Algebraic XML Construction and its Optimization in Natix

Thorsten Fiebig
Software AG
Alsfelderstr. 15-19
64289 Darmstadt
Germany
Thorsten.Fiebig@softwareag.com

Guido Moerkotte
D7, 27
University of Mannheim
68131 Mannheim
Germany
moerkotte@informatik.uni-mannheim.de
Tel.: +49 621 181 2582
Fax.: +49 621 181 2588

February 18, 2002

Abstract

While using an algebra that acts on sets of variable bindings for evaluating XML queries, the problem of constructing XML from these bindings arises. One approach is to define a powerful operator that is able to perform a complex construction of a representation of the XML result document. The drawback is that such an operator in its generality is hard to implement and disables algebraic optimization since it has to be executed last in the plan. Therefore we suggest to construct XML documents by special query execution plans called *construction plans* built from simple, easy to implement and efficient operators.

The paper proposes four simple algebraic operators needed for XML document construction. Further, we introduce an optimizing translation algorithm of construction clauses into algebraic expressions and briefly point out algebraic optimizations enabled by our approach.

Keywords: XML, database management systems, algebraic query evaluation, query optimization

1 Introduction

The success of the eXtensible Markup Language (XML) [5] as a representation format for Web data has motivated a lot of current research work targeting the development of efficient methods for the management of XML data. Quite an active subarea is the development of declarative XML query languages and the efficient evaluation.

There are numerous proposals for XML query languages [3, 7, 8, 11, 21, 24, 25]. The most well-known query languages are XML-QL [11], Lorel [3], YATL [8], XQL [25] and XQuery [7]. Most of the cited candidates structure a query into three parts: a binding (or pattern) clause, a filter clause and a constructor clause. The *binding clauses* provide path or tree patterns that are matched against a given set of XML documents. Every match results in bindings for a given set of (named) variables. Thus the result of a single match can be seen as a tuple that is not necessarily in first normal form. The result of the binding clause is a bag or set of sets of variable bindings, i.e. a relation. Out of these variable bindings some are selected by means of a filter predicate given in the *filter clause*. Finally, the *constructor clause* specifies how the remaining variable bindings are transformed back into XML data. The specification of the constructor clause differs slightly among the different query languages. For example, YATL utilizes a tree structured specification and in XML-QL and XQuery XML templates with embedded subqueries are used.

The development of declarative XML query languages has motivated research concerning the evaluation of XML queries [4, 8, 20, 22, 28]. Lorel queries are evaluated by an extended relational algebra [22] and the evaluation of YATL queries is based on an object algebra [8]. Since both approaches propose the management of variable bindings in table structures, the final step of query evaluation is the construction of XML data from relational data according to the given construction specification. This problem not only arises while evaluating declarative XML query languages against XML data [8, 22] but also while managing XML data in relational or object relational databases [27, 29, 6, 13]. The construction specifications of the query languages provide a variety of constructs like nested queries, skolem functions and explicit grouping operations to achieve high restructuring capabilities. In any case, the grouping of variable bindings becomes the most important issue in constructing the resulting XML data.

As we will see, a result construction may require a combination of several grouping operations which may have to be combined in different ways. We distinguish two different combination types, the *serial* and the *parallel combination* of grouping operations. A *serial combination* of two grouping operations is given if the second grouping operation is performed on the result of the first grouping operation. A *parallel combination* of two grouping operations applies them independently on the same input.

Another important issue of XML result construction is the format of the resulting XML data. Usually the XML result construction produces a text file [1, 2]. If it is consumed by an XML processing application program, this produces a considerable overhead of file creation and parsing. This overhead can be avoided by transferring the XML result directly to the application program via an application programming interface (API) such as the Document Object Model (DOM) [12] or the Simple API for XML (SAX) [23]. In [18] it is shown how a result of an SQL query can be passed

to an XML application via the DOM or the SAX interface without creating an XML document. Our approach goes a step further by pushing the calling of the DOM and the SAX functions inside the query engine.

In this paper we propose the construction of XML data from variable bindings by special query execution plans called *construction plans*. These are built from simple, easy to implement algebraic operators which are designed to support different XML result formats. By introducing construction plans we transform the optimization of the result construction into the problem of optimizing algebraic expressions. Hence, we are able to apply well-known optimization techniques for order optimization [28]. We also give an algorithm that compiles a declarative construction specification into a construction plan.

Summarized, our contributions are as follows:

- Special grouping operation primitives well suitable for implementation in a physical algebra are introduced.
- A result construction approach that supports various XML output formats is developed.
- An optimized translation procedure from construction clauses into the algebra is specified.
- Sample optimizations on the resulting plan are given.

Due to space restrictions we only give some optimizations that can be derived on XML construction without considering the process of binding generation.

1.1 Related Work

There has been a lot of work developing special algebras for the evaluation of XML queries [4, 8, 22] and for their optimization [4, 20]. The YATL algebra [8] introduces a *Bind* operator for the creation of variable bindings and a *Tree* operator [8, 10] for the result construction. The *Tree* operator creates XML data according to a given tree structured construction specification. The difference to our approach is that the *Tree* operator is quite complex which makes it hard to implement. Moreover optimizations are not possible as the *Tree* operator is always executed last in the plan. The description of the evaluation of Lorel queries [22] does not include the result construction.

In contrast to the algebras described in [8, 22], the SAL query algebra [4] performs the whole query evaluation on graph structured XML data. This means variable bindings are not maintained explicitly. So the result of a SAL expression is always XML and a special result construction is not needed.

There are strong relations between our work and the construction of XML data from relational or object relational data [27]. The cited work concentrates on managing XML data within non-XML database systems. The problem of converting relational or object-relational data is done outside the database management system, which results in severe inefficiencies [26].

A recent paper that thoroughly investigates several approaches to construct XML data out of relational data is described in [26]. For the specification of XML data construction nested SQL queries with user-defined functions are suggested. The proposed evaluation works in two steps. The first step is the content creation, that means the creation of a relation that holds the data for the XML document. For the efficient content creation nested queries are de-correlated. To avoid data redundancy caused by multi-valued dependencies, an outer union operator is introduced. The second step is the construction of XML data from the relational content. Opposed to our approach, this is not done by the application of algebraic operators. So there is no way of optimizing the so-called *tagging process* by algebraic rewriting. Another major difference to our approach is that the result of a query is still a relation. Therein, attributes hold XML data as (large) strings. As the authors of [26] point out, this is a severe performance problem. In contrast we suppose that the XML result is created as a "side effect", which provides more flexibility for algebraic optimizations.

1.2 Outline

The remainder of this paper is structured as follows. To illustrate the problem of constructing XML from variable bindings, we start by giving some YATL queries [8]. In section 3 we describe the algebraic result construction by introducing several construction operators and giving some construction plans for the example queries. We continue by describing an algorithm that translates a result construction specification into a construction plan. Based on this algorithm, we investigate techniques for the optimization of construction plans in section 5. Section 6 provides some preliminary performance results. Finally, section 7 concludes the paper.

2 Result Construction in XML Query Languages

In this section we review the construction specification in declarative XML query languages. As a representative XML query language we have chosen YATL, but the results of the paper apply to any other XML query language which allows to construct XML data. Indeed, we use it in Natix, a native XML database system [17], to evaluate NQL (Natix Query Language) queries. Since we only give a very brief introduction to YATL, we refer to [8] for a detailed description.

We discuss result construction by means of three examples. The first example contains a very simple result construction, the second illustrates serial grouping, and the third parallel grouping. These examples will be used throughout the rest of the paper.

To simplify the exposition we have restricted ourselves to a subset of the YATL query language. The restriction includes that we consider neither subqueries in the construction clause nor list-valued variable bindings.

Our examples will be based on the example *Document Type Definition*(DTD) shown in Figure 1. It describes a simplification of the bibliographic entries that can be found in the TOC_OUT file available on the DBLP server [19]. A bibliographic entry contains certain informations like the type of the publication, authors, title and so on. We con-

```
<!ELEMENT bib
          (conference|journal)*>
<!ELEMENT conference
          (title,year,article+)>
<!ELEMENT journal
          (title,volume,no?,article+)
<!ELEMENT article (title,author+)>
<!ELEMENT author (last,first)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT last (#PCDATA)>
<!ELEMENT first (#PCDATA)>
```

Figure 1: DTD that specifies the structure of bibliographic entries on the DBLP server.

sider two publication types: conference proceedings and journals. Hence, the content of the `<bib>` element is built by `<conference>` and `<journal>` elements. A `<conference>` element contains the conference name, the conference year and one or more `<article>` elements. Each `<article>` element consists of the article title and the article authors. A `<journal>` element consists of the journal name, the publication year, the volume and one or more `<article>` elements. Additionally it contains an optional journal number. For the example queries we assume that the XML data is located at the URL `a.b.c/ley.xml`.

Our first example query is:

Query 1

```
MAKE
 result
   [ *conference
     [ title[$t],
       year[$y] ] ]
MATCH "a.b.c/ley.xml"  WITH
 bib
   [ *conference
     [ year[$y],
       title[$t] ] ]
WHERE $y > 1991
```

A YATL query consists of a MAKE, a MATCH and a WHERE clause. The MATCH clause creates variable bindings by performing pattern matching. Hence, it contains a textual representation of a tree pattern with labeled nodes and edges. The node labels are build by XML elements (denoted by the element label), XML attributes (denoted by '`@`') and variables (denoted by a '`$`'). If a subpattern should occur several times the incident edge is labeled with a '`*`'.

The tree pattern of the MATCH clause describes a `<bib>` element that contains several `<conference>` elements. The content of the `<conference>` elements is built by

<title> and a <year> element. By matching the pattern with the XML data stored in a.b.c/ley.xml, the MATCH clause creates bindings for the variables $t and $y for each <conference> element. $t is bound to the conference title and $y is bound to the conference year.

Each matching results in a tuple [$y,$t]. These tuples are filtered by the WHERE clause according to the given predicate that selects those tuples with a $y value greater then 1991.

The MAKE clause of a YATL query converts the tuples into XML data. In our example query it creates a <result> element that contains a <conference> element for each filtered tuple.

In contrast to the MAKE clause of Query 1 that describes only simple structuring, the construction specification of Query 2 imposes a complex nesting of the variable bindings. Therefore it exhibits a *serial combination* of two grouping operations. This means the variable bindings are grouped by the first operation and each resulting partition is grouped by the second operation.

Query 2

```
MAKE
 result
   [ *($j)journal
     [ name[$j],
       *($t)article
       [ title[$t],
         *author[$a] ] ] ]
MATCH "a.b.c/ley.xml"  WITH
 bib
   [ *journal
     [ title[$j],
       article
         [ title[$t],
           author[$a] ] ] ]
```

The MATCH clause of the query scans for <journal> elements and creates a tuple [$j,$t,$a] for each author of a journal article, where $j is bound to the journal name, $t is bound to the article title and $a to the author.

The MAKE clause creates a <result> element that contains a <journal> element for each different journal name. Therefore the variable bindings have to be grouped according to the $j attribute. So the pattern of the MAKE clause contains an edge incident from the <result> node that is labeled with the skolem function *($j). The content of the <journal> consists of the journal name and an <article> element for each different binding of the variable $t. Therefore the pattern contains an edge incident from the <journal> node labeled with the skolem function *($t) that describes a grouping of the variable bindings according to the article title.

The construction specification of Query 3 exhibits a *parallel combination* of two grouping operations. That means the variable bindings are independently grouped ac-

cording to the first operation and the second operation. Hence, the construction specification contains two edges labeled with a skolem function (`*($a)`,`*($t)`), where the edges belong to different paths leading from the root to a leaf.

Query 3

```
MAKE
 result
   [ *($c)conference
     [ title[$c],
       authorlist
         [ *($a)author[$a] ],
       articlelist
         [ *($t)article[$t] ] ] ]
MATCH "a.b.c/bib.xml"  WITH
 bib
   [ *conference
     [ title[$c],
       article
         [ title[$t],
           author[$a] ] ] ]
```

The MATCH clause of the query creates a tuple `[$c,$t,$a]` for each author, where `$c` is bound to the conference title, `$t` to the article title and `$a` to the author.

The MAKE clause creates a `<result>` element that contains a `<conference>` element for each conference. It contains the conference title, an `<authorlist>` and an `<articlelist>` element. The `<authorlist>` element contains an `<author>` element for each author who published an article on the regarded conference. The content of the `<articlelist>` is a list of `<article>` elements that contains the title of the represented article.

3 Construction Operators

For the algebraic result construction we introduce four new construction operators. Most of their functionality results from an interaction between the operators that differs from conventional query execution plans [14, 15]. Here, assuming the iterator principle for operator implementation, the operators interact via *Open*, *GetNext* and *Close* operations. The iterator principle is depicted in figure 2(a). The dashed arrows represent the operation calls to an operator issued by its consumer and the operation calls issued by the operator to its producers. The solid arrows represent the operator's input and output tuple streams.

In this section we first give the main design goals before we introduce the operators by giving an informal description. Further, we give several example construction plans to illustrate the application of our construction operators.

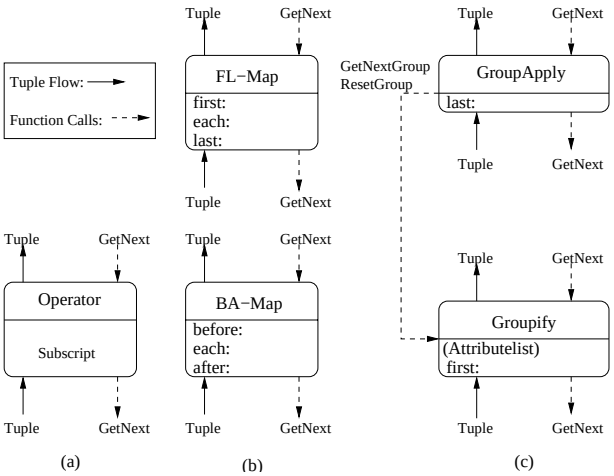

Figure 2: Interfaces of the construction operators.

3.1 Design Goals

The output format of an XML query varies. For example if the result of a query is stored in a document the query should create a textual representation. In contrast to that, a DOM tree or sequence of SAX callback function calls might be appropriate, if a query result is directly passed to an application program.

Therefore, one main design goal of the construction operators was a great flexibility concerning the output format. To meet this requirement, our operators construct the result applying construction functions. These functions can either be simple print statements or XML API function calls. By exchanging the construction functions in a construction plan the output format can be changed without altering the plan structure. The results of the construction functions is not added to the output tuples of construction operators. Hence, applying the construction functions results in an XML construction by "side effect".

The advantages are twofold. First, the constructed XML data does not interfere with the algebraic optimization. In [26] it is shown that adding the XML data as a string attribute to the output tuples of the construction operators has a considerable performance impact. Hence a query processor has find a query plan that minimizes the costs of such string attributes. A good heuristic is to perform the XML data creation last in plan, but this seems to be a severe restriction on query optimization.

Second, the greater flexibility concerning the output format makes it possible to choose an optimal way to pass the XML data result to the application. We will show how the query evaluation time can be reduced by passing the XML result via SAX callback functions.

The second main design goal was to provide a powerful mechanism for the combination of grouping operations. Conventional grouping operations [14, 15, 16] are a combination of partitioning and application of aggregate functions. Hence, each partition is mapped to a single tuple that holds the results of the aggregate functions. A regrouping of the partitions is not possible. That means conventional grouping operations cannot be serially combined. So we have separated the partitioning from the

aggregate function. By extending the operator interaction this can be done without creating non-first normal form relations.

Considering the above ideas we suggest the operators *FL-Map*, *BA-Map*, *GroupApply* and *Groupify* for algebraic result construction. But before we start the operator description we give a brief introduction to the construction functions used for result construction.

3.2 Construction Functions

We consider three types of construction functions, print statements, DOM and SAX function, where the print statements simply create a textual representation of an XML document that can be stored as an XML document.

By applying DOM functions query evaluation results in a main memory representation of an XML document. The Document Object Model is a W3C recommendation that defines how to access and update the content, structure and style of documents. Therefore it provides a set of objects for representing hierarchical documents. Objects that represent a document build a treelike structure, called DOM tree. Usually a DOM tree is created by a DOM parser.

SAX is an event-based API for parsing XML documents. That means a SAX conforming parser reports events such as start and end of elements to the application via callback functions while scanning an XML document. The advantage of SAX over the DOM approach is that no main memory structure is built that holds a complete XML document. For the result construction it means that even large query results can be efficiently passed to an XML application.

3.3 The *Map* Operators

The purpose of the *Map* operators is the application of construction functions on tuple sets. To achieve a greater flexibility, we extend the *Map* operator known from the OO context [9]. Concerning an input relation we distinguish several possibilities of applying construction functions. Think of an input relation that builds the content of an XML element. So we need a construction function that creates the open tag of the element before the tuples are converted by a second construction function that transforms each tuple of the input relation to an XML fragment. For the creation of the closing tag we need a third construction function.

For applying construction functions on relations we propose the *FL-Map* and *BA-Map* operator shown in Figure 2 (b). The subscript of the *FL-Map* exhibits three functions *first*, *each* and *last*. During result construction the *first* function is executed by the operator on the first tuple, the *each* function is executed for each tuple and the *last* function is executed on the last tuple of the input stream. The subscript of the *BA-Map* consists of the functions *before*, *each* and *after*. The *before* function is evaluated during the *Open* operation, where the *after* function is evaluated during the *Close* operation.

Obviously the functionality of the *BA-Map* can be subsumed by the *FL-Map* functionality. But during evaluation of the *FL-Map* each tuple of the input relation has to

	DOM	SAX
before	rootNode := doc.createElement(result);	startElement("result",null);

each	n1 := doc.createElement("conference");
	n2 := doc.createElement("title");
	n2.appendChild(doc.createTextNode($t));
	n1.appendChild(n2);
	n2 := doc.createElement("year");
	n2.appendChild(doc.createTextNode($y));
	n1.appendChild(n2);
	rootNode.appendChild(n1);
		startElement("conference",null);
		startElement("title",null);
		characters($t);
		endElement("title");
		startElement("year",null);
		characters($y);
		endElement("year");
		endElement("conference");

after		endElement("result");

Table 1: Construction function for the query plan shown in figure 3.

be temporally stored. This may lead to additional costs which can be avoided by the *BA-Map* operator.

To illustrate the application of the *BA-Map* operator, let us consider the construction plan depicted in Figure 3. It constructs the result of Query 1. The input of the plan is a relation with the attributes `$t` and `$y`. The plan consists of a single *BA-Map* operator where the *before* function generates the open tag of the enclosing `<result>` element. The *each* function creates a `<conference>` element for each tuple of the input relation. The *after* function generates the close tag of the `<result>` element.

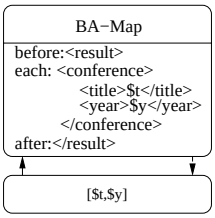

Figure 3: Construction plan of Query 1.

To simplify the exposition the plans we give in this paper do not show which result construction functions are applied by the construction operators. Instead, the construction functions are replaced by XML fragments. To give an idea how this fragments are created, table 1 shows the DOM and SAX construction functions applied by the *BA-Map* operator of the query plan depicted in figure 3.

3.4 The *Groupify* Operator

A more complex functionality is provided by the *Groupify* operator depicted in Figure 2(c). It partitions an input relation according to a given attribute list. That means all tuples of a partition have equal values in their grouping attributes. For the partitioning the *Groupify* operator informs the caller of the *GetNext* operation on the occurrence of a partition limit by returning an end of tuple stream.

In order to read the tuples of the next partition the caller has to invoke the *GetNextGroup* operation. If there is no next partition the operation returns an end of tuple stream. The *Groupify* also enables the evaluation of more than a single subplan by providing the *ResetGroup* operation. This operation resets the tuple stream so that the tuples of the actual partition can be read again.

Finally, the subscript of the operator contains a *first* function in addition to the attribute list. This function is executed by the operator on the first tuple of the tuple input stream. For the *Groupify* operator we propose a sort-based implementation. That means we first sort the input tuples according to the attribute list. Hence the partitioning can be done with constant memory. Further, we separate the sorting from the sort-based partitioning. So a *Groupify* only detects partition limits based on the order of the input relation. Before we can illustrate the *Groupify* operator by an example we have to consider the *GroupApply* operator.

3.5 The *GroupApply* Operator

In order to evaluate subplans on partitions generated by a *Groupify* operator we need the *GroupApply* operator 2(c). In addition to a reference to a *Groupify* the operator gets a list of operators. Each of the list's elements represent a subplan that has to be evaluated on the partitions produced by the referenced *Groupify*.

For result construction the *GroupApply* sequentially reads the tuple stream of each subplan. For each tuple stream the *GroupApply* returns the last tuple via its *GetNext* operation. Between reading the tuple stream of two subplans the *GroupApply* has to reset the *Groupify* by calling the *ResetGroup* operation. After the *GroupApply* has read the tuple streams of all subplans it calls the *GetNextGroup* operation of the *Groupify* and starts rereading the tuple streams of the subplans. In order to get terse construction plans the subscript contains a *last* function that is evaluated on the last tuple of each partition.

Figure 4 illustrates the application of *GroupApply* and *Groupify* operators by showing the construction plan of Query 2[1]. The input of the plan is a relation with the attributes `$j`, `$t` and `$a`. The *before* function of the *BA-Map* operator 1 generates the open tag of the `<result>` element, while the *BA-Map* 3 creates the close tag of the `<result>` element.

The subplan between the two *BA-Maps* partitions the input relation according to the values of the `$j` attribute. For each partition a `<journal>` element is created. The partition is done by the *Groupify* operator 1. With its *first* function it creates the open tag of the `<journal>` element and the `<name>` element that contains the journal name. The close tag of the element is created by the *GroupApply* operator 1 that controls the *Groupify* 1. The content of the `<journal>` element is created by the subplan between the *Groupify* 1 and the *GroupApply* 1.

By application of *Groupify* 2 the subplan partitions the input relation according to the attributes `$j` and `$t`. For each partition an `<article>` element is created. Therefore the *first* function of the *Groupify* 2 generates the open tag of the `<article>` element and the `<title>` element containing the title of the article. The controlling *GroupApply*

[1] To simplify the construction plans we leave out the *Sort* operations.

2 creates the close tag. For the content the enclosed *BA-MAP* operator 2 creates an `<author>` element for each tuple of the partition.

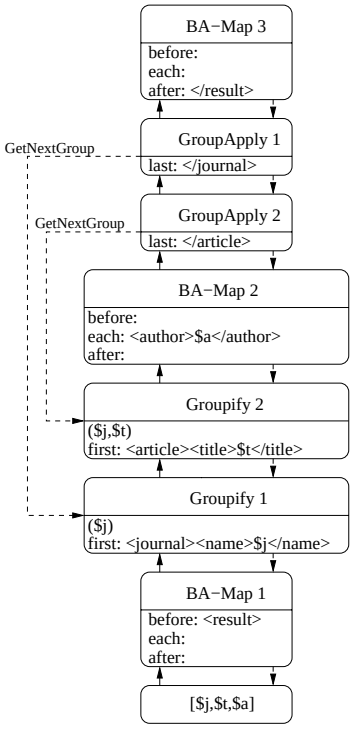

Figure 4: Construction plan of Query 2.

A more advanced example is provided by the result construction of Query 3. The construction plan shown in Figure 5 exhibits a parallel combination of grouping operations. As in the previous construction plan the root and the leaf of the construction plan is built by *BA-Map* operators that create the enclosing `<result>` element. The enclosed subplan consists of a *GroupApply*, *Groupify* pair that partitions the input relation according to the attribute `$c`. The pair creates a `<conference>` element for each partition.

The content of the `<conference>` element consists of the elements `<authorlist>` and `<articlelist>`. These elements are created by the two subplans that are applied to each partition by the *GroupApply* operator 1. The operator first evaluates the left subplan and then the right one.

The left subplan performs a partitioning on the input tuple stream according to the `$a` attribute. Therefore it contains a *GroupApply*, *Groupify* pair that creates an `<author>` element for each partition. The pair is followed by the *BA-Map* operator 2 that finishes the construction of the `<authorlist>` element.

For the construction of the content of the `<articlelist>` element the right subplan partitions its input tuple stream according to the `$t` attribute. For each partition the *Groupify* operator 3 constructs an `<article>` element.

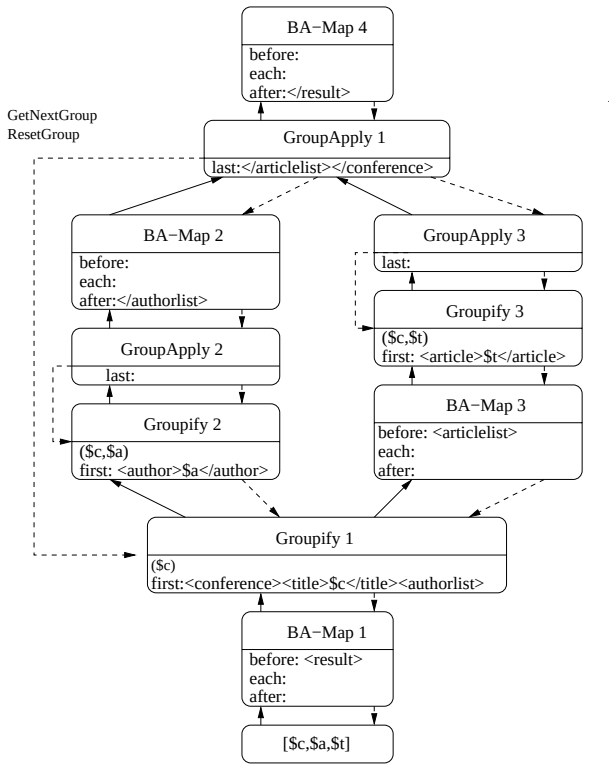

Figure 5: Construction plan of Query 3.

4 Generating Construction Plans

In this section we show how a declarative construction specification can be translated into a construction plan. Therefore we introduce an algorithm that translates YATL construction trees. To avoid the generation of plans that contain unnecessary operators the algorithm applicates operator and plan coalescence during the translation. By the coalescence of two plans we mean the concatenation of the plans followed by an elimination of unnecessary operators. By operator coalescence we mean merging two operators into a single one.

Our algorithm for the translation of a YATL construction tree is based on a depth first traversal. During the traversal it recursively creates a subplan for each subtree. Their combination yields the final construction plan. For the translation of a subtree rooted at node n the algorithm performs four steps. First, it generates a subplan that initiates the construction of the subtree rooted at n. Second, it generates a subplan for each subtree rooted at a child of n, which leads to a collection of so-called content plans. Then the algorithm generates the plan that finalizes the construction of n. In the last step the algorithm glues together the initiating plan, the content plans and the finalizing plan to a single plan for the whole subtree.

4.1 Generating the Initiating Subplan

For the construction of the initiating plan for a node n the label of the incident edge is important. If the edge is not labeled, the initiating subplan is a single *FL-Map* operator. Its subscript depends on the format of the result. For the generation of textual XML data and if n is labeled with an XML element, the *first* function prints the open tag of the element. If the incident edge is labeled with a skolem function, the initiating plan consists of a *Groupify* operator. The grouping attribute list in the operator subscript is built by the variable list of the skolem function. The *first* function depends on the node label. Since the * label is a special skolem function that describes a grouping according to all attributes of the input relation, the initiating subplan consists of a *Groupify* operator. The attribute list contains all the attributes of the input relation.

For an example generation we assume the YATL construction tree of Query 3 that is shown in Figure 6.

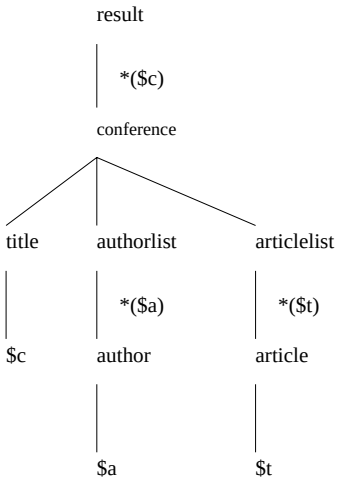

Figure 6: Construction tree of Query 3.

Suppose that we want to generate the subplan for the subtree rooted at the `<conference>` node. The subplan that initiates the construction of the root node is shown in Figure 7.

4.2 Generating the Content Subplans

The creation of the subplans for the children of n results in a collection of subplans, that is also shown in Figure 7.

Our algorithm tries to coalesce these plans. For the coalescence of two subplans we have to distinguish four cases. In the first one both subplans do not contain a grouping operation. In the second only the first subplan contains a grouping operation. In the third only the second subplan contains a grouping operation. Finally, both subplans contain a grouping operation. In the latter case the plans cannot be coalesced.

In the first and the second case for plan coalescence the first subplan is prepended to the second. This concatenation is followed by an operator coalescence.

We have three rules for the coalescence of construction operators. First, two *FL-Map* operators can be merged into a single operator by combining their subscript functions. Second, a *Groupify* followed by a *FL-Map* operator can be substituted by a single *Groupify*, if the *each* and the *last* function of the *FL-Map* is empty. Finally, a *FL-Map* followed by a *GroupApply* operator can be merged into a single *GroupApply* operator if the *first* and *each* function of the *FL-Map* are empty.

The third case of plan coalescence is more complex. From the fact that the second subplan does not contain a grouping operation it follows that it consists of a single *FL-Map* operator with an empty *each* and *last* function. For the coalescence the *first* function is moved to the *last* function. After this transformation the *FL-Map* operator can be prepended to the first subplan.

In our example shown in Figure 7 the first two content subplans can be coalesced. Since the first subplan does not contain a grouping operation, we apply the first case of plan coalescence. As Figure 7 shows, for operator coalescence the *FL-Map* operator 1 is coalesced with the *FL-Map* operator 3 according to the first operator coalescence rule.

4.3 Generating the Finalizing Subplan

As for the generation of the initiating plan for a node n we have to consider the label of the edge incident to n. If the edge has no label the finalizing plan is built by a single *FL-Map* operator with an empty *first* and *each* function. The *last* function depends on the node label and the result format. For construction of textual XML data and assuming that n is labeled with an XML element, it prints the according close tag.

If the incident edge is labeled with a skolem function or with a $*$ the finalizing plan consists of a *GroupApply* operator. Its *last* function is set according to the label of n and the output format. Since the incident edge in our example is labeled with a skolem function, the finalizing subplan consists of a *GroupApply* operator as shown in Figure 7.

4.4 Gluing it Altogether

For the final step the content plans are prepended to the initiating subplan. If there are several content plans and the last operator of the initiating subplan is not a *Groupify*, a *Groupify* with an empty attribute list and an empty *first* function is prepended. While prepending the first content plan its first operator can be coalesced with the last operator of the initiating plan. Finally the finalizing plan is prepended to the content plans. If there are several content plans and the finalizing plan does not begin with a *GroupApply*, a *GroupApply* is appended to the finalizing plan. While prepending the finalizing plan its first operator can be coalesced with the last operator of the last content plan.The result of the gluing step of our example translation is shown in Figure 7.

If the resulting subplan consists of a single subplan enclosed by a *Groupify* and a *GroupApply* pair where the attribute list of the *GroupApply* contains all attributes of the input relation, the whole subplan can be substituted by a single *FL-Map* operator. Its *each* function is built by the combination of the construction functions of the substituted subplan.

Further, all *FL-Map* operators where the *first* and the *last* function is a constant expression are substituted by *BA-Map* operators. Only the root operator of the subplan is not substituted if it is not the root operator of the whole construction plan. The reason for this is that a substitution could inhibit an operator coalescence.

5 Optimization of Construction Plans

Based on our translation algorithm in this section we describe techniques for the optimization of construction plans. This includes the optimization of serially combined grouping operations and the optimization of parallelly combined construction plans.

5.1 Optimizing Serial Combined Construction Plans

Based on a sort-based implementation of the *Groupify* operator the serial combination of grouping operators can be optimized by factorizing sort operations. Figure 8 shows the construction plan of Query 2 with the added *Sort* operators.

By applying order optimization techniques as described in [28] we are able to push *Sort* operators down the construction plan. In [28] a tuple stream in a query execution plan has an *interesting order* that is defined by a list of attributes. For two *interesting orders* I_1 and I_2, I_2 is a *cover order* of I_1 and I_2, iff I_1 is a prefix of I_2.

Since, the *interesting orders* resulting from the *Sort* operations of serial combined grouping operation exhibits the prefix relationship, the *Sort* operations can be factorized by a single *Sort* operation. Figure 9 shows the result of factorizing the *Sort* operators.

5.2 Optimizing Parallel Combined Construction Plans

The execution of parallel combined construction plans can be optimized by minimization of intermediate result storage and the factorization of *Sort* operations.

An ad-hoc approach for the evaluation of parallel combined subplans would lead into storing the input tuple stream of the subplans in a temporary file. Since our translation algorithm guarantees that parallel combined subplans always act on a *Groupify* this approach can be optimized by storing just a single partition.

If parallel subplans have *interesting orders* that can be covered by a third one we are also able to factorize a single *Sort* operation for the plans. If there is no such *covering order* for all subplans we might still be able to determine a covering order for a subset of the subplans.

6 Preliminary Performance Results

We ran some preliminary performance experiments to measure the effect of sort factorization and the impact of different types of construction functions. We implemented the construction operators in C++ and integrated them into the native XML database system Natix [17], developed at the University of Mannheim. We conducted the experiments on a Sun SPARC Ultra 2 with Solaris 2.6 and 256 Mb main memory. Our data

	Query 1	Query 2	Query 3
Print execution time	3.0 s	3.0 s	4.4 s
SAX execution time	< 1 s	< 1s	< 1s

Table 2: Times spend on the execution of construction functions.

set was the content of the TOC_OUT file from the DBLP server. We extracted from the file the variable bindings for the construction plans of Query 1, 2 and 3.

Executing the different construction plans revealed the benefit of sort factorization. The running time for Query 2 without sort factorization was 20.6 seconds. Sort factorization reduced the running time to 17.6 seconds. For Query 3 without sort factorization we measured a running time of 49 seconds. Considering the construction plan of Query 3, shown in 5 it follows that there is no *cover order* for the whole construction plan. Hence, we considered two alternative factorizations. First, sorting the input variable bindings according to the attributes `$c` and `$a`. The running time was 43.2 seconds. Second, we sorted the input of the construction plan according to the attributes `$c` and `$t`. The running time was 43.6 seconds.

To investigate the impact of different construction functions we measured the time that was needed to create text files or a sequence of SAX callback function calls. Table 2 shows that the time spent on the execution of construction functions can be considerably reduced by applying the SAX API.

7 Conclusions

In this paper we showed how XML can be constructed from table structured variable bindings by applying algebraic operators. In contrast to powerful operators that can perform complex result construction, we proposed the XML construction by construction plans. These are special query execution plans consisting of simple, easy to implement and efficient operators.

For generating construction plans from constructor clauses we introduced an optimizing translation procedure that reduces the number of operators in a construction plan by plan and operator coalescence.

Together with the XML construction by "side effect" our approach provides a great flexibility that can be exploited during query optimization. We sketched some of the optimization possibilities and provided some preliminary performance results for their evaluation.

References

[1] DB2 XML Extender. http://www-4.ibm.com/software/data/db2/extenders/xmlext/.

[2] XML, XSLT and Oracle8*i*.
http://technet.oracle.com/sample_code/tech/xml/xsql_servlet/sample_code_index.htm.

[3] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The Lorel Query Language for Semistructured Data. *Journal of Digital Libraries*, 1(1), November 1996.

[4] C. Beeri and Y. Tzaban. SAL: An Algebra for Semistructured Data and XML. In *International Workshop on the Web and Databases*, pages 37–42, 1999.

[5] T. Bray, J. Paoli, and C. M. Sperberg-McQueen. Extensible Markup Language (XML) 1.0. Technical report, World Wide Web Consortium, 1998. W3C Recommendation 10-Feb-98.

[6] Michael J. Carey, Daniela Florescu, Zachary G. Ives, Ying Lu, Jayavel Shanmugasundaram, Eugene J. Shekita, and Subbu N. Subramanian. XPERANTO: Publishing Object-Relational Data as XML. In *International Workshop on the Web and Databases*, pages 105–110, 2000.

[7] D. Chamberlin, J. Clark, D. Florescu, J. Robie, J. Simeon, and M. Stefanescu. Xquery 1.0: An xml query language. Technical report, World Wide Web Consortium, 2001. W3C Working Draft 07 June 2001.

[8] V. Christophides, S. Cluet, and J. Simeon. On Wrapping Query Languages and Efficient XML Integration. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 141–152, 2000.

[9] S. Cluet and G. Moerkotte. Nested Queries in Object Bases. In *Int. Workshop on Database Programming Languages*, pages 226–242, 1993.

[10] S. Cluet and G. Moerkotte. Query Processing in the Schemaless and Semistructured Contex. Unpublished, 1996.

[11] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. XML-QL: A Query Language for XML. Technical report, World Wide Web Consortium, 1989. http://www.w3.org/TR/NOTE-xml-ql.

[12] Lauren Wood et al. Document Object Model (DOM) level 1 specification, version 1.0, October 1998. W3C Recommendation, available at http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001.

[13] Daniela Florescu and Donald Kossmann. Storing and Querying XML Data unsing an RDBMS. *IEEE Data Engineering Bulletin*, 22(3), 1999.

[14] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database System Implementation*. Prentice Hall, 2000.

[15] G. Graefe. Query Evaluation Techniques for Large Databases. *ACMS*, 25(2):73–170, June 1993.

[16] A. Gupta, V. Harinarayan, and D. Quass. Aggregate-Query Processing in Data Warehousing Environments. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 358–369, 1995.

[17] C.-C. Kanne and G. Moerkotte. Efficient storage of XML data. In *Proc. IEEE Conference on Data Engineering*, page 198, 2000.

[18] R. Laddad. XML APIs for Databases, 2000. http://www.javaworld.com/javaworld/jw-01-2000/jw-01-dbxml_p.html.

[19] Michael Ley. Databases & Logic Programming. http://www.informatik.uni-trier.de/ ley/db/index.html.

[20] H. Liefke. Horizontal Query Optimization on Ordered Semistructured Data. In *International Workshop on the Web and Databases*, pages 61–66, 1999.

[21] D. Maier. Database Desiderata for an XML Query Language, 1998. http://www.w3.org/TandS/QL/QL98/pp/maier.html.

[22] Jason McHugh and Jennifer Widom. Query Optimization for XML. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 315–326, 1999.

[23] D. Megginson. Megginson Technologies. http://www.megginson.com/SAX/.

[24] J. Robie, D. Chamberlin, and D. Florescu. Quilt: An XML Query Language for Heterogeneous Data Sources. In *International Workshop on the Web and Databases*, 2000.

[25] J. Robie, J. Lapp, and D. Schach. XML Query Language (XQL), 1998. http://www.w3.org/TandS/QL/QL98/pp/xql.html.

[26] J. Shanmugasundaram, R. Barr E. J. Shekita, M. J. Carey, B. G. Lindsay, H. Pirahesh, and B. Reinwald. Efficiently Publishing Relational Data as XML Documents. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 65–76, 2000.

[27] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, and J. F. Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 302–314, 1999.

[28] David E. Simmen, Eugene J. Shekita, and Timothy Malkemus. Fundamental Techniques for Order Optimization. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 57–67, 1996.

[29] M. Yoshikawa, T. Amagasa, T. Shimura, and S. Uemura. XRel: A Path-Based Approach to Storage and Retrieval of XML Documents Using Relational Databases. *ACM Transactions on Internet Technology*, 1(1), June 2001.

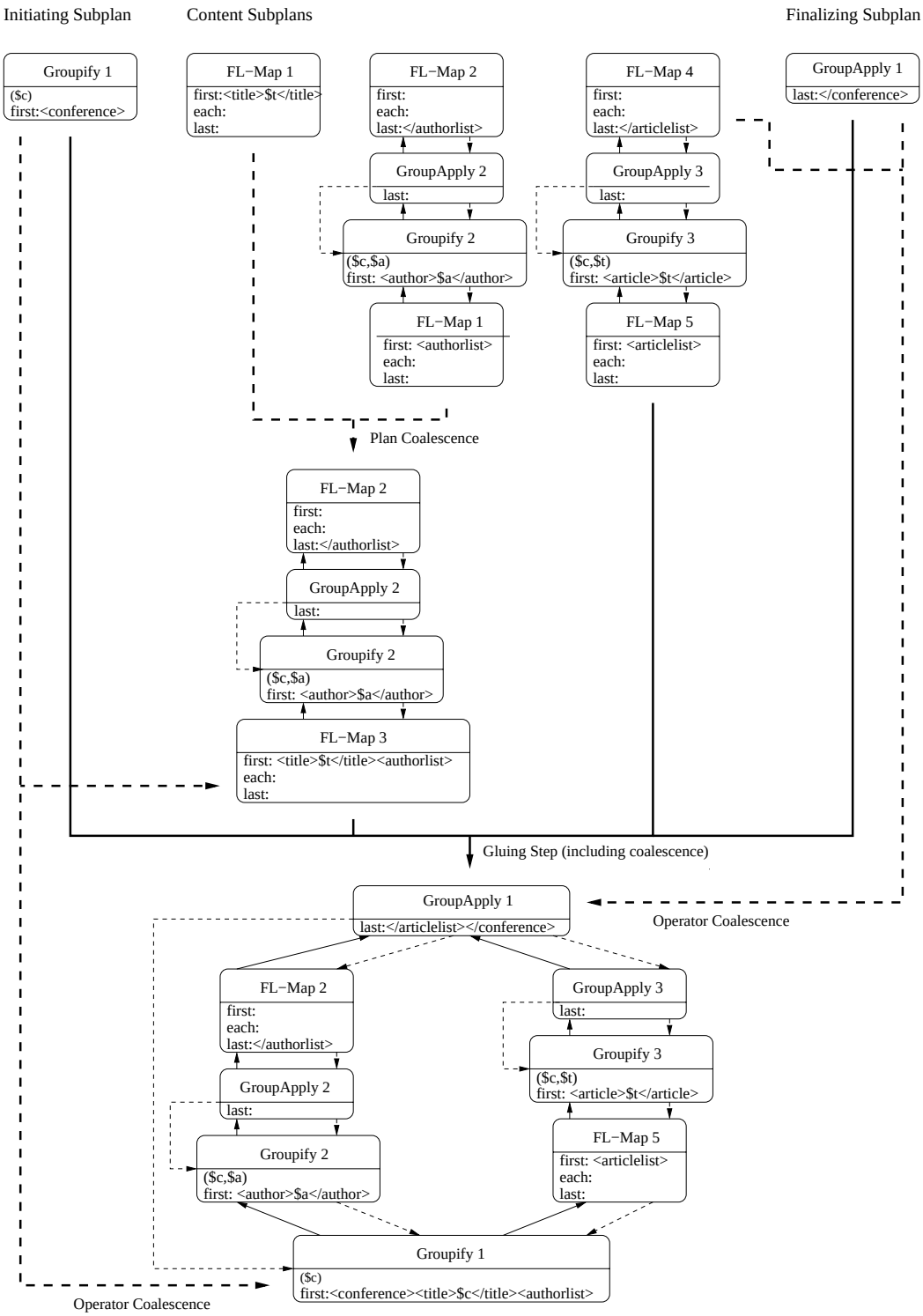

Figure 7: Plan creation for Query 3.

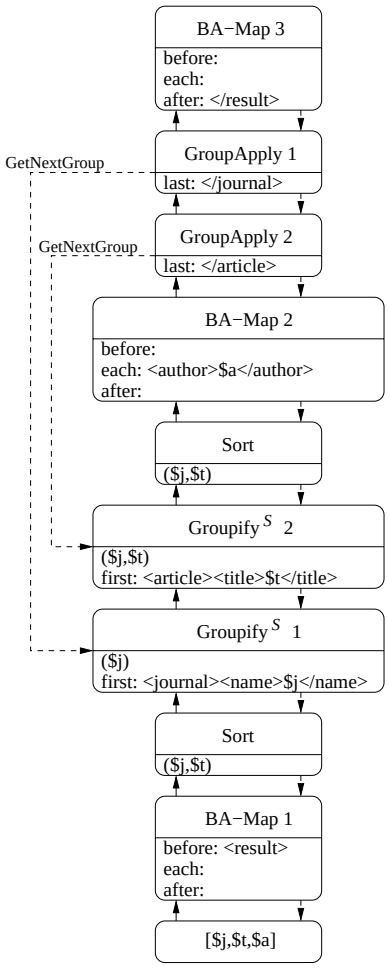

Figure 8: Construction plan of Query 2 with *Sort* operators.

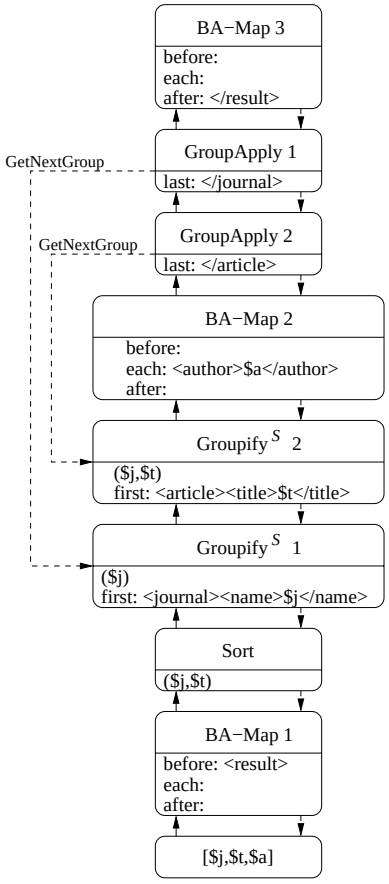

Figure 9: Construction plan of Query 2 with factorized *Sort* operators.