Efficient Implementation of XQuery Constructor Expressions

© Maxim Grinev Leonid Novak Ilya Taranov

Institute of System Programming

Abstract

Element constructor is one of most expensive operations of the XQuery language as it requires deep copy of the nodes which make up the content of the constructed element. In this paper we propose various optimization and implementation techniques to avoid copying of the nodes during constructor evaluation. The proposed techniques are based on using special kind of XQuery constructors with modified semantics which evaluation does not require content node copying. We also provide optimization rules which replace standard constructors with modified ones without changing query result. The proposed techniques are designed to minimize modifications of an existing implementation. Possible technique extensions which might depend on implementation-specific features are also considered. We present results from experimental study of the techniques which demonstrate performance improvement of constructor evaluation.

1 Introduction

Support for element constructors is a powerful feature of XQuery language. Element constructors allow expressing transformation of XML data. Element constructors are also useful for compounding (sub-)query results into a document structured in such a way that it can be efficiently processed by the database client [1]. Although useful, element constructor is one of the most expensive XQuery operations when implemented straightforwardly. According to the XQuery specification [2], constructing a new element node requires deep copy of the nodes that makes up its content. Moreover, the presence of nested (the term nested is used here in the same manner as in 3.7.1 of [2]) element constructors (very common for practical queries) causes repeated deep copying of the same nodes.

Fortunately, a variety of optimizations are possible to avoid content node copying in many cases. In this paper we propose query optimization methods based on element constructors with modified semantics. Such modified constructors do not require copying of content node and in same cases even instantiation of the constructed nodes. Since replacing standard element constructors with modified ones might result in a nonequivalent query for each modified constructor we specify the conditions on which such replacement is allowed.

The conditions can be characterized as follows: a restricted set of operations are applied to the results of element constructors. Maximum effect is achieved when this set consists of only "identity" ("node-preserving") operations such as `return` from *FLWOR* -expression, `then` and `else` from `if`-expression, element constructor, etc. Many practical queries are originally expressed in such a way that they meet the conditions (for example, all queries of popular XMark benchmark [3] and the majority of examples from the XQuery specification). In many other cases queries can be rewritten by the optimizer into an equivalent form that meets the condition as proposed in [4]. Such rewriting techniques are complementary to our method and not considered in this paper.

Introducing our optimization methods we pursue a goal to minimize modifications of an existing implementation required to support the methods. Thus for each method we first describe a basic method which requires minimal modifications of existing implementations and then consider extensions of the basic method which might improve its efficiency and usability but requires more complex modifications to incorporate it into an existing implementation.

The rest of the paper is organized as follows. In Sections 2, 3, 4 we introduce embedded, virtual and serialized constructors respectively and query optimization methods using the constructors. Section 6 gives an overview of related work. Section 5 contains the results of our performance experiments. Section 7 concludes this paper with suggestions for future work.

2 Embedded Constructor

2.1 Motivating Example

Let us consider the following query:

```
for $x in doc('auc.xml')//person return element
                     "user" {
 element "info" { attribute "name" {$x/name}...
                        } }
```

The query results in a sequence of n elements, where n is the number of <*person*> in original XMark[17] auction document. Let us count how many node copies are made during the evaluation of expression. First of all n of <*user*>, <*info*> and *name* nodes are created. For each element <*info*> a copy of attribute *name* is made. Yet, for each element <*user*> a deep copy of element <*info*> is made (i.e. 2*n nodes are copied). In total, we copy 3*n nodes during evaluation of this query. Copying of the nodes can be avoided if nodes constructed in

Proceedings of the Spring Young Researcher's Colloquium on Database and Information Systems, Saint-Petersburg, Russia, 2008

enclosed expressions of newly created elements are constructed as children nodes of the element(like in nesting of direct constructors). In the next section we introduce embedded constructors which create nodes in enclosed expressions in the way described.

2.2 Semantics of Embedded Constructor

In order to define the embedded constructor we introduce a new component of the XQuery dynamic context: *context parent node*. During the constructor evaluation the value of the context parent node is set to the newly constructed element node. It must be done before processing the enclosed expression. After the evaluation of the constructor expression the previous value of the context parent node component is assigned back to the dynamic context. The semantics of the content of an element constructor is changed as follows. Instead of *"For each node returned by an enclosed expression, a new copy is made of the given node..."* there should be: *"For each node returned by an enclosed expression, such that parent property of the node is not equal to the constructed node, a new copy is made..."* The latter means that not all the nodes from the result of the enclosed expression are copied.

We use the context parent node in order to define *embedded constructor*. We modify the semantics of the standard constructor with respect to assigning values to the properties of constructed node as follows: instead of *"parent is set to empty"* there should be *"The parent property is set to context parent node"*.

In order to save space in examples we introduce a new syntax for embedded constructors by adding the prefix *"emb-"* to the keywords (e.g. *emb-element*, *emb-attribute* etc). However we point out that we do not have intension to expand the XQuery language. The embedded constructors are intended to be used in the query execution plan rather than in XQuery expressions written by the user (the same is true for types of constructors described later in the paper).

Let us return to the motivating example described above. This query can be rewritten using embedded constructors as follows.

```
for $x in doc('auc.xml')//person return
          emb-element "user" {
  emb-element "info" { emb-attribute "name"
          {$x/name}...  } }
```

The result of the latter query is the same as of the former one but no node copying is made at all. One can easily see that embedded constructors help to avoid node copying and cut down the time expenses on constructor evaluation (see section 5 for performance results). In fact, this is generalization of the rule of the processing nested direct constructors inside the content of the element constructors (see Section 3.7.1.3-1-d in [2]) in the case of arbitrary constructors. But unfortunately, we cannot replace standard constructors with the embedded ones everywhere in the query. The next section explains why.

2.3 Query Optimization Using Embedded Constructors

We start with the following example.

```
<a>{(<b>text</b>)/..}</a>
```

This query returns `<a></a>`. Suppose that we replace the `<b>` direct element constructor with the embedded version.

```
<a>{(emb-element "b" {"text"})/..}</a>
```

Let us examine the set of instances of the data model created by the latter query. Due to the semantics of embedded constructors the parent property of $<a>$ is set to $<a>$ but it is not contained in the children sequence of $<a>$. It means that we have got the violation of the XQuery data model constraint: *"if a node N has a parent element E then N must be among the children of E"*. Moreover, the violation of the data model leads to the wrong result of the latter query (i.e. `<a><a/></a>`) , because the parent axis applied to $<b>$ returns nonempty result. It is easy to see that the constraint is violated due to the semantics of embedded constructor: while creating $<b>$-element we set the parent value to $<a>$-element. Thus, we should introduce the condition of applicability of the replacement of the XQuery constructors by the embedded ones. In order to do this we have to introduce a set of XQuery expressions named node preserving expressions.

Definition 1 *An XQuery expression is called* node preserving *if for each item n in the result of the expression there is the same (or equal in case of atomic values) item in the input of the expression and vice versa each input item is appended to the result of the expression.*

We denote the set of node-preserving expressions by Δ. Δ includes sequence expressions, `then`- and `else`-expressions, return clause of *FLWOR* expression, etc.

For any given query q we define the class of constructor expressions C_q such that for any e from C_q the following holds: *there is another constructor expression f such that e is nested to the content of f and for any expression g, such that, e is nested to g, nested to the content of f, g belongs to Δ*

As the latter condition might seem to be obscure let us consider various cases when g does not belong to Δ in order to show that optimization of e is impossible:

a. Path expressions: let g be XPath expression (for instance, `(element a {...})//b[..]//c..)`. It is obvious that the result of the path expression may differ from the value returned by embedded constructor;

b. Non-element and non-document constructors: `attribute a {<a>txt</a>}`. The value of the node created by embedded constructor is normalized. Thus the embedding is impossible.

c. Comparisons, Instance of, Cast, Quantified Expressions and all the other expressions which either require atomization of returned values of nested expressions or return atomic values.

d. `Where` and `Order By` clauses of *FLWOR* .

e. `For` and `Let` clauses: `(element a {for $x in <b/> return 1})`. In fact, the usage of the constructors nested to the `for` clause is a complex issue which is the subject of further researches. Let us take a look at the following example: `element a {for $x in <b/>return $x}`. As one can see, the embedding of inner element is possible. But in order to apply the optimization we have to distinguish different cases of using binding variables inside the return clause.

In order to complete the description of optimization framework for embedded constructors we have to prove the following proposition:

For the given query Q: any constructor expression from C_Q can be replaced with the embedded version.

The usage of the formal algebra for XQuery [5] in order to provide the rigorous proof of the statement is the subject of our future research.

2.4 Further Extensions

The optimization method described above is generic and does not depend on the peculiarities of the implementation of XQuery. Now let us describe other optimization methods which depend on the properties of an XQuery engine.

Resetting the parent property. Consider an arbitrary expression which does not belong to Δ, thus making the optimization of nested constructor expression impossible. Let us change the semantics of that expression. First, in the beginning of the evaluation of the expression, before any of sub-expression is evaluated, we set the value of context parent node to empty sequence. Then, at the end of the evaluation of the expression, before the result is returned, we reassign the previous value of context parent node.

Due to the semantics of the embedded constructor the node-preserving requirement for intermediate expressions becomes obsolete thus making the optimization algorithm much easier. Unfortunately this improvement requires the modification of the semantics of majority of XQuery expressions, which can be inadmissible for existing implementations.

Dynamic embedding. Let us forget for a while about the context parent node and embedded constructors and start from the very beginning: we have standard constructors and nothing else. In order to get rid of node-copying operations we change the semantics of the constructors. In the definition of contructors (see Section 3.7.1.3 in [2]) instead of *"For each node returned by an enclosed expression, a new copy is made of the given node..."* should be: *"For each node returned by an enclosed expression, such that parent property of the node is not set to empty, a new copy is made ... otherwise the parent property of the node is set to the node constructed by this constructor and the node returned by enclosed expression is appended to its children sequence"*. One can easily see, that there will be no redundant node-copying and the embedding is made automatically (there are some exceptional queries with variable binding and utilization of node identity and parent properties of the node – the discussion of these artifacts is behind the scope of the work). But the dynamic embedding (i.e. the ad-hoc modification of the parent property of the node) is hardly implemented in some XQuery implementations especially in database systems: it changes the tree structure that can be unacceptable in presence of numbering schemes [6], data guides or some other type of descriptive schema and structural XML indexes [7].

2.5 Constructors in Let clauses

Let us consider the following example.

```
let $x := <a/> return element "b" {$x}
```

As it is shown in [2, Basics section] XQuery does not allow variable substitution if the variable declaration contains construction of new nodes. However at the same time in case of lazy evaluation of XQuery queries the creation of the element $<b>$ happens before the creation of the element $<a>$, thus making the embedding of $<a>$ to $<b>$ possible. So, if the implementation supports lazy evaluation the constructor nested to the Let clause can be replaced with the embedded version. Of course it works only for Let expressions and only when there is no `where` clause.

3 Virtual Constructor

Whereas embedded constructors allow avoiding copying of elements constructed in enclosed expression virtual constructors are designed to avoid copying of persistent nodes retrieved in enclosed expression.

3.1 Motivating example

Let us consider the following query.

```
                    <result>
{document("source.xml")//item}</result>
```

According to the semantics of the standard element constructor a deep copy of each node returned by the XPath expression is created. We can avoid copying the nodes by referencing them as children elements of the newly constructed result element. It is not allowed by the XQuery data model as in this case the nodes will have more then one parent. Nevertheless in this particular query it will not lead to any anomaly as there are no operations that traverse the result XML tree. In the following section we introduce virtual constructors that can be used to implement such optimization.

3.2 Semantics of Virtual Constructors

First of all, we extend the XQuery data model[10] with the new kinds of nodes: *Virtual Document (VDocument)* and *Virtual Element (VElement)*. In order to save space we consider only VElements. VElements have the same properties and satisfies to the same constraints as standard Elements except the following:

1. the **children** and **attribute** properties are always empty sequences;
2. new properties, **child-list** and **attr-list** are added;
3. the **child-list** must consist exclusively of Element, Processing Instruction, Comment, and Text Nodes if it is not empty. Attribute, Namespace, and Document Nodes can never appear in **child-list**;
4. the Attribute Nodes from **attr-list** of an VElement Node must have distinct `xs:QNames`;
5. the **string-value** property of a VElement Node must be the concatenation of the **string-values** of the nodes from the **child-list** in document order.

The **child-list** and **attr-list** properties are similar to **children** and **attribute** properties except that there are no constraints set on the **child-list** and **attr-list** properties which require the parent-child relationship between the VElements and nodes from the **child-list** or **attr-list** sequences. We use **child-list** and **attr-list** properties instead of children and attributes in order to serialize[9] virtual node instances.

Next, we introduce a function which converts virtual nodes to standard nodes:

```
fn:materialize ($arg as item*) as item*
```

Each atomic value and non-virtual node from the input sequence is returned unchanged. Each VElement inp is replaced with a new standard Element $result$ constructed according to the following rules:

a. node-name($result$)=node-name(inp);

b. base-uri($result$)=base-uri(inp);

c. string-value($result$)=string-value(inp);

d. the concatenated (**attr-list**, **child-list**) sequence is processed like the enclosed expression in the content of the node constructor (3.7.1.3 section of XQuery [2]) and the parent property of the top-most nodes is set to result.

Finally, we introduce a new type of the constructor expression, namely *virtual element constructor*. A virtual element constructor expression returns VElement instance. Below we use *"virt-"* prefix to keywords to denote the usage of virtual constructors instead of standard ones. The content of the constructed VElement returned by the enclosed expression is processed exactly in the same way as the content of the standard constructor except the following changes:

a. The nodes returned by the enclosed expression **are not copied**.

b. The adjacent text nodes in the content sequence are not merged.

c. The children and attribute properties of the constructed node are left empty.

d. The **child-list** property consists of all the element, text, comment, and processing instruction nodes returned by the enclosed expression.

e. The **attr-list** property consists of all the attribute nodes specified in the start tag, together with all the attribute nodes returned by the enclosed expression.

Proposition 1 *Let $f(...)$ be an arbitrary element constructor expression, e an XQuery query which contains f. The query e' derived from e by replacing the $f(...)$ by $materialize(f'(...))$, where f' is virtual constructor expression, evaluates to the equivalent result.*

We use the notion of equivalency based on the deep-equivalence [8] of the XQuery result sequences. To determine the deep equality for the virtual nodes we use **attr-list** and **child-list** properties instead of children and attribute. As far as the virtual node constructor expression is nested immediately into the materialize function and is side-effect-free, its result can be processed only by the materialize function. It allows us to consider $materialize(f'(...))$ as atomic expression which evaluates to the result equivalent $f(...)$ due to the definition of the $materialize$ function.

Using the proposition we can rewrite the query so that all occurrences of standard element constructors are replaced with virtual element constructors nested in materialize function calls. However the efficiency of the rewritten query should be the same as of the original expression as materialize function copies the nodes from **child-list**. In the next subsection we consider cases when it is possible to remove materialize function calls from query.

3.3 Query Optimization Using Virtual Constructors

Consider we have an XQuery expression e rewritten into the new expression e' with all element constructors replaced by virtual element constructors nested into materialize function calls. First of all, we define two classes of XQuery expressions. The first class, denoted by Σ, consists of the expressions which use children, parent or attribute properties of the input node (e.g. XPath expressions and some accessors). The second class, denoted by Γ, consists of the non-Σ expressions such that none of the input nodes of the expression are returned to the output.

Proposition 2 *If there is no occurrences of Σ-expressions in e' then all the materialize function calls can be removed from the query.*

Let us consider a new query e'', which is constructed from e' by removing all calls to the materialize function. The only difference between e'' and e expressions is that all element constructors are replaced by virtual element constructors. Let f be an arbitrary expression which takes the result of a virtual element constructor as input. As f is not a Σ-expression it does not utilizes children,parent and attributes properties of VElement node n returned by f'. Thus the materialization of n is obsolete in this step. Finally, the octet sequence which represents the serialized virtual node is equal to the octet sequence of the corresponding materialized node.

Proposition 2 allows us to rewrite the following query:

```
<result> {document('source.xml')}</result>
```

In the rewritten query due to the definition of the virtual element constructor we avoid copying the content of 'sources.xml' document. Next, we extend Proposition 2 as follows:

Proposition 3 *An element constructor expression f can be replaced by virtual element constructor expression preserving the result of the query if one of the following conditions holds:*

a. f (or variable reference to f) is not nested to any Σ-expression;

b. f (or variable reference to f) is nested to Σ-expression g, but there exists Γ-expression h, such that f (or variable reference to f) is nested to h, and h (or variable reference to h) is nested to g.

The latter proposition is sufficient to rewrite the query from the motivating example in order to avoid deep cpying of persistent nodes.

```
virt-element "result"
{document('source.xml')//item}
```

However, there is still a common case which does not meet the requirements of proposition 3: when an XPath expression is applied to the constructed node. The typical example of such query is the application of an XPath expression to the transformed version of the original document (see I.4 section of XQuery specification [2]). In this case, we cannot replace the element constructors by the virtual ones as far as Σ-expressions cannot be applied to virtual (non-materialized) nodes.

3.4 Further Extensions

Optimization method based on virtual constructors as proposed above can be realized by extending existing implementations with their minor modifications required. Below we consider extensions which requires more complex modifications of existing implementations.

Postponed materialization The idea of the postponed materialization resembles the dynamic embedding presented in Section 2.4. We can regard the materialize function as a part of functionality of any Σ-expression (i.e we apply the materialization to any input virtual node of a Σ-expression). Thus, we can remove all occurrences of the materialize function from the query and replace all constructor expressions with its virtual forms. The advantage of this approach is that we do not have to extend the XQuery optimizer: instead of checking whether the requirements of Proposition 3 are fulfilled all the constructor expressions can be simply rewritten to the virtual form and the decision to call the materialize function is made during execution when a Σ-expression is evaluated over a virtual node. The disadvantage of this approach is that it requires modifications of semantics and implementation of many XQuery operations.

This approach also requires a further modification of the materialize function. Let us consider the following query.

```
let $x:=<a b="c"/> return $x//@b/..  is $x
```

After rewriting we have the following query.

```
let $x:=virt-element "a" {attribute "b" "c"}
  return fn:materialize($x)//@b/..  is $x.
```

As far as the materialized version of the virtual node has its own identity, the query is evaluated to false while the original query evaluates to true. In order to fix this we have to change the semantics of materialize function as follows: the identity of the new node should be the same as its corresponding virtual node.

Evaluating XPath on virtual nodes Let us consider the following query:

```
let $x := virt-element "result"
{document('source.xml')//item} return
fn:materialize($x)//id
```

To support such XPath expression over virtual nodes we redefine the
`dm:children` accessor on a virtual node. If e is a virtual node then *dm:children(e)* return the **child-list** property of e instead of the children property. As far as descendant axis is defined as transitive closure of child axis, which in its turn is defined via `dm:children` we can assert that we have defined the semantics of some axes (child, descendant, etc.) on the virtual nodes. We can remove the `fn:materialize` function from the above query as the result is equivalent to the original one. Obviously, such optimization is possible only in case of in-depth traversal of virtual nodes.

In order to specify query optimization rules to support such queries we define a new class of XPath expressions, denoted by Λ, which do not use parent property of the nodes. An XPath expression e belongs to Λ if none of the following axes are used in e: following-sibling, following and all the reverse axes. Next, we modify definitions of Σ-expressions: $\Sigma' \equiv \Sigma \setminus \Lambda$. The proposition 3 also holds for Σ' class.

4 Serialized Constructor

In this section we assume that the result of the query is serialized to the *octet sequence*. Next, we introduce another optimization method in addition to embedded and virtual constructors.

4.1 Motivating example

The idea behind serialized constructors described below can be demonstrated by the following example.

```
for $x in document('auc.xml')//person return
<user>$x/name</user>
```

This query results in the sequence of *<user>* elements. It requires constructing of n elements. Since we assume that the query result is serialized to octets, it might be a good idea to rewrite it as follows:

```
for $x in document('auc.xml')//person return
("<user>",$x/name,"</user>")
```

In the latter query we avoid the element constructions at all, while serialized result remains almost the same. To make the second query return the same serialized result as the first one we have to modify it further. The problem is that there is a character expansion phase during serialization. During this phase the "<" and ">" characters are replaced by "&lt;" and "&gt;" correspondingly (escape rules). Thus, the resulting octet sequence contains escape characters for XML tags In order to avoid escaping of "<" and ">" characters during serialization we can use a standard mechanism of character maps which replace "<" and ">" with themselves:

```
declare option use-character-maps ">>"
declare option use-character-maps "<<"
for $x in document('auc.xml')//person return
("<user>",$x/name,"</user>")
```

The latter form of the second query is equivalent to the first query (up to the whitespace characters). This method allows avoiding node construction at all and essentially improves query performance. Below we propose *serialized constructors* which generalizes the method described above.

4.2 Semantics of Serialized Constructors

We start with the definition of a new XQuery function which has the following signature:

```
fn:serialized-value ($arg as node()) as
xs:string
```

`fn:serialized-value` returns the input node serialized to octets. Precisely speaking this function is defined as follows:

a. If `$arg` is an element node, let $child[1..n]$ $\equiv$ `dm:children($arg)` and $attr[1..m]$ $\equiv$ `dm:attributes($arg)`. Then, `fn:serialized-value($arg)` $\equiv$ `fn:concat("<", dm:node-name($arg), " ", fn:serialized-value(`$attr_1$`), ..., " ", fn:serialized-value(`$attr_m$`), ">", fn:serialized-value(`$child_1$`), ..., fn:serialized-value(`$child_m$`), "</", dm:node-name($arg),">");`

b. In order to define the `fn:serialized-value` on attribute nodes we have to expand XQuery type system by adding new simple type: `xs:Attribute` which is the descendant of `xs:string` type and is used to represent serialized attributes. If `$arg` is attribute, then `fn:serialized-value($arg)` $\equiv$ `fn:concat(dm:node-name($arg), "=", dm:string-value($arg)) cast as xs:Attribute`.

c. If `$arg` is text node, then `fn:serialized-value($arg)` $\equiv$ `dm:string-value($arg))`.

d. For other node kinds the function is defined in similar manner.

Now we can define serialized constructors. In order to save space we will define only serialized element and attribute constructors and omit those details that are common for serialized and standard constructors.

Serialized element constructor returns an `xs:string` value. It has the same signature as the computed element constructor and proceeds as follows:

1. The name expression (if exists) is processed just like in the computed element constructor. Let us denote the result of the name expression by **name**. If the keyword `element` is followed by a QName rather than a name expression **name** denotes that QName literal.

2. The content is evaluated to produce a sequence of `xs:string` values called the **content sequence** as follows:

   (a) For each adjacent sequence of one or more `xs:Attribute` values returned by the enclosed expression, a new `xs:Attribute` value is build, containing the result of casting each atomic value to a string, with a single space character inserted between adjacent values;

   (b) For each adjacent sequence of one or more atomic values (which is not of `xs:Attribute` type) returned by the enclosed expression, a new `xs:string` value is build, containing the result of casting each atomic value to a string, with a single space character inserted between adjacent values;

   (c) For each node arg returned by the enclosed expression, the `serialized-value($arg)` is appended to the sequence.

3. If the content sequence contains an `xs:Attribute` value following the value which is not of type `xs:Attribute`, a type error is raised.

4. The content sequence is used to build two `xs:string` values: **attributes** and **content**. Attributes value is concatenation of all `xs:Attribute` values from content sequence with a single space character inserted between adjacent values. Correspondingly, content value is concatenation of all non-`xs:Attribute` values from content sequence with a single space character inserted between adjacent values.

5. If the content value is empty, then the resulting value of serialized expression is concatenation of the following strings: `("<",name," ",attributes,"/>")`. Otherwise, the following strings are merged: `("<", name, " ", attributes, ">", content, "</", name, ">")`.

Serialized attribute constructor returns an `xs:Attribute` value and is defined as follows:

1. The name expression (if exists) is processed just like in computed attribute constructor. Let us denote the result of the name expression by **name**.

2. The resulting value of serialized expression is concatenation of the following strings casted to `xs:Attributes: (name,"=",value)`. Where value is the result of content expression after the processing defined in computed attribute constructor.

Proposition 4 *Constructor expression f can be replaced by corresponding serialized constructor expression preserving the result of the query (up to whitespace characters) if the following holds: (a) f (or variable reference to f) is not nested to any non-Δ- expression, except the content expression of other constructors.*

In order to save space we omitted the descriptions of handling the namespace declaration attributes, typing and some other features of constructors but we hope that one who has an interest to this topic can easily find the solution in order to implement those features in the optimization framework described above.

5 Performance Experiments

In this section, we will discuss the results of experiments for each type of constructors proposed above. The experiments are conducted using Sedna XML database system [15].

Our measurements were performed on the computer equipped with T2300E (1.6 MHz two-core), 1024 Mb of RAM, running Windows. The buffer size for Sedna was set to 100Mb.

We ran each test 6 times and took the average value of last 5 runs (the first run is omitted in order to ignore data buffering time). In our experiments we only measure time needed to execute the query plan.

Figure 1: Embedded constructor performance - large document constructed

scaling	data size	embedded	standard
0.001	115 Kb	0.56 s	13.52 s
0.002	210 Kb	0.86 s	14.98 s
0.003	318 Kb	1.26 s	18.22 s
0.004	457 Kb	1.60 s	26.41 s
0.005	567 Kb	2.61 s	29.96 s
0.006	681 Kb	2.24 s	34.75 s

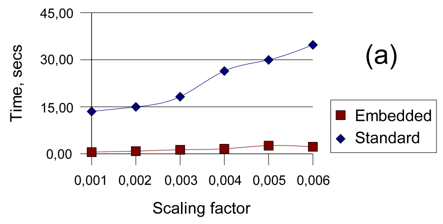
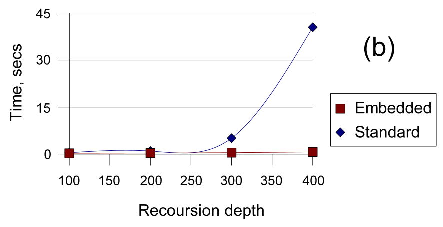


Figure 2: Embedded constructor performance - recoursive construction

recursion depth	embedded	standard
100	0,24 s	0,41 s
200	0,32 s	0,95 s
300	0,46 s	5,04 s
400	0,67 s	40,46 s



5.1 Embedded constructor

Since this type of modifed constructor helps to avoid redudant copying of temporary nodes, embedded constructors should provide the best results when the large amount of nested constructors are used. The *XMark* benchmark [17] is used to generate these constructors.

In the first test we use the XMark document as an input query (i.e. each XML element of the XMark document is evaluated as direct element constructor). We vary the XMark data from 100Kb to 700Kb (approx.) (Fig. 5.1).

The next test is a slightly modified Parts use case from the XQuery test suite [16] (Fig. 5.1). The dataset for this test is modified to make the recursion deep enough to evaluate the performance.

```
declare function local:one_level($p as element())
 as element()
{ element part {
    attribute partid { $p/@partid },
    attribute name { $p/@name },

    for $s in doc("partlist")//part
     where $s/@partof = $p/@partid
      return local:one_level($s) } };

element parttree {
  for $p in doc("partlist")//part[@partid = 1]
   return local:one_level($p)
}
```

Figure 3: Virtual constructor performance

case	expression	nodes	virtual	standard
1	//book	1 227	0,14 s	0,18 s
2	//book/cite	3 319	0,21 s	0,35 s
3	//article/title	$\approx 3 \cdot 10^5$	4,58 s	39,75 s
4	//article	$\approx 4 \cdot 10^6$	103,67 s	649,46 s

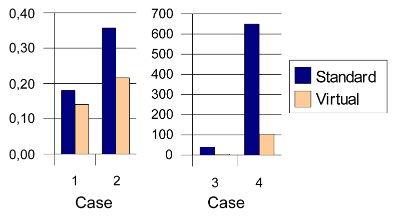
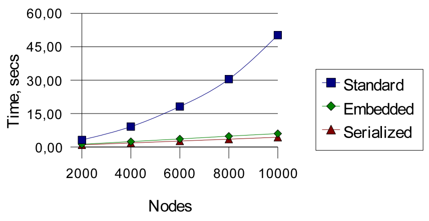


Figure 4: Serialized and embedded constructor performance

case	nodes	standard	embedded	serialized
1	2000	3,16 s	1,22 s	0,95 s
2	4000	9,13 s	2,42 s	1,80 s
3	6000	18,17 s	3,61 s	2,66 s
4	8000	30,43 s	4,85 s	3,50 s
5	10000	50,21 s	6,00 s	4,37 s
6	33333	668,97 s	25,31 s	14,41 s
7	100000	-	190,84 s	43,28 s



5.2 Virtual constructor

We use simple queries (as that presented below) against the DBLP database that generate large amounts of nodes to be copied from the database.

```
<result> { for $i in document("dblp")//book
           return $i } </result>
```

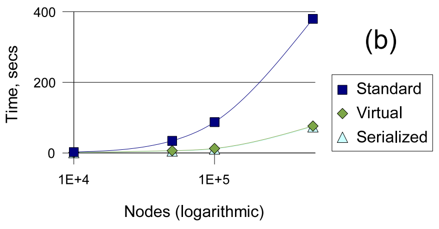
We run the test several times with different XPath expressions varying the size of output document tree.

5.3 Serialized constructor

We run the XMark query Q10 over large XMark datasets generated with various scaling factors. There are many nested nodes that contain simple data in this test. Con-

Figure 5: Serialized and virtual constructor performance

n	standard	serialized	virtual
10000	2,66 s	1,1908 s	1,3874 s
50000	34,55 s	5,6098 s	6,4998 s
100000	87,46 s	11,3126 s	12,9154 s
500000	379,71 s	74,1028 s	76,4968 s



sequetly there are many temporary nodes to be copied. Both embedded and serialized constructors should work fine here. But as our measure show serialized constructors provide better performance.

The execution of the next query can be boosted by both virtual and serialized constructors. The document "auction" is generated by XMark with scaling factor set to 10. The query is as follows.

```
                    for $x in
doc("auction")/descendant::person[position() <
    n] return <result> { $x } </result>
```

6 Related Work

The issue of efficient implementation of XQuery constructors is still not widely presented in the research literature. In contrast to our work where we do query analysis to choose the efficient plan for constructor execution existing techniques solely exploit logical query optimization where a query is rewritten to remove constructor expressions when it is possible. First of all, an algorithm of "predicate push-down" is proposed [4], which helps to avoid intermediate node constructions. Second, the following statement is proved in [11, 12] *"for each expression containing operations that construct new nodes and whose XML result contains only original nodes, there exists an equivalent "flat" expression in XQuery that does not construct new nodes."* The term "flat" originates from SQL and means the table without any nesting tables. In [13] a benchmark intended to validate optimization is described. The authors of the latter work come to conclusion that some implementations use optimization algorithms similar to the ones described above but description of the algorithm are not available in the literature. Finally the efficient implementation of XQuery constructors on SQL hosts [14] is studied.

7 Conclusions and future work

In this paper we introduced various methods of efficient evaluation of XQuery constructor expressions by modifying semantics of the standard constructors. We investigated conditions on which standard constructors can be replaced with new ones in a query plan. Finally, we described some advanced techniques for query optimization which depend on the peculiarities of an XQuery implementation. Experiments presented in this paper prove that using proposed methods leads to significant performance improvements. In further research we plan to provide a formal framework for constructor optimization based on many-sorted algebra representation of XQuery and to prove the criteria of optimization applicability in terms of the algebra. Finally we want to examine our optimization methods for constructor expressions bounded to variables (for and let clauses) in presence of lazy and strict evaluation of a query.

References

[1] A. Boldakov et al. XQuery, XSLT, and OmniMark: Mixed Content Processing, http://www.xml.com/pub/a/2006/12/06/xquery-xslt-omnimark-mixed-content-processing.html

[2] XQuery 1.0: An XML Query Language. W3C Recommendation, 23 january 2007.

[3] Schmidt, A. et al. XMark: A Benchmark for XML Data Management. In Proceedings of 28th VLDB Conference, Hong Kong, China (2002)

[4] M. Grinev, P. Pleshachkov. Rewriting-based Optimization for XQuery Transformational Queries. In IDEAS 2005

[5] L. Novak, A. V. Zamulin. An XML Algebra for XQuery. ADBIS 2006: 4-21

[6] Aznauryan N. et al. "SLS: A numbering scheme for large XML documents". Programming and Computing Software 32(1): 8-18, 2006.

[7] R. Kaushik et al. Updates for structure indexes. In VLDB, 2002.

[8] XQuery 1.0 and Xpath 2.0 Functions and Operators. W3C Recommendation.

[9] XSLT 2.0 and XQuery 1.0 Serialization. W3C Recommendation, 23 january 2007.

[10] XQuery 1.0 and Xpath 2.0 Data Model. W3C Recommendation, 23 january 2007.

[11] Wim Le Page et al. On the Expressive Power of Node Construction in XQuery. WebDB 2005: 85-90

[12] Wim Le Page et al. Expressive Power of Xquery Node Construction . Technical Report 2005-2006

[13] Ioana Manolescu et al. Towards micro-benchmarking XQuery, in Experimental Evaluation of Data Management Systems (EXPDB), 2006.

[14] T. Grust, S. Sakr, J. Teubner. XQuery on SQL Hosts, VLDB 2004, Toronto, Canada, August/September 2004

[15] Sedna XML Database Home Page, http://www.modis.ispras.ru/sedna

[16] XML Query Test Suite, http://www.w3.org/XML/Query/test-suite/

[17] An XML Benchmark Project, http://monetdb.cwi.nl/xml/