Searching and Querying Wide-Area Distributed Collections *

M. Franklin[2] G. Mihaila[3] L. Raschid[1] T. Urhan[1] M.E. Vidal[1]
V. Zadorozhny[1]

[1]University of Maryland, College Park, MD 20742
[2]University of California, Berkeley, CA, 94720-1776
[3]University of Toronto, Toronto, Ontario M5S 1A1 Canada

Abstract

The rapid proliferation of widely-distributed data and document collections raises the need for wrapper/mediator architectures that can handle the challenges of wide area query processing. Traditional query and search techniques do not scale to large numbers of repositories and cannot cope with the unpredictable performance and (un)availability of access to such repositories. Research at the University of Maryland is aimed at addressing the following challenges:

- Query planning for wide area networks: We describe Web query optimization techniques that use a Web-Wrapper cost model (WCM) and WebPT - a tool to predict response times from WWW sources.
- Coping with unexpected delays: *Query Scrambling* is a reactive query execution scheme that adapts the query plan in response to runtime delays. *XJoin* is a small footprint, fully pipelinable join operator that automatically adjusts the flow of tuples during query execution.
- Planning with alternate sources: We investigate strategies for chosing among multiple alternative data sources, and techniques to adjust these decisions when severe delays are encountered.
- Source publishing and selection: We describe how sources can be published on the WWW using XML, and we investigate source selection using content and quality metadata.

Acknowledgements: We are very grateful to Laura Bright and Tao Zhan for their research and programming support.

1 Introduction

The rapid growth of the Internet and Intranets, vendor support of database interoperability protocols such as JDBC [26], OLE/DB [6, 14], etc., and the emergence of XML to facilitate the exchange of semi-structured data via the WWW, has dramatically increased the number of Web accessible data sources, *WebSources*. Wrapper/mediator architectures [39] that are able to handle query processing with heterogeneous sources have been developed in the following projects: TSIMMIS/RQDL [42, 30, 38], Garlic [25, 23], DISCO [17, 29, 35, 5], and Information Manifold (IM) [22].

These architectures have to be tailored for query processing with large numbers of sources in a dynamic wide area environment. Traditional query and search techniques do not scale to large numbers of repositories and cannot cope with the unpredictable performance and (un)availability of access to such repositories. Research at the University of Maryland is aimed at these problems and we report on our results in addressing the following two challenges:

Query planning for wide area networks: Mediators must respect the limited capability of Web accessible *WebSources* when generating plans. Both the capability and the cost of the queries submitted to these WebSources must be considered in obtaining a good plan. Providing costs for accessing WebSources is complicated since these sources are autonomous, and details about their implementation is unknown. Further loads on the WebSource and the network may affect the query processing costs. We have developed Web query optimization techniques that use a WebWrapper cost model (WCM) and WebPT - a tool to predict response times from WebSources.

Coping with unexpected delays: Traditional distributed query processing technology performs poorly in the wide-area environment because unexpected delays encountered during query execution *directly* increases the query response time. The apparent randomness of such delays in the wide-area environment makes planning for them during query optimization nearly impossible. We have developed two techniques to cope with runtime problems. *Query Scrambling* is a reactive query execution scheme that adapts the query plan in response to delays that are detected at runtime. Query Scrambling has been shown to be highly effective in coping with initial delays. *XJoin* is a small footprint, fully pipelinable join operator that automatically adjusts the flow of tuples during query execution in response to all types of delay, including slow delivery and bursty arrival rates. XJoin focuses on streaming answer tuples incrementally to the users as quickly as possible, rather than on optimizing the delivery of the last tuple.

In addition to our research on these two challenges to support efficient query execution, there are two other tasks that are currently being investigated. The first task con-

*This research has been partially supported by the Defense Advanced Research Project Agency under grant 01-5-28838; the National Science Foundation under grant IRI9630102 and IRI9501353 and by CONICIT, Venezuela.

First Russian National Conference on
DIGITAL LIBRARIES:
ADVANCED METHODS AND TECHNOLOGIES,
DIGITAL COLLECTIONS
October 19 - 21, 1999, Saint-Petersburg, Russia

siders the situation where alternate sources may be available. Strategies for chosing among multiple alternative data sources, and techniques to adjust these decisions when severe delays are encountered will be discussed. Finally, we address the problem of using the WWW and XML to publish and locate sources, and their content and quality metadata.

In the next section, we briefly introduce our wrapper mediator architecture. We then discuss our research in query planning for WebSources; coping with unexpected delays; query processing with alternate sources; and source publishing and selection using content and quality data described in XML. Section 7 concludes.

2 Architecture

Figure 1 presents our wrapper mediator architecture. The mediator is an extension of the Predator ORDBMS [31, 32] and our mediator uses the relational data model. The shaded modules represent extended Predator modules and the unshaded ones are new modules that have been developed to support the mediator.

WebWrappers [9] are built to reflect the limited capability of *WebSources*. The capabilities of the sources are reflected in the forms-based interfaces that accept queries on the WWW. A *WebWrapper* utilizes both simple and complex extractors [9]. Typically, a simple extractor is constructed corresponding to the format of some HTML or XML document, and extracts answers from it. A complex extractor may use the output of one or more extractors, where each extractor provides a subset of values used by the complex extractor. Complex extractors typically access multiple documents where each document may have links to other documents. The *WebWrapper* provides relevant statistics and costs (delays) for the *WebSource*. To do so it uses a *WebWrapper Cost Model (WCM)*, and a *Web Prediction Tool (WebPT)* that can estimate response times (delays) for *WebSources.*

A *WebWrapper* Query Broker provides interoperability between the Predator engine (in C++) and *WebWrappers* (in Java) a la CORBA. The Predator evaluation engine was extended with several operators such as the *external scan* and *dependent join* operator [13, 18] to implement the (limited) query processing of the *WebSources.*

A Web query optimizer (WQO) is responsible for the task of planning and query optimization with limited capability sources and it uses a tool for capability based rewriting (CBR Tool). It also has the responsibility to choose particular implementations of wrapper queries to be evaluated at the WebSources. A Query Scrambling enabled optimizer has the responsibility of re-planning when unexpected delays are encountered. The Predator evaluation engine has also been extended with XJoin, a fully pipelined join operator.

The Web query optimizer, the WebWrapper cost model and the WebPT, the scrambling enabled optimizer and the XJoin operator will be discussed in detail in this paper.

3 Query Planning for WebSources

A Web query optimizer for WebSources must be able to produce good query execution plans that respect the limited capabilities of WebSources and reflect the often unpredictable costs of query execution. In this subsection, we briefly describe a cost model for WebSources, and a tool for predicting response times, WebPT. We then briefly review the functionality of the Web query optimizer (WQO).

3.1 Cost Model for WebSources

Providing a cost model for WebSources is difficult since sources are autonomous, and their implementation details are unknown and may change. There is also variability in a wide area environment. We use query feedback from submitting queries to WebSources to construct a cost model.

A *WebWrapper Cost Model (WCM)* maintains several statistics for a source. One statistic is the *result cardinality*, or the number of tuples returned by an extractor, when a query is submitted to a WebSource. One of our observations is that there could be some skew in this cardinality and this skew may depend on the specific bindings in the query that is submitted to the WebSource. A second statistic is the number of page access by a simple (or complex) extractor, to extract all the data for some query. This number may also be affected by the specific binding. A third statistic is the amount (quantity) of data that is downloaded by an extractor. Typically, when a query is submitted to a WebSource, an HTML page is constructed corresponding to the answer, or a part of the answer, if the result cardinality is very large. This page may have links to other pages, which affects the number of pages that are accessed. All three statistics impact the cost of a mediator query and are provided to the Web Query Optimizer to chose a good plan.

A *WebWrapper* should also provide estimates of the response time (delay) of accessing a *WebSource.* Further, this time may be decomposed into *Remote_Cost* and *Download_Cost.* The *Remote_Cost* is the time elapsed after the source accepts the query and when it begins returning data and the *Download_Cost* is the actual time to download the answers. Typically, the Remote_Cost depends on the result cardinality, and the Download_Cost depends on the amount of data that is downloaded. There are other factors that affect the cost in a wide area network and they are discussed shortly. Consequently, there may be considerable variance in the response times of queries to a *WebSource* and the WCM makes use of a tool to predict response times. This tool, the WebPT, is discussed in the next section.

Our research reported in [10] analyzes the factors that affect the response time. Figure 2 plots query processing times at the WebSource, or Remote_Cost, for the ACM digital library (ACM) [1], and the California Campaign Contribution database (CA) [12]. The Figure plots the time versus the result cardinality, for random queries submitted over several months in 1999. The linear equation plotted in the Figure represents the results of a linear regression of the time versus the cardinality. Figure 3 plots the time to download data from the California Campaign Contribution database (CA) [12] and FishBase (FB) [16]. The linear equation is the result of the linear regression of time versus file size. As can be observed, there is considerable variance in these costs, and this behavior is typical of a wide area network.

3.2 WebPT - A Tool to Predict Response Times from WebSources

In the previous subsection, experimental data collected from WebSources indicates that result cardinality and file size were not the sole factors that affect remote query processing costs and time to download data from WebSources. There is little knowledge about the impact that *dimensions* such as *Time of day*, *Day*, etc., can have on response times. Our hypothesis is that these dimensions can be used to reflect usage or load, on the WebSource and the network, and thus, can help predict response times. There has been some research on learning traffic patterns for the Internet [34]. The *Net-*

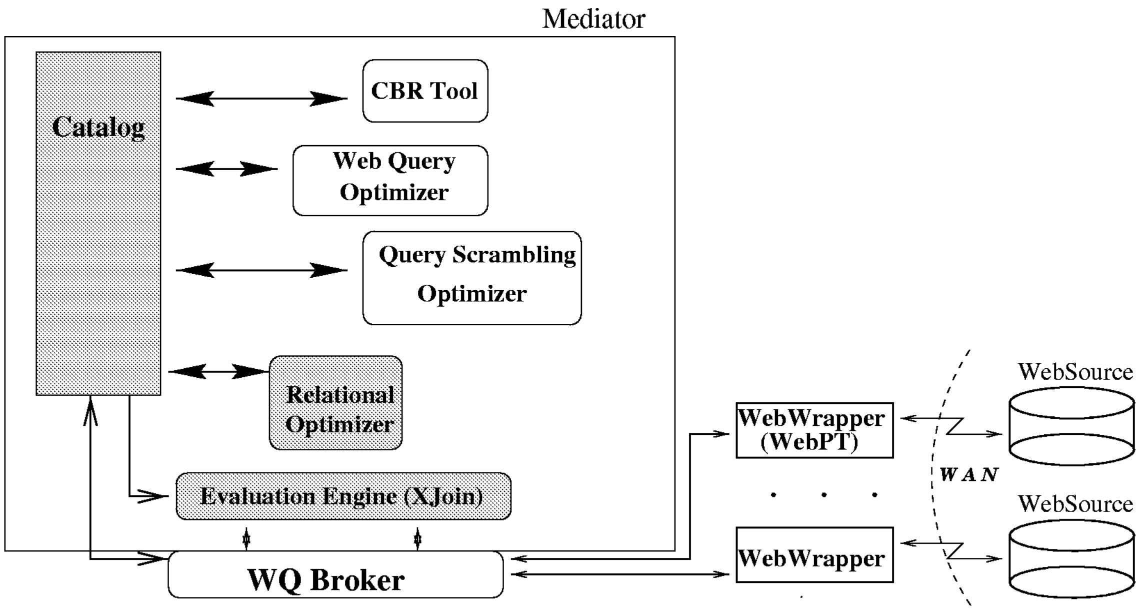


Fig. 1: Mediator Wrapper Architecture for *WebSources*

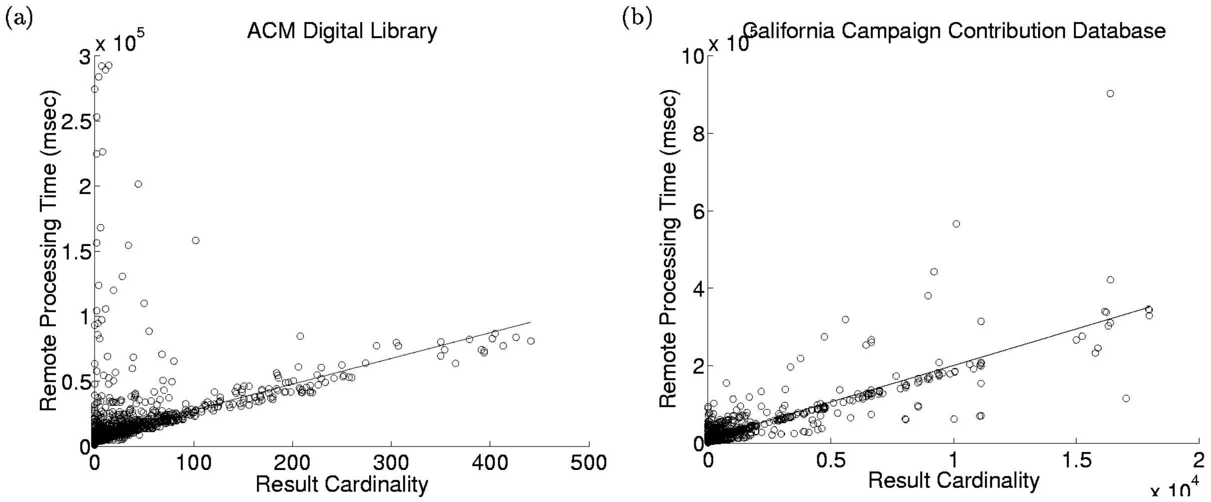


Fig. 2: Remote processing time as a function of result cardinality

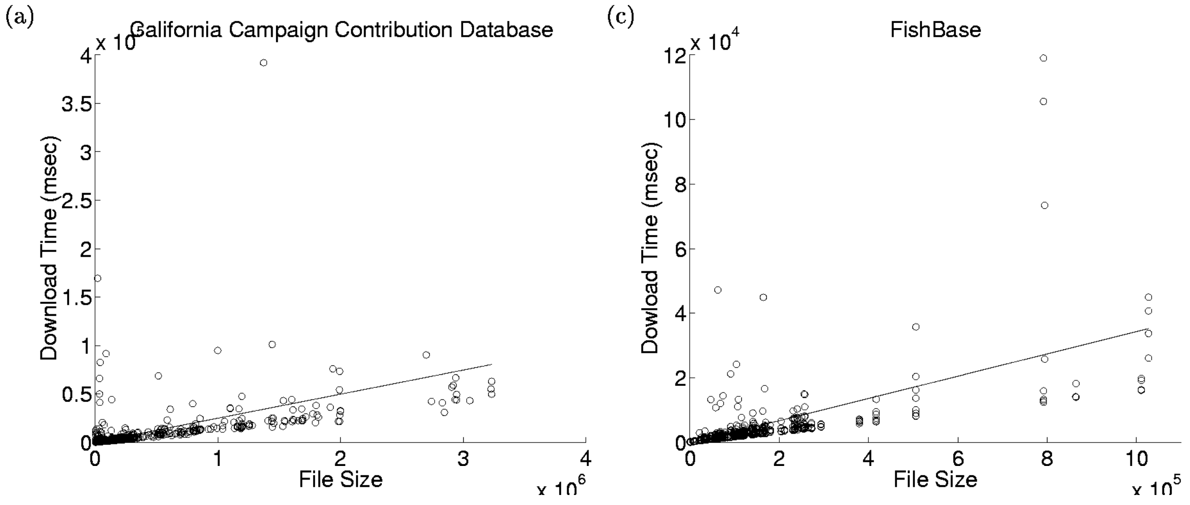


Fig. 3: Download time as a function of file size

work Weather Service, NWS [41], is a general facility that provides dynamic resource performance forecasts for wide area networks. However, these systems do not use learning from WebSources to predict response times.

We have developed a Web Prediction Tool (WebPT), that is based on learning using query feedback from WebSources [20]. The WebPT uses dimensions Time of day, Day, Quantity of data (file size), result cardinality, etc., to learn response times from a particular WebSource, and to predict the expected response time (delay) for some query. WebPT learning is similar to CART [8], except that it uses simpler techniques (compared to regression) to split the query feedback along each of the relevant dimensions, during the learning process.

Experiment data (query feedback from submitting queries) was collected from several sources, and those dimensions that were significant in estimating the response time were determined. The WebPT has been trained on the collected data, to use the significant dimensions to predict the response time, as well as a confidence in the prediction. In those situations where the Remote_Cost and Download_Cost are both significant, these times will be learned and predicted independently. We describe the WebPT learning algorithms, and report on the WebPT learning for WebSources in [20]. Our research shows two significant results with respect to WebPT learning. The first result is that the WebPT does learn, and that as it is trained, the (cumulative) error decreases, and the confidence in the prediction increases. The second result is that we can improve the quality of learning by tuning the WebPT features. These features include training the WebPT on the logarithm of the input data; including significant dimensions in the WebPT; and changing the ordering of significant dimensions in the WebPT.

We have compared WebPT learning with the more traditional Neural Network (NN) learning in [11]. WebPT learning is always *online*, i.e., it learns from each new query feedback. NN training can be online (per-pattern learning), which is time consuming and can be very sensitive to the choice of training parameters. The more common and robust learning is offline batch learning (per-epoch). This is less suitable for a Web environment, with large variances in response time, where there may be insufficient training data, and where the environment is dynamic and should be continually monitored. We compared the WebPT learning with both types of NN learning, in a number of experiments where the WebPT and the online (per-pattern) and offline (batch) NN were trained and tested on the same data.

Figure 4a compares the WebPT with the offline (batch) NN, for experimental data from which outliers (noise) have been eliminated, and Figure 4b similarly compares the offline and online NN. The performance of the WebPT is very comparable to the performance of the offline (batch) NN.

The results of our study indicate that the ease of training the WebPT makes it preferable, compared to the per-pattern NN, for online monitoring and prediction. We can infer that for many WebSources, corresponding to an unstable environment, the more sophisticated NN learning does not provide much of an advantage. From further study, reported in [10], we conclude that both the WebPT and the more sophisticated NN learning are useful in constructing a WebWrapper Cost Model for the dynamic Web environment.

The WebPT approach has some advantages over other learning based techniques such as regression techniques or neural networks. One advantage is the simplicity of the WebPT prediction model and the flexibility provided by the WebPT to manipulate the parameters that control learning. A second advantage is that the WebPT can also learn when the dimensions may not be significant in predicting the response time, and where a (lack of) confidence in the WebPT prediction reflects the unpredictable nature of prediction for WebSources. This knowledge can be used to tune a Web query optimizer and to interpret its choices.

3.3 A Web Query Optimizer

A Web query optimizer (WQO) is responsible for the task of planning and query optimization with limited capability WebSources. It uses a tool for capability based rewriting (CBR Tool) and an extended randomized relational optimizer to produce good plans.

In a pre-optimization phase, the CBR Tool uses *WebSource limited capability descriptions* to produce (multiple)

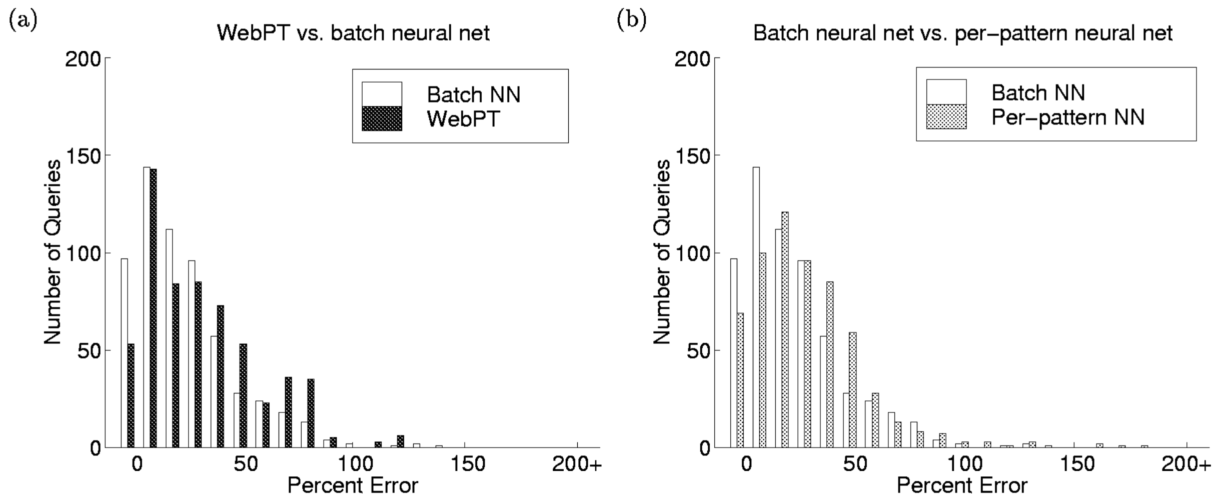


Fig. 4: Comparison of the WebPT with the Offline and Online NN for OZ Without Noise

pre-plan(s) for a mediator query. A pre-plan consists of (possibly ordered) subgoals to be executed in the *WebSources* or the mediator. The pre-plan identifies (1) ordering among subgoals, (2) relevant *WebSource* implementations (WSI) to evaluate subgoals in the *WebSources*, and (3) restrictions on queries submitted to the wrapper for the corresponding *Web Source*. A Web Query Optimizer (WQO) uses a two-fold optimization on pre-plans to obtain a good plan. In the first stage, the WQO explores the search space of specific *WebSource implementations* (WSI), corresponding to wrapper calls, to choose a good WSI. In the second stage, the WQO uses a (randomized) relational optimizer to re-order mediator subgoals following traditional (cost-based) optimization strategies. The pre-plan knowledge (ordering and binding restrictions on the WSIs) is used by the Web optimizer to drive the relational optimizer, so that the latter respects the pre-plan.

Our approach has the advantage that the WQO uses the pre-plan to actively explore the search space of WSI for mediator subgoals. While doing so, the WQO may exploit knowledge of trade-offs particular to query evaluation with *WebSources* to choose specific WSI that produce good plans. The WQO may explore specific evaluation strategies such as top-down versus bottom-up evaluation of mediator subgoals corresponding to different WSI that are chosen. Top-down evaluation is usually associated with an ordering of subgoals as is common in *WebSources*. The WQO choice of WSI depends on the space of WSI corresponding to some subgoal, and the cost of *WebSource* queries. The cost depends on various statistics and measured execution times that may be provided to the WQO by a WebWrapper. The space of WSIs for some subgoal includes "atomic" or "composed" solutions which are particular to *WebSources*, and which also impact the cost of the plan. Typically, a "composed" solution increases the number of wrapper calls. After the WSI are chosen, the WQO would generate a good plan for this choice of WSI.

Consider the ACM digital library (ACM DL) *WebSource* [24], and mediator schemas, which are relational. We characterize the limited query capability as an *input-output* relationship IOR, *Input* $\rightarrow$ *Output*, on a relation, where *Input* are the set of attributes that must be bound and *Output* are the set of projected attributes. The mediator schema and IORs for the ACM DL are as follows:

Paper(1stAuthor, Title, PaperSrc, PaperId,
  Keywords,Publisher)
    ior_1: {1stAuthor} $\rightarrow$
      {Title, PaperId, PaperSrc, Keywords}
    ior_3: {1stAuthor} $\rightarrow$
      {Title, PaperId, Keywords,Publisher}
    ior_4: {PaperId} $\rightarrow$ {PaperSrc}
CoAuthor (PaperI d, CoAuthor)
    ior_2: {PaperId} $\rightarrow$ {CoAuthor}

We consider the following mediator query Q1:

Select Title, PaperSrc, Coauthor
From Paper, CoAuthor
Where 1stAuthor="franklin" and
    CoAuthor.PaperId=Paper.PaperId

Depending on the WQO choice of WSI for the query on Paper, the relational optimizer can have different options for plan generation. If the WQO chooses an atomic WSI, WSI_1, corresponding to ior_1, then the relational optimizer will only be able to generate plan P1 of Figure 5, which has a single dependent join. If the WQO choice is the composition of WSI_2 and WSI_3, corresponding to ior_3 and ior_4, then, the relational optimizer can produce two plans P2 and P3 of Figure 5. Note that there are two external scan operators on the mediator relation Paper in both these plans.

The time-to-first-tuple and time-to-last-tuple behavior for a set of random executions of query Q1, for each of the above plans, is in Figure 6a and b, respectively. This figure indicates that in a volatile environment of *WebSources*, execution times vary significantly. However, there are consistent trends that are observed over all executions. To prove that these trends hold, we present quantile plots of time-to-first-tuple and time-to-last-tuple for all executions of Figure 6 in Figure 7a and b.

As seen in Figure 7b, plans P1 and P3 are statistically comparable using time-to-last-tuple. These two plans perform better than plan P2 in all cases, for time-to-last-tuple. To explain informally, (many of) the selected papers had multiple co-authors, and plan P2 performed a costly download of the paper multiple times, once for each co-author. Thus, in this case, the *conservative* WQO choice of an atomic WSI, e.g., in plan P1, may have been safer, since it avoids the risk of choosing the costly plan P2. In contrast, plans

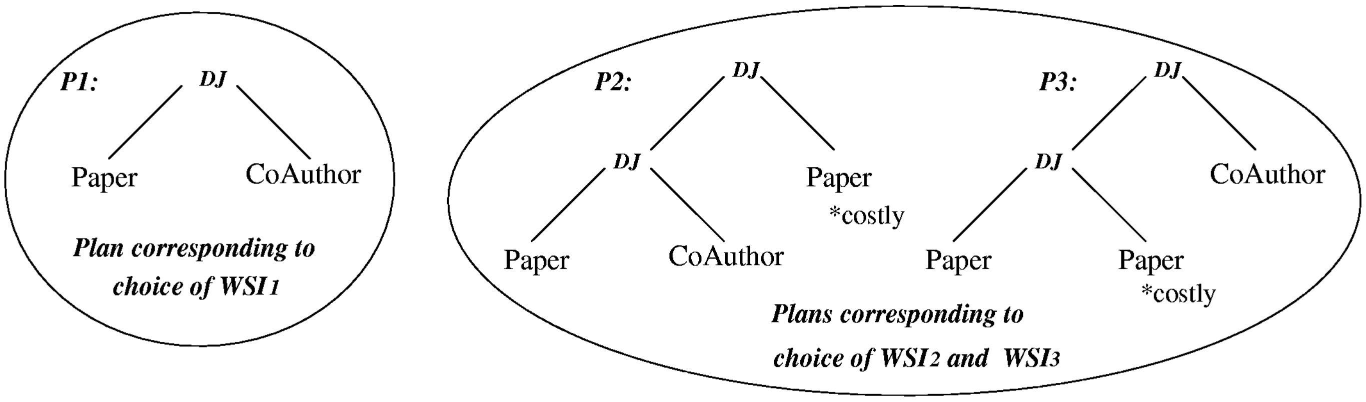

P1:
DJ
Paper
CoAuthor
Plan corresponding to choice of WSI1
P2:
DJ
DJ
Paper
*costly
Paper
CoAuthor
P3:
DJ
DJ
CoAuthor
Paper
Paper
*costly
Plans corresponding to choice of WSI2 and WSI3

Fig. 5: Plans for query Q1 in the ACM Digital Library *WebSource*

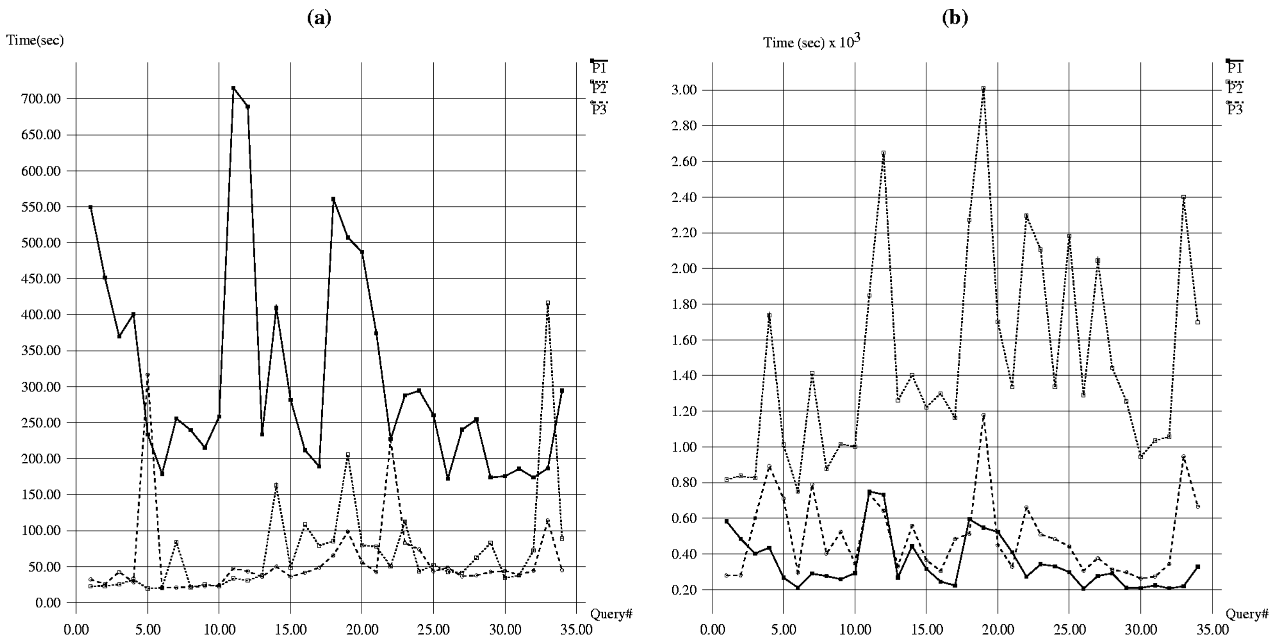
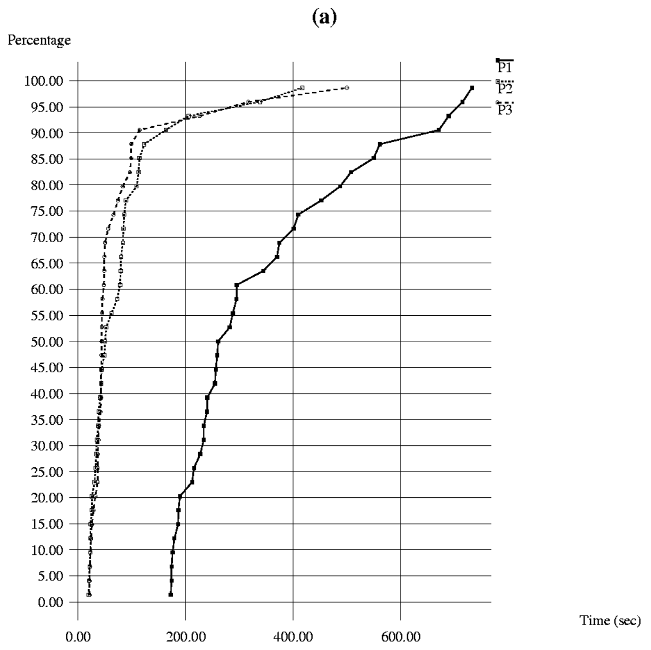


Fig. 6: Time-to-first-tuple (a) and time-to-last-tuple (b) for query Q1



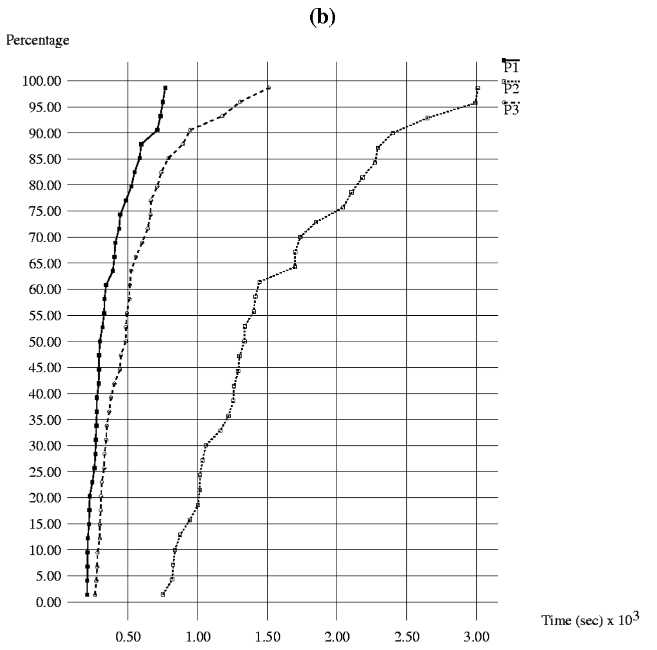


Fig. 7: Time-to-first-tuple (a) and time-to-last-tuple (b) quantile plots for query Q1

P1 and P3 only downloaded each paper once, independent of the number of co-authors. Figure 7a shows the time-to-first for all the plans. We observe that in general, a composed WSI, as in plans P2 or P3, typically has less initial delay, and provides smoother delivery of data. This is important in a Web environment when long delays are not uncommon. We note that further testing and statistical analysis is needed to completely validate our observations on the query optimizer and to tune its performance.

4 Coping with Unreliable Sources

The preceding sections described approaches for producing efficient query execution plans when accessing wrapped data sources. Such plans are generated using knowledge of the *typical* behavior of the sources, and thus, will be good plans for many situations. While such an approach on its own would be sufficient for a stable environment such as a tightly-coupled cluster or local area network, the wide-area environment provides additional complications that must be addressed in order to provide responsive data access. In particular, a major challenge that must be addressed for wide-area distributed information systems is that of *response-time variability.* Data access over wide-area networks involves a large number of remote data sources, intermediate sites, and communications links, all of which are vulnerable to congestion and failures. Such problems can introduce significant and unpredictable *delays* in the access of information from remote sources.

Traditional distributed query processing technology performs poorly in the wide-area environment because unexpected delays encountered during a query execution *directly* increase the query response time. Query execution plans are generated statically, based on a set of assumptions about the costs of performing various operations and the costs of obtaining data. The execution of a statically optimized query plan is likely to be sub-optimal in the presence of unexpected response time problems that arise during the query *runtime.* In the worst case, a query execution may be blocked for an arbitrarily long time if needed data fail to arrive from remote data sources across a wide-area network. The apparent randomness of such delays in the wide-area environment makes planning for them during query optimization nearly impossible.

We have identified three types of delays that can arise when requesting data from remote sources: 1) *Initial delay* is a longer than expected lag between the time that a source is initially contacted and the time that the *first* tuple from that source arrives at the query site (i.e., mediator). Such delays can arise, for example, from difficulty in establishing a connection with the source, or because of some unexpected start-up costs involved with running the subquery sent to that source; 2) *Slow delivery* is said to occur when data arrives at a regular rate that is much slower than expected. Such problems can arise, for example, if an alternative communication path is used; 3) *Bursty arrival*, where data arrive at a fluctuating rate, is likely the most typical of the three problems. Bursty arrival can occur due to congestion or errors at any point in the path of data from the source to the query site, including at the source itself.

We have developed two techniques to cope with these three types of runtime problems. *Query Scrambling* is a reactive query execution scheme that adapts the query plan in response to delays that are detected at runtime. Query Scrambling has been shown to be highly effective in coping with the initial delays. *XJoin* is a small footprint, fully pipelinable join operator that automatically adjusts the flow of tuples during query execution in response to all three types of delays. XJoin focuses on streaming answer tuples incrementally to the users as quickly as possible, rather than on optimizing the delivery of the last tuple. These two techniques are discussed in the following sections.

It should be noted that the definitions of the above three problems implicitly assume that the requested data *eventually* arrive at the query site. Of course, if needed data is missing, the query cannot be correctly answered as originally posed. There are several alternatives in such cases. First, as we will discuss in Section 5, it may be possible to obtain missing data from alternative sources. Second, it may be necessary to change the query being answered in some way. One such change is to focus on delivering at least part of the answer as quickly as possible. This is the approach taken by the XJoin operator described in Section 4.2. Another approach is to modify the query in some way so that the answer can be provided with the data that has arrived. One example of such an approach is called Parachute Queries [2, 7]. Such *semantic* approaches, however, are beyond the scope of our current work.

4.1 Query Scrambling

We have developed *Query Scrambling* [4, 3, 37] to address the issue of unpredictable delays in the wide-area environment. Query Scrambling reacts to unexpected delays by modifying, *on-the-fly*, the execution plan of a query so that progress can be made on other parts of the plan. In other words, rather than simply stalling for delayed data to arrive, as would happen in a typical, static scheduling scheme, query scrambling attempts to *hide* unexpected delays by performing other useful work.

Query Scrambling reacts to delays in receiving data from remote data sources in two ways (referred to as Phase I and Phase II respectively in [4]):

- *Scrambling Rescheduling* - the execution plan of a query can be dynamically rescheduled when a delay is detected. In this case, the basic shape of the original query plan generated by the optimizer remains unchanged.
- *Operator Synthesis* - new operators (e.g., a join between two relations that were not directly joined in the original plan) can be created when there are no other operators that can execute. In this case, the shape of the query plan can be significantly modified through the addition, removal and/or re-arrangement of query operators.

Scrambling rescheduling works by dynamically creating *additional parallelism*, beyond what may have already been compiled into a query execution plan. Through this parallelism, scrambling is able to perform useful work, such as obtaining other data from remote sources or performing query operations, while delays are experienced on other parts of the query plan. In order to implement rescheduling, the run-time system must sometimes introduce materializations of intermediate results and base data into the query execution plan. For this and other reasons, scrambling may increase the total *cost* of query execution in terms of network contention, memory usage, and/or disk I/O. The potential costs caused by rescheduling raise some basic performance trade-offs between extensive use of local resources (at the mediator site) versus stalling on delayed data with no extra expense than the costs of delays.

Operator Synthesis is a somewhat more drastic response to delays that is invoked only after the options for productive rescheduling of existing operators have been exhausted. In contrast to rescheduling, Operator Synthesis creates new operators that were not present in the original query plan (e.g., a join between two data sources that were not directly joined in the original plan). Because the operations that are created in this manner were not chosen by the optimizer when the original query plan was generated, it is possible that these operations may entail a significant amount of additional work. If the newly created operators are too expensive, query scrambling could potentially result in a significant degradation in performance.

As described in the original paper [4], both of these phases where *heuristic-driven*. That is, the algorithm was specified as a set of heuristic rules that were activated as delays in obtaining remote data were detected. The heuristics described in that paper were shown to be very effective at hiding initial delays in some situations, but they were also shown to be prone to making poor scrambling decisions in other cases. In some cases, the proposed heuristics could result in performance that is worse than simply waiting for the delayed data to arrive. Thus, it became clear that it would be useful to exploit query optimization to aid in making intelligent scrambling choices.

In [37] we presented three different approaches for extending a query optimizer for scrambling. Two of the approaches use an objective function based on response time, while the other approach used a more traditional optimizer based on total work. A key insight behind this work is that a response time-based optimizer, if given an estimate of the expected delay duration, can automatically schedule the accesses to delayed data at the proper place in the plan execution. Unfortunately, the current state of delay estimation for wide-area data access is quite poor (e.g., note the loading time estimates that appear at the bottom of your browser), so we developed and studied alternative ways to provide delay estimates and to deal with inaccurate ones.

A performance study using queries from the TPC-D benchmark (reported in [37]), showed that optimizer-based scrambling can effectively hide initial delays and outlined the fundamental tradeoffs between risk aversion and effectiveness that arise in the absence of good predictions of expected delay durations. Our current work on query scrambling is focused on the integration of the approach into an existing database system. We have extended the PREDATOR Object-Relational database system [32] by converting its execution model to run in a thread-per-operator mode rather than as a thread-per-query, and have integrated our scrambling optimizer into the system.

4.2 XJoin - A Fully Pipelinable Join Operator

While query scrambling has been shown to be effective at hiding initial delays, it is less effective at dealing with the other two types of delays: slow delivery and bursty arrival. To cope with these additional problems, we have developed a complementary approach, based on a non-blocking join operator we call XJoin. XJoin extends the symmetric hash join (SHJ) [40] to use secondary storage, which allows it to be used with large inputs and to run concurrently with other query operators in a bushy query plan. Simply extending SHJ to use secondary storage, however, is insufficient for tolerating significant delays in receiving data from remote sources. For this reason, a key component of XJoin is a *reactively scheduled* background process, which opportunistically utilizes delays to produce more tuples earlier. In a recent paper [36], we have shown that by using XJoins it is possible to produce query execution plans that can better cope with data delivery problems and that can deliver initial results orders of magnitude faster than traditional techniques, with in many cases, little or no degradation in the time required to deliver the entire result.

The XJoin approach is based on two fundamental principles:

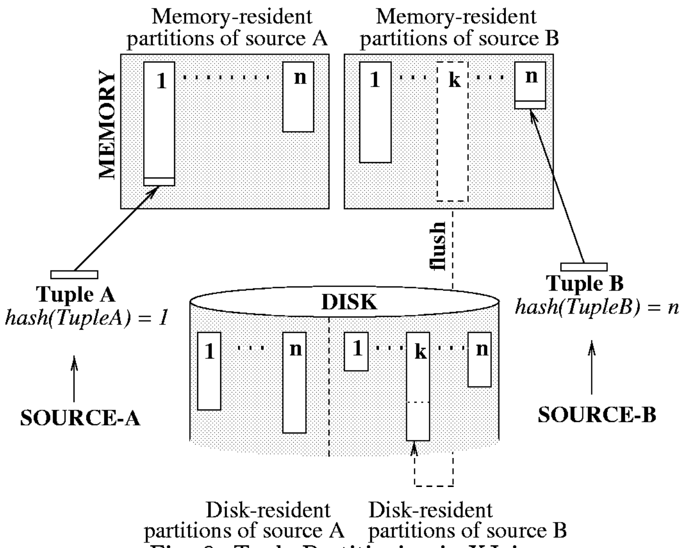
1. *It is optimized for producing results incrementally as they become available.* When used in a fully pipelined query plan, answer tuples can be returned to the user as soon as they are produced. The early delivery of initial answers can provide tremendous improvements in the responsiveness of the system. Furthermore, in many situations, users require only a small subset of the total query answer, so returning initial results quickly is the key to system usability.
2. *It allows progress to be made even when one or more sources experience delays.* There are two reasons for this. First, by using less memory XJoin allows for bushier query plans than are possible with other pipelined join methods. Thus, some parts of a query plan can continue while others are stalled waiting for input. This enables the query plan to make progress on other parts of a query plan even if some relations are temporarily unavailable. Second, by employing background processing on *previously received* tuples from both of its inputs, an XJoin operator can produce results even at times when both inputs are stalled simultaneously.

The symmetric hash join, on which XJoin is based, was aimed at addressing similar issues. As originally proposed, however, symmetric hash join requires that hash tables for both of its inputs be kept in main memory until all of the tuples have been received from both of its inputs. As a result, symmetric hash join cannot be used for joins with large inputs, and the ability to run multiple joins (e.g., in a bushy query plan) is severely limited. XJoin avoids these problems by allowing tuples from one or both of the inputs to be temporarily spooled to secondary storage. In a sense, XJoin provides for symmetric hash join, the same flexibility that the hybrid hash join provides for the classic hash join [33]. Not surprisingly, similarly to hybrid hash join it is based on partitioning.

XJoin splits both of its inputs into a number of partitions based on a hash function.[1] Each partition is composed of a *memory-resident* portion and a *disk-resident* portion. The memory-portion contains the tail (i.e., recently arrived tuples) of the partition, and the disk-resident portion contains the rest. The memory-resident portions are maintained as *hash tables* as in symmetric hash join. Each memory-resident portion (for *both* sources) has at least one block of memory reserved for it at all times. The remaining memory (if any) is divided evenly between the two sources and is used to allow the memory-resident portions to grow as tuples arrive from the sources.

When XJoin receives a tuple from one of its sources, it inserts the tuple into its corresponding partition, which is found by applying a hash function to its join attribute (Figure 8). When the memory becomes full, the tuples of the partition with the largest memory-resident portion are

[1]The number of partitions is determined by using the formula $\sqrt{F \times ||R||}$ where F is the "fudge" factor, and $||R||$ is the number of pages in the smaller input [33].


Memory-resident partitions of source A
Memory-resident partitions of source B
MEMORY
1
n
1
k
n
flush
Tuple A
hash(TupleA) = 1
Tuple B
hash(TupleB) = n
DISK
1
n
1
k
n
SOURCE-A
SOURCE-B
Disk-resident partitions of source A
Disk-resident partitions of source B
Fig. 8: Tuple Partitioning in XJoin

written to the disk. This grows the size of the disk-resident portion (e.g., partition k of source B in Figure 8). The memory-resident portion of that partition is then reduced to a single (initially, empty) block, and the remaining free blocks are made available for use by any of the partitions as new tuples arrive. The flushing process is repeated whenever the memory becomes full.

XJoin proceeds in three stages, each of which is performed by a separate thread. The first stage joins tuples in the memory resident portions of the partitions, acting similarly to the standard symmetric hash join. The second stage joins tuples from disk with tuples that have not yet been flushed to disk. The third stage is a clean-up stage, which performs any necessary matching to catch any results missed by the first two stages. The first and second stages run in an interleaved fashion — the second stage takes over when the first becomes blocked due to a lack of input. These stages are terminated after all input has been received, at which point the third stage is initiated.

The second stage is the key to XJoin's ability to cope with unexpected delays. Like the original symmetric join algorithm, the first stage of XJoin can tolerate the blocking of one of its two inputs, but if both inputs block, the first stage itself blocks. In such a case, the second stage will be resumed, and data that has previously been spilled to the disk for one input will be used to join with data that has arrived more recently from the other input. Thus, like query scrambling, the second stage of XJoin attempts to hide delays by performing other useful work.

The implementation details of XJoin, which are beyond the scope of this paper, are presented in [36]. There are some important issues that deserve mention, however. First, due to the interactions among the various stages of XJoin, if care is not taken, spurious duplicate result tuples can be produced. In order to prevent duplicates XJoin employs a fast, on-the-fly duplicate detection mechanism. The duplicate prevention mechanisms rely on timestamp values that are maintained by the XJoin operator itself. Secondly, the scheduling of the second stage must be done in a controlled manner, as the second stage can consume additional resources that can slow down normal processing if data arrival resumes. A simple scheme for controlling the execution of the second stage is presented in [36]. Finally, the overall performance of the operator can be improved by dedicating a small amount of additional memory for use as a "cache" during the second stage. When tuples are read from disk during the second stage, they can be read into this cache. The cache can then be probed using the disk resident tuples of the corresponding partition of the other input.

We have implemented XJoin in the PREDATOR Object-Relational DBMS, and compared its performance with that of hybrid hash join using real network traces. We performed a detailed experimental study, which investigated the performance of XJoin in the presence of different data delivery rates, memory sizes, and query complexity [36]. In all the cases studied, XJoin had much better (often by several orders of magnitude) interactive performance (i.e., in terms of producing the initial portions of the result) than hybrid hash join, and in most cases it performed better than hybrid hash join for the entire query, delivering even the final result tuple as fast or faster. These results indicate that XJoin is indeed an effective solution for providing fast query responses to users in the presence of slow and bursty remote sources.

5 Processing with Alternate Sources

Query Scrambling and XJoin provided solutions to deal with a variety of delays. Recall that query planning in the Web query optimizer choose the "best" WebSource to submit a query. If there are multiple alternative sources, then another option to hide network delays during the query processing is to choose among multiple alternative data sources. Informally, two sources are considered alternatives of each other if they provide *similar* data. In other words two sources are alternatives if the missing tuples from one source can be obtained from another one. The tuples may be missing either because they are delayed, or because one of the source does not have complete data. Notice that in the first case where the tuples are missing due to the delays, using an alternative source may improve the *query response*, whereas in the second case using an alternative source improves the *answer quality*. Depending on their contents and schemas three kinds of alternative sources are possible:

- **Replicas:** in this case the alternative relations have the same content. The unavailability of one relation may be hidden by the tuples from the other which may improve the query response.
- **Complements:** a relation may have missing tuples which can be supplied from an alternative source. It may be the case that none of the relations are complete, but collectively they cover all the domain.
- **Column-partial alternatives:** Alternatives may have similar tuples, however some columns may be missing from one source which is supplied by an alternative.

In this paper we focus on utilizing *replicas* in the presence of unpredictable delays in order to improve the response time. Replication can be used to increase availability and to allow clients to recover from delays by contacting available replicas. When a remote source (relation) is unavailable or slow, other replicas can be fetched instead to improve the response time. We consider *fetching strategies* for a system in which relations are replicated across a wide area network at different servers which are accessed by many clients.

We make the following assumptions about an execution scenario:

- A large client population may exist and compete for services and network bandwidth.
- Physical layout of the replicated tables and their implementation is unknown to the clients. A replica may be a base relation or a complex view at the remote site.
- The sources are assumed to be delayed if the arrival rate of its tuples is below a (perhaps user defined) threshold or when a significant delay occurs.

Processing replicas requires two separate mechanism. First a *Fetching Policy* is needed to drive the fetching of replicas. The fetching policy has to decide which replicas to fetch, in which order, and what to do when delays occur. Second, a *Merging Algorithm* is needed. Since we are dealing with replicas the main goal of the merging algorithm is duplicate elimination. Duplicates may arise if multiple replicas are fetched. In this subsection we will focus on fetching policies in the presence of delays. We also developed a pipelined duplicate elimination algorithm which is used to merge replicas, which is not considered in this paper.

5.1 Fetching policies for replicas

Consider a simple consumer/producer interpretation of the fetching process. The tuples produced by a source are received at the destination with a particular arrival rate AR (tuples/sec). There is also a processing rate PR characterizing the speed of processing of the tuples, in a corresponding query plan, or a consumer, at the destination. The problem of finding a fetching policy can be formulated in the following way: *Given a consumer C with a processing rate PR and a set of alternate sources with corresponding (estimated) arrival rates, find a subset of the alternative sources which should be fetched by C simultaneously.* Possible approaches to the solution of this problem are as follows:

1. Aggressive fetching: Fetch all the available replicas. Tuples are unioned (possibly using a pipelined duplicate elimination algorithm) as they arrive and output is produced at least as fast as the fastest source. In general we have the following obstacles which render an aggressive fetching policy inefficient:
   - *Large client population (scalability issue):* many clients opening all the sources at once will increase the average load on the servers. This will increase the response time perceived by all the clients.
   - *Network bottlenecks:* Even with one client, fetching multiple sources at the same time could create congestion in the network. For instance the client's network connection may become the bottleneck if its bandwidth is lower than that of the servers.
   - *Client processing:* even with one client and infinite network bandwidth the client processing (mainly in the form of duplicate elimination due to duplicate data arriving from sources) may become a bottleneck.
   - *Diminishing returns:* depending on whether sources send their tuples in the same order or not, fetching from multiple sources does not improve the arrival behavior perceived by the client too much. This is especially true if we focus on the last tuple response time.
2. Moderate fetching: Find a subset of the replicas, while minimizing the fetching cost. An optimal strategy in

fetching replicas should result from a cost-based optimization on the consumer side. The strategy should also respect the scalability issue. This approach requires further research to prove its feasibility, since the factors which influence the fetching cost are highly unpredictable.

3. Conservative fetching: Find a replica with a good (best) AR and utilize it while it provides a reasonable AR; replace it when the AR becomes low, or in extreme case is delayed. In the next subsection, we will further investigate this conservative policy.

5.2 Conservative fetching and cost-based replacement for replicas

A conservative fetching algorithm would behave as follows:

1) Compute an initial ordering of replicas.
2) Select the best available replica and prepare the query.
3) Open the replica and start reading all the tuples until EOF or there is significant delay.
4) If there is significant delay, repeat from step 2.

There are several questions that must be answered to obtain a usable algorithm with desirable properties. Decisions must be made on ordering replicas (step 1), detecting the timeout (step 3), etc. We discuss these issues next.

5.2.1 Computing the ordering of the replicas

The ordering of the replicas may be defined in the following manners:

- *Random ordering.*
- *Ordering based on statistics.* We assume some arrival rate statistics (for example, a maximum arrival rate AR_{max} for each source) is maintained. The sources are ordered in decreasing order of AR.
- *Ordering based on initial probing.* Probing queries are submitted to all remote sites to sample their arrival rates AR_{init}. The sources are ordered with respect to this AR_{init}.

If there are multiple clients, then contacting the replicas in the same order will tax some sources. For instance if the servers are picked in the decreasing order of AR_{max}, then the powerful sources are taxed too much, since all the client request will be made to the same server significantly increasing its load. This is especially important if there is some initial server overhead associated with processing client request. Even if there is no initial startup overhead, the data transfer rate from the first source may decrease with increasing number of clients. An added disadvantage is that other servers will be underutilized. Thus, there should be a way to balance the access to these sources.

5.2.2 Replica replacement strategy

The decision criteria to replace the current source with a new one is as follows: if the arrival rate of the current source falls below a threshold value, then select the "next" replica and proceed to step 3. Determine if the current, or new replica produce the next tuple, and drop the looser. The replacement cost can be estimated on the basis of such parameters as replacement overheads, cardinality of tuples to be delivered from the current source, and that can be expected from the next replica, AR of the current and new source, etc.

Another important issue is how to handle partial results:

- **Re-fetch approach:** Discard tuples from current replica, and re-fetch all of them from the new source. The advantage of this approach is its simplicity. The disadvantage is that the redundancy can depreciate possible speed-up.
- **Remainder query approach:** use tuples from current replica to generate a *remainder query*, and fetch only new tuples from the new replica. The problem then is how to generate the remainder query. Generating a reminder query may be easy when both replicas are ordered, assuming the replica allows the client to identify particular tuples to be downloaded. Another solution is to generate a selection predicate using knowledge of tuples obtained from the current replica. For example, one may generate a disjunction of equalities (Attr=Value). However, the size of the selection predicate may be huge. The replica should be able to process selection conditions with negation.

We illustrate the replica replacement condition in Figure 9, which plots time to obtain tuples versus cardinality of the resulting relation. Initially, a replica with arrival rate $AR1$ was selected. Suppose the arrival rate for this source has decreased to $AR1'$, which triggered the replacement condition check in Step 3. Suppose the number of tuples that have been received is equal to $Ccurr$. Assuming $Cfin$ is a cardinality of the entire result, the estimated time to obtain all tuples from the current replica is $T1$. Suppose $T2$ is the time to obtain all tuples from the next replica with arrival rate $AR2$. Then, the next replica should be chosen if $T2 < T1$.

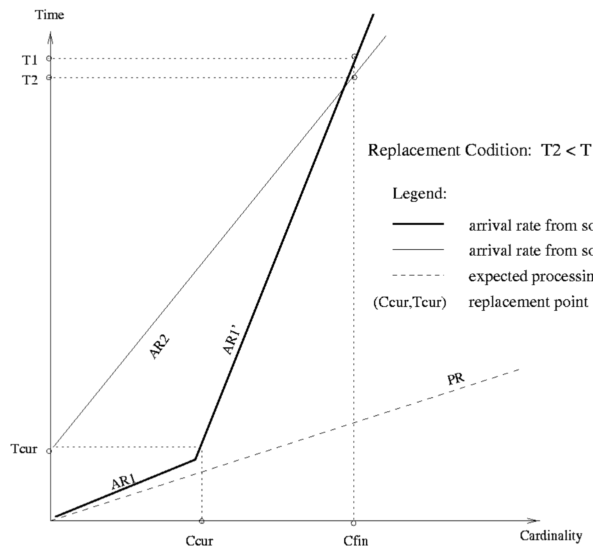


Fig. 9: Source replacement condition

6 Locating and Selecting Data Sources based on Content and Quality Descriptions

We are witnessing an increasing interest in using the WWW as a platform for publishing structured and semi-structured data, such as scientific datasets in various disciplines. For example, large collections of data about the environment are publicly available in online repositories [19]. In order to facilitate data and metadata exchange, standard languages and interchange formats such as XML, XML-Data, RDF, have been proposed. While providing access to sources on the WWW has been simplified, there still remains the problem of finding a set of sources that are relevant to a query, and then ranking among these sources. In this subsection we show how source *quality* and *content* metadata can be shared on the WWW, and how this can be used to publish sources, and to select and rank sources.

6.1 Publishing Sources in XML

We describe an XML-based encoding format for source description metadata. Consider a scientist who measures air quality parameters in Canada. She measures the concentration of greenhouse gases in the atmosphere and stores the results of her daily measurements in a DB2 database. In order to make this data source available, she publishes the necessary connection information and source description metadata about the source's contents in an WS-XML document in Figure 10.

```
<?xml version="1.0"?>
  <title>Environmental Data for Canada</title>
  <ws>
    <wssci>
       <wrapper type="JDBC"/>
       <source name="EnvCanada"
               type="Oracle"
               location="jdbc:oracle://db.env.ca/federal"/>
    </wssci>
    <metadata>
        <schema>http://www.env.org/EnvSchema.xml</schema>
        <type name="AirQuality"
             completeness="0.90"
             last_update="Mon Feb  1 09:26:11 EST 1999"
             update_frequency="5:00:00"
             granularity="1:00:00">
           <attribute name="#city" atttype="ENUMERATION"
            values="Toronto Waterloo Ottawa"/>
        </type>
        ...
    </metadata>
    <desc> This repository contains daily measurements
           of air quality parameters in Ontario for the
           year 1998 and 1999. </desc>
  </ws>
```

Fig. 10: Source Content and Quality Metadata in XML

This document specifies the name, type, and location of the database (in the corresponding attributes of the <source> tag), the type of wrapper needed (<wrapper> tag), and a textual description of the source's contents (<desc> tag). The document also states that the types in the data source conform to a *schema* specified in a separate XML document, which is shared by a community of environmental scientists. The shared schema is specified in a document, "EnvSchema.xml" in this case, that uses the XML-Data [21] conventions for describing strongly typed relational schemas. The <type> tag tag is used to publish the *source description* metadata for each individual type. It corresponds to the values for the quality of data parameters that will be discussed next. The <attribute> tag can be used to specify the domain covered by the source. In this example the <attribute> tag provides the domain of the cities for which air quality information is provided by this source.

6.2 Quality of Data (QoD) Dimensions

We define four commonly encountered Quality of Data (QoD) dimensions: completeness, recency, frequency of updates and granularity. We note that additional QoD dimensions can be defined for each domain and schema.

- **Completeness**: Suppose there is some ideal, possibly virtual, complete source, that contains all the relevant data. Now, any particular source generally contains a fraction of the data (tuples) in the complete source. An estimate of the fraction of the data of the complete source that is in the particular data source represents its *completeness.* The completeness measure is useful in query evaluation to select sources that are most likely to contain relevant information. Completeness may be estimated using query feedback, for example, based on the queries that were successfully answered at this source, or may be provided by an administrator or expert user.

- **Recency**: Another important aspect of data quality is the *recency* of the data. One can imagine an environment where several data sources provide similar information, but some sources have older data than others. In many application domains data quality tends to depreciate in time so it makes sense whenever we have a choice to try to use the most recent data.

- **Frequency of updates**: A large class of sources are updated at fixed length intervals (eg. weekly or daily). If the frequency of updates is available it is reasonable to take it into consideration as an indication of the quality of a data source. Also, it can be used to estimate the recency, if the last update time is unknown.

- **Granularity**: An important class of data available online consists of so-called "time-series" data consisting of periodic samplings of some time-varying parameter. Examples include meteorological information, stock closing prices, currency exchange rates, etc. A common characteristic of these data types is the presence of a time attribute in the type definition and a functional dependency of all the other attributes on this one. For this class of data, one can distinguish among sources based on the sampling granularity (eg. hourly, daily, weekly).

6.3 Model for Source Content and Quality Descriptions

We consider the following model:

- T_1, T_2, ..., T_n are relational types, each type T_i has attributes A_{i1}, A_{i2}, $\cdots$, A_{ik_i}. Every attribute A_{ij} is associated with a domain D_{ij}.

- A source S contains data for a subset of T_1, T_2, ..., T_n

- A source S may have several *source content quality descriptions* (*scqd*'s) describing its contents.

An *scqd* is a tuple (t,*cd*,c,r,f,g), where t is a type and *cd* is a content description that specifies domains for some of the attributes of t. The parameters c,r,f,g correspond to the following Quality of Data (QoD) parameters: *completeness*, *recency*, *frequency of updates* and *granularity*, respectively. The QoD parameters qualify the data in the source described by the *cd*. They are as follows: c estimates the fraction of the data in the *complete type* [2] available in the source; r states how old is the data; f represents the length of the intervals when the data is updated; and g represents sampling granularity of the data.

Example 6.1 Example *scqd*'s may be as follows:

```
Temperature(time,city,value)
  S1   scqd11: (Temperature, [(city,{Toronto}),
           (time,YearSince1990)], 1.0, 3 days,_, 1 hour)
  S2   scqd21: (Temperature, [(city,{Kingston}),
           (time,CurrentYear)], 0.8, 2 days,_, 12 hours)
  S3   scqd31: (Temperature, [(city,CityInCanada),
           (time,YearSince1950)], 0.5, 1 day,_, 24 hours)
Rainfall(time,city,value)
  S2   scqd22: (Rainfall, [(city,{Kingston}),
           (time,CurrentYear)], 1, 2days,_, 12 hours)]
  S3   scqd32: (Rainfall, [(city,CityInCanada),
           (time,YearSince1950)], 0.5, 1 day,_, 24 hours)
```

6.4 Queries for Selecting and Ranking Sources

We propose a query language that exploits QoD parameters to select among a collection of data sources and rank them. The language can express queries with both strict and fuzzy conditions on the QoD dimensions associated with specific content descriptions. Fuzzy conditions are proximity predicates allowing one to imprecisely specify a desired target value for a certain QoD parameter. The evaluation of a query returns a list of sources that support the specified content description and satisfy the strict QoD conditions. The sources are ranked according to the degree to which they satisfy the fuzzy conditions. We illustrate the features of this language by the following query:

Query 6.1 *Find the best 5 sources that maintain information for the temperature in Toronto for the current year. Relevant sources must maintain 60% of all the data and the intervals of samples must be close to 1 hour.*

```
select   best 5 s
from     Source s,
         Scqd q In s
where    q.type = "Temperature"
         And q.cd = [(city,{Toronto}),(year,{1999})]
         And q.completeness > 0.6
         And q.granularity Closeto "1 hour";
```

The above query selects sources that contain an *scqd* matching the specified *cd*, whose completeness is better than the cutoff value and returns an ordered list the five sources whose granularity comes closest to 1 hour.

The query language also supports weighted combinations of fuzzy conditions. For example, suppose we are interested in sources that contain data approximately one week old and granularity close to 1 hour, and we care twice as much about the granularity being close to the target value as we care about the recency. Then, we can express this by including explicit weights for each fuzzy condition:

```
select   best 5 s
from     Source s,
         Scqd q In s
where    q.type = "Temperature"
         And (2/3)*(q.granularity Closeto "1 hour")
         And (1/3)*(q.recency Closeto "1 week");
```

The combined score is computed from the individual scores according to the following formula introduced by Fagin in the context of multimedia databases [15].

Details of the query language and computational issues in source selection and ranking are in [27, 28]. One important issue requiring additional research is scaling the task of indexing *scqd*'s over the WWW, and building searchable indexes a la search engines for text. The other task is maintaining these indexes to be current, as sources appear and disappear, and their content and quality metadata changes over time.

7 Conclusions

In this paper, we presented our wrapper mediator architecture for WebSources. We addressed two issues in scaling these architectures for wide area query processing. We report on our results on developing a Web query optimizer (WQO) that uses WebPT - a tool for predicting response times. We also report on results for coping with unexpected delays. This includes *Query Scrambling* – a reactive query execution scheme that adapts the query plan in response to runtime delays, and *XJoin* – a small footprint, fully pipelinable join operator that automatically adjusts the flow of tuples during query execution.

We also introduce two additional issues that must be addressed when scaling to large numbers of sources. The first task is query processing with alternate sources. The second task is using the WWW and XML to publish and locate sources and their content and quality metadata.

References

[1] ACM digital library. http://www.acm.org/dl/Search.html.

[2] Laurent Amsaleg, Philippe Bonnet, Michael J. Franklin, Anthony Tomasic, and Tolga Urhan. Improving responsiveness for wide-area data access. *IEEE Data Engineering Bulletin*, September 1997.

[3] Laurent Amsaleg, Michael J. Franklin, and Anthony Tomasic. Dynamic query operator scheduling for wide-area remote access. *Journal of Distributed and Parallel Databases*, 6(3), 1998.

[4] Laurent Amsaleg, Michael J. Franklin, Anthony Tomasic, and Tolga Urhan. Scrambling query plans to cope with unexpected delays. In *Proceedings of the International Conference on Parallel and Distributed Information Systems*, Miami Beach, Forida, USA, December 1996.

[2] The complete type is a possibly virtual relation that contains all the relevant data for the type.

[5] A.Tomasic, L.Raschid, and P.Valduriez. Scaling access to distributed heterogeneous data sources with disco. *IEEE Transactions On Knowledge and Data Engineering*, 1998.

[6] J. Blakeley. Data access for the masses through ole db. *Proceedings of the ACM SIGMOD Conference on Management of Data*, 1999.

[7] Philippe Bonnet and Anthony Tomasic. Parachute queries in the presence of unavailable data sources. Technical Report RR-3429, INRIA, May 1998.

[8] L. Breiman, J.H. Friedman, R.A. Olshen, and C.J. Stone. *Classification and Regression Trees.* Wadsworth and Brooks/Cole, 1984.

[9] L. Bright et al. A wrapper generation toolkit to specify and construct wrappers for webaccesible data sources (websources). *To appear in Computer Systems Special Issue on Semantics in the WWW*, 1999.

[10] L. Bright and L. Raschid. Cost modeling of wrappers for web accesible data sources (websources). *Under review.*, 1999.

[11] L. Bright, L. Raschid, V. Zadorozhny, and T. Zhan. Learning response times for websources: A comparison of a web prediction tool (webpt) and a neural network. *Under review*, 1999.

[12] 1998 california campaign contribution database. http://ca98.election.digital.com/query.html.

[13] S. Chaudhuri and K. Shim. Query optimization in the presence of foreing fucntions. *Proc. of VLDB*, 1993.

[14] Microsoft Corporation. *OLE2 Programmer's Reference.* Microsoft Press, Redmond WA, 1996.

[15] R. Fagin. Fuzzy queries in multimedia database systems. In *Proceedings of the 17th Symposium on Principles of Database Systems (PODS)*, pages 1–10, Seattle, Washington, June 1998.

[16] Fishbase 99.
http://www.cgiar.org/iclarm/fishbase/search.cfm.

[17] D. Florescu et al. Answering queries using oql view expressions. *Workshop on Materialized Views: Techniques and Applications in conjunction with ACM SIGMOD*, 1996.

[18] D. Florescu et al. Query optimization in the presence of limited access patterns. *Proc. of the ACM Sigmod Conference*, 1999.

[19] The global historical climatology network (GHCN). http://www.ncdc.noaa.gov/ol/climate/research/ghcn/ghcn.html.

[20] J.-R. Gruser, L. Raschid, V. Zadorozhny, and Tao Zhan. Learning response time for websources using query feedback and application in query optimization. *Under review*, 1999.

[21] A. Layman et al. The xml- data home page. *http://www.microsoft.com/standards/xml/.*

[22] A.Y. Levy et al. Querying heterogeneous information sources using source descriptions. *Proc. of VLDB*, 1996.

[23] L.Haas, D.Kossmann, E.Wimmers, and J.Yang. Optimizing queries across diverse data sources. *Proceedings of VLDB Conference*, 1997.

[24] ACM Digital Library.
http://www.acm.org/dl/Search.html.

[25] M.Carey et al. Towards heterogeneous multimedia information systems: the garlic approach. *Technical Report, IBM Almaden Research*, 1995.

[26] Sun Microsystems. Java (tm): Programming for the internet. *http://java.sun.com.*

[27] George Mihaila, Louiqa Raschid, and María Esther Vidal. Query evaluation for source selection and ranking. *Technical Report, UMIACS, University of Maryland*, 1999.

[28] George Mihaila, Louiqa Raschid, and María Esther Vidal. Querying "quality of data" metadata. In *Proceedings of the Third IEEE Meta-Data Conference*, Bethesda, Maryland, April 1999.

[29] O.Kapitskaia, A.Tomasic, and P.Valduriez. Dealing with discrepancies in wrapper functionality. *Technical Report INRIA*, 1997.

[30] Y. Papakonstantinou et al. Capabilities-based query rewriting in mediator systems. *Proc. of the Conference on Parallel and Distributed Information Systems*, 1996.

[31] R. Ramakrishnan P.Seshadri, M.Livny. The case for enhanced abstract data types. *Proc. of VLDB*, 1997.

[32] Praveen Seshadri and Mark Paskin. Predator: An ordbms with enhanced data types. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, Tucson, Arizona, USA, 1997.

[33] Leonard D. Shapiro. Join processing in database systems with large main memories. *ACM Transactions on Database Systems*, 11(3), 1986.

[34] K. Thompson, G. Miller, and R. Wilder. Wide-area internet traffic patterns and characteristics. *IEEE Network, November/December*, 1997.

[35] A. Tomasic et al. Scaling heterogeneous databases and the design of disco. *Proceedings of the Intl. Conf. on Distributed Computing Systems*, 1996.

[36] Tolga Urhan and Michael J. Franklin. Xjoin: Getting fast answers from slow and bursty networks. Technical Report CS-TR-3994, UMIACS-TR-99-13, University of Maryland, February 1999.

[37] Tolga Urhan, Michael J. Franklin, and Laurent Amsaleg. Cost based query scrambling for initial delays. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, Seattle, Washington, USA, 1998.

[38] V. Vassalos and Y. Papakonstantinou. Describing and using query capabilities of heterogeneos sources. *Proceedings of the VLDB Conference*, 1997.

[39] G. Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, pages 38–49, March 1992.

[40] Anitta N Wilschut and Peter M. G. Apers. Pipelining in query execution. In *Proc. Conf. on Databases, Parallel Architectures, and their Applications*, Miami, FLorida, USA, 1991.

[41] R. Wolski. Dynamically forecasting network performance to support dynamic scheduling using the network weather service. *Proc. of the 6th High-Performance Distributed Computing Conference*, 1997.

[42] Y.Papakonstantinou, A. Gupta, H. Garcia-Molina, and J. Ullman. A query translation scheme for rapid implementation of wrappers. *Proceedings of the Intl. Conference on DOOD*, 1996.