Framework for Web Service Query Algebra and Optimization

QI YU
Virginia Tech, USA
and
ATHMAN BOUGUETTAYA
Virginia Tech, USA and CSIRO ICT Center, Australia

We present a query algebra that supports optimized access of Web services through service-oriented queries. The service query algebra is defined based on a formal service model that provides a high-level abstraction of Web services across an application domain. The algebra defines a set of algebraic operators. Algebraic service queries can be formulated using these operators. This allows users to query their desired services based on both functionality and quality. We provide the implementation of each algebraic operator. This enables the generation of Service Execution Plans (SEPs) that can be used by users to directly access services. We present an optimization algorithm by extending the Dynamic Programming (DP) approach to efficiently select the SEPs with the best user-desired quality. The experimental study validates the proposed algorithm by demonstrating significant performance improvement compared with the traditional DP approach.

Categories and Subject Descriptors: H.3.5 [**Information Storage and Retrieval**]: Online Information Services—*Web-based services*

General Terms: Algorithms, Design

Additional Key Words and Phrases: Web service, service oriented computing, service query, query optimization

ACM Reference Format:

Yu, Q. and Bouguettaya, A. 2008. Framework for Web service query algebra and optimization. ACM Trans. Web 2, 1, Article 6 (February 2008), 35 pages. DOI = 10.1145/1326561.1326567 http://doi.acm.org/10.1145/1326561.1326567

This work was supported by the National Science Foundation under the CNS—Cyber Trust program with contract 0627469.

Authors' addresses: Q. Yu, Department of Computer Science, Virginia Tech, Blacksburg, VA 24060; email: qyu@vt.edu; A. Bouguettaya, CSIRO ICT Center, Canberra, ACT, Australia; email: athman@cs.vt.edu.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permission@acm.org.
© 2008 ACM 1559-1131/2008/02-ART6 $5.00 DOI 10.1145/1326561.1326567 http://doi.acm.org/10.1145/1326561.1326567

1. INTRODUCTION

Service-oriented computing is emerging as a new computing paradigm for efficient deployment and access of the exponentially growing plethora of Web applications [Papazoglou et al. 2005]. The development of enabling technologies for such an infrastructure is expected to change the way of conducting business on the Web. *Web services* have become *de facto* the most significant technological by-product. Simply put, a Web service is a piece of software application whose interface and binding can be defined, described, and discovered as XML artifacts [Alonso et al. 2003]. It supports direct interactions with other software agents using XML-based messages exchanged via Internet-based protocols. Examples of Web services include online reservation, ticket purchase, stock trading, and auction.

The ability to *efficiently* access Web services is necessary, in light of the large and widely geographically disparate space of services. Using Web services would typically consist of invoking their operations by sending and receiving messages. However, complex applications, for example, a travel package that accesses multiple Web services, would need an *integrated* framework to *efficiently access* and *manipulate* Web services' functionalities. The increasing adoption of Web services requires a *systematic* support of query facilities. The *service-oriented queries* would enable users to access multiple Web services in a *transparent* and *efficient* manner. In addition, as Web services with *similar* functionality are expected to be provided by competing providers, a major challenge is devising *optimization* strategies for finding the "*best*" Web services or *composition* thereof with respect to the expected user-supplied *quality*.

Query optimization technologies have been intensively investigated [Chaudhuri 1998; Du et al. 1992; Florescu et al. 1999; Dalvi et al. 2001; Papadimitriou and Yannakakis 2001; Haas et al. 1997; Fernandez and Suciu 1998; Yerneni et al. 1999]. However, service-oriented queries are inherently different from data queries partially because of the rich semantics embodied in Web services. This poses several new research challenges:

—Web services have been so far mainly driven by standards. This is historically similar to the progress of DBMSs. The DBMS technology has a tremendous progress with the advent of the relational model which was instrumental in giving databases a solid theoretical foundation. Web services have yet to have such a solid theoretical underpinning.

—Web services are usually modeled as function calls, with focus on their input and output types. However, services with the same input-output types may provide totally different functionalities. For example, assume services S_i and S_j take a *string* as the input and output a *float number*. Also assume that S_i is a book price checking service, which takes the book title as its input and returns the book price. On the other hand, assume S_j is a stock checking service, which takes the stock name as its input and returns the stock price. Querying Web services based on such limited syntactic information may result in erroneous results. Necessary semantics need to be defined to extend the current service models.

—Web service operations may not be invoked in arbitrary orders. There may be *dependency constraints* between different service operations. The constraints may require that the invocation of some service operations occur only after their dependent operations have been successfully invoked. A simple example is that an online transaction service may require a login operation before making an order. Therefore, a service query should be answered by considering all the dependency constraints of the retrieved Web services.

—The proliferation of Web services is expected to introduce competition among multiple Web service providers. Querying services only based on functionality may result in large numbers of candidate services. Users may also have special requirements on the service qualities. In this case, the service query optimizer should not generate efficient query plans based only on performance but also differentiate competing Web services based on user expected *Quality of Web Service (QoWS)*.

Some preliminary research efforts are underway to provide query and optimization facilities for Web services [Srivastava et al. 2006; Dong et al. 2004; Ouzzani and Bouguettaya 2004]. These approaches mostly treat Web services as function calls that take some input and generate some output [Dong et al. 2004; Srivastava et al. 2006]. Web services are usually not queried based on their functionalities. In addition, traditional optimization techniques mainly focus on performance, that is, efficiently processing queries. Quality of services are not considered by the optimization process. Ouzzani and Bouguettaya [2004], proposed a query model that offers query optimization functionalities for Web services. The query model consists of three levels: *query level, virtual level*, and *concrete level*. The query model uses the predefined mapping rules to map relations defined at the query level to virtual operations defined at the virtual level. Users can thus directly use relations to query Web services. Our approach goes beyond this ad hoc query model by proposing a solid foundation, upon which algorithms can be developed for optimizing service queries.

We propose a query algebra that enables users to access Web services via service oriented queries. This foundational framework aims to layout a theoretical underpinning for the deployment of Web Service Management Systems (WSMSs) that would be to Web services what DBMSs have been to data [Yu et al. 2007]. The WSMS is proposed as a comprehensive system that offers query optimization [Ouzzani and Bouguettaya 2004], composition [Ponnekanti and Fox 2002; Casati and Shan 2001], transaction [Papazoglou 2003], security [Mecella et al. 2006; Bhatti et al. 2005], and other management supports [Papazoglou and van den Heuvel 2005; Casati et al. 2003] for accessing and managing Web services. In this paper, we focus on defining service algebraic operators and the physical implementation of these operators. This enables us to generate Service Execution Plans (SEPs) that can be directly used by users to access services. We also define a set of algebraic equivalent rules to perform algebraic optimization. We propose an efficient algorithm that extends the DP-based approach to perform service query optimization that can select the SEP with the best quality. The contributions of this article are summarized as follows:

—*Service Query Model*. We present a formal service model that captures the three key features of Web services: *functionality*, *behavior*, and *quality*. Functionality is specified by the operations offered by a Web service. Behavior reflects how the service operations can be invoked. It is decided by the dependency constraints between service operations. Quality determines the nonfunctional properties of a Web service.

—*Service Query Algebra*. We propose a service algebra based on the service model. The algebra consists of a set of algebraic operators. A set of algebraic equivalent rules is also present to transform user provided algebraic expression into the ones that can be more efficiently processed by the query processor. We provide the implementation of each algebraic operators. This enables the generation of SEPs.

—*Service Query Optimization*. Multiple SEPs might be generated by the query processor due to the competing service providers. We present a score function to calculate the quality of SEPs. We propose two algorithms to efficiently find the SEP with the best quality (i.e., highest score). The first algorithm is based on the traditional dynamic programming approach. The second algorithm extends the first one by incorporating a divide-and-conquer strategy. This strategy greatly improves the efficiency of the optimization algorithm.

—*Experimental Study*. We present an analytical model to evaluate the proposed optimization algorithms. We then conduct a set of experiments. The experiments compare the performance of the algorithms in terms of both efficiency and the quality of the generated SEPs.

The remainder of this article is organized as follows. In Section 2, we describe a scenario to motivate the need of a service query optimization framework. We present the service model and service algebra in Section 3 and 4 respectively. In Section 5, we present the implementation of the algebraic operators. In Section 6, we propose a QoWS model, which serves as the cost estimation criteria in the QoWS optimization. Based on the model, we propose two optimization algorithms. We present an analytical model in Section 7 and conduct experimental studies in Section 8. We overview the related work in Section 9. We provide some concluding remarks in Section 10.

2. CASE STUDY

As a way to motivate this work, we use an application from the *car brokerage* domain (see Figure 1). A typical scenario would be of a customer, say Mary, planning to buy a used car having a specific model, make, and mileage. She naturally wants to get the best deal. Assume that Mary has access to a Web service infrastructure where the different entities that play a role in the car purchase are represented by Web services. Examples of Web services that need to be accessed include *Car Purchase (CP)*, *Car Insurance (CI)*, and *FInancing (FI)*. A single Web service may provide multiple operations. Different operations may also have dependency relationships. For example, the `paymentHistory` and `financingQuote` operations are both offered by the *financing* service. The latter operation depends on the former operation; that is, the payment history

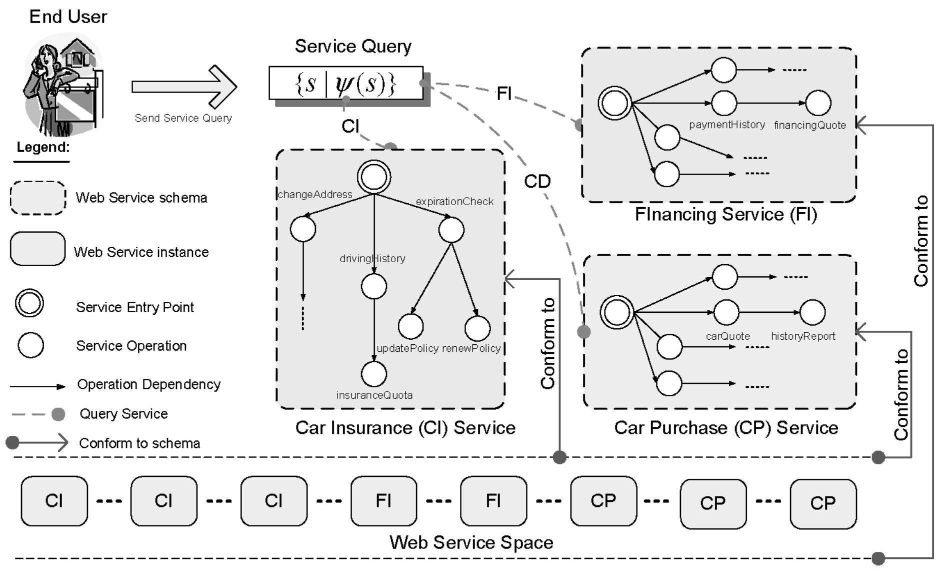


Fig. 1. The car brokerage scenario.

decides the financing quote. We also anticipate that there will be multiple competitors to provide each of the services mentioned. It is important that the users' quality requirements be reflected in the service query as criteria for service selection. To purchase an entire car package, Mary would first like to know the price quote of the selected car and the vehicle history report. She then needs to get the insurance quote. Finally, since Mary needs the financing assistance, she also wants to know the financing quote. In addition, Mary may have special requirements on the quality of the service operations. For example, she wants to spend less than 20 dollars to get the vehicle history report.

3. SERVICE QUERY MODEL

We present our service model in this section. The service model proposes and formally describes two key concepts: *service schema* and *service relation*. The service schema is defined to capture the key features of all Web services across an application domain. It provides a fixed vocabulary and enables the definition of the service query languages. A set of service instances that conform to a service schema form a service relation.

Finite State Machines (FSMs) and Petri-net have previously been proposed to model Web services [Berardi et al. 2005; Hamadi and Benatallah 2003]. However, these models are mainly designed for automating the composition of Web services [Pu et al. 2006]. Our proposed service model differs from these existing service models by providing foundational support for service query optimization. It is worth to note that the objective of this work is not to define a completely new model. Instead, we are inspired by the standard relational model and make some key extensions to it that enable service users to efficiently access services with their best desired quality. The benefit of presenting such a model is twofold. First, we can use this model to capture the rich semantics of Web services, including functionality, behavior, and quality. These features are of primary interest for users to access services, which also makes them fundamental for specifying service queries. In the proposed two-level service model, the graph-based service schema is used to capture the functionalities of Web services in terms of the operations they offer. It also captures the dependency relationships between the operations, which determine how these operations can be accessed (called behavior of the service). The service relation is used to capture the quality of the service providers. Second, we can leverage the existing technologies developed for the standard relational databases. For example, we can store our service relation in a relational database and use some relational operators to help implement the proposed service algebra (refer to Section 5 for details). In what follows, we first formally define several important concepts about the service schema. We then give the definition of the service relation.

Definition 3.1 (*Service Schema*). A service schema is defined as a tuple

$$\begin{aligned} \mathbf{S} &= (SG_1, SG_2, \ldots, SG_n, \mathcal{D}), \text{with} \\ SG_i &= (V_i, E_i, \epsilon_i),\ i = 1, \ldots, n \end{aligned}$$

is a directed acyclic graph (DAG), called *service graph* where

—$V_i = \{op_{ij} | 1 \leq j \leq m\}$ represents a set of service operations.

—ϵ_i is the root of the service graph. It represents the entry point, through which all other operations in the service graph can be accessed. ϵ_i can also be regarded as a special service operation, denoted by op_{i0}. A service graph has only one root.

—$E_i = \{e_{ij} | 1 \leq j \leq l\}$, represents the dependencies between two service operations from the same service graph, denoted by $\prec_{ii}$. $e_{ij} = (op, op')$ is an edge, where $op \in V_i, op' \in V_i$, and $op \neq op'$.

—$\mathcal{D} = \{D_{i,j} | 1 \leq i \leq n \wedge 1 \leq j \leq n \wedge i \neq j\}$, represents the dependencies between two non-root operations from *different* service graphs, denoted by $\prec_{ij}$. $D_{i,j} = \{e^k_{i,j} | 1 \leq k \leq l\}$ represents the dependencies between two non-root operations from service graph SG_i and SG_j. $e^k_{i,j} = (op, op')$ is an edge, where $op \in V_i$ and $op' \in V_j$.

—$SG' = SG_i \circ SG_j$, the concatenation of two service graphs is formed by coalescing the root of SG_i and SG_j. Furthermore, $V' = \{op | op \in V_i \vee op \in V_j\}$ and $E' = \{e | e \in E_i \vee e \in E_j \vee e \in D_{i,j}\}$. Figure 2 shows an example of the concatenation of two service graphs.

The dependency between two service operations is modeled as $op \prec op'$, where $\prec \in \{\prec_{ii}, \prec_{ij}\}$. $\prec_{ii}$ refers to the *intra-service dependency*, which can only be satisfied by invoking the two service operations by the specified order in the service graph. $\prec_{ij}$ refers to the *inter-service dependency*. It should be satisfied when multiple services are accessed. We assume in Definition 3.1 that there is no dependency between the roots of different service graphs. We also assume

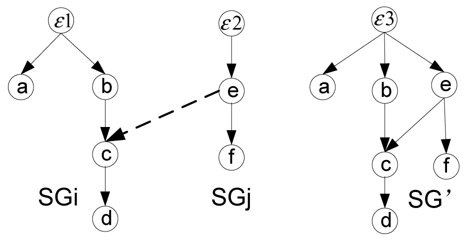


Fig. 2. Concatenation of service graphs.

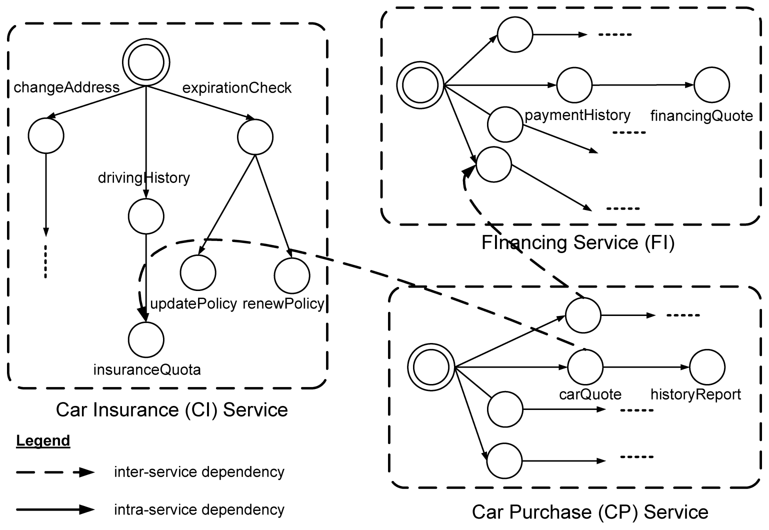


Fig. 3. The service schema for the car brokerage scenario.

that multiple dependency constraints on a single operation have an "And" relationship. For example, there are two dependency constraints on op_k, one with op_i and the other with op_j. In this case, both op_i and op_j should be accessed before op_k. It is also worth to note that when there is only one service graph in the service schema **S**, that is, $n = 1$ in Definition 3.1, **S** becomes a *single-graph* service schema.

Example 3.1. Figure 3 shows the service schema for the car brokerage service base. The service schema contains three service graphs, representing the *Car Purchase (CP)*, *Car Insurance (CI)*, and *FInancing (FI)* services. For example, in *CI*, there is a set of service operations, such as `drivingHistory` and `insuranceQuote`. These operations collectively represent the functionality of the *CI* Web service. The dependencies between service operations are captured by the edges in the service graph. For example, `drivingHistory` $\prec_{ii}$ `insuranceQuote` means that the execution of `insuranceQuote` depends on the result of `drivingHistory`. Service operations from different Web services could have an inter-service dependency. For example, there is a dependency between `carQuote` and `insuranceQuote`. It is denoted as `carQuote` $\prec_{ij}$ `insuranceQuote`.

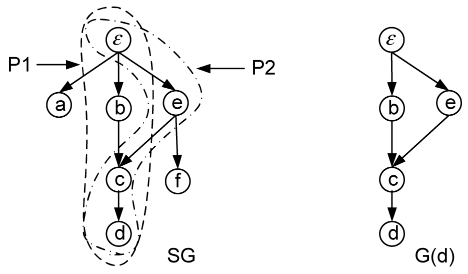


Fig. 4. An example of an operation graph.

In what follows, we define a set of key concepts derived from the service schema, including *service path, operation graph*, and *operation set graph*. We focus on identifying the important properties they offer that are fundamental to specifying and processing service queries.

Definition 3.2 (*Service Path*). For a service graph $SG = (V, E, \epsilon)$, we defined a service path $P_i = (\{op_{i1}, \ldots, op_{ij}, \ldots op_{ik}\}, E', \epsilon\})$ where ϵ is the root of SG, $E' \subseteq E$, and $k \geq 1$; $op_{ij} \in V$ for $1 \leq j \leq k$; and for each $op_{ij}, 0 \leq j \leq (k-1), \exists e_j \in E' : e_j = (op_{ij}, op_{i(j+1)})$ (note that when $j = 0$, op_{ij} becomes ϵ). P_i is an induced subgraph of service graph SG.

LEMMA 3.1. *For any service operation $op \in SG$, there must be at least one service path P that can reach op from ϵ.*

PROOF. This directly follows the definition of ϵ. Since ϵ is the entry point to access any other operation (including *op*) in the service graph, there must be at least one path from it to *op*. □

Definition 3.3 (*Operation Graph*). For a service graph $SG = (V, E, \epsilon)$, an operation graph $G(op)$ is the *union* of all the service paths in SG that lead to operation *op*, $G(op) = \cup Pi$, where $P_i = (\{op_{i1}, \ldots, op_{ij}, \ldots op\}, E_i, \epsilon\})$. $G(op)$ is an induced subgraph of the service graph SG. Figure 4 shows an operation graph $G(d)$, which is formed from SG by the union of two service paths, P_1 and P_2, that both lead to the service operation d.

Definition 3.4 (*Operation Set Graph*). For a service graph $SG = (V, E, \epsilon)$, we define an operation set graph $G(\mathbf{op}) = \cup_{i=1}^{k} G(op_i)$, where $\mathbf{op} = \{op_i | 1 \leq i \leq k\}$. $G(\mathbf{op})$ is an induced subgraph of service graph SG. For example, in Figure 4, the operation set graph for {a,d,f} is SG itself, that is, $G(\{a, d, f\}) = SG$.

Operation graph and operation set graph are central to service query specification and processing. We identify their key properties by using the concepts of *accessible operation* and *accessible graph*. It is worth noting that the root of a service graph is accessible because the root is defined as the entry point of a service. The formal definitions are given as follows.

Definition 3.5 (*Accessible Operation*). *op* is an operation in a service graph SG and a general graph G is a subgraph of SG. *op* is an accessible operation of G if the following two conditions are satisfied:

(i) $op \in G$;
(ii) $\forall op', (op', op) \in SG \Rightarrow op'$ is an accessible operation of G.

LEMMA 3.2. *An operation op is an accessible operation of G iff op and all of its preceding nodes in the service graph are included in G.*

PROOF. This directly follows from Definition 3.5. □

Definition 3.6 (*Accessible Graph*). G is an accessible graph if each operation in G is an accessible operation of G.

THEOREM 3.1. *An operation graph $G(op)$ is a minimal accessible graph for op.*

PROOF. We use two steps to prove this theorem. First, we prove that $G(op)$ is an accessible graph. Second, we prove its minimality.

We can prove $G(op)$ is an accessible graph by proving that op is an accessible operation of $G(op)$. Assume that op is not an accessible operation of $G(op)$. Since $op \in G(op)$, there exists an operation op' where op' is a preceding node of op in SG and op' is not a node of $G(op)$ (Lemma 3.2). Let P_1 be the path from op' to op. From Lemma 3.1, there is a path from ϵ to op', which is denoted by P_2. Connecting P_1 and P_2, we get a new path P_3 from ϵ to op, which is not included in $G(op)$. This contradicts definition 3.3. Therefore, no such op' exists. We can conclude that $G(op)$ is an accessible graph.

Assume that $G(op)$ is an accessible graph but not minimal for op. Therefore, there exists a graph $G'(op) = G(op) - op'$, where $op' \neq op$, such that $G'(op)$ is still an accessible graph and op is its accessible operation. From definition 3.3, there is a path to op which passes through op' in SG. Therefore, op' is a preceding node of op in SG. Since op is an accessible operation of $G'(op)$, from Lemma 3.2, we can get $op' \in G'(op)$. This contradicts the fact that op' is removed from $G'(op)$. Therefore, $G(op)$ is a accessible graph and minimal for op. □

THEOREM 3.2. *An operation set graph $G(\mathbf{op})$ is a minimal accessible graph for* $\mathbf{op}$.

PROOF. This directly follows from Definition 3.4 and Theorem 3.1. □

Remark 3.1. In a service query, users only need to specify the operation(s) they want to access (i.e., in a declarative way). An operation (set) graph will be generated when the query is processed. For example, a user wants `financingQuote` and formulates a service query to access it. An operation graph G(financingQuote) will be generated. The query processor will use the operation (set) graph as the single-graph service schema to generate service execution plans (i.e., SEPs). Since an operation (set) graph is an accessible graph, it guarantees that the operations specified in the service query are accessible through the generated SEPs. In addition, the minimality of the graph also guarantees that only minimum number of service operations (i.e., the ones that the operations in the query depends on) are included in the SEPs.

Table I. QoWS Parameters

Parameter	Definition	Domain	Index
Latency	$\text{Time}_{process}(op) + \text{Time}_{results}(op)$ where $\text{Time}_{process}$ is the time to process *op* and $\text{Time}_{results}$ is the time to transmit/receive the results	number	1
Reliability	$\text{N}_{success}(op)/\text{N}_{invoked}(op)$ where $\text{N}_{success}$ is the number of times that *op* has been successfully executed and $\text{N}_{invoked}$ is the total number of invocations	number	2
Availability	UpTime(*op*)/TotalTime(*op*) where UpTime is the time *op* was accessible during the total measurement time TotalTime	number	3
Fee	Dollar amount to execute the operation	number	4
Reputation	$\sum_{u=1}^{n} Ranking_u(op)/n$, $1 \leq \text{Reputation} \leq 10$ where $Ranking_u$ is the ranking by user u and n is the number of the times *op* has been ranked	number	5

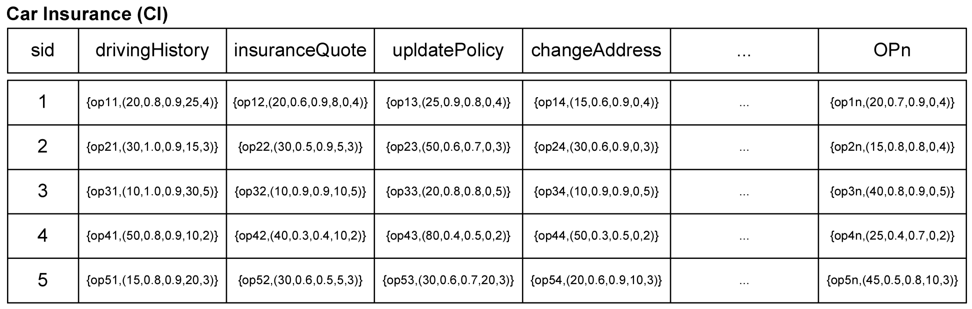
We have now defined the service schema related concepts and identified the key properties they offer for querying service. The service relation defines a set of service instances that conform to the service schema. The service instances offer the operations and follow the dependency constraints defined in the service graphs. However, since the service instances are provided by different service providers, they may have different quality properties. In what follows, we first define a QoWS model to capture the quality features of services. We then give the definition of a service relation.

Definition 3.7 (*QoWS Model*). The QoWS model formally defines a set of quality parameters for Web services (see Table I). It divides the quality parameters into two categories: *runtime* quality and *business* quality.

—*Runtime quality.* It represents the measurement of properties that are related to the execution of an operation *op*. We identify three runtime quality parameters: *latency*, *reliability*, and *availability*. The *latency* measures the expected delay between the moment when *op* is initiated and the time *op* sends the results. The *reliability* of *op* is the ability of the operation to be executed within the maximum expected time frame. The *availability* is the probability that the operation is accessible. Service providers could publish runtime qualities of their Web service operations in the service description or offer mechanisms to query them.

—*Business quality.* It allows the assessment of an operation *op* from a business perspective. We identify two business quality parameters: *fee* and *reputation*. The *fee* gives the dollar amount required to execute *op*. The *reputation* of *op* is a measure of the operation's trustworthiness. It mainly depends on the ratio to which the actual provision of the service is compliant with its promised one. The fee quality can be obtained based on the service providers' advertisement in the service description whereas the reputation is based on the ranking of the end-users.

The values of the parameters defined in the QoWS model are from the number domain, which consists of integer, float, and double. The proposed QoWS



Car Insurance (CI)

sid	drivingHistory	insuranceQuote	upldatePolicy	changeAddress	...	OPn
1	{op11,(20,0.8,0.9,25,4)}	{op12,(20,0.6,0.9,8,0,4)}	{op13,(25,0.9,0.8,0,4)}	{op14,(15,0.6,0.9,0,4)}	...	{op1n,(20,0.7,0.9,0,4)}
2	{op21,(30,1.0,0.9,15,3)}	{op22,(30,0.5,0.9,5,3)}	{op23,(50,0.6,0.7,0,3)}	{op24,(30,0.6,0.9,0,3)}	...	{op2n,(15,0.8,0.8,0,4)}
3	{op31,(10,1.0,0.9,30,5)}	{op32,(10,0.9,0.9,10,5)}	{op33,(20,0.8,0.8,0,5)}	{op34,(10,0.9,0.9,0,5)}	...	{op3n,(40,0.8,0.9,0,5)}
4	{op41,(50,0.8,0.9,10,2)}	{op42,(40,0.3,0.4,10,2)}	{op43,(80,0.4,0.5,0,2)}	{op44,(50,0.3,0.5,0,2)}	...	{op4n,(25,0.4,0.7,0,2)}
5	{op51,(15,0.8,0.9,20,3)}	{op52,(30,0.6,0.5,5,3)}	{op53,(30,0.6,0.7,20,3)}	{op54,(20,0.6,0.9,10,3)}	...	{op5n,(45,0.5,0.8,10,3)}

Fig. 5. An example of CI service relation.

model can be extended by adding other quality parameters. The index number given in Table I will be used by the *labeling function* defined in the service relation.

Definition 3.8 (*Service Relation*). A service relation SR with a service graph $SG = (V, E, \epsilon)$ is defined as a set of service instances $\mathcal{I} = \{(sid, op_1, \ldots, op_n)\}$, where

—sid is the unique service id;

—op is a service operation and defined as a pair $op = (opid, \lambda(op))$, where $opid$ is the operation id and λ is a labeling function that assign to each service operation op a set of values to its QoWS parameters, denoted by $\mathbf{Q} = \bigcup_{i=1}^{k} Q_i$. $op \xrightarrow{\lambda} \mathbf{Q}$ gives the quality parameter values for op. $\lambda_i(op) = Q_i$ specifies the ith quality parameter for op, where i is the index for the quality parameter. Table I specifies the indices for all the QoWS parameters. We will use these indices to refer to the different QoWS parameters in later sections.

—Each service instance $\mathcal{I}$ in SR conforms to the service graph SG; that is, operations in $\mathcal{I}$ are defined in SG and the operations follow the dependency constraints specified by SG.

We define the domain of $\lambda(op)$ as $dom(\lambda(op)) = dom\{(\lambda_i(op))|1 \leq i \leq m\}$, where m is the number of QoWS parameters that can be applied to op. Therefore, we can further define $dom(op) = \{dom(opid), dom(\lambda_i(op))\}$.

Based on the domain definition, we can restate the above definition of a service relation as follows. A service relation SR is a $(n+1)$-degree relation on the domains of $dom(sid), dom(op_1), dom(op_2), \ldots, dom(op_n)$, where $op_i \in SG$ for i = $1, \ldots,$ n,

$$r(SG) \subseteq (dom(sid) \times dom(op_1) \times \cdots \times dom(op_n)).$$

Figure 5 shows an example of CI service relation. The service relation contains 5 service instances (aka service tuples) and has $(\mathrm{n}+1)$ fields, which correspond to the sid and n service operations offered by the service instances.

The *functionality, behavior*, and *quality* parameters of the Web service are captured in the service model. This provides fundamental support for querying services. Since the functionality of a Web service is offered through a set of service operations, the vertices in V collectively represent the functionality of the Web service. The behavior of the service is reflected by the operation graphs (or operation set graph), which contain a set of service operations and the dependency relationship between them. The quality parameters can be attached to the service operations to evaluate QoWS parameters of operations from service instances.

4. SERVICE QUERY ALGEBRA

We define a service query algebra that enables the specification of algebraic service queries. The proposed service algebra contains three major operators that help service users query their desired services:

—*Functional Map (*`F-map`*).* It facilitates users to locate and invoke their desired functionalities in terms of service operations by making use of the key properties provided by the operation (set) graphs.
—*Quality-Based Selection (*`Q-select`*).* It allows users to locate service instances (i.e., service providers) with their desired quality.
—*Composition (*`Compose`*).* It enables service composition when users need to access multiple services.

Service users can use the algebraic operators to access one or multiple services with their desired functionality and quality. A service query is typically specified with the combination of the above three operators (More detailed definition for each of these operators are given in Section 4.1). Querying services is an integrated process that requires to query both the service graphs defined in the service schema and the service relation. Processing an algebraic service query will result in a service relation SR' and a service graph, G_s, that serves as the schema for the retrieved service relation. G_s is constructed based on the operations specified in the service query (i.e., the functionalities a user wants to access). G_s consists of the operations in the service query and all their dependent operations. It also specifies the dependency constraints between these operations. In this section, we first present the service query algebra. We then present a set of algebraic equivalent rules that enable the query optimizer to rewrite the algebraic expressions.

4.1 Algebraic Operators

The service algebra consists of three major operators: `F-map` (χ), `Q-select` (δ), and `Compose` ($\oplus$). The algebra operators are applied to a service relation and produce a new service relation.

4.1.1 *F-map.* The `F-map` operator, denoted by χ, is used to map a service relation SR onto a subset of (user selected) service operations and result in a service relation SR'. The service schema of SR' is a service graph G_s, which is an operation set graph built upon the user specified operations and the service graph of SR. The `F-map` operator is also called **functional map** because users can choose their desired functionality (in terms of service operations) using this operator. `F-map` is formally defined as follows:

$$\chi_{\mathbf{op}}(SR) = \{s.sid, s[G(\mathbf{op})] | SR(s)\},$$

where

—**op** is the set of service operations that a user wants to select to achieve the desired functionality.

—G(**op**) is an operation set graph. It contains the service operations specified by the service query and all their dependent operations. Recall that in Theorem 3.1, we proved that an operation graph is a minimum accessible graph for *op*. It consists of all the necessary operations that make the operation specified in the service query accessible. It also specifies the dependency constraints for accessing these operations. Therefore, the operation (set) graph can be regarded as the schema (i.e., G_s) for the retrieved service relation.

—The [ ] operator selects the operations from s with respect to G(**op**); that is, the retrieved service tuple only contains all the operations of G(**op**).

Example 4.1. An algebraic service query that retrieves the insuranceQuote operation from the CI service relation is interpreted as follows:

$$\chi_{\{insuranceQuote\}}(CI) = \{s.sid, s[G(\{insuranceQuote\}]|CI(s)\}.$$

4.1.2 *Q-Select.* The `Q-select` operator, denoted by δ, is used to select a subset of service tuples from service relation SR that satisfy some *quality requirement*. The selected tuples form a service relation SR', which has the same service schema as SR. `Q-select` is also called **quality select**. The quality requirement is specified by the *select quality predicate* p_s. `Q-select` is formally defined as follows:

$$\delta_{p_s}(SR) = \{s|SR(s) \wedge p_s(s)\},$$

where

—*SR* is a general service algebra expression which results in a service relation; *SR*(*s*) takes *True* if *s* belongs to the generated service relation, *False* otherwise.

—p_s is a select quality predicate, which connects a set of *clauses* using the boolean operators: conjunction ($\wedge$), disjunction ($\vee$), and negation ($\neg$). A clause has the form of $x\ \theta_q\ c$ and returns a boolean, where (i) θ_q is a QoWS parameter comparison operator taken from $\{=, >, \geq, <, \leq, \neq\}$; (ii) x is a QoWS parameter and c is a constant.

Example 4.2. An algebraic service query that retrieves the *CI* service instances with a vehicle history report cost less than 20 dollars can be interpreted as follows:

$$\delta_{\lambda_4(drivingHisotry)\leq 20}(CI) = \{s|CI(s) \wedge \lambda_4(s.drivingHisotry) \leq 20\}$$

The following algebra expression combines the `F-map` and the `Q-select` operators:

$$\chi_{\{insuranceQuote\}}\left(\delta_{\lambda_5(insuranceQuote)\geq 3}(CI)\right)$$
$$= \{s.sid, s[G(insuranceQuote)]|CI(s) \wedge \lambda_5(s.insuranceQuote) \geq 3\}.$$

4.1.3 *Compose.* The Compose operator, denoted by $\oplus$, combines two service relations SR_1 and SR_2 into a single service relation SR'. The service schema of SR' is the concatenation of the service graphs of SR_1 and SR_2; that is, $G_s = G_1 \circ G_2$. It is used to address the complex service queries that require the cooperation of multiple services. Users can specify their quality requirement over multiple services by using the *compose quality predicate* p_c. These requirements will be used to select service tuples from the combined service relation. Compose is formally defined as follows:

$$SR_1 \oplus_{p_c} SR_2 = \{s.sid, s[G_1 \circ G_2] | (\exists s_1)(\exists s_2)$$
$$(SR_1(s_1) \wedge SR_2(s_2) \wedge s.\mathbf{op} == (s_1.\mathbf{op} \circ s_2.\mathbf{op}) \wedge p_c(s))\},$$

where

—G_1 and G_2 are the service graphs of SR_1 and SR_2 respectively. $G_1 \circ G_2$ represents graph concatenation (see Definition 3.1 for details).

—p_c is a *quality predicate*, which connects a set of clauses using the conjunction ($\wedge$) operator. Each clause takes the form of $x \ \theta_q \ y$, where x is a QoWS parameter of a service operation from SR_1 and y is a QoWS parameter of a service operation from SR_2.

—$s.\mathbf{op} == t.\mathbf{op}$ takes a *True* value if s and t have the same set of service operations (i.e., the signatures are the same) whereas it takes a *False* value otherwise, where $s.\mathbf{op}$ denotes the service operations of service tuple s and $t.\mathbf{op}$ denotes the service operations of service tuple t.

—$s_1.\mathbf{op} \circ s_2.\mathbf{op}$ represents service tuple concatenation. If s_1 and s_2 have m and n service operations respectively, the result of service tuple concatenation will be an operation set with $(m + n)$ service operations,[1] with the first m operations from s_1 and the following n operations from s_2.

Example 4.3. The following algebraic expression applies to two service relations, *CP* and *CI*. It retrieves the combined tuples, in which the insuranceQuote operation has a higher availability than the carQuote operation.

$$\chi_{\{op_1, op_2\}}(CP \oplus_{\lambda_3(op_1) < \lambda_3(op_2)} CI) = \{s.sid, s[G(op_1) \circ G(op_2)] | (\exists cp)(\exists ci$$
$$(CP(cp) \wedge CI(ci) \wedge s.\mathbf{op} == (cp.\mathbf{op} \circ ci.\mathbf{op}) \wedge \ \lambda_3(op_1) < \lambda_3(op_2))\}$$

where op_1 and op_2 represent the service operations carQuote and insuranceQuote respectively; $\lambda_3(op_1)$ and $\lambda_3(op_2)$ refer to the availability of op_1 and op_2. The query combines the Compose operator and the F-map operator. The schema of the resulted service relation SR' is the concatenation of two operation (set) graphs, that is, $G_s = G(op_1) \circ G(op_2)$.

If a user does not specify any quality predicate, p_c becomes empty. We define a Compose operator with empty quality predicate as Crossover, denoted as $\otimes$. Crossover is interpreted the same way as Compose by only removing the p_c part.

[1]The service schema is designed to have different service graphs with different functionalities (i.e., different set of operations). In this regards, the set of operations for two service graphs are disjoint.

Table II. Algebraic Equivalent Rules

1. Associative rule $SR_1\Omega(SR_2\Omega SR_3) = (SR_1\Omega SR_2)\Omega SR_3,\ \forall\Omega \in \{\oplus_p, \otimes\}$ if p is applicable to operations from SR_1 and SR_2
2. Communicative rule $SR_1\Omega SR_2 = SR_2\Omega SR_1,\ \forall\Omega \in \{\oplus_p, \otimes\}$
3. Cascading rule   **3.1.** $\chi_{\mathbf{op}_1}(\chi_{\mathbf{op}_2}(\ldots(\chi_{\mathbf{op}_n}(SR))) = \chi_{\mathbf{op}_1\cup\mathbf{op}_2\ldots\cup\mathbf{op}_n}(SR)$ **3.2.** $\delta_{p_1}(\delta_{p_2}(\ldots(\delta_{p_n}(SR))) = \delta_{p_1\wedge p_2\ldots\wedge p_n}(SR)$
4. Swapping rule   **4.1.** $\chi_{\mathbf{op}}(\delta_p(SR)) = \delta_p(\chi_{\mathbf{op}}(SR))$, if operations in p are only from **op**   **4.2.** $\chi_{\mathbf{op_1}\cup\mathbf{op_2}}(SR_1\Omega SR_2) = (\chi_{\mathbf{op_1}}(SR_1))\Omega(\chi_{\mathbf{op_2}}(SR_2))$ if $\mathbf{op}_1$ and $\mathbf{op}_2$ are the operations from SR_1 and SR_2 respectively   and p is applicable to operations from $\chi_{\mathbf{op_1}}(SR_1)$ and $\chi_{\mathbf{op_2}}(SR_2)$ **4.3.** $\delta_{p_1\wedge p_2}(SR_1\Omega SR_2) = (\delta_{p_1}(SR_1))\Omega(\delta_{p_2}(SR_2)), \forall\Omega \in \{\oplus_p, \otimes\}$ if operations in p_1 and p_2 are only from SR_1 and SR_2 respectively   **4.4.** $\Omega_1(SR_1\Omega_2 SR_2) = (\Omega_1(SR_1))\Omega_2(\Omega_1(SR_2))$, $\forall\Omega_1 \in \{\delta_p, \chi_{\mathbf{op}}\},\ \forall\Omega_2 \in \{\oplus_p, \otimes\}$

4.2 Algebraic Equivalent Rules

We present a set of *algebraic equivalent rules* in this section. Algebraic rewriting can be performed based on these rules. This enables algebraic optimization to generate efficient Service Query Plans (SQPs). Table II gives the details of each of these algebraic equivalent rules.

The correctness of the algebraic rules can be proved directly from the definition of our service algebra. These rules lay out a foundation for algebraic rewriting. We now present our heuristic rules for finding efficient SQPs. We assume that the internal form of a service query is implemented using a parse tree.

(1) Split the `Q-select` with multiclause predicates into a set of cascading `Q-selects` with single-clause predicates by using cascading rule **3.2.**

(2) Move `Q-selects` towards the leaves of the parse tree by using swapping rules, **4.1, 4.3**, and **4.4**

(3) Split the `F-map` with multiple sets of operations into a set of cascading `F-maps` with single set of operations by using cascading rule **3.1.**

(4) Move `F-maps` towards the leaves of the parse tree by using swapping rules, **4.2, 4.3**, and **4.4**.

(5) Combine cascading `Q-selects` and `F-maps` into a single `Q-select`, a single `F-map`, or a `Q-select` followed by an `F-map` by using cascading rules, **3.1, 3.2**, and swapping rule **4.1**.

5. IMPLEMENTING THE ALGEBRAIC OPERATORS

We present the implementation of the algebraic operators in this section. This enables the generation of SEPs that can be directly used by service users to access services. In relational databases, there is a strong correspondence between algebraic operators and the low-level primitives of the physical system [Selinger et al. 1979]. This correspondence comes from the mapping between relations

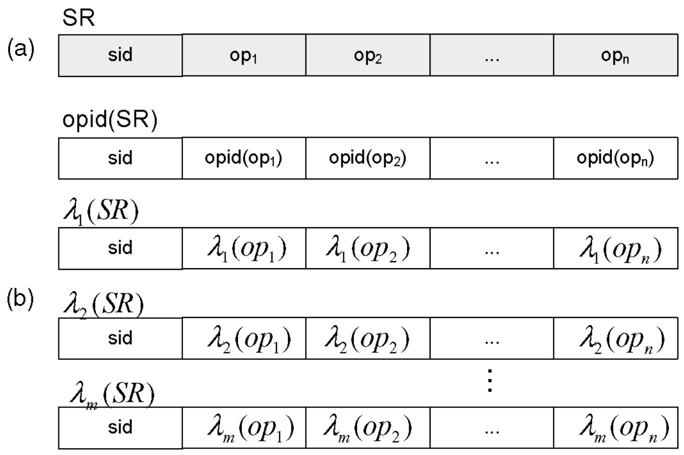


Fig. 6. Normalizing service relations into 1NF. (a) A service relation *SR*. (b) Decomposing *SR* into 1NF relations opid(*SR*), $\lambda_1(SR), \ldots, \lambda_m(SR)$.

and files, and tuples and records [Straube and Özsu 1995]. In our service query framework, the service execution plan depends on both the service instances and the *shape* of the service graph. The service instances are offered by the actual service providers. The service graph serves as the schema of the service instances. It specifies the dependency constraints of accessing the service operations in the service instances.

5.1 Storing the Service Relations

We leverage the relational database approach to store service instances. A service relation can be mapped to a set of database relations and stored in a relational database. Service relations allow *nonatomic* attributes. For example, a service operation is a composite attribute, which consists of an operation id (*opid*) and a set of QoWS values ($\lambda(op)$). This makes a service relation a *nested relation*. Therefore, a service relation contradicts with the First Normal Form (**1NF**) of relational databases. To normalize service relations into 1NF, we need to remove the nested relation attributes into new relations and propagate the primary key into it. A service relation

$$SR(sid, \{op_1(opid, \lambda(op_1))\}, \ldots, \{op_n(opid, \lambda(op_n))\})$$

is decomposed into $(m+1)$ database relations:

$$opid(SR)(sid, opid(op_1), \ldots, opid(op_n)) \qquad (1)$$
$$\lambda_1(SR)(sid, \lambda_1(op_1), \ldots, \lambda_1(op_n)) \qquad (2)$$
$$\ldots$$
$$\lambda_m(SR)(sid, \lambda_m(op_1), \ldots, \lambda_m(op_n)) \qquad (m+1).$$

Figure 6 illustrates the normalization process. *SR* is a service relation (see Figure 6(a)). The op_i attribute is multivalued, which contradicts with 1NF. The normalization decomposes *SR* into $(m+1)$ 1NF relations, as shown in

```
ALGORITHM $OSG_{gen}$
Input: A set of service operations $\mathbf{OP}=\{op_1, ..., op_n\}$,
       A service graph SG.
Output: An operation set graph OSG.
1.  $OSG = \phi$;
2.  for each $op \in$ OP
3.      $OG = OG_{gen}(SG, op)$;
4.      $OSG = OSG \cup OG$;
FUNCTION $OG_{gen}$
Input: A service graph $SG$, a service operation $op$
Output: An operation graph OG
5.  $OG.E=\phi$, $OG.V=\phi$;
6.  $OG.V = OG.V \cup \{op\}$;
7.  for each $e \in SG.E$ do
8.      if (e.to == op) then
9.          $OG.E = OG.E \cup \{e\}$
10.         $op' = e.from$;
11.  $OG' = OG_{gen}(OG, op')$;
12.  $OG = OG \cup OG'$;
```

Fig. 7. Operation set graph generation algorithm.

Figure 6(b). Normalization transforms a service relation into a set of database relations. Service relations can thus be stored in relational databases.

5.2 Implementing the Service Algebra

New service graphs might need to be generated when processing the algebraic service queries. Generating a new service graph is usually not as straightforward as generating a new relational data schema. Among the three algebraic operators, `F-map` and `Compose` require the generation of new service graphs. Therefore, implementation of these two operators consists of two major tasks: *generating the service graph* and *retrieving service instances*. The `Q-select` operator does not involve any update of the service graph. It only retrieves the service instances based on the quality requirement.

5.2.1 *F-map. Service Graph Generation.* The `F-map` operator employs an algorithm called OSG_{gen} (i.e., Operation-Set-Graph generation) to output an operation set graph given a set of service operations and a service graph. The resultant operation set graph will be the schema for the new service relation generated by the `F-map` operator. Figure 7 illustrates the Operation-set-graph generation process.

Service Instance Retrieval. The `F-map` operator performs database projections on each of the quality value relations and the service id relation, i.e., $\Pi_{opid(Gop_1),...,opid(Gop_k)}(opid(SR))$, $\Pi_{\lambda_1(Gop_1),...,\lambda_1(Gop_k)}(\lambda_1(SR)), \ldots,$ and $\Pi_{\lambda_m(Gop_1),...,\lambda_m(Gop_k)}(\lambda_m(SR))$, where $Gop_1, \ldots,$ and Gop_k are the operations from service graph G.

```
ALGORITHM $G_{con}$
Input: Two service graphs $SG_1$, $SG_2$,
       A edge set $E_D$={$e_1, e_2, ..., e_n$},
Output: A service graph SG.
1. $SG.V = \phi$, $SG.E = \phi$
2. $SG.V = SG_1.V \cup SG_2.V$
3. $SG.E = SG_1.E \cup SG_2.E \cup E_D$;
4. $SG$=$SG$-{$SG_1.\epsilon$, $SG_2.\epsilon$};
5. $SG.\epsilon = \epsilon_0$
6. for each $op \in SG.V$
7.     if op does not have any incoming edge
8.         $e' = \{\epsilon, op\}$
9.         $SG.E = SG.E \cup$ {e'}
```

Fig. 8. Service graph concatenation algorithm.

5.2.2 *Q-select. Service Instance Retrieval.* The `Q-select` operator uses five steps to retrieve service instances:

(1) Divide the selection predicate $p(\lambda_1, \lambda_2 \ldots, \lambda_k)$ into k subpredicates $p(\lambda_1)$, $p(\lambda_2) \ldots, p(\lambda_k)$, where k is the number of distinct quality parameters in the selection predicate.
(2) Use the subpredicates to perform database selection, $\sigma_{p(\lambda_1)}(\lambda_1(SR))$, $\sigma_{p(\lambda_2)}(\lambda_2(SR)) \ldots$, and $\sigma_{p(\lambda_k)}(\lambda_k(SR))$.
(3) Perform database projection on the resultant relations from step 2 to retrieve the service id, $sid_{\lambda_1} = \Pi_{sid}(\sigma_{p(\lambda_1)}(\lambda_1(SR)))$, $sid_{\lambda_2} = \Pi_{sid}(\sigma_{p(\lambda_2)}(\lambda_2(SR))), \ldots$, and $sid_{\lambda_k} = \Pi_{sid}(\sigma_{p(\lambda_k)}(\lambda_k(SR)))$.
(4) Combine $sid_{\lambda_1}, sid_{\lambda_2}, \ldots, sid_{\lambda_k}$ to generate the relation SID, which contains all the sids to be retrieved. The combination process is performed based on the Boolean operators used to connect different quality parameter clauses in $p(\lambda_1, \lambda_2 \ldots, \lambda_k)$. The Boolean operator can take one of the following two forms: $\vee$ and $\wedge$. For example, if $p = c(\lambda_1) \wedge c(\lambda_2)$, $SID = sid_{\lambda_1} \cap sid_{\lambda_2}$; if $p = c(\lambda_1) \vee c(\lambda_2)$, $SID = sid_{\lambda_1} \cup sid_{\lambda_2}$
(5) Perform database natural join, $opid(SR) \infty SID$, $\lambda_1(SR) \infty SID, \ldots$, and $\lambda_k(SR) \infty SID$ to retrieve the operation id and quality values for the retrieved service instances.

5.2.3 *Compose. Service Graph Generation.* The `Compose` operator employs an algorithm called G_{con} (i.e., Graph concatenation) to output a new service graph given two service graphs and a set of edges representing interservice dependencies. The resultant service graph will be the schema for the new service relation generated by the `Compose` operator. Figure 8 illustrates the graph concatenation process.

Service Instance Retrieval. The `Compose` operator uses five steps to retrieve service instances:

Table III. Complexity of Service Algebraic Operators

Algebraic Operator	Complexity
F-map	$O(n \times (V + E) + m \times t)$
Q-select	$O((k + m) \times t \times s_F \times log\ t)$
Compose	$O((V + E) + (k \times t + m) \times t \times log\ t)$
Crossover	$O((V + E) + (m + 1) \times t^2)$

(1) Divide the selection predicate p into a set of subpredicates, each of which contains a single clause.
(2) Perform database θ-join, $\lambda_i(SR_1) \bowtie_{p(\lambda_i,\lambda_j)} \lambda_j(SR_2)$, where $p(\lambda_i, \lambda_j)$ is a subpredicate containing λ_i and λ_j and λ_i. λ_j can refer to the same quality parameter, for example, $p = \lambda_3(carQuote) < \lambda_3(insuranceQuote)$.
(3) Perform database projection on the resultant relations from step 2 to retrieve the service id, $sid_{\lambda_i,j} = \Pi_{sid_i,sid_j}(\lambda_i(SR_1) \bowtie_{p(\lambda_i,\lambda_j)} \lambda_j(SR_2))$.
(4) Combine $sid_{\lambda_i,\lambda_j}$ to generate the relation SID, which contains all the sids to be retrieved. The combination process is similar to step 4 of the `Q-select` operator.
(5) Perform database natural join, $(opid(SR_1)\infty SID)\infty(opid(SR_2)$, $(\lambda_1(SR_1)\infty SID)\infty\lambda_1(SR_2), \ldots$, and $(\lambda_k(SR_1)\infty SID)\infty\lambda_k(SR_2)$ to retrieve the operation id and quality values for the retrieved service instances.

5.3 Complexity of Service Algebraic Operators

We analyze the complexity of the service algebraic operators in this section. The complexity is defined with respect to the cardinalities of the service relations that are independent of the physical implementation details. We assume the cardinality of the original service relations is t. Table III summarizes the complexity of each service algebraic operator.

F-map. The `F-map` operator first generates a new service graph by applying OSG_{gen} that has a complexity of $O(n \times (V + E))$. It then retrieves the service instances by performing projection on each relation, include the operation id relation and the m quality value relation. This requires a complexity of $O(m \times t)$. Therefore, the overall complexity is $O(n \times (V + E) + m \times t)$.

Q-select. The `Q-select` operator uses five major steps for retrieving service instances. We analyze the complexity of each step and derive the total complexity in the end.

(1) Step 1 divides a complex selection predicate into k subpredicates. It requires one pass of the k subpredicates and has a complexity of $O(k)$.
(2) Step 2 performs selection using the k subpredicates on k quality value relations respectively. The complexity is $O(k \times t)$.
(3) Step 3 projects the resultant quality value relation onto the sid attribute. We assume that the average selection factor in Step 2 is s_F. Therefore, the complexity of step 3 is $O(k \times t \times s_F)$.
(4) Step 4 uses the set operators (i.e., $\cap$ and $\cup$) to get the final sids. To eliminate duplicates, we assume that the set operators usually need to sort

the relations on the sids and then compare the sids from both relations. Therefore, the complexity is $O((k-1) \times t \times s_F \times log\ (t \times s_F))$.

(5) Step 5 joins the final sids with the operation id relation and all the m quality value relation. We assume that natural join also needs to sort the relations on sids. Therefore, the complexity is $O(m \times t \times log\ t)$.

We can derive the overall complexity of the `Q-select` operator as:

$$O(k + k \times t + k \times t \times s_F + (k-1) \times t \times s_F \times log\ t \times s_F + m \times t \times log\ t) \\ = O((k+m) \times t \times s_F \times log\ t)$$

Compose. The `Compose` operator first applies the G_{con} to generate a new service graph that requires a complexity of $O(V+E)$. It then uses five major steps for retrieving service instances. We analyze the complexity of each step and derive the overall complexity in the end.

(1) Step 1 divides a complex project predicate into k subpredicates. It requires one pass of the k subpredicates and has a complexity of $O(k)$.

(2) Step 2 performs θ-join using the k subpredicates respectively. The complexity is $O(k \times t \times log\ t)$.

(3) Step 3 projects the resultant quality value relation onto the *sid* attribute. We assume that the average selection factor in Step 2 is s'_F. Therefore, the complexity of step 3 is $O(k \times t^2 \times s'_F)$.

(4) Step 4 uses the set operators (i.e., $\cap$ and $\cup$) to get the final sids. To eliminate duplicates, we assume that the set operators usually need to sort the relations on the sids and then compare the sids from both relations. Therefore, the complexity is $O((k-1) \times t^2 \times s'_F \times log\ (t^2 \times s'_F))$.

(5) Step 5 performs $(opid(SR_1)\infty SID)\infty(opid(SR_2)$, $(\lambda_1(SR_1)\infty SID)\infty\lambda_1(SR_2)$, $\ldots$, and $(\lambda_m(SR_1)\infty SID)\infty\lambda_k(SR_2)$ to retrieve the operation id and quality values for the retrieved service instances. Since SID has been sorted in Step 4, the complexity is $O(m \times t \times log\ t)$.

We can derive the overall complexity of the `Compose` operator as:

$$O((V+E) + k + k \times t \times log\ t + k \times t^2 \times s'_F + (k-1) \times t^2 \times s'_F \\ \times log\ (t^2 \times s'_F) + m \times t \times log\ t) \\ = O((V+E) + (k \times t + m) \times t \times s'_F \times log\ t)$$

Crossover. The `Crossover` first applies the G_{con} to generate a new service graph that requires a complexity of $O(V+E)$. It then applies to the operation id relation and the m quality value relation. Therefore, they have a complexity of $O((V+E) + (m+1) \times t^2)$.

5.4 Generating SEPs

The retrieved service relation conforms to the service graphs that define the service operations and their dependency constraints for the service instances in the retrieved service relation. We investigate in this section how to generate SEPs from the retrieved service instances based on the service graphs. The service graphs could be generated during query processing (e.g., if `F-map` and `Compose` are involved) or originally defined by the service schema. We propose an algorithm that generates a *service execution path* from a service graph. The service execution path arranges all operations in the service graph into a sequence with respect to all the dependency constraints. A service execution path is nonexecutable because operations in the path are at the schema level; that is, these operations are not from any particular service instances. SEPs can be generated by instantiating the service execution path with the operations from the service instances. A SEP is executable in that it is formed by the operations from service instances that correspond to the actual service providers.

Remark 5.1. A Service Execution Plan (SEP) is different from a Service Query Plan (SQP) from two major aspects:

(1) A SEP consists of a set of service operations from the retrieved service instances. A SQP, on the other hand, is composed of algebraic operators and service relations. It is used for retrieving service instances.
(2) A SEP specifies the order to execute the service operations. A SQP, on the other hand, specifies the order of the algebraic operators and service relations for retrieving service instances.

Figure 9 illustrates the algorithm of generating a service execution path from a service graph. The produced service execution path is an ordered list of service operations. This algorithm has two major features: *edge coloring* and *depth-first traversal*. First, the orders among operations need to conform to the dependency constraints. Therefore, an operation can be executed only after all of its depending operations have been executed. To fulfill this requirement, we color the edges of the graph as green and red. An edge is initially colored as red (line 3–4). Once a service operation is visited, all of its outgoing edges are colored as green (line 9–10 and line 24–25). Only the operations with all green incoming edges will be visited (line 17–20). Edge coloring guarantees that service operations can be executed in the order that conforms to dependencies. Second, we use the recursive call to achieve a graph depth-first traversal (line 26–27). The graph depth-first traversal will put service operations that come from the same service paths (e.g., `paymentHistory` and `financingQuote`) close to one another in the service execution path. This enables users to continuously perform a set of related operations.

THEOREM 5.1. *The execution path generation algorithm has the complexity of* $\mathcal{O}|V + E|$.

PROOF. This algorithm is a graph depth-first traversal algorithm. Therefore, it has the complexity of $\mathcal{O}|V + E|$. □

6. SERVICE QUERY OPTIMIZATION

The retrieved service relation usually contains more than one service instances. Therefore, multiple SEPs can be generated from the service relation. All these SEPs satisfy the functional and quality requirement specified by the query. However, users may have special preference for some QoWS parameters over

```
ALGORITHM Service Execution Path Generation
Input: A Service Graph G
Output: A Service Execution Path $P = (\epsilon, op_1, op_2, ..., op_n)$.
1.  for each operation $op \in G$ do
2.      $op.order = -1$;
3.  for each edge $e \in G$ do
4.      $e.color = red$;
5.  $current\_order = 0$;
6.  $P[current\_order] = \epsilon$;
7.  $\epsilon.order = current\_order + 1$;
8.  $current\_order = current\_order + 1$;
9.  for each $\epsilon$'s outgoing edge $e$ do
10.     $e.color = green$;
11. for each $\epsilon$'s next operation $op$ do
12.     $depth\_first\_traversal(G, op, P, current\_order)$;
13. for each operation $op'$ do
14.     if $(op'.order == -1)$ then
15.         $depth\_first\_traversal(G, op', P, current\_order)$;

FUNCTION depth_first_traversal
Input: $G, op, P, order$
Output: $void$
16. boolean $executable = true$;
17. for each $op'$s incoming edge $e$ do
18.     if $(e.color == red)$ then
19.         $executable = false$; break;
20. if $(executable == false)$ then return;
21. $P[current\_order] = op$;
22. $op.order = current\_order + 1$;
23. $current\_order = current\_order + 1$;
24. for each $op$'s outgoing edge $e$ do
25.     $e.color = green$;
26. for each $op$'s next operation $op'$ do
27.     $depth\_first\_traversal(G, op', P, current\_order)$;
```

Fig. 9. Execution path generation algorithm.

others. For example, Mary wants to use the SEP with the highest reputation; that is, she prefers to buy the car package from the most reputable providers. The preference is on the *entire* SEP that usually contains multiple operations. It is different from the quality requirement in the service query that is on some *individual* operations (e.g., get the history report for less than 20 dollars). The QoWS aware query optimization is a "user centered" optimization. It is to find the SEP with the best quality based on user preference over the entire SEP.

Table IV. QoWS for a Service Execution Plan

QoWS Parameter	Aggregation Function
Latency	$\sum_{i=1}^{n} latency(op_i)$
Reliability	$\prod_{i=1}^{n} rel(op_i)$
Availability	$\prod_{i=1}^{n} av(op_i)$
Fee	$\sum_{i=1}^{n} fee(op_i)$
Reputation	$\frac{1}{n}\sum_{i=1}^{n} rep(op_i)$

In this regard, the service query optimization performs a *global selection* as opposed to *local search* that is performed by the query processor. In this section, we first present a set of aggregation functions to compute the QoWS for SEPs. They combine the QoWS parameters from multiple service operations. An score function is then presented to evaluate the entire SEPs. Finally, we present two optimization algorithms to find the best SEPs.

6.1 QoWS for SEPs

We need now to compute the *QoWS* parameters for the entire service execution plan that may contain multiple service operations. Based on the meaning of *QoWS*, we define a set of aggregation functions to compute *QoWS* of service execution plans, as shown in Table IV. The quality of a SEP can thus be characterized as a vector of QoWS,

$$Quality(SEP_i) = (lat(SEP_i), rel(SEP_i), av(SEP_i), fee(SEP_i), rep(SEP_i)).$$

lat (latency) and *fee* (usage fee) take scalar values ($\Re^+$). *av* (availability), and *rel* (reliability) represent probability values (a real value between 0 and 1). Finally, *rep* (reputation) ranges over the interval [0,5].

6.2 Score Function

We define a score function to compute a scalar value out of the QoWS vector of the SEPs. This can facilitate the comparison of the quality of the SEPs. Since users may have preferences over how their queries are answered, they may specify the relative importance of *QoWS* parameters. We assign *weights*, ranging from 0 to 1, to each *QoWS* parameter to reflect the level of importance. Default values are otherwise used.

We use the following score function F to evaluate the quality of the service execution plans. By using the score function, the QoWS optimization is to find the execution plan with the maximum score.

$$F = \left(\sum_{Q_i \in neg} W_i \frac{Q_i^{max} - Q_i}{Q_i^{max} - Q_i^{min}} + \sum_{Q_i \in pos} W_i \frac{Q_i - Q_i^{min}}{Q_i^{max} - Q_i^{min}} \right),$$

where *neg* and *pos* are the sets of negative and positive *QoWS* respectively. In negative (resp. positive) parameters, the higher (resp. lower) the value, the worse is the quality. W_i are weights assigned by users to each parameter. Q_i is the value of the ith *QoWS* of the service execution plan obtained through the aggregate functions from Table IV. Q_i^{max} is the maximum value for the

```
ALGORITHM DP Plan Optimization
1.  for $i = 2$ to $k$ do
2.      for each $PR \subseteq \{SR_1, ..., SR_k\}$ s.t. $||PR|| = i$ do
3.          bestSQP = a system generated SQP with $+\infty$ cost;
4.          for each pair $PR_j, SR_j$ s.t. $PR = \{SR_j\} \cup PR_j$ and $\{SR_j\} \cap PR_j = \phi$ do
5.              tempSQP = composePlan(bestSQP($PR_j$), $SR_j$);
6.              if (tempSQP.cost < bestSQP.cost) then
7.                  bestSQP = tempSQP;
8.          bestSQP(PR) = bestSQP;

9.  $\{SEP_1, ..., SEP_m\}$ = execute(bestSQP($\{SR_1, ..., SR_k\}$));
10. bestSEP = a system generated SEP with $-\infty$ score;
11. for $i = 1$ to $m$ do
12.     $SEP_i$.score = F($SEP_i$);
13.     if ($SEP_i$.score < bestSEP.score) then
14.         bestSEP = $SEP_i$;
15. return bestSEP;
```

Fig. 10. DP-based plan optimization algorithm.

ith *QoWS* parameter for all potential service execution plans and Q_i^{min} is the minimum. These two values can be computed by considering the operations from service instances with the highest and lowest values for the ith *QoWS*.

6.3 Optimization Algorithms

The algebraic optimization depends on the "predicate pushdown" rules to perform `Compose` and `Crossover` as late as possible. Since the service algebraic rules also include associative and communicative rules for `Compose` and `Crossover`, many equivalent expressions can still be produced after the algebraic optimization. As the number of `Compose` or `Crossover` in a service query increases, the number of different composition orders may grow rapidly. The objective of service query optimization is to select the most efficient composition order to form a fast SQP. It then selects the SEP with the best user desired quality from the multiple candidates resulted from the SQP.

Join ordering optimization has been intensively investigated in database research [Chaudhuri 1998; Selinger et al. 1979]. One of the most adopted approaches is the System-R bottom-up dynamic programming query optimization [Selinger et al. 1979]. A straightforward solution for our service query optimization is to extend the DP optimization approach. Figure 10 shows the extended DP based plan optimization algorithm. It consists of two major phases. The first phase depends on dynamic programming to select the most efficient query plan (line 1–8). The query plan is then executed in the second phase (line 9), which results in a set of SEPs. The second phase then proceeds to select the SEP with the best quality (i.e., the maximum score) (line 10–15). It is worth noting that multiple query plans may coexist even if they share the same join order in the first phase of the algorithm. This is because some query plan may

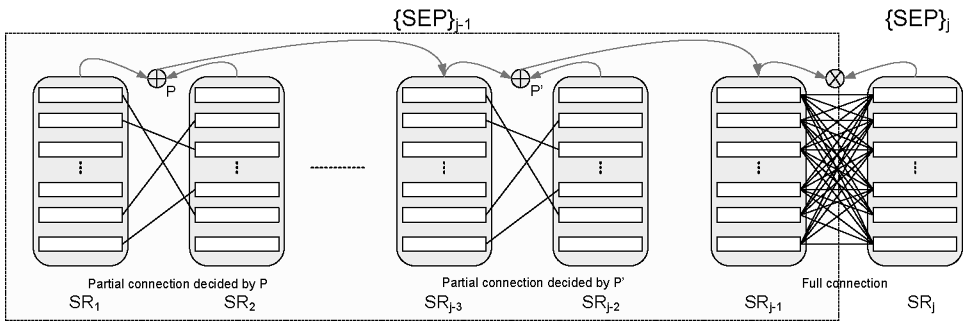

{SEP}j-1
{SEP}j
P
P'
Partial connection decided by P
Partial connection decided by P'
Full connection
SR1
SR2
SRj-3
SRj-2
SRj-1
SRj

Fig. 11. Properties of `Crossover`.

place the service tuples in *interesting orders* that can be beneficial to subsequent algebraic operators [Selinger et al. 1979].

DP optimization performs all `Crossovers` as late in the join sequence as possible [Selinger et al. 1979]. This implies that in composing service relations $SR_1, SR_2, \ldots, SR_n$ only those orderings $SR_{i1}, SR_{i2}, \ldots, SR_{in}$ are examined in which for all j, where $j = 2, \ldots, n$, either

(1) SR_{ij} has at least one join predicate with some relation SR_{ik}, where $k < j$, or

(2) $\forall k > j, SR_{ik}$ has no compose quality predicate with $SR_{i1}, \ldots, SR_{i(j-1)}$.

All query plans that satisfy the above composition ordering requirement can generate relatively smaller number of intermediate results than other query plans. Therefore, they can be performed more efficiently to generate the SEPs. Since service query optimization aims to select the best SEP, it will be necessary if a query plan can return a subset of the SEPs where the optimal solution is in it. This can effectively reduce the enumeration space. Therefore, we can further improve the DP optimization. The challenge is how to ensure that the optimal solution is included in the subset of SEPs. This is addressed by a special treatment on the `Crossover` operator.

A Divide-And-Conquer Query Optimization Strategy. `Crossover` is an expensive algebraic operator. The number of service tuples returned is exponential to the number of service relations that are involved. In this section, we propose an approach to deal with the `Crossover` operator. This approach enables the optimization process to consider only a small subset of the service execution plans. It guarantees that the subset encompasses the (semi) optimal solution. This greatly improves the system performance while preserving the quality of the selected SEP.

We first investigate some important properties of the `Crossover` operator. Figure 11 helps illustrate these properties. Service relation SR_{j-1} is the result of a sequence of compositions. The service relations (e.g., SR_1 and SR_2) under the `Compose` operator are "partially connected" based on the compose quality predicate. By "partially connected," we mean that only a subset of service tuples from SR_1 and SR_2 are selected and combined to form the composed relation (see the connections between SR_1 and SR_2 in Figure 11). The number of service generated is $card(SR_1) \times card(SR_2) \times SF_J(SR_1, SR_2, p)$, where $SF_J(SR_1, SR_2, p)$ is the *join selectivity factor*. The selectivity factor depends on service relations and the compose quality predicate. It takes a real value between 0 and 1. After the sequence of compositions are performed, service relation SR_{j-1} is combined with SR_j to generate the final result using the `Crossover` operator. Since there is no compose predicate, SR_{j-1} is "fully connected" with SR_j to perform the `Crossover` (see the connections between SR_{j-1} and SR_j in Figure 11). This results in $card(SR_{j-1}) \times card(SR_j)$ service tuples. The searching space would be greatly increased by the `Crossover` operator.

We adopt a *divide-and-conquer* strategy to deal with `Crossover`. The strategy generates a set of (sub) optimal *partial SEPs* through local search. It then combines these partial SEPs to form the final SEP. Before going into the details, we first introduce some approximation functions which will be used in this strategy. Two aggregation functions (i.e., the functions for reliability and availability) presented in Table IV do not combine QoWS parameters from multiple service operations in a linear manner. We propose two linear functions to approximate the original functions for aggregation purpose. Specifically,

$$Reliability = \sum_{i=1}^{n} log(rel(op_i)), \quad Availability = \sum_{i=1}^{n} log(av(op_i)).$$

These enable us to express the score of the final SEP as a linear combination of the scores from the partial SEPs. In the example shown in Figure 11, the score of the final SEP is the sum of those from its partial SEPs, that is, score(SEP_{ik}) = score(SEP_i^{j-1}) + score(SEP_k^j).

SR_{j-1} is fully connected with SR_j through the `Crossover` operator. There must be a connection between the best partial plan in $\{SEP\}_{j-1}$ and the best partial plan in $\{SEP\}_j$. The aggregation of these two partial plans forms the best service execution plan. This is because best($\{SEP\}$).score = best($\{SEP\}_{j-1}$).score + best($\{SEP\}_j$).score. Therefore, the optimization algorithm can perform the sequence of joins and query a single (or a set of) service relation(s) separately. This enables to achieve the (semi) final optimal solution through a set of local search without the need to really perform the `Crossover`. We call this optimization strategy the *Divide-And-Conquer-DP (DAC-DP) optimization*. The best partial SEPs can finally be combined to form the (semi) optimal solution. For the case shown in Figure 11, only ($card(SR_{j-1}) + card(SR_j)$) service tuples are returned instead of ($card(SR_{j-1}) \times card(SR_j)$). Figure 12 shows the DAC-DP service query optimization algorithm. The algorithm first perform the sequence of compositions on the first t service relations and generate m partial SEPs (line 9). It then selects the best partial SEP (line 10–14). The algorithm then proceeds to query each of the remaining service relations that need to be combined using the `Crossover` operator. It selects the best partial SEP from the query result of each service relation (line 15–21). All the best partial SEPs are then combined to form the final SEP (line 22).

```
ALGORITHM DAC-DP Plan Optimization
1.  for $i = 2$ to $t$ do
2.      for each $PR \subseteq \{SR_1, ..., SR_t\}$ s.t. $||PR|| = i$ do
3.          bestSQP = a system generated SQP with $+\infty$ cost;
4.          for each pair $PR_j, SR_j$ s.t. $PR = \{SR_j\} \cup PR_j$ and $\{SR_j\} \cap PR_j = \phi$ do
5.              tempSQP = composePlan(bestSQP($PR_j$), $SR_j$);
6.              if (tempSQP.cost < bestSQP.cost) then
7.                  bestSQP = tempSQP;
8.          bestSQP(PR) = bestSQP;

9.  $\{SEP_1, ..., SEP_m\}$ = execute(bestSQP($\{SR_1, ..., SR_t\}$));
10. bestSEP = a system generated SEP with $-\infty$ score;
11. for $i = 1$ to $m$ do
12.     $SEP_i$.score = F($SEP_i$);
13.     if ($SEP_i$.score < bestSEP.score) then
14.         bestSEP = $SEP_i$;
15. for $i = (t + 1)$ to $k$ do
16.     $\{SEP^i_1, ..., SEP^i_{m^i}\}$ = execute(bestSQP($SR_i$));
17.     bestSEP$_i$ = a system generated SEP with $-\infty$ score;
18.     for $j = 1$ to $m^i$ do
19.         $SEP^i_j$.score = F($SEP^i_j$);
20.         if ($SEP^i_j$.score < bestSEP$_i$.score) then
21.             bestSEP$_i$ = $SEP^i_j$;
22.     bestSEP = bestSEP $\cup$ bestSEP$_i$;
23. return bestSEP;
```

Fig. 12. DAC-DP plan optimization algorithm.

Table V. Symbols and Parameters

Variables	Definition
N_{SR}	Total number of service relations
N^J_{SR}	Number of service relations under join
N^C_{SR}	Number of service relations under Cartesian product
N_{IO}	Total number of interesting orders
N^i_{SI}	Number of service instances in the ith service relation
$SF_J(SR_i, SR_{i+1})$	Join selectivity factor between SR_i, SR_{i+1}

7. ANALYTICAL MODEL

In this section, we present the analytical model for the above optimization algorithms. We analyze the complexity of the DP-based optimization algorithm and the DAC-DP algorithm. Table V defines the parameters and the symbols used in this section.

7.1 DP-based Query Optimization

We start by studying the complexity of the DP-based optimization algorithm. There are two major phases in this algorithm. The first phase is to select the most efficient query plan whereas the second phase is to generate the best service execution plan. In the first phase, the DP optimization algorithm uses two heuristics to reduce the enumeration space. First, it eliminates the permutations that involve Cartesian products. Second, the commutatively equivalent strategies with the highest cost are also eliminated. Therefore, these heuristics help reduce the size of the enumeration space from $N_{SR}!$ to $2^{N_{SR}}$. The algorithm also considers interesting orders; hence, the complexity of the first phase is:

$$O\left(2^{N_{SR}} \times N_{IO}\right). \tag{1}$$

The second phase enumerates the space of SEPs. The size of the SEP space is determined by the number of service relations, number of service instances per service relation, and the join selectivity factor. The join selectivity factor for Cartesian products takes the value of 1 whereas the selectivity factor for the normal join operators takes a value between 0 and 1. The complexity of the second phase is:

$$O\left(\prod_{i=1}^{N_{SR-1}} N_{SI}^{i} \times \prod_{i=1}^{N_{SR-1}} SF_J(SR_i, SR_{i+1})\right). \tag{2}$$

Therefore, the complexity of the entire DP-based optimization algorithm is:

$$O\left(2^{N_{SR}} \times N_{IO} + \prod_{i=1}^{N_{SR-1}} N_{SI}^{i} \times \prod_{i=1}^{N_{SR-1}} SF_J(SR_i, SR_{i+1})\right). \tag{3}$$

7.2 DAC-DP Query Optimization

The DAC-DP optimization algorithm consists of two similar phases as the DP-based optimization algorithm. It relies on the divide-and-conquer strategy to reduce the enumeration space in both phases. We assume that there are N_{SR}^{J} service relations to be combined using join. Therefore, the complexity of the first phase is:

$$O\left(2^{N_{SR}^{J}} \times N_{IO}\right). \tag{4}$$

The second phase selects a set of best partial SEPs and then combines them to form the final optimal SEP. Based on our analysis in Section 6.3, we can derive the complexity of the second phase is:

$$O\left(\prod_{i=1}^{N_{SR}^{J}} N_{SI}^{i} \times \prod_{i=1}^{N_{SR}^{J}-1} SF_J(SR_i, SR_{i+1}) + \sum_{i=N_{SR}^{J}+1}^{N_{SR}} N_{SI}^{i}\right). \tag{5}$$

Therefore, the complexity of the entire DAC-DP optimization algorithm is:

$$O\left(2^{N_{SR}^{J}} \times N_{IO} + \prod_{i=1}^{N_{SR}^{J}} N_{SI}^{i} \times \prod_{i=1}^{N_{SR}^{J}-1} SF_J(SR_i, SR_{i+1}) + \sum_{i=N_{SR}^{J}+1}^{N_{SR}} N_{SI}^{i}\right). \tag{6}$$

Table VI. Parameter Settings

Parameters	CP		CI		FI	
	carQuote	historyReport	paymentHistory	financingQuote	drivingHistory	insuranceQuote
latency	0–300(s)		0–300(s)		0–300(s)	
reliability	0.5–1.0		0.5–1.0		0.5–1.0	
availability	0.7–1.0		0.7–1.0		0.7–1.0	
fee	0–30($)		0–30($)		0–30($)	
reputation	0–5		0–5		0–5	

8. EXPERIMENTAL STUDY

We conducted a set of experiments to assess the performance of the proposed approach. We use the car brokerage scenario as our testing environment to setup the experiment parameters. The purpose is to demonstrate how our approach can help Mary select the best deal. The Web services are developed on Systinet WASP Server, which is a complete platform for development, deployment, and management of Web service based applications [Systinet 2004]. We run our experiments on a cluster of *Sun Enterprise Ultra 10* workstations under *Solaris* operating system.

We create a service schema containing three services, which is similar to the one shown in Figure 3. For simplicity, we omit the unnecessary service operations. Each service contains two operations: CP (careQuote, historyReport), CI (drivingHistory, insuranceQuote), and FI (paymentHistory, financingQuote). We create three service relations, which conform to the service schema. The number of service instances in each service relation varies from 10 to 60. We use five QoWS parameters to evaluate service operations: latency, reliability, availability, fee, and reputation. The values of these parameters are generated within a range based on uniform distribution. The user's role is to give the weights for these parameters. Table VI summarizes the potential values for the QoWS parameters for each service operation.

We consider a service query that helps Mary get an entire car package, including the price quote and history report of a used car, insurance quote, and financing quote. The service query can be expressed as a service algebraic expression as follows:

$$Q : \chi_{\{op_1,op_2,op_3,op_4\}}\left[\delta_{\lambda_4(op_2)\leq 20}(CP) \oplus_{\lambda_3(op_1)<\lambda_3(op_4)} \delta_{\lambda_4(op_3)\leq 20}(CI) \otimes FI\right],$$

where op_1, op_2, op_3, op_4 represent service operations carQuote, historyReport, insuranceQuote, and fincancingQuote respectively.

Performance measure. We measure the performance of the optimization approaches using *computational time* and *score function value*. We use the formulae defined in Section 6.2 to compute the score of SEPs. We compare the scores of the best SEPs generated by the two query optimization algorithm. Since both query optimization algorithms both have two phases, we study and compare the computational time for each phase and the entire algorithm. We also investigate and compare the number of SEPs generated by each algorithm.

Figure 13 shows the optimization time resulting from the experiments. DP-DAC is much more efficient than DP due to the divide-and-conquer strategy. The chart on the left compares the optimization time for the first phase, where

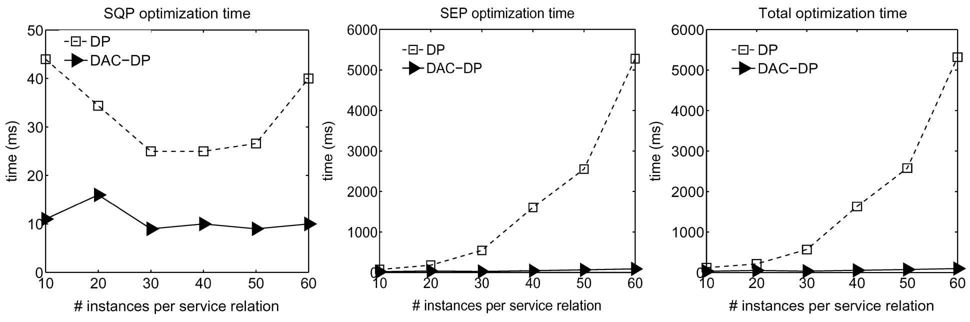


Fig. 13. Performance comparison: DP Vs. DAC-DP.

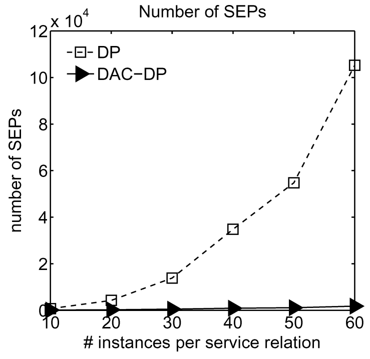


Fig. 14. Number of SEPs.

the most efficient service query plans (SQPs) are selected. The chart in the middle compares the optimization time for the second phase, where the best service execution plans (SEPs) are generated. DP-DAC outperforms DP in both cases. The right-hand-side chart compares the total optimization time.

Figure 14 shows the number of SEPs generated by performing the most efficient SQP selected in the first phase. DP-DAC generates much less SEPs than DP. This also justifies why DP-DAC is more efficient in its second phase than DP. Figure 15 shows the scores of the best SEPs generated by DP and DAC-DP. In two cases (number of instances 10 and 60), DP and DAC-DP output the best SEPs with the same score. In the other four cases, the scores of DAC-DP are slightly lower (less than three percent) than those of DP. The difference comes from the two approximation functions used by DAC-DP to aggregate QoWS parameters. We have conducted a set of additional experiments to further evaluate the scalability of the DAC-DP algorithm. We increase the number of instance per service relation with two orders of magnitude and test of the performance of DAC-DP with the number of service instances varying from 1000 to 5000. Experiment results (presented in Figure 16) show that DAC-DP can still perform very efficiently (using less than 1 second to query three service relations, each of which has 5000 service instances) on large number of service instances.

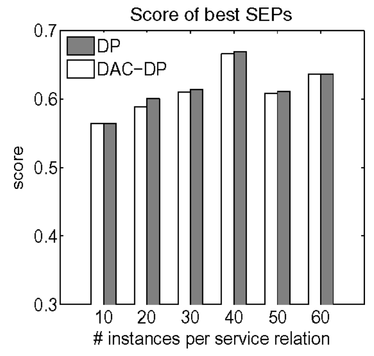


Fig. 15. Score of SEPs.

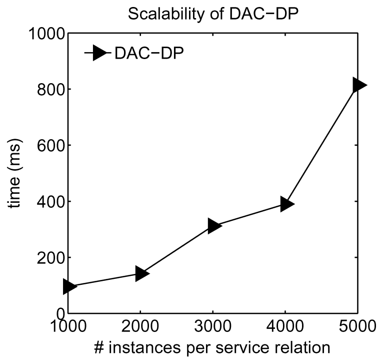


Fig. 16. Scalability.

Compare with the analytical model. We further compare the experiment results with the results from the analytical model presented in Section 7. Although the analytical model only predicts the upper bound of the time complexity, we can still perform an approximate comparison. This is to justify that the experiment results follow the same trends as predicted by the analytical model. We focus on the improvement (in terms of optimization time) achieved by DAC-DP over DP. We define a variable DP/DAC-DP (i.e., the time used by DP divided by the time used by DAC-DP) to demonstrate the improvement. Figure 17 illustrates the detailed comparison results.

The chart on the left-hand-side shows the improvement for the first optimization phase. The complexity of this phase is mainly decided by the number of service relations under join. In our experiment settings, the DP approach needs to join three service relations whereas the DAC-DP only needs to join two service relations (the divide-and-conquer strategy removes the service relation *FI* from the join list by considering it separately). Therefore, in the analytical model, DP/DAC-DP should take an approximate value of $(2 \times N_{IO}^{DP})/N_{IO}^{DAC\text{–}DP}$, where N_{IO}^{DP} and $N_{IO}^{DAC\text{–}DP}$ represent the number of interesting orders considered by DP and DAC-DP in the first optimization phase. Since the number of interested orders are unknown, they are not reflected by the analytical result in the chart (that is why the analytical result curve is a horizontal straight

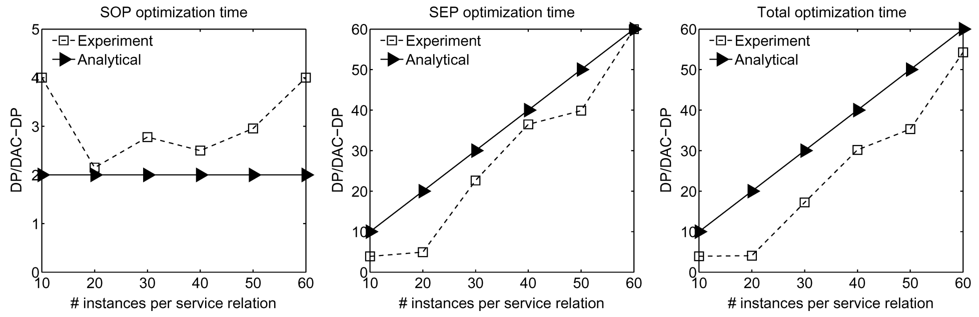


Fig. 17. Experiment result vs. analytical result.

line). The actual experiment result curve stays above the analytical one. This is because DP joins one more service relations than DAC-DP, it naturally needs to consider more interesting orders; that is, $N_{IO}^{DP}/N_{IO}^{DAC\text{-}DP} > 1$. This makes $(2 \times N_{IO}^{DP})/N_{IO}^{DAC\text{-}DP} > 2$.

The middle chart in Figure 17 shows the improvement in the second optimization phase. To calculate DP/DAC-DP for the analytical model, we use Equations (2) and (5) defined in Section 7. If we neglect the last item in Equation (5) (i.e., $\sum_{i=N_{SR}^{J}+1}^{N_{SR}} N_{SI}^{i}$), we can derive that DP/DAC-DP takes an approximate value of N_{SR}^{3}, that is, the number of service instances in the third service relation *FI*. Therefore, the analytical curve is an 45-degree straight line. The experiment curve has a very similar trends. It stays below the analytical curve because we neglect the last item in the denominator for the analytical curve. Since the total optimization time is dominated by the second phase, the total optimization time has very similar trends to the second phase. The right-hand-side chart shows the result of the total optimization time.

9. RELATED WORK

The proliferation of Web services is fostering a very active research area. We give an overview of some work in this area which are most closely related to our work. Since the field of Web service research is still in its infancy, there is little foundational work to date.

The concept of quality of service (QoS) has been widely used in middleware and networking communities [Aurrecoechea et al. 1998; Marchetti et al. 2004; Gillmann et al. 2002]. In these communities, QoS usually refers to a broad collection of networking technologies which aim to provide guarantees on the ability of a network to deliver predictable results. Elements of network performance within the scope of QoS often include availability (uptime), bandwidth (throughput), latency (delay), and error rate. Research efforts in middleware and networking communities mainly focus on the performance of network and devices. Therefore, these works are centered on the network transport and system level [Zeng et al. 2004]. This is different from the Web service domain that treats QoS from a broader perspective and focuses on the application and process levels. In this regards, QoS in the Web service domain subsumes existing technologies in middleware and networking communities, which can be leveraged to improve the system performance (e.g., availability). More importantly, quality metrics for Web services put more focus on the user experiences (called QoE) and business return (QoBiz) [van Moorsel 2001]. Optimization for Web services based on these quality metrics is "user centered" as opposed to the "system performance centered" optimizations in middleware and networking communities. Moreover, since multiple Web services usually need to be combined to work as a complex process (called composite service), an important and distinct issue is to combine the QoS from different services to evaluate the overall QoS of the composed service.

Finite state automata have been adopted to model Web services (e-Services). A typical example is the work in Berardi et al. [2005]. A service model in this work consists of an external schema and an internal schema. The external schema is to specify the exported behavior of a service. The behavior is represented by a set of actions and the corresponding state transitions. The internal schema, on the other hand, specifies the information on which services execute each given action. The FSMs service model provides fundamental support for the service composition theory. Petri net has also been used to to model Web services, which is referred to as a service net in Hamadi and Benatallah [2003]. Based on the Petri-net service model, a service level algebra is proposed. The proposed algebra verifies the closure property. The algebra can be used to construct complex composite services by aggregating and reusing existing services. A fundamental difference between the automata and Petri-net service models and the model used in this paper is the purpose and usage of the formalisms embodied in these models. The service model in this paper is designed to provide formalisms for service query optimization. Specifically, the graph-based service schema and the service relation enable to query services based on their functionalities. They also enable the design of optimization algorithms to select services with the best user desired quality. Formalisms defined in the automata and Petri-net models are mainly used for automatically composing services and verifying the composed service is well formed and meet the composition requirement.

Srivastava et al. [2006] propose a Web Service Management System (WSMS) to enable optimized querying of Web services. A Web service $WS_i(\mathcal{X}_i^b, \mathcal{Y}_i^f)$ is modeled as a virtual table in the proposed WSMS. The values of attributes in $\mathcal{X}_i$ must be specified whereas the values of attributes in $\mathcal{Y}_i$ are retrieved. An algorithm is proposed to optimized access Web services. The optimization algorithm takes as input the classical Select-Project-Join queries over Web services. It arranges Web services in a query based on a cost model and returns a pipelined execution plan with minimum total running time of the query. In our service query optimization framework, we adopt a formal service model. The service model goes beyond the simple function call by effectively capturing the key features of Web services: functionality, behavior, and quality. The service calculus is proposed based on the service model. It enables users to declaratively query Web services based on these features. The optimization algorithm in [Srivastava et al. 2006] focuses on the total running time. In contrast, in our optimization algorithm, process, both the response time and the quality of Web service are optimized.

Zeng et al. [2004] propose a composite service optimization approach based on several quality of service parameters. Composite services are represented as a state-chart. The optimization problem is tackled by finding the best Web services to execute a composite service in the form of a linear programming problem. Our work focuses on Web service querying instead of generating composite services. We propose a query algebra that to access Web services. Our optimization algorithm aims to efficiently find the service execution plan with the best quality.

10. CONCLUSION

We present in this article a query algebra that allows users to efficiently and optimally access Web services. The query algebra consists of a set of algebraic operators. Service queries can thus be specified as algebraic expressions using these operators. We present the physical implementation of each algebraic operator. This enables us to generate SEPs that can directly be used to access services by the users. A divide-and-conquer query optimization algorithm is proposed to efficiently select the SEP with the best user-desired quality. Experimental results demonstrate significant performance improvement over the traditional DP-based optimization approach.

REFERENCES

Alonso, G., Casati, F., Kuno, H., and Machiraju, V. 2003. *Web Services: Concepts, Architecture, and Applications*. Springer Verlag.

Aurrecoechea, C., Campbell, A., and Hauw, L. 1998. A survey of QoS architectures. *ACM/Springer Verlag Multimed. Syst. J. 6*, 3, 138–151.

Berardi, D., Calvanese, D., Giacomo, G. D., Hull, R., and Mecella, M. 2005. Automatic composition of transition-based semantic Web services with messaging. In *Proceedings of the International Conference on Very Large Databases*.

Bhatti, R., Bertino, E., and Ghafoor, A. 2005. A trust-based context-aware access control model for Web-services. *Distrib. Para. Data. 18*, 1, 83–105.

Casati, F., Shan, E., Dayal, U., and Shan, M. C. 2003. Business-oriented management of Web services. *Commu. ACM 46*, 10, 55–60.

Casati, F. and Shan, M. C. 2001. Definition, execution, analysis, and optimization of composite e-services. *IEEE Data Eng. Bull. 24*, 1, 29–34.

Chaudhuri, S. 1998. An overview of query optimization in relational systems. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. 34–43.

Dalvi, N., Sanghai, S., Roy, P., and Sudarshan, S. 2001. Pipelining in multi-query optimization. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*.

Dong, X., Halevy, A. Y., Madhavan, J., Nemes, E., and Zhang, J. 2004. Simlarity search for Web services. In *Proceedings of the International Conference on Very Large Databases*.

Du, W., Krishnamurthy, R., and Shan, M.-C. 1992. Query optimization in a heterogeneous DBMS. In *Proceedings of the International Conference on Very Large Databases*.

Fernandez, M. and Suciu, D. 1998. Optimizing regular path expressions using graph schemas. In *Proceedings of the IEEE International Conference on Data Engineering*. 14–23.

Florescu, D., Levy, A., Manolescu, I., and Suciu, D. 1999. Query optimization in the presence of limited access patterns. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*.

Gillmann, M., Weikum, G., and Wonner, W. 2002. Workflow management with service quality guarantees. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 228–239.

HAAS, L., KOSSMANN, D., WIMMERS, E., AND YANG, J. 1997. Optimizing queries across diverse data sources. In *Proceedings of the International Conference on Very Large Databases*.

HAMADI, R. AND BENATALLAH, B. 2003. A petri net-based model for web service composition. In *Proceedings of the 14th Australasian Database Conference on Database Technologies*. 191–200.

MARCHETTI, C., PERNICI, B., AND PLEBANI, P. 2004. A quality model for multichannel adaptive information. In *Proceedings of the International Conference on World Wide Web*. New York, NY.

MECELLA, M., OUZZANI, M., PACI, F., AND BERTINO, E. 2006. Access control enforcement for conversation-based Web services. In *Proceedings of the International Conference on World Wide Web*. 257–266.

OUZZANI, M. AND BOUGUETTAYA, B. 2004. Efficient access to Web services. *IEEE Internet Comput. 37*, 3.

PAPADIMITRIOU, C. H. AND YANNAKAKIS, M. 2001. Multiobjective query optimization. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*.

PAPAZOGLOU, M. P. 2003. Web services and business transactions. *WWW 6*, 1, 49–91.

PAPAZOGLOU, M. P., TRAVERSO, P., DUSTDAR, S., LEYMANN, F., AND KRÄMER, B. J. 2005. Service-oriented computing: A research roadmap. In *Service-Oriented-Computing*.

PAPAZOGLOU, M. P. AND VAN DEN HEUVEL, W. 2005. Web services management: A survey. *IEEE Internet Comput. 9*, 6, 58–64.

PONNEKANTI, S. AND FOX, A. 2002. SWORD: A developer toolkit for Web service composition. In *Proceedings of the International Conference on World Wide Web*.

PU, K., HRISTIDIS, V., AND KOUDAS, N. 2006. A syntactic rule based approach to Web service composition. In *Proceedings of the IEEE International Conference on Data Engineering*.

SELINGER, P., ASTRAHANAND, M., CHAMBERLIN, D., LORIE, R., AND PRICE, T. 1979. Access path selection in a relational database management system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 23–34.

SRIVASTAVA, U., WIDOM, J., MUNAGALA, K., AND MOTWANI, R. 2006. Query optimization over Web services. In *Proceedings of the International Conference on Very Large Databases*.

STRAUBE, D. D. AND ÖZSU, M. T. 1995. Query optimization and execution plan generation in object-oriented data management systems. *IEEE Trans. Knowl. Data Eng. 7*, 2, 210–227.

SYSTINET. 2004. Systinet server for Java. http://www.systinet.com/products/ssj/overview.

VAN MOORSEL, A. 2001. Metrics for the internet age: Quality of experience and quality of business. Tech. rep., HP Labs.

YERNENI, Y., LI, C., ULLMAN, J., AND GARCIA-MOLINA, H. 1999. Optimizing large join queries in mediation systems. In *Proceedings of the International Conference on Database Theory*.

YU, Q., LIU, X., BOUGUETTAYA, A., AND MEDJAHED, B. 2007. Deploying and managing Web services: Issues, solutions, and directions. *The VLDB J.*, To appear.

ZENG, L., BENATALLAH, B., NGU, A., DUMAS, M., KALAGNANAM, J., AND CHANG, H. 2004. Qos-aware middleware for web services composition. *IEEE Trans. Softw. Eng. 30*, 5, 311–327.

Received September 2006; revised May 2007; accepted August 2007