Mixing Querying and Navigation in MIX

Pratik Mukhopadhyay
CSE Department,
University of California, San Diego
pmukhopa@cs.ucsd.edu

Yannis Papakonstantinou
CSE Department,
University of California, San Diego
yannis@cs.ucsd.edu

Abstract. Web-based information systems provide to their users the ability to interleave querying and browsing during their information discovery efforts. The MIX system provides an API called QDOM (Querible Document Object Model) that supports the interleaved querying and browsing of virtual XML views, specified in an XQuery-like language. QDOM is based on the DOM standard. It allows the client applications to navigate into the view using standard DOM navigation commands. Then the application can use any visited node as the root for a query that creates a new view. The query/navigation processing algorithms of MIX perform decontextualization, i.e., they translate a query that has been issued from within the context of other queries and navigations into efficient queries that are understood by the source outside of the context of previous operations. In addition, MIX provides a navigation-driven query evaluation model, where source data are retrieved only as needed by the subsequent navigations.

1 Introduction

Mediators provide integrated views of information from heterogeneous sources [4, 1, 15]. The interaction paradigm they offer to their users/clients is the conventional database one; the user/client issues queries and the mediator server responds with the full query answer. This interaction paradigm does not capture the continous interleaving of browsing and querying that Web users exhibit during their information discovery efforts. For example, consider an electronic customer of the photo equipment section of an auction site such as eBay. He first issues a query for, say, cameras that cost less than $300. He browses the first few result objects and he realizes that his query is too general. He also realizes that the objects carry the useful attributes "autofocus speed" and "Popular Photo Magazine Rating". So he *refines* the current query result by requiring that the autofocus speed is less than 0.4sec and the rating is at least "medium". He browses into the page for a specific camera, say Nikon123 and then continues browsing into the "matching lens" list only to figure out that there are too many lenses. So he issues a query against the list of lenses for Nikon123 in order to find ones costing less than $200 with diameter greater than 10mm, where the current owner is in Southern California. Then he browses the "lenses" in the answer.

The MIX (Mediation In XML) mediator provides virtual (i.e., non-materialized) integrated views of distributed XML sources and facilitates the interleaved browsing and querying of the views at both the front-end level and the programmatic level. At the front-end level it provides the BBQ [14] GUI, which blends querying and browsing and is reminiscent of IBM's PESTO GUI for "in-place querying" and browsing of object oriented databases [2]. The user of BBQ navigates into XML data. At every time he may issue a query relative to the point that his navigation has reached. (This feature was first introduced in IBM's PESTO and was called "query in place".) For example, the user may navigate into a "camera" node and issue a query for "lens" subobjects with diameter greater than 10mm.

At the programmatic level the MIX mediator provides the *Querible Document Object Model (QDOM)* Application Programmatic Interface (API) that natively supports interleaved querying and navigation of XML data, as needed by client applications such as the BBQ. The navigation commands are a subset of the navigation commands of the standard DOM API. In addition, *QDOM* allows an "in-place query" to be issued from any node in the result of previous queries. The query generates a new "answer" object from which a new series of navigation commands may start.

The efficient support of *QDOM* commands by the MIX mediator required the resolution of major challenges. First, MIX supports the efficient on-demand (lazy) evaluation of XQuery[9] queries. To the best of our knowledge, other XML mediator systems, even those based on the virtual approach, compute and return the full result of the user query. Thus, they often materialize data that the user will not inspect – it is well known that Web users browse just a few results from their query and then move on to the next query. Evaluating the full result unnecessarily overloads the mediator and the sources, reduces the response time as perceived by the client, and uses more memory (for intermediate results) than needed.

Instead the MIX mediator produces the XML result tree as the user navigates into it, hence avoiding unnecessary computations. The framework for doing so is conceptually simple and does not complicate the client's code: The client receives a virtual answer document (QDOM object) in response to his query. This document is not really computed or transferred into the client memory until navigation commands request a part of it. The QDOM framework hides from the client the fact that the result is not really materialized and the client receives a *virtual* answer XML document (DOM object) in response to his query. The non-materialization 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 a main memory resident XML document.

Nevertheless, the non-materialization of the answer document poses an important challenge to evaluating "queries in-place". A query q' issued from a node x of the result of a prior query q has to be evaluated on the non-materialized tree rooted at x. An obvious evaluation strategy would be to retrieve and materialize the tree rooted at x and evaluate q' using standard XML query processing techniques. However, this solution is unacceptable for all the reasons mentioned above: the tree rooted at x may be large and the client is not really interested in it - the client simply wants to be able to navigate in the result of q'. Instead, the mediator handles the evaluation of q' by using *decontextualization*. Decontextualization is a multi-step process that given a query q and a query q' issued from a node x (reached by some navigation n) it produces a query q'' that delivers the same result with q' but without relying on the context created by q and x. Note that decontextualization solves an additional challenge in developing QDOM for mediator views: The typical underlying sources do not support a QDOM-like interaction protocol, which blends querying and navigation. (Some object-oriented databases may be considered an exception to this statement.) Typical sources receive a query and return a result. They often allow the pipelined production of the result but they rarely allow a query to be issued from within the context of previous queries and navigations. The fact that MIX's decontextualization delivers a query q'' that does not depend on the context set by q and x makes the solution applicable to sources with no powerful context mechanisms.

The last step of the challenge is the optimization of the composed query q''. We are concerned with logical optimizations that are performed by a *composition optimization* phase that is responsible for simplifying and optimizing the composed query by removing unecessary intermediate results, pushing selections down, and, in general, performing a set of algebraic rewritings that improve performance in critical ways.

This paper presents how MIX supports QDOM on views of relational databases. Relational databases support a basic form of partial result evaluation: The client issues an SQL query to the

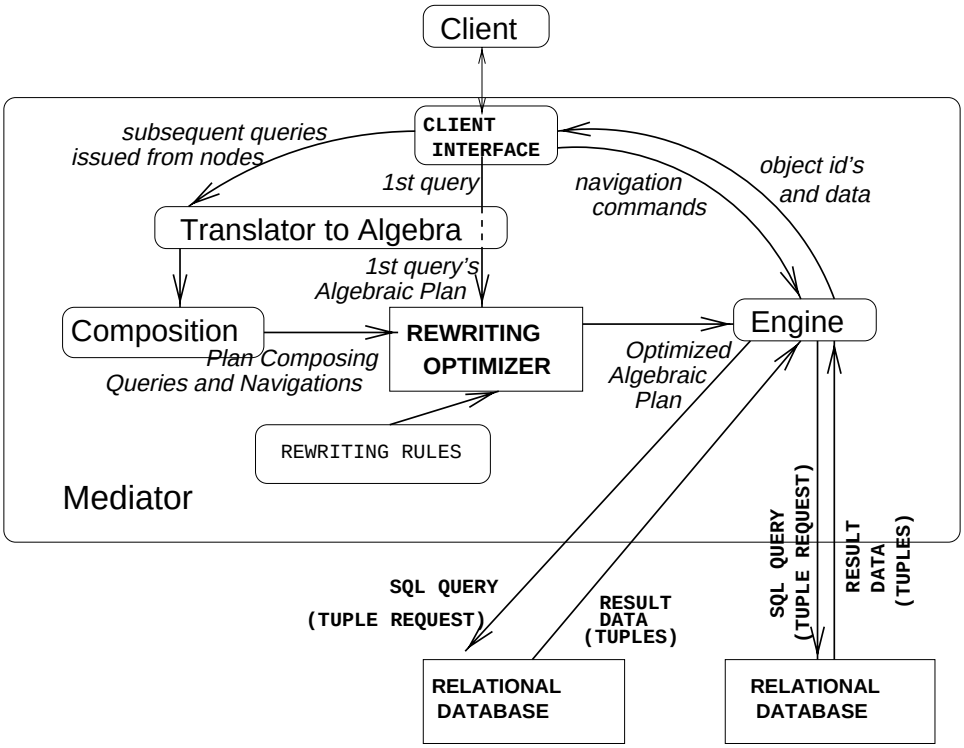

Figure 1: The Architecture of the MIX mediator

database server and receives a cursor, which allows the partial evaluation of the result. However, relational databases do not support any form of issuing queries from within a context created by queries and visited tuples.

In summary, we believe that this paper makes four contributions. It provides the framework and semantics for the interleaving of querying and browsing. It provides the XMAS query algebra and shows how it is used for enabling partial query evaluation – much in the way that iterator models were built on the relational algebra and enabled the pipelined evaluation of SQL queries. It provides the decontextualization algorithm which enables the mediator to support operations not directly supported by the underlying sources. Finally, it provides optimization rules that allow us to push the most efficient queries to the underlying relational databases.

Architecture The XMAS algebra-based architecture of the MIX mediator is depicted in Figure 1. The current system accesses XML files and relational database sources, which are wrapped to offer an XML view of themselves. The paper focuses on the issues pertaining to relational databases and neglects file sources, where the opportunities for efficient QDOM evaluation are limited.

The first XQuery [9] query issued by the client is translated into an XMAS algebra plan, which is subsequently rewritten into an optimized plan. (We are not concerned with cost-based optimization issues in this paper.) The engine receives the optimized algebraic plan and returns to the client the root of the result object. Then the client's navigations into the result object are translated by the engine into additional SQL queries and tuple requests sent to the sources. When the client issues a new XQuery query q' from a node n of the result of the previous query q the composition module combines the algebraic form of q with the node-id of n and the algebraic form of query q' to deliver the composed query. Finally *composition optimization* is performed.

Related Work A number of mediator systems have been developed previously; for example, Garlic [4], XPERANTO [3], TSIMMIS [12], SilkRoute [13], YAT [6], [11]. The SilkRoute project [13, 11] addresses the problem of having a mediator present a unified XML view of multiple relational sources and transform the queries presented to the mediator by the clients into SQL queries which can be sent to the sources. The sources return sorted tuples, which are then marked with appropriate XML tags to create the result XML document presented to the user.

In [11] the authors focus on optimization issues in accessing relational databases. They reach the result that pushing the maximum amount of work to the relational source is not always optimal.

In [3] the authors mention the advantages of leveraging on existing relational query optimization systems for use in query composition and optimization for XML-query languages, and describe a relational system extended with support for querying XML data. Again the focus is on delivering full query results.

In [7, 5] an algebra and rewritings suitable for use in an OODB are presented. In the case of OQL the main emphasis is on un-nesting the queries. The XMAS algebra builds on the OQL algebra adapted for use with semi-structured data in mediator systems. Other researchers have looked at the capabilities problem, and proposed several solutions [12].

Navigation driven evaluation of query results was first proposed in [16] along with a version of an XML query algebra. The XMAS algebra presented in the current paper supports nested plans, has introduced a group-by ability that follows OQL's group-by, and accounts for accessing relational sources.

Overview In Section 2 we introduce the QDOM query and navigation model. In Section 3 the XMAS operators are specified. Section 6 deals with the query composition process. In Section 4 the navigation-driven evaluation is described, and in Section 5 the techniques for supporting mixed browsing and querying are introduced.

2 Framework

Data Model We use a *labeled ordered tree* abstraction of XML where, for simplicity, we have excluded attributes. The set of labeled ordered trees $\mathcal{T}$ is defined as follows:

- The vertex id's are elements of $\mathcal{O}$. Let us denote the elements of $\mathcal{O}$ as $\&1, \&2, \&root, \&XYZ123, \ldots$. The id's may be random surrogates or they may carry semantic meaning. For example, see Figure 2 where the XML equivalent of a relational database is presented. The relational database wrapper exporting the database assigns the tuple keys (eg, `XYZ123`) to be the oid's of the corresponding "tuple" objects – after it precedes them with the `&`.

- Vertices have *labels* in the set of constants $\mathcal{D}$, which is disjoint from $\mathcal{O}$. The labels of leaf nodes will also be called "values".[1]

- The edges $e_1, \ldots, e_n$ starting at a non-leaf node v are ordered. We will refer to the list of trees $t_1, \ldots, t_n$ pointed by $e_1, \ldots, e_n$ as the *value* of v.

In short, the set $\mathcal{T}$ of labeled ordered trees can be described by the signature

$$\mathcal{T} = (vertexId : \mathcal{O}, label : \mathcal{D}, \quad | \quad (vertexId : \mathcal{O}, label : \mathcal{D}, value : [\mathcal{T}^*]) \ .$$

[1] $\mathcal{D}$ includes all "string-like" data, i.e., element names, character content, etc.

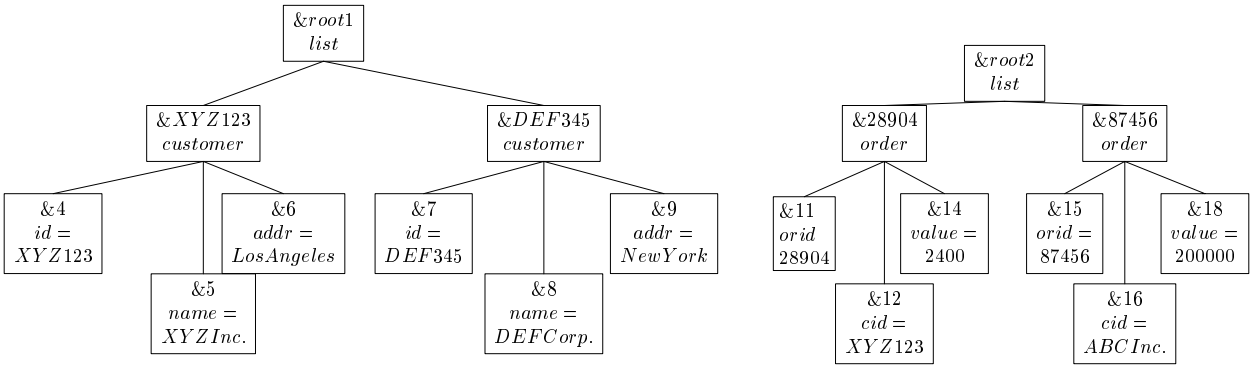

Figure 2: An XML database

```
(Q1) FOR $C IN source(&root1)/customer          % bind $C to the customer tuples
        $O IN document(&root2)/order            % bind $O to order tuples
     WHERE $C/id/data() = $O/cid/data()
     RETURN                                     % Construct the root element containing
        <CustRec>                               % one CustRec element
           $C                                   % per customer $C . CustRec contains
           <OrderInfo>                          % a single customer element $C followed by
              $O                                % OrderInfo elements (one for each order element $O)
           </OrderInfo> {$O}
        </CustRec> {$C}                         % (one CustRec element for each $C)
```

Figure 3: The example query

Query Language We use a subset of the W3C-recommended XQuery XML query language[9], augmented with a group-by operation [8]. Figure 3 shows a simple view that will serve as our running example. The view assumes a source exporting a straightforward XML equivalent of a relational database with relations "*customer*" and "*orders*" (see Figure 2). Note that we use the terms query and view interchangeably.

Figure 4 provides the syntax of the XQuery subset that we consider. Notice that we consider only a subset of path expressions. The complete XQuery path expressions can include predicates, horizontal navigation features, and other features that we exclude. Also notice that the "IN" clauses of the "FOR" statement can only be path expressions.

The QDOM Queries and Navigation Commands The view exported by the mediator can be accessed by a series of navigation commands and queries. The navigation commands presented below abstract a subset of the DOM API for XML. The queries can in-principle be full-fledged XQuery queries. Overall, the client can issue the following commands, where p and p' are vertex id's of the virtual document.

- 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_l (label fetch): $l = f_l(p)$ assigns to l the label of p.

```
Query     ::= ForClause WhereClause ReturnClause
ForClause ::= FOR Variable IN PathExpression
            | ForClause Variable IN PathExpression
WhereClause ::=WHERE PathExpression RelOp PathExpression
              | WhereClause AND PathExpression RelOp PathExpression
ReturnClause ::= RETURN Element
Element  ::= <Label> ElementList </Label>  OptGroupByList
           | Variable
ElementList ::= Element
              | Query
      |ElementList ElementList
OptGroupByList ::= { GroupByList }
                 | (empty)
GroupByList ::= Variable
              | GroupByList , Variable
```

Figure 4: The Syntax of XQuery subset

- f_v (value fetch) : $v = f_v(p)$ assigns to v the value of the leaf p; if p is not a leaf then $f_v(p) = \perp$. In order to simplify the discussion we will assume that the down and right commands are accompanied by fetch commands that deliver the label and value of the reached object.
- q (query): $p' := q(\langle XQuery\ query \rangle, p)$ assigns to p' the root of the answer to the query q. The query q uses a special root, which is lexically denoted as "`root`" in the query. The `root` is assigned the id of p before the query is executed. (Note that p need not have a printable representation.)

EXAMPLE 2.1 We describe next the sequence of programmatic level actions happening while a user browses and queries the integrated view (Q1). The client initially has access only on the root p_0 of the view. By issuing the command

$$p_1 = d(p_0)$$

the client obtains the first `CustRec` node. The command

$$p_2 = r(p_1)$$

fetches the second `CustRec` node while the command

$$p_3 = d(p_1)$$

fetches the `customer` node which is the child of the first `CustRec` node. After some navigation the client may decide that the result is too large and it should be *refined*. So the client issues the command

$$p_4 = q(Q2, p_0)$$

that asks for `CustRec` subobjects of p_0 where the customer's name starts with "A".

(Q2)
```
FOR $P IN document(root)/CustRec
WHERE $P/customer/name < "B"
RETURN $P
```

The command

$$p_5 = d(p_4)$$

navigates into the first `CustRec` and with the sequence of commands the client

$$p_6 = d(p_5); p_7 = r(p_6); p_8 = r(p_7)$$

navigates into the `customer` (p_6) of the first `CustRec` and then into the two first `OrderInfo` objects (p_7 and p_8). At this time the client may decide that there are too many orders for this specfic customer. So he issues the following query. Notice that the query is *contextualized* by the first customer.

$$p_9 = q(Q3, p_5)$$

(Q3)
```
FOR $O IN document(root)/OrderInfo
WHERE $O/order/value < 500
RETURN $O
```

A new navigation and query sequence can start from p_9. ♢

In MIX's implementation the p_i's listed above are really Java objects that are resident on the client's memory – they are *not* the vertex id's exported by the mediator. However, a "thin" *client-side library* associates with each p_i the object id of the corresponding object exported by the mediator. In the rest of the paper we denote by p_i the id of the object exported by the mediator, rather than the actual object used by the client.

3 The Algebra

In this section we describe the operators in the XMAS algebra, which is used to evaluate the queries at the mediator. Unlike the XML Query Algebra and the XQuery Core [10], which are algebras based on functional languages, the XMAS algebra is tuple-oriented and draws on the relational and nested relational algebras. Tuple orientation allows the construction of an iterator model, which extends the iterator model of relational databases and enables navigation-driven partial evaluation. In addition, it enables a better fit with the underlying relational databases, which naturally return tuples.

The connection between the XMAS algebra and XQuery is the following: The `FOR` construct of XQuery creates tuples of variable bindings. The `WHERE` clause is evaluated for each tuple of bindings and qualifies or disqualifies the tuple. Finally, the `RETURN` clause, which contains element creation, list concatenation, and even nested `FOR-WHERE-RETURN` expressions, is evaluated once for every tuple. As is the case with the XML Query Algebra, the XMAS algebra contains individual operators that are responsible for element creation, concatenation, etc, but, unlike the XML Query Algebra where the context of the tuples of bindings is implied by the enclosing `FOR` statements, the input of our operators is explicitly expected to be the list of tuples of bindings that the operators corresponding to the enclosing `FOR` statements have created. (Note that we may have multiple `FOR` statements nested inside each other.)

We should also note the following two features of the algebra, which are important with respect to accessing relational databases and decontextualization respectively.

1. The algebra has a "relational query" operator that is reponsible for turning a part of the plan (which typically corresponds to the `FOR` and `WHERE` parts of the XQuery) into an SQL query

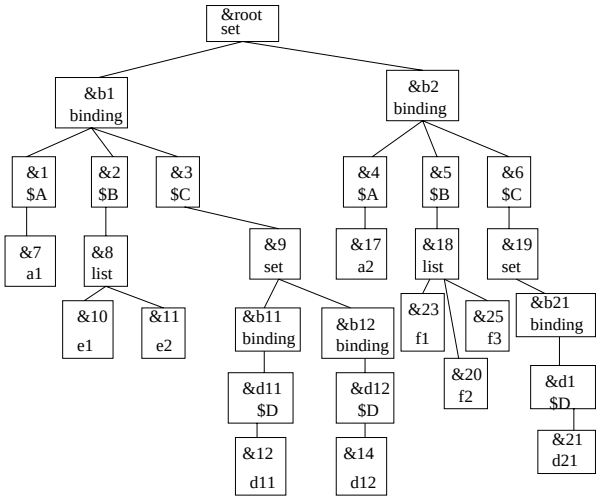

Figure 5: The tree representation of binding lists

that returns tuples of bindings. As we will see in the rest of this paper, the mediator tries to carve out for the relational database the most constrained and efficient query that it can build. This is not trivial when queries are issued on top of views. We will discuss in Section 6 how the mediator achieves optimization of the queries sent to the underlying database.

2. The "create element" operator allows one to construct semantically meaningful id's for the constructed elements. We will see that the constructed id's include all information necessary for tracing the ancestry of an object.

The input and output of most operators is a set of tuples $\{b_i \mid i = 1, \ldots, n\}$, also referred to as *set of binding lists*. Each binding list b_i is a tuple $[\$var_1 = val_1^i, \ldots, \$var_k = val_k^i]$ of variable-value pairs, also referred to as bindings. We say that the variable $\$var_j$ is bound to the value val_j^i in the binding list b_i if the pair $\$var_j = val_j^i$ appears in b_i. All input (resp. output) tuples of an operator have the same list of variables and no variable appears more than once in a tuple. Each value v_j^i can either be a single element, a list of elements or a set of binding lists.

For the purposes of evaluating navigational commands, the output of each operator is also viewed as a tree. The tree representation of a set of binding lists consists of a root node with label list whose children nodes are labelled "binding" and represent the binding lists. Each node representing a binding list has an id which identifies that specific tuple, and has a child node corresponding to every variable for which it has a binding. Each node representing a variable has a single child node representing the value to which the corresponding variable is bound in the binding list. The value nodes may either be a leaf node representing a single element (object), or the root of a subtree representing a list of elements or a set of binding lists. A subtree representing a list of elements l consists of a root node with the label *list* and child nodes corresponding to the values of the elements of the list l. For example, the tree representation of the set of binding lists $B = \{[\$A = a_1, \$B = list[e_1, e_2], \$C = \{[\$D = d_{11}], [\$D = d_{12}]\}], [\$A = a_2, \$B = list[f_1, f_2, f_3], \$C = \{[\$D = d_{21}]\}]\}$ is shown in Fig. 5.

In the rest of the paper $b_i.\$x$ is used to represent the value to which the variable $\$x$ is bound to in the binding list b_i. The notation $b_j = b_i + (\$v = w)$ means that the binding list b_j contains all the bindings from the binding list b_i and the binding $\$v = w$, and $b_k = b_i + b_j$ means that the binding list b_k contains all the bindings from the binding lists b_i and b_j. The notation $l[e_1, \ldots, e_k]$ is used to describe an XML element l with subelements $e_1, \ldots, e_k$.

The operators in the XMAS algebra are as follows:

1. **source :** The source operator $mksrc_{\&srcid,\$X}$ takes a document whose root node $r = \text{list}(\&srcid)[e_1, \ldots, e_n]$ has the id $\&srcid$, and introduces additional nodes to create a tuple structure for each child of the root node. Specifically, it creates a set of binding lists which has a tuple b_i for each child of the node r, and tuple b_i has the value of the attribute $\$X$ set to the value e_i for $i \in 1 \ldots n$. This means the output of the $mksrc_{\&srcid,\$X}$ can be represented as
$$mksrc_{\&srcid,\$X} = \{[\$X = e_1], \ldots, [\$X = e_n]\}$$

2. **getD :** The "get descendants" operator $getD_{\$A.r \to \$X}$ is used to obtain bindings for the variables from the sources. The output of the $getD_{\$A.r,\$X}$ operator on input $I = \{b_i \mid i \in 1 \ldots n\}$ is as follows : Suppose $b_i = [\$A = v_i, \ldots]$. Let $Y_i = \{y_{ij}\}$, where y_{ij} is reachable from the node v_i by a path p such that the labels on this path satisfy the regular expression r. (The path contains the labels of both the start and finish node, unlike path expressions in other query languages, where the label of the first node is not included in the path.) Then the output is the set of binding lists (tuples) defined by
$$getD_{\$A.r \to \$X}(I) = \{b_i + (\$X = y) \mid b_i \in I, y \in Y_i\}$$
Basically this means for every tuple (binding list) in the input, for the node n bound to the variable $\$A$, we find the set of nodes reachable from n by a path satisfying the regular expression r, and these nodes are the bindings for the variable $\$X$. Every pair of bindings for $\$A$ and $\$X$ is inserted into a new tuple (binding list), and bindings for all the other variables in the corresponding input tuple (the tuple from which we took the value of $\$A$) are copied into the output tuple.

3. **select:** The output of the select operator σ_θ is the set of binding lists that satisfy the condition θ.
$$\sigma_\theta(I) = \{b | b \in I, \theta(b) \equiv true\}$$
The conditions θ are of the form
   (a) $\$v$ *op* c, where c is a constant and *op* is one of $=, \neq, >, <, \geq, \leq$. The condition $\$v$ *op* c is true for a given binding list b if $b.\$v$ is bound to a leaf node whose value is x and $(x \ op \ c) \equiv true$.
   (b) $\$v_1$ *op* $\$v_2$ is true for a given binding list b if $b.\$v_1$ is bound to a leaf node whose value is x_1, $b.\$v_2$ is bound to a leaf node whose value is x_2, and x_1 *op* x_2.

4. **projection :** The projection operator $\Pi_{\vec{v}}$ is like the relational project with duplicate elimination.

5. **join:** The $join_\theta$ operator has a single parameter θ which is a boolean predicate. The join operator is like the relational join operator except that it operates on binding lists instead of relational tuples, with variables taking the place of attributes. So the output of the $join_\theta(I_1, I_2)$ operator is
$$join_\theta(I_1, I_2) = \{b_1 + b_2 \mid b_1 \in I_1, b_2 \in I_2, \theta(b_1, b_2) \equiv true\}$$
The condition θ is of the form $\$v_1$ *op* $\$v_2$ where v_1, v_2 are variables and *op* is one of $=, <, \leq, >, \geq$. The condition is true for a given pair of binding lists b_1, b_2 if $b_1.\$v_1$ is bound to a leaf node whose value is x_1, $b_2.\$v_2$ is bound to a leaf node whose value is x_2, and x_1 *op* x_2 is true.

6. **semi-join :** Let I_1 and I_2 be two sets of binding lists with bindings for the sets of variables V_1 and V_2 respectively, where $V_1 \cap V_2 = \phi$. The output of $rightSemijoin_\theta$ on inputs I_1, I_2 is defined as

$$rightSemijoin_\theta(I_1, I_2) = \Pi_{V_1}(join_\theta(I_1, I_2))$$

The output of $leftSemijoin_\theta$ on inputs I_1, I_2 is defined as

$$leftSemijoin_\theta(I_1, I_2) = \Pi_{V_2}(join_\theta(I_1, I_2))$$

7. **createElement:** The $crElt_{l,f(\vec{g}),\$ch\to\$name}$ operator has four parameters: a label l, a skolem function symbol f whose parameters are the list of grouping variables $\vec{g}$, a variable $\$ch$ whose value is the list of children and a variable $\$name$ to which the newly created object is bound. For every incoming tuple b_{in} the operator outputs a tuple $b_{out} = b_{in} + (\$name = l[v_1, \ldots, v_k])$, where the v_i are the elements of the list $b_{in}.\$ch$. The element created has an object id $f(\vec{g})$ which is a skolem of the values of the variables in $\vec{g}$. Formally

$$crElt_{l,\$ch\to\$name}(I) = \{b_{out} | b_{in} \in I, b_{out} = b_{in} + (\$name = l[v_1, \ldots, v_k]), b_{in}.\$ch = \text{list}[v_1, \ldots, v_k]\}$$

8. **cat:** The $cat_{\$x,\$y\to\$z}$ has three parameters: input variables $\$x, \$y$ and the output variable $\$z$. One or both of the input variables may be qualified by a *list* constructor. For every tuple in the input, the $cat_{\$x,\$y\to\$z}$ operator concatenates the lists to which $\$x$ and $\$y$ are bound to in that tuple, and the resulting list is bound to the variable $\$z$ in the corresponding output tuple. If one of the input variables, say $\$x$ is qualified by a *list* constructor, the value to which $\$x$ is bound is first inserted into a list, and this list is concatenated with the list to which the other input variable is bound. So the result is

$$\begin{aligned}
cat_{\$x,\$y\to\$z}(I) &= \{b + (\$z = \text{list}[e_1, \ldots, e_k, f_1, \ldots, f_k]) \mid b \in I, b.\$x = \text{list}[e_1 \ldots e_k], \\
&\quad b.\$y = \text{list}[f_1, \ldots, f_k]\} \\
cat_{list(\$x),\$y,\$z}(I) &= \{b + (\$z = \text{list}[e_1, f_1, \ldots, f_k]) \mid b \in I, b.\$x = e_1, b.\$y = \text{list}[f_1, \ldots, f_k]\} \\
cat_{\$x,list(\$y),\$z}(I) &= \{b + (\$z = \text{list}[e_1, \ldots, e_k, f_1]) \mid b \in I, b.\$x = \text{list}[e_1, \ldots, e_k], b.\$y = f_1\} \\
cat_{list(\$x),list(\$y),\$z}(I) &= \{b + (\$z = \text{list}[e_1, f_1]) \mid b \in I, b.\$x = e_1, b.\$y = f_1\}
\end{aligned}$$

9. **tuple destroy :** The output of the tuple destroy operator $tD_{\$A}$ on the input $I = \{b_i \mid i \in 1 \ldots n\}$ is well defined when every tuple $b_i \in I$ has a binding for $\$A$, and let $b_i.\$A = v_i$. Then the output is the tree T whose root node has the label 'list', where

$$T = tD_{\$A}(I) = \text{list}[v_1, \ldots, v_n]$$

The tuple destroy operator may take an optional second argument which specifies the id of the root node output by this operator. The tuple destroy operator is used as the final operator in every XMAS plan to keep the tuple structure of the intermediate results (the binding lists) internal to the mediator and to export to the users the view of the result they expect, i.e. a view conforming to the *DOM*.

10. **group-by:** The $groupBy_{gl\to\$name}(I)$ operator partitions the input I into sets of tuples $P_{\vec{gl}}(I)$ which agree on the values of the variables in the group-by list gl. The output consists of one tuple with attributes corresponding to the variables in the group-by list gl and $\$name$ for each partition created. The variable $\$name$ is bound to the partition set $P_{\vec{g}}(I)$, while

the variables in gl are bound to corresponding value of the tuples in the partition set $P_{\vec{g}}(I)$. Formally, let $I = \{b_i \mid i \in 1 \ldots n\}$. Let $g_{ij} = b_i.\$v_j$ where $b_i \in I, j \in 1 \ldots k$, and let G be the set of distinct $\vec{g_i}$ where $\vec{g_i} = (g_{i1} \ldots g_{ik})$. Let $\theta_{\vec{g_j}}$ be the condition $\bigwedge_{i=1}^{k}(\$v_i = g_{ji})$. Let $P_{\vec{g_j}}(I) = \{b \mid b \in I, \theta_{\vec{g_j}}(b) \equiv true\}$. The output of $groupBy_{gl \to \$name}$ can then be defined as

$$groupBy_{gl \to \$name}(I) = \{[\$v_1 = g_1, \ldots, \$v_k = b_{in}.g_k, \$name = P_{\vec{g}}(I)] \mid \vec{g} \in G\}$$

11. **apply operator :** The $apply_{p,\$inp \to \$l}$ operator has 3 parameters: a plan p, an input specifier $\$inp$ and an output variable $\$l$. The output of the $apply_{p,\$inp \to \$l}$ is well defined only when the value of $\$inp$ is a set of tuples in every tuple in the input I. Formally, the output is

$$apply_{p,\$inp \to \$l}(I) = \{b + (l = res) \mid b \in I, res = p(b.\$inp)\}$$

    The plan p may operate on input which does not depend on the current binding of the parameters, and in this case the parameter specifying the input is set to *null*.

12. **nestedSrc operator :** The $nestedSrc_{\$x}$ operator has one parameter, and may be used only in nested plans, i.e. plans which appear as a parameter to an apply operator in some other plan. It functions mainly as a placeholder. The output of the $nestedSrc_{\$x}$ operator is the set of binding lists to which the variable $\$x$ is bound to in the current tuple in the outer plan (the plan in which the corresponding *apply* operator appears).

13. **relational query operator:** The relational query operator $rQ_{s,q,m}$ is the source access operator for sources which are relational databases. It takes as parameters

    (a) a server name s,

    (b) a SQL query q

    (c) a map m which which contains the mapping between the variables in the binding lists output by the operator, and the attribute positions in the result of the SQL query.

    The relational query operator is also responsible for creating the nodes corresponding to the tuple objects. The relational query operator may only appear as a leaf of the plan, i.e., it has no input. The relational query operator exports the set of bindings specified by its map parameter.

14. **orderBy operator** The $orderBy_{[\$V_1,\ldots,\$V_m]}$ operator sorts the input tuples according to the bindings of the variables $\$V_1, \ldots, \$V_m$. XMAS does not have currently an order-by that is based on actual values. Instead the order-by orders only according to the id's of the nodes.

The MIX mediator translates the queries it receives into plans in the XMAS algebra. For example, the query shown in Fig. 3 has the plan shown in Fig. 6. The process of translating the query posed by the user into an expression in the XMAS algebra is fairly straightforward. The `FOR, WHERE` and `RETURN` clauses are translated into corresponding subexpressions E_f, E_w and E_r, and these are combined to form the expression for the query. We now consider the translation of each clause separately.

1. The `FOR` clause consists of subclauses of the form `$v IN pathExpr`, where `pathExpr` is either `document("src")/label/path` or `Variable/path`. If the subclause is of the form `document("src")/label/path`, add the expression $getD_{\$z.path,\$v}(mksrc_{src,\$z})$ to the current set, where $\$z$ is a new variable not used in the query. If the subclause is of the form `Variable/path`, add the expression $getD_{\$z.label.path,\$v}(E)$ to the current set, where E is the

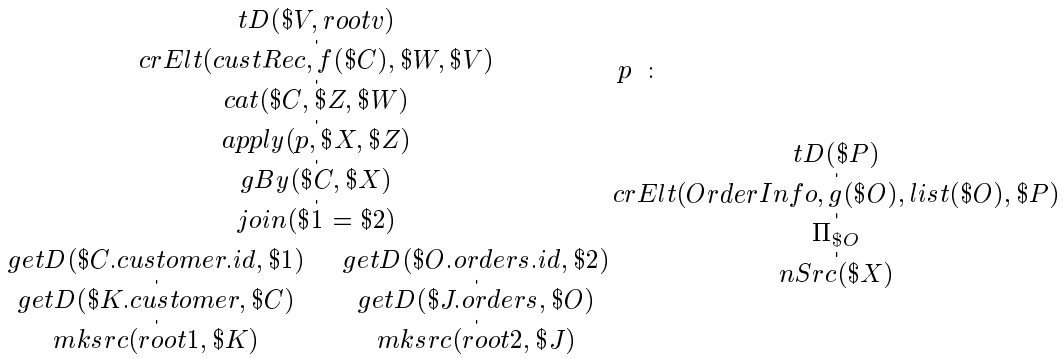

Figure 6: A plan in the for the query in Fig. 3

expression in the current set where an operator of the form $getD_{p,\$z}$ appears. The expression E is then removed from the current set. So after processing the `FOR` clause we have expressions $E_1, \ldots, E_k$ which constitute the current set.

2. The `WHERE` clause is first reduced to subclauses of the form `variable1 = variable2` or `variable = constant`. Initially the current set is the set of expressions corresponding to the `FOR` clause. If the subclause is of the form $v_i = v_j$ and the variables v_i, v_j appear in different expressions E_m, E_n in the current set, remove the expressions E_m, E_n from the current set, and add the expression $join_{v_i=v_j}(E_m, E_n)$ to the current set. If the subclause is of the form $v_i = c$ where c is a constant and v_i appears in expression E_k in the current set, or if the subclause is of the form $v_i = v_j$ and both variables appear in the expression E_k in the current set, replace E_k with $select_{v_i=v_j}(E)$ or $select_{v_i=c}(E)$, as appropriate. If after processing all the subclauses of the `WHERE` clause the current set has more than one expression, combine them into a single expression using the cartesian product.

3. The `RETURN` clause creates a subexpression as follows. Corresponding to each element creation we have a *crElt* operator. The subelements of an element are concatenated together by the *cat* operator. The results of nested subqueries are incorporated by means of the apply operator. Any grouping necessary prior to element creation is done using the gBy operator. The subexpression corresponding to the `WHERE` clause is made the input to the subexpression corresponding to the `RETURN` clause to give the algebraic equivalent of the query.

4 Navigation-Driven Lazy Evaluation

The MIX client receives a virtual answer document (object) in response to its query. The virtual document is not materialized into the client memory until the client starts navigating into it using the r and d navigation commands. Client navigations into the virtual answer are translated by the mediator into various kinds of commands sent to the sources. In the ideal case where the underlying source is an XML source that supports navigation (e.g., a MIX mediator can be such a source to another MIX mediator) client navigations are translated into r and d commands sent to the source. In the case where the underlying source is a relational database, navigations are translated into either queries or moves of the cursors that have been obtained by prior queries. [2]

[2] In the case that the underlying source does not support any form of navigation then the mediator simply obtains the full source result in one step. For example, if the underlying source is an HTML page the mediator obtains it in a single step.

The decomposition of client navigations into commands sent to the sources is achieved by having every XMAS algebra operator in the plan operate as a lazy mediator. When an operator (other than the source access operators) receives a navigation command $n(p)$ from an operator that is above it in the plan, it sends navigation commands to the operators below, and combines the results it receives to produce the result of $n(p)$.

In particular, each operator *op* of the engine is implemented by a Java class supporting the six calls described below:

- The *getRoot*() call normally returns the node id of the `list` element that is at the root of the table that is exported by *op*. The *getRoot*() call always makes *getRoot*() calls to the operators that are the input of *op*.
- Each operator supports the $r(p)$, $d(p)$, $f_l(p)$, and $f_v(p)$ calls for navigation into the result table, its tuples, and the attribute values.
- To facilitate access to the attributes of the bindings, operators support calls of the form $f(p, \$V)$, where p has to be the node id of a `binding` node and $\$V$ is one of the variables of the binding. The call returns the node id of the attribute value.

Node-Id Structure for Navigation Based Query Evaluation In general, each node id p exported by an operator *op* contains state information, which includes information about the nodes $p_s^1, \ldots, p_s^m$ exported by the operator(s) that are the input of *op* and are required to produce the node p. This information enables each operator to evaluate the result of the navigation commands $r(p), d(p), p.\$V$, $f_l(p)$, and $f_v(p)$ by issuing navigation commands involving one or more of $p_s^1, \ldots, p_s^m$.

We denote node id's as $\langle state\_descriptor; state\_info \rangle$. If every node id p exported by an operator *op* has all the information needed for executing $r(p), d(p), p.\$V, f_l(p)$ and $f_v(p)$ we say that *op* is a stateless operator, since the operator itself does not need to keep any information. Most operators have only stateless implementations. For example, consider the stateless $\sigma_{\$A=5}$ operator. When it exports a binding tuple t^o (or an attribute of a tuple) it includes in its state information field the id of the corresponding input tuple t^i. If the client of the σ operator issues a $r(t^o)$ command then the operator will issue an $r(t^i)$ command and receive the next tuple t_1^i. Then it will issue a $f(t_1^i, \$A)$ fetch command to receive the value of $\$A$. If the fetch command returns a 5 then the tuple t_1^i is exported. Otherwise, the next tuple t_2^i is requested and so on.

Other operators, such as the gBy (group-by) may be either stateless or stateful, depending on the input. The stateless gBy assumes that its input is sorted along the group-by variables. The stateful gBy makes no such assumptions, and hence needs buffers to store the input stream. Note that in theory one may build a stateless implementation of every operator. For example, consider the gBy with non-sorted input. One may encode the input stream buffers in the exported id. Then the operator does not need to keep any state. Obviously, such a solution is totally impractical since the size of the id depends on the size of the input data. We follow the practical approach of considering as stateless only the operators that need to keep no state if they are able to include a constant amount of information in the exported id's.

EXAMPLE 4.1 To illustrate the above ideas, let us consider the $gBy(\$C \mapsto \$X)$ operator of the plan of Figure 6. Its goal is to build for each customer a nested table which contains the orders for that customer. Assume that its input is presorted on $\$C$. Table 1 provides the actions and output produced by the presorted gBy operator, which is stateless since it assumes the input is sorted on the values of the variables in its group by list.

Navigation	Actions and Output
$getRoot()$	$r_s = input.getRoot()$ return $\langle root; r_s \rangle$
$d(\langle root; r_s \rangle)$	$b_s = d(r_s)$ $g_1 = b_s.\$G_1; \ldots; g_n = b_s.\$G_n$ return $\langle binding; b_s, [g_1, \ldots, g_n] \rangle$
$r(\langle binding; b_s, [g_1, \ldots, g_n] \rangle)$	repeat $b'_s = r(b_s)$ if $b'_s = \perp$ return $\perp$ $g'_1 = b'_s.\$G_1; \ldots; g'_n = b'_s.\$G_n$ until $g_1 \neq g'_1 \vee \ldots \vee g_n \neq g'_n$ return $\langle binding; b'_s, [g'_1, \ldots, g'_n] \rangle$
$\langle binding; b_s, [g_1, \ldots, g_n] \rangle.\$G_i$	return $\langle identity; g_i \rangle$
$\langle binding; b_s, [g_1, \ldots, g_n] \rangle.\$X$	return $\langle group; b_s, [g_1, \ldots g_n] \rangle$
$d(\langle group; b_s, [g_1, \ldots g_n] \rangle)$	return $\langle in\_binding; [g_1, \ldots g_n] \rangle$
$r(\langle in\_binding; b_s, [g_1, \ldots g_n] \rangle)$	$b'_s = r(b_s)$; if $b'_s = \perp$ return $\perp$ $g'_1 = b'_s.\$G_1; \ldots; g'_n = b'_s.\$G_n$; if $g_1 = g'_1 \wedge \ldots \wedge g_n = g'_n$ return $\langle in\_binding; b'_s, [g'_1, \ldots g'_n] \rangle$ else return $\perp$
$\langle in\_binding; b_s, [g_1, \ldots g_n] \rangle.\$G_i$	return $\langle identity; g_i \rangle$
$\langle in\_binding; b_s, [g_1, \ldots g_n] \rangle.\$V$, where $\$V \notin [G_1, \ldots G_n]$	$v = b_s.\$V$ return $\langle identity; v \rangle$

All undefined navigation commands return $\perp$

Table 1: The implementation of the presorted stateless $gBy_{[\$G_1,\ldots,\$G_n] \mapsto \$P}$

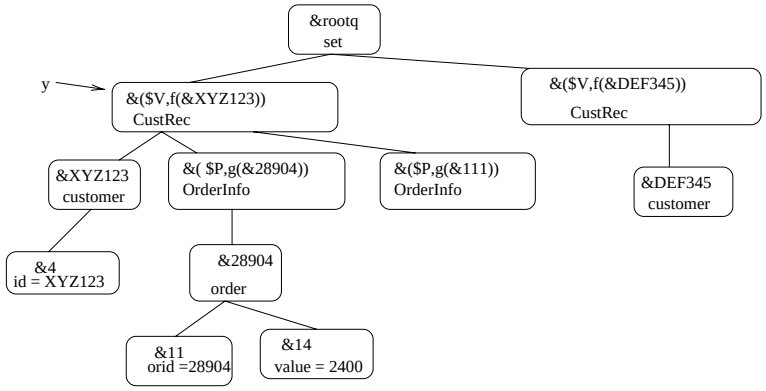

Figure 7: The result of query in Fig. 3

```
FOR $O IN document(&root)/orderInfo/order
WHERE $O/value > 2000
RETURN $O
```

Figure 8:

The first method invoked is the $getRoot()$. This returns the result root node id $\langle root; r_s \rangle$, where the token "$root$" denotes the "state" of the navigation and r_s is the root node of the result exported by the join operator of Figure 6. The node id r_s is all we need to continue the navigation. On receiving a $d(\langle root; r_s \rangle)$ command the operator retrieves the first binding $b_s^1 = d(r_s)$ exported by the semijoin. In addition, note that the value $c_1 = b_s^1.\$C$, which determines the first group, is also retrieved and included in the result node id $\langle binding; b_s^1, [c_1] \rangle$.

The navigation $\langle binding; b_s^1, [c_1] \rangle.\$C$ will return $\langle identity; c_1 \rangle$. The "identity" token indicates that every navigation command on this node is simply propagated to the node below. For example, an $r(\langle identity; c_1 \rangle)$ results into an $r(c_1)$ being sent to the source. If $n = r(c_1)$ then the operator returns $\langle identity; n \rangle$. ♢

5 Queries from nodes reached by Navigations

The MIX system allows the user to issue queries from the nodes reached during navigation. In order to provide this functionality the mediator needs to encode into the id of the node information about its position in the result in a form that is understood at the sources. This is achieved by including in the id of each node information about the values of the group-by attributes associated with the nodes that enclose the given node in the result, and the variable to which this node was bound to in the representation of the intermediate results at the mediator (i.e., the variable to which this node was bound to before the tD operator). If the variables in the group-by list are bound to compound elements, the id needs to encode the values of the fields of this compound element that form a key for elements of this type. To illustrate the above points consider the example query Q shown in Fig. 3 having the plan in Fig. 6. A possible result for the query Q is shown in Fig. 7.

Suppose the user issued the query q_1 shown in Fig. 8 that has the plan p_{q_1} shown in Fig. 9 from the node y (shown in Fig. 7) reached by issuing the navigation command *down* from the root node of the result. Then the mediator could compute by decoding the information available in the id of node y that in the plan p_Q for query Q the node y was the value bound to the variable $\$V$ in the

$$
\begin{array}{c}
tD(\$O) \\
\vdots \\
select(\$P > 2000) \\
\vdots \\
getD(\$O.order.value, \$P) \\
\vdots \\
getD(\$M.orderInfo.order, \$O) \\
\vdots \\
mksrc(\&root, \$M)
\end{array}
$$

Figure 9: Plan for the query in Fig. 8

$$
\begin{array}{c}
tD(\$O) \\
\vdots \\
select(\$P > 2000) \\
\vdots \\
getD(\$O.order.value, \$P) \\
\vdots \\
getD(\$C.orderInfo.order, \$O) \\
\vdots \\
select(\$C = \&XYZ123) \\
\vdots \\
crElt(custRec, f(\$C), \$W, \$V) \\
\vdots \\
cat(\$C, \$Z, \$W) \\
\vdots \\
apply(p, \$X, \$Z) \\
\vdots \\
gBy(\$C, \$X) \\
\vdots \\
join(\$1 = \$2) \\
\begin{array}{cc}
getD(\$C.customer.id, \$1) & getD(\$O.orders.id, \$2) \\
\vdots & \vdots \\
getD(\$K.customer, \$C) & getD(\$J.orders, \$O) \\
\vdots & \vdots \\
mksrc(root1, \$K) & mksrc(root2, \$J)
\end{array}
\end{array}
\qquad
p \ :
\qquad
\begin{array}{c}
tD(\$P) \\
\vdots \\
crElt(OrderInfo, g(\$O), list(\$O), \$P) \\
\vdots \\
\Pi_{\$O} \\
\vdots \\
nSrc(\$X)
\end{array}
$$

Figure 10:

tuple in which the the variable $\$C$ was bound to the node with id $\&XYZ123$. This enables the mediator to create the complete plan for the query q_1 being the plan shown in Fig. 10.

After simplifying this expression using the rewritings in Table 2, the mediator sends to the sources the queries necessary to produce the result of query q_1.

The above example illustrates the technique employed by the mediator to convey the identity of a node reached by navigating into a query result to the sources that do not support navigation. If the client issues a query Q from a node n reached by navigating into the result of a previous query, and values of the group-by attributes (or the values of the fields that form a key, in case the value of the group-by attribute is a complex object created at the mediator) associated with nodes which include the start node n are available, the mediator can replace references to the node n in the plan p_Q for the query Q by a plan constructed by replacing operators of the form $mksrc(\&root, \$Z)$ in plan p_Q by the plan p' constructed as follows. The information in the id of node n is decoded to determine the variable which is to be used in place of $\$Z$ in plan p_Q, and this variable renaming step is carried out in plan p_Q. (Note that the node n is referred to in the query Q by the keyword `root`, and appears only in the *mksrc* operator in p_Q) The top tD operator in the plan p which produced the node n is removed from the plan to give a plan p''. Appropriate selection conditions are added to the plan p'' to fix the values of the variables which have been fixed as a result of the navigation, and this is the plan p'.

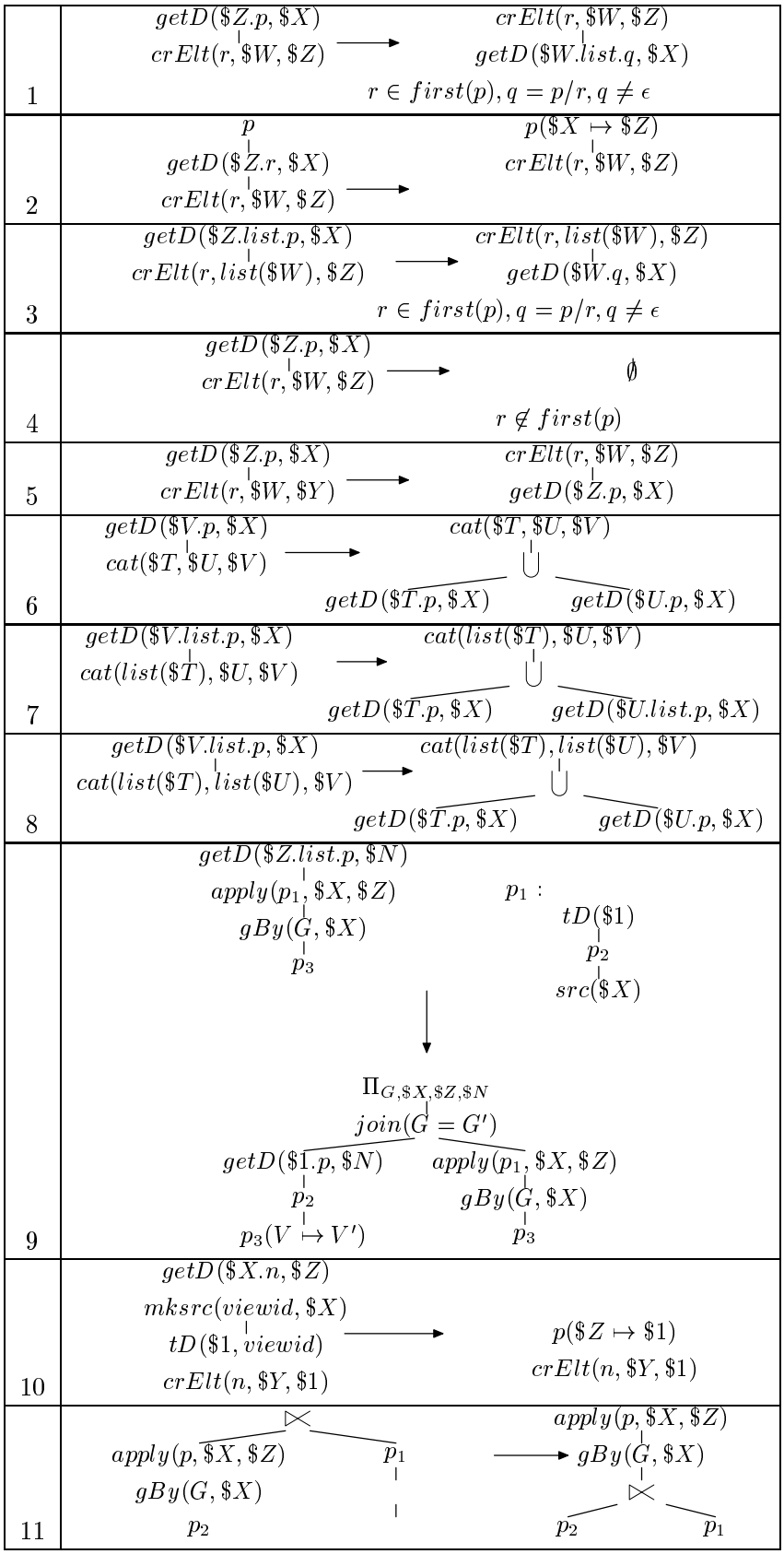

Table 2: Rewriting Rules

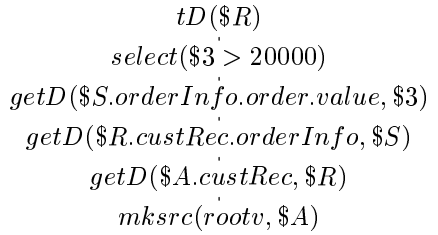

Figure 11: Plan for query in Fig. 12

6 Efficient Composition of Queries

When the client issues a query q_2 from the root of the result of a previous query q_1 the MIX mediator computes and runs the composite plan $q_c \equiv q_1 \circ q_2$. When the query q_c is evaluated on the sources $\mathcal{S}$ the result $q_c(\mathcal{S})$ is equivalent to $q_2(q_1(\mathcal{S}))$, i.e., the result of applying q_2 on the result of q_1. If effiency is not a concern, the algorithm for delivering an algebraic plan p_c for q_c is derived trivially: The mediator simply uses the algebraic plans p_1 and p_2 of q_1 and q_2, and for every source operator in p_2 that refers to the root of q_1, the mediator sets the input of the source operator as the plan p_1, and this gives the resulting plan p_c. For example the query in Fig. 12 has the plan shown in Fig. 11. When the plan in Fig. 11 is composed with the plan in Fig. 6, which is the plan for the view, the resulting plan is shown in Fig. 13.

Unfortunately the simple composition algorithm described above results in plans with multiple inefficiencies which need to be removed.

- The "RETURN" clause of q_1 creates and groups many objects that are consequently matched by the conditions of q_2 but are not used in the result of q_2. Our rewritings remove the construction of unecessary objects from the plan p_1. They also remove path conditions of q_2 that are provably true or provably false.

- The trivial plan p_c is evaluated by pushing to the sources only the conditions of q_1. Our rewriter combines the conditions of q_1 and q_2 and pushes them to the sources the most restrictive queries, which results in the transfer of the minimum amount of data between the mediator and the sources. In order to push selections in the presence of element creation and grouping operations join operations not present in p_1 may need to be introduced into the result plan p_c.

Efficient composition plans are derived in MIX by having a rewriter module optimize the straightforward (and inefficient) composition plans similar to that shown in Fig. 13.

The rewriter accepts as input a set of rules of the form

(search pattern, result pattern)

and a plan on which to apply the rewritings. The changes made by a single rewriting step to the structure of a plan are local, i.e. only the part of the plan which matches the search pattern may have its structure changed. The only change made in the rest of the plan by a rewriting rule application is the possible renaming of variables. The set of rewriting rules is shown in the Table 2.

Next we illustrate the important steps in the application of the rules of Table 2 on the plan shown in Fig. 13, and show how this leads to the pushing of the query selection conditions to the sources and the elimination of unnecessary grouping and element creation operations at the mediator. In the figures showing the algebraic plans, the following convention is used. The part of the plan matching the search pattern is shaded, while the part of the plan which changed as a result of the previous rewriting is enclosed in a box. In order to save space the figures show overlapping

```
FOR $R in document(rootv)/CustRec
 $S in $R/OrderInfo
WHERE $S/order/value > 20000
RETURN
 $R
```

Figure 12: An example query

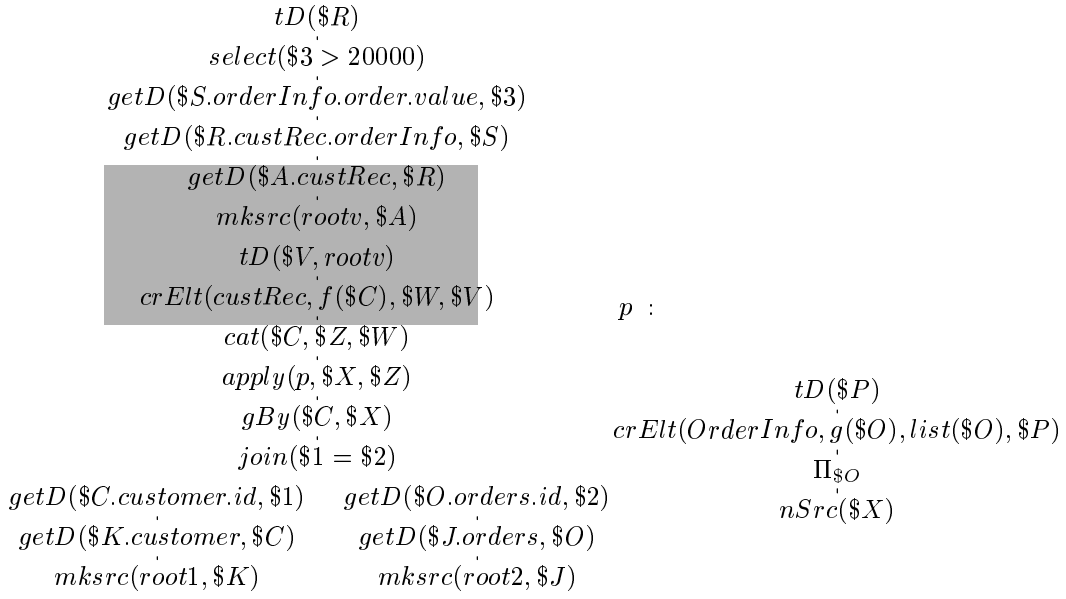

Figure 13: Naive composition of query and view

steps, i.e. figure i shows the matched search pattern for the i^{th} step in a shaded region, and the changes introduced by the $(i-1)^{th}$ step enclosed in a box.

After applying the rewriting rule 11 which eliminates the tD and $mksrc$ operators and establishes the equivalence of the variables $\$V$ and $\$O$, the plan takes the form shown in Fig. 14.

In Fig. 14 we see that the rewriting rule 1 is applicable. We match the path expression in the $getD$ operator against the structure of the view that is specified by the $crElt$ operators in the view. In the plan resulting from the application of the rewriting (shown in Fig. 15), the $getD$ operator instance appears with a simpler path expression. In particular initially the $getD(\$V.custRec.OrderInfo.orders, \$S)$ operator has the path expression $\$V.custRec.OrderInfo.orders$. The $crElt(custRec, \$W, \$C, \$V)$ operator specifies that the child node of $\$V$ has the label $custRec$ and children of this node are the elements of the list which is the child of the $\$W$ node. So the nodes reachable by the path $\$W.list.OrderInfo.orders$ are the same nodes which were reachable by the path $\$V.custRec.OrderInfo.orders$. Thus the $getD$ operator no needs only to check the simpler path condition $\$W.list.lensObj.lens.id$, and this operator can be placed before the $crElt(custRec, \$W, \$C, \$V)$ operator. The result of applying rule 1 to the plan gives the plan shown in Fig. 15 for the query.

Applying the rules for pushing the $getD$ operators downwards leads to one of the following situations :

- The rewriting rule 2 shown in table 2 is applicable, in which case we establish an equivalence between two variables.

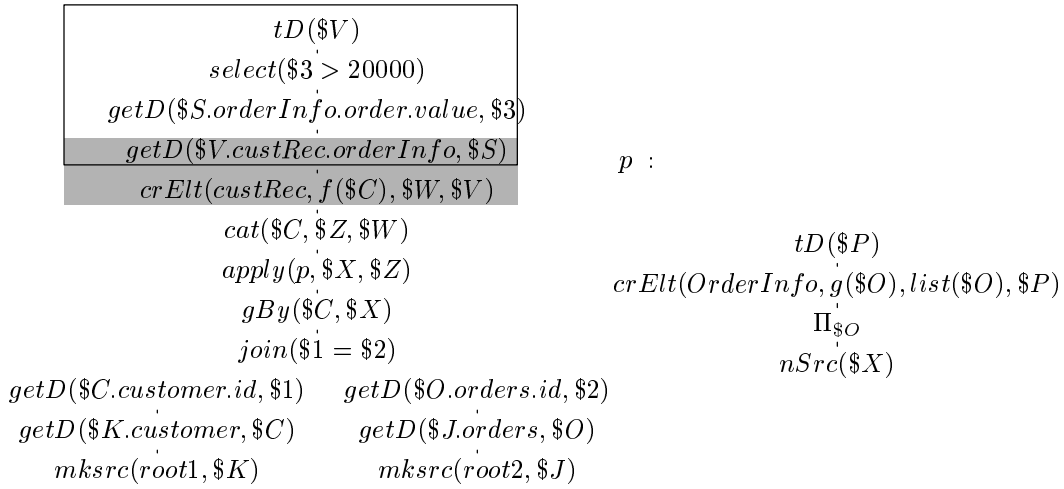

Figure 14:

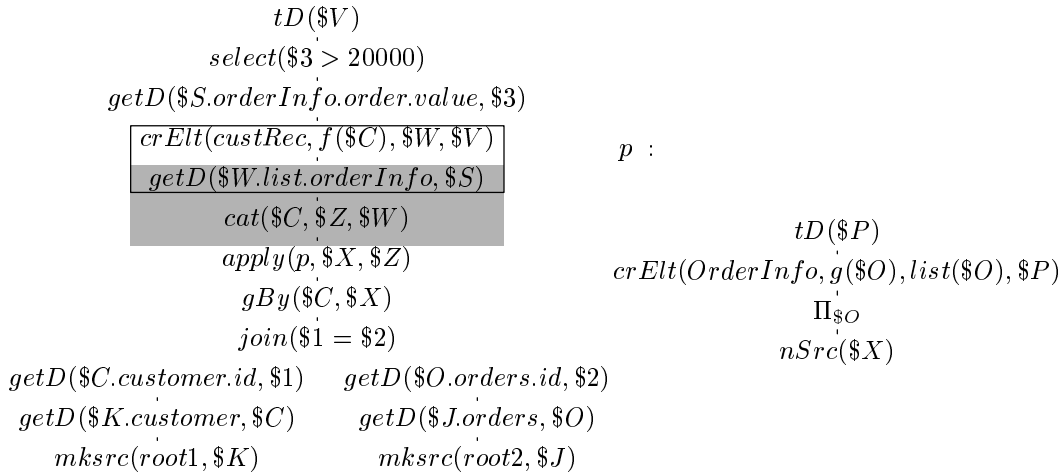

Figure 15:

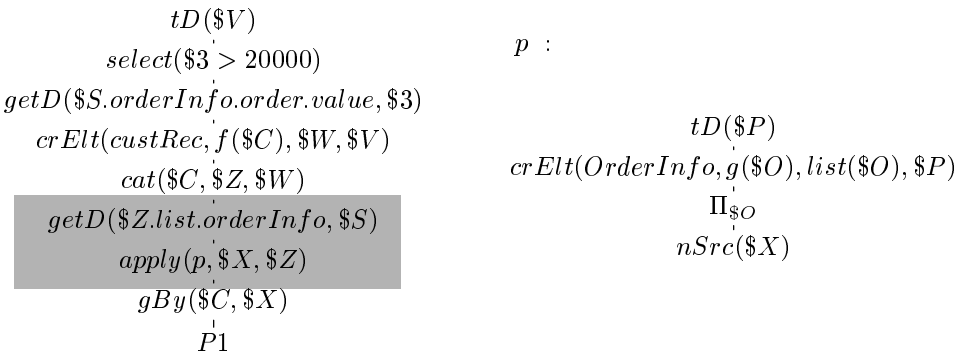

Figure 16:

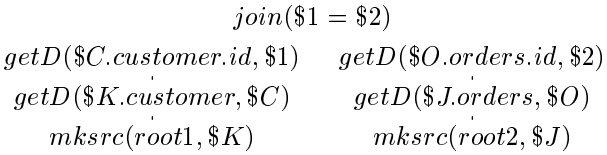

Figure 17: The part of the plan represented as $P1$

- We reach a situation in which we cannot match the path expression in the *getD* operator against the *crElt* operators in the plan. This situation means that the path expression in the *getD* operator expresses a condition on elements (objects) which are fetched from the sources : in this case no further simplification is possible at the mediator in the absense of additional information (e.g. schema information for the source, like DTD or XML-Schema), and these conditions can be pushed to the source. We do not consider the case of using source schema in the rest of the discussion, except note that by adding additional rewrite rules we can include this case easily in our framework.
- The rewriting rule 4 shown in Table 2 is applicable, which tells us that the path condition expressed by the *getD* operator is unsatisfiable.

Applying the rules for pushing the *getD* operators downwards leads to the plan shown in Fig. 16.

The other rewrite rule of interest is the rule 9 in table 2. This rule tries to match the path expressions against elements which are created by the nested plans in the view. In this rule we introduce an additional join operation into the plan. This has the effect of creating an additional copy of the bindings of the variables appearing in the nested plan. This allows us to push the selection conditions on the variables in the nested plan along one branch of the join without losing the any of the bindings for the variables. This is necessary because a variable depending on the nested plan may be appearing in the result of the query. In our example, we see that the variable $\$V$ which depends on the result of the nested plan p appears in the result and the operator $getD(\ldots, \$S)$ can be matched against the structure created by the nested plan p. The application of rule 9 to the plan in Fig. 16 results in the plan in Fig. 18.

Once the path expressions in the *getD* operators in the query have been matched with the corresponding *crElt* operators in the plan of the view, the selection conditions are pushed down as far as possible. This is followed by a live variable analysis, and all operators which create bindings which are not used by the query can simply be removed. The example plan after the selection condition on $3 is pushed down as far as possible is shown in Fig. 19. The live variable analysis of the plan in Fig. 19 shows that the variable $\$P$ is dead : this allows us to convert the join operation

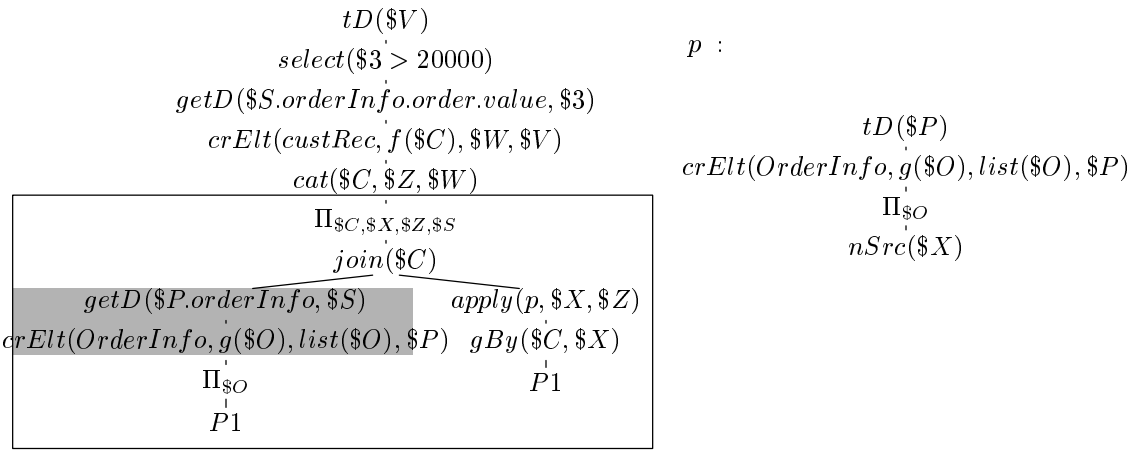

Figure 18:

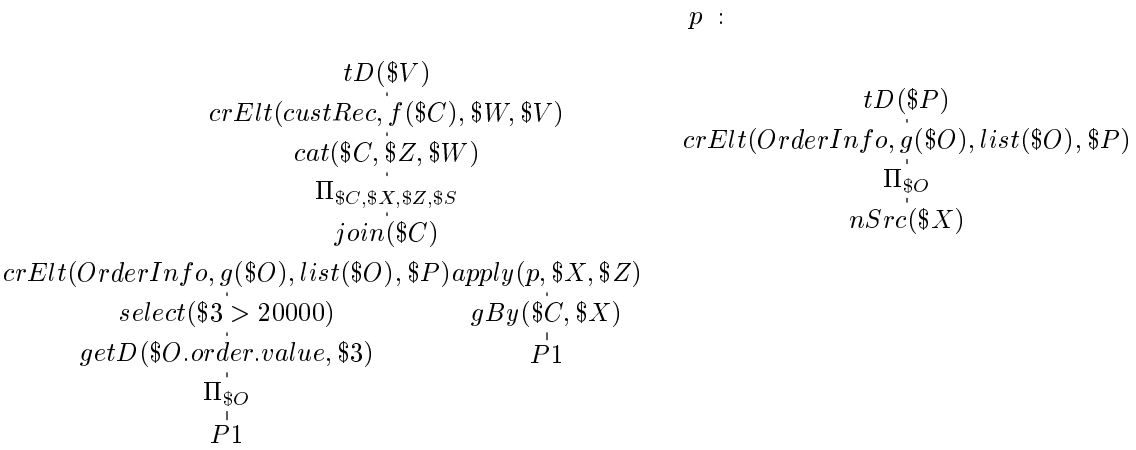

Figure 19:

into a semi-join, and the resulting plan is shown in Fig. 20.

In the plan shown in Fig. 20 the rewriting rule 12 is applicable. This rewriting enables us to push the semi-join operation below the grouping, which is advantageous because the semi-join operation can then be pushed to the sources instead of being evaluated at the mediator. The final simplified plan is shown in Fig. 21. The simplified algebraic plan can then be input to a module which splits the plan into two components : one part consisting of restructuring and grouping operators which is executed at the mediator. The second part, which forms the input to the first consists of the initial *getD*, *select*, and *join* operators and is translated into a query in the appropriate query language for sending to the sources, and is represented at the mediator by a source access operator of the appropriate type. The result of this step is shown in Fig. 22.

7 Conclusions

In this paper we demonstrated the feasibility of a simple algebraic framework which supports pipelined (demand driven) evaluation of queries on tree structured data. The framework allows the development of mediators which offer their clients an API with a full feature set which allows the clients to interleave queries and browsing in their information discovery process, even though

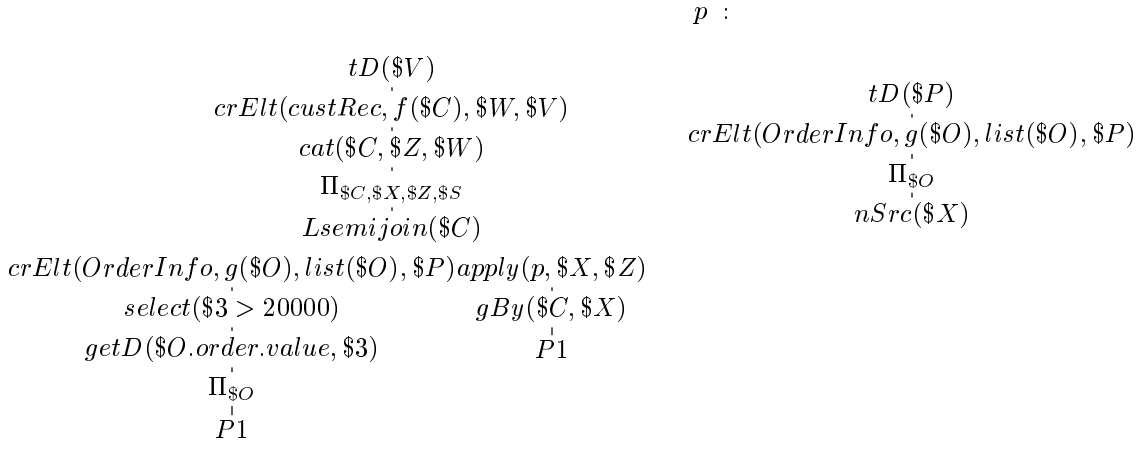

Figure 20:

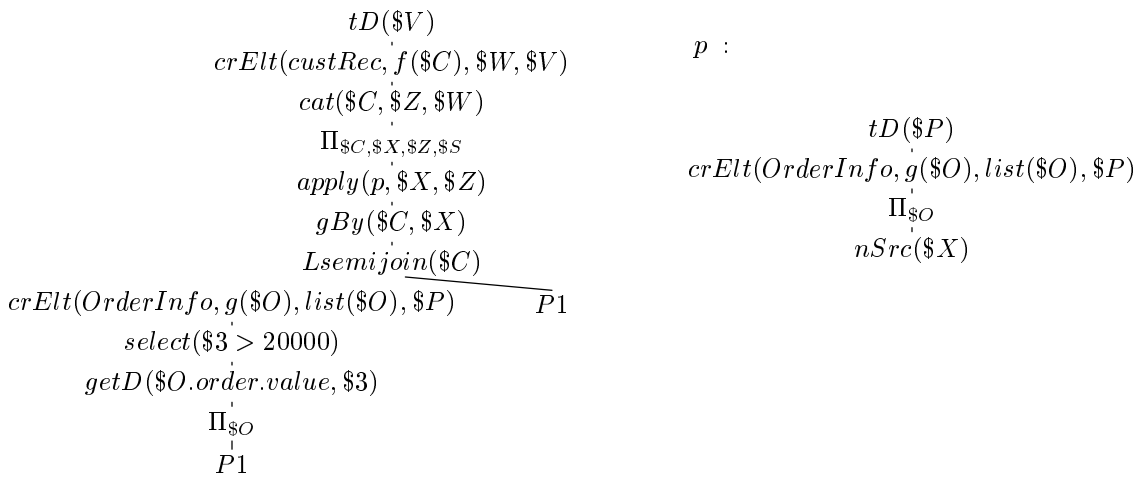

Figure 21:

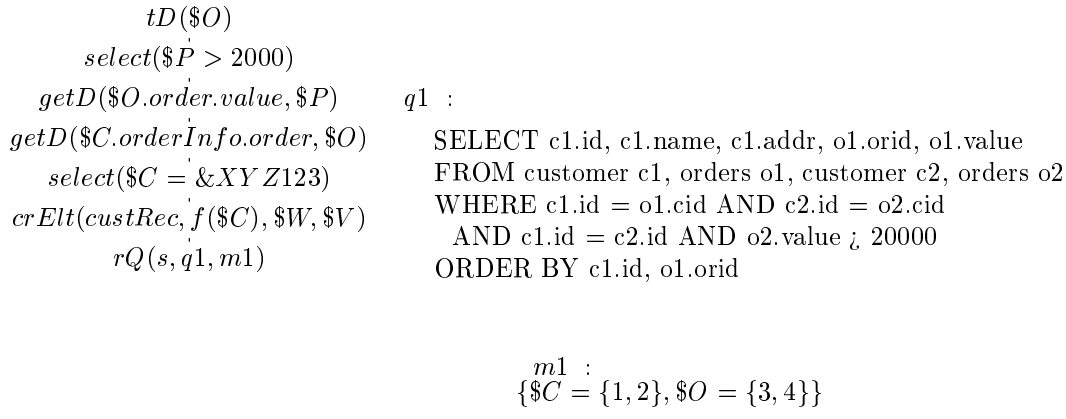

Figure 22:

all the operations may not be directly supported by the sources. The framework builds on previous work done in optimization and query processing for relational databases, and adapts them to the specific challanges posed by XML (tree structured data).

References

[1] S. Adali, S. C. Candan, Y. Papakonstantinou, and V. S. Subrahmanian. Query caching and optimization in distributed mediator systems. In *Proc. SIGMOD*, pages 137–48, 1996.

[2] M. Carey, L. Haas, V. Maganty, and J. Williams. PESTO:An integrated query/browser for object databases. In *Proc. VLDB*, 1996.

[3] Michael Carey, Daniela Florescu, Zachary Ives, et al. XPERANTO: Publishing object-relational data as XML. In *Proc. of the Third International Workshop on the Web and Databases*, 2000.

[4] M.J. Carey et al. Towards heterogeneous multimedia information systems: The Garlic approach. In *Proc. RIDE-DOM Workshop*, pages 124–31, 1995.

[5] Vassilis Christophides, Sophie Cluet, and Guido Moerkotte. Evaluating queries with generalized path expressions. In H. V. Jagadish and Inderpal Singh Mumick, editors, *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996*, pages 413–422. ACM Press, 1996.

[6] S. Cluet, C. Delobel, J. Simeon, and K. Smaga. Your mediators need data conversion! In *Proc. ACM SIGMOD Conf.*, 1998.

[7] Sophie Cluet and Guido Moerkotte. Nested queries in object bases. In Catriel Beeri, Atsushi Ohori, and Dennis Shasha, editors, *Database Programming Languages (DBPL-4), Proceedings of the Fourth International Workshop on Database Programming Languages - Object Models and Languages*, Workshops in Computing, pages 226–242. Springer, 1993.

[8] A. Levy Y. Papakonstantinou D. Suciu V. Vassalos D. Draper, Z. Ives. A proposal for including GroupBy in XQuery. private communication. Submitted as proposal to W3C XML query language committee.

[9] D. Chamberlin et al. XQuery: A query language for XML. W3C working draft, Latest version available at `http://www.w3.org/TR/xquery/`.

[10] P. Fankhauser et al. XQuery 1.0 formal semantics. W3C working draft, Latest version available at `http://www.w3.org/TR/query-semantics/`.

[11] M. Fernandez, A. Morishima, and D. Suciu. Efficient evaluation of xml middle-ware queries. In *Proc. SIGMOD*, 2001.

[12] Chen Li, Ramana Yerneni, Vasilis Vassalos, Hector Garcia-Molina, Yannis Papakonstantinou, Jeffrey D. Ullman, and Murty Valiveti. Capability based mediation in TSIMMIS. In *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*, pages 564–566. ACM Press, 1998.

[13] Mary Fernandez and Wang-Chiew Tan and Dan Suciu. SilkRoute: Trading between Relations and XML. In *WWW9*, May 2000.

[14] K. Munroe and Y. Papakonstantinou. BBQ: A visual interface for browsing and querying xml. In *Visual Database Systems*, 2000.

[15] A. Tomasic, L. Raschid, and P. Valduriez. Scaling heterogeneous databases and the design of DISCO. Technical report, INRIA, 1995.

[16] P. Velikhov, B. Ludascher, and Y. Papakonstantinou. Navigation-driven evaluation of virtual mediated views. In *Proc. EDBT Conf.*, 2000.