WS-Data Model: A Data Model for Web Services Composition and Optimization

Chien-Hsiang Lee and San-Yih Hwang
Department of Information Management
National Sun Yat-sen University
Kaohsiung 80424, Taiwan
syhwang@mis.nsysu.edu.tw

Abstract

Recent enthusiasm in service science attracts more attention on the design and analysis of service systems. A service system, from the engineering point of view, can be regarded as the interactions between various service elements, and service elements can be innovatively assembled to provide customizable services. To embody service elements in service-oriented architecture, Web services that conform to industrial standard have been proposed and widely adopted, and many researches are devoted into automatic Web service composition. However, most of these researches focus on control flow specification and enforcement. Data mismatch between Web services appears as an orthogonal problem that can be solved using XML-based query languages. We argue that greater optimization is possible if Web service composition can be considered into data manipulation operations. A proposed model, the WS-data model, focuses on data exchanges in composing Web services. Several operators with varying properties can compose Web services and manipulate their input and output data. Experiments conducted on the Amazon Cloud platform show that these operators' properties can help identify a more efficient way to realize a complex task, expressed according to the proposed WS-data model.

Keywords: Optimization of Services Composition, Web Services Interoperability, Operational Model, Data Model

Published by AIS Electronic Library (AISeL), 2014
www.manaraa.com

1. Introduction

With the emergence of service science, or called service science, management, engineering, and design (Spohrer and Kwan, 2009), there is a growing trend to look into services in a scientific and systematic way. Unlike most of previous studies that focus more on the customization aspect of services and service encounters, service science intends to systematically examine the entire service system that involves the interaction of various service elements from both front and back stages (Glushko and Tabas, 2009). Service encounters between service elements can then be regarded as information exchange. The abstract service elements can be innovatively assembled to provide customizable services, and their modular composition eases the task of analyzing the value that the service system delivers (Caswell et al., 2008). The standardization also helps reduce variability and foster fast mass customization.

In response to the demand of constructing agile service systems, various Web service standards have been proposed to describe a service (e.g., WSDL (Christensen et al., 2001)), to specify the format of message exchange (e.g., SOAP (Gudgin et al., 2007) and RESTful (Fielding, 2000)), and to compose services (e.g., BPEL (Jordan and Evdemon, 2007) and BPMN (OMG, 2013)). Innovative applications can be developed by integrating several Web services into composite versions. Previous research into Web service composition mostly has focused on the specification and enforcement of the invocation order of the constituent Web service operations and assumed that output messages serve, perhaps with minor modifications, as input messages for a succeeding Web service. Theoretical process meta-models, such as Petri net, finite state machine, π-calculus, and linear temporal logic, describe the invocation order in a process, which enables subsequent analysis and validation of the system (Berardi et al., 2005; Gerede et al., 2004; Lucchi and Mazzara, 2007; Narayanan and McIlraith, 2002; Ouyang et al., 2007; Sloan and Khoshgoftaar, 2009; Tan et al., 2009). However, the syntax, structure, or semantics of messages generated by one Web service operation may not exactly match those required by another operation. For example, an information enquiry service, such as Amazon Web Services (AWS) or Google Code, might return a list of items that satisfy a certain enquiry condition. Each retrieved item then may be processed by another Web service operation. Therefore, a proper extraction of data from the output of a previous Web service operation is critical to the successful invocation of the next operation. In some applications, the required input data may combine the outputs of multiple Web service operations. Therefore, the capability to manipulate data is essential for the successful composition of Web services.

XQuery, proposed by W3C, has become the de facto standard for manipulating XML documents, but it is limited to XML data and does not involve Web services. In contrast, we propose a model that considers both the workflow of Web services and the data mediation between Web services. We also take into account the possibility that a task might be realized by more than one set of Web services, called an abstract service in previous research (Thomas, 2007). This concept is a major principle of service-oriented architecture as a means to ease process design. An abstract service can have several implementations with equivalent functionalities. Previous works mostly regard an abstract service as a black box; we examine each implementation in an attempt to optimize the entire process, with the ultimate goal of identifying an efficient means to execute the process through a workflow of constituent Web service operations and methods for exchanging data.

Consider the stock replenishment process in Figure 1. A supplier needs to determine the quantity of stock to replenish and the corresponding prices of products, using the sales and inventory information of each store. The entire process requires three Web services: StoreInfo, StockService, and

https://aisel.aisnet.org/pajais/vol6/iss2/2
DOI: 10.17705/1pais.06202
www.manaraa.com

PriceService, which maintains sales and inventory information for each store, performs replenishment planning, and which determines the price of each item respectively. StoreInfo provides the operation getSalesInv() that returns sales and inventory information of a list of items. StockService includes the operation replenish() that takes as input the sales and inventory information of an item and returns the amount of supply to replenish. PriceService offers an operation pricing() that determines the unit price of an item based on the refilled quantity. The entire process consists of the four steps shown in Figure 1.

The stock replenishment process can be manually specified using BPEL and XQuery (or any other process and data transformation languages), as shown in the right-hand side of Figure 1. A straightforward way to realize this process, which we summarize in Figure 2(a), performs the four tasks sequentially. However, such a sequential execution delays each task execution until the previous task has been completed, which hinders overall performance. Figure 2(b) depicts another execution plan, in which each item returned by StoreInfo.getSalesInv() gets processed to determine replenishment in a pipelined

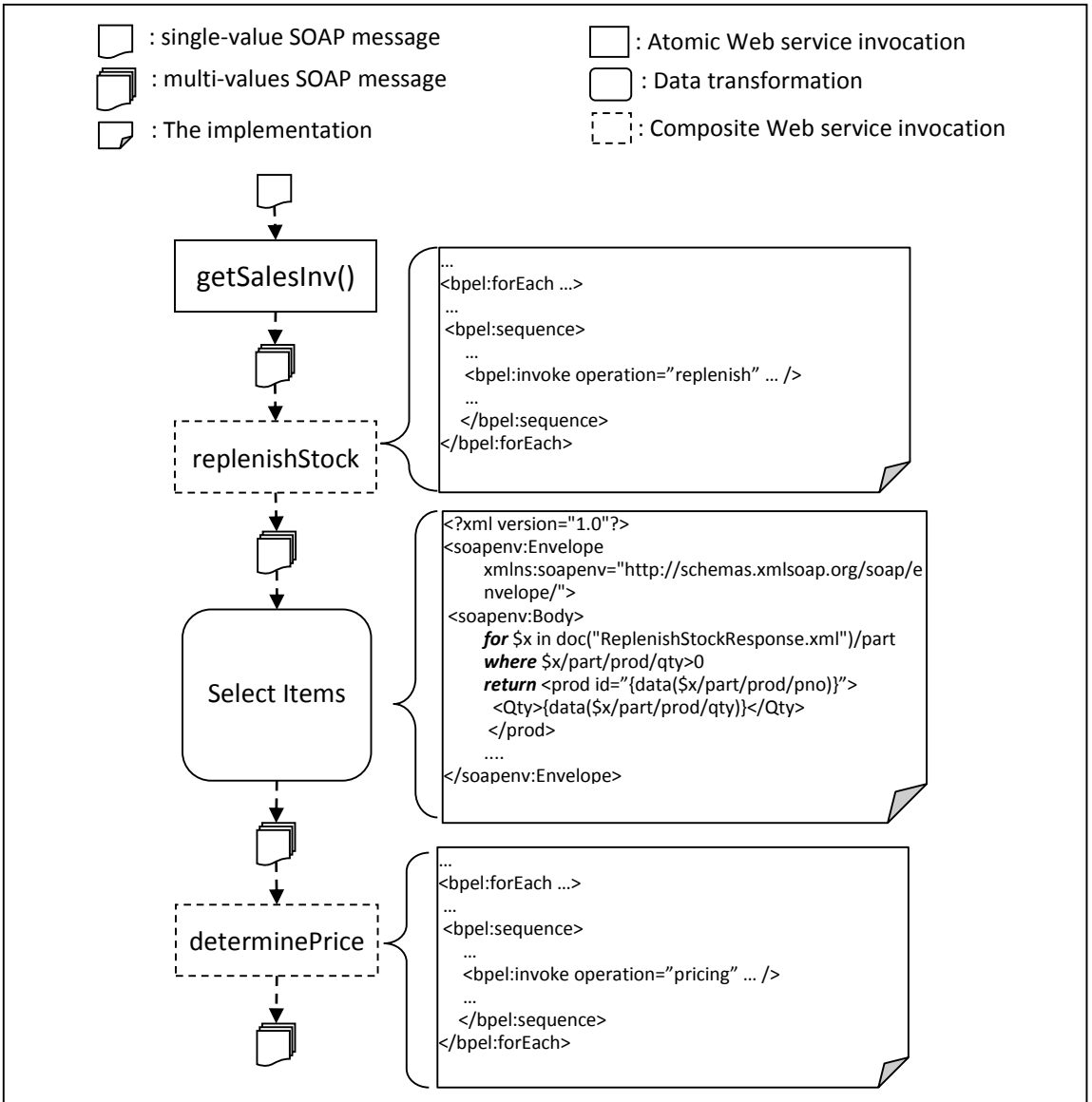


Figure 1 - Sample Stock Replenishment Process in BPEL and XQuery

Published by AIS Electronic Library (AISeL), 2014
www.manaraa.com

manner. The subsequent processing of each item also can be optimized by switching or combining some data manipulation operations, similar to relational database query optimization (Ullman, 1989). The execution plan in Figure 2(b) thus may yield better performance than that in Figure 2(a).

The model we propose describes a workflow of Web service operations and their data exchanges by including the Web service operations, XML documents required or produced by the Web service operations, and the operators for manipulating them. The WS-data model thus resembles a relational model but also has unique features that demand several novel operators. We outline transformation rules that pertain to the set of proposed operators and show that a Web

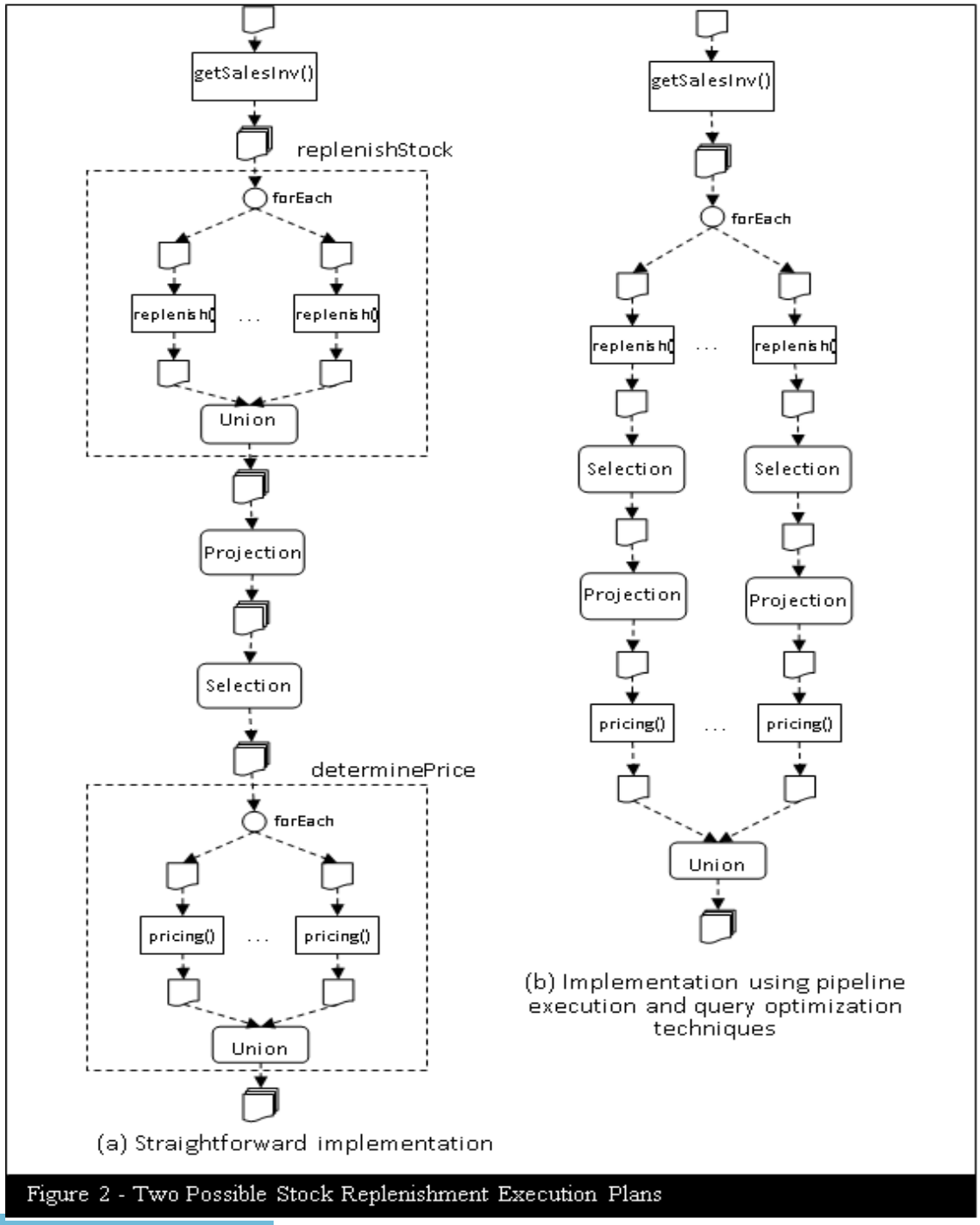


Figure 2 - Two Possible Stock Replenishment Execution Plans

https://aisel.aisnet.org/pajais/vol6/iss2/2
DOI: 10.17705/1pais.06202
www.manaraa.com

service composition expressed using the WS-data model can be optimized by applying these rules, which produce a more efficient execution plan.

In Section 2, we review previous Web service composition and XML algebra research. Section 3 presents the WS-data model and defines operators for manipulating the elements in the model. After describing our proposed approach for optimizing a WS-data expression, in Section 4, we evaluate this approach using an emulation performed on Amazon Cloud. Section 5 contains the experimental results. In Section 6, we summarize our findings and offer some research directions.

2. Literature review

Web service composition is a key research topic for services computing. The emergence of the process language BPEL has allowed for the manual composition of Web services, yet automatic Web service composition remains imperative to deal with the continual evolution of Web services. Most prior research treats automatic Web service composition as a search problem in a state space, such that the execution of a Web service transits from one state to another, and the goal is to find an efficient plan that leads from the initial to some final state. Several planning techniques attempt to tackle this problem, such as forward/backward search (Akkiraju et al., 2006; McDermott, 2002; Paganelli and Parlanti, 2013; Park and Park, 2008), hierarchical task network planning (Sirin et al., 2004), planning based on model checking (Pistore et al., 2005; Zou et al., 2014), and planning based on Markov decision processes (Chen et al., 2009; Doshi et al., 2004). I/O messages of Web services in these works are often considered as variables in states or predicates and used for determining compatible Web services; yet the dependency between messages are seldom considered.

In fact, the problem of data dependency between services can be traced back to the enthusiastic research on model management in the 1980s, in which a model consists of a number of operators, each of which transforms a set of data objects to another subject to some assertions (Dolk and Konsynski, 1984). Thus, a model can be, in modern terminology, understood as a service. The focus in this line of research is how to construct a model management system (MMS) in order to support group decisions (T.P. Liang, 1988). Of the various issues pertaining to the development of a MMS, how to automatically select and integrate a subset of models so as to meet the requirement of a new and larger model, called automatic modelling, had attracted a lot of researches (T.P. Liang and Jones, 1988). Many graph-based models have been proposed for automatic modelling, e.g., entity relationship diagram (Bonczek et al., 1980), graph (T.P. Liang and Jones, 1988), and Meta-graph (Basu and Blanning, 1998, 2000). The goal is to identify a “good” process of models given a source and a target sets of data objects. However, details about data objects and their potential mismatch are not addressed in these works.

Some more recent works focus on data dependency between services for composing services (Gu et al., 2008; Q. A. Liang and Su, 2005; Xia and Yang, 2013; Zeng et al., 2008). Liang and Su (Q. A. Liang and Su, 2005) use an AND/OR graph to represent dependency among data and operation nodes, which originate from Web services in some service categories. A bottom-up search strategy can reveal a complete solution subgraph of the AND/OR graph with minimum cost. Gu et al. (Gu et al., 2008) extend their work by distinguishing the instance level from the abstract level in an enhanced service dependency graph. They also identify cardinality relationships of messages exchanged between two operations and suggest using XSLT or XPath for the attribute transformation. Zeng et al. (Zeng et al., 2008) define three rules, forward-chaining, backward-chaining, and data flow, to indicate the preconditions of a task, the effects after executing it, and the data dependency among tasks, respectively. Their rules-inference approach constructs qualified execution plans

Published by AIS Electronic Library (AISeL), 2014
www.manaraa.com

and measures weighted QoS scores of the plans to identify the best plan. However, above studies ignore message mismatch issues, despite their importance for Web services in the real world, which often are developed by different providers without common standards for defining exchanged messages.

Messages that are semantically equivalent may differ in their structures and values. For example, an address element may appear as a single string or a combination of multiple strings, such as street, city, and state. Another example is the price element that could be based on different currencies. Such heterogeneities between source and target messages must be resolved to support sensible interoperations of Web services. Previous works resolve message mismatch issue by adapting source message to fit target schema. There are two approaches: identifying predefined conversion rule according to semantic heterogeneity (X. Li et al., 2013; Mrissa et al., 2007; Nagarajan et al., 2007) and generating conversion script in XSL through measuring message similarity (Boukottaya and Vanoirbeek, 2005; Lecue et al., 2008). In the former approach, a message element is annotated with the corresponding concept of ontology while the conversion rules between different concepts are defined in advance. The semantic heterogeneity between two messages can be identified using ontology matching and message adaptation can be realized by applying predefined conversion rules. Nagarajan et al. use a set of annotation attributes, provided by SAWSDL, to map the elements in WSDL to external ontology and develop conversion rules between ontology concepts and message elements. As a result, the specification of Web services in WSDL contains semantic annotations for message transformation. They propose a middleware architecture for invoking Web services, in which a data mediation module intercepts source SOAP body and transforms it into target SOAP body according to semantic annotation in WSDL (Nagarajan et al., 2007). However, a complex ontology that involves a large set of concepts causes explosion of conversion rules between different concepts. To reduce the complexity of ontology, some works abstract a small set of generic concepts and differentiate the variety by means of contexts. Mrissa et al. (Mrissa et al., 2007) enrich domain ontology with contextual information by attaching context modifiers to each concept. Message elements in WSDL can be annotated with concepts of domain ontology and their corresponding context modifier. They also propose a mediation process by which a BPEL process can be analyzed and additional mediator Web services can be automatically incorporated into the process to facilitate message transformation. A similar strategy is adopted by Li et al.(X. Li et al., 2013), but they applied SAWSDL to annotate message elements. They improved the generation of conversion rules by considering composite conversion and providing additional implementation methods, such as XPath functions.

In contrast to the first approach, the second approach generates conversion script according to structure differences between message schemas (Boukottaya and Vanoirbeek, 2005; Lecue et al., 2008). The difference is detected by similarity measuring approach. Algergawy et al. (Algergawy et al., 2010) aggregate two similarity measures, namely internal element similarity and external element similarity. Internal element similarity considers the features of message element, such as tag name, cardinality, data type, and annotation information. External element similarity focuses on the structure of message, such as ancestor path, children elements, leaf nodes, and sibling elements. They also proposed weighted-sum and nonlinear aggregation methods by which several experiments were conducted to demonstrate that nonlinear combination outperforms linear method and the recommendation values for similarity threshold are between 0.4 to 0.6. Boukottaya and Vanoirbeek (Boukottaya and Vanoirbeek, 2005) adopted a similar approach for measuring element similarity to match compatible elements between two XML

https://aisel.aisnet.org/pajais/vol6/iss2/2
DOI: 10.17705/1pais.06202
www.manaraa.com

schemas. They proposed a matching process to automatically generate XSLT scripts by analyzing structural differences between compatible elements. Based on the work of Boukottaya and Vanoirbeek, Lecue et al. (Lecue et al., 2008) incorporate semantic similarity into matching process to measure similarity between message schemas of interacting Web services and produce XSL transformations that reconcile data heterogeneity in Web service interoperations.

These studies all employ XQuery, XSLT, XPath, or user-defined functions implemented as Web services for the data transfer between Web services. To achieve efficiency in this data transformation, algebraic operations help manipulate the XML documents; even the input and output messages of Web service operations are XML documents. Jagadish et al. (Jagadish et al., 2001) therefore represent XML elements as data trees and propose a tree algebra, called TAX, that includes selection, projection, product, and grouping operators to retrieve desired elements from XML documents. They also show that any XQuery statement can be translated into an expression in TAX. Magnani and Montesi (Magnani and Montesi, 2006) introduce a general abstract data model to accommodate both relational and XML data. They confirm the validity of several equivalence rules, which are equally applicable to the query optimization of XML data. In addition, Frasincar et al. (Frasincar et al., 2002) propose three types of operators to manipulate XML documents and adopt a heuristic algorithm that employs 14 equivalence rules to transform a query tree into an optimized query tree.

Although previous research has paved the way for automatic Web service composition and resolution of data exchange between Web services, the global optimization of the entire execution plan involving data transformation is seldom addressed. Some works, however, address the data mismatch problem specifically for adjacent Web service execution. Gu et al. (Gu et al., 2008) discuss the cardinality mapping problem; for example, in the many-to-one relationship, each item in a set returned by a Web service operation must be fed into another Web service operation, in a phenomenon called repeated invocation. The service dependency graph can be enhanced to specify the repeated invocation, but no execution mechanism exists. To solve the problem of repeated invocation, Srivastava et al. (Srivastava et al., 2006) propose pipelined parallelism: Execution threads for Web service operations are launched, and a Web service operation is invoked immediately after the input is ready. Because all Web service operations are executed in parallel, the cost of the execution plan can be determined by the Web service operation that provides the maximum execution time. The authors therefore propose a bottleneck cost metric, combining message selectivity and Web service operation execution time, to evaluate the execution costs of different execution plans and choose the optimal one. In addition to repeated invocation and pipelined parallelism, our approach incorporates more data manipulation operators that optimize the execution of Web services–based workflows systematically.

3. WS-data model

Message exchange between SOAP-based Web services uses an XML format. An XML document is intrinsically a tree, as shown in existing XML data models (Fernandez et al., 2007). We follow the same practice.

3.1. WS-data tree and forest

Definition 1: (WS-data node) A WS-data node is a tuple $e = (name, txt, A)$, where *name* identifies the node, *txt* is its textual content, and *A* is a set of associated attribute-value pairs. A WS-data node (*name*, *txt*, *A*) corresponds to an element in an XML document, where *name*, *txt*, and *A* represent the tag name, textual content, and attributes of the element, respectively.

Definition 1: (WS-data tree) A WS-data tree of an XML document *D* is an ordered tree *T* of WS-data nodes that correspond to elements in *D*. The WS-data nodes in *T* preserve the

Published by AIS Electronic Library (AISeL), 2014
www.manaraa.com

same parent–child relationship of elements in *D*, and the order among children of each node in *T* follows that in *D*.

Figure 3 shows an example (response) SOAP message that contains inventory data. For simplicity, we focus on the <body> part of the message, which can be represented as a WS-data tree, as in Figure 4

Definition 3: (WS-data subtree) Given a WS-data tree $T = (V, E)$, we can construct a WS-data subtree from T by selecting a WS-data node $r \in V$, which serves as the root of the subtree.

```
<?xml version="1.0"?>
 <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
      <soapenv:Body>
           <ns1:ListProductInvResult xmlns:ns1="http://www.nsysu.edu.tw/sample/">
          <ns1:Merchant>Acer</ns1:Merchant>
                    <ns1:Year>2008</ns1:Year>
                    <ns1:Week>50</ns1:Week>
          <ns1:ProductInv Promotion='Y'>
                         <ns1:SKU>1416549714</ns1:SKU>
            <ns1:Category>Laptop Computer </ns1:Category>
             <ns1:Name>Acer Aspire One 150-1126</ns1:name>
                         <ns1:WeekInv>
                              <ns1:BOW>5</ns1:BOW>
                              <ns1:EOW>3</ns1:EOW>
                         </ns1:WeekInv>
                    </ns1:ProductInv>
           <ns1:ProductInv Promotion='N'>
                         <ns1:SKU>1416549725</ns1:SKU>
                         <ns1:Category>Laptop Computer </ns1:Category>
          <ns1:Name>Acer Aspire 7726-6307</ns1:name>
                         <ns1:WeekInv>
                              <ns1:BOW>4</ns1:BOW>
                             <ns1:EOW>2</ns1:EOW>
                         </ns1:WeekInv>
                    </ns1:ProductInv>
                   <ns1:ProductInv Promotion='Y'>
                         <ns1:SKU>1416549728</ns1:SKU>
                         <ns1:Category>Computer Peripheral</ns1:Category>
          <ns1:Name>Acer V173B 17" Monitor</ns1:name>
                         <ns1:WeekInv>
                              <ns1:BOW>1</ns1:BOW>
                             <ns1:EOW>2</ns1:EOW>
                         </ns1:WeekInv>
                    </ns1:ProductInv>
      </ns1:ListProductInvResult>
      </soapenv:Body>
</soapenv:Envelope>
```

Figure 3 - The Example of SOAP Message

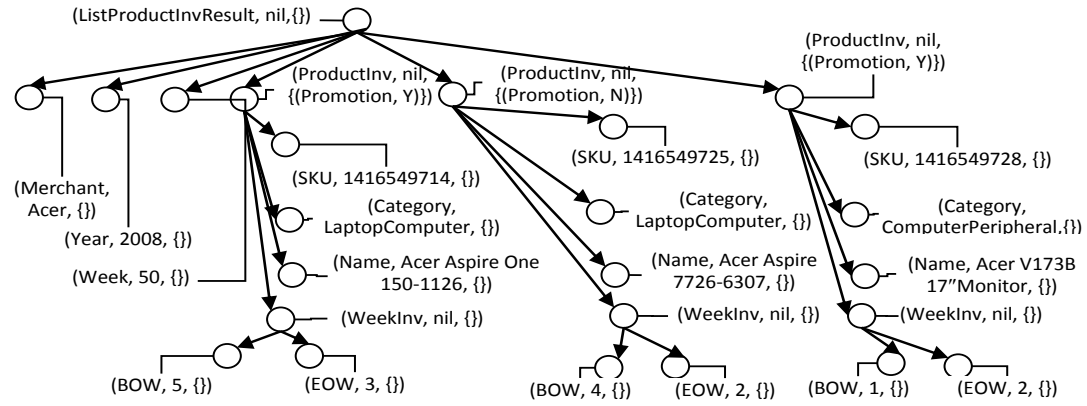


Figure 4 - The WS-Data Tree for the Body Part of the SOAP Message in Figure 3

https://aisel.aisnet.org/pajais/vol6/iss2/2
DOI: 10.17705/1pais.06202
www.manaraa.com

Definition 4: (WS-data forest) A data forest is a multiset of WS-data trees.

In Figure 5(a), we provide an example WS-data subtree of the WS-data tree from Figure 4, whose root has the tag name ProductInv. Figure 5(b) shows a data forest that contains three WS-data trees with the root tag name WeekInv.

3.2. WS-data algebra

We define a set of operators for manipulating WS-data forests. As we described in Section 2, previous research has demonstrated that operators defined in relational algebra can be applied to manipulate XML documents. Set operators, such as union, intersection, difference, and Cartesian product, also can be applied directly to the WS-data forest (which is a set); we do not explore this obvious application. Rather, we reveal how other primitive relational operators—selection, projection, and join—can be applied to WS-data forests.

Location paths defined in XPath 2.0 play indispensable roles for defining our proposed operators. A location path retrieves a set of elements from an XML document, and when applied to a WS-data tree, it returns a set of corresponding WS-data subtrees. Take the WS-data tree in Figure 4 as an example: The location path //ListProductInvResult/ProductInv returns a set of WS-data subtrees with root tag names of ProductInv, which is shown in Figure 6.

The location path takes as input a WS-data tree and generates a set of WS-data trees, or a WS-data forest. We propose an extraction operation that applies a location path to each WS-data tree of a given WS-data forest and returns the union of resultant forests.

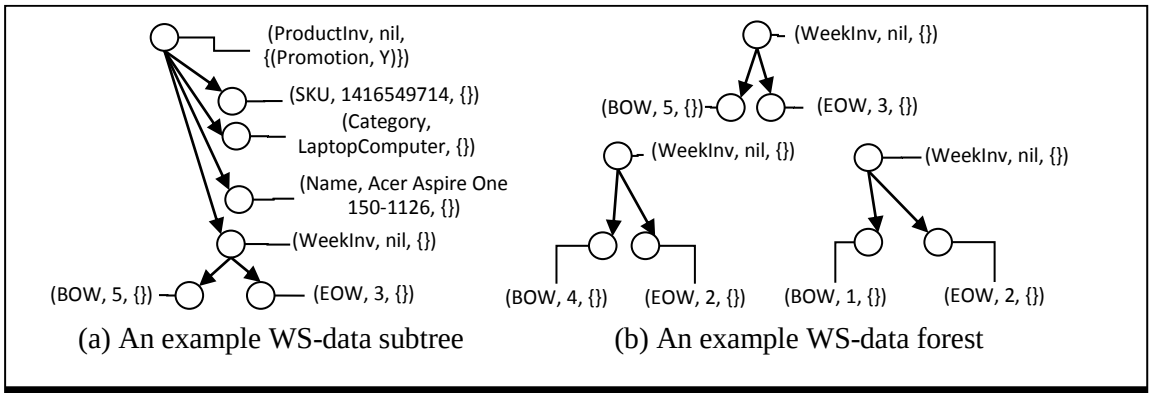


Figure 5 - The Example of WS-Data Subtree and WS-Data Forest

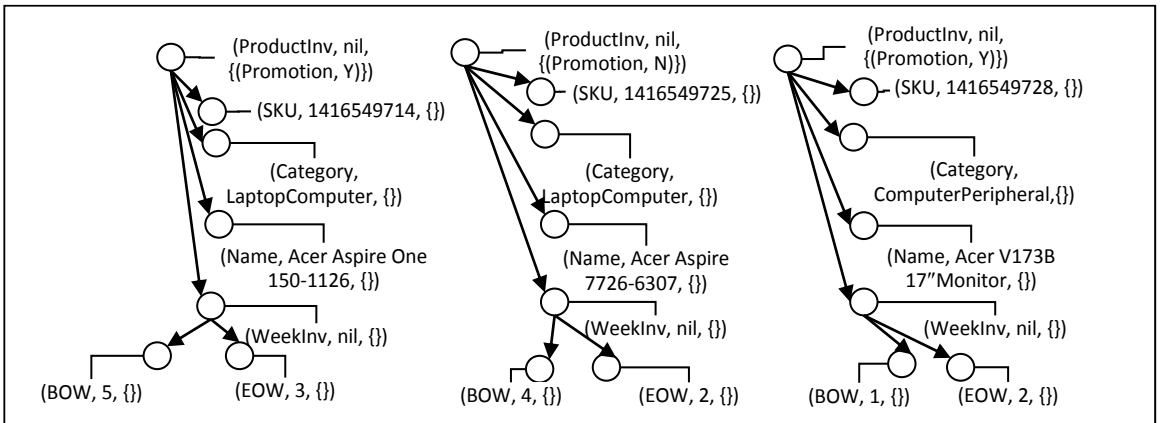


Figure 6 - Result of Applying Location Path //ListProductInvResult/ProductInv to the WS-Data Tree in Figure 4

Published by AIS Electronic Library (AISeL), 2014
www.manaraa.com

Definition 5: (Extraction) Given a WS-data forest F and a location path τ, the extraction operator $\chi_\tau(F)$ returns a WS-data forest that combines WS-data forests resulting from applying τ to each constituent WS-data tree in F. Formally, $\chi_\tau(F) = \cup_{t \in F} \tau(t)$.

If we let F_1 be a WS-data forest that consists of the single WS-data tree in Figure 4, the following WS-data expression produces the WS-data forest in Figure 6, according to the extraction operator with local path //ListProductInvResult/ProductInv:

$\chi_{\text{//ListProductInvResult/ProductInv}}(F_1)$.

Definition 6: (Selection) Given a WS-data forest F and a predicate P, specified as a location path, the selection operator $\sigma_P(F)$ returns a subset of F, such that each retaining WS-data tree returns a non-empty set when P applies to it. Formally, $\sigma_P(F)=\{t \mid t \in F \wedge P(t) \neq \varnothing\}$.

Then let the WS-data forest in Figure 6 be F_2. We retain inventory information about only products of the category LaptopComputer by applying the WS-data expression, $\sigma_{\text{//ProductInv[Category='LaptopComputer']}}(F_2)$. The result, displayed in Figure 7, shows that one WS-data tree, with category being ComputerPeripheral, has been excluded.

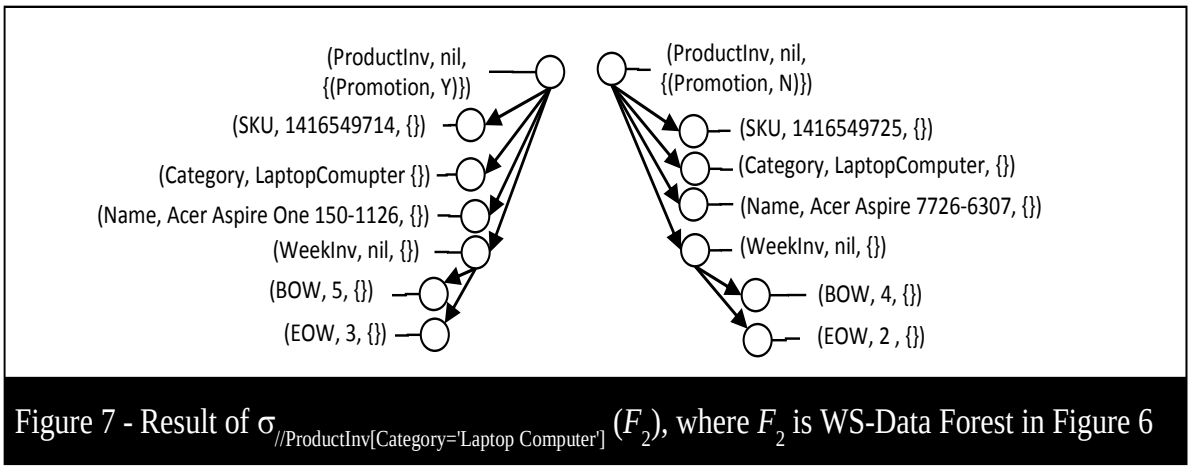


Figure 7 - Result of $\sigma_{\text{//ProductInv[Category='Laptop Computer']}}(F_2)$, where F_2 is WS-Data Forest in Figure 6

Definition 7: (Projection) Given a WS-data forest F and a location path τ for F, the projection operator $\pi_\tau(F)$ returns a forest in which each WS-data tree contains all WS-data subtrees obtained by applying τ to the WS-Data tree t, where $t \in F$, and all the nodes located in any path from the root of t to some WS-Data subtrees. Formally, $\pi_\tau(F)=\cup_{t \in F}\{t' \mid t'$ is a subtree of t, and n is a node in t' if $\exists\, t''(t'' \in \tau(t)$ and $(n \in t''$ or $\exists\ p$ (p is a path from root(t) to root(t'')), $n \in p))\}$.

For example, if we want to extract the SKU, Name, and EOW nodes in Figure 6, we might apply the following WS-data expression to obtain the desired outcome shown in Figure 8, which consists of SKU, Name, and WeekInv/EOW, as well as their common parent ProductInv:

$\pi_{\text{//ProductInv/(SKU | Name | WeekInv/EOW)}}(F_2)$.

Definition 8: (Join) Given two WS-data forests, $F=\{t_1, t_2, \ldots, t_n\}$ and $F'=\{t_1', t_2', \ldots, t_m'\}$, a join condition c specified on some location paths of WS-data trees from F and F', and a WS-data node r for each generated join root node, the join operator $F \bowtie_{c,r} F'$ returns a WS-data forest in which each constituent WS-data tree has a root node equivalent to r, whose left child and right child are WS-data trees from F and F', respectively, and both satisfy c. The join condition c includes one or more conditions connected using the Boolean operators AND, OR, or NOT. The form of the condition is $\tau_1\ \theta\ \tau_2$, where τ_1 and τ_2 are XPath expressions that specify some node contents or attribute values of WS-data trees in F and F', respectively, and θ is a comparison operator such as =, <, >, ≤, ≥, or ≠.

https://aisel.aisnet.org/pajais/vol6/iss2/2
DOI: 10.17705/1pais.06202
www.manaraa.com

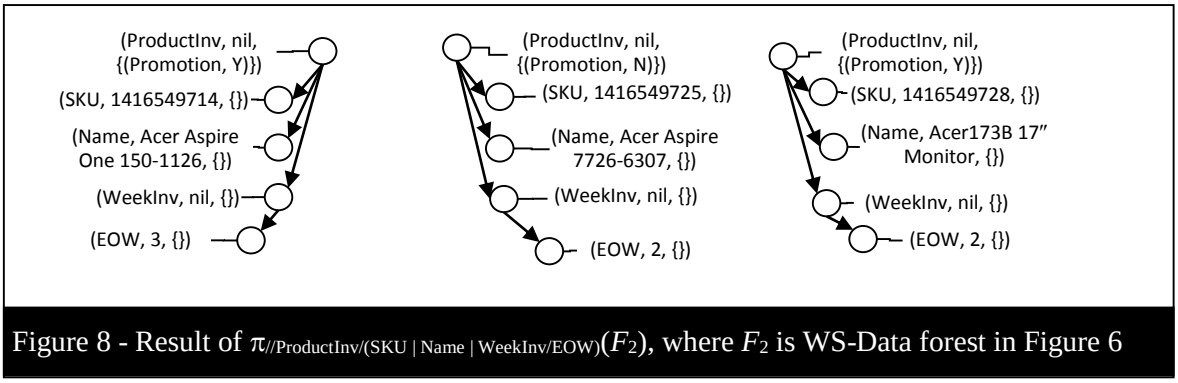


Figure 8 - Result of $\pi_{//ProductInv/(SKU \mid Name \mid WeekInv/EOW)}(F_2)$, where F_2 is WS-Data forest in Figure 6

Consider another WS-data forest F_3, as in Figure 9, that contains sales quantity information. To determine both the inventory and sales quantity of each product, we join F_2 and F_3 based on the SKU values. Specifically, we can specify the following expression:

$F_2 \bowtie_{//SKU[text()]=//SKU[text()],\ (prod\_root,\ nil,\ \{\})} F_3$.

Both the SKU values 1416549714 and 1416549728 appear in some WS-data trees in F_2 (Figure 6) and F_3 (Figure 9).

Therefore, the resultant WS-data forest in Figure 10 contains two WS-data trees whose root node takes the tag name prod_root, with the left and right child from F_2 and F_3, respectively.

Thus far we have defined operators that manipulate data; we next define a novel operator that involves Web service operations. We assume each Web service operation takes as input a WS-data tree (or XML document) and returns another WS-data tree, consistent with the specification of SOAP-based Web services.

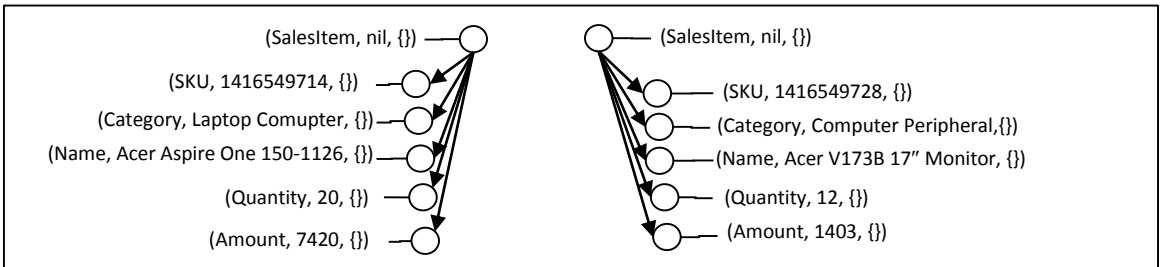


Figure 9 – The Example of WS-Data Forest F_3 for Product Sales

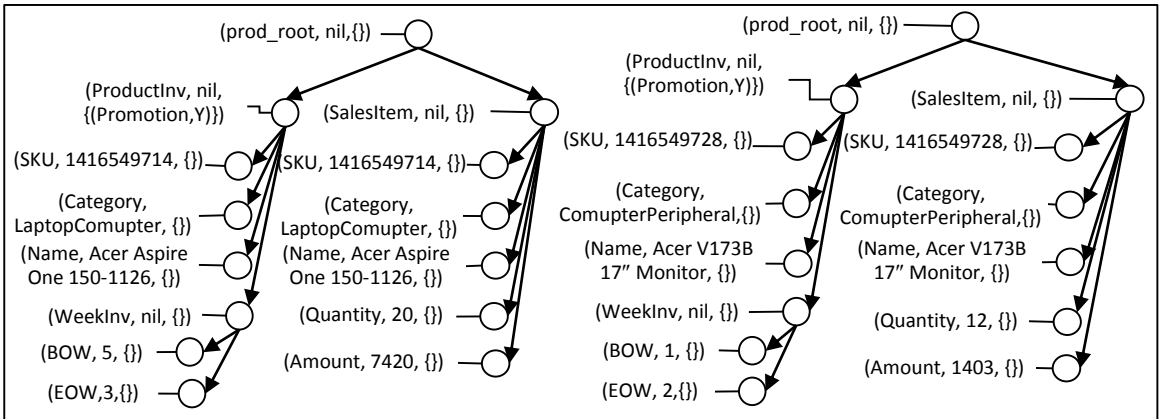


Figure 10 - Result of $F_2 \bowtie_{//SKU[text()]=//SKU[text()],\ (prod\_root,nil,\{\})} F_3$

Published by AIS Electronic Library (AISeL), 2014
www.manaraa.com

Definition 9: (Repeated invocation) A repeated invocation of a Web service operation *W.o*() on a forest *F*, denoted $\Gamma_{W.o()}(F)$, returns a WS-data forest that represents the union of invoking *W.o* on each WS-data tree of *F*. Formally, $\Gamma_{W.o()}(F) = \{W.o(t) \mid t \in F\}$.

Let F_4 be a WS-data forest containing WS-data trees about some desired products whose SKUs are 1416549714 and 1416549728. A Web service *ProdService* contains the operation getPrice() that takes as input a single product and returns its SKU, price, and order quantity.

The expression $\Gamma_{\text{ProdService.getPrice()}}(F_4)$ returns a WS-data forest, in which each constituent WS-data tree contains the price and ordered quantify for each product in F_4, as we show in Figure 11.

Each Web service operation has a specific format for its input message. Before invoking a Web service operation, we must prepare an input message that conforms to its format. We consider message heterogeneity in the WS-data model and treat the data conversion process as a special Web service operation, which can be used with a repeated invocation operator to achieve an appropriate data conversion.

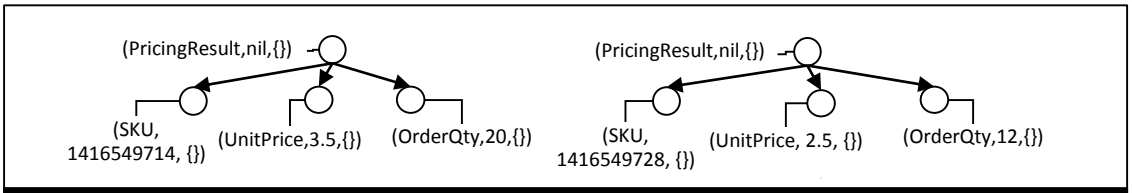
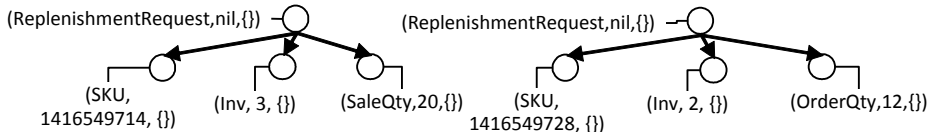


Figure 11 - Result of $\Gamma_{\text{ProdService.getPrice()}}(F_4)$

Consider the operation Replenishment() of the Web service *OrderService*, whose input message type appears in Figure 12(a). It takes SKU, current inventory level (Inv), and sales quantity (SaleQty) as input and generates a replenishment plan. The required input can be derived from the result of the join operation in Figure 10 (named F_5); specifically, the contents of SKU, Inv, and SaleQty correspond to the element contents of SKU, EOW, and Quantity. Let the Web service operation CVT.Sale2Replenishment() implement this conversion. The expression $\Gamma_{\text{CVT.Sale2Replenishment()}}(F_5)$ then converts the WS-data trees in Figure 10 into the desired WS-data forest in Figure 12(b), which serves as the input to OrderService.Replenishment().

```
<xsd:element name="ReplnishmentRequest">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="SKU" type="xsd:stirng"/>
      <xsd:element name="Inv" type="xsd:integer"/>
      <xsd:element name="SaleQty" type="xsd:integer"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

(a) Input message type for OrderService.Replenishment()



(b) Final result

Figure 12 - Results of $\Gamma_{\text{CVT.Sale2Replenishment()}}(F_5)$, where F_5 is the WS-Data Forest in Figure 10

https://aisel.aisnet.org/pajais/vol6/iss2/2
DOI: 10.17705/1pais.06202
www.manaraa.com

By properly combining several WS-data operations to form a WS-data expression, we can satisfy data requirements. For example, consider a Web service *RetailService* that contains an operation ListProductSales(*t*) that takes as its input the merchant and a specific week (WS-data tree *t*) and then generates detailed sales information for each item (also represented as a WS-data tree). Figure 13 provides the sample output of RetailService.ListProductSales(*t*), listing sales information about several items sold by the merchant in that week, as well as relevant related information. The following WS-data expression can derive information about sale items:

$\chi_{\text{//ListProductSalesResult/SalesItem}}(\Gamma_{\text{RetailService.ListProductSales()}}(\{t\}))$.

We provide the result of this WS-data expression in Figure 14. To list only the ordered items with quantity greater than 20, we add another selection operator, resulting in the following WS-data expression:

$\sigma_{\text{//Quantity[text()>20]}}(\chi_{\text{//ListProductSalesResult/SalesItem}}(\Gamma_{\text{RetailService.ListProductSales()}}(\{t\})))$.

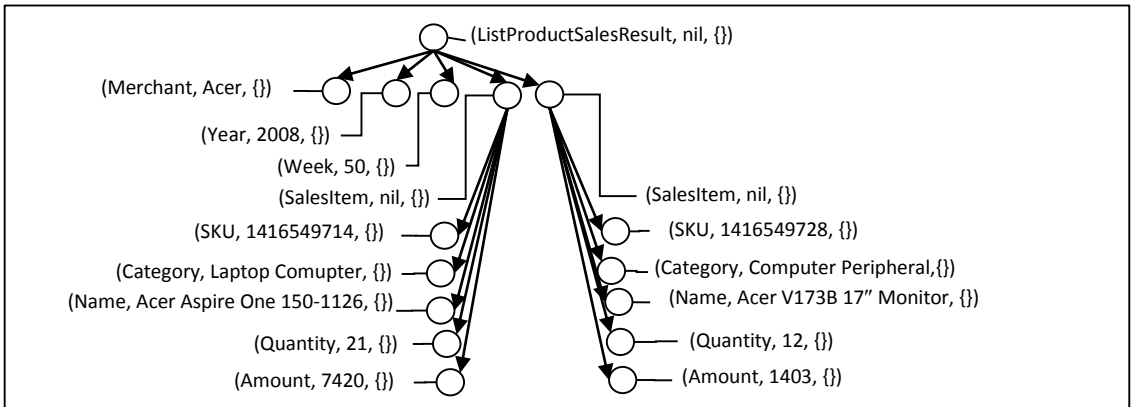


Figure 13 - Sample Result of RetailService.ListProductSales(*t*)

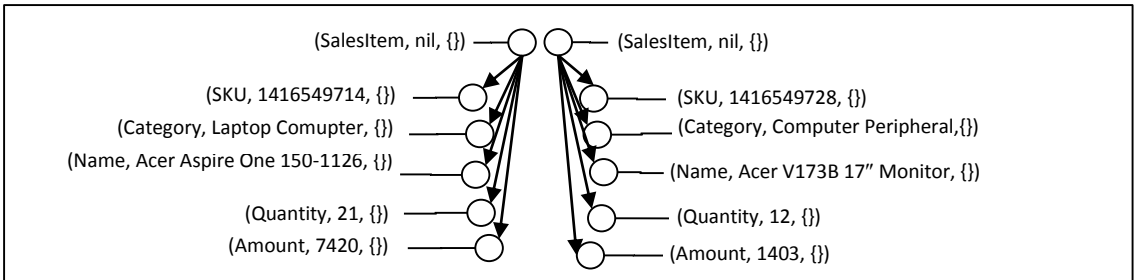


Figure 14 - Result of $\chi_{\text{//ListProductSalesResult/SalesItem}}(\Gamma_{\text{RetailService.ListProductSales()}}(\{t\}))$

Published by AIS Electronic Library (AISeL), 2014
www.manaraa.com

3.3. Equivalence rules

Operators from relational algebra offer some nice properties that enable a functionally equivalent, optimal rearrangement of operators in a query expression. Most operators borrowed from relational algebra preserve the same equivalence rules (Frasincar et al., 2002; Ullman, 1989). In addition, the WS-data model includes novel equivalence rules that apply to its unique operators.

Rule 1 (Cascade of selections): When a selection operator is associated with a conjunction of several predicates, it can be split into a cascade of selection operators, each associated with a single predicate. Formally, $\sigma_{P_1 \wedge \ldots \wedge P_n}(F) \equiv \sigma_{P_1}(\cdots(\sigma_{P_n}(F))$, where P_i, $1 \le i \le n$, is a predicate, and F is a WS-data forest.

Rule 2 (Commutativity of selections): When a WS-data expression involves a cascade of two selections, their execution order can be swapped without affecting the final result. Formally, $\sigma_{P_1}(\sigma_{P_2}(F)) \equiv \sigma_{P_2}(\sigma_{P_1}(F))$, where P_1 and P_2 are predicates, and F is a WS-data forest.

Rule 3 (Commutativity of selection with projection): When a WS-data expression involves a cascade of selection and projection operators, in which the selection attributes is a subset of the projection attributes, their execution order can be swapped without affecting the final result. Formally, $\sigma_P(\pi_\tau(F)) \equiv \pi_\tau(\sigma_P(F))$, where P is the predicate, τ is the location expression, and F is a WS-data forest.

Rule 4 (Selection push-down): When a selection operator is applied to the output of a join operator, it can be applied first to the operands of the join operator without affecting the final result. Formally, $\sigma_P(F \bowtie_{\rho, (\text{prod\_root, nil, } \{\})} G) \equiv \sigma_P(F) \bowtie_{\rho, (\text{prod\_root, nil, } \{\})} \sigma_P(G)$, where P and ρ are the predicates for selection and join, respectively, and F and G are two WS-data forests.

Rule 5 (Commutativity of selection with union): When a selection operator is applied to the output of a union operator, it can be applied first to the operands of union before employing the union operator without affecting the final result. Formally, $\sigma_P(t_1 \cup \ldots \cup t_n) \equiv \sigma_P(t_1) \cup \ldots \cup \sigma_P(t_n)$,where t_i, $1 \le i \le n$, is a WS-data tree, and P is the predicate.

Rule 6 (Commutativity of projection with union): When a projection operator is applied to the output of a union operator, it can be applied first to the operands of the union before employing the union operator, without affecting the final result. Formally, $\pi_\tau(t_1 \cup \ldots \cup t_n) \equiv \pi_\tau(t_1) \cup \ldots \cup \pi_\tau(t_n)$, where t_i, $1 \le i \le n$, is a WS-data tree, and τ is the location path.

Rule 7 (Commutativity of extraction with union): When an extraction operator is applied to the output of a union operator, it can be applied first to the operands of the union before employing the union operator, without affecting the final result. Formally, $\chi_\tau(t_1 \cup \ldots \cup t_n) \equiv \chi_\tau(t_1) \cup \ldots \cup \chi_\tau(t_n)$, where t_i, $1 \le i \le n$, is a WS-data tree, and τ is the location path.

Rule 8 (Distributivity of RI-selection): When a selection operator is applied to the output of a repeated invocation, it can be concatenated to the associated Web service operation. Formally, $\sigma_P(\Gamma_{W.o()})(F)) \equiv \Gamma_{\sigma_P(W.o())}(F)$, where F is a WS-data forest, and P is a predicate. Note that $\sigma_P(W.o())$ is a concatenation of $W.o()$ and the selection operator (σ_P), which first applies $W.o()$ to the given WS-data tree. The output WS-data tree then can be filtered by the selection operator (σ_P), depending on whether it satisfies P.

Rule 9 (Distributivity of RI-projection): When a projection operator is applied to the output of a repeated invocation, it can be concatenated to the associated Web service operation. Formally, $\pi_\tau(\Gamma_{W.o()}(F)) \equiv \Gamma_{\pi_\tau((W.o()))}(F)$, where F is a WS-data forest, and τ is the location path. Similarly, $\pi_\tau(W.o())$ is a

https://aisel.aisnet.org/pajais/vol6/iss2/2
DOI: 10.17705/1pais.06202
www.manaraa.com

concatenation of $W.o()$ and the projection operator (π_τ), which first applies $W.o()$ to the given WS-data tree and then projects the result by the projection operator (π_τ).

Rule 10 (Distributivity of RI-extraction): When an extraction operator is applied to the output of a repeated invocation, it can be concatenated to the associated Web service operation. Formally, $\chi_\tau(\Gamma_{W.o()}(F)) \equiv \Gamma_{\chi_\tau((W.o()))}(F)$, where F is a WS-data forest, and τ is a location path. Again, $\chi_\tau(W.o())$ is regarded as a concatenation of $W.o()$ and the extraction operator (χ_τ).

Rule 11 (RI-expansion): A repeated invocation on a WS-data forest F, $\Gamma_{W.o()}(F)$, is equivalent to the union of two repeated invocations on F_1 and F_2, where $\{F_1, F_2\}$ is a partition of F. Formally, $\Gamma_{W.o()}(F) \equiv \Gamma_{W.o()}(F_1) \cup \Gamma_{W.o()}(F_2)$, where F is a WS-data forest, and $\{F_1, F_2\}$ is a partition of F.

Rule 12 (RI-pipeline): A cascade of repeated invocations, $\Gamma_{W_2.o_2()}(\Gamma_{W_1.o_1()}(F))$ is equivalent to the repeated invocation on a concatenation of $W_1.o_1()$ and $W_2.o_2()$ on F. Formally, $\Gamma_{W_2.o_2()}(\Gamma_{W_1.o_1()}(F)) \equiv \Gamma_{W_2.o_2(W_1.o_1())}(F)$, where F is a WS-data forest.

4. Representation of Web service composition

The set of equivalent rules supports the transformation of a WS-data expression to another that can be executed more efficiently. For example, consider the RI-pipeline rule $(\Gamma_{W_2.o_2()}(\Gamma_{W_1.o_1()}(F)) \equiv \Gamma_{W_2.o_2(W_1.o_1())}(F))$. In this case, $\Gamma_{W_2.o_2(W_1.o_1())}(F)$ is usually more efficient because WS-data trees in F can be concurrently processed by $W_2.o_2()$ and $W_1.o_1()$ in a pipelined manner. We next describe a scenario that involves several component Web services, as well as abstract services, or interfaces that allow several implementations yet achieve the same functionality. This scenario also serves to illustrate our cost model.

4.1. Scenario for Web service composition

We consider a vendor-managed inventory (VMI) business model in business-to-business e-commerce, an extension of the replenishment process described in Section 1. Many supply chains use VMI successfully to prevent out-of-stock situations while still reducing inventory in the supply chain. With VMI, the supplier takes full responsibility for maintaining an established inventory level for the retailer, determines the replenishment order quantity, and delivers products to the retailer in a regular basis.

We assume that the retailer provides two Web service operations. ListProductInv() and ListProductSales(), that return inventory and sales data, respectively, in a specified period. Information about stock held at the distribution center and by the transportation agent is also crucial. Thus, the distribution center and transportation agent offer Web service operations ListDCInv() and ListGITInv(), respectively, to indicate their inventory. Stock information at various stages of the supply chain then can be aggregated using the supplier-provided abstract Web service operation, ListTotalStock(). Furthermore, the supplier offers a Web service operation Replenishment() to compute the replenishment order quantity according to the total stock, a Web service operation getPrice() for the price of a single product, and a Web service operation PriceList() for the prices of all products.

The goal of the query in this scenario is to list the order quantities and prices of promoted laptop computers that need replenishment. Starting with the concurrent invocation of ListDCInv(), ListGITInv(), and ListProductInv(), it obtains stock information across the entire supply chain. The stock information then can be aggregated by invoking ListTotalStock(), and the query obtains product sales information in a specific period (e.g., week) by invoking ListProductSales(). Both stock and sales information serve as inputs to

Published by AIS Electronic Library (AISeL), 2014
www.manaraa.com

Replenishment() to determine suitable replenishment order quantities. Because the focus of the query is laptop computers, only the replenishment orders for promoted products in the category LaptopComputer with order quantity greater than 0 are needed. Finally, the prices of replenishment products are based on their order quantities, determined by invoking getPrice() or PriceList().

To simplify the query representation, we use an abstract service to determine product prices, which supports several implementations. We use the notation $\Lambda_{AWS.op(),\,I}(F)$ to represent the application of an abstract service to a WS-data forest *F*, where *AWS.op*() is the abstract service operation, and *I* is the set of functionally equivalent implementations. For example, the determination of product prices can be implemented by repeatedly invoking getPrice() for each item from the order list or by joining the order list and price list returned by getPriceList(). In the following two implementations, *F* is a WS-data forest about ordered products and CVT.rule2() is a data conversion Web service operation that converts a WS-data tree to the desired format:

1. $\Gamma_{\text{ProdService.getPrice()}}(F)$.
2. $\Gamma_{\text{CVT.rule2()}}(F \bowtie_{\text{//SKU[text()]=//SKU[text()]}, (\text{prod\_root, nil, \{\}})} \chi_{\text{//ProductItem}}(\Gamma_{\text{ProdService.PriceList()}}()))$.

Figure 15 depicts the entire query tree. The initial query tree uses two abstract services, $\Lambda_{StockService.\text{ListTotalStock}(),I_1}()$and $\Lambda_{PriceService.\text{Pricing}(),I_2}()$, to aggregate stock information and determine replenished product prices, respectively. For clarity, we portray each WS-data expression for manipulating WS-data as a box with curved angles and the invocation for abstract services as a box with dotted lines. The equivalent WS-data expressions of the initial query tree are:

$E_1 = \Lambda_{StockService.\text{ListTotalStock}(),\,I_1}()$.

$E_2 = \Gamma_{\text{SalesService.ListProductSales()}}()$.

$E_3 = \chi_{\text{//ProductInv}}(E_1) \bowtie_{\text{//SKU[text()]=//SKU[text()], (prod\_root, nil, \{\})}} \chi_{\text{//SalesItem}}(E_2)$.

$E_4 = \sigma_{\text{//Category[text()='LaptopComputer']}\wedge\text{//ProductInv[@Promotion='Y']}}(E_3)$.

$E_5 = \Gamma_{\text{CVT.rule1()}}(E_4)$.

$E_6 = \Gamma_{\text{OrderService.Replenishment()}}(E_5)$.

$E_7 = \sigma_{\text{/ReplenishmentResult[OrderQty>0]}}(E_6)$.

$E_8 = \Gamma_{PriceService.\text{Pricing}(),\,I_2}(E_7)$.

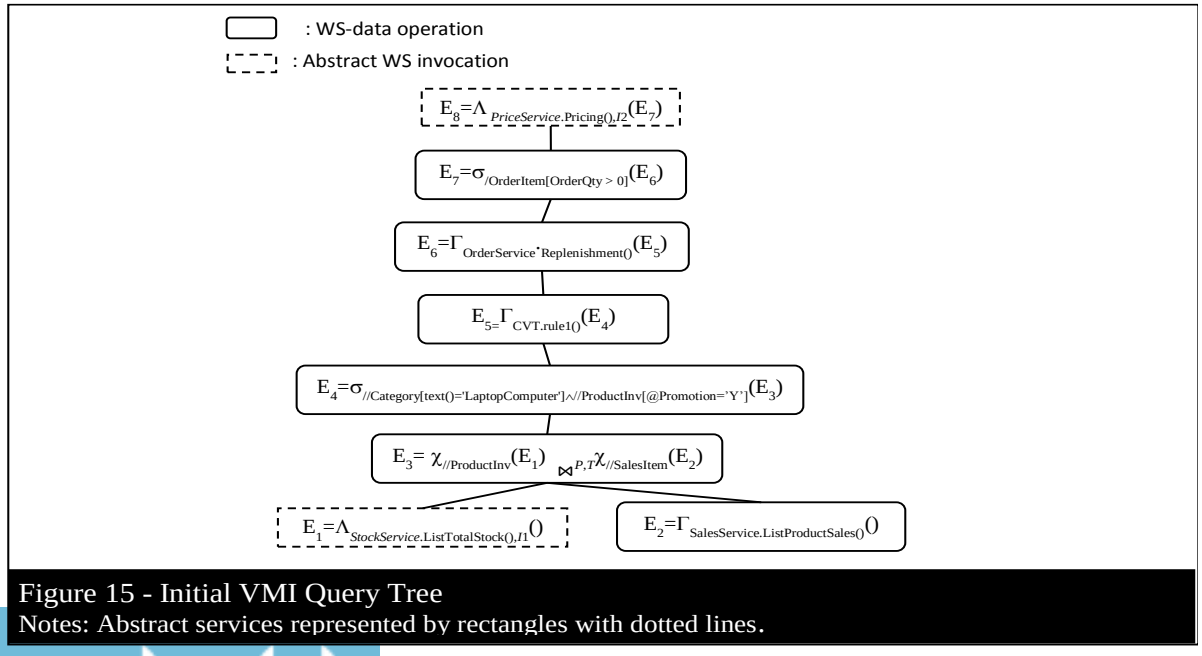


Figure 15 - Initial VMI Query Tree
Notes: Abstract services represented by rectangles with dotted lines.

https://aisel.aisnet.org/pajais/vol6/iss2/2
DOI: 10.17705/1pais.06202
www.manaraa.com

The execution plan can be derived by choosing one implementation for each abstract service. Figure 16 shows an executable query tree that joins results from ListDCInv(), ListGITInv(), and ListProductInv() and the repeated invocation of getPrice() for abstract services, $\Lambda_{StockService}$.ListTotalStock() and $\Lambda_{PriceService}$.Pricing(). The shaded nodes, numbered 1–6, constitute the implementation of $\Lambda_{StockService}$.ListTotalStock(),, and the shaded node, numbered 13, forms the implementation of $\Lambda_{PriceService}$.Pricing().

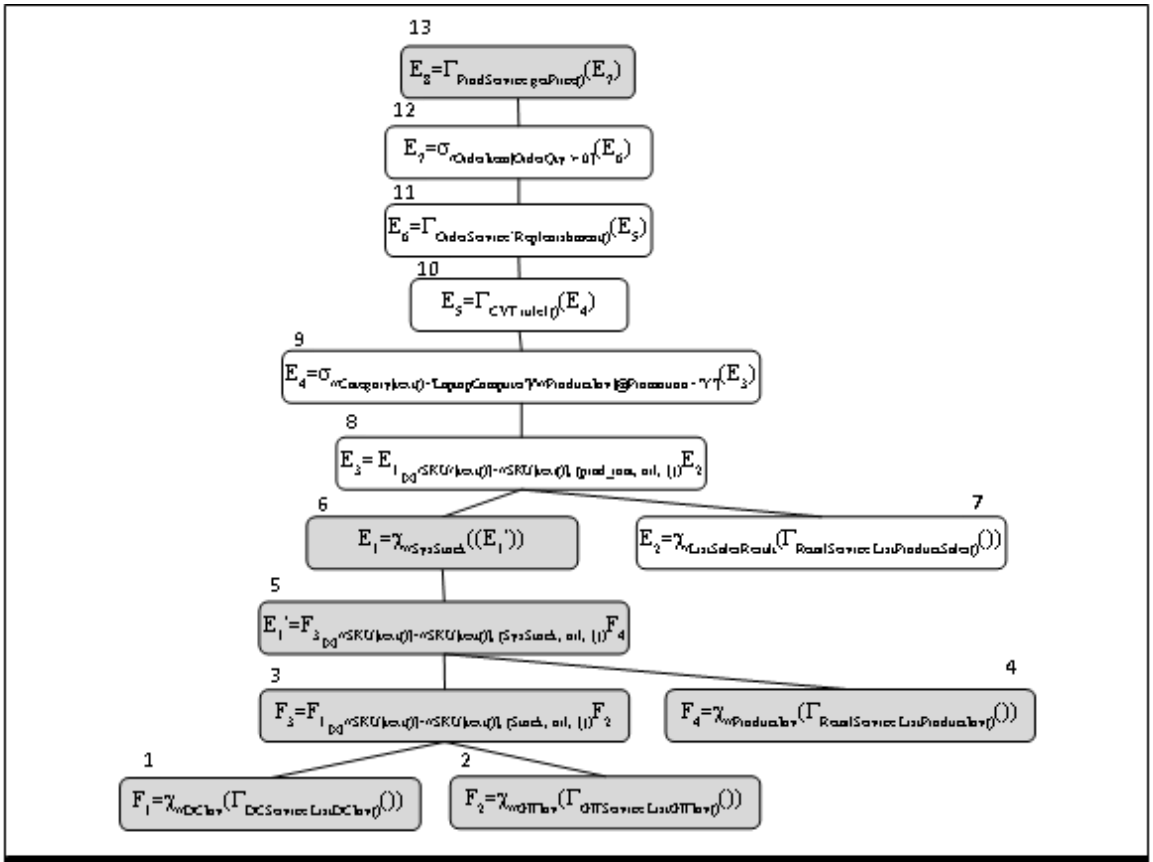


Figure 16 - Executable VMI Query Tree
Notes: Implementations of abstract services represented by shaded boxes.

4.2. WS-data cost model

We propose a cost model for estimating the execution time of an executable query tree. Because a query tree is iteratively constructed using nodes, each involving a WS-data operator, its cost can be derived as soon as we can estimate the cost of each WS-data operator. There are four means to estimate the cost of a WS-data expression involving different operators.

Cost equation 1: Cost of repeated invocation on a single operation $E_0=\Gamma_{W.o()}(E_1)$. Here, cost(E_0) = $|E_1|\times$time(W.o())+cost(E_1), where the cardinality of E_1, denoted $|E_1|$, is the number of WS-data trees in the result of E_1, and time(*W.o*()) denotes the response time of the physical Web service operation (*W.o*()). For example, consider E_6 (node 11) in Figure 16. Assume that the response time of the Web service operation *OrderService*.Replenihsment() is the 10 ms, the cost of E_5 is 250 ms, and the cardinality of the result of E_5 is 500; the cost of E_6 is 500 $\times$ 10 + 250 = 5250 ms.

Published by AIS Electronic Library (AISeL), 2014
www.manaraa.com

Cost equation 2: Cost of repeated invocation on a sequence of operations $E_0=\Gamma_{W_n.o_n(\ldots(W_1.o_1())\ldots)}$ (E_1). To execute $\Gamma_{W_n.o_n(\ldots(W_1.o_1())\ldots)}$ (E_1), we can execute $W_n.o_n$, …, $W_1.o_1$ in a pipelined fashion. Therefore, cost(E_0) = ($|E_1|$–1)×Max$_{1\le i\le n}$time($w_i.o_i$)+$\Sigma_{1\le j\le n}$time(w_j,o_j)+cost(E_1)+$|E_1|\times n\times\delta$, where time($W_i.o_i$) and time($W_j.o_j$) are response times of the physical Web service operations, $W_i.o_i$ and $W_j.o_j$ respectively, and δ is the overhead for data transmission. Consider nodes 11 to 13 in Figure 16. By applying equivalence rules 8 and 12, we can combine them into the following expression that involves three operations:

$E_8 = \Gamma_{\text{ProdService.getPrice}(\sigma_{\text{//OrderItem[OrderQty>0]}}(\text{OrderService.Replenishment}()))}(E_5)$.

Suppose the response times of each invocation for selection (OrderQty>0), *ProdService*.getPrice() and *OrderService*.Replenishment(), are 180 ms, 500 ms, and 300 ms, respectively; the cost of E_5 is 7,660 ms; and the cardinality of E_5 is 200. The cost of E_8 therefore is 199 × 500 + (500 + 300 + 180) + 7660 + 200 × 3 × δ = 108140 + 600δ ms.

Cost equation 3: Cost of unary operation $E_0=O(E_1)$. Here O denotes a unary operator, which could be selection, projection, or extraction. The cost can be computed as cost(E_0) = time(O(E_1)) + cost(E_1), where time(O(E_1)) is the time to apply the unary operator O to the result of E_1, which is linearly proportional to $|E_1|$. We discuss the methods for determining the execution times of various unary and binary operators in Section 5.1. Consider the WS-data expression E_7 (node 12) in Figure 16. If the execution time of selection on the E_7 result is 90 ms, and the cost of E_6 is 5,250 ms, the cost of E_7 is 90 + 5250 = 5340 ms.

Cost equation 4: Cost of binary operation $E_0=E_1\ O\ E_2$. In this case, O denotes a binary operator, which could be join, union, or difference. Both E_1 and E_2 can be executed in parallel. Therefore, the cost of E_0 is cost(E_0) = time(O(E_1, E_2)) + max(cost(E_1), cost(E_2)), where time(O(E_1, E_2)) is the execution time of the binary operation O on E_1 and E_2, a function of $|E_1|$ and $|E_2|$. Consider the WS-data expression F_3 (node 3) in Figure 16, which involves a join operation. Its cost is the sum of the execution time for the join and the maximum cost of F_1 and F_2. Suppose that the execution time of the join is 300 ms and the costs of F_1 and F_2 are 4,340 ms and 4,500 ms, respectively. The cost of F_3 is 300 + max(4340, 4500) = 4,800 ms.

5. Experiments

In this section, we show how to determine empirically the execution time of each operator defined in WS-data algebra with different data sizes and then perform the cost estimation. We also describe an approach that applies the WS-data algebraic equivalence rules for transforming query trees. We implement all operators defined in WS-data algebra and seven Web services required for the stock replenishment process in Java. We further develop a query engine that executes WS-data query trees. All implementations are hosted on Amazon Elastic Compute Cloud (Amazon EC2). We finally report the results and compare them against analytical results obtained using the proposed cost model.

5.1. Performance of WS-data operators

We implemented all seven proposed WS-data operators: selection, projection, extraction, union, difference, join, and repeated invocation. The response times of the six operators that do not involve Web service operations (cf. repeated invocation) are measured. For each operator, we vary the

https://aisel.aisnet.org/pajais/vol6/iss2/2
DOI: 10.17705/1pais.06202
www.manaraa.com

cardinality of the input WS-data forest and collect its response time. In Table 1, we list the average response times of each operator on a PC server with an Intel Xeon 2.33 GHz CPU, running the Linux operating system, for different data sizes. We tried two methods to implement the join operator, hash join and nested loop, and found that hash join performed significantly better. Therefore, Table 1 contains the hash join results, and we adopt this method in our subsequent experiments.

The response time of each WS-data operator grows approximately linearly with the cardinality of the input WS-data forest, as Figure 17 shows. However, some operators are more sensitive to the cardinality of the input WS-data forest than others. Operators such as selection, join, and repeated invocation belong to this category, because of their greater processing overhead. According to our experiments, the growth slopes of projection and extraction are .08 and .006, respectively; those of selection and join are .6 and 1.3, respectively. Of the six WS-data operators, the join operator is the most time consuming, which coincides with previous relational algebra findings.

Table 1 - Execution Time of Operators with Different Data Size (ms)

Cardinality	Selection	Projection	Extraction	Union	Difference	Join
100	89.8	11.3	9.8	10.0	7.8	151.0
200	138.0	17.3	14.3	15.0	10.8	269.6
300	223.3	24.3	18.0	20.0	13.5	423.8
400	293.0	29.8	22.5	25.0	16.5	542.8
500	342.8	36.0	28.0	33.5	19.5	742.4
1320	876.0	102.2	75.7	72.8	44.3	1,749.8
3250	2,032.3	259.4	184.9	187.0	110.0	4,268.8

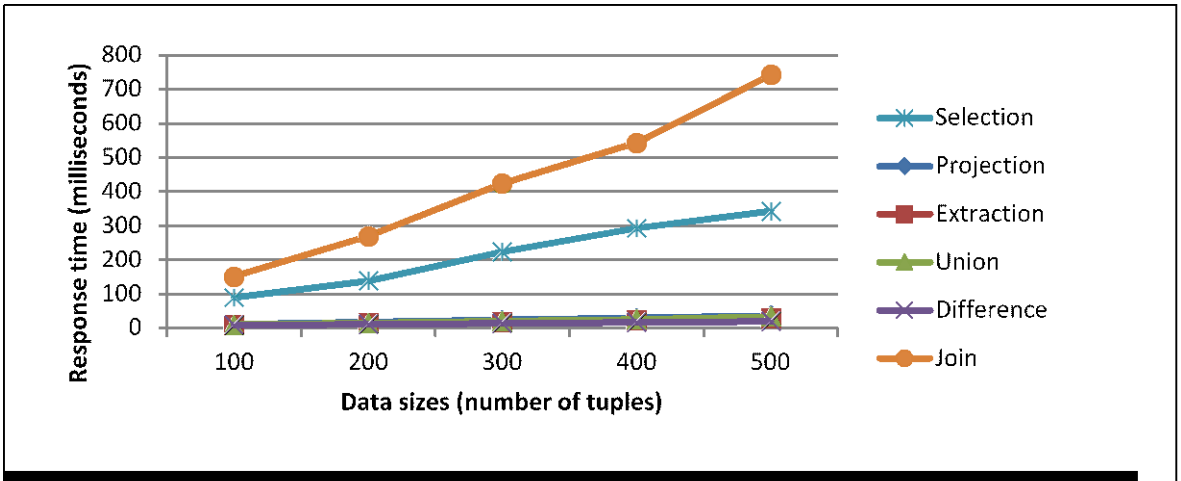


Figure 17 - Execution Times of WS-Data Operators with Different Input Sizes

Published by AIS Electronic Library (AISeL), 2014
www.manaraa.com

5.2. Cost estimation and optimization procedure

Using the cost model described in Section 4.2, we measure the cost of each node in an executable query tree. The execution times of the invoked Web services, the selectivity of the associated predicates, and the execution times of involved WS-data operators in the stock replenishment process appear in Tables 2, 3, and 4, respectively. Figure 18 depicts the estimated cost and output cardinality of each node in the executable query tree of Figure 16. The total execution time of the entire executable query tree is 154.4 seconds, and the cardinality of the final WS-data forest is 100.

We next transform the executable query tree in Figure 18 to another query tree by applying the equivalence rules from Section 3.3. To improve performance, we adopted a heuristic approach, similar to the query optimization method proposed by Ullman (Ullman, 1989) for optimizing data queries, as follows:
Step 1: Apply cascade of selections (i.e., rule (1)) to decompose multiple predicates into a cascade of selections.

Step 2: Apply selection rules (3 and 4) to push selection down as much as possible.

Table 2 - Invoked Web Services Profile

Web service operation	Execution time (seconds)	Default output cardinality
DCService.ListDCInv()	4	2000
GITService.ListGITInv()	4	2000
RetailService.ListProductInv()	4	2000
RetailService.ListProductSales()	4	2000
OrderService.Replenishment()	0.5	1
ProdService.PriceList()	3	2000
ProdService.getPrice()	0.3	1

Table 3 - Selectivity of Predicate

Predicate	Selectivity
//Category[text()='LaptopComputer']	0.5
//SalesItem[@Promotion='Y']	0.2
/ReplenishmentResult[OrderQty>0]	0.5

Table 4 - Average Execution Time of WS-Data Operators[1]

Operator	Average execution time per 100 tuples (seconds)
Selection	0.09
Extraction	0.01
Join	0.15

[1] These values were derived from the first experiment.

www.manaraa.com

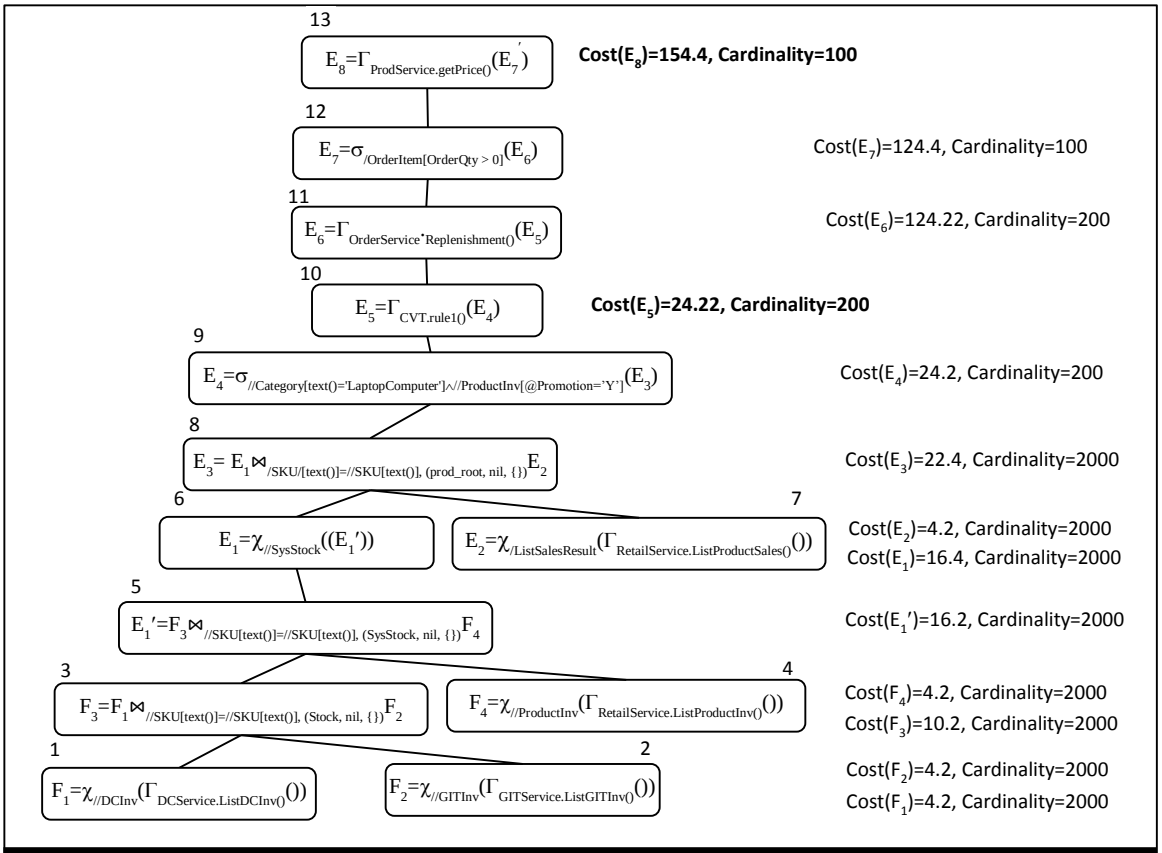


Figure 18 - Cost Estimate of the Executable Query Tree in Figure 16

Step 3: Use the most restrictive selection first (i.e., rule (2)) to reduce the cardinality of previous output.
Step 4: Apply RI rules (8, 9, and 10) to reduce the cardinality of Web service output.
After conducting steps 1 and 2 for the query tree in Figure 18, we split the predicates of WS-data expression E_4 in node 9 into a cascade of two selections, then push both selection operators down as far as possible. Thus, the predicate //Category[text()='LaptopComputer'] can be pushed down to right above nodes 1, 2, 4, and 7, and the predicate //SalesItem[@Promotion='Y'] can be pushed down to prior to node 8. After step 3, the selection with predicate //SalesItem[@Promotion='Y'] appears before that of //Category[text()='LaptopComputer'], because the former is more restrictive than the latter. Finally, by applying rule 8, as required in step 4, we combine nodes 11 and 12 in Figure 18. In the resultant query tree in Figure 19, we shade the updated nodes after applying the transformation steps for clarity.

According to Table 1, join is the most time-consuming operator among all WS-data operators. We expect performance improvements after applying our query transformation procedure because reducing input tuples through early selection should result in fewer tuples for the join. Comparing the cost(E_5) in node 10 of the original query tree in Figure 18 with the corresponding node 12 of the optimized query tree in Figure 19, we find costs of 24.22 and 12.62 Seconds, respectively, or a cost reduction of approximately 48%.

Next we consider the total costs of the original and optimized query trees, 154.4 and 142.8 seconds, respectively, which implies a small cost reduction (7.51%). Both query trees involve repeated invocations for the Web

Published by AIS Electronic Library (AISeL), 2014
www.manaraa.com

service operations Replenishment() and getPrice(), and the sequential execution of these two operations dominates the total cost. To alleviate this problem, we select the other abstract service implementation *PriceService*.Pricing(), which produces the transformed query tree in Figure 20. The revised query tree in this case reduces the total cost by almost 25%.

Alternatively, we might combine node 13 and 14 in Figure 19 into a single node (node 13), as we show in Figure 21, by applying equivalence rule 12. Thus, node 13 in Figure 21 allows for the pipelined execution of the Web service operations Replenishment() and getPrice() and the selection operator. The cost estimation of the allows for the pipelined execution of the Web service operations optimized query tree in Figure 21 is 112.92 + 600δ seconds, where δ is the overhead associated with transmitting a data item from one host to another. Because δ depends on networking and server performance, the cost improvement cannot be determined exactly. However, the optimized query tree in Figure 21 clearly outperforms the original executable query tree in Figure 18 by approximately 21% if δ is less than 15 ms.

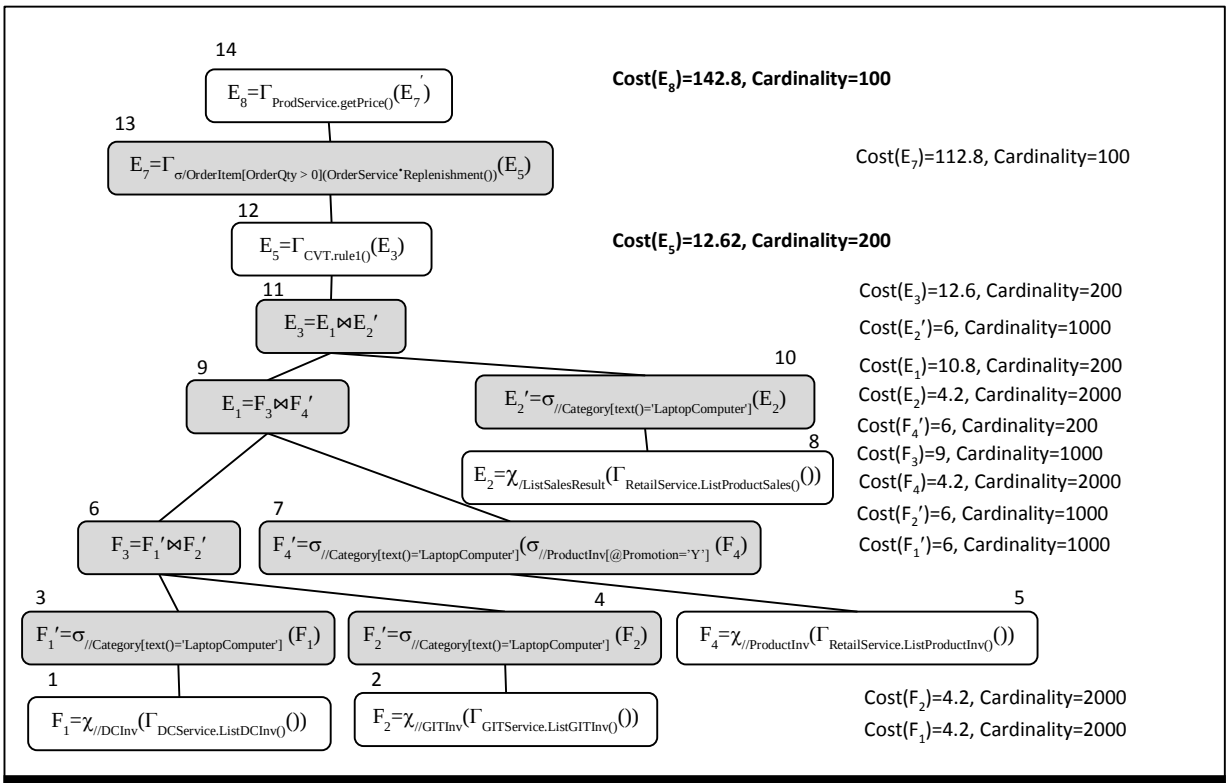


Figure 19 - Cost Estimate of the Optimized Query Tree
Notes: The shaded boxes are the revised nodes, after applying the equivalence rules.

Table 5 - Performance Comparison: Query Trees Executed on Amazon EC2

Stages / Query trees	E5 (seconds)	Reduction Ratio	Total (seconds)	Reduction Ratio
Original executable query tree in Figure 18	18.717	—	151.885	—
Optimized query tree in Figure 19	10.528	43.75%	143.738	5.36%
Optimized query tree in Figure 20	10.144	45.80%	113.083	25.55%
Optimized query tree in Figure 21	10.162	45.71%	119.073	21.60%

https://aisel.aisnet.org/pajais/vol6/iss2/2
DOI: 10.17705/1pais.06202
www.manaraa.com

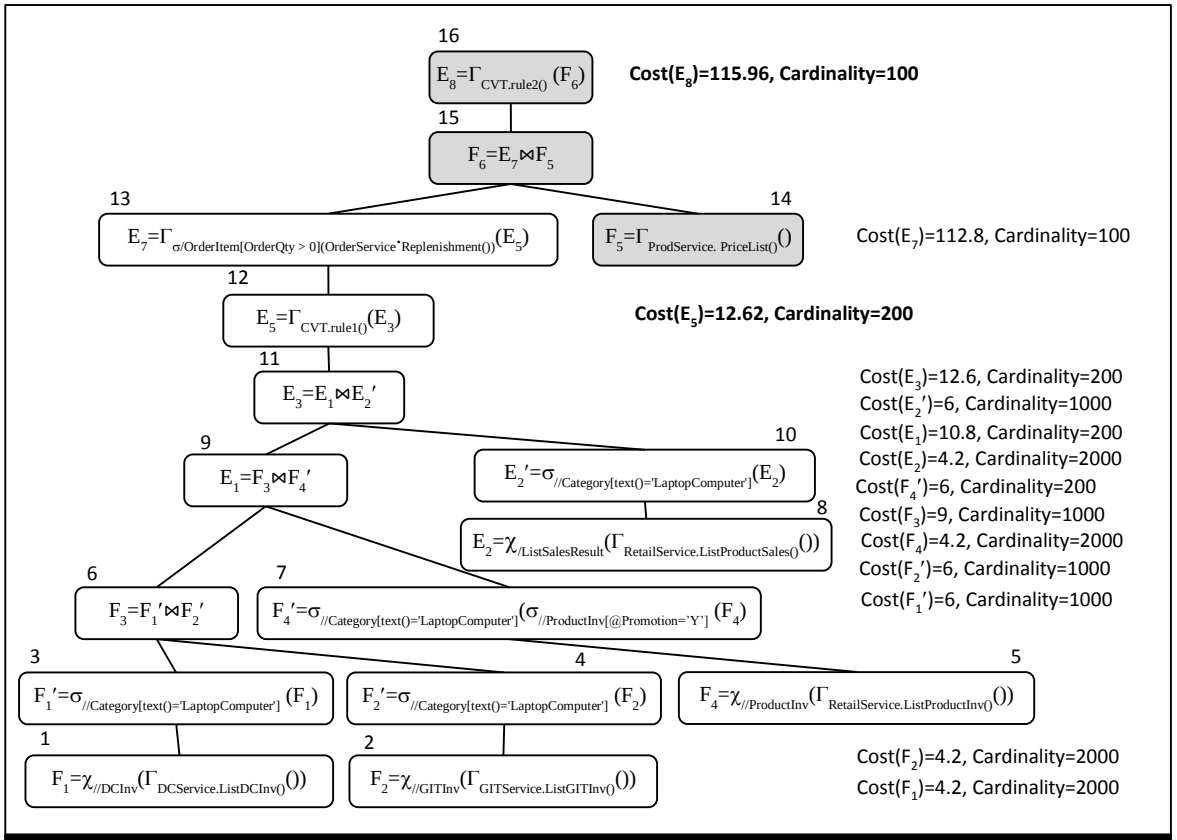


Figure 20 - Cost Estimate of the Optimized Query Tree with *PriceService*.Pricing()
Notes: The shaded boxes are the details of the implementation.

5.3. Comparing original and optimized query trees executed on Amazon cloud

To confirm the validity of our approximation measurement, we emulated the execution of the original and optimized executable query trees using the parameters in Table 2 and 3. Specifically, we deployed the Web services and query engine with two virtual machines of high CPU medium, located on Amazon EC2 on the Asia Pacific region. That is, the query engine is executed on one virtual machine, and Web services are deployed on another. We launched 10 runs for each query tree in Figures 18–21 and collected their execution times; we provide the performance results in Table 5. Compared with the analytical results, the average execution times in emulation are greater due to the overhead associated with transferring data and invoking operators. However, the cost reduction ratios match our estimation results. For example, for query processing up to E_5, the cost reductions of the optimized query tree in Figures 19, 20, and 21 are 43.75%, 45.8%, and 45.71%, respectively, and the performance improvements in terms of total response times are 5.36%, 25.55%, and 21.60%, respectively.

Published by AIS Electronic Library (AISeL), 2014
www.manaraa.com

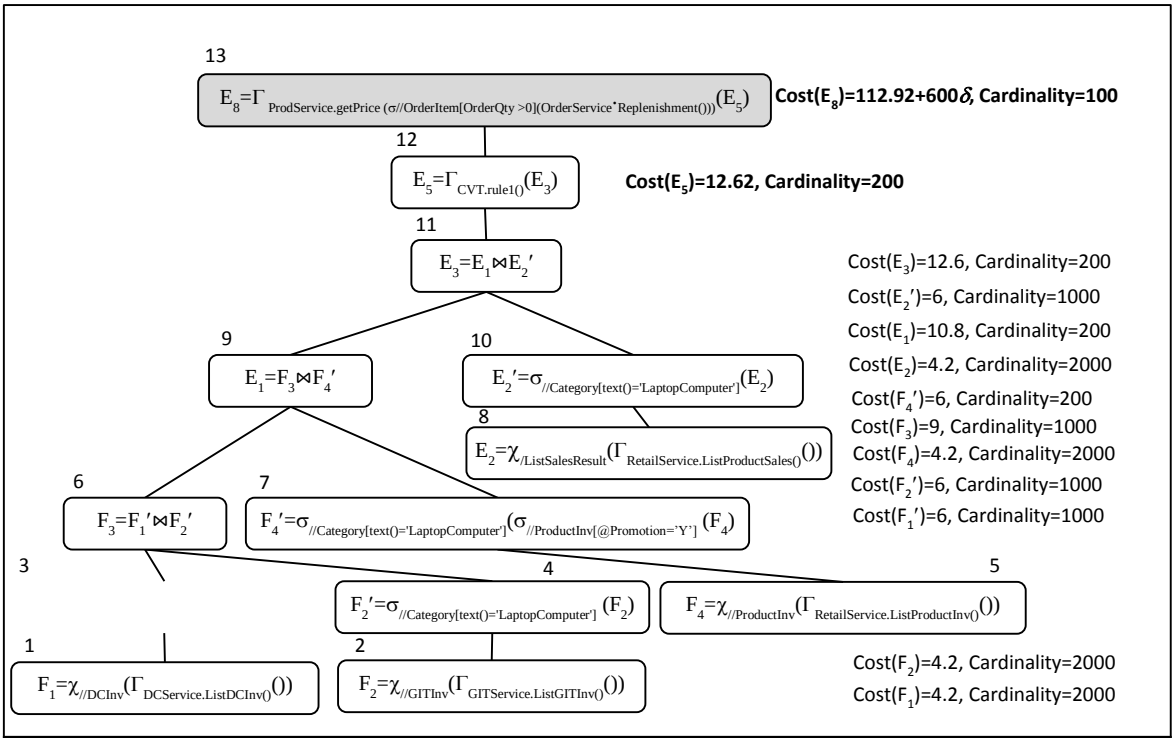


Figure 21 - Cost Estimate of the Optimized Query Tree in Pipeline Execution
Notes: The shaded box is the result of combining nodes 13 and 14 from Figure 19.

6. Conclusion

We have proposed a WS-data model that represents the input/output messages of Web service operations as WS-data trees. It involves several operators, similar to relational operators, that mediate the input/output message across Web services, including projection, selection, extraction, and join. In addition, the invocation of Web service operations is possible in the WS-data model because of the repeated invocation operator. Finally, we introduce the idea of abstract services to simplify the representation of composite Web services in WS-data expression.

We studied the equivalent rules pertaining to the WS-data model, which pave the way for optimizing a query tree based on a WS-data expression. We also develop a cost model to estimate the cost of a query tree. The heuristic procedure we describe can transform a query tree according to equivalence rules; we illustrate this approach with a VMI stock replenishment process and reveal that the optimized query tree reduces costs significantly in emulation experiments conducted on the Amazon EC2 platform.

In further research, we plan to study strategies for selecting the appropriate implementation of abstract Web service to reduce the number of executable query trees and improve the performance of the optimized query trees. Our current cost model describes the response time of Web service operation as a fixed value (e.g., average response time), but in real world, it may be a probability distribution with different parameters, such as input cardinality. The best means to estimate the cost of a query tree in such settings deserves further investigation.

Acknowledgement

This work was supported in part by the National Science Council in Taiwan under Grant NSC 101-2410-H-110-015-MY2.

www.manaraa.com

REFERENCES

Akkiraju, R., Srivastava, B., Ivan, A. A., Goodwin, R., and Syeda-Mahmood, T. (2006). "SEMAPLAN: Combining planning with semantic matching to achieve Web service composition," *Proceedings of the IEEE International Conference on Web Services (ICWS'06)*.

Algergawy, A., Nayak, R., and Saake, G. (2010). "Element similarity measures in XML schema matching," *Information Sciences, 180*(24), 4975-4998.

Basu, A., and Blanning, R. W. (1998). "The Analysis of Assumptions in Model Bases Using Metagraphs," *Management Science, 44*(7), 982-995.

Basu, A., and Blanning, R. W. (2000). "A Formal Approach to Workflow Analysis," *Information Systems Research, 11*(1), 17-36.

Berardi, D., Calvanese, D., De Giacomo, G., Lenzerini, M., and Mecella, M. (2005). "Automatic service composition based on behavioral descriptions," *International Journal of Cooperative Information Systems, 14*(4), 333-376.

Bonczek, R. H., Holsapple, C. W., and Whinston, A. B. (1980). "The Evolving Roles of Models in Decision Support Systems," *Decision Sciences, 11*(2), 337-356.

Boukottaya, A., and Vanoirbeek, C. (2005). "Schema matching for transforming structured documents," *Proceedings of the 2005 ACM Symposium on Document Engineering*, Bristol, United Kingdom.

Caswell, N. S., Nikolaou, C., Sairamesh, J., Bitsaki, M., Koutras, G. D., and Iacovidis, G. (2008). "Estimating Value in Service Systems: A Case Study of a Repair Service System," *IBM Systems Journal, 47*(1), 87-100.

Chen, K., Xu, J., and Reiff-Marganiec, S. (2009). "Markov-HTN planning approach to enhance flexibility of automatic Web service composition," *Proceedings of the IEEE International Conference on Web Services (ICWS'09)*.

Christensen, E., Curbera, F., Meredith, G., and Weerawarana, S. (2001). "Web Services Description Language (WSDL) 1.1," *At* http://www.w3.org/TR/2001/NOTE-wsdl-20010315, on May 21, 2014.

Dolk, D. R., and Konsynski, B. (1984). "Knowledge Representation for Model Management Systems," *IEEE Transactions on Software Engineering, SE-10*(6), 619-628.

Doshi, P., Goodwin, R., Akkiraju, R., and Verma, K. (2004). "Dynamic workflow composition using Markov decision processes," *International Journal of Web Services Research, 2*(1), 576-582.

Fernandez, M., Malhotra, A., Marsh, J., Nagy, M., and Walsh, N. (2007). "XQuery 1.0 and XPath 2.0 Data Model," Retrieved from http://www.w3.org/TR/xpath-datamodel/, on 2009.01.01, 2009.

Fielding, R. (2000). *Architectural Styles and The Design of Network-based Software Architectures.* Ph. D., University of California, Irvine.

Frasincar, F., Houben, G. J., and Pau, C. (2002). "XAL: an algebra for XML query optimization," *Australian Computer Science Communications, 24*(2), 49-56.

Gerede, C. E., Hull, R., Ibarra, O. H., and Su, J. (2004). "Automated composition of

Published by AIS Electronic Library (AISeL), 2014
www.manaraa.com

e-services: lookaheads," *Proceedings of the the 2nd International Conference on Service Oriented Computing*.

Glushko, R. J., and Tabas, L. (2009). "Designing Service Systems by Bridging the “Front Stage” and “Back Stage”," *Information Systems and E-Business Management, 7*(4), 407-427.

Gu, Z., Li, J., and Xu, B. (2008). "Automatic service composition based on enhanced service dependency graph," *Proceedings of the IEEE International Conference on Web Services (ICWS'08)*.

Gudgin, M., Hadley, M., Mendelsohn, N., Moreau, J. J., and Nielsen, H. F. (2007). " SOAP Version 1.2 Part 1: Messaging Framework," Retrieved from http://www.w3.org/TR/2007/REC-soap12-part1-20070427/, on May 21, 2014.

Jagadish, H., Lakshmanan, L., Srivastava, D., and Thompson, K. (2001). "TAX: A tree algebra for XML," *Proceedings of the the 8th International Workshop on Database Programming Languages (DBPL'01)*.

Jordan, D., and Evdemon, J. (2007). "Business Process Execution Language for Web Services Version 2.0," Retrieved from http://docs.oasis-open.org/wsbpel/2.0/CS01/wsbpel-v2.0-CS01.pdf, on May 21, 2014.

Lecue, F., Salibi, S., Bron, P., and Moreau, A. (2008). "Semantic and syntactic data flow in Web service composition," *Proceedings of the IEEE International Conference on Web Services (ICWS'08)*.

Li, X., Madnick, S. E., and Zhu, H. (2013). "A Context-Based Approach to Reconciling Data Interpretation Conflicts in Web Services Composition," *ACM Transations on Internet Technology, 13*(1), 1-27.

Liang, Q. A., and Su, S. Y. W. (2005). "AND/OR graph and search algorithm for discovering composite Web services," *International Journal of Web Services Research, 2*(4), 48-67.

Liang, T.P. (1988). "Model Management for Group Decision Support," *MIS Quarterly*(December), 667-680.

Liang, T.P., and Jones, C. V. (1988). "Meta-Design Considerations in Developing Model Management Systems," *Decision Sciences, 19*(1), 72-92.

Lucchi, R., and Mazzara, M. (2007). "A Pi-calculus based semantics for WS-BPEL," *Journal of Logic and Algebraic Programming, 70*(1), 96-118.

Magnani, M., and Montesi, D. (2006). "A unified approach to structured and XML data modeling and manipulation," *Data & Knowledge Engineering, 59*(1), 25-62.

McDermott, D. (2002). "Estimated-regression planning for interactions with Web Services," *Proceedings of the the 6th International Conference on Artificial Intelligence Planning Systems*.

Mrissa, M., Ghedira, C., Benslimane, D., Maamar, Z., Rosenberg, F., and Dustdar, S. (2007). "A context-based mediation approach to compose semantic Web services," *ACM Transactions on Internet Technology, 8*(1), Article 4.

Nagarajan, M., Verma, K., Sheth, A. P., and Miller, J. A. (2007). "Ontology driven data mediation in web services," *International Journal of Web Services Research, 4*(4), 104-126.

https://aisel.aisnet.org/pajais/vol6/iss2/2
DOI: 10.17705/1pais.06202
www.manaraa.com

Narayanan, S., and McIlraith, S. A. (2002, 2002). "Simulation, verification and automated composition of Web services," *Proceedings of the the 11th International Conference on World Wide Web*.

OMG. (2013). "Business Process Model and Notation Version 2.0," Retrieved from http://www.omg.org/spec/BPMN/2.0.2/, on May 21, 2014.

Ouyang, C., Verbeek, E., Van Der Aalst, W. M. P., Breutel, S., Dumas, M., and Ter Hofstede, A. H. M. (2007). "Formal semantics andanalysis of control flow in WS-BPEL," *Science of Computer Programming, 67*(2-3), 162-198.

Paganelli, F., and Parlanti, D. (2013). "A Dynamic Composition and Stubless Invocation Approach for Information-Providing Services," *IEEE Transactions on Network and Service Management, 10*(2), 218-230.

Park, C.-S., and Park, S. (2008). "Efficient execution of composite Web services exchanging intensional data," *Information Sciences, 178*(2), 317-339. doi: 10.1016/j.ins.2007.08.021

Pistore, M., Traverso, P., and Bertoli, P. (2005). "Automated composition of Web services by planning in asynchronous domains," *Proceedings of the the 15 International Conference on Automated Planning and Scheduling*.

Sirin, E., Parsia, B., Wu, D., Hendler, J., and Nau, D. (2004). "HTN planning for Web service composition using SHOP2," *Web Semantics: Science, Services and Agents on the World Wide Web, 1*(4), 377-396.

Sloan, J. C., and Khoshgoftaar, T. M. (2009). "From Web service artifact to a readable and verifiable model," *IEEE Transactions on Services Computing, 2*(4), 277-288.

Spohrer, J., and Kwan, S. (2009). "Service Science, Management, Engineering, and Design: An Emerging Discipline--Outline and References," *International Journal of Information Systems in the Service Sector, 1*(3), 1-31.

Srivastava, U., Munagala, K., Widom, J., and Motwani, R. (2006). "Query optimization over web services," *Proceedings of the the 32nd International Conference on Very Large Data Bases*.

Tan, W., Fan, Y., and Zou, M. (2009). "A Petri net-based method for compatibility analysis and composition of Web services in Business Process Execution Language," *IEEE Transactions on Automation Science and Engineering, 6*(1), 94-106.

Thomas, E. (2007). *SOA principles of service design*: Prentice Hall PTR, Upper Saddle River, NJ.

Ullman, J. D. (1989). Query optimization for database systems *Principles of Database and Knowledge-Base Systems* (Vol. 2, pp. 633-725): Computer Science Press.

Xia, Y.-M., and Yang, Y.-B. (2013). "Web Service Composition Integrating QoS Optimization and Redundancy Removal," *Proceedings of the 2013 IEEE 20th International Conference on Web Services (ICWS'13)*.

Zeng, L., Ngu, A., Benatallah, B., Podorozhny, R., and Lei, H. (2008). "Dynamic composition and optimization of Web services," *Distributed and Parallel Databases, 24*(1), 45-72.

Zou, G., Gan, Y., Chen, Y., and Zhang, B. (2014). "Dynamic Composition of Web Services Using Efficient Planners in

Published by AIS Electronic Library (AISeL), 2014
www.manaraa.com

Large-Scale Service Repository," *Knowledge-Based Systems, 62*(0), 98-112.

About the Authors

Chien-Hsiang Lee is currently a technical consultant at Galaxy Software Services in Taiwan. He received the B.Sc. and MBA degrees from National Chiao Tung University in Taiwan and Ph. D. from the Department of Information Management at National Sun Yat-Sen University in Taiwan in 2012. His current research interests include Web services, service-oriented architecture, and service science.

San-Yih Hwang received the B.Sc. and M.Sc. degrees from National Taiwan University, Taiwan, and the Ph.D. degree from the University of Minnesota, Minneapolis in 1994, all in computer science. He joined the Department of Information Management at National Sun Yat-sen University, Taiwan, in 1995 and is presently a professor. His current research interests include services computing, workflow management, data mining, and recommendations.

https://aisel.aisnet.org/pajais/vol6/iss2/2
DOI: 10.17705/1pais.06202
www.manaraa.com