services and concepts on paths (lines 1-2). Secondly, the algorithm searches for services and concepts that cannot be invoked or produced (line 3), and removed the services and concepts (lines 4-9). Thirdly, the algorithm queries the graph database to find all services that can be composed to satisfy user’s functional requirements (line 10), and returns the removed services on paths (line 11).

Algorithm 7 CompositionSelection applies greedy strategy to select services according to QoS requirements. If the selection condition is lowest cost (lines 1-3), lowest response time (lines 4-6) or highest throughput (lines 7-9), the algorithm sorts service compositions from high cost to low cost, from high response time to low response time or from low throughput to high throughput respectively. For each service \( srv \) in the sorted services \( SRSort \), the algorithm exploits the greedy strategy to check if it can be removed (lines 11-14). After that, the algorithm examines if there is at least one composition that satisfies user’s functional requirements (lines 15-17). Otherwise, the algorithm adds \( srv \) and other services removed in the process of finding solutions into the graph database, in order to resume to the status before removing \( srv \) (lines 18-23). Finally, the algorithm returns the services left in \( SR \) (line 26).

**Algorithm 6: CompositionPlanning**

**Input:** \( IC \): Input Concepts; \( OC \): Output Concepts

**Output:** \( SR \): Services of All Solutions; \( RSR \): Removed Services on Paths;

1. \( SR \leftarrow SrvOnPath(IC, OC) \)
2. \( CP \leftarrow CptOnPath(IC, OC) \)
3. \( RmvSrvs, RmvCpts \leftarrow RmvInfeaSrv(SR, CP) \)
4. for each \( srv \) in \( RmvSrvs \) do
   5. \( DelSrv(srv) \)
6. end
7. for each \( cpt \) in \( RmvCpts \) do
   8. \( DelCpt(cpt) \)
9. end
10. \( SR \leftarrow SrvOnPath(IC, OC) \)
11. \( RSR \leftarrow RmvSrvs \)

VII. EXPERIMENTAL RESULTS

We conduct experiments to evaluate the suitability and performance of the proposed approach. We run experiments in an environment with software and hardware configurations as follows: 1) Operating System: Windows 10 professional 64-bit, 2) Graph Database: Neo4j 3.4.1, 3) CPU: Intel(R) Core(TM) i5-7200U 2.50GHz 2.71GHz, 4) RAM: 8.00GB DDR4-2400 5) Harddisk: LITEON T11 256GB.

A. Data Set

All experiments are run on the public datasets available for assessing web service composition [13]. Detailed information of datasets including the number of services and concepts is shown in Table V.

B. Performance Analysis

1) Searching Results: We compare the searching results with the PJSIG method in [4]. We randomly generate queries on the datasets as user requests and find optimal solutions. Table VI shows PJSN can find more solutions than the PJSIG method. For example, for queries of datasets 2-5, PGSIG cannot find a solution because it does not support plug-in matching. Regarding to the query of dataset 1, both PJSN and PJSIG can find the same solution.

2) Time for Preprocessing: The preprocessing aims to construct and store the service composition network into the graph database, which is explained in Section IV. According to Figure 6, it is shown that when the number of services increases, the preprocessing time becomes longer.
TABLE V
INFORMATION OF DATASETS.

<table>
<thead>
<tr>
<th>Dataset 1</th>
<th>Dataset 2</th>
<th>Dataset 3</th>
<th>Dataset 4</th>
<th>Dataset 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>#Num of Services</td>
<td>572</td>
<td>4129</td>
<td>8138</td>
<td>8301</td>
</tr>
<tr>
<td>#Num of Concepts</td>
<td>1578</td>
<td>12388</td>
<td>18573</td>
<td>18673</td>
</tr>
</tbody>
</table>

TABLE VI
RESULTS OF EXPERIMENTS.

<table>
<thead>
<tr>
<th>Dataset 1</th>
<th>Dataset 2</th>
<th>Dataset 3</th>
<th>Dataset 4</th>
<th>Dataset 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>PJSN</td>
<td>Minimal Cost</td>
<td>140</td>
<td>74</td>
<td>141</td>
</tr>
<tr>
<td>#Num of Services</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>PJSN</td>
<td>Minimal Response Time</td>
<td>500</td>
<td>670</td>
<td>650</td>
</tr>
<tr>
<td>#Num of Services</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>PJSN</td>
<td>Maximal Throughput</td>
<td>16000</td>
<td>12000</td>
<td>14000</td>
</tr>
<tr>
<td>#Num of Services</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>#Num of Services</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>#Num of Services</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>#Num of Services</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

NOTE 1. #Num: number of services in the solution of service composition.

Algorithm 7: CompositionSelection

Input: IC: Input Concepts; OC: Output Concepts; SR: Services of All Solution; SC: Select Condition

Output: ASR: Services of a Solution

1. if SC is cost then
2.    SRSort ← Sort services in SR from high to low
3. end
4. if SC is response then
5.    SRSort ← Sort services in SR from high to low
6. end
7. if SC is throughput then
8.    SRSort ← Sort services in SR from low to high
9. end
10. RmvSrvs ← ∅
11. for each service srv from the first to the last in SRSort do
12.    if srv ∉ RmvSrvs then
13.        RmvSrvs ← RmvSrvs ∪ {srv}
14.        DelSrv(srv)
15.    end
16.    SolutSrvs, RmvSrvsOnPath ← CompositionPlanning(IC, OC)
17.    if SolutSrvs ≠ ∅ then
18.        RmvSrvs ← RmvSrvs ∪ RmvSrvsOnPath
19.    else
20.        AddSrv(srv)
21.        for each delSrv in RmvSrvsOnPath do
22.            AddSrv(delSrv)
23.        end
24.    end
25. end
26. ASR ← SR \ RmvSrvs

3) Time for Service Composition Search: In the experiments, we randomly generate some requests to search for optimal solutions. “Service Composition Planning” component finds possible services in all paths, and “Service Composition Selection” component finds an optimal solution. For each dataset, the execution time for “Service Composition Planning” and “Service Composition Selection” components is illustrated in Figure 7. When the number of services and concepts increase, the execution time for “Service Composition Planning” component increases accordingly.

According to Figure 7, the execution time of “Service Composition Selection” component is decided by the the number of services in the paths as well as the the number of services and concepts in the dataset.

VIII. CONCLUSION

The objective of this paper is to solve service composition problem with plug-in semantic matching by constructing the
pre-joined service network in Neo4j graph database. Firstly, services, concepts and possible service compositions are pre-processed and stored as vertices and connections into a service network graph in graph database. Secondly, when a user request arrives, it calculates an optimal solution that satisfies user’s functional requirements and nonfunctional QoS requirements through service composition planning and selection stages performed on the service network. Experimental results show that the proposed approach can find better solutions than the PJSIG approach in [4]. Besides, the Neo4j database is able to be deployed on cloud as a cloud database, the proposed approach has a potential to be immigrated to the cloud accordingly. As future work, we plan to extend and explore our work on cloud database.