Execution Optimization for Composite Services Through Multiple Engines*

Wubin Li[1], Zhuofeng Zhao[1], Jun Fang[1], and Kun Chen[2]

[1] Research Centre for Grid and Service Computing
Institute of Computing Technology, Chinese Academy of Sciences
P.O.Box 2704, 100080, Beijing, China
[2] Department of Computer Science and Technology
Shandong University of Science and Technology, Qingdao 266510, China
{liwubin, zhaozf, fangjun, chenkun}@software.ict.ac.cn
http://sigsit.ict.ac.cn/

Abstract. Web services are rapidly emerging as a popular standard for sharing data and functionality among heterogeneous systems. We propose a general purpose Web Service Management System (*WSMSME*) that enables executing composite services through multiple engines. This paper tackles a first basic *WSMSME* problem: execution optimization for composite services through multiple engines. Our main result comprises two dynamic programming algorithms. One helps minimizes the number of engines required to complete a composite service when computational capability of each engine is relatively changeless; the other optimally minimizes the heaviest load of engines by segmenting a pipelined execution plan into sub-sequences before they are dispatched and executed; Both of the two can obtain optimal solutions in polynomial time. Experiments with an initial prototype indicate that our algorithms can lead to significant performance improvement over more straightforward techniques.

Keywords: Web Services, Execution Optimization, Multiple Engines, Dynamic Programming.

1 Introduction

Web services [1] are becoming a standard method of sharing data and functionality among loosely-couple, heterogeneous systems. Many organizations and enterprises are considering exposing their existing data and business logic as Web services (to both internal and external audiences). On the other hand, the composition of Web services to handle complex transactions such as finance, billing, and traffic information services is gaining considerable momentum as a way to enable business-to-business (B2B) collaborations. There has been a considerable amount of recent work [2, 3] on the challenges associated with discovering

* This work is supported in part by the National Science Foundation of China (Grant No. 90412010), the National Basic Research Program of China (973 Program) (Grant No. 2007CB310805), and the China R&D Infrastructure and Facility Development Project (Grant No. 2005DKA64201).

B. Krämer, K.-J. Lin, and P. Narasimhan (Eds.): ICSOC 2007, LNCS 4749, pp. 594–605, 2007.
© Springer-Verlag Berlin Heidelberg 2007

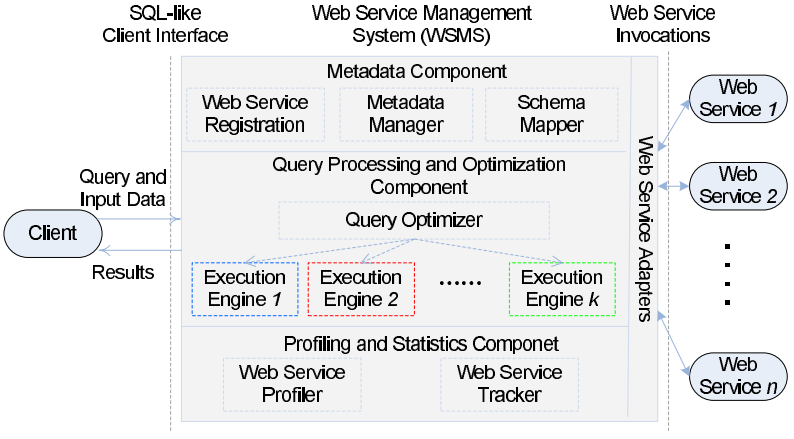


Fig. 1. A Web Service Management System with Multiple Engines (*WSMSME*)

and composing web services to solve a given problem. We are interested in the more basic challenge of providing scheduler-like capabilities when scheduled jobs are composite services. To this end we propose the development of a Web Service Management System with Multiple Engines *WSMSME*: a general-purpose system that enables clients to execute composite services simultaneously in a transparent and integrated fashion.

Overall, we expect a *WSMSME* to consist of three major components; see Figure 1. The *Metadata* component deals with metadata management, registration of new web services, and mapping their schemas to an integrated view provided to the client. Given an integrated view of the schema, a client can request the *WSMSME* through a client interface. The *Execution Processing and Optimization* component handles optimization and execution of such declarative request, i.e., it chooses and executes a plan whose operators invoke the relevant composite service which comprise several web services. The *Profiling and Statistics* component profiles web services for their response time characteristics, spatial cost; and maintains relevant statistics over the web service data, to the extent possible. This component is used primarily by the execution optimizer for making its optimization decisions.

What make *WSMSME* different from other congeners [13] are the number of execution engines and the locations of them. We argue that, multiple execution engines and distribution could be exploited to achieve parallelism in execution and reduce the response time to the user. One can expect a reasonable speedup and cost-like load balancing because of the following reasons:

- A single Engine might be incapable to finish the whole process especially when computational capability and resources are insufficient in a separate machine.
- Architecture with multiple engines distributed in different systems is usually a preferable solution. Distributing the query makes more computational power available for the execution of the composite service.

- Cutting the composite service into execution segments could achieve parallelism and spatial cost balance over multiple engines.
- Multiple Engines could provide reliability guarantee in a certain degree. Composite services could be executed in *K-1* engines when one of the *K* engines is crashed.
- A large number of third-party businesses make money out of service execution. They either charge money on per execution basis (micro money) or through advertising. Such businesses, very likely, would run an execution engine on their machine and make it available to users to send requests to. They would either charge money for each execution or embed advertising in the result XML documents. E.g., if a user wants a search on all sites that keep the old car sales data, the only way a query engine can execute this query is by distributing sub queries to each of these sites [13].

Moreover, our *WSMSME* architecture is similar to mediators in distributed data integration system [16, 17, 18, 19, 20];

2 Related Work

2.1 Parallel and Distributed Execution Processing

In our setting of execution processing over web services, only data shipping is allowed, i.e., dispatching data to web services that process it according to their preset functionality. In traditional distributed or parallel execution processing, each of which has been addressed extensively in previous work [4,5,6], in addition to data shipping, code shipping also is allowed, i.e., deciding which machines are to execute which code over which data. Due to lack of code shipping, techniques for parallel and distributed execution optimization, e.g., fragment-replicate joins [6], are inapplicable in our scenario. Moreover, most parallel or distributed execution optimization techniques are limited to a heuristic exploration of the search space whereas we provide provably optimal plans for our problem setting.

2.2 Web Service Composition and Choreography

A considerable body of recent work addresses the problem of composition (or orchestration) of multiple web services to carry out a particular task, e.g. [7, 8]. In general, that work is targeted more toward workflow-oriented applications (e.g., the processing steps involved in fulfilling a purchase order), rather than applications coordinating execution optimization through multiple engines, as addressed in this paper. Although these approaches have recognized the benefits of pipelined processing, they have not, as far as we are aware, included formal cost models or techniques that result in provably optimal pipelined execution strategies.

Languages such as *BPEL4WS* [9] are emerging for specifying web service composition in workflow-oriented scenarios. While we have not yet specifically applied our work to these languages, we note that *BPEL4WS*, for example, has

constructs that can specify which web services must be executed in a sequence and which can be executed in parallel. But there is no consideration about running environments, nor optimal execution plans - no specification relative about how to complete the executions through multiple engines. We are hopeful that the optimization techniques developed here will extend to web-service workflow scenarios as they become more standardized, and doing so is an important direction for future work.

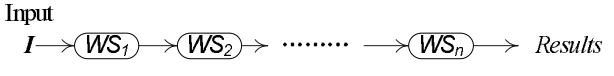
3 Preliminaries

Consider a *WSMSME* as shown in *Figure 1* that provides an integrated interface to invoke a composite service which involves n web services WS_1,...WS_n. We assume that each web service possesses a property (referred to as C) which represents how much cost that is required to finish executing it. C might include time, memory sizes, money, etc. consequently, we write WS_i (C_i) to denote that, treating WS_i as a program whose running cost is C_i. An important direction of future work is to provide more sophisticated mechanisms to describe those requirements, because every-way, the notion of a single cost factor C is overly a little bit simplistic.

3.1 Composite Pattern Considered

The composite patterns of service we consider for optimization are sequential services over one or more web services WS_1,...WS_n. We assume that the correspondence among various inputs of services is tracked by the Metadata component of the *WSMSME* (*Figure 1*).

DEFINETION 3.1 (SEQUENTIAL SERVICES).



Where inputs of WS_{i+1} come from the outputs of WS_i ,where $i \in$ [1,2...n-1].

We assume in *Definition 3.1* that all web services run in order: In a given process, WS_{i+1} can not be executed before WS_i has been executed. We also assume that inputs of each web service can be delivered accurately without any other consideration about how it can be.

EXAMPLE 3.1. *Suppose a credit card company wishes to send out mailings for its new credit card offer. The company continuously obtains lists of potential recipients from which it wants to select only those who have a good payment history on a prior credit card, and who have a credit rating above some threshold. For processing this query, the company has the following three web services at its disposal.*

WS_1 : *name (n)* $\rightarrow$ *credit rating (cr)*
WS_2 : *name (n)* $\rightarrow$ *credit card numbers (ccn)*
WS_3 : *card number (ccn)* $\rightarrow$ *payment history (ph)*
With a WSMSME, one possible way of executing the query is as follows: The Company's initial list of names (we assume names are unique) is first processed by WS_1 *to determine the corresponding credit ratings, and those below threshold are filtered out (either by* WS_1 *itself or by the WSMSME). The remaining names are then processed by* WS_2 *to get the corresponding credit card numbers. Each card number is then processed by* WS_3*, and if the card is found to have a good payment history, then the name is output in the result of the query, as below.*

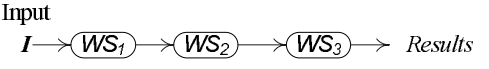


Fig. 2. Plan for Example 3.1

Other patterns mentioned in previous works [12] are left out of in this paper, and would be kept for the future work.

3.2 Problem Definition

There are two basic scenarios involved in this paper, and we try to solve problems related with those scenarios. Such problems are all about execution optimization through multiple execution engines in *WSMSME*.

3.2.1 Scenario A: Minimize the Number of Execution Engines

In this scenario, we suppose that there are K execution engines available in *WSMSME*, and client requests arrive in chunks. Web service WS_i of a composite service requires C_i*-weighed* resources so that it can be executed correctly. In order to keep the number of available execution engines as great as possible, we try to minimize the number of execution engines required when a request of a composite service arrives, then allocate the chosen engines to the composite service being invoked. Suppose there are 8 execution engines available when a request of composite service *CS* comes, we can allocate 5 engines to complete this request; but if 2 is ok, then we choose 2 and thereby 6 is left before others become available again. Currently, we also simply assume that the computational capability of each engine is equivalent relatively, and we write a value L to denote this.

EXAMPLE 3.2.1. *Consider the plan in Figure 2. Let the requirement of the web services and L to be as follows:*

i	*1*	*2*	*3*
Requirement of WS_i (C_i)	*4*	*2*	*3*
Value of L	*6*		

Then if we dispatch each service to a different execution engine (it is possible because L is greater than C_i), 3 execution engines are needed. Obviously, it is not the optimal solution. At least, we have two more plans for this problem: (a) allocate an execution engine to run WS_1 and WS_2, and another for WS_3; (b) allocate an execution engine to run WS_1 and another for WS_2 and WS_3.

Now we can not tell which is better, because plan a and b both need 2 execution engines. To evaluate how a plan is different from another that needs the same number of execution engines, we introduce a dissatisfaction index function(referred to as *DI*) to calculate how bad the plan is. Suppose two different plans both need M engines (referred to as EE_1, EE_2,...EE_M), then:

$$Dissatisfaction\ Index = \sqrt{\frac{\sum_{i=1}^{M}(L - \sum_{EE_i\ is\ allocated\ to\ WS_k} C_k)^2}{M}}$$

In this scenario, we treat C_i as a random variable, whose dissatisfaction is the standard deviation. The standard deviation is the root mean square (RMS) deviation of values from their arithmetic mean, and it is most common measure of statistical dispersion, measuring how widely spread the values in the data set $\{C_1,C_2,...\}$ are. The less that standard deviation is, the better the solution is. Concretely, if dissatisfaction index is large, that means many C_i are far from the mean, and correspondingly,vast sum of resources of engines are wasted.

We can now calculate which is better in Example 3.2.1, because $DI(a) = \sqrt{\frac{(6-4-2)^2+(6-3)^2}{2}} = \frac{3\sqrt{2}}{2}, DI(b) = \sqrt{\frac{(6-4)^2+(6-2-3)^2}{2}} = \frac{\sqrt{10}}{2}$.

Apparently, *Dissatisfaction Index(a)>Dissatisfaction Index(b)* which tells that plan b is more optimal than plan b, then we choose the preferable one.

3.2.2 Scenario B: Load Balancing Through Multiple Engines

In this scenario, which is independent with scenario A, we try to dispatch K web services (which comprise a composite service) to M available engines and obtain load balancing over those M engines.

EXAMPLE 3.2.2. *Consider the plan in Figure 2. Let the requirement of the web services and L to be as follows:*

i	*1*	*2*	*3*
Requirement of WS_i (C_i)	*4*	*2*	*3*
Value of L	*7*		
Number of Engines Allocated	*2*		

And now we have two plans for this problem: (a) allocate an execution engine M_1 to run WS_1 and WS_2, and another M_2 for WS_3; (b) allocate an execution engine M_1 to run WS_1 and another M_2 for WS_2 and WS_3. In this scenario, we can easily tell that plan (a) is better, because the maximum load among M_1 and M_2 is 6 (the sum of C_1 and C_2), which is less than that of plan (b). In other

words, we redefine the dissatisfaction index function(calculate the maximum load of engines) in this scenario to be as follows:

$$Dissatisfaction\ Index = \max_{M} \left\{ \sum_{EE_i\ is\ allocated\ to\ WS_k} C_k \middle| i \in [1, 2...M] \right\}$$

For EXAMPLE 3.2.2, we get

$$DI(a) = \max\{(2+4), 3\} = 6, DI(b) = \max\{2, (4+3)\} = 7$$

Which shows that plan (a) is preferable.

4 Algorithms for Execution Plans in the Two Scenarios

4.1 Optimal Execution Plans for Scenario A

To solve the problem described in scenario A, we use the optimal substructure to show that we can construct an optimal solution from optimal solutions to subproblems. Firstly, we make two denotations as follows:

- $F(k)$:The minimum number of execution engines to complete the first k web services in a composite service.
- $D(k)$:The value of dissatisfaction index when the first k web services are optimally scheduled.

Thus, the minimum number of execution engines to complete the first *k+1* web services $F(k+1)$ is either of

1. $F(k)+1$, that is to say, allocate a new engine to the web services WS_{k+1}.
2. $F(j)+1$,when allocating services WS_{j+1}, WS_{j+2},...WS_{k+1} the same engine.

Using this we get:

$$F(k+1) = Min\{F(k)+1, F(j)+1 \big| j < k+1 \& \textstyle\sum_{i=j+1}^{k+1} C_i \le L\}$$

Moreover, we can update $D(k)$ when computing $F(k)$.

Algorithm Dynamic Programming A
1. $F(1) \leftarrow 1, D(1) \leftarrow \sqrt{(L-C_1)^2}, k \leftarrow 1, T \leftarrow number\ of\ webservices$
2. $While(k+1<T)$
3. $F(k+1) \leftarrow \infty, D(k+1) \leftarrow \infty$
4. $While(j<k+1 \& \sum_{i=j+1}^{k+1} C_i \le L)$
5. $if(F(k+1)>F(j)+1)$
6. $F(k+1) \leftarrow F(j)+1$
7. $Re{-}calculate\ D(k+1)\ using\ Dissatisfaction\ Index\ Function.$
8. $elseif(F(k+1) = F(j)+1)$
9. $Re{-}calculate\ D(k+1)\ using\ Dissatisfaction\ Index\ Function.$
10. $Return F(T), D(T).$

4.2 Optimal Execution Plans for Scenario B

Conditions involved in scenario B are different from that of scenario A, but the solution is similar. And we still using dynamic programming algorithms to solve that problem. We write $F(i, j)$ to denote the maximum load among j engines when executing i web services through those j engines. Thus, considering the definition of F, we get

$$F(i,j) = Min\Big\{ Max\Big\{ F(i-1, t-1), \sum_{k=t}^{i} C_k \Big\} \Big| t \geq i \Big\}.$$

Using the equation above, we get algorithm as follows:

Algorithm Dynamic Programming B
1. $k \leftarrow$ *number of available execution engines*, $n \leftarrow$ *number of web services*
2. $for(i = 1; i \leq n; i++)$
3. $\quad F(1, i) = SUM(C_1, C_2 ... C_i)$
4. $for(i = 2; i \leq k; i++)$
5. $\quad for(j = i; j < n; j++)$
6. $\quad\quad for(t = j; t \geq i; t--)$
7. $\quad\quad\quad tem = Max\{F(i, t-1), \sum_{k=t}^{i} C_k\}$
8. $\quad\quad\quad if(tem < F(i, j))$
9. $\quad\quad\quad\quad F(i, j) \leftarrow tem$
10. $Return F(k, n)$.

4.3 Analysis of Algorithms

See the algorithms above, algorithm A computes an optimal plan in $O(n^2)$ time where n is the number of web services involved in the composite service which is invoked; algorithm B computes an optimal plan in $O(kn^2)$ time where n is the number of web services and k is the number of engines.

5 Implementation and Experiments

We implemented an initial prototype *WSMSME*, described in*Section 5.1*. Here we report on a few experiments with it. Not surprisingly, in our experiments, plan execution performance of composite service reflects our theoretical results (thereby validating our cost model). Using minimum number of execution engines on demand and maximum load among execution engines as metrics, we compared the execution plan produced by our optimization algorithm (referred to as Optimizer) against the plans produced by the following simpler algorithms:

1. *Greedy*: This algorithm attempts to exploit the minimum possible number of execution engines by dispatching services to execution engine whenever possible. For example, if a subsequent service WS_i is supposed to require 40M memories, and the available computational capability of execution engine

EE_k is 46M (which is greater than 40M), then WS_i would be dispatched to execution engine EE_k, and that would decrease the available computational capability of EE_k to be 6M. Thereby, if the following service WS_{i+1} requires more than 6M to complete, an additional execution engine is needed. Greedy is used to finish a comparison with our algorithm in the first scenario (scenario A) that mentioned before.

2. *Random*: Segmenting a composite service into K sub-sequences can be obtained randomly. Random is used to finish a comparison with our algorithm in the second scenario (scenario B) that mentioned.

We first describe our *WSMSME* prototype and the experimental setup in *Section 5.1.* We then describe our experiments for scenarios mentioned before.

5.1 Prototype and Experimental Setup

The experimental setup consists of two parts: the client side, consisting of our *WSMSME* prototype, and the server side, consisting of web services set up by us.

Our *WSMSME* prototype is a multithreaded system written in Java [14]. It implements the two core dynamic programming algorithms we proposed in this paper. For communicating with web services using SOAP, our prototype uses Codehaus XFire [11] tools. Given a description of a web service in the Web Service Definition Language [15], Xfire generates a class such that the web service can be invoked simply by calling a method of the generated class. The input and out types of the web service are also encapsulated in generated classes. The function of executing a web service is realized inside Execution Engines. Execution engine here is a little bit "virtual" within our prototype, and it is implemented to be a common multi-threaded object (*ExecutionEngine*) which possesses of one special property (*ComputationalCapability*) that specifies its computational capability, which means we can create and delete an engine that has specific computational capability on demand.

We use Apache Tomcat [10] as the application server and Codehaus XFire [11] tools for web service deployment. Each of our experimental web services WS_i runs on a different machine, and has a table T_i (int a, int b, primary key a) associated with it. Given a value for attribute a, WS_i retrieves the corresponding value for attribute b from T_i (by issuing a SQL query) and returns it. The tables T_i are stored using the lightweight MySQL DBMS. Since attribute a is the primary key, MySQL automatically builds an index on a.

For our experiments, we needed web services with different costs and requirements. To obtain different costs, we introduce a delay between when a web service obtains the answer from its database and when it returns the answer to the caller of the web service. The web service cost is varied by varying this delay. The *WSMSME* is run on a different machine from the ones on which the web services were running. Each composite service is compromised by a series of web services sequentially (other types of composite patterns are left out of our discussion currently).

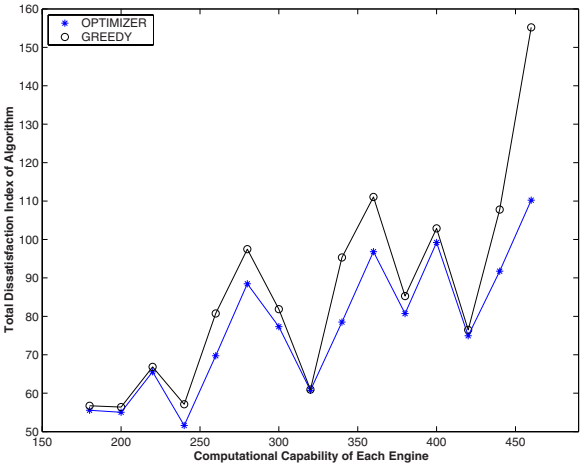


Fig. 3. Comparison of Total Dissatisfaction Index in Scenario A

5.2 Scenario A: Minimize the Number of Engines

In this experiment, given a composite service comprised by a sequence of web services with certain costs, we try to minimize the number of engines required to complete a composite service when computational capability of each engine is relatively changeless and equivalent. Furthermore, if there are multiple schedules with the minimum number of engines, we also minimize the Total *Dissatisfaction Index*. We developed 19 web services, and make them a composite service. Costs of the web service ranges from 50 to 180. When running this composite service, we dynamically increased the computational capability of engine from 170 to 460. Surprisingly, we found that the minimum possible number of engines required is nearly the same when using *OPTIMIZER* and *GREEDY*, while the *Total Dissatisfaction Index* is completely different as *Figure 3* shows.

Figure 3 shows that, the advantage of *OPTIMIZER* mounts up as the computational capability of each engine (referred to as *cmp*) increases. Only when the *cmp* is small do the *GREEDY* obtain the similar schedule plan as *OPTIMIZER*.

5.3 Scenario B: Minimize the Heaviest Load of Engines

In this experiment (independent with the last one), we try to minimize the heaviest load of engines by segmenting a pipelined execution plan into subsequences before they are dispatched and executed. Namely, it is an experiment about load balancing among multiple engines. We use the services we've mentioned in Section 5.2, and increased the number of execution engines from 1 to 15.

See the performances produced . Not surprisingly, the maximum load among execution engines descends as the number of engines increase, no matter which algorithms were applied. Nevertheless, results obtained from *OPTIMIZER* were always more excellent than that from *RANDOM*.

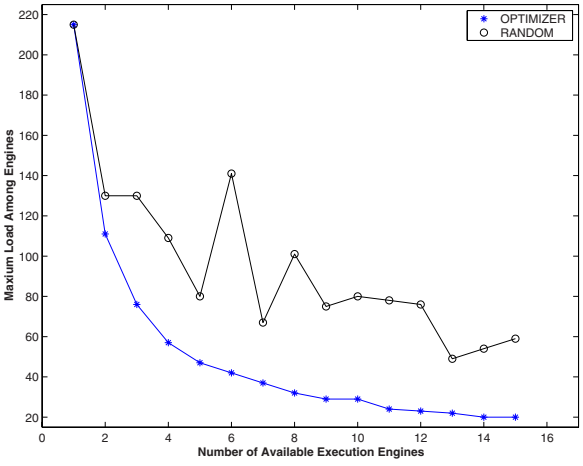


Fig. 4. Maximum Load among Engines in Scenario B

6 Conclusions and Future Works

Web services have received significant attention and there is a great deal of industry excitement around the opportunities afforded by them. While most of this attention has focused on middle-tier Web services, an increasing interest in composite service has recently emerged as organizations encapsulate their existing legacy services to be web services. In this paper, we focus on execution optimization issues that arise in a *WSMSME*. Towards two basis scenarios, we've devised new algorithms to: (a) help minimize the number of engines required to execute a composite service when computational capability of each engine is relatively changeless and (b) optimally minimizes the heaviest load of engines (namely load balancing) by segmenting a pipelined execution plan into subsequences before they are dispatched and executed. While the algorithms in this paper form the basis of a *WSMSME* execution optimizer, we believe they only scratch the surface of what promises to be an exciting new research area. There are several interesting directions for future work:

- An important next step is to extend our algorithms to support composite services which are comprised in more complicated patterns such as those mentioned in [12].
- We have not considered web services with monetary costs or other special types of costs. In those scenarios, we may wish to use optimization algorithms that minimize the running of a composite service to a certain budget limit. Moreover, it is also interesting to achieve load balancing when costs of web services are monetary or else.
- Our algorithms currently do not take much consideration about the metric of each execution engine's computational capability. We just simply expect

that it is equivalent the same. However, as the case stands, computational capability of executions might obviously vary.
– More work on service composite languages such as *BPEL4WS* is needed; Extension of our algorithms to such specifications is an interesting direction of future work.

References

1. Web services (2002), http://www.w3.org/2002/ws
2. Florescu, D., Grunhagen, A., Kossmann, D.: XL: A platform for web services. In: CIDR. Proc. First Biennial Conf. on Innovative Data Systems Research (2003)
3. Ouzzani, M., Bouguettaya, A.: Efficient access to web services. IEEE Internet Computing 8(2), 34–44 (2004)
4. DeWitt, D., et al.: The Gamma Database Machine Project. IEEE Trans. on Knowledge and Data Engineering 2(1), 44–62 (1990)
5. Hong, W., Stonebraker, M.: Optimization of parallel query execution plans in XPRS. In: Proceedings of the First Intl.Conf. on Parallel and Distributed Information Systems, pp. 218–225 (1991)
6. Ozsu, M., Valduriez, P.: Principles of distributed database systems. Prentice-Hall, Inc, Englewood Cliffs (1991)
7. Florescu, D., Grunhagen, A., Kossmann, D.: XL: A platform for web services. In: CIDR. Proc. First Biennial Conf. on Innovative Data Systems Research (2003)
8. Ouzzani, M., Bouguettaya, A.: Efficient access to web services. IEEE Internet Computing 8(2), 34–44 (2004)
9. BPEL4WS: Business Process Execution Language for Web Services, ftp://www6.software.ibm.com/software/developer/library/wsbpel.pdf
10. Apache Tomcat, http://tomcat.apache.org/
11. Codehaus XFire, http://xfire.codehaus.org/
12. Russell, N., ter Hofstede, A.H.M.: WORKFLOW CONTROL-FLOW PATTERNS-A Revised View, http://workflowpatterns.com/documentation/documents/BPM-06-22.pdf
13. Srivastava, U., Munagala, K., Widom, J., Motwani, R.: Query optimization over web services. In: Proceedings of the 32nd international conference on Very large data bases, vol. 32, pp. 355–366 (2006)
14. Java API, http://java.sun.com/j2se/1.5.0/docs/api/
15. Web Services Description Language, http://www.w3.org/TR/wsdl
16. Casati, F., Dayal, U. (eds.): Special Issue on Web Services, IEEE Data Eng. Bull., vol. 25(4) (2002)
17. Garcia-Molina, H., et al.: The TSIMMIS approach to mediation: Data models and languages. Journal of Intelligent Information Systems 8(2), 117–132 (1997)
18. Miller, R. (ed.): Special Issue on Integration Management, IEEE Data Eng. Bull., vol. 25(3) (2002)
19. Roth, M., Schwarz, P.: Don't Scrap It, Wrap It! A Wrapper Architecture for Legacy Data Sources. In: Proc. of the 1997 Intl. Conf. on Very Large Data Bases, pp. 266–275 (1997)
20. Viglas, S., Naughton, J.F., Burger, J.: Maximizing the output rate of multi-join queries over streaming information sources. In: Proc. of the 2003 Intl. Conf. on Very Large Data Bases, pp. 285–296 (2003)