Query Optimization over Web Services[*]

Utkarsh Srivastava[1] Kamesh Munagala[2] Jennifer Widom[1] Rajeev Motwani[1]

[1] Stanford University
{usriv, widom, rajeev}@cs.stanford.edu

[2] Duke University
kamesh@cs.duke.edu

ABSTRACT

Web services are becoming a standard method of sharing data and functionality among loosely-coupled systems. We propose a general-purpose Web Service Management System (WSMS) that enables querying multiple web services in a transparent and integrated fashion. This paper tackles a first basic WSMS problem: query optimization for Select-Project-Join queries spanning multiple web services. Our main result is an algorithm for arranging a query's web service calls into a pipelined execution plan that optimally exploits parallelism among web services to minimize the query's total running time. Surprisingly, the optimal plan can be found in polynomial time even in the presence of arbitrary precedence constraints among web services, in contrast to traditional query optimization where the analogous problem is NP-hard. We also give an algorithm for determining the optimal granularity of data "chunks" to be used for each web service call. Experiments with an initial prototype indicate that our algorithms can lead to significant performance improvement over more straightforward techniques.

1. INTRODUCTION

Web services [33] are rapidly emerging as a popular standard for sharing data and functionality among loosely-coupled, heterogeneous systems. Many enterprises are moving towards a *service-oriented architecture* by putting their databases behind web services, thereby providing a well-documented, interoperable method of interacting with their data. Furthermore, data not stored in traditional databases also is being made available via web services. There has been a considerable amount of recent work [12, 24] on the challenges associated with discovering and composing web services to solve a given problem. We are interested in the more basic challenge of providing DBMS-like capabilities when data sources are web services. To this end we propose the development of a *Web Service Management System* (*WSMS*): a general-purpose system that enables clients to query multiple web services simultaneously in a transparent and integrated fashion.

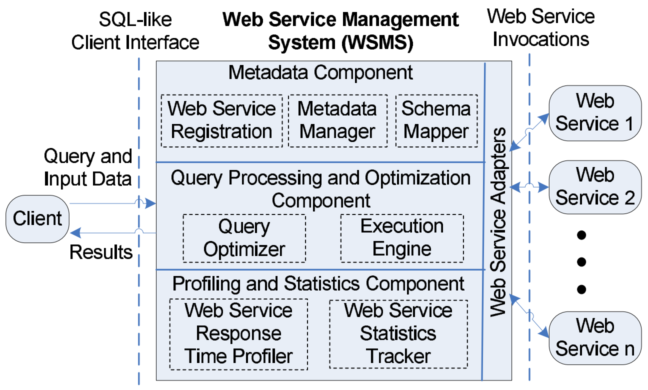

Figure 1: A Web Service Management System (WSMS)

Overall, we expect a WSMS 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. There is a large body of work on data integration, see e.g, [7, 22], that applies to the Metadata component; we do not focus on these problems in this paper. Given an integrated view of the schema, a client can query the WSMS through an SQL-like interface. The *Query Processing and Optimization* component handles optimization and execution of such declarative queries, i.e., it chooses and executes a query plan whose operators invoke the relevant web services. The *Profiling and Statistics* component profiles web services for their response time characteristics, and maintains relevant statistics over the web service data, to the extent possible. This component is used primarily by the query optimizer for making its optimization decisions. In this paper we take a first step at realizing a complete WSMS: We address the problem of query optimization for Select-Project-Join queries spanning multiple web services.

Most web services provide a function-call like interface $\mathcal{X} \rightarrow \mathcal{Y}$ where $\mathcal{X}$ and $\mathcal{Y}$ are sets of attributes: given values for the attributes in $\mathcal{X}$, the web service returns values for the attributes in $\mathcal{Y}$. For example, a web service may take a credit card number and return the card's credit limit. Due to this very restricted interface, most query processing over web services can be thought of in terms of a "workflow" or pipeline: some input data is fed to the WSMS, and the WSMS processes this data through a sequence of web services. The output of one web service is returned to the WSMS and then serves as input to the next web service in the pipeline, finally producing the query results. Each web service in the pipeline typically performs operations such as filtering out data items that are

[*]This work was supported by the National Science Foundation under grants IIS-0324431, EIA-0137761, and ITR-0331640, by grants from Media-X and SNRC at Stanford, by a Microsoft Graduate Fellowship, and by a Stanford Graduate Fellowship from Sequoia Capital.

not relevant to the query, transforming data items, or appending additional information to each data item. Transformed or augmented data items may be required for further processing of the query (effectively performing a join across web services), or may become a part of the final query result.

EXAMPLE 1.1. *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 WSMS, 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 WSMS). 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. □

The first obvious step to speed up query execution in a WSMS is to use the conventional idea of *pipelined parallelism*: data already processed by web service WS_i may be processed by a subsequent web service WS_{i+1} in the pipeline, at the same time as WS_i processes new data. Deciding the optimal way to perform this pipelining poses several new challenges:

1. Different web services may differ widely in their response time characteristics, as well as in how many output tuples they produce per input tuple on average (henceforth *selectivity*). Hence different arrangements of the web services in the pipeline may result in significantly different overall processing rates. The optimizer must decide the best arrangement.
2. The web services in the pipeline may not always be freely reordered, i.e., there might exist *precedence constraints*. (In Example 1.1, WS_2 must occur before WS_3 in the pipeline.) In such cases, the optimizer must pick the best arrangement that respects all precedence constraints.
3. A linear ordering of the web services in a pipeline (as in Example 1.1) may not be optimal. For example, if there are no precedence constraints between web services WS_i and WS_j, we need not wait for results from one to invoke the other, rather they may be invoked in parallel using the same input data. On the other hand, parallelizing all web services without precedence constraints may not be optimal either, since one or more of the web services may vastly reduce the amount of data the others need to process.
4. Each web service call usually has some fixed overhead, typically parsing SOAP/XML headers and going through the network stack. Hence some web services support sending data to them in "chunks" rather than one tuple at a time. Through experiments we found that the response time of a web service often is not linear in the input chunk size, so the optimizer must decide the best chunk size to use.

In this paper, we develop new, efficient algorithms that address each of the above challenges to arrive at the optimal pipelined execution plan for a given query over a set of web services. A simple yet significant observation that forms the basis for our algorithms is that the performance of a pipelined plan over web services (the rate of data processing through the plan) is dictated by the slowest web service in the pipeline (referred to as the *bottleneck cost metric*). In contrast, in a traditional centralized system, the cost of a pipelined plan is dictated by the sum of the costs of the plan operators (referred to as the *sum cost metric*) rather than by the cost of only the slowest operator. Previous related work [3, 8, 16, 18] has considered only the sum cost metric. To the best of our knowledge, our work is the first to consider the bottleneck metric.

We start by considering web services without precedence constraints and give a simple algorithm to find the optimal plan based on the web service response times and selectivities. Our algorithm reveals the somewhat counterintuitive property that when the selectivity of all web services is ≤ 1, the optimal arrangement depends only on the response times of the web services and is independent of their selectivities.

Next we give a polynomial-time algorithm to find the optimal plan when there may be arbitrary precedence constraints among the web services. It is surprising that such an algorithm exists, since under the sum cost metric, it is known that the optimal plan is poly-time computable only for restricted types of precedence constraints [18], and for arbitrary precedence constraints the optimal plan is hard to even approximate [6].

Finally, we consider sending data to web services in chunks. We show that our query optimization algorithm extends trivially to account for chunking. We also give an algorithm to determine the best chunk size to use for each web service. The algorithm is based on profiling the web services to determine their response times as a function of the size of the data chunk sent to them.

Since at first glance our work might seem closely related to much existing literature, we discuss related work next, in Section 2. We then present the main contributions of this paper:

- We formally define the class of queries we consider, introduce the model for query processing in a WSMS, and formalize the bottleneck cost metric that is used to compare query plans (Section 3).
- We give algorithms to decide the best arrangement of web services into a pipelined plan so that the overall processing rate is maximized, both in the absence of precedence constraints (Section 4), and in the presence of arbitrary precedence constraints (Section 5).
- We consider the case when data can be sent to web services in chunks, and we give an algorithm to decide the optimal data chunk size for each web service in a query plan (Section 6).
- We have implemented an initial prototype WSMS query optimizer (with simple instantiations of the other WSMS components in Figure 1), and we report an experimental evaluation of our algorithms (Section 7).

2. RELATED WORK

2.1 Web Service Composition

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. [12, 24]. 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 data obtained from multiple web services via SQL-like queries, 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* [4] 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, similar to the presence and absence of precedence constraints in our model. 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.

ActiveXML is a paradigm in which XML documents can have embedded web service calls in them. However, optimization work on ActiveXML [1] mostly focusses on deciding which web service calls in the document need to be made in order to answer a query posed over the XML document. As ActiveXML gains acceptance, it can be seen as an interesting mechanism to set up a distributed query plan over web services: ActiveXML fragments might be input to a web service, thereby making it invoke other web services.

2.2 Parallel and Distributed Query Processing

In our setting of query 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 query processing, each of which has been addressed extensively in previous work [10, 17, 26], 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 query optimization, e.g., fragment-replicate joins [26], are inapplicable in our scenario. Moreover, most parallel or distributed query optimization techniques are limited to a heuristic exploration of the search space whereas we provide provably optimal plans for our problem setting.

2.3 Data Integration and Mediators

Our WSMS architecture has some similarity to *mediators* in data integration systems [7, 14, 22, 28]. However, query optimization techniques for mediators, e.g., [13, 25, 35], focus mostly on issues such as choosing the right *binding access pattern* to access each data source, and aim at minimizing the *total* consumption of resources rather than at minimizing running time by exploiting parallelism. An important focus in mediators is to optimize the cost incurred at the data integration system itself, for which classical relational database optimization techniques (or modifications thereof) often can be applied. However, our techniques focus not on optimizing the processing at the WSMS, but on optimizing the expensive web service calls by exploiting parallelism among them.

A capability frequently required in a WSMS is that of using the results from one web service to query another. This operation is essentially the same as the *Dependent Join*, which has been studied in [13, 21], and whose techniques are applicable in a WSMS.

2.4 Query Processing over Remote Sources

Exploiting parallelism among data sources has generally not been the focus of prior work. WSQ/DSQ [15] does exploit parallelism by making multiple asynchronous calls to web sources, but does not perform any cost-based optimization. Other work [19, 31] has considered adaptive query processing over remote data sources, with dynamic reoptimization when source characteristics change over time, but does not include optimizations to exploit parallelism among sources.

Our execution model of pipelined processing resembles *distributed Eddies* [29]. However, unlike our work, the Eddies framework does not perform static optimization of queries. A problem mathematically similar to ours has been considered in [9, 20], but only for the simpler case of no precedence constraints and all web services being selective, i.e., returning fewer data items than are input to it. Interestingly, in distributed Eddies, as well as in [9, 20], different input tuples may follow different plans, a possibility that we have not considered in our work. So far, we have focused on the problem of finding the optimal single plan for all tuples. An important direction of future work is to combine our techniques with those developed in [9, 20], thereby leading to even higher performance.

3. PRELIMINARIES

Consider a WSMS as shown in Figure 1 that provides an integrated query interface to n web services WS$_1, \ldots,$WS$_n$. We assume that for querying, each web service WS$_i$ provides a function-call like interface $\mathcal{X}_i \rightarrow \mathcal{Y}_i$, i.e., given values for attributes in $\mathcal{X}_i$, the web service returns values for the attributes in $\mathcal{Y}_i$. Using the notation of *binding patterns* [13], we write WS$_i(\mathcal{X}_i^b, \mathcal{Y}_i^f)$ to denote that, treating WS$_i$ as a virtual table, the values of attributes in $\mathcal{X}_i$ must be specified (or *bound*) while the values of attributes in $\mathcal{Y}_i$ are retrieved (or *free*).

Let $\bar{x}$ and $\bar{y}$ denote value assignments to the attributes in $\mathcal{X}_i$ and $\mathcal{Y}_i$ respectively. Logically, virtual table WS$_i$ has a tuple $(\bar{x}, \bar{y})$ whenever the value assignment $\bar{y}$ is among those returned for $\mathcal{Y}_i$ when WS$_i$ is invoked with $\mathcal{X}_i = \bar{x}$. There may be zero, one, or many tuples in WS$_i$ for each possible $\bar{x}$. Note that if $\mathcal{Y}_i$ is empty, then web service WS$_i$ acts as a filter, and virtual table WS$_i$ contains a tuple $\bar{x}$ for every value assignment $\bar{x}$ passing the filter.

3.1 Class of Queries Considered

The class of queries we consider for optimization are Select-Project-Join (SPJ) queries over one or more web services WS$_1, \ldots,$ WS$_n$, and a table I corresponding to data input by the client to the WSMS (e.g., the initial set of names in Example 1.1). We assume that the correspondence among various attributes of various web services, required for joins, is tracked by the Metadata component of the WSMS (Figure 1).

DEFINITION 3.1 (SPJ QUERIES OVER WEB SERVICES).

$$\begin{array}{ll} \textit{SELECT} & \mathcal{A}_S \\ \textit{FROM} & I(\mathcal{A}_I) \bowtie \textit{WS}_1(\mathcal{X}_1^b, \mathcal{Y}_1^f) \bowtie \ldots \bowtie \textit{WS}_n(\mathcal{X}_n^b, \mathcal{Y}_n^f) \\ \textit{WHERE} & P_1(A_1) \wedge P_2(A_2) \wedge \ldots \wedge P_m(A_m) \end{array}$$

where $\mathcal{A}_s$ is the set of projected attributes, $\mathcal{A}_I$ is the set of attributes in the input data, and $P_1, \ldots, P_m$ are predicates applied on attributes $A_1, \ldots, A_m$ respectively. □

We assume in Definition 3.1 that all predicates are on single attributes, i.e., there are no join conditions except implicit natural equijoins. We also assume there is only one source of information for each attribute: each attribute is either specified in the input data, or is obtained as a free attribute from exactly one web service. (When values are available from multiple web services it becomes important to address issues such as deciding which web service is of higher quality, which are beyond the scope of this paper.)

DEFINITION 3.2 (PRECEDENCE CONSTRAINTS). *If a bound attribute in $\mathcal{X}_j$ for WS$_j$ is obtained from some free attribute $\mathcal{Y}_i$ of WS$_i$, then there exists a* precedence constraint *WS$_i \prec$ WS$_j$, i.e., in any feasible execution plan for the query, WS$_i$ must precede WS$_j$.*

The precedence constraints may be represented as a directed acyclic graph (DAG) $\mathcal{G}$ in which there is a node corresponding to each web service, and there is a directed edge from WS$_i$ to WS$_j$

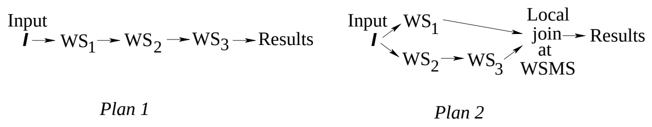

Figure 2: Plans for Example 3.4

if there is a precedence constraint WS$_i \prec$ WS$_j$. Note that $\mathcal{G}$ is not specified by the query, it is implied by which attributes are bound and which ones are free in the web services involved in the query.

EXAMPLE 3.3. *We continue with Example 1.1. With binding patterns, the three web services can be expressed as* $WS_1(\mathtt{n}^b, \mathtt{cr}^f)$, $WS_2(\mathtt{n}^b, \mathtt{ccn}^f)$, *and* $WS_3(\mathtt{ccn}^b, \mathtt{ph}^f)$. *Denoting the input names by I, the example query can be expressed as:*

SELECT $\mathtt{n}$
FROM $I(\mathtt{n}) \bowtie WS_1(\mathtt{n}^b, \mathtt{cr}^f) \bowtie WS_2(\mathtt{n}^b, \mathtt{ccn}^f) \bowtie WS_3(\mathtt{ccn}^b, \mathtt{ph}^f)$
WHERE $\mathtt{cr} >$ *threshold* $\wedge$ $\mathtt{ph} =$ *good*

Since the bound attribute $\mathtt{ccn}$ *in* WS_3 *is provided by* WS_2, *there exists a precedence constraint* $WS_2 \prec WS_3$. □

3.2 Query Plans and Execution Model

In the following example, we motivate possible execution plans.

EXAMPLE 3.4. *We continue with Example 1.1. The execution plan discussed in Example 1.1 is shown by Plan 1 in Figure 2. Although we show direct arrows between web services, in reality the arrows imply that data is returned to the WSMS, relevant predicates are applied, and the results are passed to the next web service.*

However, since there is no precedence constraint between WS_1 *and* WS_2, *we need not wait for output from* WS_1 *to invoke* WS_2. *Thus, an alternative execution plan for the same query is shown by Plan 2 in Figure 2, where the input list of names I is dispatched in parallel to* WS_1 *and* WS_2 *(denoted by two outgoing arrows from I). The results from* WS_2 *are then used to invoke* WS_3 *as in Plan 1. The final query result is obtained by joining the results from the two branches locally at the WSMS.* □

In general, an execution plan is an arrangement of the web services in the query into a DAG $\mathcal{H}$ with parallel dispatch of data denoted by multiple outgoing edges from a single web service, and rejoining of data denoted by multiple incoming edges into a web service. Note that the plan DAG $\mathcal{H}$ is distinct from the DAG $\mathcal{G}$ of precedence constraints among web services, although if WS$_i$ is an ancestor of WS$_j$ in $\mathcal{G}$, then it must also be so in $\mathcal{H}$ (i.e., $\mathcal{H}$ *respects the precedence constraints* specified by $\mathcal{G}$).

Given a plan DAG $\mathcal{H}$, it is executed as follows (see Figure 3). A thread T_i is established for each web service WS$_i$. Thread T_i takes input tuples from a separate join thread J_i that joins the outputs of the parents of WS$_i$ in $\mathcal{H}$. In the special cases when WS$_i$ has no parents in $\mathcal{H}$, T_i takes input from the table I, and when WS$_i$ has exactly one parent in $\mathcal{H}$ (say WS$_p$), T_i takes input directly from the output of thread T_p. Thread T_i uses its input tuples to invoke WS$_i$, filters the returned tuples, and writes them to its output. The final query result is obtained from the output of a join thread J_{out} that performs a join of the outputs of all the web services that are leaves in the plan $\mathcal{H}$. In the special case when there is only one leaf web service WS$_l$ in $\mathcal{H}$ (e.g., Plan 1 in Figure 2), the output from WS$_l$ directly forms the query result and thread J_{out} is not needed.

The join threads perform a *multiway stream join* of their inputs, and there are known techniques to perform such joins efficiently, e.g., [32]. Furthermore, using a technique similar to *punctuations* in data streams [11], a unique marker is inserted when branching, so the join threads know when joining tuples are complete and state can be discarded.

According to Figure 3, the WSMS has only one outstanding call to any individual web service at a time, i.e., while the WSMS is waiting for the results from a previous call to WS$_i$ to arrive, it does not make another call to WS$_i$. However, this assumption is not important to our approach. As we will see, our algorithms only rely on a quantity representing the maximum rate at which data can be obtained from a web service. This rate can often be boosted by making multiple outstanding calls to the web service [15]; how the rate is achieved does not affect the applicability of our algorithms.

Also, in this paper we assume that results from one web service are returned to the WSMS before being passed on to another web service. However, with sufficient standardization, it might be possible for one web service to send it results directly to another. Our model for pipelined execution and our optimization algorithms would not change under that model.

3.3 Bottleneck Cost Metric

As a first step in developing a query optimizer for web services, we assume that the goal of optimization is to minimize the total running time of queries. In reality, there are other metrics that might also be important. For example, if a web service call incurs a monetary cost, we may wish to minimize the total number of web service calls. A study of other interesting metrics and their tradeoff with query running time is an interesting topic of future work.

To obtain an expression for the running time of a particular query execution plan, we assume that the following two quantities can be tracked and estimated for each web service by the Profiling and Statistics component of the WSMS (Figure 1); we do not focus on profiling techniques in this paper.

1. Per-tuple Response time (c_i): If r_i is the maximum rate at which results of invocations can be obtained[1] from WS$_i$, we use $c_i = 1/r_i$ as the effective per-tuple *response time* (or intuitively, the cost) of WS$_i$. The maximum rate r_i for a web service WS$_i$ (and hence its effective per-tuple response time c_i) can often be boosted by batching several calls together (data chunking, Section 6), or making multiple parallel calls (as described at the end of Section 3.2). Our optimization algorithms are applicable regardless of how the best response time for a web service is achieved.

The response time of a web service may depend on a variety of factors, such as the web service provisioning, the load on the web service, and the network conditions. In this paper, as a first step, we give algorithms assuming the response time is a constant, so the query may need to be reoptimized if significant changes in response time are detected. As future work, we plan to explore ideas such as adaptive plans, or plans that are provably robust to variations in response time.

2. Selectivity (s_i): Recall lines 2-7 of thread T_i (Figure 3) where a tuple is input to WS$_i$ and relevant filters are applied to the returned results. The average number of returned tuples (per tuple input to WS$_i$) that remain unfiltered after applying all relevant predicates is denoted by s_i, and is referred to as the *selectivity* of web service WS$_i$. s_i may be ≤ 1. For instance, in Example 3.3, if 10% of the names in I have credit rating above threshold, then $s_1 = 0.1$. In general, s_i may also be > 1. In the same example, if every person holds 5 credit cards on average, then $s_2 = 5$.

In this paper, we assume web service selectivities are *indepen-*

[1]Note that r_i incorporates the time for transmission over the network, as well as the queuing delays and the processing time at the web service.

```
Algorithm ExecutePlan($\mathcal{H}$)
$\mathcal{H}$: An arrangement of web services into a DAG
 1. for each web service $\text{WS}_i$
 2.     launch a thread $T_i$
 3.     if $\text{WS}_i$ has no parents in $\mathcal{H}$
 4.         set up $T_i$ to take input from $I$
 5.     else if $\text{WS}_i$ has a single parent $\text{WS}_p$ in $\mathcal{H}$
 6.         set up $T_i$ to take input from $T_p$'s output
 7.     else
 8.         launch a join thread $J_i$
 9.         set up $T_i$ to take input from $J_i$'s output
10. launch join thread $J_{out}$
11. return output of $J_{out}$ as query result
```

```
Thread $T_i$:
1. while (tuples available on $T_i$'s input)
2.     read a tuple $s$ from $T_i$'s input
3.     invoke $\text{WS}_i$ with values $s.\mathcal{X}_i$
4.     for each returned tuple $t$
5.         apply all predicates $P_j(A_j)$ where $A_j \in \mathcal{Y}_i$
6.         if $t$ satisfies all predicates
7.             write $s \bowtie t$ to $T_i$'s output

Thread $J_i$
1. perform the join of the outputs of $\text{WS}_i$'s parents in $\mathcal{H}$
Thread $J_{out}$
1. perform the join of the outputs of web services that are leaves in $\mathcal{H}$
```

Figure 3: Query Execution Algorithm

dent, i.e., the selectivity of a web service does not depend on which web services have already been invoked. Extending our algorithms to work with *correlated* selectivities is an important direction for future work. Although, our model of selectivities is fairly general, it is not adequate to capture scenarios where the web service performs some form of aggregation, i.e., producing a fixed number of output tuples irrespective of the number of input tuples. Extension of our algorithms to such web services is an interesting direction of future work.

Consider the pipelined execution of a plan as specified in Figure 3. There is a time period at the beginning (respectively end) of query execution when the pipeline is filling up (respectively emptying out), after which a steady state is reached during which input tuples flow through the pipeline at a constant rate. For long-running queries—typically queries in which the input table I is large—the time spent to reach steady state is negligible compared to the total running time of the query. In such cases, minimizing the total running time is equivalent to maximizing the rate at which tuples in I are processed through the pipeline in steady state. When time to reach steady state is nonnegligible, then the query is typically short-running and less in need of optimization anyway. Thus, we focus on the processing rate during steady state.

Since all web services can be executing in parallel, the maximum rate at which input tuples can be processed through the pipelined plan is determined by the *bottleneck* web service: the web service that spends the most time on average per *original* input tuple in I. Next, we derive a formal expression for this cost metric.

Consider a query plan $\mathcal{H}$ specified as a DAG on the web services in the query. Let $P_i(\mathcal{H})$ denote the set of predecessors of WS_i in $\mathcal{H}$, i.e., all web services that are invoked before WS_i in the plan. Formally,

$$P_i(\mathcal{H}) = \{\text{WS}_j \mid \text{WS}_j \text{ has a directed path to } \text{WS}_i \text{ in } \mathcal{H}\} \quad (1)$$

Given a set $\mathcal{S}$ of web services, we define the combined selectivity of all the web services in $\mathcal{S}$ as $R[\mathcal{S}]$. By the independence assumption among selectivities, $R[\mathcal{S}]$ is given by:

$$R[\mathcal{S}] = \prod_{i \mid \text{WS}_i \in \mathcal{S}} \sigma_i \quad (2)$$

Then, for every tuple in I input to plan $\mathcal{H}$, the average number of tuples that WS_i needs to process is given by $R[P_i(\mathcal{H})]$. Since the average time required by WS_i to process a tuple in its input is c_i, the average processing time required by web service WS_i (or intuitively, the cost incurred by WS_i) per original input tuple in I is $R[P_i(\mathcal{H})] \cdot c_i$. Recall that plan cost is determined by the web service with maximum processing time per original input tuple in I. Thus the cost of the query plan $\mathcal{H}$ is given by the following metric (referred to as the *bottleneck cost metric*):

$$\text{cost}(\mathcal{H}) = \max_{1 \le i \le n} \Big(R[P_i(\mathcal{H})] \cdot c_i \Big) \quad (3)$$

EXAMPLE 3.5. *Consider Plan 1 in Figure 2 for the query in Example 3.3. Let the costs and selectivities of the web services be as follows:*

i	*1*	*2*	*3*
Cost of WS_i (c_i)	*2*	*10*	*5*
Selectivity of WS_i (s_i)	*0.1*	*5*	*0.2*

Let $|I|$ be the number of tuples in the input data I. In Plan 1, with the example selectivities, WS_1 needs to process $|I|$ tuples, WS_2 needs to process $0.1|I|$ tuples , and WS_3 needs to process $0.5|I|$ tuples. Thus, the time taken by WS_1, WS_2 and WS_3 per tuple in I is $2, 1,$ and 2.5 respectively. The cost of the plan is then $\max(2, 1, 2.5) = 2.5$. We arrive at the same number using (3).

Now consider Plan 2 in Figure 2. Its cost (using (3)*) is* $\max(2, 10, 25) = 25$. *Thus, for this example, Plan 2 is 10 times slower than Plan 1.* □

It may appear that the bottleneck cost metric ignores the work that must be done by the WSMS threads (Figure 3). Formally, we can treat all the work done at the WSMS as just another call in the pipeline. Our algorithms are designed under the assumption that the pipeline stage constituted by the WSMS is never the bottleneck, which seems realistic since it is unlikely that the simple operations the WSMS needs to perform will be more expensive than remote web service calls. This assumption is also validated by our experiments (Section 7).

We can now formally define the query optimization problem solved in this paper.

DEFINITION 3.6. (QUERY OPTIMIZATION OVER WEB SERVICES). *Given an SPJ query over web services (Definition 3.1) implying a DAG $\mathcal{G}$ of precedence constraints, find a query plan arranging the web services into a DAG $\mathcal{H}$ that respects all precedence constraints in $\mathcal{G}$, where cost($\mathcal{H}$) as given by* (3) *is minimized.* □

It is important to understand the basic differences between our scenario and a traditional centralized setting which also has query operators characterized by costs and selectivities. In the traditional setting, each operator is running on the same machine, so the cost of the plan is given not by the bottleneck cost but by the sum of the costs incurred by the operators (referred to as the *sum cost metric*). The sum cost metric has been considered in much previous work [3, 8, 16, 18], but to the best of our knowledge, our work is the first to consider the fundamentally different bottleneck cost metric. One critical difference between the two metrics as brought out in this paper is that under the bottleneck cost metric, the optimal plan can be found in polynomial time for general precedence constraints (as shown in Section 5), while under the sum cost metric, for general precedence constraints the optimal plan is hard to even approximate in polynomial time [6, 18].

In the next two sections, we consider the problem given by Definition 3.6 first without, and then with precedence constraints. Then in Section 6, we consider sending data to web services in chunks. Finally, we report on our experiments in Section 7.

4. NO PRECEDENCE CONSTRAINTS

In this section, we consider the special case of the problem given by Definition 3.6 when there are no precedence constraints, i.e., the DAG $\mathcal{G}$ has no edges. The absence of precedence constraints implies that no web service depends on another for its bound attributes, i.e., all bound attributes are available directly from the input data I. Then, a simple execution plan is to dispatch the input I in parallel to each of the web services, with the results joined back at the WSMS.

The main problem with simultaneously dispatching I to all of the web services is simply that each web service must process all of the tuples in I. If some web services are selective (i.e., have selectivity ≤ 1), then it is better for the slower web services to come near the end of the pipeline, reducing how much data they must process. This basic observation forms the intuition behind our algorithm for selective web services (Section 4.1). When web services may be proliferative (i.e., have selectivity > 1), we do use the idea of dispatching input in parallel to multiple web services. One interesting observation in our results is that the optimal arrangement of web services depends only on whether their selectivity is ≤ 1 or > 1, but not on the exact selectivity value.

4.1 Selective Web Services

In this section, we focus on the case when there are no precedence constraints and the selectivity of each web service is ≤ 1. Our algorithm for selective web services follows directly from the following two simple but key observations.

LEMMA 4.1. *There exists an optimal plan that is a linear ordering of the web services, i.e., has no parallel dispatch of data.*

PROOF. Suppose the optimal plan $\mathcal{H}$ has parallel dispatch of data to WS_i and WS_j. Modify $\mathcal{H}$ to $\mathcal{H}'$ where WS_i is moved to the point before the parallel dispatch, and the rest of $\mathcal{H}$ remains unchanged. The amount of data seen by each web service in $\mathcal{H}'$ is either the same as in $\mathcal{H}$, or s_i times that in $\mathcal{H}$. Since $s_i \leq 1$, the bottleneck in $\mathcal{H}'$ is at most as much in $\mathcal{H}$. Continuing this flattening, we find a linear plan having cost at most that of $\mathcal{H}$. $\square$

LEMMA 4.2. *Let $WS_1, \ldots, WS_n$ be a plan with a linear ordering of the web services. If $c_i > c_{i+1}$, then WS_i and WS_{i+1} can be swapped without increasing the cost of the plan.*

PROOF. Let $\mathcal{H}$ be the plan in which the ordering of the web services is $\text{WS}_1, \ldots, \text{WS}_n$, and let $\mathcal{H}'$ denote the same plan but with WS_i swapped with WS_{i+1}. Let f denote $\prod_{j=1}^{i-1} s_j$. By (3):

$$\text{cost}(\mathcal{H}) = \max(fc_i, fs_ic_{i+1}, \text{other terms}) \qquad (4)$$

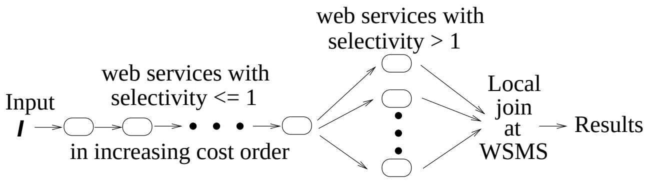

Figure 4: Optimal Plan (No Precedence Constraints)

where the other terms are the cost terms for the rest of the web services. These other terms remain the same when we consider $\mathcal{H}'$. Thus:

$$\text{cost}(\mathcal{H}') = \max(fc_{i+1}, fs_{i+1}c_i, \text{other terms}) \qquad (5)$$

Consider the terms in $\text{cost}(\mathcal{H}')$. $fc_{i+1} < fc_i$ by the lemma statement, and $fs_{i+1}c_i \leq fc_i$ since $s_{i+1} \leq 1$. Since other terms in $\text{cost}(\mathcal{H}')$ are also present in $\text{cost}(\mathcal{H})$, $\text{cost}(\mathcal{H}') \leq \text{cost}(\mathcal{H})$. $\square$

Lemmas 4.1 and 4.2 immediately lead to the following result.

THEOREM 4.3. *For selective web services with no precedence constraints, the optimal plan is a linear ordering of the web services by increasing response time, ignoring selectivities.*

PROOF. From Lemma 4.1, there exists a linear ordering of the web services that is optimal. Consider any linear ordering of the web services that is optimal. If, in this ordering, there is a higher cost service followed immediately by a lower cost service, by Lemma 4.2 we can swap them without increasing the cost of the plan. We continue such swapping until there does not exist a higher cost web service followed immediately by a lower cost one, thereby obtaining the result. $\square$

Recall that in the derivation of the cost expression for plans (Section 3.3), we assumed that the selectivities of web services are independent. If independence does not hold, the cost of the query plan can be written in terms of conditional rather than absolute selectivities. However, as long the conditional selectivities are also ≤ 1, Theorem 4.3 applies. Thus our result extends to web services with *correlated selectivities*.

4.2 Proliferative Web Services

We now consider the case when some web services may have selectivity > 1.

THEOREM 4.4. *The overall optimal plan for a query consisting of both selective and proliferative web services with no precedence constraints is as shown in Figure 4.*

PROOF. Consider a query consisting of a set of selective web services $\mathcal{W}_s$ and a set of proliferative web services $\mathcal{W}_p$, and having no precedence constraints. In the absence of precedence constraints, a web service $\text{WS}_i \in \mathcal{W}_p$ should not occur before any other web service WS_j in a pipeline, since it will only increase work for WS_j. Thus, the web services in $\mathcal{W}_p$ should be invoked in parallel at the end of the plan. Using the results from Section 4.1, the web services in $\mathcal{W}_s$ should be placed in increasing cost order. Thus, the overall optimal plan is as shown in Figure 4. $\square$

5. PRECEDENCE CONSTRAINTS

In this section, we develop a general, polynomial-time algorithm for the problem given by Definition 3.6 when there may be precedence constraints among some of the web services in the query (recall Definition 3.2). Recall that precedence constraints are specified as a DAG $\mathcal{G}$.

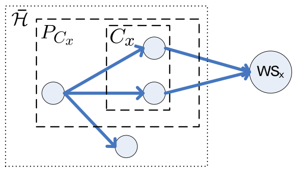

Figure 5: Placing WS$_x$ after a cut C_x in $\bar{\mathcal{H}}$

We define M_i as the set of all web services that are prerequisites for WS$_i$, i.e.,

$$M_i = \{\text{WS}_j \mid \text{WS}_j \prec \text{WS}_i\} \qquad (6)$$

Our overall algorithm (described in Section 5.2) builds the plan DAG $\mathcal{H}$ incrementally by greedily augmenting it one web service at a time. At any stage the web service that is chosen for addition to $\mathcal{H}$ is the one that can be added to $\mathcal{H}$ with minimum cost, and all of whose prerequisite web services have already been added to $\mathcal{H}$. The crux of the algorithm lies in finding the minimum cost of adding a web service to $\mathcal{H}$, described next.

5.1 Adding a Web Service to the Plan

Suppose we have constructed a partial plan $\bar{\mathcal{H}}$ and we wish to add WS$_x$ to $\bar{\mathcal{H}}$. To find the minimum cost of adding WS$_x$ to $\bar{\mathcal{H}}$ (without modifying $\bar{\mathcal{H}}$), we compute the best *cut* C_x in the DAG $\bar{\mathcal{H}}$, such that on placing edges from the web services in C_x to WS$_x$, the cost incurred by WS$_x$ is minimized. An example of a cut is shown in Figure 5. Formally, a cut C in a DAG $\mathcal{H}$ is defined as any set of web services in $\mathcal{H}$ such that there does not exist a directed path in $\mathcal{H}$ from one web service in the cut to another. For the cut C_x in $\bar{\mathcal{H}}$, we also define the set P_{C_x} (also shown in Figure 5) as consisting of all the web services in C_x and all their predecessors in $\bar{\mathcal{H}}$, i.e., (recall (1) for definition of $P_j(\bar{\mathcal{H}})$)

$$P_{C_x} = C_x \cup \{\text{WS}_i \mid \text{WS}_i \in P_j(\bar{\mathcal{H}}) \text{ for } \text{WS}_j \in C_x\} \qquad (7)$$

Note that given P_{C_x}, the cut C_x can be easily computed as only those web services in P_{C_x} that are not predecessors of some other web service in P_{C_x}.

Recall definition of $R[\mathcal{S}]$ from (2). When we place edges from the web services in C_x to WS$_x$ (as shown in Figure 5), the total cost incurred by WS$_x$ is given by:

$$\text{cost}(\text{WS}_x) = R[P_{C_x}] \cdot c_x$$

Let us associate a variable z_i with every WS$_i \in \bar{\mathcal{H}}$ that is set to 1 if WS$_i \in P_{C_x}$, and to 0 otherwise. Then, from (2), we have:

$$R[P_{C_x}] = \prod_{i \mid \text{WS}_i \in \bar{\mathcal{H}}} (\sigma_i)^{z_i} \qquad (8)$$

Then the optimal set P_{C_x} (and hence the optimal cut C_x) such that cost(WS$_x$) is minimized, is obtained by solving the following linear program where the variables are the z_is.

$$\boxed{\begin{array}{llll} \text{Minimize} \log c_x + \sum_{i \mid \text{WS}_i \in \bar{\mathcal{H}}} z_i \log \sigma_i \text{ subject to} & & & \\ z_i & = & 1 & \forall i \mid \text{WS}_i \in M_x \\ z_i & \geq & z_j & \forall i, j \mid \text{WS}_i \in P_j(\bar{\mathcal{H}}) \\ z_i & \in & [0,1] & \forall i \end{array}} \qquad (9)$$

The objective function of the above linear program minimizes $\log(\text{cost}(\text{WS}_x))$ that is equivalent to minimizing cost(WS$_x$). We take logarithms to ensure that the objective function is linear in the variables. The first constraint in (9) ensures that P_{C_x} includes all the prerequisite web services for WS$_x$ (so that it is feasible to add WS$_x$ after the cut C_x). The second constraint ensures that the set P_{C_x} is chosen according to the current structure of $\bar{\mathcal{H}}$, i.e., if a web service WS$_i$ is chosen in P_{C_x}, all predecessors of WS$_i$ in $\bar{\mathcal{H}}$ are also chosen in P_{C_x}. Note that the third constraint relaxes the linear program to include fractional z_i's instead of just integers. However, there always exists an optimal integer solution to the above linear program as shown by the following theorem (the proof appears in [23]).

LEMMA 5.1. *The linear program* (9) *has an optimal solution where each z_i is set to either* 0 *or* 1.

The optimal integer solution to (9) can be computed by converting the linear program into a network flow problem [6]. Once the optimal integer solution has been found, all web services with $z_i = 1$ in the solution define the optimal set P_{C_x}, which in turn defines the optimal cut C_x.

```
Algorithm Greedy
1. $\bar{\mathcal{H}} \leftarrow \phi$; $F(\bar{\mathcal{H}}) \leftarrow \{\text{WS}_i \mid M_i = \phi\}$
2. while ($\bar{\mathcal{H}}$ does not include all web services in $\mathcal{F}$)
3.     for each web service WS$_x$ in $F(\bar{\mathcal{H}})$
4.         $v_x \leftarrow$ optimal value of linear program (9)
5.         $C_x \leftarrow$ optimal cut in $\bar{\mathcal{H}}$ from the solution to (9)
6.     WS$_{opt} \leftarrow$ web service WS$_x$ with minimum $v_x$
7.     add WS$_{opt}$ to $\bar{\mathcal{H}}$ placing edges from $C_{opt}$ to WS$_{opt}$
8.     update $F(\bar{\mathcal{H}})$ according to Equation (10)
```

Figure 6: Greedy Algorithm for Bottleneck Cost Metric

5.2 Greedy Algorithm

We now describe our general greedy algorithm shown in Figure 6. For a partial plan DAG $\bar{\mathcal{H}}$, we first define the *frontier set* $F(\bar{\mathcal{H}})$ as the set of all web services that are candidates for addition to $\bar{\mathcal{H}}$, since all their prerequisite web services have already been added to $\bar{\mathcal{H}}$. Formally :

$$F(\bar{\mathcal{H}}) = \{\text{WS}_i \mid \text{WS}_i \notin \bar{\mathcal{H}} \land M_i \subseteq \bar{\mathcal{H}}\} \qquad (10)$$

We start by initializing $\bar{\mathcal{H}}$ as empty, and the frontier set $F(\bar{\mathcal{H}})$ as all those web services that do not have any prerequisite web services (Line 1). Then for each web service WS$_x \in F(\bar{\mathcal{H}})$, we solve the linear program (9) to determine the optimal cost of adding WS$_x$ to $\bar{\mathcal{H}}$ (Line 4). Let WS$_{opt}$ be the web service having least such cost (Line 6), and let the optimal cut for adding WS$_{opt}$ be C_{opt} as given by the solution to the linear program. WS$_{opt}$ is then added to $\bar{\mathcal{H}}$ by placing directed edges from the web services in cut C_{opt} to WS$_{opt}$ (Line 7). We update the frontier set $F(\bar{\mathcal{H}})$ according to Equation (10), and continue in this fashion until the DAG $\bar{\mathcal{H}}$ includes all the web services.

5.3 Analysis of Greedy Algorithm

We now show that our algorithm *Greedy* (Figure 6) is correct, i.e., it produces the optimal plan. Note that since the cost of a plan is determined only by the bottleneck in the plan, in general there are many possible optimal plans. We show that our greedy algorithm finds an optimal plan. The proof is by induction on the number of web services added by *Greedy* to the partial plan $\bar{\mathcal{H}}$.

Our inductive hypothesis is that when k web services have been added to the DAG $\bar{\mathcal{H}}$ constructed by *Greedy*, $\bar{\mathcal{H}}$ agrees (in terms of edges placed) with some optimal solution restricted to just the web services in $\bar{\mathcal{H}}$, i.e., there exists an optimal solution that has $\bar{\mathcal{H}}$

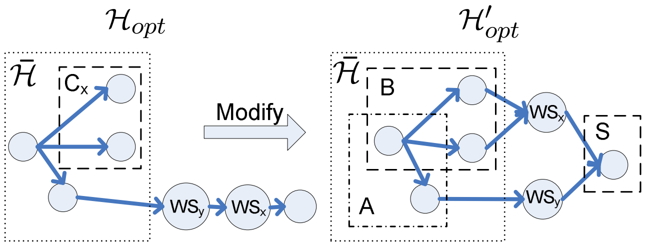

Figure 7: Modifying $\mathcal{H}_{opt}$ to $\mathcal{H}'_{opt}$

as a subgraph. The base case for our induction is $k = 0$ which is trivially satisfied since the empty DAG is a subgraph of any DAG.

LEMMA 5.2. *When Greedy adds the $(k+1)$th web service, the inductive hypothesis still holds.*

PROOF. Let $\bar{\mathcal{H}}$ denote the partial DAG when k web services have been added by *Greedy*. Let $\bar{\mathcal{H}}$ be a subgraph of some optimal plan $\mathcal{H}_{opt}$ (by the inductive assumption). Suppose the $(k+1)$th web service chosen by *Greedy* to be added to $\bar{\mathcal{H}}$ is WS$_x$. Let the optimal cut in $\bar{\mathcal{H}}$ for adding WS$_x$ be C_x. An example is shown in Figure 7.

Consider the position of WS$_x$ in $\mathcal{H}_{opt}$. Suppose $\mathcal{H}_{opt}$ has some other web service WS$_y \in F(\bar{\mathcal{H}})$ that takes input only from web services in $\bar{\mathcal{H}}$, and WS$_x$ is placed such that WS$_y \in P_x(\mathcal{H}_{opt})$ (see Figure 7). In general, there could be many such WS$_y$'s that are predecessors of WS$_x$ in $\mathcal{H}_{opt}$; the proof remains unchanged. Modify $\mathcal{H}_{opt}$ to $\mathcal{H}'_{opt}$ as follows. Remove the input to WS$_x$ in $\mathcal{H}_{opt}$ and make its input the cut C_x (just as *Greedy* does). The output of WS$_x$ in $\mathcal{H}_{opt}$ is replaced by the join of the output of WS$_x$ in $\mathcal{H}'_{opt}$ and the input to WS$_x$ in $\mathcal{H}_{opt}$. An example of this modification is shown in Figure 7. We now show that $\mathcal{H}'_{opt}$ is also an optimal plan.

CLAIM 5.3. *On modifying $\mathcal{H}_{opt}$ to $\mathcal{H}'_{opt}$, the cost incurred by any web service except WS$_x$ cannot increase.*

PROOF. The only web services except WS$_x$ whose cost in $\mathcal{H}'_{opt}$ may be different from their cost in $\mathcal{H}_{opt}$ are those for which WS$_x$ is a predecessor in $\mathcal{H}_{opt}$. Let S denote this set of web services. Let A be the set $P_x(\mathcal{H}_{opt}) \cap \bar{\mathcal{H}}$ and B be the set $P_x(\mathcal{H}'_{opt})$. See Figure 7 for examples of S, A, and B. Note that B is the same as P_{C_x}, i.e., the set that *Greedy* chooses to place before WS$_x$. The combined selectivity of the set $B-A$, i.e., $R[B-A]$, can be at most one; if not, *Greedy* would have chosen P_{C_x} to be $B \cap A$ instead of B. Note that $B \cap A$ is a feasible choice for P_{C_x} since A and B are both feasible sets of web services to place before WS$_x$. In $\mathcal{H}_{opt}$, the web services in S had input from the set of web services $A \cup \{\text{WS}_x\} \cup \{\text{other web services} \notin \bar{\mathcal{H}}\}$. In $\mathcal{H}'_{opt}$, the web services in S have input from the expanded set of web services $A \cup B \cup \{\text{WS}_x\} \cup \{\text{same other web services} \notin \bar{\mathcal{H}}\}$. Since $R[B-A]$ is at most 1, the number of data items seen by web services in S in $\mathcal{H}'_{opt}$ is at most as many as in $\mathcal{H}_{opt}$. Thus the cost of any web service in S cannot increase on modifying $\mathcal{H}_{opt}$ to $\mathcal{H}'_{opt}$. □

Now consider the cost incurred by WS$_x$ in $\mathcal{H}'_{opt}$. If $R[P_x(\mathcal{H}'_{opt})] \leq R[P_x(\mathcal{H}_{opt})]$, the cost incurred by WS$_x$ also does not increase, hence combined with Claim 5.3, we have cost$(\mathcal{H}'_{opt}) \leq$ cost$(\mathcal{H}_{opt})$. If $R[P_x(\mathcal{H}'_{opt})] > R[P_x(\mathcal{H}_{opt})]$, there are two cases:

1. Suppose WS$_x$ is the bottleneck in $\mathcal{H}'_{opt}$. Then the cost incurred by any other web service, specifically by WS$_y$ in $\mathcal{H}'_{opt}$, is smaller. But then since WS$_y \in F(\bar{\mathcal{H}})$, *Greedy* would have chosen WS$_y$ to add to $\bar{\mathcal{H}}$ instead of WS$_x$. Hence this case is not possible.
2. If WS$_x$ is not the bottleneck in $\mathcal{H}'_{opt}$, then cost$(\mathcal{H}'_{opt})$ is given by the cost incurred by some other web service. Hence, by Claim 5.3, we have cost$(\mathcal{H}'_{opt}) \leq$ cost$(\mathcal{H}_{opt})$.

Thus in all cases, cost$(\mathcal{H}'_{opt}) \leq$ cost$(\mathcal{H}_{opt})$. Since $\mathcal{H}_{opt}$ is an optimal plan, $\mathcal{H}'_{opt}$ is also optimal. After *Greedy* adds WS$_x$ to $\bar{\mathcal{H}}$, $\bar{\mathcal{H}}$ is a subgraph of $\mathcal{H}'_{opt}$. Hence assuming that the inductive hypothesis holds when k web services have been added to $\bar{\mathcal{H}}$, it still holds on adding the $(k+1)$th web service. □

THEOREM 5.4. *Algorithm Greedy computes an optimal plan in $O(n^5)$ time where n is the number of web services.*

PROOF. The correctness is immediate from Lemma 5.2 by induction on the number of web services added to $\bar{\mathcal{H}}$. The running time of *Greedy* is at most the time taken to solve the linear program (9) $O(n^2)$ times. The linear program (9) can be solved in $O(n^3)$ time using a network flow algorithm [6]. Thus the total running time of *Greedy* is $O(n^5)$. □

Although $O(n^5)$ complexity may seem high, Theorem 5.4 is still very interesting since it demonstrates that under the bottleneck cost metric, the optimal plan can be found in polynomial time for arbitrary precedence constraints. This result is somewhat surprising given previous negative results for the analogous problem under the sum cost metric [6, 18]. Also note that the analysis in Theorem 5.4 to obtain the $O(n^5)$ bound is pessimistic since it assumes the frontier set is constantly of size n; in practice, the frontier set will be smaller due to precedence constraints.

EXAMPLE 5.5. *We demonstrate the operation of our algorithm for optimization of the query in Example 3.3 with costs and selectivities as given in Example 3.5. Initially, WS$_1$ and WS$_2$ belong to the frontier set $F(\bar{\mathcal{H}})$. Since $c_1 < c_2$, WS$_1$ is added first to the plan $\bar{\mathcal{H}}$. $F(\bar{\mathcal{H}})$ remains unchanged. Now to add WS$_2$, there are two possibilities: either after WS$_1$, or in parallel with WS$_1$. Since the former possibility has lower cost, WS$_2$ is added after WS$_1$. $F(\bar{\mathcal{H}})$ is now updated to $\{WS_3\}$. There is only possibility for its addition: after WS$_2$. Thus we find that the optimal plan is a linear one as shown by Plan 1 in Figure 2.* □

6. DATA CHUNKING

There is usually some amount of overhead incurred on making any web service call, e.g., parsing SOAP/XML headers and fixed costs associated with network transmission. Hence it can be very expensive to invoke a web service separately for each tuple. To amortize the overhead, a web service may provide a mechanism to pass tuples to it in batches, or *chunks*. Each tuple is still treated individually by the web service, but the overall overhead is reduced.

When a chunk of input data is passed to web service WS$_i$, we assume the entire answer arrives back at the WSMS as a single chunk. The response time of WS$_i$ usually depends on the size of the input chunk. We use $c_i(k)$ to denoting the response time of WS$_i$ on a chunk of size k. We assume there is a limit k_i^{max} on the maximum chunk size accepted by web service WS$_i$. Chunk-size limits can arise, e.g., from limits on network packet lengths.

When web services can accept input in the form of chunks, the query optimizer must decide the optimal chunk size to use for each web service. The optimal chunk size for web service WS$_i$ will obviously depend on how the response time $c_i(k)$ of WS$_i$ varies as a function of the chunk size k. We first give an example, based on a real experiment we conducted, showing that $c_i(k)$ may depend in unexpected ways on k. We then show that the optimal chunk size for a web service depends only on $c_i(k)$ and is independent of the query plan in which it is being invoked, and we give an algorithm for choosing optimal chunk sizes.

EXAMPLE 6.1. *We implemented an ASP.NET web service as follows. We created a table* `T(int a, int b, primary key a)` *in a commercial database system, with a clustered index on attribute a. The table was loaded with 100,000 tuples. The web service accepted a list of values for a (the chunk) and returned the corresponding values for b, by issuing a SQL query to the database system in which the list of a values was put in an IN clause.*

We measured the response time of the web service when queried by a remote host with various chunk sizes. We found that the response time was not just linear in the chunk size, but also had a small quadratic component to it. Thus, the time per tuple $r(k)/k$ first decreases, and then increases with k. Our current (unverified) hypothesis is that the quadratic component may be due to sorting of the IN list by the database query optimizer. The main point to glean from this example is that depending upon implementation, web service response times may vary in unexpected ways with chunk size. □

The following theorem gives the optimal chunk size for each web service WS_i and shows that it is independent of the query plan.

THEOREM 6.2. *The optimal chunk size to be used by WS_i is k_i^* such that $c_i(k_i^*)/k_i^*$ is minimized for $1 \leq k_i^* \leq k_i^{max}$.* □

PROOF. Let c_i denote the average response time of WS_i per input tuple as in Section 3.2. If WS_i uses a chunk size k_i, its per-tuple response time is given by $c_i = c_i(k_i)/k_i$. Recall from Equation (3) that the cost of a plan is given by $\max_{1 \leq i \leq n} \left((\prod_{j=1}^{i-1} s_j) c_i \right)$. Since the selectivity values remain unchanged in the presence of chunking, the cost of the plan is minimized when c_i is minimized for each web service WS_i. Hence, independent of the actual query plan, the optimal chunk size for WS_i is k_i^* such that $c_i(k_i^*)/k_i^*$ is minimized. □

In general, the response time $c_i(k)$ of a web service WS_i may be any function of the chunk size k, as demonstrated by Example 6.1 above. Hence, to apply Theorem 6.2, the optimizer relies on the Profiling and Statistics component to measure $c_i(k)$ for different values of k. Profiling may be combined with query processing by trying out various chunk sizes during query execution and measuring the corresponding response times. Once the optimal chunk size k_i^* for each web service WS_i has been determined, the optimal plan is found by setting $c_i = c_i(k_i^*)/k_i^*$ for each WS_i, and applying our query optimization algorithm from Section 5.

Note that according to Theorem 6.2, it might be optimal to use different chunk sizes with different web services. In steady state, this is ensured by maintaining a buffer of intermediate results between any two consecutive web services in the pipelined plan.

7. IMPLEMENTATION AND EXPERIMENTS

We implemented an initial prototype WSMS, described in Section 7.1. Here we report on a few experiments with it. Not surprisingly, in our experiments, query plan performance reflects our theoretical results (thereby validating our cost model). Using total running time of queries as a metric, we compared the plans produced by our optimization algorithm (referred to as *Optimizer*) against the plans produced by the following simpler algorithms:

1. *Parallel*: This algorithm attempts to exploit the maximum possible parallelism by dispatching data in parallel to web services whenever possible. For example, if there are no precedence constraints, data is dispatched in parallel to all web services followed by a join at the end. An example of how this algorithm operates in the presence of precedence constraints will be given in Section 7.3.
2. *SelOrder*: One heuristic for efficient query processing is to reduce data as early as possible by putting the web services with lower selectivities earlier in the pipeline. *SelOrder* models this heuristic by building a (linear) plan as follows: Out of all web services whose input attributes are available, the web service with lowest selectivity is placed in the plan, and the process is repeated until all web services have been placed.

We also compared the running times of queries with and without data chunking, to demonstrate the benefits of chunking. Finally, we compared the total CPU cost at the WSMS against the cost of the slowest web service to substantiate our claim that the WSMS is not the bottleneck in pipelined processing. The main findings from our experiments are:

1. For scenarios both with and without precedence constraints, the plans produced by *Optimizer* can perform vastly better (up to about 7 times better for the problem instances we experimented with) than the plans produced by *Parallel* or *SelOrder*.
2. Using data chunking query running time can be reduced by up to a factor of 3.
3. The WSMS cost is significantly lower than the cost of the slowest web service in the plan, demonstrating that the WSMS is not the bottleneck in a pipelined plan.

We first describe our WSMS prototype and the experimental setup in Section 7.1. We then describe our experiments for scenarios with no precedence constraints in Section 7.2, and for scenarios with precedence constraints in Section 7.3. In Section 7.4, we describe our experiments with data chunking. Finally, in Section 7.5, we report our results of measuring the cost incurred at the WSMS.

7.1 Prototype and Experimental Setup

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

Our WSMS prototype is a multithreaded system written in Java. It implements Algorithm *ExecutePlan* (Figure 3), and can execute any general execution plan with any specified chunk sizes. For communicating with web services using SOAP, our prototype uses Apache Axis [2] tools. Given a description of a web service in the Web Service Definition Language [34], Axis generates a class such that the web service can be invoked simply by calling a method of the generated class. The input and output types of the web service are also encapsulated in generated classes. Our prototype uses these classes to conveniently invoke each web service as if it were a local function call. However, since the web service that a particular thread has to interact with is known only at runtime (recall Figure 3), the names of the corresponding classes to be used are also known only at runtime. To get around this problem, our prototype uses Java Reflection [27] to load classes and their methods dynamically.

We use Apache Tomcat [30] as the application server and Apache Axis [2] 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. WS_i is of the form $\text{WS}_i(a^b, b^f)$: 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. Data chunking is implemented by issuing

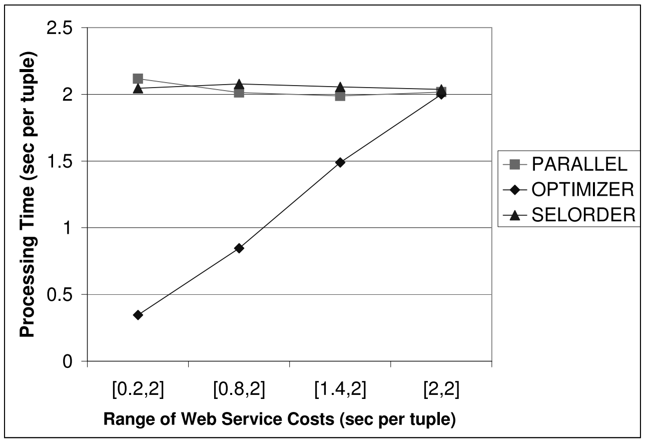

Figure 8: No Precedence Constraints

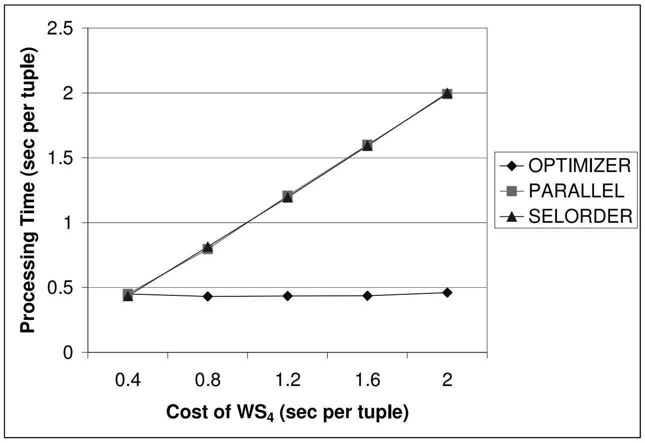

Figure 9: Precedence Constraints

a SQL query with an IN clause. The tables T_i are stored using the lightweight IBM Cloudscape DBMS. Since attribute a is the primary key, Cloudscape automatically builds an index on a. The tables T_i were each populated with tuples of the form (j, j) for j in $\{1, \ldots, 10{,}000\}$.

For our experiments, we needed web services with different costs and selectivities. To obtain different costs, we introduced 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. Our experimental web services, as described in the previous paragraph, return exactly one tuple for each input value of attribute a (since a is a key). To obtain a selectivity $s_i < 1$ for web service WS_i, we rejected each tuple returned by WS_i at the WSMS with probability $1 - s_i$. To obtain $s_i > 1$, for each tuple returned by WS_i, we created between 1 and $2s_i$ new tuples, each with the same value in attribute a as the returned tuple, and randomly generated values in attribute b from the range $1, \ldots, 10{,}000$ (so that these values of b could be used as input to another web service).

The WSMS is run on a different machine from the ones on which the web services were running. For every run, the WSMS randomly generated 2000 input tuples that formed the input table I. Each input tuple had a single attribute with value in the range $1, \ldots, 10{,}000$. The query executed was a join of all the web services and the input table I. For this query, a particular execution plan $\mathcal{P}$ along with the chunk sizes to be used by each web service was specified to the WSMS. The WSMS then processed all the tuples in I through the plan $\mathcal{P}$ in a pipelined fashion. Over 5 independent runs, the average processing time per tuple of I is then used as a metric for comparing $\mathcal{P}$ against other plans.

7.2 No Precedence Constraints

In this experiment, we set up four web services $\mathrm{WS}_1, \ldots, \mathrm{WS}_4$ with no precedence constraints among them, i.e., the single attribute in the input tuples served as the input attribute to all the web services. We did not use data chunking in this experiment. With its basic functionality of one database lookup, each web service had a response time (or cost) of approximately 0.2 second per tuple. We added additional delay to control the costs of different web services.

We consider various cost ranges $\tilde{\mathbf{c}}$, and assign $\mathrm{WS}_1, \ldots, \mathrm{WS}_4$ uniformly increasing costs in the range $\tilde{\mathbf{c}}$. To ensure that different plans are produced by *Optimizer* (which orders the web services by increasing cost according to Theorem 4.3), and by *SelOrder* (which orders the web services by increasing selectivity), we assigned selectivities to web services in the reverse order of cost: the selectivities of $\mathrm{WS}_1, \ldots, \mathrm{WS}_4$ were set as $0.4, 0.3, 0.2, 0.1$ respectively.

Figure 8 shows the costs of the plans produced by the various algorithms as the range of costs $\tilde{\mathbf{c}}$ is varied from $[0.2, 2]$ seconds per tuple to $[2, 2]$ seconds per tuple. *Parallel* dispatches data to all web services in parallel and hence has a bottleneck cost equal to the cost of the highest-cost web service WS_4. *SelOrder* puts WS_4 first since it has lowest selectivity, so it incurs the same bottleneck cost as *Parallel*. *Optimizer* is able to reduce the bottleneck cost to well below the cost of WS_4 by placing the web services in increasing order of cost. Only when all the web services become expensive does *Optimizer* incur a cost equal to that of *Parallel* or *SelOrder*. In this experiment we also verified that the actual per-tuple processing time is very close to that predicted by our cost model, thereby showing the accuracy of our cost model.

7.3 Precedence Constraints

In this experiment, we again set up for four web services WS_1, $\ldots$, WS_4. The single attribute in the input tuples served as the input attribute to WS_1 and WS_2. The output attribute from WS_1 (respectively WS_2) served as the input attribute to WS_3 (respectively WS_4). Thus, we had two precedence constraints, $\mathrm{WS}_1 \prec \mathrm{WS}_3$, and $\mathrm{WS}_2 \prec \mathrm{WS}_4$. We did not use data chunking.

WS_1 and WS_2 were set up to be proliferative, with selectivities 2 and 1 respectively. The selectivities of WS_3 and WS_4 were set as 0.1 each. The cost of each of $\mathrm{WS}_1, \ldots, \mathrm{WS}_3$ was set as 0.2 second per tuple. The cost of WS_4 was varied from 0.4 to 2 seconds per tuple.

For this scenario, *Parallel* chooses a plan in which data is first dispatched in parallel to WS_1 and WS_2. Then, to exploit parallelism between WS_3 and WS_4, WS_3 is placed in the WS_1 branch, and WS_4 in the WS_2 branch. Based on selectivities, *SelOrder* orders the web services as WS_2, WS_4, WS_1, WS_3. *Optimizer* first groups WS_1 and WS_3 together, and WS_2 and WS_4 together. Then the group containing WS_1 is placed before the other, since it has lower cost. Thus the overall order produced by *Optimizer* is WS_1, WS_2, WS_3, WS_4.

Figure 9 shows the costs of the plans produced by the various algorithms as the cost of WS_4 is increased. Both *Parallel* and *SelOrder* incur the cost of WS_4 as the bottleneck, while *Optimizer* reduces the bottleneck cost to below the cost of WS_4 by placing it last in the pipelined plan.

7.4 Data Chunking

In this experiment, we again set up four web services with no

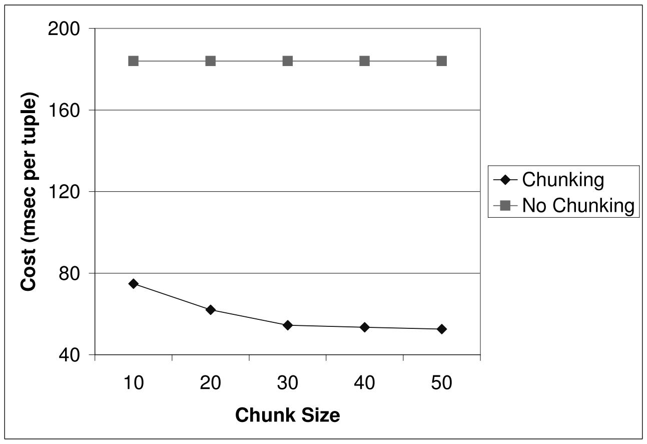

Figure 10: Effect of Data Chunking

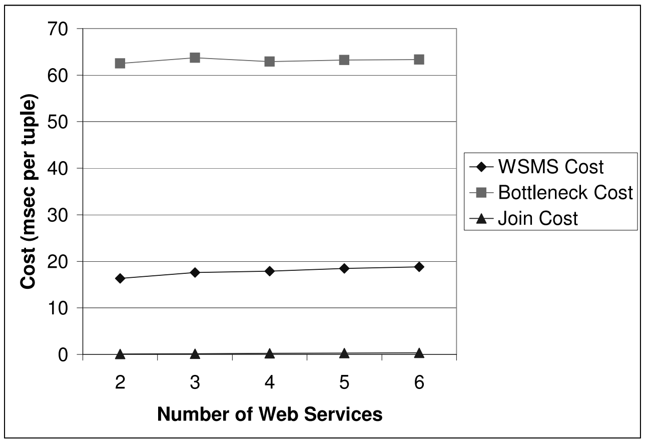

Figure 11: WSMS Cost vs. Bottleneck Cost

precedence constraints as in Section 7.2. For this experiment, we did not add additional delay to any web service so costs were uniform, and the selectivity of each web service was set as 0.5. The web services were arranged in a linear pipeline according to *Optimizer*, and the chunk size used by each web service was equal.

Figure 10 shows how the per-tuple cost varies as the chunk size used by the web services is increased. For comparison, we also show the per-tuple cost without chunking, i.e., with chunk size 1. Even using a chunk size of 20 reduces the per-tuple cost by more than a factor of 3. However, on increasing the chunk size further, the cost does not reduce significantly. Hence most of the benefit of chunking can be achieved even by using a relatively small chunk size.

7.5 WSMS Cost

In this experiment, we compared the cost incurred at the WSMS against the bottleneck web service cost in a pipelined plan. We varied the number of web services involved in the query. There were no precedence constraints, uniform web service costs, and the selectivity of each web service was set as 0.5. To demonstrate that the WSMS does not become the bottleneck even with data chunking, each web service used a chunk size of 20.

The WSMS cost was measured with the web services arranged in a linear pipeline according to *Optimizer*. To demonstrate that the join threads in a plan (recall Figure 3) does not make the WSMS the bottleneck, we also executed another plan in which data was dispatched in parallel to all web services, and the cost of joining the results at the WSMS was measured.

Figure 11 shows the bottleneck cost, the WSMS cost, and the cost of the join thread as the number of web services involved in the query is increased. Even as the number of web services increases, the WSMS cost remains significantly lower than the cost of the bottleneck web service. Figure 11 also shows that the cost of the join thread is negligible compared to the bottleneck cost.

It is important to note that our measurements in the above experiment are only conservative, and numbers in a real setting can only be better, due to the following reasons:

- In our experiments, the WSMS and the web services were running on different machines but on the same network. Accessing web services over the internet may add an additional order of magnitude to their response time.
- Our WSMS prototype makes heavy use of the Java Reflection API [27], which is known to be extremely slow compared to direct method invocations. In a separate experiment, we found that a method call using Reflection can be 15 to 20 times slower than a direct call to the same method. The inefficiency of Reflection is also evident in how the cost of the join thread (which does not use Reflection) compares with the rest of the WSMS cost. In the next version of our prototype, we plan to redesign the system to avoid the use of Reflection, giving up the convenience of classes generated by Axis, but decreasing the cost incurred at the WSMS by at least an order of magnitude.

Given the above factors, it is unlikely that in any real setting, the WSMS cost can become the bottleneck in the pipelined processing of a query over multiple web services.

8. CONCLUSIONS

We have proposed the overall goal of a general-purpose Web Service Management System (WSMS), enabling clients to query a collection of web services in a transparent and integrated fashion. In this paper, we focus on new query optimization issues that arise in a WSMS. Our execution model consists of pipelined query processing over web services, and we derive the “bottleneck” cost metric to characterize the cost of a pipelined plan. For this cost metric, we have devised new algorithms to: (a) decide the optimal arrangement of web services in a pipelined plan, respecting precedence constraints, and (b) decide the optimal chunk size to use when sending data to each web service. While the algorithms in this paper form the basis of a WSMS query 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 allow different input tuples to follow different plans as in [9, 20], leading to even higher overall performance.
- Our algorithms currently do not incorporate variance or uncertainty in the response times of web services, or more generally, quality of service (QoS) information about web services. It is important to address the problem of finding plans that consistently choose the highest-quality available web services and that adapt to changes in web service response times.
- Our query optimization algorithm relies on knowledge of web service response times and selectivities. Hence we need to develop profiling techniques that can accurately track these quantities and detect changes in them. Work on self-tuning histograms [5] may be relevant to track selectivities.
- We have not considered web services with *monetary* costs.

In that scenario, we may wish to use optimization algorithms that minimize the running time of a query subject to a certain budget limit. The dual problem, i.e., minimizing the cost incurred subject to a limit on the running time of the query, is also interesting. Moreover, the response time of a web service, or the QoS offered by a web service, may be a function of how much money is paid per invocation.

- Caching of web service results at the WSMS may lead to significant speedups in query processing. Extending the query optimization algorithms to incorporate caching is an important direction for future work.

Acknowledgements

We are grateful to Gene Pang for help with implementation of the prototype, and to Mike Abbott, Shivnath Babu, and David DeWitt for helpful discussions and suggestions.

9. REFERENCES

[1] S. Abiteboul et al. Lazy query evaluation for active XML. In *Proc. of the 2004 ACM SIGMOD Intl. Conf. on Management of Data*, pages 227–238, 2004.

[2] Apache Axis. http://ws.apache.org/axis/.

[3] S. Babu et al. Adaptive ordering of pipelined stream filters. In *Proc. of the 2004 ACM SIGMOD Intl. Conf. on Management of Data*, pages 407–418, 2004.

[4] Business Process Execution Language for Web Services. ftp://www6.software.ibm.com/software/developer/library/ws-bpel.pdf.

[5] N. Bruno, S. Chaudhuri, and L. Gravano. STHoles: A multidimensional workload-aware histogram. In *Proc. of the 2001 ACM SIGMOD Intl. Conf. on Management of Data*, pages 211–222, 2001.

[6] J. Burge, K. Munagala, and U. Srivastava. Ordering pipelined operators with precedence constraints. Available at http://dbpubs.stanford.edu/pub/2005-40.

[7] F. Casati and U. Dayal, editors. *Special Issue on Web Services, IEEE Data Eng. Bull., 25(4)*, 2002.

[8] S. Chaudhuri and K. Shim. Optimization of queries with user-defined predicates. *ACM Trans. on Database Systems*, 24(2):177–228, 1999.

[9] A. Condon, A. Deshpande, L. Hellerstein, and N. Wu. Flow algorithms for two pipelined filter ordering problems. In *Proc. of the 2006 ACM Symp. on Principles of Database Systems*, 2006. To appear.

[10] D. DeWitt et al. The Gamma Database Machine Project. *IEEE Trans. on Knowledge and Data Engineering*, 2(1):44–62, 1990.

[11] L. Ding and E. Rundensteiner. Evaluating window joins over punctuated streams. In *Proceedings of the 2004 ACM Conf. on Information and Knowledge Management*, pages 98–107, 2004.

[12] D. Florescu, A. Grünhagen, and D. Kossmann. XL: A platform for web services. In *Proc. First Biennial Conf. on Innovative Data Systems Research (CIDR)*, 2003.

[13] D. Florescu, A. Levy, I. Manolescu, and D. Suciu. Query optimization in the presence of limited access patterns. In *Proc. of the 1999 ACM SIGMOD Intl. Conf. on Management of Data*, pages 311–322, 1999.

[14] H. Garcia-Molina et al. The TSIMMIS approach to mediation: Data models and languages. *Journal of Intelligent Information Systems*, 8(2):117–132, 1997.

[15] R. Goldman and J. Widom. WSQ/DSQ: A practical approach for combined querying of databases and the web. In *Proc. of the 2000 ACM SIGMOD Intl. Conf. on Management of Data*, pages 285–296, 2000.

[16] J. Hellerstein and M. Stonebraker. Predicate migration: Optimizing queries with expensive predicates. In *Proc. of the 1993 ACM SIGMOD Intl. Conf. on Management of Data*, pages 267–276, 1993.

[17] W. Hong and M. Stonebraker. Optimization of parallel query execution plans in XPRS. In *Proceedings of the First Intl. Conf. on Parallel and Distributed Information Systems*, pages 218–225, 1991.

[18] T. Ibaraki and T. Kameda. On the optimal nesting order for computing N-relational joins. *TODS*, 9(3):482–502, 1984.

[19] Z. Ives, A. Halevy, and D. Weld. Adapting to source properties in processing data integration queries. In *Proc. of the 2004 ACM SIGMOD Intl. Conf. on Management of Data*, pages 395–406, 2004.

[20] M. Kodialam. The throughput of sequential testing. In *Integer Programming and Combinatorial Optimization*, pages 280–292, 2001.

[21] I. Manolescu, L. Bouganim, F. Fabret, and E. Simon. Efficient querying of distributed resources in mediator systems. In *CoopIS/DOA/ODBASE*, pages 468–485, 2002.

[22] R. Miller, editor. *Special Issue on Integration Management, IEEE Data Eng. Bull., 25(3)*, 2002.

[23] K. Munagala, U. Srivastava, and J. Burge. Ordering filters with precedence constraints, 2005. In preparation.

[24] M. Ouzzani and A. Bouguettaya. Efficient access to web services. *IEEE Internet Computing*, 8(2):34–44, 2004.

[25] M. Ouzzani and A. Bouguettaya. Query processing and optimization on the web. *Distributed and Parallel Databases*, 15(3):187–218, 2004.

[26] M. Ozsu and P. Valduriez. *Principles of distributed database systems*. Prentice-Hall, Inc., 1991.

[27] Java Reflection API. http://java.sun.com/docs/books/tutorial/reflect/.

[28] M. Roth and P. Schwarz. 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*, pages 266–275, 1997.

[29] F. Tian and D. DeWitt. Tuple routing strategies for distributed Eddies. In *Proc. of the 2003 Intl. Conf. on Very Large Data Bases*, pages 333–344, 2003.

[30] Apache Tomcat. http://tomcat.apache.org/.

[31] T. Urhan and M.J. Franklin. XJoin: A reactively-scheduled pipelined join operator. *IEEE Data Eng. Bull.*, 23(2):27–33, 2000.

[32] S. Viglas, J. F. Naughton, and J. Burger. Maximizing the output rate of multi-join queries over streaming information sources. In *Proc. of the 2003 Intl. Conf. on Very Large Data Bases*, pages 285–296, 2003.

[33] Web Services, 2002. http://www.w3c.org/2002/ws.

[34] Web Services Description Language. http://www.w3.org/TR/wsdl.

[35] V. Zadorozhny et al. Efficient evaluation of queries in a mediator for websources. In *Proc. of the 2002 ACM SIGMOD Intl. Conf. on Management of Data*, pages 85–96, 2002.