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 them. The evaluation scheme minimizes superfluous source access through the use of *lazy mediators* that translate incoming client navigations on virtual XML views into navigations on lower level mediators or wrapped sources. The proposed demand-driven approach is inevitable for handling up-to-date mediated views of large Web sources or query results. 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 and 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 also 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).

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

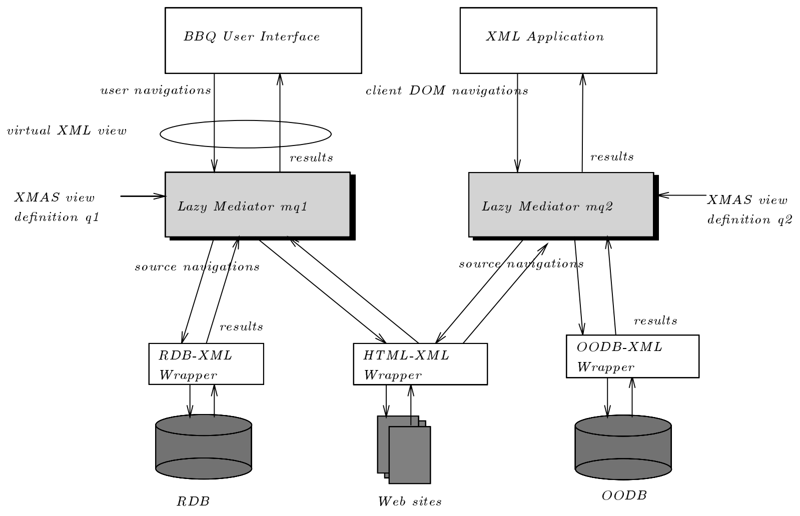

Fig. 1. Virtual XML Document (VXD) mediation architecture

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 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 (Fig. 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 [17] and Lorel [1].

The key idea of the framework is simple and does not complicate the client's code: In response to a query, the client receives a *virtual* answer document. This document is not computed or transfered into the client memory until the user starts navigating it. The virtuality of the document is transparent to the client who accesses it using a subset of the DOM[2] API, i.e., in exactly the same way as main memory resident XML documents.

Query evaluation is navigation-driven in our architecture: The client application first sends a query to the mediator which then composes the query with the view definition and translates the result into an algebraic evaluation plan. After the preprocessing phase, the mediator returns a "handle" to the root element of the virtual XML answer document without even accessing the sources. When the client starts navigating into the virtual answer, the mediator translates these

[1] ***X****ML* ***M****atching* ***A****nd* ***S****tructuring Language* [16]
[2] ***D***ocument ***O***bject ***M***odel [6]

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.

The paper is organized as follows: In Section 2 we introduce lazy mediators and our navigation model. Section 3 elaborates on query evaluation: The XMAS algebra is presented and it is shown how its operators are implemented as lazy mediators. 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.

The complete presentation of XMAS, its algebra, the algorithms, and the software architecture is found in [16]. A preliminary abstract on the architecture appeared in [11]. The implementation is available at [14].

Related Work. Our navigation-driven architecture extends the virtual view mediator approach as used, e.g., in TSIMMIS, YAT, Garlic, and Hermes [15, 4, 5, 10]. The idea of on-demand evaluation is related to pipelined plan execution in relational [8] and object-relational [12] databases where child operators proceed just as much as to be able to satisfy their parent operator's requests. However, in the case of (XML) *trees* the client may proceed from multiple nodes whose descendants or siblings have not been visited yet. In contrast, in relational databases a client may only proceed from the current cursor position. Also the presence of order has an impact on the complexity wrt. navigations and the implementation of lazy mediators.

Note that we use the terms *lazy* and *demand-driven* synonymously, whereas in the context of functional languages, *lazy evaluation* refers to a specific and different on-demand implementation technique for non-strict languages [2, 9].

Our XML query language XMAS borrows from similar languages such as XML-QL, MSL, FLORID, Lorel, and YAT [17, 15, 7, 1, 4]. 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.

2 Navigations in the VXD Framework

We employ XML as the data model [18]. 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: XML documents are viewed as *labeled ordered trees* (from now on referred to simply as *trees*) over a suitable underlying domain $\mathbf{D}$.[3] 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* ("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. Thus, the set $\mathbf{T}$ of labeled ordered trees can be described by the signature $\mathbf{T} = \mathbf{D} \mid \mathbf{D}[\mathbf{T}^*]$.

DOM-VXD Navigation Commands. XML documents (both source and answer documents) are accessible via navigation commands. We present the navigational interface DOM-VXD (DOM for ***V***irtual ***X***ML ***D***ocuments) that is an abstraction of a subset of the DOM API for XML. More precisely, we consider the following set $\mathbf{NC}$ 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 commands can be provided in the style of [19]. E.g., we may include in $\mathbf{NC}$ 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 $c = c_1, \ldots, c_n$. □

3 Query Evaluation Using Lazy Mediators

A *lazy mediator* m_q for a query or view definition[4] q operates as follows: The client browses into the virtual view exported by m_q by successively issuing DOM-VXD navigations on the view document exported by the mediator. For each

[3] $\mathbf{D}$ includes all "string data" like element names and character content.

[4] We often use the terms *query* and *view definition* interchangeably.

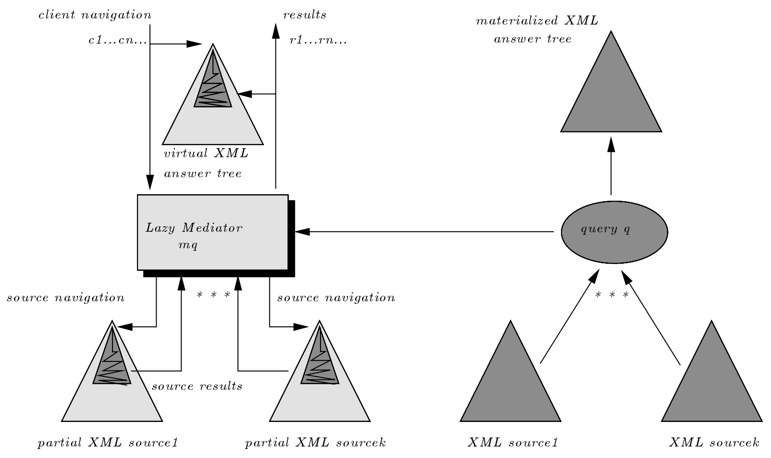

Fig. 2. Navigational interface of a lazy mediator

command c_i that the mediator receives (Fig. 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.

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

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

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 wrt. required navigations by means of rewriting optimizers.

Example 1 (Homes and Schools) Fig. 3 shows a simple XMAS query which involves two sources, `homeSrc` and `schoolsSrc`, and retrieves all homes having

```
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
```

Fig. 3. A XMAS query q

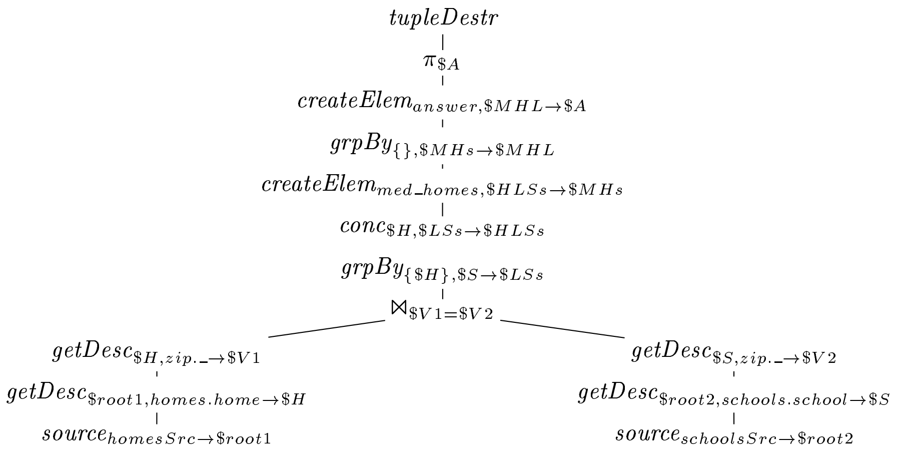
$tupleDestr$
$\pi_{\$A}$
$createElem_{answer,\$MHL\rightarrow\$A}$
$grpBy_{\{\},\$MHs\rightarrow\$MHL}$
$createElem_{med\_homes,\$HLSs\rightarrow\$MHs}$
$conc_{\$H,\$LSs\rightarrow\$HLSs}$
$grpBy_{\{\$H\},\$S\rightarrow\$LSs}$
$\bowtie_{\$V1=\$V2}$
$getDesc_{\$H,zip.\_\rightarrow\$V1}$
$getDesc_{\$root1,homes.home\rightarrow\$H}$
$source_{homesSrc\rightarrow\$root1}$
$getDesc_{\$S,zip.\_\rightarrow\$V2}$
$getDesc_{\$root2,schools.school\rightarrow\$S}$
$source_{schoolsSrc\rightarrow\$root2}$

Fig. 4. Plan (algebra expression) E_q

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 generalized path expressions, as in Lorel [1] and generalized OQL expressions [3]. 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.[5]

The head (`CONSTRUCT` clause) of the query describes how the answer document is constructed based on the variable bindings from the body. E.g., 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, `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 [16]. □

[5] 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.

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[6] to facilitate the description of operators as lazy mediators. For example, the list of variable bindings $[(\$\mathtt{X}/x_1, \$\mathtt{Y}/y_1), (\$\mathtt{X}/x_2, \$\mathtt{Y}/y_2)]$ 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 $(\$\mathtt{X}/v)$ to b_i. $\mathsf{b}_i.X$ denotes the value of X for b_i.

Algebra Operators. The XMAS algebra includes a set of operators conventional in database systems (σ, π, semi-/outer-/antisemi-join, $\times$, etc.) that operate on lists of bindings bs. Additionally, it contains operators that extend the nested relational algebras' nest/unnest operators with generalized path expressions and XML specific features such as access and creation of attribute/value pairs. The complete XMAS algebra is presented in [16]. Due to space limitations we only present operators that participate in the running example.

The semantics of algebra operators is given as a mapping from one or more input trees to the output tree. Let b_{in} and b_{out} denote variable bindings from the input and the output of the operators, respectively. The notation $op_{x_1,\ldots,x_n \to y}$ indicates that op creates new bindings for y, given the bindings for $x_1, \ldots, x_n$.

- $getDesc_{e,re \to ch}$ 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 ".", "|", "*", etc. for path expressions; "_" matches any label (Fig. 4): For each input binding b_{in} and retrieved descendant d, $getDesc$ creates an output binding $\mathsf{b}_{in} + ch[d]$. E.g., $getDesc_{\$H,zip.\_ \to \$V1}$ 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:

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

- $grpBy_{\{v_1,\ldots,v_k\},v \to l}$ groups the bindings $\mathsf{b}_{in}.v$ by the bindings of $\mathsf{b}_{in}.v_1, \ldots, \mathsf{b}_{in}.v_k$ ($v_1, \ldots, v_k$ 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

[6] Variable bindings can refer to the same elements of the input, hence are implemented as *labeled ordered graphs*. This preserves node-ids which are needed for grouping, elimination of duplicates and order preservation.

all values belonging to this group and list is a special label for denoting lists. For example, $grpBy_{\{\$H\},\$S\rightarrow\$LSs}$ applied to the input

bs[b[H[$home$[$addr$[$La\ Jolla$], zip[91220]]], S[$school$[dir[$Smith$], zip[91220]]]]
b[H[$home$[$addr$[$La\ Jolla$], zip[91220]]], S[$school$[dir[Bar], zip[91220]]]]
b[H[$home$[$addr$[$El\ Cajon$], zip[91223]]], S[$school$[dir[$Hart$], zip[91223]]]]]

will yield the output

bs[b[H[$home$[$addr$[$La\ Jolla$], ...], LSs[list[$school$[dir[$Smith$], ...], $school$[...]]]]]
b[H[$home$[$addr$[$El\ Cajon$], ...], LSs[list[$school$[dir[$Hart$], ...]]]]]]

- $conc_{x,y\rightarrow z}$ 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}, *conc* produces $\mathsf{b}_{in}+z[conc]$, where *conc* is:
  - list$[x_1,\dots,x_n,y_1,\dots,y_n]$ if $\mathsf{b}_{in}.x =$ list$[x_1\dots x_n]$ and $\mathsf{b}_{in}.y =$ list$[y_1\dots y_n]$.
  - list$[x_1,\dots,x_n,v_y]$ if $\mathsf{b}_{in}.x =$ list$[x_1\dots x_n]$ and $\mathsf{b}_{in}.y = v_y$.
  - list$[x,y_1,\dots,y_n]$ if $\mathsf{b}_{in}.x = v_x$ and $\mathsf{b}_{in}.y =$ list$[y_1\dots y_n]$.
  - list$[v_x,v_y]$ if $\mathsf{b}_{in}.x = v_x$ and $\mathsf{b}_{in}.y = v_y$.
- $createElem_{label,ch\rightarrow e}$ creates a new element for each input binding. Here *label* is a constant or variable and specifies the name of the new element. Its subtrees are the subtrees of $\mathsf{b}_{in}.ch$. Thus, for each input binding b_{in}, *createElem* outputs a binding $\mathsf{b}_{in} + e[l[c_1,\dots c_n]]$, where l is the value of $\mathsf{b}_{in}.label$ and $c_1,\dots,c_n$ are the subtrees of $\mathsf{b}_{in}.ch$. E.g., $createElem_{med\_homes,\$HLSs\rightarrow\$MHs}$ where $\$HLSs$ results from $conc_{\$H,\$LSs\rightarrow\$HLSs}$ applied to the $\$H$ and $\$LSs$ in the output of the above $grpBy_{\{\$H\},\$S\rightarrow\$LSs}$ yields:

bs[b[H[...], LSs[...], MHs[med_home[$school$[dir[$Smith$], ...], $school$[...]]]]
b[H[...], LSs[...], MHs[med_home[$school$[dir[$Hart$], ...]]]]]

- *tupleDestr* returns the element e from the singleton list bs[b[$v[e]$]]
- $source_{url\rightarrow v}$ creates the singleton binding list bs[b[$v[e]$]] for the root element e at url.

Example 2 (XMAS→Algebra) Fig. 4 shows the algebraic plan for Fig. 3. □

Implementation of Operators as Lazy Mediators. XMAS algebra operators are implemented as lazy mediators. Each operator accepts navigation commands (sent from the "client 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 new 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.

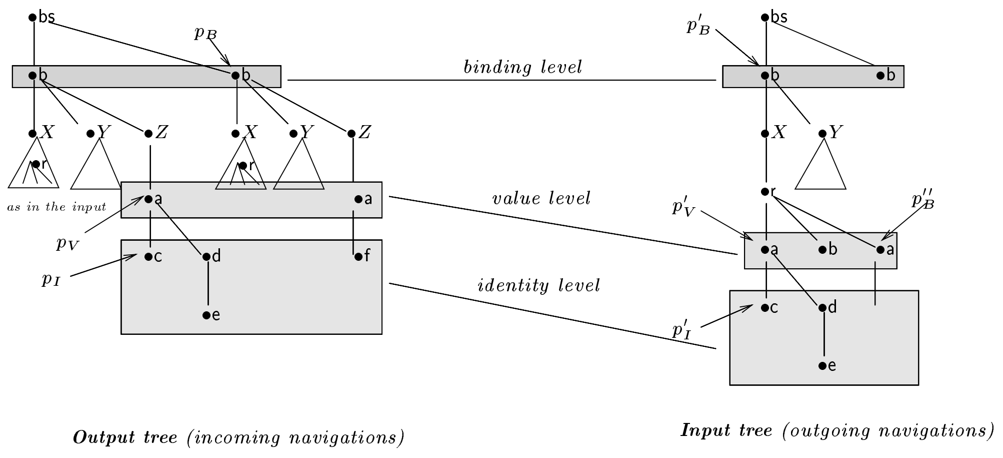

Fig. 5. Example navigations for $getDesc_{\$X,r.a\rightarrow\$Z}$

Example 3 Consider the operator $getDesc_{X,r.a\rightarrow Z}$ that operates on the input of Fig. 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. A $d(p_V)$ will result in a $d(p'_V)$ sent below. A $r(p_V)$ 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 Fig. 5. 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 difficulty 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 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. Thus, the mediator does not store node-ids and associations. Instead node-ids directly encode the association information $a(p)$ similar to *Skolem-ids* and *closures* in logical and functional languages, respectively. In Example 3, the node-id p_V is $\langle \mathsf{v}; p'_V\rangle$, the node-id p_B is $\langle \mathsf{b}; p'_B, p''_B\rangle$.

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

– when the *getDesc* operator has a recursive regular path expression as a parameter, it stores part of the visited input. In particular, it keeps the input nodes that may have descendants which 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.[7]

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.

Example 4 (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:

$$\mathsf{view}[\ \mathsf{tuple}[\mathsf{att}_1[v_{1,1}],\ldots,\mathsf{att}_k[v_{1,k}]],\ \cdots\ ,\mathsf{tuple}[\mathsf{att}_1[v_{n,1}],\ldots,\mathsf{att}_k[v_{n,k}]]\ ]$$

i.e., a list of answer tuples with relational attributes att_j. Let the wrapper receive a r (=*right*) command while pointing to some tuple element of the source view. This will be translated 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 att_j 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. This can also decrease communication overhead significantly by employing bulk transfers. E.g., a relational source may return chunks

[7] 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.

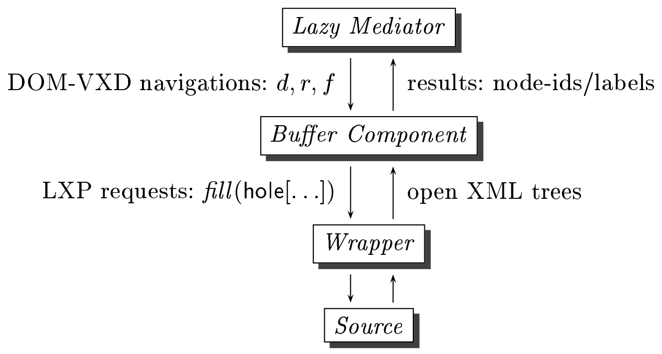

Fig. 6. Refined VXD architecture

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). 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.

Refined VXD Architecture: Buffers and XML Fragments. 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 architecture 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 (Fig. 6). The original mediator remains unchanged and interacts with the buffer using DOM-VXD commands. 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 2 (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 represent *zero or more unexplored sibling elements* of a tree:

Definition 3 (Represented Sublist) Given a tree $t = r[e_1, \ldots, e_n]$, we can replace any 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 5 (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). □

Since holes represent zero or more elements, the length of 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:[8]

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

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

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

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:

[8] 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 6 (Liberal LXP) Let u be the URI of the complete tree $t = a[b[d,e],c]$. A possible trace is:

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

□

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). 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. An 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. Fig. 7 depicts the algorithm which handles the down command $d(p)$ for returning a pointer to the first child of p.[9] 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.

Wrappers in the Refined VXD Architecture. In Example 4 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.

[9] The algorithm for $f(p)$ (fetch) is trivial and $r(p)$ is very similar to $d(p)$: replace $d(p)/r(p)$, $first\_child/right\_neighbor$, $children/right\_siblings$.

```
function $d(p)$ {
  if not $has\_children(p)$ return $\perp$      % can't 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 first child
      if $p'' \neq \perp$ return $p''$          % found one: done!
      else          % remove the empty hole & redo without it:
        $children(p) := children(p) \setminus \{p'\}$;
        return $d(p)$;
}
```

```
function $chase\_first(p)$ {
  $[x_1; \ldots; x_n] := fill(p)$;
  $update\_buffer\_with([x_1; \ldots; x_n])$;
  if $n = 0$
    return $\perp$;
  else if not $is\_hole(x_1)$
    return $pointer\_to(x_1)$;
  else
    return $chase\_first(x_1)$;
}
```

Fig. 7. Main buffer algorithm

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[10]

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

When the wrapper receives a *get_root*(*URI*) command, it connects with the database specified in the URI and returns a handle to the root of the database, i.e., hole[*db_name*]. When receiving a *fill*(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:

- at the *database level*, the wrapper returns the relational schema, i.e., the names of the database tables:[11]

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

- 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):

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

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

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

[10] 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.

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

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.

Implementation Status. A Java implementation of the MIX mediator is available from [14] along with an interface that allows the user to interactively issue Java calls that correspond to the navigation commands. The mediator is complemented by a thin *client library* [16] that sits between the mediator and the client and allows transparent access to the virtual document through (a subset of) DOM. For other components and applications of the MIX system see [13].

References

1. S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener. The Lorel Query Language for Semistructured Data. *Intl. Journal on Digital Libraries*, 1(1):68–88, 1997.
2. P. Buneman, R. E. Frankel, and N. Rishiyur. An Implementation Technique for Database Query Languages. *ACM TODS*, 7(2):164–186, June 1982.
3. V. Christophides, S. Cluet, and G. Moerkotte. Evaluating Queries with Generalized Path Expressions. In *ACM SIGMOD*, pp. 413–422, 1996.
4. S. Cluet, C. Delobel, J. Simeon, and K. Smaga. Your Mediators Need Data Conversion! In *ACM SIGMOD*, pp. 177–188, 1998.
5. W. F. Cody, et.al. Querying Multimedia Data from Multiple Repositories by Content: the Garlic Project. In *VLDB*, pp. 17–35, 1995.
6. Document Object Model (DOM) Level 1 Specification. `www.w3.org/TR/REC-DOM-Level-1/`, 1998.
7. FLORID Homepage. `www.informatik.uni-freiburg.de/~dbis/florid/`.
8. H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database System Implementation*. Prentice Hall, 1999.
9. T. Grust and M. H. Scholl. How to Comprehend Queries Functionally. *Journal of Intelligent Information Systems*, 12(2/3):191–218, Mar. 1999.
10. HERMES Homepage. `www.cs.umd.edu/projects/hermes/`.
11. B. Ludäscher, Y. Papakonstantinou, and P. Velikhov. A Framework for Navigation-Driven Lazy Mediators. In *ACM Workshop on the Web and Databases*, Philadelphia, 1999. `www.acm.org/sigmod/dblp/db/conf/webdb/webdb1999.html`
12. B. Mitschang, H. Pirahesh, P. Pistor, B. G. Lindsay, and N. Südkamp. SQL/XNF – Processing Composite Objects as Abstractions over Relational Data. In *ICDE*, pp. 272–282, Vienna, Austria, 1993.
13. MIX (Mediation of Information using XML). `www.npaci.edu/DICE/MIX/` and `www.db.ucsd.edu/Projects/MIX/`, 1999.
14. MIXm (MIX Mediator System). `www.db.ucsd.edu/Projects/MIX/MIXm`, 1999.
15. Y. Papakonstantinou, S. Abiteboul, and H. Garcia-Molina. Object Fusion in Mediator Systems. In *VLDB*, pp. 413–424, 1996.
16. P. Velikhov, B. Ludäscher, and Y. Papakonstantinou. Navigation-Driven Query Evaluation in the MIX Mediator System. Technical report, UCSD, 1999. `www.db.ucsd.edu/publications/vxd.ps.gz`.
17. XML-QL: A Query Language for XML. `www.w3.org/TR/NOTE-xml-ql`, 1998.
18. Extensible Markup Language (XML) 1.0. `www.w3.org/TR/REC-xml`, 1998.
19. XML Pointer Language (XPointer). `www.w3.org/TR/WD-xptr`.