Navigation-Driven Evaluation of Virtual Mediated Views

Bertram Ludäscher Yannis Papakonstantinou Pavel Velikhov

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

Abstract. The MIX mediator systems incorporates a novel framework for navigation-driven evaluation of virtual mediated views. Its architecture allows the *on-demand* computation of views and query results as the user navigates in them. The evaluation scheme minimizes superfluous source access through the use of *lazy mediators* that translate incoming client navigations on a virtual XML view into navigations on lower level mediators or wrapped sources. This evaluation scheme gives rise to a notion of *navigational complexity*, which yields a coarse classification of queries according to their "browsability." The proposed 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 non-materialization of the query answer is transparent to the client application since clients can navigate the query answer using a subset of the standard DOM API for XML documents. We elaborate on query evaluation in such a framework; in particular, we show how algebraic plans can be implemented as trees of lazy mediators. Finally, we present a new buffering technique that can mediate between the fine granularity of DOM navigations and the coarse granularity of real world sources. This drastically reduces communication overhead and at the same time simplifies wrapper development. An implementation of the system is available on the Web.

1 Introduction and Overview

Mediated views integrate information from heterogeneous sources. There are two main paradigms for evaluating queries against integrated views: In the *warehousing* approach, data is collected and integrated in a materialized view *prior* to the execution of user queries against the view. However, when the user is interested in the most recent data available or very large views, then a *virtual*, *demand-driven* approach has to be employed. Most notably such requirements are encountered when integrating Web sources. For example, consider a mediator that creates an integrated view, called `allbooks`, of data on books available from `amazon.com` and `barnesandnoble.com`. A warehousing approach is not viable: First, one cannot obtain the complete dataset of the booksellers. Second, the data will have to reflect the ever-changing availability of books. In contrast, in a demand-driven approach, the user query is composed with the view definition of `allbooks` and corresponding subqueries against the sources are evaluated only then (i.e., at query evaluation time and not a priori).

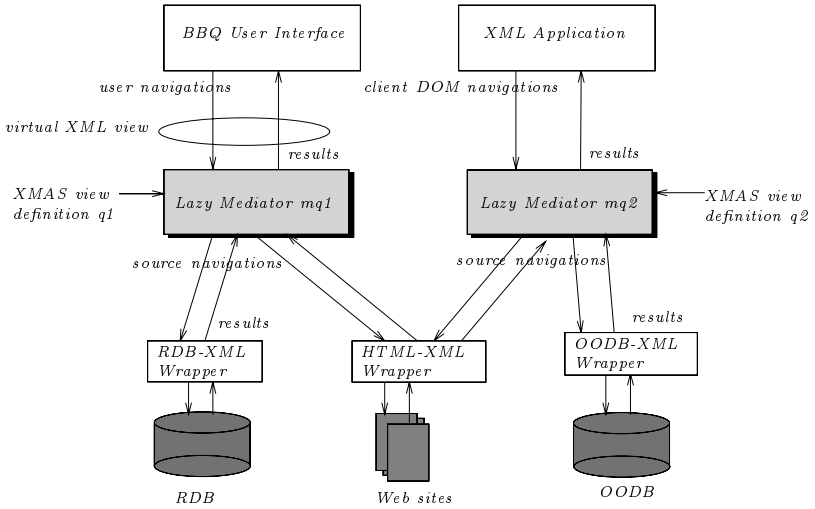

Figure 1: Virtual XML Document (VXD) mediation architecture

Current mediator systems, even those based on the virtual approach, compute and return the results of the user query *completely*. Thus, although they do not materialize the integrated view, they materialize the result of the user query. This approach is unsuitable in Web mediation scenarios where the users typically do not specify their queries precisely enough to obtain small results. Instead, they often issue relatively broad queries, navigate the first few results and then stop either because the results seem irrelevant or because the desired data was already found. In the context of such interactions materializing the full answer on the client side is not an option. Instead, it is preferable to produce results as the user navigates into the virtual answer view, thereby reducing the response time.

In this paper, we present a novel mediator framework and query evaluation mechanism that can efficiently handle such cases. We use XML, the emerging standard for data exchange, as the data model of our mediator architecture (Figure 1). User queries and view definitions are expressed in XMAS[1], a declarative XML query language borrowing from and similar to other query languages for semistructured data like XML-QL and Lorel [XML98a, AQM+97].

The key idea of the framework is simple and does not complicate the client's code: The client receives a *virtual* answer document (object) in response to his query. This document is not computed or transfered into the client memory until the user starts navigating into it. The virtuality of the answer document is completely transparent to the client who accesses the virtual document using (a subset of) the DOM API, i.e., in exactly the same way as main memory resident XML documents.

Query evaluation is navigation-driven in the VXD architecture: The client application first sends a

[1] ***X**ML **M**atching **A**nd **S**tructuring Language* [LPVV99]

query to the mediator which then composes the query with the view definition and translates the result into an algebraic evaluation plan. After this preprocessing phase, the mediator returns a "handle" to the root element of the virtual XML answer document without even accessing the sources (cf. Sections 3,5). When the client starts navigating into the virtual answer, the mediator has to translate these navigations into navigations against the sources. This is accomplished by implementing each algebra operator of the evaluation plan as a *lazy mediator*, i.e., a kind of transducer that translates incoming navigations from above into outgoing navigations and returns the corresponding answer fragments. The overall algebraic plan then corresponds to a tree of lazy mediators through which results from the sources are pipelined upwards, *driven by the navigations* which flow downwards from the client. For the navigation commands, we use a subset of DOM[2], the standard API for XML documents. The outline and contributions of the paper are as follows:

In Section 2 we introduce lazy mediators and our navigation model, called DOM-VXD (DOM for ***V***irtual ***X***ML ***D***ocuments). Since navigation-driven query evaluation differs significantly from the usual setting, we propose a new notion of *navigational complexity*, which yields a coarse classification for comparing the cost of query evaluation in a framework like ours.

Section 3 elaborates on query evaluation: The XMAS algebra is presented and it is shown how its operators are implemented as lazy mediators.

The fine grained nature of the DOM navigations is the basis of our demand-driven evaluation model. However, communicating only small pieces of data in this way can result in considerable overhead. Section 4 refines the architecture by introducing a buffer component between mediators and sources, thereby reconciling the fine granularity of our navigation model and the coarse granularity of results returned by real sources. To this end, we present a simple, yet flexible, XML fragment exchange protocol and the corresponding buffer algorithms.

A Java implementation of the MIX mediator and its navigation-driven evaluation can be found at [MIX99] (cf. Section 5). In Section 6, conclusions and future directions are given.

Related Work. Our navigation-driven architecture extends the virtual view mediator approach as, for example, used in TSIMMIS, YAT, Garlic, and Hermes [PAGM96, CDSS98, CHN+95, HER]. The idea of navigation-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 from multiple nodes whose descendants or siblings have not been visited yet. In contrast, in relational databases a client "navigation" may only proceed at the current cursor position. Another major difference to the relational

[2] ***D***ocument ***O***bject ***M***odel [DOM98]

case is that the presence of order in general changes the navigational complexity and implementation of lazy mediators.

Our XML query language XMAS borrows from similar languages such as XML-QL, MSL, FLORID, Lorel, and YAT [XML98a, PAGM96, FLO, AQM+97, CDSS98]. However, most of the above rely on Skolem functions for grouping, while XMAS uses explicit group-by operators thereby facilitating a direct translation of the queries into an algebra. In that sense our implementation is closer to implementations of the nested relational and complex values models.

A preliminary abstract on the navigation-driven MIX architecture has appeared in [LPV99].

2 Navigations and Browsability in the VXD Framework

We employ XML as the data model [XML98b]. Note that the techniques presented here are not specific to XML and are applicable to other semistructured data models and query languages. The paper uses the following abstraction of XML where, for simplicity, we have excluded attributes.[3]

XML documents are viewed as *labeled ordered trees* (from now on referred to simply as *trees*) over a suitable underlying domain $\mathbf{D}$.[4] The set of all trees over $\mathbf{D}$ is denoted by $\mathbf{T}$.

A tree $t \in \mathbf{T}$ is either a *leaf*, i.e., a single atomic piece of data $t = d \in \mathbf{D}$, or it is $t = d[t_1, \ldots, t_n]$, where $d \in \mathbf{D}$ and $t_1, \ldots, t_n \in \mathbf{T}$. We call d the *label* and $[t_1, \ldots, t_n]$ the *ordered list of subtrees* (also called *children*) of t.

In XML parlance, t is an *element*, a non-leaf label d is the *element type* (or *tag name*), $t_1, \ldots, t_n$ are *child elements* (or *subelements*) of t, and a leaf label d is an atomic object such as character content or an empty element. In short, the set $\mathbf{T}$ of labeled ordered trees can be described by the signature

$$\mathbf{T} = \mathbf{D} \ \mid \ \mathbf{D}[\mathbf{T}^*] \ .$$

Lazy Mediators. A *lazy mediator* m_q for a query (or view definition[5]) q operates as follows: The client navigates into the virtual view exported by m_q by successively issuing DOM-VXD *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.

[3] See the system description at [MIX99] for the details of how we incorporated attributes in the implementation.

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

[5] Queries can be regarded as view definitions and vice versa, so we use the terms interchangeably.

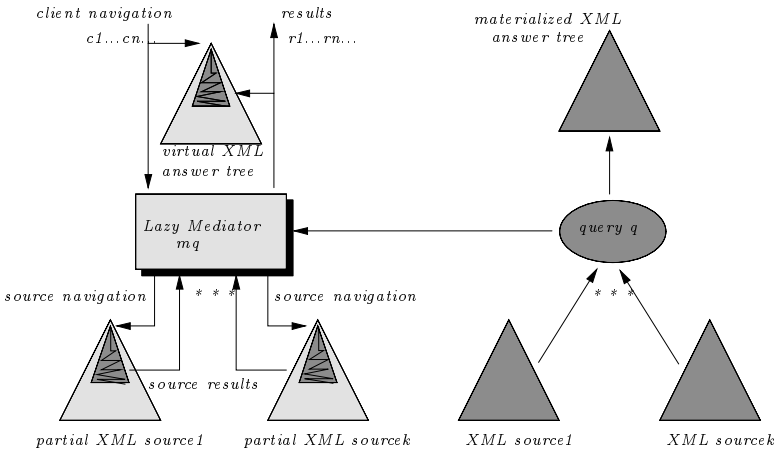

Figure 2: Navigational interface of a lazy mediator

DOM-VXD Navigation Commands. XML documents (both source and answer documents) are accessible via navigation commands. We present the navigational interface DOM-VXD that is an abstraction of a subset of the DOM API for XML. More precisely, we consider the following set **NC** of *navigation commands*, where p and p' are node-id's of (pointers into) the virtual document that is being navigated:

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

This minimal set of navigation commands is sufficient to completely explore arbitrary virtual documents. Additional navigation commands can be provided in the style of [XPo]. For example, we may include in **NC** a command for selecting certain siblings:

- *select* (σ_θ): $p' := \sigma_\theta(p)$ assigns to p' the first sibling to the right whose label satisfies θ (else $\perp$).

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

$$p'_0 := c_1(p_0),\ p'_1 := c_2(p_1),\ \ \ldots\ ,\ p'_{n-1} := c_n(p_{n-1})$$

where each $c_i \in \mathbf{NC}$ and each p_i is a p'_j with $j < i$.

The *result* (or *explored part*) $c(t)$ of applying the navigation c to a tree t is the unique subtree comprising only 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., p'_{n-1}. For notational convenience, we sometimes omit the pointer argument and simply write $c = c_1, \ldots, c_n$. □

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

Example 1 (Browsability) Consider a view q_{conc} that computes the concatenation of first level elements of two sources by "decapitating" the root nodes and concatenating all first-level children. It is easy to see that the required source navigations for this view just mirror the given client navigations. Hence the mediator can provide a very strong guarantee regarding the number of source navigations it takes to respond to a client navigation on such a view. We will call this class of views *bounded browsable*.

Conversely, consider a view q_θ that picks all first-level children whose label satisfy a 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 $s = d, f, r, f, r, ...$ 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 number of source navigations.

Finally, consider a view that *reorders* the selected elements according to some arithmetic attribute such as `age`. Browsing (navigating) such a view is 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*. □

Definition 2 (Browsability) Let q be a view definition, $c = c_1, \ldots, c_n$ a sequence of client navigations.

- The sequence c on 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.
- Conversely, the sequence c on q is called *browsable*, if the result of c on $q(t)$ may be computed without accessing any list of t in its entirety.
- Finally, the 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 $\leq f(n)$.

Based on the above, q is called *unbrowsable*, if there is a c such that c on q is unbrowsable. Conversely, q is *(bounded) browsable*, if for all c, c on q is (bounded) browsable. □

Note that the degree of browsability depends on the given set of navigation commands. For example, if **NC** includes the sibling selection σ_θ, the query q_θ in Example 1 becomes bounded browsable, since one source command is sufficient to retrieve the first child satisfying θ.

```
CONSTRUCT <answer>                                  % Construct the root element containing ...
             <med_home> $H                          % ... med_home elements followed by
                        $S {$S}                     % ... school elements (one for each $S)
             </med_home> {$H}                       % (one med_home element for each $H)
          </answer> {}                              % create one answer element (= for each {})
WHERE homesSrc homes.home $H AND $H zip._ $V1       % get home elements $H and their zip code $V1
AND   schoolsSrc schools.school $S AND $S zip._ $V2 % ... similarly for schools
AND   $V1 = $V2                                     % ... join on the zip code
```

Figure 3: A XMAS query q

3 Query Evaluation

Query processing in the MIX mediator system involves the following steps:

Preprocessing: At compile-time, a XMAS mediator view q is first translated into an equivalent algebra expression E_q that constitutes the *initial plan*. The interaction of the client with the mediator may start by issuing a query q' on q. In this case the preprocessing phase will compose the query and the view and generate the initial plan for $q' \circ q$. In the balance of the paper q will also denote the composition.

Query Rewriting: Next, during the *rewriting phase*, the initial plan is rewritten into a plan E'_q which is optimized with respect to navigational complexity. Due to space limitations we do not present rewriting rules.

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} that transforms incoming navigations (from the client or the operator above) into navigations that are directed to the operators below or the wrappers.

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, these plans may be optimized with respect to navigational efficiency by means of rewriting optimizers. Finally, as we will show below, it is relatively easy to implement individual algebra operators that map incoming client navigations to outgoing navigations against the sources.

Example 2 (Homes with Local Schools) Figure 3 shows a simple XMAS query that will serve as our running example. The query assumes two sources, `homeSrc` and `schoolsSrc` and retrieves all homes having a school within the same zip code region. For each such home the query creates a `med_home` element that contains the home followed by all schools with the same zip code. The body (`WHERE` clause) includes

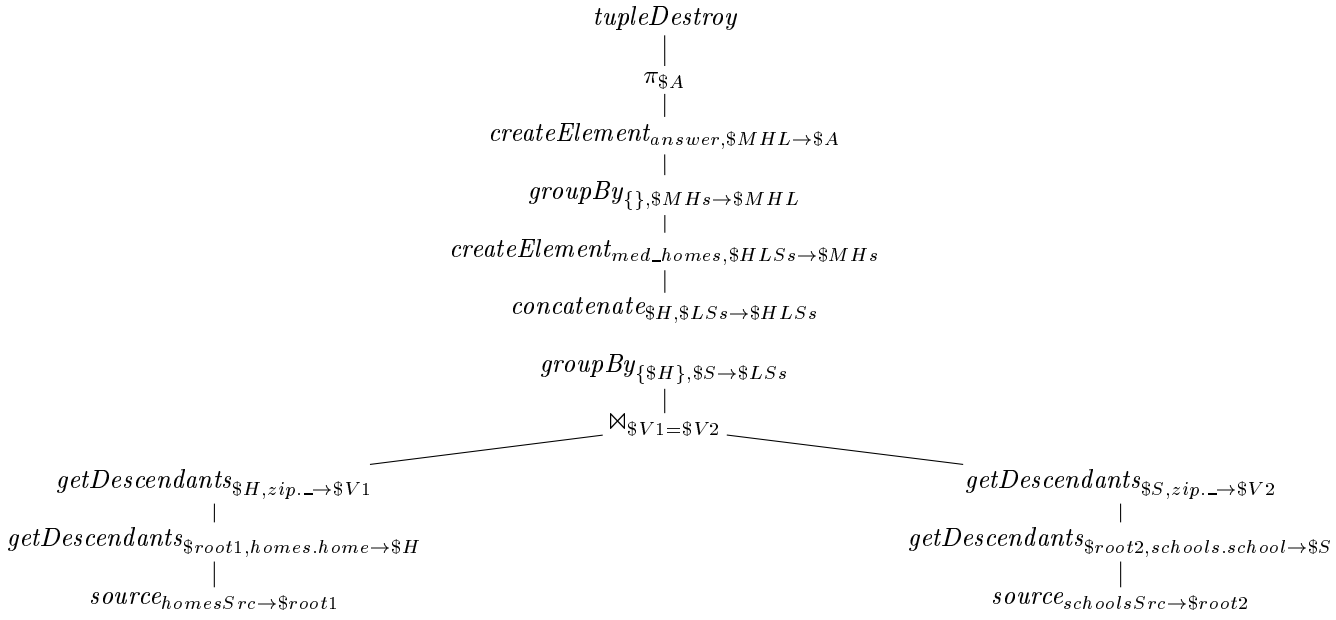

Figure 4: Plan (algebra expression) E_q

generalized path expressions, as in Lorel [AQM+97] and generalized OQL expressions [CCM96]. Bindings to the variables are generated as the path expressions are matched against the document. In our example `$H` binds to `home` trees, reachable by following the path `homes.home` from the root of `homesSrc`; `$S` binds to `school` trees. The result of evaluating the body is a list of variable bindings.[6]

The head (`CONSTRUCT` clause) of the query describes how the answer document is constructed based on the variable bindings from the body. In particular, the clause `<med_home> ... </med_home> {$H}` dictates that for each binding h of `$H` exactly one `med_home` tree is created. For each such h the corresponding `med_home` contains h, followed by the list of all bindings s of `$S` such that (h, s) is contained in a binding of the body. For a more detailed exposition of XMAS see [LPVV99]. □

The XMAS Algebra. Each XMAS query has an equivalent XMAS algebra expression. The algebra operators input lists of variable bindings and produce new lists of bindings in the output. We represent lists of bindings as trees[7] to facilitate the description of operators as lazy mediators. For example, the list

[6] XMAS also supports tree patterns in the style of XML-QL, e.g., `<homes> $H: <home> <zip>$V1</zip> </home> </homes> IN homesSrc` is the equivalent of the first line in the `WHERE` clause in Fig. 3.

[7] Strictly speaking, 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*. This preserves node identities which are needed when grouping elements, eliminating duplicates and preserving the input order of elements.

of variable bindings $[(\$X/x_1, \$Y/y_1), (\$X/x_2, \$Y/y_2)]$ is represented as the following tree

$$\mathsf{bs}[\ \mathsf{b}[\ X[x_1], Y[y_1]\ ],\ \mathsf{b}[\ X[x_2], Y[y_2]\ ]\ ]\ .$$

Here, the $\mathsf{bs}[\ldots]$ element holds a list of variable bindings $\mathsf{b}[\ldots]$.

By b_i we denote the i-th element of bs; the notation $\mathsf{b}_i + X[v]$ adds the binding $(\$X/v)$ to b_i. $\mathsf{b}_i.X$ denotes the value of X for b_i.

Algebra Operators. The XMAS algebra includes the conventional relational operators σ, π, $\bowtie$, $\times$, $\cup$, and $\setminus$ (now operating on lists of bindings bs) along with operators that extend the nested relational algebras' nest/unnest operators with generalized path expressions. The notation $op_{x_1,\ldots,x_n \to y}$ indicates that op creates new bindings for y, given the bindings for $x_1, \ldots, x_n$.

First, we describe the semantics of the main operators as mappings from one or more input trees to the output tree. Their implementation as lazy mediators, which specifies their navigational behavior, is presented subsequently. In the following, b_{in} and b_{out} denote variable bindings from the input and the output of the operators, respectively.

- $getDescendants_{e,re \to ch}$ extracts descendants of the parent element $\mathsf{b}_{in}.e$ which are reachable by a path ending at the extracted node, such that this path matches the regular expression re. We consider the usual operators ".", "|", "*",... for path expressions; "_" matches any label (cf. Fig. 4). For each input binding b_{in} and retrieved descendant d, *getDescendants* creates an output binding $\mathsf{b}_{in} + ch[d]$.

  For example (cf. Fig. 4), $getDescendants_{\$H,zip._ \to \$V1}$ evaluated on the list of bindings:

  $$\begin{array}{l}\mathsf{bs}[\ \mathsf{b}[\ H[\ home[addr[La\ Jolla],\ zip[91220]]\ ]\ ]\\ \quad\ \mathsf{b}[\ H[\ home[addr[El\ Cajon],\ zip[91223]]\ ]\ ]\ ]\end{array}$$

  produces the list of bindings:

  $$\begin{array}{l}\mathsf{bs}[\ \mathsf{b}[\ H[\ home[addr[La\ Jolla],\ zip[91220]]\ ],\ V1[\ 91220\ ]]\\ \quad\ \mathsf{b}[\ H[\ home[addr[El\ Cajon],\ zip[91223]]\ ],\ V1[\ 91223\ ]]\ ]\end{array}$$

- $groupBy_{\{v_1,\ldots,v_k\},v \to l}$ groups the bindings $\mathsf{b}_{in}.v$ by the bindings of $\mathsf{b}_{in}.v_1, \ldots, \mathsf{b}_{in}.v_k$ (so $v_1, \ldots, v_k$ are the *group-by variables.*) For each group of bindings in the input that agree on their group-by variables, one output binding $\mathsf{b}[\ v_1[\mathsf{b}_{in}.v_1], \ldots, v_k[\mathsf{b}_{in}.v_k], l[\mathsf{list}[coll]]]$ is created, where $coll$ is the list of all values belonging to this group and list is a special label for denoting lists. For example, $groupBy_{\{\$H\},\$S \to \$LSs}$ applied to the input

  $$\begin{array}{l}\mathsf{bs}[\ \mathsf{b}[\ H[\ home[addr[La\ Jolla],\ zip[91220]]\ ],\ S[\ school[dir[Smith],\ zip[91220]]\ ]\ ]\\ \quad\ \mathsf{b}[\ H[\ home[addr[La\ Jolla],\ zip[91220]]\ ],\ S[\ school[dir[Bar],\ zip[91220]]\ ]\ ]\\ \quad\ \mathsf{b}[\ H[\ home[addr[El\ Cajon],\ zip[91223]]\ ],\ S[\ school[dir[Hart],\ zip[91223]]\ ]\ ]\ ]\end{array}$$

will yield the output

$$\mathsf{bs}[\ \mathsf{b}[\ H[\ home[addr[La\ Jolla], \ldots\ ],\ LSs[\ \mathsf{list}[school[dir[Smith],\ \ldots], \ school[\ dir[Bar],\ \ldots]\ ]]\ ]$$
$$\mathsf{b}[\ H[\ home[addr[El\ Cajon],\ \ldots\ ],\ LSs[\ \mathsf{list}[school[dir[Hart], \ldots\ ]\ ]\ ]\ ]\ ]$$

- $concatenate_{x,y \to z}$ concatenates subtrees or lists of subtrees of $\mathsf{b}_{in}.x$ and $\mathsf{b}_{in}.y$, depending on their types. For each input tuple b_{in}, *concatenate* produces $\mathsf{b}_{in} + z[conc]$, where *conc* is:
  - $\mathsf{list}[x_1, \ldots, x_n, y_1, \ldots, y_n]$ if $\mathsf{b}_{in}.x = \mathsf{list}[x_1 \ldots x_n]$ and $\mathsf{b}_{in}.y = \mathsf{list}[y_1 \ldots y_n]$.
  - $\mathsf{list}[x_1, \ldots, x_n, v_y]$ if $\mathsf{b}_{in}.x = \mathsf{list}[x_1 \ldots x_n]$ and $\mathsf{b}_{in}.y = v_y$.
  - $\mathsf{list}[x, y_1, \ldots, y_n]$ if $\mathsf{b}_{in}.x = v_x$ and $\mathsf{b}_{in}.y = \mathsf{list}[y_1 \ldots y_n]$.
  - $\mathsf{list}[v_x, v_y]$ if $\mathsf{b}_{in}.x = v_x$ and $\mathsf{b}_{in}.y = v_y$.
- $createElement_{label,ch \to e}$ creates a new element for each input binding. The parameter *label* specifies the label of the new element and can be either a constant or a variable. The subtrees of the new element are the subtrees of $\mathsf{b}_{in}.ch$. Thus, for each input binding b_{in}, *createElement* outputs a binding $\mathsf{b}_{in} + e[l[c_1, \ldots c_n]]$, where l is the value of $b_{in}.label$ and $c_1, \ldots, c_n$ are the subtrees of $b_{in}.ch$. E.g., $createElement_{med\_homes,\$HLSs \to \$MHs}$ where $\$HLSs$ results from $concatenate_{\$H,\$LSs \to \$HLSs}$ applied to the $\$H$ and $\$LSs$ in the output of the above $groupBy_{\{\$H\},\$S \to \$LSs}$ yields:

$$\mathsf{bs}[\ \mathsf{b}[\ H[\ \ldots\ ],\ LSs[\ \ldots\ ],\ MHs[\ med\_home[\ school[dir[Smith], \ldots], school[dir[Bar], \ldots]]]]$$
$$\mathsf{b}[\ H[\ \ldots\ ],\ LSs[\ \ldots\ ],\ MHs[\ med\_home[\ school[dir[Hart], \ldots]]\ ]\ ]\ ]$$

- $orderBy_{x_1,\ldots,x_k}$ reorders the bindings in the output according to the occurrence of bindings $\mathsf{b}_{in}.x_1, \ldots, \mathsf{b}_{in}.x_k$ in the input.
- *tupleDestroy* returns the element e from the singleton list $\mathsf{bs}[\mathsf{b}[\ v[e]\ ]]$
- $source_{url \to v}$ creates the singleton binding list $\mathsf{bs}[\mathsf{b}[\ v[e]\ ]]$ for the root element e at *url*.

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

Implementation of Operators as Lazy Mediators. Next we describe the implementation of the XMAS algebra operators as lazy mediators. Each operator accepts navigation commands (sent from the client or the operator above) into its output tree and in response to each command c it

(i) generates the required navigation sequence into its input tree(s), i.e., it sends navigation commands to the sources/operators below, and

(ii) combines the results to produce the result of c.

This computation model reminds of pipelined execution in relational databases. However there is a novel challenge: An incoming navigation command $c(p)$ may involve any previously encountered pointer p. Responding to $c(p)$ requires knowledge of the *input associations* $a(p)$ of p. These associations encode sufficient information for continuing the navigation, either down or right, from p.[8]

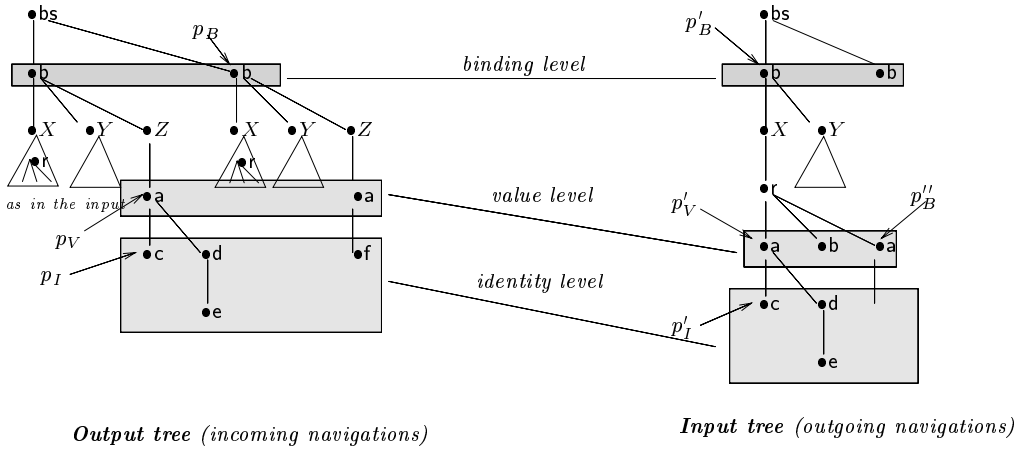

Figure 5: Example navigations for $getDescendants_{\$X,r.a \to \$Z}$

Example 4 Consider the operator $getDescendants_{X,r.a \to Z}$ that operates on the input of Figure 5. Given a node-id p_V at the "value" level of the output, the association $a(p_V)$ contains the token "v" (to indicate that p_V is at the value level) and the corresponding node-id p'_V in the input. It is easy to see that a command $d(p_V)$ will result in a $d(p'_V)$ sent below. A command $r(p_V)$ will result in a $\perp$. Similarly, given a node-id p_I at the "identity" level of the output the association $a(p_I)$ contains the token "id" and the corresponding node p'_I. A $d(p_I)$ results in a $d(p'_I)$ and a $r(p_I)$ results in a $r(p'_I)$.

Finally note that a pointer p_B at the "binding" level requires two associated pointers p'_B and p''_B, as shown in the figure. A command $r(p_B)$ will result in a series of commands

$$p''_B := r(p''_B);\ l := f(p''_B)$$

until l becomes "a" or p''_B becomes $\perp$. In the second case the operator will proceed from p'_B to the next input binding b and will try to find the next a node in the x attribute of b. □

The implementation challenge is that the operator has to know $a(p)$ for each p that may appear in a navigation command and has to retrieve them efficiently. Maintaining association tables for each operator

[8] Interestingly, there is a very close relationship between the associations of p and the lineage of p, where the lineage of p is the input that led to p's generation.

is wasteful because too many pointers will typically have been issued and the mediator cannot eliminate entries of the table without the cooperation of the client. In light of these issues the mediator does not store the node-ids and their associations. Instead the node-ids directly encode the association information $a(p)$ similar to *Skolem-ids*. For example, the node-id p_V in Example 4 is $\langle \mathsf{v}; p'_V \rangle$ and the node-id p_B is $\langle \mathsf{b}; p'_B, p''_B \rangle$.

However, the mediator is not completely stateless; some operators perform much more efficiently by caching parts of their input. For example,

- when the *getDescendants* operator has a recursive regular path expression as a parameter it stores a part of the already visited input. In particular, it keeps around the input nodes that may have descendants that satisfy the path condition,
- the nested-loops join operator stores the parts of the inner argument of the loop. In particular, it stores the "binding" nodes along with the attributes that participate in the join condition.[9]

Appendix A describes the implementation of *createElement* and *groupBy* in more detail.

4 Managing Sources with Different Granularities

The lazy evaluation scheme described in the previous section is driven by the client's navigations into the virtual answer view. Thus, it can avoid unnecessary computations and source accesses. So far, we have assumed "ideal" sources that can be efficiently accessed with the fine grained navigation commands of DOM-VXD, and thus return their results node-at-a-time to the mediator. However, when confronting the real world, this fine granularity is often prohibitively expensive for navigating on the sources:

First, if wrapper/mediator communication is over a network then each navigation command results in a packets being sent over the wire. Similarly high expenses are incurred even if the wrapper and the mediator communicate via interprocess sockets. Second, if the mediator and wrapper components reside in the same address space and the mediator simply calls the wrapper, the runtime overhead may not be high, but the wrapper development cost still is, since the wrapper has to bridge the gap between the fine granularity of the navigation commands and the usually much coarser granularity at which real sources operate. Below we show how to solve this problem using a special buffer component that lets the wrapper *control the granularity* at which it exports data.

[9] We assume a low join selectivity and we do not store the attributes that are needed in the result, assuming that they will be needed relatively infrequently.

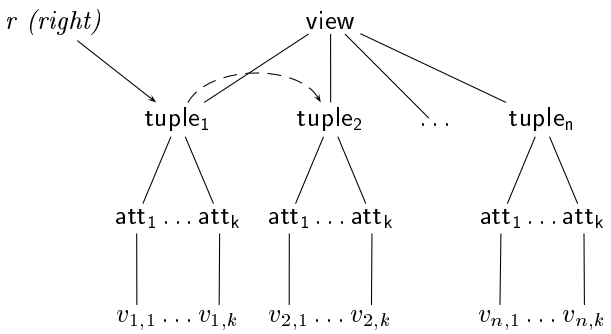

Figure 6: Relational data (source view) in XML tree format

Example 5 (Relational Wrapper) Consider a relational wrapper that has translated a XMAS query into an SQL query. The resulting view on the source has the following format:

$$\mathsf{view}[\mathsf{tuple}^*[\mathsf{att}_1[\ldots], \ldots, \mathsf{att}_k[\ldots]]]$$

i.e., a list of answer tuples whose relational attribute names are $\mathsf{att}_1, \ldots, \mathsf{att}_k$. Assume that the wrapper receives a r (=*right*) command while pointing to some tuple element of the source view (Figure 6). A relational wrapper will translate this into a request to advance the relational cursor and fetch the complete next tuple (since the tuple is the quantum of navigation in relational databases). Subsequent navigations into the attribute level $\mathsf{att}_1, \ldots, \mathsf{att}_k$ can then be answered directly by the wrapper without accessing the database. Thus, the wrapper acts as a *buffer* which mediates between the node-at-a-time navigation granularity of DOM-VXD and the tuple-at-a-time granularity of the source. □

The previous example illustrates that typical sources may require some form of buffering mechanism. Moreover, when the mediator and the wrapper run in different address spaces, communication overhead can be decreased significantly by employing bulk transfers.[10] For example, a relational sources may return chunks of 100 tuples at a time. Similarly, a wrapper for Web (HTML) sources may ship data at a page-at-a-time granularity (for small pages), or start streaming of huge documents by sending complete elements if their size does not exceed a certain limit (say 50K). Finally, a buffer can be used to decouple the client-driven view navigation ("pull from above") and the production of results by the wrapped source ("push from below") based on an asynchronous prefetching strategy. Clearly, additional performance gains can be expected from such an architecture. In the following, we discuss how such extensions can be incorporated into the VXD framework.

[10]... assuming that source data can be beneficially retrieved at a coarser granularity, which is often the case.

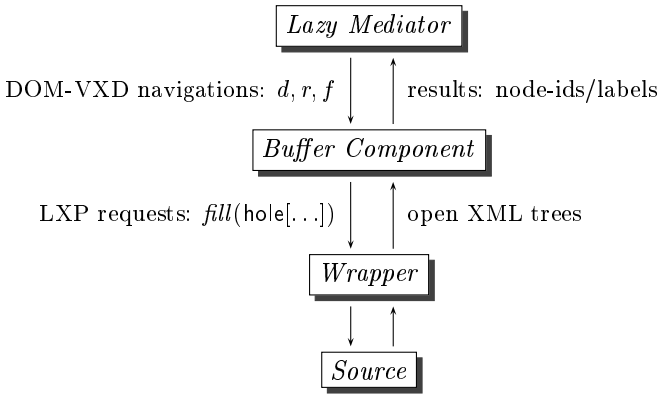

Figure 7: Refined VXD architecture

Refined VXD Architecture: Buffers and XML Fragments (Open Trees). As motivated above, source results usually have to be buffered in order to reconcile the different access granularities of mediators and sources and to improve performance. One way to accomplish this without changing the mediator architecture at all is by incorporating into *each* wrapper some ad-hoc buffering mechanism. While this has the advantage that the buffer implementation can be tailored to the specific source, it also leads to "fat" wrappers with increased development cost. Moreover, similar buffer functionality has to be reinvented for each wrapper in the system.

Therefore, instead of having each wrapper handle its own buffering needs, we introduce a more modular architecture with a separate generic buffer component that conceptually lies between the mediator and the wrapper (Figure 7). The original mediator component remains unchanged and interacts with the buffer using DOM-VXD navigations. If the buffer cannot satisfy a request by the mediator, it issues a request to retrieve the corresponding node from the wrapper. The crux of the buffer component is that it stores *open (XML) trees* which correspond to a *partial* (i.e., incomplete) version of the XML view exported by the wrapper. The trees are open in the sense that they contain "holes" for unexplored parts of the source view. When the mediator sends a navigation command to the buffer component, the latter checks whether the corresponding node is available from the buffer and if so immediately returns the result to the mediator. However, if the incoming navigation "hits a hole" in the tree, then the buffer sends a *fill* request to the wrapper. At this point, the granularity issue is resolved since the wrapper answers the fill request by sending not only the single requested node but possibly the whole XML tree rooted at the node or at least larger parts of it, with further holes in place of the missing pieces.

Definition 3 (Holes, Open Trees) An element of the form $t = \mathsf{hole}[id]$ is called a *hole*; its single child *id* is the unique *identifier* for that hole. No assumption is made about the structure of *id*. We assume

that $\mathsf{hole} \in \mathbf{D}$ is a reserved name. A tree $t \in \mathbf{T}$ containing holes is called *open* (or *partial*), otherwise *closed* (or *complete*). Instead of $\mathsf{hole}[id]$ we may simply write $*_{id}$. □

Holes are used to represent *zero or more unexplored sibling elements* of a tree. More precisely:

Definition 4 (Represented Sublist) Given a tree $t = r[e_1, \ldots, e_n]$, we can replace an arbitrary subsequence $s_{i,k} = [e_{i+1}, \ldots, e_{i+k}]$ $(k \geq 0)$ in t by a hole $*_{i,k}$. In the resulting open tree t', the hole $*_{i,k}$ is said to *represent the sublist* $s_{i,k}$ of t. □

Example 6 (Holes) Consider the complete tree $t = r[a, b, c]$. Possible open trees t' for t are, e.g., $r[*_1]$, $r[a, *_2]$, and $r[*_3, b, c, *_4]$. The holes represent the following unexplored parts: $*_1 = [a, b, c]$, $*_2 = [b, c]$, $*_3 = [a]$, $*_4 = []$. Syntactically, one can substitute a hole by the list of children which it represents (assuming that brackets around inner lists are dropped). □

Observe that since holes represent zero or more elements, the number of items in an open list is generally different from the length of the complete list which it represents.

The Lean XML Fragment Protocol (LXP). LXP is very simple and comprises only two commands *get_root* and *fill*: To initialize LXP, the client (=buffer component) sends the URI for the root of the virtual document, thereby requesting a handle for it:[11]

$$get\_root(URI) \longrightarrow \mathsf{hole}[id]$$

This establishes the connection between the buffer (client) and the wrapper (server). The wrapper answers the request by generating an identifier for the root element. This id and all id's generated as responses to subsequent *fill* requests are maintained by the wrapper. The main command of LXP is

$$fill(\mathsf{hole}[id]) \longrightarrow [\mathbf{T}^*] \ .$$

When the wrapper receives such a *fill* request, it has to (partially) explore the part of the source tree, which is represented by the hole. Different versions of the LXP protocol can be obtained by constraining the way how the wrapper has to reply to the fill request. A possible policy would be to require that the wrapper returns list of the form $[e_1, \ldots, e_n, *_k]$, i.e., on the given level, children have to be explored left-to-right with at most one hole at the end of the list. On the other hand, LXP can be much more liberal, thereby providing interesting alternatives for query evaluation and propagation of results:

[11] In general, sources do not export a single fixed XML view but, depending on the sources capabilities, can accept different XML queries. In this case, the source generates a URI to identify the query result. We assume this step has been done before starting the LXP.

Example 7 (Liberal LXP) Let u be the URI of the complete tree $t = a[b[d, e], c]$. A possible trace is:[12]

$$
\begin{array}{rcll}
\mathit{get\_root}(u) & = & *_0 & \%\ \textit{get a handle for the root} \\
\mathit{fill}(*_0) & = & [a[*_1]] & \%\ \textit{return a hole for a's children} \\
\mathit{fill}(*_1) & = & [b[*_2], *_3] & \%\ \textit{nothing to the left of b; possibly more to the right} \\
\mathit{fill}(*_3) & = & [c] & \%\ \textit{nothing left/right/below of c} \\
\mathit{fill}(*_2) & = & [*_4, d[*_5], *_6] & \%\ \textit{there's one d and maybe more around} \\
\mathit{fill}(*_4) & = & [] & \%\ \textit{dead end} \\
\mathit{fill}(*_5) & = & [] & \%\ \textit{also nothing here} \\
\mathit{fill}(*_6) & = & [e] & \%\ \textit{another leaf}
\end{array}
$$

□

The use of such a liberal protocol has several benefits, most notably, that results can be returned early to the mediator without having to wait before the complete source has been explored (this assumes that the DOM-VXD navigation commands are extended such that they can access nodes not only from left to right). Moreover, when implemented as an asynchronous protocol, the wrapper can prefetch data from the source and fill in previously left open holes at the buffer.

Generic Buffer Algorithms. One advantage of the refined VXD architecture is that a single generic buffer component can be used for different wrappers. The buffer component has to answer incoming navigation commands and, if necessary, issue corresponding LXP requests against the wrapper. Figure 8 depicts the algorithm which handles the down command $d(p)$ for returning a pointer to the first child of p.[13] Note that both the function $d(p)$ and the auxiliary function *chase_first(p)* are recursive. This is because they have to work correctly for the most liberal LXP protocol, in which the wrapper can return holes at arbitrary positions. To ensure correctness and termination of LXP, we only require that (i) the sequence of refinements of the open tree which the buffer maintains can be extended to the complete source tree using fill requests, and that (ii) "progress is made", i.e., a non-empty result list cannot only consist of holes, and there can be no two adjacent holes.

[12] Observe that open trees are somewhat similar to open Prolog lists: e.g., in $[x|Y]$, x is the first element, Y is a variable for the restlist. If say $Y = [b, c|D]$, we obtain $[x, b, c|D]$. With open trees, this corresponds to $[x, *_Y]$ and $*_Y = [b, c, *_D]$. However, the "," in open trees is more general than Prolog's "[*element*|*restlist*]" operator since "," allows partial results on both sides.

[13] We omit a discussion of $f(p)$ (fetch) and $r(p)$, since the former is trivial and the latter is very similar to $d(p)$ – replace: $d(p)/r(p), \mathit{first\_child}/\mathit{right\_neighbor}, \mathit{children}/\mathit{right\_siblings}$.

```
function d(p) {
  if not has_children(p) return ⊥              % cannot go down: done!
  else
    p' := first_child(p);
    if not is_hole(p') return p'               % regular child: done!
    else                                       % p' is a hole
      p'' := chase_first(p);                   % chase down the first child
      if p'' ≠ ⊥ return p''                    % found one: done!
      else
        children(p) := children(p) \ {p'};     % remove empty hole
        return d(p);                           % redo without the empty hole
}
```

```
function chase_first(p) {
  [x_1; ...; x_n] := fill(p);
  update_buffer_with([x_1; ...; x_n]);
  if n = 0
    return ⊥;
  else if not is_hole(x_1)
    return pointer_to(x_1);
  else
    return chase_first(x_1);
}
```

Figure 8: Main buffer algorithm

Wrappers in the Refined VXD Architecture

In Example 5 we discussed a relational wrapper which communicates directly with the mediator, i.e., without an intermediate buffer component. The development cost of such a wrapper is quite high if one wants to avoid severe performance penalties due to the mismatching DOM vs. relational granularities. In contrast, the use of a buffer component provides the same performance benefits while also simplifying wrapper development significantly. The following example sketches the relational wrapper which has been developed for the MIX*m* system.

Relational LXP Wrapper. In order to be able to answer subsequent fill requests, the wrapper has to keep track of the hole id's it has generated. For example, the wrapper could just assign consecutive numbers and store a lookup table which maps the hole id's to positions in the source. Whenever feasible, it is usually better to encode all necessary information into the hole id and thus relieve the wrapper from maintaining the lookup table. For example, the MIX*m* relational wrapper uses hole identifiers of the form[14]

$$\mathsf{hole}[db\_name.table.row\_number] \ .$$

When the wrapper receives a $get\_root(URI)$ command, it connects (through JDBC) with the database specified in the URI and returns a handle to the root of the database, i.e., $\mathsf{hole}[db\_name]$. When receiving a $fill(\mathsf{hole}[id])$ command, the wrapper can initiate the necessary updates to the relational cursor, based on the form of the *id*. In particular, the following structures are returned:

[14] It is transparent to the buffer component whether the hole identifier is sent as a nested XML element db_name[table[...]] or as a '.'-delimited character string.

- at the *database level*, the wrapper returns the relational schema, i.e., the names of the database tables:[15]

$$\mathit{fill}(\mathsf{hole}[\mathit{db\_name}]) \longrightarrow \mathit{db\_name}[\ \mathit{table}_1[\mathsf{hole}[\mathit{db\_name.table}_1]],\ \ldots\ ,\ \mathit{table}_k[\mathsf{hole}[\mathit{db\_name.table}_k]]\ ]$$

- at the *table level*, the wrapper returns the first n tuples *completely* (n is a parameter) and leaves a hole for the remaining tuples (provided the are at least n rows in the table):

$$\mathit{fill}(\mathsf{hole}[\mathit{db\_name.table}_i]) \longrightarrow \mathit{table}_i[\ \mathit{row}_1[a_{1,1}[v_{1,1}],\ldots,a_{1,m}[v_{1,m}]],\ \ldots\ ,\ \mathit{row}_n[a_{n,1}[v_{n,1}],\ldots,a_{n,m}[v_{n,m}]],\ \mathsf{hole}[\mathit{db\_name.table}_i.(n+1)]\ ]$$

- at the *row level*, the wrapper returns the next n tuples (if available):

$$\mathit{fill}(\mathsf{hole}[\mathit{db\_name.table}_i.j]) \longrightarrow \mathit{db\_name.table}_i.j[\ \mathit{row}_{j+0}[\ldots],\ \ldots\ ,\ \mathit{row}_{j+(n-1)}[\ldots],\ \mathsf{hole}[\mathit{db\_name.table}_i.(j+n)]\ ]$$

Observe how the relational *wrapper controls the granularity* at which it returns results to the buffer. In the presented case, n tuples are returned at a time. In particular, the wrapper does not have to deal with navigations at the *attribute level* since it returns complete tuples without any holes in them.

5 Implementation Status and Usage

The Java implementation of the MIX mediator is available from [MIX99] along with an interface to a Python interpreter that allows the user to interactively issue Java calls that correspond to the navigation commands.

A thin *client library* between the mediator and the client application makes the virtual document exported by the mediator indistinguishable from a main memory resident document accessed via DOM by hiding from the client application the complex node-id's exported by the mediator and manipulated with DOM-VXD. In particular, each memory resident XML element has a private field `node_id` that contains the corresponding node-id exported by the mediator. When the client issues a command such as

```
XMLElement r = p.right()
```

the client library issues a call `right(p.node_id)` to the mediator, creates an `XMLElement` object `r` and stores in `r.node_id` the result that it gets from the mediator.

[15] In the real system also column names/types and constraints are returned. We omit these details here.

Finally note that in our current implementation the mediator and the client application run in the same address space, hence avoiding communication overheads. In the future we will allow the client and the mediator to communicate over the network, however this will require exchanging fragments of XML documents to avoid the communication overhead.

6 Conclusions and Future Work

We have presented a novel mediation framework for evaluating queries against virtual mediated XML views. Current mediator systems which employ a virtual approach, compute and return the results of the user query *completely*. Thus, although they do not materialize the integrated view they materialize the result of the user query. Such an approach is problematic or even infeasible in Web mediation scenarios where the user cannot specify his queries precisely enough to obtain a result of managable size.

In our approach this problem is solved by computing answers only partially, as the client navigates into the virtual view. In this paper, we have described how such a demand-driven lazy evaluation can be realized, based on the use of an algebra whose operators are implemented as *lazy mediators*. Lazy mediators serve incoming client navigations by sending corresponding navigations to the sources and returning the processed answers. One advantage of using the XMAS algebra is that each algebra operator can be easily implemented as a lazy mediator. The composition of these operators then yields the lazy mediator for the overall plan.

The fine grained nature of the DOM-VXD navigations is the basis of our demand-driven evaluation model. However communicating only small pieces of data in this way can result in considerable overhead. Moreover, real sources often return their results in a much coarser granularity. We have shown how to reconcile these different granularities using a special buffer component which is based on a simple yet very flexible XML fragment exchange protocol.

Query processing in a DOM-VXD framework differs significantly from the traditional setting and presents new challenges and opportunities for optimization: First, unlike the relational case, data in XML documents is *ordered*. For example, a well-formed XML document is valid only if it conforms to the order prescribed in the associated DTD.

The goal of serving client navigations into the view by issuing only the smallest possible number of source navigations gives rise to the new notion of *navigational complexity* for relating client navigations with the required source navigations. We plan to exploit the measure provided by navigational complexity for optimizing parts of algebraic plans for which a lazy evaluation is *not* beneficial. The resulting strategy will be a combination of lazy demand-driven evaluation and intermediate eager steps.

A prototype of the navigation-driven MIX mediator is available from [MIX99]. We currently develop

client applications that exploit the navigation-driven evaluation of the MIX mediator. Particularly interesting is the DTD-oriented query interface BBQ which blends browsing and querying of XML data, similar in spirit to GARLIC's PESTO interface [CHMW96]. A non navigation-driven earlier version of BBQ and the MIX mediator system was presented at [BGL+99].

References

[AQM+97] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener. The Lorel Query Language for Semistructured Data. *Intl. Journal on Digital Libraries (JODL)*, 1(1):68–88, 1997.

[BGL+99] C. Baru, A. Gupta, B. Ludäscher, R. Marciano, Y. Papakonstantinou, and P. Velikhov. XML-Based Information Mediation with MIX. In *ACM Intl. Conference on Management of Data (SIGMOD)*, Philadelphia, 1999. (exhibition program).

[CCM96] V. Christophides, S. Cluet, and G. Moerkotte. Evaluating Queries with Generalized Path Expressions. In *ACM Intl. Conference on Management of Data (SIGMOD)*, pp. 413–422, 1996.

[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.

[CHN+95] W. F. Cody, L. M. Haas, W. Niblack, M. Arya, M. J. Carey, R. Fagin, M. Flickner, D. Lee, D. Petkovic, P. M. Schwarz, J. T. II, M. T. Roth, J. H. Williams, and E. L. Wimmers. Querying Multimedia Data from Multiple Repositories by Content: the Garlic Project. In *Intl. Conference on Very Large Data Bases (VLDB)*, pp. 17–35, 1995.

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

[FLO] FLORID Homepage. `http://www.informatik.uni-freiburg.de/~dbis/florid/`.

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

[HER] HERMES Homepage. `http://www.cs.umd.edu/projects/hermes/`.

[LPV99] B. Ludäscher, Y. Papakonstantinou, and P. Velikhov. A Framework for Navigation-Driven Lazy Mediators. In *ACM SIGMOD Workshop on the Web and Databases (WebDB)*, Philadelphia, 1999.

[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] The MIX Mediator System (MIXm). `www.db.ucsd.edu/Projects/MIX/MIXm`, 1999.

[PAGM96] Y. Papakonstantinou, S. Abiteboul, and H. Garcia-Molina. Object Fusion in Mediator Systems. In *Intl. Conference on Very Large Data Bases (VLDB)*, pp. 413–424, 1996.

[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.

[XPo] XML Pointer Language (XPointer). W3C working draft, `http://www.w3.org/TR/WD-xptr`.

A Implementation of XMAS Algebra Operators as Lazy Mediators

Command	*Input Node-Id*		*Output (node-id or label)*	
d	$\langle \mathsf{bs}; p_{i_o} \rangle$	$\mapsto$	$\langle \mathsf{b}; d(p_{i_o}) \rangle$	$if\ \ d(p_{i_o}) \neq \perp$
			$\perp$	$otherwise$
r	$\langle \mathsf{b}; p_b \rangle$	$\mapsto$	$\langle \mathsf{b}; r(p_b) \rangle$	$if\ \ r(p_b) \neq \perp$
			$\perp$	$otherwise$
$\mathsf{b}.H$	$\langle \mathsf{b}; p_b \rangle$	$\mapsto$	$\langle \mathsf{id}; p_b.H \rangle$	
$\mathsf{b}.LHs$	$\langle \mathsf{b}; p_b \rangle$	$\mapsto$	$\langle \mathsf{id}; p_b.LHs \rangle$	
$\mathsf{b}.MHs$	$\langle \mathsf{b}; p_b \rangle$	$\mapsto$	$\langle \mathsf{v}; p_b \rangle$	
d	$\langle \mathsf{v}; p_b \rangle$	$\mapsto$	$\langle \mathsf{id}; d(p_b.HLSs) \rangle$	
f	$\langle \mathsf{v}; p_b \rangle$	$\mapsto$	"med_home"	
d	$\langle \mathsf{id}; p_i \rangle$	$\mapsto$	$\langle \mathsf{id}; d(p_i) \rangle$	
r	$\langle \mathsf{id}; p_i \rangle$	$\mapsto$	$\langle \mathsf{id}; r(p_i) \rangle$	
f	$\langle \mathsf{id}; p_i \rangle$	$\mapsto$	$f(p_i)$	

Figure 9: Lazy mediator for the *createElement* operator

- $createElement_{med\_homes,\$HLSs \rightarrow \$MHs}$: Figure 9 depicts the function of the *createElement* of Example 4. For each navigational command and an incoming node-id the lazy mediator will output a new node-id or a label (in case of a fetch command). Note that we have included navigational commands $\mathsf{b}.H$ and $\mathsf{b}.LSs$ that directly navigate into the attribute values H and LSs of the node-id argument. Since the client of the lazy mediator for *createElement* is another lazy mediator, it is wasteful to navigate over the attribute lists of the input mediator. Instead we allow the operators to directly request values of attributes.

  The root node-id of the result of the lazy mediator is $\langle \mathsf{bs}, p_{i_0} \rangle$ where p_{i_0} is the root node-id of the input operator.

  An interesting mapping in the figure is the f command applied to $\langle \mathsf{v}; p_i \rangle$ node-id, i.e. the command fetching the label of the created element (the 7th mapping in the figure). In this case the operator just returns the label "med_homes".

  The other interesting mapping is the d command applied to the newly constructed node $\langle \mathsf{v}; p_b \rangle$ (the 6th mapping). In this case the the operator navigates down into the list of children of the $HLSs$ attribute of the input binding. The resulting node-id that is returned to the client is thus $\langle \mathsf{id}; d(p_b.HLSs) \rangle$.

  Note that some mappings are ommited from the diagram, they correspond to navigations that result in the $\perp$ output.

- $groupBy_{\{\$H\},\$S \rightarrow \$LSs}$: Figure 9 depicts the function of the *groupBy* operator of Example 4. We illustrate the operation of the lazy mediator with the following example:

  Example 8 (Lazy Mediator Implementation for GroupBy) Consider the input of the groupBy oper-

Command	*Input Node-Id*		*Output (node-id or label)*	
d	$\langle \mathsf{bs}; p_{i_o} \rangle$	$\mapsto$	$\langle \mathsf{b}; d(p_{i_o}, \{\}) \rangle$	$if\ \ d(p_{i_o}) \neq \perp$
			$\perp$	$otherwise$
r	$\langle \mathsf{b}; p_g, G_{prev} \rangle$	$\mapsto$	$\langle \mathsf{b}; next_{gb}(p_g), G_{prev} \cup \{p_g\} \rangle$	$if\ \ next_{gb}(p_g) \neq \perp$
			$\perp$	$otherwise$
$\mathsf{b}.H$	$\langle \mathsf{b}; p_b \rangle$	$\mapsto$	$\langle \mathsf{id}; p_b.H \rangle$	
$\mathsf{b}.LSs$	$\langle \mathsf{b}; p_b, p_g \rangle$	$\mapsto$	$\langle LSs; p_b, p_g \rangle$	
d	$\langle LSs; p_b, p_g \rangle$	$\mapsto$	$\langle LS; p_b, p_g \rangle$	
f	$\langle LSs; p_b, p_g \rangle$	$\mapsto$	"$list$"	
d	$\langle LS; p_b \rangle$	$\mapsto$	$\langle \mathsf{id}; d(p_b.LS) \rangle$	
r	$\langle LS; p_b, p_g \rangle$	$\mapsto$	$\langle LS; next(p_b, p_g), p_g \rangle$	$if\ \ next(p_b, p_g) \neq \perp$
			$\perp$	$otherwise$
f	$\langle LS; p_b \rangle$	$\mapsto$	$f(p_b.LS)$	
d	$\langle \mathsf{id}; p_i \rangle$	$\mapsto$	$\langle \mathsf{id}; d(p_i) \rangle$	
r	$\langle \mathsf{id}; p_i \rangle$	$\mapsto$	$\langle \mathsf{id}; r(p_i) \rangle$	
f	$\langle \mathsf{id}; p_i \rangle$	$\mapsto$	$f(p_i)$	

Figure 10: Lazy mediator for the *groupBy* operator

ator:

$$\begin{array}{l} \mathsf{bs}[\ \mathsf{b}[\ H[\ home_1\ ],\ LSs[\ school_1\ ] \\ \quad \mathsf{b}[\ H[\ home_1\ ],\ LSs[\ school_2\ ] \\ \quad \mathsf{b}[\ H[\ home_2\ ],\ LSs[\ school_3\ ] \\ \quad \mathsf{b}[\ H[\ home_1\ ],\ LSs[\ school_4\ ] \\ \quad \mathsf{b}[\ H[\ home_3\ ],\ LSs[\ school_5\ ] \end{array}$$

The output of the operator is the following list of bindings:

$$\begin{array}{l} \mathsf{bs}[\ \mathsf{b}[\ H[\ home_1\ ],\ LSs[\ list[\ school_1, school_2, school_4\ ]\ ] \\ \quad \mathsf{b}[\ H[\ home_2\ ],\ LSs[\ list[\ school_3\ ]\ ] \\ \quad \mathsf{b}[\ H[\ home_3\ ],\ LSs[\ list[\ school_5\ ]\ ] \end{array}$$

One interesting navigation that the *groupBy* implements is the navigation from the binding node b to the right sibling b (the second mapping in the figure). The second navigation we discuss is the navigation from a grouped value to its right sibling (for example navigation from the $school_2$ node to the $school_4$ node, this is the 8th mapping in the figure).

In the first case, the mediator receives an r command from a node-id $\langle \mathsf{b}, p_g, G_{prev} \rangle$ where p_g is the node-id of the first input binding with the same group-by list as the output binding, and G_{prev} is the set of previously encountered group-by lists. The mediator has to find the next output binding with a different group-by list than it has previously encountered. This scan is performed by the function $next_{gb}(p_g)$. Thus, it scans the input bindings until a binding with a new group-by list has been found.

Suppose the navigation is from the first b element of the result to the next binding. The mediator skips the input binding with the same group-by list and outputs $\langle \mathsf{b}; \mathsf{b}_3, \{\ H[\ home_1\ ]\ H[\ home_2\ ]\ \} \rangle$.

In the second case, the mediator receives an r command from a node-id $\langle LS; p_b, p_g \rangle$, where p_b is the input binding that contains the value of variable LS and p_g is the the same as above. The mediator has to find the next input binding with the same group-by list as p_g and from it fetch the value of the LS attribute. Function $next(p_b, p_g)$ in the diagram performs the scan for the next binding with the same group-by list as p_g. The output of the mediator is a new node-id $\langle LS; next(p_b, p_g), p_g \rangle$.

Suppose the client navigates from the $school_2$ node to the $school_4$ node. The navigation is issued from the node-id $\langle LS; \mathsf{b}_2, \mathsf{b}_1 \rangle$, where b_1 and b_2 are node-ids of the first and second input bindings respectively. The mediator scans input binding to the right of b_2, until it finds a binding with the same group-by list as b_1, which is b_4 in the example instance above. It then outputs a new node-id $\langle LS; \mathsf{b}_4, \mathsf{b}_1 \rangle$. □

Since the list of previously seen group-by lists G_{prev} only grows as the client navigates further into the result, for efficiency reasons the mediator stores the list in the buffer and uses a reference to the buffer in the node-ids to recover the correct list G_{prev}. The *groupBy* operator also stores the grouped-by values for each group-by list in G_{prev} and stores the associated lists of values for each group-by list.

For example when the client navigates from $school_2$ to $school_4$ the mediator caches the group-by list $\{ H[home_2] \}$ and the associated list of grouped values $[list[school_3]]$ as it navigates over the third binding of the input. Then, as the client navigates from the first to the second binding of the result, the mediator can retrieve the result of the navigation from the buffer.

These optimizations are incorporated into the $next$ and $next_{gb}$ functions and are not discussed further.