Scalable and efficient web services composition based on a relational database

Daewook Lee\textsuperscript{a}, Joonho Kwon\textsuperscript{b}, Sangjun Lee\textsuperscript{d,}\textsuperscript{*}, Seog Park\textsuperscript{a}, Bonghee Hong\textsuperscript{c}

\textsuperscript{a} Department of Computer Science and Engineering, Sogang University, Seoul 121-742, Republic of Korea
\textsuperscript{b} Institute of Logistics Information Technology, Pusan National University, Busan 609-735, Republic of Korea
\textsuperscript{c} Department of Computer Engineering, Pusan National University, Busan 609-735, Republic of Korea
\textsuperscript{d} School of Computing, Soongsil University, Seoul 156-743, Republic of Korea

\begin{abstract}
Recently, there has been growing interest in web services composition. Web services composition gives us a possibility to fulfill the user request when no single web service can satisfy the functionality required by the user. In this paper, we propose a new system called PSR for the scalable and efficient web services composition search using a relational database. In contrast to previous work, the PSR system pre-computes web services composition using joins and indices and also supports semantic matching of web services composition. We demonstrate that our pre-computing web services composition approach in RDBMS yields lower execution time for processing user queries despite of and shows good scalability when handling a large number of web services and user queries.
\end{abstract}

1. Introduction

In recent years, web services have become a universal technology for integration of distributed and heterogeneous applications over the Internet. Web Services are self-contained, self-describing, modular applications that can be published, located, and invoked across the Web. The emergence of technologies and standards for web services has promoted the wide application of web service in many areas such as business, finance and tourism.

Fig. 1 shows an architecture of the web services (Champion et al., 2002). Service providers publish their services in UDDI business registries (Clement et al., 2004) after describing the interface to their services using WSDL (Christensen et al., 2001). Service requesters find web services suitable for their needs by browsing the registry or searching UDDI registry with keywords.

Fig. 2 depicts an example WSDL file which describes a “TourInfo Lookup” web service. We can extract the service name “TourInfo Lookup” from a “service” element and an operation name “get-TourInfo” from an operation element and a binding element. From “portType” and “message” elements, we can obtain parameters of the web service: “packageID” is an input parameter and “tourInfo” is an output parameter.

As shown in Fig. 2, a WSDL file contains the interfaces of web services and describes the functionalities of web services. Each web service has a web service name, an operation, one or more inputs and outputs. Generally, a web services have more than one operation. However, we consider several operations as several web services because the unit participating in a web services composition is an operation of a web service.

As a growing number of web services are available on the Web and in organizations, storing and searching the right set of web services become ever more important. To address this problem, several approaches have been proposed to search single web services efficiently (Dong et al., 2004; Makris et al., 2006; Kuang et al., 2007; Song et al., 2007; Cheng et al., 2009).

Since the functionalities of the individual web services are limited, the systems that search not only single web service but also web services composition have been researched and developed. Through the web services composition, we can provide more sophisticated functionalities with the limited individual web services.

Fig. 3 depicts example web services for web services composition. The web services composition is useful when we are looking for a web service with specific input and output and there is no single web service which satisfies the request. For example, assume that a user wants to find the web service having “packageID” and “nationName” as input parameters and “hotel”, “airlineInfo” and “carSize” as output parameters among six web services in Fig. 3. If a search system does not support the web services composition, there is no answer to the user query. However, the system pro-
Most solutions for web services composition use in-memory based algorithms (Gekas and Fasli, 2005a,b; Berardi et al., 2005; Liu et al., 2005; Kona et al., 2007; Kwon et al., 2008). However, when there are a large number of web services available, it is nontrivial to quickly find web services composition satisfying the given request. Moreover, these approaches cannot support the semantic matching of web services to the user query.

Keywords for describing input and output parameters of one service may be different from another. Thus, if a search system fulfills ontology based semantic matching in addition to the web services composition search, we can have more answers to user queries. For example, if we have the ontology in which “city” is equal to “state” and “hotel” is similar to “motel”, a composition of ws3 and ws5 could be an answer to the user who wants costs of motels in the specific state. There has been a lot of studies on semantic matching of web services (Paolucci et al., 2002; Bansal and Vidal, 2003; Sirin et al., 2003; Sivashanmugam et al., 2003; Dong et al., 2004; Pathak et al., 2005). However, although they add semantic matching to searching step, they do not support web services compositions.

The key characteristic of our approach is to use a relational database system to build a scalable and efficient web services composition search system. The web services composition are computed in advance and stored in tables and ontology information is extracted from web services and also stored in tables. Pre-computing and searching is done by SQL statements. For this purpose, we propose a web services composition search system called PSR (pre-computing solutions for web service composition in a RDBMS) in this paper. Note that a preliminary version of this paper, omitting many details, examples, and results, appeared as Kwon et al. (2007).

The key contributions of our work are summarized as follows:

- We have developed web services composition system in a relational database management system. To the best of our knowledge, this paper is the first work for computing web services composition search using a relational database.
- We proposed pre-computing algorithms for web services composition and a search algorithm. For this purpose, web services into the weighted, directed graph called a composition graph. The composed web services are considered as visited (intermediate) vertices of paths between web services in the composition graph. Pre-computing web services composition is done by joins. Web services composition search can be done by simple SQL statements.
- The PSR system supports web services having single input and single output and can handle web services having multiple input and output parameters.
- Semantic matching of web services is supported by storing ontology information into tables. Pre-computing algorithm uses ontology information during a join operation.
- For performance evaluation, we conducted extensive experiments over different sets such as synthetic data and WSC (Web Services Challenge) data which is a benchmark for the web services composition. The experimental results shows the benefits of our pre-computing approach in RDBMS with a large number of web services and user queries.

The remainder of this paper is organized as follows. Section 2 provides an overview of the PSR system in Section 3. In Section 4 we describe the basic table structures and algorithms for web services composition. In Section 5, we present an extension of the PSR system to support semantic matching of web services composition. Sec-
2. Related work

In this section, we survey current efforts related to web services composition. We first review research work related to web services discovery. We then discuss related work on semantic services search and look at several recent efforts about web services composition search. Finally, we have provided motivation for our work.

2.1. Web services discovery

The growing number of web services on the Web raises a problem for searching desire web service. In this regard, there is a growing body of work for finding web services based on keywords and descriptions (Dong et al., 2004; Makris et al., 2006; Kuang et al., 2007; Song et al., 2007; Cheng et al., 2009). Woogle (Dong et al., 2004) supports not only a simple keyword search but also a similarity search for web services using clustering. The limitation of Woogle is that although the semantic matching is added to searching step, it does not support web services compositions. In other words, they just check whether a single service is able to fulfill the query. Makris et al. (2006) suggested an effective and efficient dynamic selection of web services in terms of performance factors such as execution time and response time. They showed that the time complexity of an optimal selection algorithm for a series of web services is $O(kn^3)$, where $k$ is the number of maximum edges of a web service and $n$ is the number of web services. Kuang et al. (2007) proposed a fast web services discovery algorithm based on an inverted indexing technique. Song et al. (2007) studied how to use general-purpose search engines to discover web services with different methods for publishing. Recently, a counting Bloom filter (CBF) (Cheng et al., 2009) is used to solve the service discovery problem. In this work, they encoded both the registered web services and incoming service requests as bit strings and searched the appropriate web service by a string matching. The computational complexity of CBF approach is $O(|A||A|)$, where $A$ is a set of elements which a hash function maps to and $|A|$ represents the size of $A$. Traditionally, these service discovery algorithms (Makris et al., 2006; Kuang et al., 2007; Song et al., 2007; Cheng et al., 2009) consider only one service as a suitable candidate fulfilling a user request while they do not consider web services compositions.

2.2. Semantic web services search

A limitation of web services discovery approaches based on the keywords and descriptions is the lack of an explicit semantics. For example, two identical web service descriptions including a keyword “apple” may mean very different things depending on the context. For this reason, many efforts have been made in the semantic web services search area.

Paolucci et al. (2002) proposed a semantic-based algorithm for web service discovery using DAML-S (superseded by OWL-S Martin, 2004) profiles. Their algorithms compares requested inputs and outputs against the advertised inputs and outputs. Bansal and Vidal (2003) used a DAML-S process model to provide more accurate matchmaking. DAML-S semantic service descriptions are directly combined with the WSDL descriptions to execute the composed services in Sirin et al. (2003). Adding semantics to web services by storing the semantic annotation of web services in the existing structures of UDDI was proposed in Sivashanmugam et al. (2003). Medjahed et al. (2003) proposed an ontology-based framework for the automatic composition of web services. They specified an ontology using DAML+OIL language (Horrocks, 2002) and defined composability rules which compare the syntactic and semantic features of web services. Pathak et al. (2005) provided an approach for flexible discovery of web services over the semantic web. Their discovery algorithm is based on the match between inputs and outputs. These semantic web services search approaches (Paolucci et al., 2002; Bansal and Vidal, 2003; Sirin et al., 2003; Sivashanmugam et al., 2003; Pathak et al., 2005) has a weakness of not supporting web services composition during the semantic search process. An EASY (Mokhtar et al., 2008) system supports efficient, semantic, context- and QoS-aware service discovery in a pervasive computing environment. But, they only concerned with the web service discovery problem. Recently, Forte et al. (2008) suggested the use of ontology and web services in a ubiquitous environment. They introduced the service discovery and composition in the mobile environments. However, their main focus is to solve problems relating to content adaptation for mobile devices.

2.3. Web services composition

Composing web services in the registry gives us a possibility to fulfill the user request, when no single web services can satisfy the functionality. This has triggered a considerable number of research efforts and made web services composition problem as an active issue of a research area.

Most of the previous approaches (Gekas and Fasli, 2005a,b; Kwon et al., 2008) have been based on transforming web services composition search into the graph search problem. Web services are treated as vertices and the web services composition search is transformed into finding paths between vertices. To find paths, the depth first search is used iteratively until we reach a web service with same output as user query’s. In Gekas and Fasli (2005a), authors also proposed the PageRank technique for ordering compositions by reference frequency of web services. In Gekas and Fasli (2005b), all possible compositions among web services is searched using by multiply adjacency matrices $N$ times based on DFS (depth first search). Kwon et al. (2008) proposed a two-phase algorithm which returns web services composition without redundant web services.

An important issue related to web services composition is concerned with computing automatic composition of web services. Colombo (Berardi et al., 2005) is a framework for automatic web services composition. The aim of Colombo is not for web services composition search but for real time transmission of output values when web services are actually performed through workflow using messaging technique. An automatic composition approach that can generate a composed services with minimal execution price is proposed in Liu et al. (2005). Kone et al. (2007) defined the automatic discovery and composition problems formally and presented an algorithm which automatically selects the individual services involved in composition for a given query.

Another issue of web services composition is to adopt QoS-based models or mechanisms for selecting best web services composition available. In this QoS-Aware composition approach, there also exists a number of research proposals (Zeng et al., 2003; Liu et al., 2005; Batra and Batra, 2005; Berbner et al., 2006). Zeng et al. (2003) proposed an optimizing service selection at a composite service based on a generic QoS model using established linear programming techniques. In the extension of this work (Zeng et al., 2004), they presented a QoS-aware middleware platform, called AgFlow, supporting quality driven web services composition. The AgFlow system consists of a service quality model and two alternative service selection approaches. Batra and Batra (2005) proposed an approach to trigger and perform composite service replanning during execution because the actual QoS values may deviate from the estimation. WSQoSX (Berbner et al., 2006) is a workflow engine which calculates an exe-
cution plan that maximizes the overall QoS by use of heuristics. The limitation of QoS-aware approaches is that they do not consider supporting the semantic matching for web services composition. Recently, Ko et al. (2008) suggested an efficient QoS-oriented web services composition algorithm that combines tabu search and simulated annealing meta search and Canfora et al. (2008) proposed a QoS-aware composite service binding approach based on genetic algorithms (GAs). We leave the QoS-aware web services composition implemented on a relational database as a future research topic.

Other issues related to web services composition include collision-free and RESTful web services composition. Vaculin and Sycara (2009) present a matchmaking algorithm for retrieval of the top-k collision-free services combinations. The time complexity the algorithm is $O((m \log m) \cdot n)$, where $n$ is the number of output of a user request and $m$ is the maximum number of web services able to produce some output of effect in the request. Their composition results avoid collision such as unwanted side-effects, effect duplications and contradictory effects. However, they return only top-$k$ results. Zhao and Doshi (2009) introduced the challenges of the RESTful web services composition and proposed a formal model for RESTful web services composition.

2.3.1. Our motivations

Much of the previous work in semantic web services search and web services composition is based on in-memory algorithms. However, the scalability of in-memory approach is limited by the amount of available physical memory. In addition, many approaches in the semantic web service search do not consider web service compositions and many algorithms in the web services composition search do not support the semantic matching of web services.

These shortcomings have motivated us to build the scalable and efficient web services composition search system using relational database. In our approach, web services composition are computed in advance and stored in tables. For web services composition searches, we look up the pre-computed tables rather than actual web services.

3. The PSR system

In this section, we present the key ideas of the PSR system. We first formulate the web services composition problem that we address in this paper. We also provide an architectural overview of the PSR system and briefly describe its core components. We then explain the basics of ontology.

3.1. Problem definition

Formally the problem of web services composition search can be stated as follows.

Given a set $N$ of web services in the UDDI registry and an user query $Q$ which specifies web services’s input and output, find the subset $N' \subseteq N$ such that every $ws \in N'$ has a match in $Q$ including composed web services.

We can pre-compute the web services composition based on the weighted, directed graph. The graph is defined as follows:

Composition graph $G=(V, E)$ is a weighted, directed graph with a weight function $W$. $V$ is a set of web services and $E$ is a set of directed links (edges). There is a direct link from web service $ws_x$ to $ws_y$ when the output of $ws_x$ is equal to the input of $ws_y$. $W$ is a weight function which maps edges to real-valued weight based on ontology.

In the composition graph, we consider web services as vertices and make a directed connection between two web services when the output of a web service is equal to the input of the other web service. Paths between vertices are the web services composition in the composition graph.

Our approach for pre-computing web services composition consists of two steps: building a composition graph and searching answers from the composition graph. At building time, we use all edges between web services to construct all possible paths between web services. At searching time, we check the each path whether the input of the start vertex is equal to user query’s input and the output of end vertex is equal to user query’s output.

3.2. Architectural overview

In this subsection, we shall describe the core components of the PSR system. Fig. 4 shows an architectural overview of our system.

The core engine of the PSR system is shown in a dotted box. Service Providers register web services into the UDDI registry server. When a web service is added to the UDDI, the WSDL description is parsed and information of the web service is extracted. The operation, input and output of the web service are stored in tables in a relational database. All links of the composition graph obtained from web services are stored into an Edge table. The possible web services composition is stored into a Path and a VisitedWS table. A hierarchical tree of ontology concepts represents an external ontology information and stored in a “OntClass” table. The input and output parameters of web services are assigned into the classes of the external ontology information. The basic column is extended by adding columns or joining ontology information. The names for extended tables begin with the keyword “Ont”. A user query is given by the Service Requester. After searching tables, the answers which include web services composition will be return to the Service Requester.

The details of the basic structures and algorithms are explained in Section 4, and an extension for supporting semantic matching is described in Section 5.
4. Web services composition search algorithm

In our system, the WSDL description of web services, the composition graph and the possible web services composition are stored in tables of a relational database. In this section, we shall describe the relational representation for web services and the algorithms for web services composition search.

4.1. Relational representation of web services

In this subsection, we shall describe the schema of the tables for representing web services and web services composition.

4.1.1. Web services and parameters

The operation, input and output information of web service are extracted from the WSDL file and stored as tuples in the corresponding table. The unique identification for web service and the web service name is stored as a tuple in the WS table. A Pars table stores all the names of parameters for web services. A parameter can be used as an input parameter of a web service and can be used as output parameter of a different web service. To model this property, we introduce an Input table and Output table. The Input table and the Output table stores the input parameters of web services and output parameters of web services, respectively.

Fig. 5 shows the schema of tables for web services and input/output parameters and how the example web services in Fig. 3 are stored. For example, a parameter “packageID” is used as an input parameter of a web service ws1. This is stored in the first row of the Input table. A parameter “city” of the forth row in the Pars table is used as an output of the web service ws2 and as input of the web service ws3. This is stored in the forth row of the Input table and the second row of the Output table.

4.1.2. Composition graph

All links of the composition graph transformed from web services is stored in the Edge table. Fig. 6(a) shows the Edge table. We can obtain the weighted, directed graph representation of web services in Fig. 6(b) from the join operations of Pars, Input, Output and Edge tables, and vice versa. Each tuple in the Edge table means that a weight, directed link between two vertices (web services). The columns ws_s and ws_e are the start vertex (web service) and the end vertex (web service) of the link.

Example 1. Consider the web services in Fig. 3 and the Edge table in Fig. 6(a). The first row in the Edge table means that there exists a directed link from ws1 to ws3. The parameter of the first row is “TourInfo” which is the output of ws1 and the input of ws3.

4.1.3. Web services composition

The Path table stores all paths between web services, which means all possible web services composition is pre-computed and stored. The tuple in the Path table is a transitive closure of the start and the end web service. To return all participating web services for web services composition, we need to know the intermediate vertices of a path. The VisitedWS Table stores intermediate vertices(web services) of a path. These two tables can be computed from the Edge table. The detailed algorithms are explained in Section 4.2.

Example 2. Consider the first and forth tuples at the Path table in Fig. 7. The path P1 consists of one edge which connects two web
services ws1 and ws3. The path P4 consists of two edges and three web services. Its start web service and end web service is ws1 and ws5. The intermediate web services of the path P4 is stored at the VisitedWS table in Fig. 7(b). The 7th, 8th, and 9th rows in the VisitedWS table denotes that the path P4 has three web services ws1, ws3 and ws4.

Algorithm BEP-Join, which computes all the web services composition and stores them into the Path table, is presented in Algorithm 1. The key idea of BEP-Join is to repeatedly construct Pi table which stores the paths with i edges, by joining P1 and Pk−1 until there are no tuple in table P. This means several intermediate tables are created during the pre-computing.

Algorithm 1. A bulk loading EP-join (BEP-Join)

1: Relation P1 ← Relation Edge;
2: i ← 2;
3: repeat
   4: Pi ← Pi−1 union ws_i ws_i Pi−1;
   5: i ← i + 1;
   6: k ← i;
   until Pi has no tuples;
7: Relation Path ← P1 ∪ P2 ∪ ... ∪ Pk−1 ∪ Pk;

4.2. Pre-computing web services composition

In this section, we describe how to pre-compute web services composition from the Edge table and to store them into Path and VisitedWS table. We propose two algorithms for computing web services composition: a bulk loading EP-Join algorithm (BEP-Join) for populating a database from scratch and an incremental EP-Join (IEP-Join) algorithm for insertion of new web services into already existing database.

4.2.1. Bulk loading EP-Join algorithm

Since the popularity of web services brings about amounts of WSDL documents to be stored, efficient bulk loading techniques are necessary.

4.2.2. Time complexity of BEP-Join

If we are given web services having E edges, the time complexity of each join operation in Line 4 is O(E^2). Thus the time complexity for computing all paths using BEP-Join algorithm is O(E^3). If there exists appropriate indices, the complexity can be reduced to O(E^2 log E).

4.2.3. Web services composition

The overall procedure for pre-computing web services composition is described in Algorithm 2. BEP-Join in Line 6 algorithm can be replaced with IEP-Join algorithm. Fig. 8 shows a graphical representation of the query plan for Algorithm 2. The algorithms are described using SQL and PL/SQL statements. We do not show the SQL and PL/SQL statements here because the SQL and PL/SQL statements are long and complex.
Algorithm 2. Web services composition

Input: \{WS, Pars, Input, Output, Edge, Path, VisitedWS\}; tables;
\[/\]
\[
//Fetch Information of web services from UDDI and store them in
WS, Pars, Input, Output table
1: WS ← \{WS\|ws = (ws.id, name);\}
2: Pars ← \{pars|pars = (p.jd, p_name);\}
3: Input ← \{(input|input = (i.jd, p.jd, ws.id));\}
4: Output ← \{(output|output = (o.jd, p.jd, ws.id));\}
5: Edge ← \{(Input\&Output);\}
6: Pre-compute web services composition using the IEP-Join
Algorithm;
\[/\]

4.2.4. Incremental EP-Join algorithm

Since changes to the composition graph, caused by new web
services being added to the system, occur often, the incremental
algorithm is also important. In this subsection, we shall explain the
incremental algorithm, called IEP-Join, for web services composition.

The services compositions are computed in advance and stored
into the Path table. Thus we do not need the intermediate tables \(P_i\)
created the BEP-Join in the IEP-Join.

The basic idea of IEP-Join is to pre-compute web services com-
oposition as follows: Create a newEdge table from newly inserted
web services. Get the first tuple from the newEdge table and generate
all possible paths including this edge. Then, insert the generated
paths into the Path table. If the paths consist of more than two
edges, we store the intermediate vertices of the path into the Visi-
tedWS table. This procedure is repeated until we get all tuples from
the newEdge table. The IEP-Join algorithm is shown in Algorithm 3.


Input: \{newEdge, Path\}; tables;

1: foreach e in the newEdge table do
2: insert e into the Path table;
3: foreach p in the Path table do
4: if case 1 then
5: newPath ← connect e at the start of p;
6: else if case 2 then
7: newPath ← connect e the end of p;
8: else
9: newPath ← connect e at the middle of p;
10: endif
endfor
11: insert the newPath into the Path table;
12: if numEdge(newPath) \geq 2 then
13: insert the intermediate vertices of newPath into the VisitedWS table;
14: endif
endfor

The three cases in Fig. 9 is a key to performing Algorithm 3. If we
choose an edge \(e\) and a path \(p\), we can generate the new path based
on the following cases. First, if the end vertex of path \(p\) is equal to
the start vertex to an edge \(e\), then the new path is generated by
connecting edge \(e\) at the end of path \(p\). Second, if the start vertex
of path \(p\) is equal to the end vertex of \(e\), the new path is generated
by connecting edge \(e\) at the start of path \(p\). Third, if the start vertex
and end vertex of edge \(e\) are equal to intermediate vertices of path
\(p\), the new path is generated by connecting the edge \(e\) at the middle
of path \(p\).

Example 3. Consider the tuples of the Edge table in Fig. 6(a), and
assume the Path table is empty. The first tuple is \(1, “ws1”, “ws3”\)
and the second tuple is \(2, “ws2”, “ws3”\). The paths \(P1\) and \(P2\) which
have only one edge are inserted into the Path table. No other paths
can be generated according to cases in Fig. 9.

The third tuple is a row having a value of \(3, “ws3”, “ws4”\). The
one-edge path \(P3\) is inserted into the Path table. We can generate
the path \(P4\) by connecting the path \(P1\) and the edge \(E3\) (the third

tuple) according to cases in Fig. 9. When the path P4 is inserted into the Path table, its intermediate vertices are also stored in the VisitedWS table because it has more than two edges. This process is repeated until we get all tuples from the edge table.

4.2.5. Time complexity of IEP-Join

Assume again that we are given web services having E edges. The time complexity of the foreach loop in Line 1 is O(E). The number of rows in the Path table is increased as IEP-Join proceeds. Initially the number of paths in the Path is same as the number of edges E. At the every execution of Line 3, if we assume that the probability of connected edges is p_{e}, then the number of newly generated paths is bounded to p_{e} · || Path ||, where || Path || is the number of rows in the Path table. || Path || is computed as follows: p_{e} · E + p_{e}^2 · E + · · · + p_{e}^F · E = k · E, where k is represented as p · 1 − p^0/1 − p. Thus, the total time complexity is O(E^2).

Algorithm 4. Web service composition search

| Input: | Q: user query; |
| Output: | FinalWSs: pairs of web services; |
| 1: PathIDs ← Index scan on Path, Input, Output tables using the query Q; |
| 2: InPar ← Index Scan on Input, VisitedWS tables; |
| 3: OutPar ← Index Scan on Output, VisitedWS tables; |
| 4: UserInPar ← all input parameters in the query Q; |
| 5: if InPar – OutPar – UserInPar <> NULL then |
| 6: ImpWSs ← Index Scan on Input table using InPar – OutPar – UserInPar; |
| 7: ImpPathIDs ← Index scan on VisitedWS table using ImpWSs; |
| 8: PathIDs ← PathIDs – ImpPathIDs; |
| 9: Go to Line 2; |
| endif |
| 10: FinalWSs ← Index scan on VisitedWS using PathIDs; |
| 11: return FinalWSs; |

4.3. Web services composition search

After pre-computing web services composition, we can perform the web service compositions search against a given user query. The searching procedure is done as follows: First, finds all paths PathIDs which can be obtained from user inputs and outputs in Line 1. Second, determines the input parameters not having values in Lines 2–3. The paths from PathIDs have lots of web services containing several input parameters. To produce the output parameters as results of their services, the values of input parameters must be provided. Thus we need to check all input parameters of web services in paths form PathIDs. From the paths (PathIDs), we compute input parameters (InPar) and output parameters (OutPar) in Lines 2–3. The UserInPar is a set of input parameters from the user query Q. The set of parameters computed from InPar–OutPar–UserInPar in Line 5 means the no value input parameters. Third, if the no value input parameters are existed, finds and eliminates web services containing input parameters with no values from PathIDs in Lines 5–8 and again checks all input parameters of web services in paths from newly computed PathIDs in Line 9. Finally, if the no value input parameters are no longer existed, all paths constituting web services composition are found in Line 10. This procedure is described in Algorithm 4.

4.3.1. A running example

Having discussed the technical approaches for web services composition, we now describe a running example with SQL statements.
1. **PathIDs** = (P4, P5, P7, P8, P10, P11, P14, P15)

   ```java
   SELECT PathID FROM Path WHERE ws_s IN (SELECT p_id FROM Input WHERE p_id IN
   (SELECT p_id FROM Pars WHERE p_name IN ('packageID', 'nationName')))
   AND ws_e IN (SELECT p_id FROM Pars WHERE p_name IN ('cost', 'reserveID'))
   ```

2. **InPars** = (1, 2, 3, 4, 5, 6, 7, 8, 9)

   ```java
   SELECT p_id FROM Input WHERE ws_id IN (SELECT ws_id FROM VisitedWS WHERE p_id IN (PathIDs))
   ```

3. **OutPars** = (3, 4, 5, 6, 7, 9, 10, 11)

   ```java
   SELECT p_id FROM Output WHERE ws_id IN (SELECT ws_id FROM VisitedWS WHERE p_id IN (PathIDs))
   ```

4. **UserInPars** = (1, 2)

   ```java
   SELECT p_id FROM Pars WHERE p_name IN ('packageID', 'nationName')
   ```

5. **InPars - OutPars - UserInPars = (8) <> NULL

6. **ImpWSs** = (ws6)

   ```java
   SELECT ws_id FROM Input WHERE p_id IN (InPars - OutPars - UserInPars)
   ```

7. **ImpPathIDs** = (P9, P10, P11)

   ```java
   SELECT DISTINCT p_id FROM VisitedWS WHERE ws_id IN (ImpWSs)
   ```

8. **PathIDs - ImpPathIDs → PathIDs** = (P4, P5, P7, P8, P14, P15)

9. Goto 2

   Current **PathIDs** = (P4, P5, P7, P8, P14, P15)

2. **InPars** = (1, 2, 3, 4, 5, 6, 9)

   ```java
   SELECT p_id FROM Input WHERE ws_id IN (SELECT ws_id FROM VisitedWS WHERE p_id IN (PathIDs))
   ```

3. **OutPars** = (3, 4, 5, 6, 7, 9, 10, 11)

   ```java
   SELECT p_id FROM Output WHERE ws_id IN (SELECT ws_id FROM VisitedWS WHERE p_id IN (PathIDs))
   ```

4. **UserInPars** = (1, 2)

   ```java
   SELECT p_id FROM Pars WHERE p_name IN ('packageID', 'nationName')
   ```

5. **InPars - OutPars - UserInPars = (1) = NULL

10. **FinalWSs** = (ws1, ws2, ws3, ws4, ws5)

   ```java
   SELECT DISTINCT ws_id FROM VisitedWS WHERE p_id IN (PathIDs)
   ```

---

4.4. Supporting pre-conditions/post-conditions of web services

Interface of a web service describes syntactical information which inputs are required and which outputs are returned by a web service. The IEP-Join and BEP-Join algorithms as described in Section 4.2 are based on the interface of a web service.

However, two services having same inputs and outputs offer different functionalities. For example, consider two web services having one input and one output. They take as input the name of a city and returns a set of flight numbers. The first service returns departures from the name of an input city, whereas, the other returns arrivals from the name of an input city.

The pre-conditions and post-conditions can be augmented with the PSR system to support the functionality of a web service. If pre-conditions of a web service exists, it does not execute correctly unless its preconditions are qualified. A post-condition is the condition that holds once the service has been executed successfully.

Fig. 11 shows the extended tables for handling pre-conditions and post-conditions of a web service. A **Cond** table records the pre-conditions and post-conditions of web services. Each tuple of **Cond** table has three columns: (1) **c_id** is an identifier for the condition,
(2) oper denotes an operator, and (3) val is the value of pre- or post-conditions. ExtInput and ExtOutput tables have a c_id column for the pre-condition and post-condition. This column roles as a foreign key when joining ExtInput (ExtOutput) and Cond tables.

**Example 4.** Examples of pre-condition and post-condition are depicted in Fig. 11. We can limit the web service which considers only “Compact” cars, by joining Pars, ExtInput and Cond Tables. The blue arrow shows an usage of a post-condition which refines the web services whose output value is lower than 5000.

The one of major benefits of handling pre-conditions and post-conditions is the reduced number of intermediate results during joins. If the selectivity of pre-condition (post-condition) is high, we can evaluate pre-conditions (post-conditions) before join operations.

5. Semantic matching of web services composition

Given the basic architectures for web services composition search in Section 4, we now move on to explain an extension of PSR for supporting semantic matching of web services composition.

5.1. Supporting ontology-based web services composition

5.1.1. Ontology information

An ontology is simply a description of concepts relevant to a given domain along with attributes/properties characterizing these concepts. It comprises of concepts with their relationships and properties. We can construct an ontology tree as shown in Fig. 12 by using the relationship from concepts.

The degree of match between two ontology keywords in the ontology tree can be divided into four cases: (1) exact, (2) plug-in, (3) subsumes and (4) fail (Paolucci et al., 2002).

**Definition 1.** Let Ci be a provided (based) ontology concept including an ontology keyword ki and Cj be a required ontology concept containing an ontology keyword kj. Then, the degree of match is defined as follows:

1. If two keywords ki and kj are equivalent or two concepts Ci and Cj is equivalent, then it is an exact match. (2) If a provided ontology concept Ci subsumes Cj, then it is a plug-in match. (3) If a required ontology concept Cj subsumes Ci, then it is a subsumes match. (4) If there is no relationship between two ontology concepts Ci and Cj, it is a fail match.

The exact matching means that two compared keywords are existed in the same class. The plug-in matching means that a based ontology concept includes the other concept, whereas, the subsumes matching means vice versa. The fail means that there is no relationship between two ontology concepts.

To improve the efficiency of computing the degree of match, we used the numbering scheme technique from the XML community (Li and Moon, 2001). Each ontology concept is assigned three numbers, (LV, order, size) to describe the tree structure. The “LV” is the level of the node in the tree. The “Order” is assigned by the preorder traversal of the tree. and the “size” means the number of children/descendant nodes. These values are assigned as follows:

- For node y and its parent x, the range [order(x), order(x) + size(x)] contains the range [order(y), order(y) + size(y)].
- For two sibling nodes x and y, if x is the predecessor of y in preorder traversal, order(x) + size(x) < order(y).

By examining the values of “Order” and “size”, we can easily compute the degree of match. That is, for two concepts x and y in the ontology, x subsumes y (⇒ x is an ancestor of y) if and only if

**Fig. 11.** Supporting the functionality of web service.

**Fig. 12.** Example of ontolog.
order(x) < order(y) < order(x) + size(x). With an additional condition, \( LV(y) = LV(x) + 1 \), \( x \) can be determined as the parent of \( y \).

**Example 5.** Consider the ontology concepts in Fig. 12. Since the concepts “Postcode” and “Zipcode” belong to the same ontology class “Ont1”, they can be considered as an “exact” matching. “Temperature” belongs to the class “Ont5” and “Weather” belongs to the class “Ont2”. Since the range of “Ont2” \([110, 20 + 10]\) includes the range of “Ont5” \([11, 12 + 8]\) and the level difference is the value of 1, there exists a parent–child relationship between two concepts. Thus the “subsumes” matching or “plug-in” matching exists in these concepts according to the base concept.

We assume that the input, output, and operation name of a web service are mapped into ontological concepts. How to map keywords into ontologies automatically and efficiently is beyond the scope of this paper.

5.1.2. Ontology computation

In this subsection, we shall explain how to store ontology information into a table and how to compute degree of matching from stored ontology information.

**5.1.2.1. Tables for ontologies.** The ontology information is stored in the OntClass and Pars tables. The tree structure of the ontology (Paolucci et al., 2002) are represented as the “LV”, “Start” and “End” columns of OntClass table. The Pars table has an extra column “ClassID” for mapping parameters to ontology concepts. By joining the OntClass table and Pars table, we can obtain the complete form of the ontology tree. Fig. 13 shows the schema for ontology information.

For the semantic matching of web services composition, let us consider four web services in Fig. 14. For the simplicity of the explanation, we assume that each web service has one input parameter and one output parameter.

By joining the OntClass, Pars, and Input (Output) tables we can obtain the OntInput (OntOutput) table. The OntInput and OntOutput tables store web services information which have the relevant ontologies. Fig. 15 represents SQL statements for OntInput and OntOutput tables.

A semantic matching of web services composition is stored into an OntEdge table in Fig. 16(a) and can be expressed as a semantic composition graph in Fig. 16(b). The semantic composition graph can be generated from the OntEdge table, and vice versa. The OntEdge table has columns \( \text{ws}_e \) and \( \text{ws}_s \) denoting the start web service and the end web services of a web services composition. The column DM (degree of matching) means the accuracy of matching based on the ontology. The value of DM column is a weight of the link. For example, the value ‘0’ means the ‘exact’ matching and the value ‘2’ means the ‘plug-in’ matching. How to compute values of the DM column will be explained in Section 5.1.2.

The forth column in the OntEdge table appears due to the semantic matching of parameters “Temperature” and “Weather”.

**Fig. 13.** Tables for storing ontology information.

```
<table>
<thead>
<tr>
<th>OntClass table</th>
<th>Pars table</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClassID</td>
<td>LV</td>
</tr>
<tr>
<td>Ont0</td>
<td>1</td>
</tr>
<tr>
<td>Ont1</td>
<td>2</td>
</tr>
<tr>
<td>Ont2</td>
<td>2</td>
</tr>
<tr>
<td>Ont3</td>
<td>2</td>
</tr>
<tr>
<td>Ont4</td>
<td>2</td>
</tr>
<tr>
<td>Ont5</td>
<td>3</td>
</tr>
</tbody>
</table>
```

**Fig. 15.** SQL statements for the OntInput (OntOutput) table.

```
<table>
<thead>
<tr>
<th>OntInput SQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT l_id, p_id, ClassID, LV, Start, End, ws_id</td>
</tr>
<tr>
<td>FROM Input In, Pars Pa, OntClass On</td>
</tr>
<tr>
<td>WHERE In.p_id = Pa.p_id</td>
</tr>
<tr>
<td>AND Pa.ClassID = On.ClassID</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>OntOutput SQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT l_id, p_id, ClassID, LV, Start, End, ws_id</td>
</tr>
<tr>
<td>FROM Output Out, Pars Pa, OntClass On</td>
</tr>
<tr>
<td>WHERE Out.p_id = Pa.p_id</td>
</tr>
<tr>
<td>AND Pa.ClassID = On.ClassID</td>
</tr>
</tbody>
</table>
```

**Fig. 14.** Sample web services for the semantic matching.
The dash link in the semantic composition graph represent this semantic matching.

Fig. 17 compares a Path table with an OntPath table using web services in Fig. 14. Due to the forth row in the OntEdge table, two more paths such as P5 and P6 are generated in the OntPath table.

Example 6. Assume that a user wants to find web services whose input is “City” and output is “SportsOK”. The path P4 and P5 in the Path table satisfy the use query. Because path P4 and P5 have more than two edges, we search the VisitedWS table to obtain the intermediate web services of each path. The final results are as follows: (WS1, WS2, WS4) and (WS1, WS3, WS4).

5.1.2. Degree of matching. As explained in Section 5.1.1, there are four types of semantic matching with ontology information: exact, plug-in, subsumes, and fails. By assigning a weight to each edge, we can support the semantic matching. For example, the values 0, 2, and 4 can be assigned to edges for the exact, plug-in, subsumes matchings, respectively. We do not generate an edge for the fail. Thus, the weight between two web services, ws1 in the OntInput and ws2 in OntOutput table, can be computed from the following formula:

\[ (-ws₁.LV - ws₂.LV) + ABS(ws₁.LV - ws₂.LV) + 3 \]

The OntEdge table in Fig. 16(a) can be considered as a materialized view defined by the SQL statement in Fig. 18, where the weight formula is included. Note that columns for the numbering scheme and LV is used in the SQL statement.

**Algorithm 5.** Web services composition supporting functionalities

```
Input: Q: user query, f₀: user’s functionality requirement;
Output: CandidateWSs: pairs of web services;

1: CandidateWSs ← WSC(Q); // Algorithm 4
2: foreach web service ws in CandidateWSs do
3:     if add the functionality of ws in the set Sₖ;
4:     else return false;
5: endforeach
```

5.2. Supporting functionalities of web services

In this subsection, we shall explain how to support functionalities of web services.

Although two services have the same type of inputs, outputs, preconditions, and effects, there may be quite different functionalities. This leads to the results of web services composition not satisfying the user. To resolve this problem, many researches try to use the functionality of a web service during web services compositions.

Generally, a functionality of a web service can be defined as an action–object pair of the web service (Shin et al., 2009). The action is what the functionality of the web service does and the object is the target of the action. For example, the functionality of a web service that returns departures from the name of the city can be specified as (Retrieve, DepartureInfo).

For adapting the functionalities of web services, first we extend a WS table to a FuncWS table by augmenting func_act and func_obj columns (shown in Fig. 19).

Second, we extended the web services composition search by adding a step for checking the functionalities of web services (Algorithm 5). A user gives a query Q and a functionality requirement f₀ to the system. Based on the user query, the system searches the candidate web services composition search in Line 1. After computing the functionalities of candidate web services denoted by $Sₖ$, the system checks the relationship between the user’s functionality requirement $f₀$ and the computed set of functionalities $Sₖ$ in Line 4.
6. Experimental results

In this section, we present the results of some experiments to analyze the performance of PSR system. Our experiments compared our methods which we refer BEP and IEP since no previous work is done on a relational database.

6.1. Experimental setup

We performed the experiments using two web services data sets: synthetic and WS-Challenge data sets. The synthetic data set composed of web services with a simple structure and is used to see the scalability of the PSR system. WSC data set is a benchmark data set which consists of web services with a complex structure.

The characteristics of the parameters and their values used to generate web services in the synthetic data sets as workload are summarized in Table 1. We varied the number of web services that were stored in PSR system from 1000 to 1,000,000. The parameter \( p_e \) is computed with the following formula:

\[
p_e = \frac{\text{# of connected edges}}{\text{# of total edges}}.
\]

It determines the probability of connected web services, which affects the number of composed web services.

Each web service in WSC data sets has multiple input parameters and output parameters. The data set “composition-50-32” means as follows: “50” denotes the number of web services composition and “32” means that a web service has 32–36 input and output parameters.

For the ontology information, we generated random trees for the web services datasets. The total number of keywords and total number of ontology concepts are 10 times and 5 times of the total number of web services, respectively. The maximum depth of ontology tree is obtained from the formula: \( \log N_w \). For example, when we use 10,000 web services for the experiment, the maximum depth of ontology tree is 4. The input and output parameters of web services were assigned to the nodes randomly.

For the user requests, we have generate random queries over the synthetic datasets and WSC datasets to find the web services composition search.

We ran our experiments on a 2.13 GHz Intel Core2 Duo machine with 2 GB memory running Windows XP Professional. Our algorithms were implemented using SQL and PL/SQL statements on the Oracle 10g Standard Edition. To make the experimental results more sound and reliable, we did every test in 10 times and we averaged all the reported experimental results over the entire repetitions.

6.2. Performance analysis

6.2.1. Web services composition

In this experiment, we investigate the performance advantage of PSR system only considering the exact matching.

6.2.1.1. Time for pre-computing web services

In order to examine the effect of the bulk loading EP-join, we first performed experiments where we measured the elapsed time for pre-computing web services composition. We varied the number of web services from 1000 to 1,000,000 and the value of \( p_e \) from 0.1 to 0.3.

Fig. 20 shows the results with varying number of web services and with varying the probability \( p_e \). Note that the log scale is used for denoting the values of the y-axis. As the time for pre-computing web services composition increased linearly with the number of web services in Fig. 20(a). In Fig. 20(b), elapsed time is increased slightly with the probability \( p_e \).

6.2.1.2. Time for web services composition search

To see effects of pre-computing web services composition, we conducted experiments measuring the execution time needed to process web services composition search. We varied the number of web services from 1000 to 1,000,000 and used random queries over the synthetic dataset to find the web services composition search. The execution time means the elapsed time for processing a set of user queries which is varied from 50 to 200 in steps of 50. The results are shown in Fig. 21. Note that the log scale is used for denoting the x-axis and the y-axis. The query time increases as the number of web service increases. However, the query time of PSR system does not increase considerably as compared to the increases of web services.

6.2.2. Performance comparison between IEP-Join and BEP-Join

In this experiment, we compared the bulk loading EP-Join (BEP-Join) with the incremental EP-Join (IEP-Join) in terms of execution time. For the IEP-Join, the total number of web services which were already existed in the PSR system was fixed to 100,000 or 1,000,000 and the number of inserted web services was varied from 1 to 10,000. For the BEP-Join, the total number of web services was var-

![Fig. 20. Time for pre-computing web services composition.](image)
ied from 100,001 to 110,000 or from 1,000,001 to 1,010,000. This is the same number of web services used for IEP-Join.

Fig. 22 shows the results. As the number of newly inserted web services increased, the execution time for IEP-join increased dramatically. This is explained from Table 2. As the number of inserted web services increased, the number of newly generated edges and paths increased linearly. The execution time for BEP-join remained the same, because the number of web services is changed slightly. The another observation from Fig. 22 is that when the number of newly inserted web services are more than 0.1% of the existed web services, the BEP-join shows the better performance than the IEP-Join.

6.2.3. Web services composition on WSC data sets

In this experiment, we demonstrate the effectiveness of the PSR system using the WSC (Web Service Challenge) data set in which each web service has a complex structure. Since the IEP-Join showed better performance than the BEP-join in the previous experiments, we used the IEP-join for pre-computing algorithm.

6.2.3.1. Pre-computing web services composition. In Table 3, we summarize the performance results for pre-computing web services composition. The number of web services in the “Composition-20-32” and “Composition-50-32” are 1000. The elapsed time for “Composition-20-32” is about 10 min, whereas the elapsed time for “Composition-50-32” is more than 14 h. This can be explained as follows: each web service in the data sets can have 32–36 input/output parameters, the number of edges and paths are increased sharply. For “Composition-50-32”, the number of paths are more than 29 million.

6.2.3.2. Web services composition search. Fig. 23 shows the performance results for queries Q1 through Q10. The number of results for web services composition are 10 for queries Q1 – Q3, 15 for queries Q4–Q6 and 20 for queries Q7–Q10, respectively. Although the pre-computing time for “Composition-50-32” takes about 14 h due to large numbers of edges and paths, the web services composition search takes a reasonable time (less than 7 s). In the case

<table>
<thead>
<tr>
<th>Data set</th>
<th>No. of edges</th>
<th>No. of paths</th>
<th>Pre-computing time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Composition-20-32</td>
<td>11,024</td>
<td>930,394</td>
<td>9 m 59 s</td>
</tr>
<tr>
<td>Composition-50-32</td>
<td>58,270</td>
<td>29,363,370</td>
<td>14 h 0 m 40 s</td>
</tr>
</tbody>
</table>

Table 2

The newly generated edges and paths.

<table>
<thead>
<tr>
<th>The number of inserted web services</th>
<th>100,000 web services</th>
<th>1,000,000 web services</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Edges</td>
<td>Paths</td>
</tr>
<tr>
<td>1</td>
<td>0.3</td>
<td>0.6</td>
</tr>
<tr>
<td>10</td>
<td>6</td>
<td>8.4</td>
</tr>
<tr>
<td>100</td>
<td>61.1</td>
<td>107.3</td>
</tr>
<tr>
<td>1,000</td>
<td>597</td>
<td>1065.2</td>
</tr>
<tr>
<td>10,000</td>
<td>6275.9</td>
<td>11230.1</td>
</tr>
</tbody>
</table>

Table 3

Pre-computing web services composition using WSC data sets.
of “composition-20-32”, the time is less than 0.14 s. In addition, this experiment shows that the web services composition search method based on a relational database works well on the WSC data.

6.2.4. Web services composition using ontology information

In this experiment, we investigate the performance advantage of FSR system when we are also given the ontology information. We experimented with the synthetic data sets for the semantic web services composition.

6.2.4.1. Time for pre-computing semantic web services composition

First, we compared the pre-computing time when ontology information was considered with the pre-computing time when no ontology information was given. Fig. 24 shows the results. As expected, the pre-computing with only exact matching shows better performance than pre-computing with ontology information. This is explained from Table 4. Table 4 shows the number of all possible web services composition for considering exact matching only and considering with ontology information. When the ontology information is given, the number of possible web services composition is about 20 times more than that of all possible web services considering only exact matching at maximum.

6.2.4.2. Time for semantic web services composition search

We also measured the query time with ontology information. The results are shown in Fig. 25. As expected, the processing with ontology information needs more time than processing with only exact matching. Table 5 depicts the maximum depth (length) of web services composition. The maximum depth of web services means the maximum number of web services participating web services composition. The maximum depth for considering only the exact matching remained less than 6, whereas the maximum depth for considering ontology information is more than 14. Moreover, the number of answers with ontology information is 7 times more than that of answers without ontology information at 1,000,000 web services as shown in Table 4. These cause the semantic web services composition search to take more time than only exact matching composition.

<table>
<thead>
<tr>
<th>Table 4</th>
<th>The number of all possible web services composition.</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N_n$</td>
<td>Only exact matching</td>
</tr>
<tr>
<td>---------</td>
<td>---------------------</td>
</tr>
<tr>
<td>1000</td>
<td>123</td>
</tr>
<tr>
<td>10,000</td>
<td>1095</td>
</tr>
<tr>
<td>100,000</td>
<td>11,333</td>
</tr>
<tr>
<td>1,000,000</td>
<td>110,989</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 5</th>
<th>The maximum depth of web services composition.</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N_n$</td>
<td>The maximum depth in only exact matching web services composition</td>
</tr>
<tr>
<td>---------</td>
<td>------------------------------------------------</td>
</tr>
<tr>
<td>1000</td>
<td>3</td>
</tr>
<tr>
<td>10,000</td>
<td>3</td>
</tr>
<tr>
<td>100,000</td>
<td>4</td>
</tr>
<tr>
<td>1,000,000</td>
<td>6</td>
</tr>
</tbody>
</table>
6.2.5. Web services composition supporting the functionalities

In this experiment, we investigate the performance advantage of PSR system when we are also given the functionalities of web services. We experimented with the synthetic data sets for the semantic web services composition.

Fig. 26(a) shows the experimental results when the functionalities of web services are considered. First, we varied the probability of connected edges \( p_e \) from 10\% to 30\% whereas we fixed the number of web service to 100,000. The performance of web services composition search considering the functionalities (referred to Functionality) is slower than that of original web services composition search (referred to Original). This is mainly due to the overhead for checking user's functionality requirements in the composition search results. Next, we varied the number of web services while we fixed the probability of connected edges \( p_e \) to 10\%. As we increased the number of web services, the performance gaps between two methods gradually increase.

It shows that considering the functionalities of web services affects the execution time of composition search. However, it is obvious that web services search considering functionalities will return more accurate since a user gives the functionalities of web services to the PSR system with his intention.

7. Conclusion

In this paper, we developed a web services composition search system called PSR. This is the first study for computing web services composition using a relational database instead of in-memory algorithms. The pre-computing algorithms works on a weighted, directed graph which is transformed from web services and ontology information. We considered the composed web services as intermediate vertices of paths between web services in the composition graph. The PSR system stores the graph into tables and computes answers for semantic web services composition search in advance by joining tables. Our experiments showed that our PSR system yields lower execution time for processing user queries and good scalability when handling a large number of web services and user queries.

Acknowledgements

This research was supported by National Research Foundation of Korea Grant funded by the Korean Government (2009-0071442), the Grant of Korean Ministry of Knowledge Economy under the ITRC(Information Technology Research Center) support program supervised by the NIPA(National IT Industry Promotion Agency) (NIPA-2010-(C1090-1031-0008)), the Grant of the Korean Ministry of Education, Science and Technology (The Regional Core Research Program/Institute of Logistics Information Technology) and Pusan National University Research Grant, 2011.

References


Daewook Lee is a research professor of Computer Science and Engineering at Sogang University. He received the BS and MS degrees in the Department of Computer Engineering from Kyungpook National University, Daegu, Korea in 1995 and 1997, and PhD degree in the School of Electrical Engineering and Computer Science from Seoul National University, Seoul, Korea in 2005, respectively. His research interests include XML filtering, XML indexing and query processing, Web services, RFID data management, multimedia databases and serious games.
Sangjun Lee received his BS and MS degrees in the Department of Computer Engineering from Seoul National University, Seoul, Korea, in 1996 and 1998, respectively and PhD in the School of Electrical Engineering and Computer Science, Seoul National University, Seoul, Korea in 2004. He is currently a associate professor of the School of Computing, Soongsil University, Seoul, Korea. His current research interests include database systems, Web services, systems and cloud computing.
Seog Park is a professor of Computer Science and Engineering at Sogang University. He received the BS degree in Computer Science from Seoul National University, Seoul, Korea in 1978, the MS and the PhD degrees in Computer Science from Korea Advanced Institute of Science and Technology (KAIST), Daejeon, Korea in 1980 and 1983, respectively. Since 1983, he has been working in the Department of Computer Science and Engineering, Sogang University. His major research areas are database security, real-time systems, data warehouse, digital library, multimedia database systems, role-based access control and web database. He is a member of the IEEE Computer Society, ACM and the Korean Institute of Information Scientists and Engineers (KIISE). Also, he has been a member of Database Systems for Advanced Application (DASFAA) Steering Committee since 1999.
Bonghee Hong received the MS and PhD degrees in Computer Engineering from Seoul National University, Seoul, Korea, in 1984 and 1988. In 1987, he joined the faculty of Computer Engineering of the Pusan National University (PNU). He is working as a Professor of Database in the Department of Computer Engineering at the PNU. He is a Director of Institute of Logistics Information Technology. He is also a steering committee member of DASFAA and was a co-chair of APWeb 2010. His research interests include theory of database systems, moving object databases, spatial databases, RFID system and RFID/RTLS/Sensor middleware.