Distributed Queries without Distributed State[*]

Vassilis Papadimos David Maier
vpapad@cse.ogi.edu maier@cse.ogi.edu

Department of Computer Science and Engineering
OGI School of Science & Engineering
Oregon Health & Science University

Abstract

Traditionally, distributed queries have been optimized centrally and executed synchronously. We outline a framework that relaxes both of these constraints using *mutant query plans*: XML representations of query plans that can also include verbatim XML data, references to resource locations (URLs), or abstract resource names (URNs). Servers work using local, possibly incomplete knowledge, partially evaluate as much of the query plan as they can, incorporate the partial results into a new, mutated query plan and transfer it to some other server that can continue processing. We present preliminary performance results, and discuss issues and strategies for mutant query optimization.

1 Introduction

The Internet is probably the most successful distributed computing system ever. However, our capabilities for data querying and manipulation on the Internet are primitive at best. The queries we can ask of remote servers are limited: Get the data for a given URL, or ask predefined queries using some form interface. Also, queries only involve a single client pulling data from a single server: There is no infrastructure for distributed queries. Many queries that we routinely want to ask require combining data from different data sources. We cannot always move the data pertinent to a query to a single server and do the processing there, for both technical and political reasons.

We will use a query about films, reviews, and theaters as an example. Our user, Bob, wants to see a movie tonight. Bob visits his favorite portal, *BobsPortal.com*, where he can ask queries about XML documents with films and showings. Bob uses some GUI front-end to come up with an XML query[1] such as the following:

```
FOR $r in document(‘‘film_reviews’’)//review, $g in document(‘‘preferences’’)//genre,
    $s in document(‘‘film_showings’’)/showing[date = ‘‘15 March 2002’’]
WHERE $r/genre = $g AND $r/title = $s/title
RETURN  <film> { $r/title } { $r/rating } { $s/theater } </film>
```

This query works on three XML documents: `film_reviews`, `preferences`, and `film_showings`. There are several kinds of magic going on here; *BobsPortal.com* is smart enough to know that `preferences` means Bob's preferences, and `film_showings` only includes theaters in Bob's town. The query also treats these documents as abstract resources; it does not mention their actual locations anywhere.

The query processor will start by translating Bob's query into a *logical query plan* (Figure 1), which is a directed graph of *logical query operators*, such as `select` or `join`, that consume and produce sequences of tuples. A tuple contains references to XML fragments. We also have special pseudo-operators, such as `document`, which creates a sequence of tuples by fetching data from a URL, and `display`, which presents results on the client's computer. We turn a logical query plan into a *physical query plan* by selecting an implementation algorithm for query operators, such as *nested-loops*, or *hash join* for logical join. A physical query plan can be executed directly by the query processor.

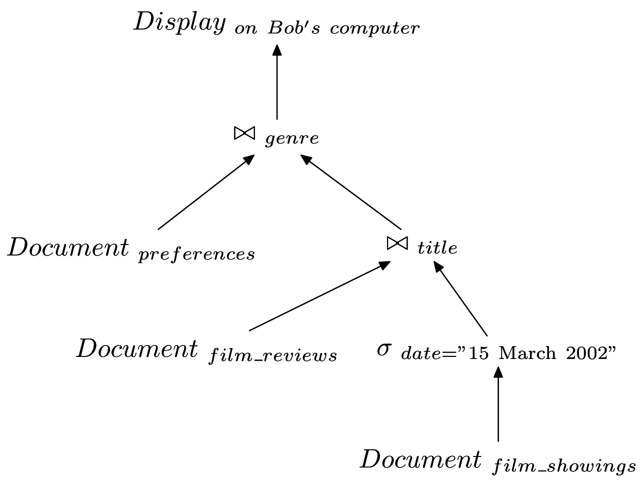
BobsPortal.com however, may not have the `film_reviews` and `film_showings` data locally; so how should it process the query? If it knows how to resolve these abstract documents into actual URLs, it could download them, and process the query. This approach transfers large amounts of data (every movie showing at Bob's town, and reviews

[*]Funding for this work was provided by DARPA through NAVY/SPAWAR contract N66001-99-1-8908 and by NSF ITR award IIS0086002.

[1]The real query is more complex if we want to nest `theater`s inside `film`s, and not repeat `title` and `rating` for each showing.

for every movie currently playing anywhere), even though we only need a subset of these data. We would like to do some processing near the data sources to reduce data transfer.

In the traditional distributed query processing model, one site (the *coordinator*) optimizes a client's query into a *distributed* query plan: a query plan where operators are annotated with the sites where they should run. The coordinator sends the sub-plans to the *apprentice* sites, and coordinates query execution. For query optimization to work well, the coordinator needs detailed statistics about data placement and capabilities of the other participating sites and the network. There are problems in scaling this approach to large networks of autonomous sites. A remote site may refuse to run subqueries (it may be down, off line, or overloaded). Further, we cannot hope to maintain accurate and timely statistics on the location and characteristics of all possible Internet resources at a single centralized location.



$Display_{on\ Bob's\ computer}$
$\bowtie_{genre}$
$Document_{preferences}$
$\bowtie_{title}$
$Document_{film\_reviews}$
$\sigma_{date=\text{"15 March 2002"}}$
$Document_{film\_showings}$

Figure 1: A logical query plan.

We believe that to implement distributed queries efficiently over the Internet, we must abandon the notion of an omniscient, omnipotent coordinator able to optimize queries centrally and to oversee all aspects of their execution. We introduce a framework using *mutant query plans* to decentralize query optimization and execution. Mutant query plans can cope with incomplete metadata, can be optimized in a decentralized fashion, respect the autonomy and the local policies of sites they execute at, and adapt to server and network conditions even while being evaluated.

2 Mutant query plans

Mutant query plans (MQPs) [PM02] are the unit of communication in our framework for distributed XML queries. The usual way to reference data sources in an XML query is to use their URL. An MQP is a query plan graph, serialized in XML, that in addition to URLs, can refer to abstract resources using URNs, and include verbatim XML fragments. An MQP is also tagged with a network address to send its result to, once it is fully evaluated. To evaluate a mutant plan that includes URNs, we *resolve* these URNs to their corresponding URLs. Figure 1 is actually an MQP in disguise, with the three documents referenced as abstract resources.

Figure 2 shows how a server processes an MQP. Mutant query plans are transferred as XML documents. A server parses a MQP into a tree of query operators and constant data. Every server maintains a local *catalog* that maps each URN to either a URL, or to a set of *servers that know more about the URN*. The server resolves the URNs it knows about, then its *optimizer* component (re)optimizes the plan, and finds or creates sub-plans that can be evaluated locally, with their associated costs. The *policy manager*, at this point, makes the decision to accept or reject the mutant plan (maybe the server is overloaded, or the plan's cost is too high). The policy manager also decides how much of a plan to evaluate locally, and passes those sub-plans to the *query engine*. The server then substitutes each evaluated sub-plan with its results (as an XML fragment), to get a new, *mutated* query plan.

If the plan is not yet fully evaluated, we must decide the next server to send it to. Again, consulting the catalog, we send the plan to a server that knows how to resolve at least one of the remaining resources. A given server does not need to know how to resolve every URN in a plan. As long as the plan *eventually* passes through a server that does, it can be evaluated. At some point, a server will hopefully reduce the plan to an XML document and forward it to its final destination (which may be different than its origin), or alternatively, report its failure to process the plan further.

We illustrate with Bob's query. *BobsPortal.com* inserts Bob's preferences (science fiction movies) as a constant XML fragment, to obtain the mutant plan in Figure 3(a). It does not know how to resolve `film_reviews` or `film_showings`, but does know that *YourTimes.com* can resolve `film_showings` for Bob's town, so it sends the plan there. *YourTimes.com* `select`s today's movies, and inserts the results as another XML fragment, as shown in Figure 3(b). The new plan is then forwarded to *movies.yoohoo.com*, which can resolve the `film_reviews` resource. *Movies.yoohoo.com* performs the two `join`s (result not shown here), and sends the final results to Bob's computer for `display`ing.

Note that we can use an MQP to represent a distributed query at every stage of its processing. We can transform

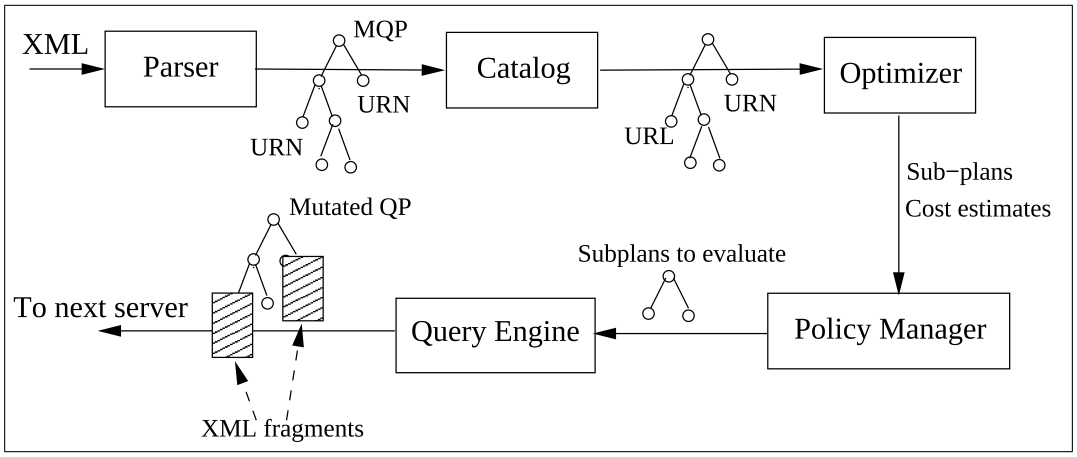


Figure 2: Mutant query processing.

the original query into a query plan and encode it in XML, obtaining an MQP; different servers can evaluate parts of the query plan and insert the intermediate results as verbatim XML fragments; and the final XML document produced as the result is again a (trivial) MQP, consisting only of XML content.

We see several kinds of services coexisting in the MQP framework. Any HTTP server is a valid MQP server (albeit of limited functionality). It only accepts *point queries* of the form: "Give me the data for this local URL". A vanilla Niagara server can also function as an MQP server; one that accepts all MQPs that can be fully evaluated locally (no URNs). A full-blown MQP server handles URNs, can partially evaluate a query, and can construct and forward the mutated plan to another server for further processing.

3 Performance

We prototyped mutant query plans using the Niagara query engine [NDM99], plus three new pseudo-operators: `constant` encapsulates an XML fragment, `resource` represents a URN, and `display` specifies the final destination. Each server has a metadata catalog that maps each URN to a URL, or to servers that can resolve it. Our current prototype uses a greedy approach to evaluate MQPs. A server resolves as many URNs in an MQP as possible, evaluates as many operators in the MQP as it can, and then sends the mutated plan to the server that can resolve the most remaining URNs.

We also implemented a traditional, pipelined distributed plan processor for Niagara. Each plan node has a `location` attribute, to indicate the server the operator should run on. When a server A processing a query plan encounters a sub-plan marked for server B, it is sent to B, with a `send` operator on top. A replaces the sub-plan locally with a `receive` operator, which will connect to the corresponding `send` at B. The execution of the two

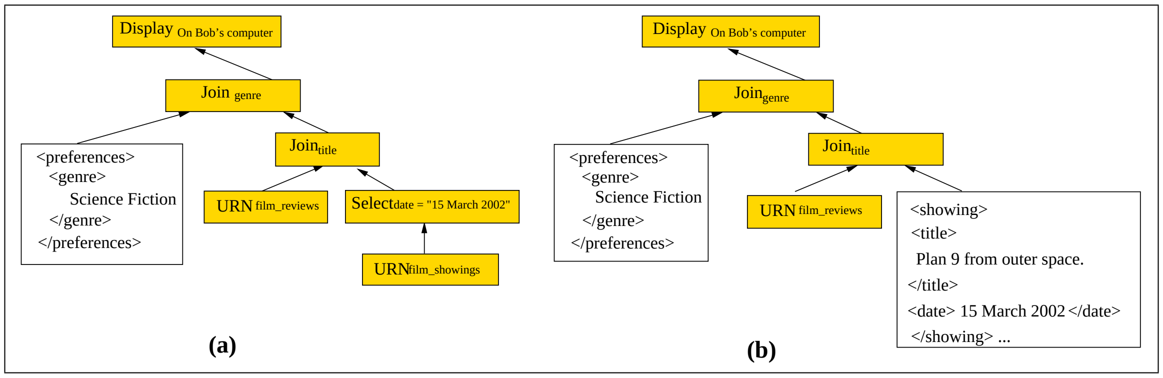


Figure 3: Bob's mutant query plan: **(a)** at BobsPortal.com, **(b)** at YourTimes.com.

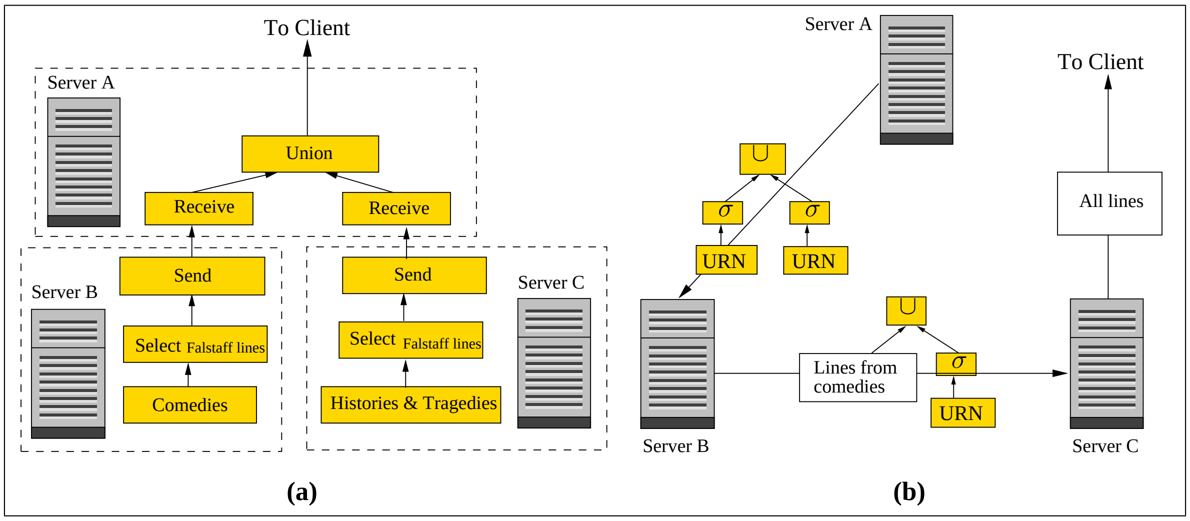


Figure 4: Execution strategies for the Falstaff query: **(a)** pipelined plan, **(b)** mutant plan.

query plans proceeds concurrently. Server B serializes tuples at the `send` node into XML, and transmits them to the `receive` node at server A, which parses them back into in-memory tuples.

We compare the two prototypes on a simple query, using the XML-encoded plays of Shakespeare[2]. The query asks for all the lines of Sir John Falstaff, in any play. Our setup has three identical servers, A, B, and C, with a fourth machine as the client. B stores all the comedies, while C stores all the histories and tragedies. We timed pipelined and mutant versions of our query, using two scenarios: normal load, and artificially increased load at C.

In the pipelined plan (Figure 4(a)) the client submits the query to the coordinator, A, which unions two sub-plans, running on B and C. Sub-plans scan their local plays, `select` Falstaff's lines, and stream them to A. In the MQP version (Figure 4(b)), the query plan is again the `union` of two `select`s, but contains just URNs to `tragedies` and `comedies`. The client sends the MQP to A, which routes it to B with no local evaluation. B resolves the `comedies` URN to a set of URLs, executes that part of the plan, appends its local lines to the mutant plan, and sends it to C. C resolves the `tragedies` URN, executes the rest of the plan, and sends the final results to the client.

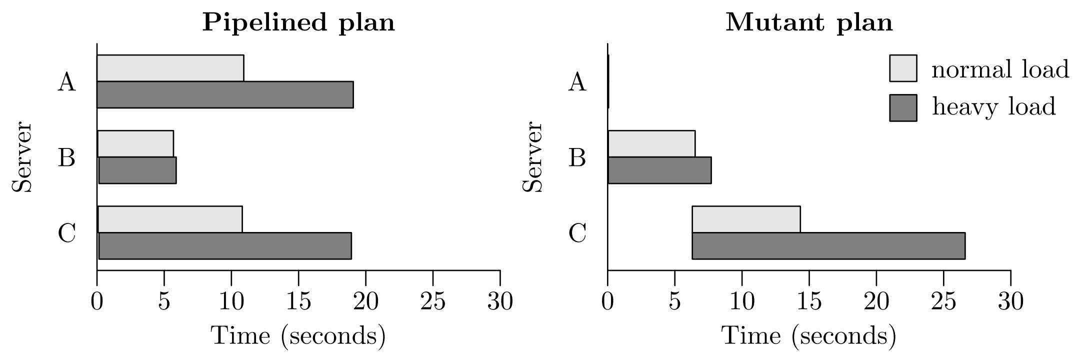
For all versions of the query, we ran each query once to warm the caches, then averaged elapsed times for ten runs. The results, and timing sequences for the three servers are shown in Figure 5. Columns **A**, **B**, and **C** show elapsed time on the three servers, while **Client** is the time, at the client, between sending the query and receiving the results. A takes negligible time to transfer a mutant plan, or start a distributed sub-plan ($< 0.1s$). The mutant plan has worse latency than the pipelined one, since it only works on one server at a time. Notice that the mutant plan's total time is less than the sum of the B and C times of the pipelined plan: $13.9s$ vs. $16.3s$ under normal load. The mutant plan transfers fewer tuples: Falstaff lines from the "tragedies" are transferred once, between C and the client, instead of going from C to A to the client. **Footprint** is the sum of elapsed times for all the servers involved. The footprint of the pipelined plan is worse than for the mutant plan (~$13s$ under normal load, ~$16s$ under heavy load). The reason is that A must wait for both apprentice sites to finish, before it finishes.

While one shouldn't read too much into a single test, it does illustrate the trade-off available with MQPs. While performance on individual queries is somewhat worse, overall load on servers is reduced. Traditional distributed query processing requires distributing, activating and simultaneous communication with sub-plans; it requires *distributed state*. Our approach, in contrast, allows evaluation of distributed queries while maintaining only *local state*, at any point (except for brief periods to transfer MQPs).

4 Related work

Hybrid shipping [FJK96] combines *query shipping* ("send query to data") with *data shipping* ("send data to query"), thus allowing query operators to execute on both clients and servers. MQPs, being simultaneously queries *and* data,

[2]Marked up by Jon Bosak, available from `http://metalab.unc.edu/bosak/xml/eg/shaks200.zip`



Scenario	**A** (sec)	**B** (sec)	**C** (sec)	**Client** (sec)	**Footprint** (sec)
Pipelined (normal load)	10.9 ± 0.1	5.6 ± 0.1	10.7 ± 0.1	11.0 ± 0.1	27.3 ± 0.3
Pipelined (heavy load)	19.1 ± 0.7	5.7 ± 0.1	18.8 ± 0.7	19.1 ± 0.7	43.6 ± 1.4
Mutant (normal load)	< 0.1	6.5 ± 0.1	8.0 ± 0.1	13.9 ± 0.1	14.6 ± 0.1
Mutant (heavy load)	< 0.1	7.7 ± 0.1	20.3 ± 0.5	26.1 ± 0.5	28.0 ± 0.5

Figure 5: Performance results for the Falstaff query, under normal and heavy loads.

provide another alternative, which we might term *combined shipping*. The ObjectGlobe project [BKK$^+$01] is building a distributed query processing framework over the Internet. It depends on centralized query optimization, and centralized metadata maintenance. Sahuguet et al. [SPT00] propose incorporating new execution models such as referral, chaining, leasing, and publish-subscribe into distributed query execution. We can currently implement chaining with MQPs, and are working on other mechanisms, especially publish-subscribe. Jim and Suciu [JS01] propose *intensional answers* (partially evaluated queries, in the form of facts and rules) in a distributed query setting, to accommodate site independence and dynamic site discovery. MQPs provide the same flexibility for *queries* as well as for answers.

Bonnet and Tomasic [BT98] address the problem of temporarily unavailable data sources by partially evaluating a query using the sources currently available. These partial results provide useful user feedback, and can be used to construct a *parachute query*, that combines the partial results with the remaining data sources to get the complete answer. We can route MQPs around unavailable data sources. A mutant plan will head for servers that can perform some work, leaving the unavailable servers for last. If those servers are still unavailable, it can either lay dormant, or head to the client with the partial results it has gathered.

5 MQP Optimizations: Consolidation, absorption, deferment

To handle mutant query plans effectively, servers must interact with their cost estimator and query optimizer components in interesting and unconventional ways. In this section, we consider local optimization on MQPs. Suppose we receive the MQP shown in Figure 6(a), where resources A, B, and C are local, while X and Y are not. We cannot evaluate any of the joins locally, since they all depend on unavailable resources, but we can rewrite this plan into an equivalent plan where we can evaluate more operators locally. We call this process *consolidation*.

We define an operator to be *local* if all its inputs are available locally, otherwise it is *remote*. A query plan is consolidated if at most one local operator has a remote parent. We can use join associativity and commutativity to rewrite our plan so that A, B, and C are brought together in a consolidated plan (Figure 6(b)). We can specify the consolidation process in a top-down query optimizer such as Columbia [SMB$^+$01] using simple *rewriting rules*, which are special cases of associativity and commutativity. We model the locality of an operator as a logical property. Here are the five rules we need for query plans that contain only $\bowtie$ and `Get` operators. Expressions L_i are local, while R_i are remote. For example, we can derive the plan in Figure 6(b) by applying rules 1, 2, 2, 3, 1 and 2 to the original plan.

$$R \bowtie L \quad \rightarrow \quad L \bowtie R \tag{1}$$
$$L_1 \bowtie (L_2 \bowtie R) \quad \rightarrow \quad (L_1 \bowtie L_2) \bowtie R \tag{2}$$
$$R_1 \bowtie (L \bowtie R_2) \quad \rightarrow \quad (L \bowtie R_1) \bowtie R_2 \tag{3}$$
$$L \bowtie (R_1 \bowtie R_2) \quad \rightarrow \quad (L \bowtie R_1) \bowtie R_2 \tag{4}$$

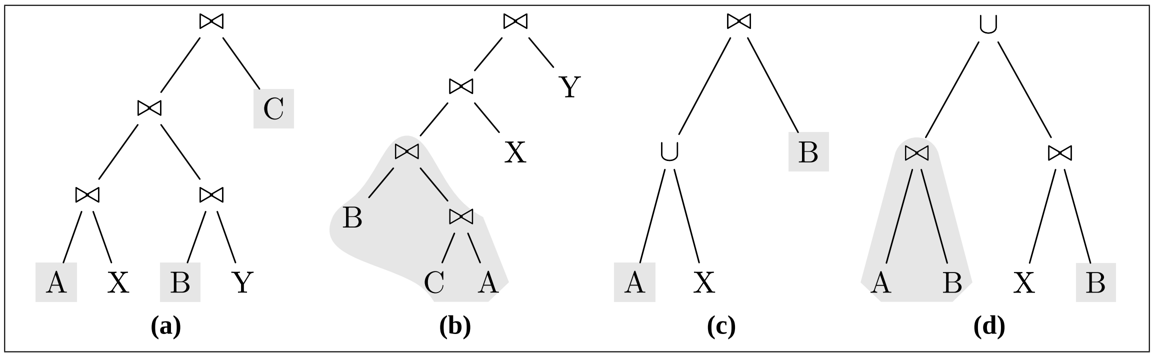


Figure 6: Phases of mutant query optimization: **(a)** incoming plan, **(b)** a consolidated plan, **(c)** a plan that cannot be consolidated, **(d)** plan after absorption. Shaded operators are locally evaluable.

$$R_1 \bowtie (R_2 \bowtie R_3) \quad \rightarrow \quad (R_1 \bowtie R_2) \bowtie R_3 \tag{5}$$

We have proved that these rules always terminate with an equivalent consolidated expression. We can extend these rules to handle various other operators. Intersection and cross-product are special cases of join. Most unary operators are also easy to handle. Can we consolidate every query plan? Unfortunately, no. Depending on the algebra, there may be expressions that we cannot consolidate. Consider the algebra that includes selection, join, and union, two relations L and R representing books in bookstores, with schema (id, author, title, price), and the expression $(\sigma_{author='X'}L) \cup (L \bowtie_{L.id=R.id \wedge L.price<R.price} R)$. Suppose only L is local. We cannot consolidate the two appearances of L using this algebra. However, consolidation would be trivial if we added a 'tee' operator that replicates its input into multiple outputs.

Under some circumstances, we can still apply rewriting rules to increase the useful work we can perform locally, even though the resulting plan will remain non-consolidated. We can rewrite the plan in Figure 6(c) into Figure 6(d). If $|L_1 \bowtie L_2| < |L_1|$, this rule reduces the size of the resulting MQP. We call this process *absorption*. A traditional relational optimizer would usually not consider such a rewriting. For MQPs, we frequently have to materialize (in space instead of time) intermediate results of query sub-plans, and anything that can shrink these results is important. While consolidation can occur before query optimization, absorption is a parallel process to query optimization, since we need the cost estimator to decide whether an absorption rewriting is truly beneficial.

Finally, even though we may be able to evaluate an operator locally, it may not pay to do so. Suppose the join between B and $C \bowtie A$ in Figure 6(b) is a Cartesian product. Instead of evaluating it, we should just include B as verbatim XML data, to avoid inflating the resultant MQP. We call deciding which locally evaluable operators to postpone *deferment*. We are investigating means to cost all alternative deferments with one run of our query optimizer.

References

[BKK$^+$01] Reinhard Braumandl, Markus Keidl, Alfons Kemper, Donald Kossmann, Stefan Seltzsam, and Konrad Stocker. ObjectGlobe: Open Distributed Query Processing Services on the Internet. *IEEE Data Eng. Bulletin*, 24(1):65–69, 2001.

[BT98] Philippe Bonnet and Anthony Tomasic. Parachute queries in the presence of unavailable data sources. Technical Report RR-3429, INRIA Rocquencourt, France, 1998.

[FJK96] Michael J. Franklin, Björn Thór Jónsson, and Donald Kossmann. Performance tradeoffs for client-server query processing. In *Proceedings of the ACM SIGMOD Conference*, pages 149–160, Montreal, Canada, June 1996.

[JS01] Trevor Jim and Dan Suciu. Dynamically Distributed Query Evaluation. In *Proceedings of the Twentieth ACM PODS Symposium*, pages 28–37, Santa Barbara, California, May 2001.

[NDM99] Jeffrey Naughton, David DeWitt, and David Maier. The Niagara Internet Query System. Available from `http://www.cs.wisc.edu/niagara/papers/NIAGRAVLDB00.v4.pdf`. 1999.

[PM02] Vassilis Papadimos and David Maier. Mutant Query Plans. *Information and Software Technology*, 44(4):197–206, April 2002. Preprint version at `http://cse.ogi.edu/~vpapad/mqp.pdf`.

[SMB$^+$01] Leonard D. Shapiro, David Maier, Paul Benninghoff, Keith Billings, Yubo Fan, Kavita Hatwal, Quan Wang, Yu Zhang, Hsiao min Wu, and Bennet Vance. Exploiting upper and lower bounds in top-down query optimization. In *Proceedings of IDEAS '01*, pages 20–33, 2001.

[SPT00] Arnaud Sahuguet, Benjamin Pierce, and Val Tannen. Chaining, Referral, Subscription, Leasing: New Mechanisms in Distributed Query Optimization, 2000. Online at: `http://db.cis.upenn.edu/DL/new_mechanisms.pdf`.