A Framework for Navigation-Driven Lazy Mediators

Bertram Ludäscher   Yannis Papakonstantinou   Pavel Velikhov

`ludaesch@sdsc.edu`   `{yannis,pvelikho}@cs.ucsd.edu`

Abstract

We propose a framework for navigation-driven evaluation of virtual mediated views. The approach is based on *lazy mediators* that translate incoming client navigations on a virtual XML view (or the result of an XML query) into navigations on wrapped sources or lower level mediators. Such a demand-driven approach is inevitable in order to handle up-to-date mediated views of huge Web sources or large query results, which are commonplace when querying the Web. The proposed MIX mediator provides to the client an abstraction of the DOM API, the de facto API to XML documents, hence hiding the non-materialization of the view or the query. We also describe the inherent *navigational complexity* of queries and view definitions wrt. navigations. Next, we discuss the query/navigation processing aspects of the MIX mediator. The mediator translates a view into an algebraic plan. Then each operator of the algebraic plan acts as a navigation-driven lazy mediator. This allows to characterize the complexity of plans and to compare their navigational complexity.

1 Introduction and Overview

Mediated views integrate information from heterogeneous sources. There are two main paradigms for evaluating queries against integrated views: In the *eager* (or *warehousing*) approach, data is collected and integrated *prior* to the execution of user queries against the materialized views. However, when the user is interested in the most recent data available and/or huge (or potentially infinite) views, then a *lazy* (or *demand-driven*) approach has to be employed. Most notably such requirements are encountered when considering integration of Web sources. For example, consider a mediator which integrates data on books available from `amazon.com` and `barnesandnoble.com`. Clearly, a warehousing approach based on all available books is not viable. In the lazy approach, however, the user query against the mediated view will be decomposed at the mediator and corresponding subqueries will be sent to the sources at query evaluation time.

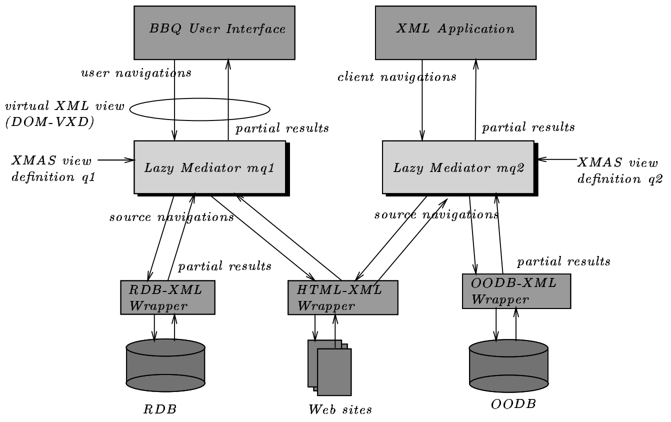

Figure 1: DOM-VXD mediation architecture

In the MIX[1] system we take the lazy mediator approach to its full potential using the architecture of Figure 1. The common model for data exchange is XML [XML98b]. Mediator views are expressed in XMAS[2], a declarative XML query language, which is very similar to XML-QL [XML98a] and YAT [CDSS98]. Nevertheless most of the concepts and results presented next are orthogonal to the details of the semistructured model and the query language.

An application, such as the ***B****lended* ***B****rowsing and* ***Q****uerying* (BBQ) interface [BGL+99], may directly access the view or issue a query against the view and consequently access the query result. In the balance of the paper we assume that the application directly accesses the view. The case of issuing a query against the view is handled by replacing the view with the composition of the query and the view.

The application accesses the view using a subset of navigation commands of the DOM[3] API. However, the lazy mediator MIXm does not materialize the view document in advance. Instead, view materialization is driven by the application's navigation. The media-

[1] ***M****ediation of* ***I****nformation using* ***XML*** [MIX99].
[2] ***XML M****atching* ***A****nd* ***S****tructuring Language* [LPVV99].
[3] ***D****ocument* ***O****bject* ***M****odel* [DOM98].

tor translates the application requested navigations into corresponding navigation commands against lower level lazy mediators or sources. Data returned by navigating the sources is then processed in the mediator. Finally, the requested pieces of the virtual view are sent back to the user.

Note that it is transparent to the application that the view document it accesses is a virtual one. We call this framework *DOM-VXD* (*DOM for Virtual XML Documents*). In this paper we outline the implementation of DOM-VXD at mediators. Due to space limitations we do not discuss the important issues involved in implementing DOM-VXD wrappers.

In Section 2, we introduce navigation-driven lazy mediators and a new notion of *navigational complexity* for describing the complexity of views in terms of navigations. In Section 3, we present the operators of the XMAS algebra and show how they can be realized as navigation-driven lazy mediators. In Section 4, conclusions and some future directions are given.

2 Navigations

A *lazy mediator* m_q for a view definition q operates as follows: The client navigates into the virtual view exported by m_q by issuing DOM *navigation commands* on the view document exported by the mediator. For each command c_i that the mediator receives (Figure 2), a minimal source navigation is sent to each source. Note that navigations sent to the sources depend on the client navigation, the view definition, and the *state* of the lazy mediator. The results of the source navigations are then used by the mediator to generate the result for the client and to update the mediator's state.

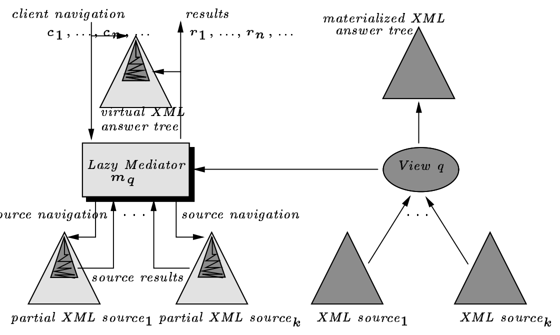

Figure 2: Navigational interface of a lazy mediator

Data and Access Model. We model XML documents as *labeled ordered trees* over a suitable underlying domain $\mathbf{D}$.[4] In the sequel, we will refer to labeled ordered trees simply as trees. The set $\mathbf{T}$ of all such trees comprises all atomic values d and constructs $d[t_1, \ldots, t_n]$, where $d \in \mathbf{D}$ is a label (or *element type* in XML parlance), $t_i \in \mathbf{T}$, and $[t_1, \ldots, t_n]$ is an *ordered list* of trees (*subelements*).

XML documents (both source and answer documents) are accessible through navigation commands. We present a navigational interface that is an abstraction of the DOM API for XML, and employs *pointers* (references) to keep track of the explored part of the document being navigated.

- d (*down*): $d(p) = p'$ if p' is the first child of p; if p is a leaf then $d(p) = \perp$ (null).
- r (*right*): $r(p) = p'$ if p' is the right sibling of p; if there is no right sibling $r(p) = \perp$.
- f (*fetch*): $f(p) = l$ if l is the label of p.

Apart from this minimal set of navigation commands, which is sufficient to explore a virtual document completely, further navigation commands can be provided. For example, we may include in $\mathbf{NC}$ a command for selecting certain siblings:

- *select* (σ_θ): $\sigma_\theta(p) = p'$ if p' is the first sibling to the right whose label satisfies θ; if there is no such sibling $\sigma_\theta(p) = \perp$.

By adding more powerful navigation commands like σ_θ, the navigational complexity of queries can be improved (see below).

Note also that it will often be impossible or too expensive to navigate a source using the fine granularity of the DOM commands. The MIX system resolves this problem by allowing wrappers, the pieces of software that are between sources and mediators, to navigate the source in the granularity that makes most sense for the specific source (as well as for the intermediate network). Then a buffer incorporated in the wrapper resolves the discrepancy between the source's granularity of navigation and the DOM. Due to lack of space, we omit a detailed discussion of this wrapping and buffering scheme.

Definition 1 (Navigation) Let p_0 be the root for some document $t \in \mathbf{T}$. A *navigation* into t is a sequence

$$c = \quad c_1(p_0),\ c_2(p_1),\ \ldots\ ,\ c_n(p_{n-1})$$

where each $c_i \in \mathbf{NC}$ and each p_{i-1} is the result of a previous command $c_j(p_{j-1})$ for some $j \leq i-1$.

The *result* (or *explored part*) $c(t)$ of applying the navigation c to t is the unique subtree comprising only

[4]$\mathbf{D}$ includes all "string-like" data, i.e., element/attribute names, attribute values, and character content.

those node-ids and labels of t which have been accessed through c. Depending on the context, $c(t)$ may also denote the final point reached in the sequence, i.e., $c_n(p_{n-1})$.

Instead of $c = c_1(p_0), c_2(p_1), \ldots, c_n(p_{n-1})$ where $p_i = c_i(p_{i-1})$, we may write $c = c_1, \ldots, c_n$. □

Navigational Complexity. When a lazy mediator receives a user navigation command it translates it into the smallest source navigation command sequence that is sufficient for returning a result to the user. The "browsability" of a mediator view depends on how small these source navigation sequences are. We split views into three categories as illustrated by the following example.

Example 1 Consider a view q_{conc} which computes the concatenation of elements of two sources by making the root elements of the sources siblings of a new root element of the answer, or by "decapitating" the root nodes and concatenating all first-level children. It is easy to see that in both cases, the optimal source navigations for this query just mirror the given client navigations. Eventually q_{conc} is bounded by a linear function. Hence the mediator can provide a very strong guarantee regarding the time it takes to respond to a navigation command on such a view. We will call this class of views bounded browsable.

Conversely, consider a view q_θ which picks all first-level subelements of a source, which satisfy some property θ. Assume the client asks for the label of the first child in the virtual view. This is accomplished by the navigation $c = d, f$. However, the length of the corresponding source navigation $s = d, f, r, f, r, \ldots$ depends on the source data, i.e., when we find the first child which satisfies θ. We will call such a view unbounded browsable in order to indicate the good news and the bad news: It may be possible to process the user request by retrieving just a (small) part of the source but, at the same time, the mediator cannot provide a strong guarantee on the processing time.

Finally, consider a view that *orders* the selected elements according to some arithmetic attribute such as age. Navigating such a view is very inefficient because the mediator cannot respond to the user until it has seen the complete list of age elements. We will call such a view unbrowsable. □

First we classify the complexity of client navigations on views. Let q be a view and c a sequence of client navigations.

Unbrowsable: A navigation sequence c on a view q is called *unbrowsable*, if in order to compute the result of c on $q(t)$, the computation requires access to at least one list of t in its entirety, independent of the input t.

Browsable: Conversely, a sequence c on q is called *browsable*, if the result of c on $q(t)$ can be computed without accessing any list of t in its entirety.

Bounded Browsable: Finally, a sequence c on q is called *bounded browsable*, if it is browsable and there is a function f such that the length m of the required source navigation is $\leq f(n)$, where n is the length of $c = c_1, \ldots, c_n$.

Based on this, a query q is called *unbrowsable*, if there is a c such that c on q is unbrowsable. A query q is *(bounded) browsable*, if for all c, c on q is (bounded) browsable.

Note that the degree boundedness depends on the given set of navigation commands. For example, if **NC** includes the sibling selection σ_θ, the last query turns from unbrowsable into browsable, since one source command is sufficient to retrieve the first child satisfying θ.

3 Query Evaluation

We employ the XMAS algebra described below in order to formulate, optimize, and evaluate XMAS query plans. Figure 3 shows a simple XMAS view which retrieves all homes having a school within the same zip code region.[5]

```
CONSTRUCT <ans> $H {$H} </ans>
WHERE <homes>
        $H: <home>
              $Z1:  <zip> $V1 </>
            </home>
      </homes> IN "http://...homesSrc.xml"
AND   <schools>
        $S: <school>
              $Z2:  <zip> $V2 </>
            </school>
      </schools> IN "http://...schoolsSrc.xml"
AND   $V1=$V2 .
```

Figure 3: XMAS query q

Query processing involves the following steps:

Preprocessing: At compile-time, a XMAS view q is first translated into an equivalent algebra expression E_q that constitutes the *initial view plan.* Note that in practice the interaction of the client with the mediator may start by issuing a query q' on q.

[5]The *collection label* {$H} defines that each binding of $H creates one subelement of the ans element [BGL+99]. XMAS includes many other features, for example, vertical and horizontal path expressions [LPVV99].

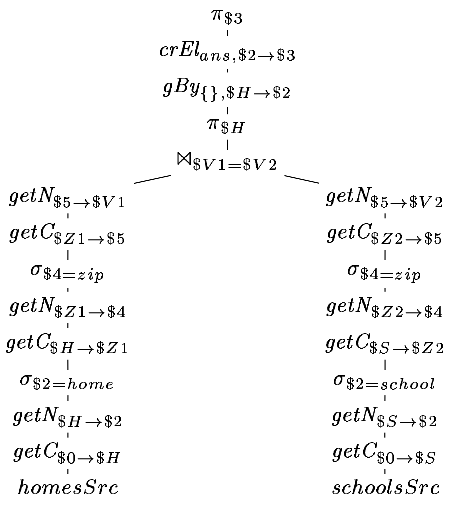

Figure 4: Plan (algebra expression) E_q

In this case the preprocessing phase will compose the query and the view and generate the initial plan of $q' \circ q$.

Query Rewriting: Next, during the *rewriting phase*, the initial plan is rewritten into a plan E'_q which is optimized wrt. navigational complexity.

Query Evaluation: At run-time, client navigations into the virtual view, i.e., into the result of q are translated into source navigations. This is accomplished by implementing each algebra operator *op* as a *lazy mediator* m_{op}. Thus, the optimized plan E'_q corresponds to a *tree of lazy mediators.*

The modularization of the plan provides important benefits: In the overall DOM-VXD architecture (Figure 1), lazy mediators $m_{q_1}, m_{q_2}, \ldots$ can be organized in a tree-like structure; communication between mediators is through the navigations. By translating each m_{q_i} into a plan E_{q_i} which itself is a tree consisting of "little" lazy mediators (one for each algebra operation), we obtain a smoothly integrated, uniform evaluation scheme. Furthermore, the mediator optimizes such plans using a rewriting optimizer. Finally, it is easy to incorporate additional operators (e.g., aggregates) in the view definition language by incorporating additional operators in the algebra.

3.1 The XMAS Algebra

In the XMAS query language, variables bind to atomic values (strings from $\mathbf{D}$), or to XML elements (trees from $\mathbf{T}$). Thus, in the XMAS algebra, a binding for k variables can be represented as a tuple $b = (b.1, \ldots, b.k)$ where each $b.i$ refers to a value from $\mathbf{D}$ or $\mathbf{T}$.

In order to realize an algebra operator *op* as a lazy mediator m_{op}, we represent its inputs (i.e., one or more lists of variable bindings) as tree-like structures. More precisely, each input to an operator is a *list of variable bindings*, represented by a root element r, whose children $b_1, b_2, \ldots$ represent tuples. Each tuple $b_i = b_i.1, \ldots, b_i.k$ encodes the i-th binding for the given k variables. Thus, the index j identifies the j-th *column* of the list of variable bindings. Instead of the variable index j, we sometimes use the variable name.

Several variable bindings can refer to the same elements of the input, so the described structure contains shared subtrees, i.e., is a *labeled ordered graph* (*log*). Thus, when evaluating a XMAS query q using an algebraic plan E_q, input trees $t_{in} \in \mathbf{T}$ may become logs. However, eventually, the intermediate logs will be expanded, so the final answer will be a tree $t_{out} \in \mathbf{T}$ again.

Algebra Operators. Every XMAS algebra operator specifies a mapping from one or more input logs to an output log, so operators can be nested and the algebra is closed. The algebra includes lazy versions of the conventional relational operators and a few operators know from object-oriented algebras. In particular, the usual relational operators σ, π, $\bowtie$, $\times$, $\cup$, and $\setminus$ are appropriately modified to operate on logs.

The non-standard operators defined below operate on a single input log only. The notation $op_{x_1,\ldots,x_n\to y}$ indicates that op is applied to input operands $x_1, \ldots, x_n$, resulting in the output y (x_j and y are variable indexes or names). Thus, for a binding b, *op* uses $b.x_1, \ldots, b.x_n$ as input and $b.y$ as output.

Operators as Queries. First, we provide an informal description of the main operators as mappings from logs to logs, without considering their navigational behavior. b_{in} and b_{out} denote tuples (holding variable bindings) from the input and output log, respectively.

- $createElement_{name,ch\to e}$ (*crEl*) creates a new element *el* for each input tuple b_{in} with $label(el) = label(b_{in}.name)$ and where the subelements of *el* are the subelements of $b_{in}.ch$. *el* is placed into $b_{out}.e$ (so n, ch, and e identify the *name*, *children* and *new element* column, respectively).

- $getChildren_{e\to ch}$ (*getC*) extracts all children $c_1, \ldots, c_k$ of $b_{in}.e$ and for each c_i creates an output tuple b_{out} where $b_{out}.ch = c_i$.

- $getName_{e\to name}$ (*getN*) extracts the label of $b_{in}.e$ for each input tuple b_{in} and creates an output tuple

b_{out}, where $b_{out}.name$ is a new element whose label equals $label(b_{in}.e)$.

- *groupBy*$_{g_1,\ldots,g_k,v\to l}$ (*gBy*) groups all $b_{in}.v$'s by the values of the group-by elements $b_{in}.g_1,\ldots,b_{in}.g_k$. The output tuple comprises the children $b_{in}.g_1,\ldots,b_{in}.g_k$ followed by a new element *collection*, whose subelements are the $b_{in}.v$'s which appear together with the corresponding $b_{in}.g_1,\ldots,b_{in}.g_k$ in the input.

- *orderBy*$_{x_1,\ldots,x_k}$ (*ordBy*) orders all tuples b_{in} according to the occurrence of $b_{in}.x_1,\ldots,b_{in}.x_k$ in the input.

- *concatenate*$_{x,y\to z}$ (*concat*) concatenates subelements of $b_{in}.x$ and $b_{in}.y$ for each input tuple b_{in} and creates a new element *conc* in $b_{out}.z$. The subelements of *conc* are the concatenation of subelements of $b_{in}.x$ and $b_{in}.y$.

Example 2 (XMAS to Algebra) Figure 4 shows the equivalent initial plan E_q for the view in Figure 3. □

Operators as Lazy Mediators. In order to implement operators as lazy mediators, we have to define their navigational input/output behavior. We model lazy mediators as state machines whose input are client navigations which are translated by the state machine into outgoing source navigations. In Figures 5 and 6 we annotate transitions from `state` to `state'` with triplets $c/s/\langle p_1,\ldots,p_k\rangle$, denoting the incoming client navigation, the outgoing source navigation, and pointer updates, respectively. The result of sending s to the source is then returned to the client above, together with $\langle p_1,\ldots,p_k\rangle$ and `state'`. We confine ourselves to the description of the lazy mediators for *createElement*, *getChildren*, and *groupBy*.

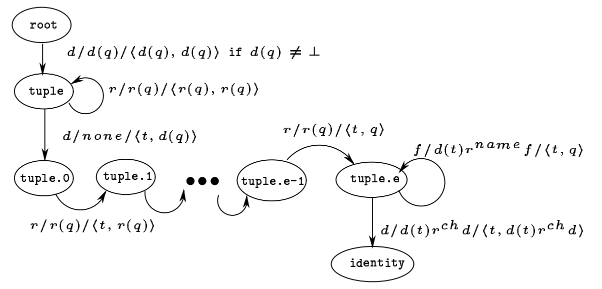
root
$d/d(q)/\langle d(q), d(q)\rangle$ if $d(q) \neq \perp$
tuple
$r/r(q)/\langle r(q), r(q)\rangle$
$d/none/\langle t, d(q)\rangle$
tuple.0
tuple.1
tuple.e-1
$r/r(q)/\langle t, q\rangle$
tuple.e
$f/d(t)r^{name}f/\langle t, q\rangle$
$r/r(q)/\langle t, r(q)\rangle$
$d/d(t)r^{ch}d/\langle t, d(t)r^{ch}d\rangle$
identity

Figure 5: Lazy mediator for *createElement*

- *createElement*$_{name,ch\to e}$: the state machine for this lazy mediator is depicted in Figure 5. It maintains a tuple $\langle t, q\rangle$ of pointers into the virtual result of the lazy mediator (or source) below. t identifies the current input tuple and is used to navigate to the *name* and *ch* elements of the input, whereas q just mirrors the client navigations in the source. The key transitions are outgoing from `tuple.e`: In case of an incoming command f (fetch), we create the source navigation $d(t), r^{name}, f$, i.e., we first move down from the current tuple t, then move right "*name*" times, and finally fetch the label of the reached element. The source pointers t, q are not changed. In case of d (down), we have to update the source pointer q to point to the children list of the source using a navigation $d(t), r^{ch}, d$. The result of this source navigation is returned to the client. When entering the `identity` state, the operator just mirrors incoming client navigations at the source. All transitions not shown in the figure implicitly lead to `identity`. This establishes that *createElement* is a *bounded operator*.

- *getChildren*$_{e\to ch}$ (Figure 6): this operator uses pointers $\langle c, q\rangle$, where c keeps track of the current child element, and q is used as in *createElement*. When receiving r in state `tuple`, we either navigate to the next child of the current input element, or to the next tuple that contains a child. Note that this behavior corresponds to a loop and thus turns the *getChildren* into an *unbounded operator*.

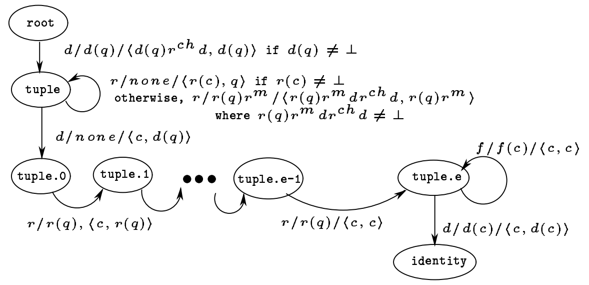
root
$d/d(q)/\langle d(q)r^{ch}d, d(q)\rangle$ if $d(q) \neq \perp$
tuple
$r/none/\langle r(c), q\rangle$ if $r(c) \neq \perp$
otherwise, $r/r(q)r^{m}/\langle r(q)r^{m}dr^{ch}d, r(q)r^{m}\rangle$
where $r(q)r^{m}dr^{ch}d \neq \perp$
$d/none/\langle c, d(q)\rangle$
tuple.0
tuple.1
tuple.e-1
tuple.e
$f/f(c)/\langle c, c\rangle$
$r/r(q), \langle c, r(q)\rangle$
$r/r(q)/\langle c, c\rangle$
$d/d(c)/\langle c, d(c)\rangle$
identity

Figure 6: Lazy mediator for *getChildren*

- *groupBy*$_{g_1,\ldots,g_k,v\to l}$: In addition to the usual states, this operator stores the current group-by list $\bar{g} = g_1,\ldots,g_k$, a set of previously encountered group-by lists G_{prev} and three pointers $\langle p, q, s\rangle$. p references the first input tuple that contains the current group-by list $\bar{g}$, q keeps track of the current input tuple, and r corresponds to the current source navigation.

The non-trivial transitions involve navigations to the next tuple of the output and navigations into the collection of grouped variables. The r command at the `tuple` state must navigate to the next input tuple which has a new group-by list. So the operator sends r commands to the underlying mediator until a group-by list is found which is not in G_{prev}.

The second relevant transition is the navigation into the collection of grouped elements. When the client navigates down to the first grouped element of the collection, the s pointer is set to the v-th child of the input tuple. Successive r commands scan the input tuples until a tuple is found whose group-by list coincides with $\bar{g}$.

4 Discussion and Conclusions

We have presented a novel mediation framework, DOM-VXD, for evaluating queries against virtual mediated XML views. In our approach query evaluation is lazy and completely driven by client DOM navigations. This is accomplished by implementing virtual views through *lazy mediators*, which serve incoming client navigations by sending corresponding navigations to the sources and returning the processed answers.

The idea of demand-driven lazy evaluation is related to pipelined plan execution in relational databases (see e.g. [GMUW99]). However, in the case of (XML) trees the client navigation may proceed at multiple incomplete points, while in relational databases a client navigation may only proceed at the current cursor position. Another major difference to the relational case is that the presence of order may change the navigational complexity and implementation of views and lazy mediators. Further note that bounded browsable and unbrowsable queries are somewhat similar to tuple and global predicates, respectively (i.e., whose output depends solely on a tuple, or on the whole relation).

Mediator views are defined in XMAS, a declarative query language in the spirit of XML-QL [XML98a] and translated by the system into equivalent XMAS algebra expressions. The notion of *navigational complexity* relates the client request with the necessary source navigations and allows the classification of queries according to their degree of browsability.

A key idea for obtaining a smoothly integrated navigation-driven architecture is to design each algebra operator as a lazy mediator itself. In general, an evaluation plan E_q consisting of nested lazy mediators can be less efficient wrt. navigations than the underlying view q. For example, the direct translation of XMAS queries into an initial plan often introduces *orderBy* operators which result in unbrowsable plans.

A prototype of the MIX mediator system, employing an eager approach for evaluating XMAS algebra expressions, has been implemented [BGL+99]. It employs a novel DTD-oriented query interface BBQ which blends browsing and querying of XML data, similar in spirit to GARLIC's PESTO interface [CHMW96]. Currently, the MIX mediator is being extended to support the navigation-driven DOM-VXD architecture.

An important issue for future research is the extension of XMAS and its algebra to include nondeterminism in the order of lists. This allows additional optimizations whenever the order of the final result is irrelevant.

Acknowledgments. We thank all members of the "MIX team" at SDSC and CSE/UCSD for many fruitful discussions.

References

[BGL+99] C. Baru, A. Gupta, B. Ludäscher, R. Marciano, Y. Papakonstantinou, and P. Velikhov. XML-Based Information Mediation with MIX. In *ACM SIGMOD*, Philadelphia, 1999. (system demonstration).

[CDSS98] S. Cluet, C. Delobel, J. Simeon, and K. Smaga. Your Mediators Need Data Conversion! In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, pp. 177–188, 1998.

[CHMW96] M. J. Carey, L. M. Haas, V. Maganty, and J. H. Williams. PESTO: An Integrated Query/Browser for Object Databases. In *Intl. Conference on Very Large Data Bases (VLDB)*, pp. 203–214, 1996.

[DOM98] Document Object Model (DOM) Level 1 Specification. W3C recommendation, `www.w3.org/TR/REC-DOM-Level-1/`, 1998.

[GMUW99] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database System Implementation.* 1999. to appear.

[LPVV99] B. Ludäscher, Y. Papakonstantinou, P. Velikhov, and V. Vianu. View Definition and DTD Inference for XML. In *Post-ICDT Workshop on Query Processing for Semistructured Data and Non-Standard Data Formats*, Jerusalem, 1999. `www-rodin.inria.fr/external/ssd99/workshop.html`.

[MIX99] Mediation of Information using XML (MIX). `www.npaci.edu/DICE/MIX/` and `www.db.ucsd.edu/Projects/MIX/`, 1999.

[XML98a] XML-QL: A Query Language for XML. W3C note, `www.w3.org/TR/NOTE-xml-ql`, 1998.

[XML98b] Extensible Markup Language (XML) 1.0. W3C recommendation, `www.w3.org/TR/REC-xml`, 1998.