Efficient Physical Operators for Cost-based XPath Execution

Haris Georgiadis
AUEB
harisgeo@aueb.gr

Minas Charalambides
AUEB
minchar86@gmail.com

Vasilis Vassalos
AUEB
vassalos@aueb.gr

ABSTRACT

The creation of a generic and modular query optimization and processing infrastructure can provide significant benefits to XML data management. Key pieces of such an infrastructure are the physical operators that are available to the execution engine, to turn queries into execution plans. Such operators, to be efficient, need to implement sophisticated algorithms for logical XPath or XQuery operations. Moreover, to enable a cost-based optimizer to choose among them correctly, it is also necessary to provide cost models for such operator implementations. In this paper we present two novel families of algorithms for XPath physical operators, called LookUp (LU) and Sort-Merge-based (SM), along with detailed cost models. Our algorithms have significantly better performance compared to existing techniques over any one of a variety of different XML storage systems that provide a set of common primitive access methods. To substantiate the robustness and efficiency of our physical operators, we evaluate their individual performance over four different XML storage engines against operators that implement existing XPath processing techniques. We also demonstrate the performance gains for twig processing of using plans consisting of our operators compared to a state of the art holistic technique, specifically Twig2Stack. Additionally, we evaluate the precision of our cost models, and we conduct an analysis of the sensitivity of our algorithms and cost models to a variety of parameters.

Categories and Subject Descriptors

H.2.1 **[Information Systems]**: Physical Design: *Access methods,*
H.2.3 **[Information Systems]**: Languages: *Query languages,*
H.2.4 **[Information Systems]**: Systems: *Query processing,*

General Terms

Algorithms, Measurement, Performance, Experimentation

Keywords

XPath, XML, Cost Models, Physical Operators

1. INTRODUCTION

There has been a lot of research and development activity in the area of XML data management in the last decade, e.g. [3],[6],[16]. A large number of techniques and systems have been developed with the goal of providing more efficient access to XML data. Many of the proposed techniques display benefits for specific query and data set characteristics, yet none can claim universal applicability. Moreover, often their coarse granularity (e.g., at the level of a query instead of an *operator*) makes it hard to take full advantage of their benefits by combining them with other techniques to perform more complex querying tasks. At the same time, the benefits of many powerful techniques are intertwined with the existence of specific auxiliary data structures (e.g. XB-Trees [7]) or XML encoding schemes (e.g. pre-post encoding [2]), making it harder to evaluate the benefit of the algorithm in a different setting. Such a separation is critical if existing and new algorithms for XML path and twig matching and filtering are to be effectively used in the context of modular query processing systems such as those used by relational databases; namely systems that create *plans* for executing queries that consist of *operations* on the data that use intermediate results of other operations. In such an environment, *operators* use the *access methods* provided by an underlying *storage engine*, as shown in Figure 1, which may or may not implement the specialized data structures necessary for the optimal performance of a particular technique. In our recent work [10] we provided such a framework that performs cost-based optimization and execution of XPath independently of both the underlying XML storage system and the techniques and algorithms used for XPath processing.

In any such system, it is important to have available a variety of high performing algorithms for XPath operations. Cost models for predicting the performance of each specific technique on specific XML databases are also critical. Despite significant amount of activity on computing the cardinality and selectivity of XPath queries [11][12], there is very little work on cost models for XPath processing.

In this paper we present two novel, efficient families of algorithms for performing the necessary forward and backward XPath navigation operations for XPath execution. In other words, we provide algorithms for implementing a full set of physical operators for an XPath execution engine, along with cost models, which we validate on a varied query workload. We study the performance characteristics of our operators on different storage engines, with varied XML encoding schemes, each providing access methods with different implementations and costs.

The main contributions of this work are the following:

- We propose the LookUp (LU) family of physical operators for XPath (Section 4), inspired by indexed nested loops join algorithms. Novel efficient algorithms are presented for holistically evaluating forward and backward multi-step paths deploying new techniques for pipelined duplicate elimination and document order preservation
- We describe the SortMerge-based (SM-based) family of physical operators (Section 5) that is inspired by sort-merge join algorithms. Novel techniques for holistic SM-based *forward path* and *backward path* operators with guaranteed low memory requirements are presented.
- We provide and experimentally validate cost models for the LU and SM-based families of operators (Sections 4.4 and 5.1)

EDBT 2010, March 22-26, 2010, Lausanne, Switzerland.

• We describe an XPath execution framework for deploying XPath physical operators (Section 2) along with their cost models and different XML storage systems. Such a framework allows for direct comparison of the available operators by a cost-based XPath optimizer. We present five different storage engine implementations (Section 3), corresponding to storage and indexing strategies proposed in recent literature [2][4][8].
• We provide a detailed experimental analysis of the algorithms' performance, including sensitivity to input and query characteristics on different storage engines (Sec. 6). We also provide performance comparisons to operators implementing well-known XPath processing techniques, namely PathStack [7] and Staircase Join [2] (Sec. 6.1). In *all* cases LU or SM-based operators are significantly faster. Query plans for twigs using a combination of our operators have far better performance than Twig^{2}Stack [14], a state of the art algorithm for twig matching.

2. ARCHITECTURE

Our XPath execution framework, illustrated in Figure 1, consists of three basic components: the *Query Parser*, the *Physical Plan Selector* and the *Physical Plan Executor*. Independence from the XML Storage System implementation is achieved via the *XPA API* (standing for XML Primitive Access API). An input XPath expression is parsed by the query parser, which generates a logical plan as its algebraic representation in XPAlgebra. Using this initial logical plan, the best physical plan is generated by the Physical Plan Selector, which retrieves the costs of candidate physical operators from their corresponding *Descriptors*.

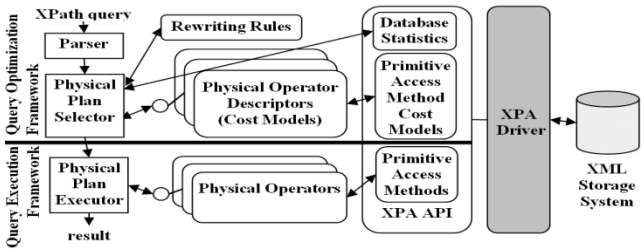


Figure 1. Architecture of our XPath execution framework

Our *Query Execution Framework* can support multiple XPath processing and XML storage methods. In order to use a different XML Storage System, one only has to provide the implementation of an XPA driver and need not modify either the existing Physical Operators or their Descriptors. This is due to the fact that Physical Operators do not have direct access to the underlying XML Storage System. Instead, they make use of a series of primitive access methods (abv *PAMs*), available through the *AccessMethods interface* of the XPA API. The cost model provided by a physical operator *Descriptor* relies on the cost models of any *PAM* calls made by the operator. These are available to the descriptor through the XPA API, which stands as an abstraction layer between the XPA Driver used and the rest of the system.

2.1 Logical Data Model

The core components of our simplified data model are *Element* and *Sequence*. An XML element has the following properties: an abstract *ID* which uniquely identifies the element, the tag name *tagname*, (possibly) a text node value *text*, and a map which associates attribute names to attribute values, named *attributes*. To keep the data model abstract enough to support different physical data models and storage techniques, the *Element* interface does not force elements to keep references to parent or child elements (though concrete implementations may explicitly include such information and more).

2.2 XML Primitive Access Methods API

The XPA API consists of the five interfaces illustrated in Figure 2. *Element* and *Sequence* correspond to the two respective components of the logical data model. The *Sequence* interface is virtually a control structure for the successive traversal of elements, similar to non-scrollable (i.e., forward-only) cursors.

Element
getName(): String
getID(): ID
getAtts(): List
getAttrValue(name): String
hasTextValue(): Boolean
getTextValue(): String
isChildOf(Element): Boolean
isDescOf(Element): Boolean
isParOf(Element): Boolean
isAncOf(Element): Boolean
isBefore(Element): Boolean
isAfter(Element): Boolean
Preceeds(Element): Boolean
Follows(Element): Boolean
getRTNPath(): Boolean

DBStatistics
Card(path): integer
Occ(basePath, path): float
Sel(basePath, path): float
DistValues(element, attrOrTextNode): integer

Sequence
open(): void
next(): Element
close(): void

AccessMethods
Children(element, tag) Sequence
Descs(element, tag): Sequence
Parent(element, tag, virtual): Sequence
Ancs(element, tag, cousinEl, virtual): Sequence
getElByNameAndID(tag, id): Element
DescsInRange(elem1, elem2, tag): Sequence
AncsInRange(elem1, elem2, tag): Sequence

AccMCostModels
CostForDescLookup(): float
CostForNextDesc(): float
CostForAncLookup(): float
CostForNextAnc(): float
CostForParLookup(): float
CostForChildLookup(): float
CostForNextChild(): float
CostForRTNPathRetrieval(): float

Figure 2. The XPA API Interfaces

The *AccessMethods* interface provides a pool of primitive access methods with the purpose of being used by physical operators to access XML data. All access methods return document ordered and duplicate free (*DODF*) sequences of *Elements*. Methods *Children(element, tag)* and *Descs(element, tag)* take as input an element (*element*) and a tag name *tag* and return a sequence of all elements of that *tag* that are children or descendents of *e*, respectively. Method *Parent(element, tag, virtual)* returns the parent of *element*. If *tag* is specified, then the parent is returned only if it is of that tag name. If *virtual* is *true*, then the returned elements will only contain their *ID* and *tag* name. This yields performance improvements for specific storage implementations (see Section 3.3) that allow indirectly evaluating the *ID* and tag name of ancestor elements without having to access the actual XML data (e.g. [8] and [3]). Similarly, method *Ancs(element, tag, cousinEl, virtual)* returns a sequence of the ancestors of *element*. The optional argument *cousinEl* restricts the results only to ancestors that are not also ancestors of element *cousinEl*. Such an access method has an efficient implementation for all considered storage engines, as described in Section 3. *DescInRange(elem1, elem2, tag)* returns elements of tag name *tag*, being descendants of *elem1* and lying between *elem1* and *elem2* and *AncsInRange(elem1, elem2, tag)* returns the ancestors of *elem2* that have tag name *tag* and lie between *elem1* and *elem2*.

The specific *PAMs* model the navigation-based access methods (similar to DOM navigational methods) provided by many storage managers for XML databases, including Natix [6], Timber [9] relational and shredding-based systems such as Pathfinder [2], XRel [4], PPFS [8] and the access methods used by the Staircase Join [2] and the TwigStack "family" of algorithms [14][7].

Interface *AccMCostModel* provides information about the (execution) cost of the abovementioned *PAMs*. This information is used in defining physical operator cost models, thus being agnostic to the actual implementation of the *PAMs*. For example, *CostForDescLookUp()* returns the average cost for retrieving the

first descendant of an element provided that its tag name is specified. The average cost for shifting to the next descendant element is given by *CostForNextDesc()*. Note that if the tag name of elements involved in an access method call is provided to the cost model, conceivably more accurate estimates could be provided. Developing such cost models for *AccessMethods* is the topic of future work.

Interface *DBStatistics* provides basic statistical information about the stored XML documents. The declared methods can be implemented using any XML cardinality and selectivity estimation technique such as [11][12], as long as the required metrics are maintained by the XML Storage System. Use of more sophisticated cardinality estimation techniques for the implementation of these basic interface methods may increase their precision and/or performance. *Card(path)* returns the cardinality estimation for a given non-predicated absolute forward path (*abs-fp*) *path*. *Sel(basepath,path)* returns the probability that an element conforming to *basepath* (that has to be a non predicated *abs-fp*) 'survives' an existential filter with the given *path*. *Occ(basepath,path)* returns the estimated average number of elements *e* such that, for each element *e'* conforming to *basepath*, *e* is in the result of following path from *e'*. Finally, *DistValues(tag, attrOrTextNode)* returns the number of distinct values of attributes or text nodes belonging to an element with a given tag name.

2.3 Logical Operators

XPAlgebra [10] is a high level and compact logical algebra appropriate for a fine-grained algebra-based translation of XPath that preserves its navigational nature and semantics. It covers a large subset of XPath that includes forward (child, descendant) and backward (parent, ancestor) axes, wildcards and non-positional predicates involving conjunctive Boolean expressions that don't involve comparisons between paths. Translation of an XPath expression into a basic XPAlgebra representation is straightforward. XPAlgebra operators are divided into *Sequence* Operators and *Boolean* Operators, with the first returning a sequence of nodes and the second resulting in a boolean value when invoked. In directly translating XPath to XPAlgebra, a series of *Sequence* Operators correspond to the steps of the main (a.k.a. backbone) path of the XPath expression, whereas *Boolean* Operators derive from expressions inside predicates.

Both the input and the output of a *Sequence* operator is a sequence of elements. The sequence operators of XPAlgebra are presented in Table 1. The first seven perform navigation into an XML document. The results of these sequence operators are *DODF* sequences. The c_a operator, corresponding to the child axis of XPath, takes as input a sequence *S* and returns the union of all *a* children for each element of *S*. Operators d_a, p_a and a_a corresponding to the descendant, parent and ancestor axis respectively, are similarly defined. fp_p takes as input a sequence *S* and returns the *DODF* union of all descendants for each element of *S* that is under relative path *p*. The relative path *p* is a simple forward XPath expression (may include '//' and/or '*'), with no predicates. Similarly, given a sequence *S*, bp_p performs backwards navigation via the relative backward path *p*. In what follows, axes /parent:: and /ancestor:: are abbreviated to ^ and ^^, respectively. The novel cousin operator $cs_{p1,p2}$ does not directly correspond to any XPath axis. Given an input sequence *S*, the returned sequence consists of those 'cousin' elements of the elements in *S* that are reachable by fist navigating backwards on the backward path *p1*, then navigating from that ancestor on forward path *p2*. Boolean operators are applied to a single node and return boolean values. Each boolean operator has a 'matching' sequence operator. The filter operator *f,* corresponding to XPath predicates, takes as input a sequence *S* and a boolean expression *BoolExpr,* which is either a constant or a conjunction of one or more boolean operators and returns a subsequent of *S*. The basic XPAlgebra operators are enumerated in Table 1. More details about XPAlgebra and all its operators can be found in [10].

Table 1. Basic XPalgebra operators

c_a(S:sequence)	child	$Ъc_a$(BE: BoolExpr)	boolean child
d_a(S:sequence)	descendant	$Ъd_a$(BE: BoolExpr)	boolean descendant
fp_p(S:sequence)	forward path	$Ъfp_p$(BE: BoolExpr)	boolean forward path
p_a(S:sequence)	parent	$Ъp_a$(BE: BoolExpr)	boolean parent
a_a(S:sequence)	ancestor	$Ъa_a$(BE: BoolExpr)	boolean ancestor
bp_{path}(S:sequence)	backward path	$Ъbp_p$(BE: BoolExpr)	boolean backward path
$cs_{p1,\ p2}$ (S:sequence)	cousin	$Ъvf_{\ n\ op\ v}$	boolean value filter
f (S:sequence, BE: BoolExpr)	filter		

Example 1: The XPath /r//c[e/*/f='x']/parent::d/ancestor::b is translated into the following algebraic expression: $bp_{\wedge d \wedge\wedge b}(f(fp_{/r//c}(root),\ Ъfp_{/e/*/f}(Ъvf_{text()=x})))$. □

It is important to note that all operators receive and produce *DODF* sequences of elements. A different approach would be to have operators that produce results with duplicates and use a separate deduplication operator, e.g., per relational algebra. The performance implications of this choice for XPath processing are not obvious, and we plan to investigate such a direction in future work. Note though that (a) many well-known XPath processing techniques, such as [2][17][18], have this property and (b) the number of duplicates produced is typically significant, and they can multiply after successive execution steps, thus significantly increasing the size of operator input sequences and hence operator cost. Due to this, operators that produce (and consume) sequences with duplicates are not expected to be beneficial in general.

3. XPA API IMPLEMENTATIONS

The XPath execution framework can work with any storage engine that implements the *XPA API*. We have developed five different versions of a native XML storage system, namely *RE-basic*, *RE-Path*, *PE-basic*, *PE-Path* and *Edge-based RE-Path*. In all versions, XML elements are stored in B-Trees with element *ID* being the key. The systems differ in the labelling scheme used, in the inclusion or not of a root-to-node path (*RTN-path*) index or in whether they keep a separate B-Tree per tag name.

3.1 Region Encoding Basic (RE-basic)

The *RE-basic* system uses *region encoding* (pre/post/level) for mapping element nesting and ordering, as in [2]. Moreover, each element holds its parent element's *pre* and tag name (*par* and *parTagName* respectively). The key used by the B-trees is the *pre* value. The implementation of the *Element* interface is a class with the following members: *pre*, which is a specialization of *Element.ID*, *post*, *par*, *parTagName*, *level* and members derived from the definition of the element in the logical data model: *tagname*, *text* and *attributes*. Methods that check structural relationships are implemented by applying appropriate comparisons among the *pre*, *post* and *par* values of the element with the counterparts of the element passed as argument, as proposed in [2]. The *getRTNPath()* method implementation constructs the *RTN-path* of the element by retrieving its ancestors one by one, following the *par* references up to the root.

The *PAMs* of the *AccessMethods* interface directly access the B-Tree structures. For example, the implementation of the *Descs()*

method performs a range lookup on the B-Tree corresponding to the given tag name, searching for elements with key greater than the *pre* of the input element. Once a hit occurs, the subsequent elements are also returned, following the linked list of the B-Tree leaves, until an element whose *post* rank is greater than or equal to the *post* rank of the input element is met. Ancestor elements are reached by going backwards via the parent reference (*par*).

Table 2. Cost-relevant variable definitions

c1	**The result of** *CostForDescLookup()*
c2	**The result of** *CostForNextDesc()*
c3	**The result of** *CostForAncLookup()*
c3'	**The result of** *CostForAncLookup(virtual=true)*
c4	**The result of** *CostForNextAnc()*
c4'	**The result of** *CostForNextAnc(virtual=true)*
c5	**The result of** *CostForParLookup()*
c5'	**The result of** *CostForParLookup(svirtual=true)*
c6	**The result of** *CostForChildLookup()*
c7	**The result of** *CostForNextChild()*
c8	**The result of** *CostForRTNPathRetrieval()*

Table 3. Constant definitions

T1	average cost for lookup in a B-Tree holding elements
T2	average cost for moving among leafs of a B-Tree holding elements
T3	cost for lookup in the Paths B-Tree

The cost models for the *PAMs* of the *RE-basic* interface rely heavily on the properties of B-Trees. In what follows, we use the variables in Table 2 and constants in Table 3. The cost functions appear in Table 4. A descendant lookup is essentially a B-Tree lookup, and therefore *c1* depends on the B-Tree height. We assign a constant *T1* to this cost, assuming that all B-Trees maintained have the same height. Once a descendant is reached, the cost for moving to the next descendant is expected to be much smaller than *T1* due to the linked list interconnecting B-Tree leaves. Assuming that *T2* is this cost on average, we assign *T2* to *c2*.

Table 4. Cost functions for RE/PE-Basic and RE/PE-Path

	RE-basic / PE-Basic	RE-Path	PE-Path
c1	T1		
c2	T2		
c3	AvgDepth(tagname(e))*T1		T3+T1
c3'	AvgDepth(tagname(e))*T1		T3
c4	0		T1
c4'	0		0
c5	T1		T3+T1
c5'	0		T3
c6	T1 + 0.5*(Card(p(e)//tagname) / Card((e)/tagname))*T2		
c7	(Card(p(e)//tagname) / Card((e)/tagname))*T2		
c8	AvgDepth(tagname(e))*T1	T3	

In retrieving document-ordered ancestors of a given element, the first ancestor to return must be the most distant one. So we must access all the elements along the *RTN-path*, following the parent references (*par*). Consequently, the cost *c3* for ancestor lookup is given by *AvgDepth(tagname(e))*T1*. (Note that this is an overestimation when *cousinEl* is provided, since backward navigation stops upon reach of an element that is a *cousinEl* ancestor.) Retrieving ancestors other than the most distant one (*c4*) incurs zero additional cost, since these have already been pre-fetched during the retrieval of the first ancestor in document order. Children are retrieved by performing a range lookup on the appropriate B-Tree searching for descendants. Once the first descendant is found, subsequent elements are read following the linked list of the B-Tree leaves, skipping elements that are not children. The average number of non-children descendants between two consecutive children is estimated by the cardinality of the path formed from the concatenation of the *RTN-path* of the input element, *p(e)*, with a descendant step of the target tag name *//tagname*, divided by the cardinality of the concatenation of *p(e)* with a child step of the target tag name, */tagname(op)*. Therefore, the cost for moving to the next child equals the cost for reading all these irrelevant descendants (*c7*), while the cost for retrieving the first child equals the cost for a descendant lookup augmented by the cost for retrieving half the number of these irrelevant descendants (*c6*). Finally, retrieving the *RTN-path* of an element involves as many B-Tree lookups (*T1*) as is the average depth of elements having the same tag name (*c8*). Constants *c3'*, *c4'*, *c5'* are not relevant yet and will be explained in Section 3.3.

3.2 Prefix Encoding Basic (PE-basic)

The *PE-basic* system uses the *dewey encoding scheme* [15] for mapping both element nesting and ordering and, therefore, *dewey* is the specialization of *Element.ID* and the key of the B-trees. Primitive access methods of the *AccessMethods* interface are again implemented by accessing directly the B-Tree structures, as in [8]. *PAMs* cost models for this driver are identical to those of the *RE-basic* driver. Hence, if our cost models are accurate, which we show in Section 6.4, operations on this storage engine will have analogous execution cost to those on *RE-basic*. For this reason, we do not discuss this system further.

3.3 RTN-paths (RE-Path, PE-Path)

The main difference introduced by the *RE-Path* and *PE-Path* systems is that we store the distinct *RTN-paths* of the XML tree at hand in a separate B-Tree. These paths are assigned a unique number (*pathId*) which is the key for storing them. This B-Tree is expected to be relatively small, since the total number of distinct *RTN-paths* found in an XML document is usually very small compared to its size (less than 514 in all experiments of Section 6). Stored elements are assigned the *pathId* of their *RTN-path*, but apart from this feature, *RE-Path* and *PE-Path* systems are identical to the *RE-basic* and *PE-basic* systems, respectively. Cost functions of *RE-Path* and *PE-Path* also appear in Table 4. Method *Element.getRTNPath()* is more efficient than in the previous drivers, because a simple lookup on the Paths B-Tree structure is required. Therefore, for both *RE-Path* and *PE-Path* the cost for *RTN-path* retrieval (*c8*) is *T3*. Due to the relatively small number of *RTN-paths*, *T3* is usually much smaller than *T1*.

Dewey and RTN-paths for fast back navigation. Specifically for the *PE-Path* driver, we can also take advantage of dewey properties and *RTN-paths* to implement the *Ancs()* and *Parent()* methods of the *AccessMethods* interface optimally. Given the dewey position and the *RTN-path* of an element, it is trivial to compute the dewey positions and the *RTN-paths* of all its ancestor elements, including the parental one using simple string manipulation. We employ this in the implementation of the *Ancs()* and *Parent()* methods when the *virtual* argument is set to *true* (*c3'* and *c5'*). Therefore, the cost for ancestor and parent lookups equals the cost of *RTN-path* retrieval (*T3*), while shifting to the next ancestor (*c4'*) incurs no cost. If *virtual* is set to *false* or omitted (*c3*, *c4* and *c5*), an extra B-Tree lookup (*T1*) is required.

3.4 Edge-based RE-Path

The *Edge-based RE-Path* storage system is similar to *RE-path* but stores all elements in a single B-Tree structure (Edge-based) as in [2]. On this driver, navigating *PAMs* do *not* fetch elements per tag

name. Therefore, when the *tag* parameter is specified, the implementations of these *PAMs* output those elements surviving a subsequent tag name filtering step. Details about the *Edge-based RE-Path* are omitted due to lack of space.

4. LOOKUP OPERATORS

The basic processing strategy of the *LookUp (LU)* family is to search a minimum window (*window*) of elements for each element in the context sequence (*contextSeq*), similar to indexed nested loops join algorithms. This window is fetched from the underlying storage as a result of calling the *PAM* corresponding to the XPath axis under evaluation. Direct implementation of this strategy does not guarantee that the resulting sequence will be *DODF*. Document order and duplicate elimination can be preserved without the need for a blocking pre-fetching phase. Duplicate elements appear when windows overlap with each other. Depending on the axis, we can use one of two strategies to avoid this. The first avoids fetching overlapping windows right from the beginning, such as direct pruning [2] used in d^{LU} or the techniques used in fp^{LUp} and a^{LU}. The second, used in bp^{LUp}, is to detect overlapping windows and make use of an intermediate structure to temporarily keep result elements.

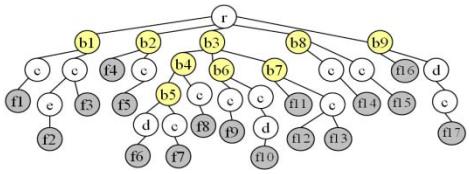


Figure 3. A sample XML document

In the remainder of the Section, the algorithms corresponding to implementations of the *LU* physical operators are presented. Common variables used throughout algorithms 1 to 4 are summarized in Table 5. Figure 3 illustrates an XML document used as a running example throughout the rest of the paper.

Table 5. Common variables in LU operators

contSeq: Sequence	The sequence given as input to the operator.
contEl: Element	A pointer to the last element read from contSeq
window: Sequence	The current window of elements
windEl: Element	A pointer to the last element read from window

4.1 Lookup descendant and forward path

The *LookUp* fp_p physical operator ($fp^{LU}{}_p$), illustrated in algorithm 1, is based on the following technique: given a context node *n* of level *l,* we can reach descendants under a specific relative path *p* by retrieving all descendants of *n* and checking whether the suffix of their *RTN-path* that starts from step *l* matches the regular expression that derives directly from *p*. The task of regular expression matching is performed by method *regExprFilter(rtn, path, level)* where *rtn* is a *RTN-path*, *path* is the relative path from which we draw the regular expression we match *rtn* against, and *level* is the point in *rtn* where regular expression matching begins. Method *regExprFilter* translates *path* into a regular expression as described in [8]. The operator is very efficient when run upon a driver that provides cheap *RTN-path* retrieval.

Example 2: If *rtn=/r/b/b/a/b/c/d/e/f/g, path=/c//f/g* and *level=5,* then *regExprFilter(/r/b/b/a/b/c/d/e/f/g, /c//f/g, 5) = true,* since regular expression '*/c/(.+/)?f/g*', deriving from *path*, matches suffix */c/d/e/f/g* of *rtn.* □

For the purposes of duplicate avoidance and document order preservation, the algorithm uses a novel technique we call *buffered-leaping*. This technique identifies nested context elements and guarantees that no windows are fetched for these elements by temporarily keeping them in a list named *chain*. Particularly, when a context element that is not nested to any previously read one is read - called *root ancestor* (*rootAnc*) - a descendant window is initialized (line 16). As long as following context elements are descendants of the last read *rootAnc*, these are kept in *chain* (lines 18-20). We then check the structural relationship of each element fetched by the window with *rootAnc* and those kept in *chain*, by calling the *regExprFilter()* method (line 9). If, right after a root ancestor, the following context element is not a descendant, *chain* is emptied. The size of *chain* at any time is usually very small and upper bounded by the depth of the XML document.

Algorithm 1. Open() and next() implementations of $fp^{LU}{}_p$
p: the path of the fp operator
rootAnc: a context element not nested into any other context element
chain: list of context elements nested into rootAnc

```
1  open() {
2    contSeq.open();
3    contEl = contSeq.next(); }
5  next() {
6    while (true) {
7      if window has more elements {
8        windEl = window.next();
9        if ∃ e in chain∪rootAnc such as: (windEl.isDescOf(e) and
10           regExprFilter(windEl.getRTNPath(), p, e.getLevel()-1))
11         return windEl;
12     } else {
13       chain = ∅;
14       if contEl is null return null;
15       rootAnc = contEl;
16       window =XPAPI.Descs(rootAnc, lastTag(p));
17       contEl = contSeq.next();
18       while (contEl is not null and contEl.isDescOf(rootAnc)){
19         chain.add(contEl);
20         contEl = contSeq.next();
21 } } }
```

Note that *pruning* [2] as suggested for the descendant Staircase join, which simply skips and ignores nested context elements, cannot be applied for the holistic evaluation of a forward path *p*: for a given context element *e*, certain descendant elements may not be reachable by navigating through *p*, but could be reachable starting the navigation from a following context element that is descendant of *e*. For example, element f_8 of Figure 3 is descendant of b_3, but is not reachable via relative path */c//f*. f_8 is also descendant of b_4 and also reachable from b_4 by navigating through */c//f*. b_4 is a descendant of b_3 and, therefore, by just skipping it, as in pruning, we would never take f_8 among the results.

Example 3: We will track the operation of $fp^{LU}{}_{/c//f}$ given its context sequence is {b1, b2, b3, b5, b7, b9}, using the XML document of Figure 3. Table 6 illustrates all the details of running this algorithm. Variable values are those set by the algorithm after the end of the corresponding iteration. At first *b1* is read memorized as *rootAnc* (line 15) and its descendant *window* is initialized (line 16). Since the next context element (*contEl=b2),* is not a descendant of the one last read (*b1*), the following iterations of the outer while-loop of line 6 read one-by-one elements from *window* and output them as long as method *regExprFilter()* returns *true* (line 9) until *window* is exhausted. Therefore, *f1* is output by the first *next()* call, *f2* is fetched but rejected and *f3* is fetched and output by the secfond *next()* call. Since *b1's window* is exhausted, *rootAnc* now points to *b2*, the window of *b2* is initialized. *f5* is fetched by that window and is output by the 3rd *next()* call. The window of *b2* is exhausted, *rootAnc* points to *b3* and the window of *b3*'s *window* is initialized. Until this point *chain* was empty. This time *b5* is descendant of *b3* and, therefore, *b5* as well as the

subsequent context element *b7* are kept in *chain*. The following iterations of the outer while-loop of line 6, since *chain*≠∅ (line 12), read each element (*windEl*) from the *window* of *rootAnc* and outputs it as long as there exists an element *e* in *chain*∪*rootAnc* such as *windEl* is reachable from *e* via *p* (line 16). Therefore, first *f6* is fetched from the window of *b3* window and is rejected, then *f7* is fetched and output by the forth *next()* call (it is reachable from *b5* via */c/f*), *f8*, *f9*, *f10* and *f11* are rejected, and, finally *f12* and *f13* are output by the fifth and sixth *next()* call, respectively, being reachable from *b7* via */c/f*. During the 7th *next()* call, *chain* gets empty, the window of *b9* is initialized and *f16* is output. □

Table 6. Sample run of $fp^{LU}_{/c//f}$ operator

		rootAnc	contEl	chain	windEl	output
1	next()	b1	b2	{}	f1	f1
2	next()				f2	-
					f3	f3
3	next()	b2	b3		f4	-
					f5	f5
4	next()	b3	b5	{b5}	f6	-
			b7	{b5, b7}		
			b9			
					f7	f7
5	next()				f8	-
					f9	-
					f10	-
					f11	-
					f12	f12
6	next()				f13	f13
7	next()	b9	null	{}	f16	-
					f17	f17

processing nested context elements

Changing line 16 to call the *Children() PAM* and the condition of line 9 to '*if ∃ e in chain∪rootAnc such as windEl.isChildOf(e)*', gives us the pseudocode for the LU c_a operator (c^{LU}_a).

Algorithm 2. Open() and next() implementations of d^{LU}_a

```
1  open() {
2    contSeq.open();
3    contEl = contSeq.next();
4    rootAnc = contEl; }
5  next() {
6    while (true) {
7      if (window has more elements) return window.next();
8      else {
9        if(rootAnc is null) return null;
10       window = XPAPI.Descs(rootAnc, a);
11       contEl = contSeq.next();
12       while (contEl is not null and contEl.isDecsOf(rootAnc)) {
13         contEl = contSeq.next();
14         rootAnc = contEl; }
15 }}}
```

Algorithm 2 illustrates the implementation of the *open()* and *next()* methods of the LU d_a operator (d^{LU}_a). d^{LU}_a adopts *pruning* [2], thus it skips and ignores context elements that are descendants of previously read ones (lines 12-14).

4.2 Lookup ancestor and backward path

While windows overlapping in forward LU operators occur only in the special case where nested context elements occur, in backward navigation windows overlapping is the usual case, due to the tree-structure of XML. The lookup bp_p physical operator (bp^{LU}_p), whose pseudocode is illustrated in Algorithm 3, calls the primitive access method *Ancs()* for each context element and *RTN-path* filtering afterwards. Assume a context element *n* and an ancestor *a* whose tag name is that of the last step of the backward path *p*. Element *a* is reachable from *n* via the backward path *p* iff the suffix of *n*'s *RTN-path* starting from the level of *a* matches the regular expression derived from the reverse of *p* (line 4) concatenated with the tag name of the context element *n*. The operator is very efficient when run upon a driver that provides cheap *RTN-path* retrieval and most efficient when the *Ancs* implementation is also cheap. The virtual argument of *Ancs()* is set to *true* (line 19), and therefore, for drivers such as *PE-Path*, the *Ancs()* calls are extremely cheap (cost components c3' and c4' are smaller than c3 and c4, respectively). However, in this case, if the bp^{LU}_p is not allowed to output virtual elements, these must be reloaded just before they are output (line 13).

Algorithm 3. Open() and next() implementations of bp^{LU}_p

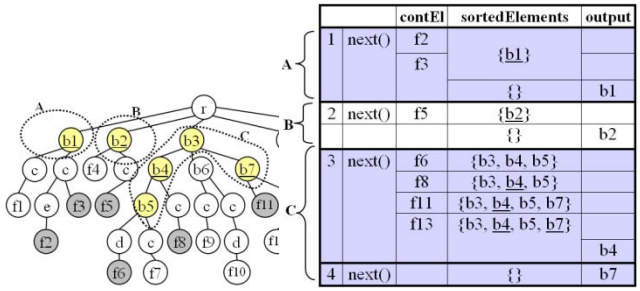
```
p: the path of the bp operator
sortedElems: the structure used for storing DODF elements
allowVirtual: whether it is ok for the op to return virtual elements
1  open() {
2    contSeq.open();
3    contEl = contSeq.next();
4    ⌐p = reverseOf(p);
5    ready = false; }
6  next() : Element {
7    while(true) {
8      if(ready) {
9        ret = nextMarked(sortedElems); //next marked element in doc order
10       if(ret is null) {ready = false; sortedElems.clear(); continue; }
11       else {
12         if not allowVirtual and ret is virtual
13           return XPAAPI.getElByNameAndID(ret.getName(), ret.getID())
14         else
15           return ret;
16       }
17     } else {
18       while(sortedElems.isEmpty() or contEl.isDescOf(sortedElems.first()) {
19         window = XPAAPI.Ancs(contEl, last(p), null, true); // virtual = true
20         while(window has more elements) {
21           windEl = window.next();
22           sortedElems.insert(windEl);
23           if(regExprFilter(contEl.getRTNPath(),⌐p, windEl.getLevel()-1)) {
24             mark(windEl); }
25         }
26         contEl = contSeq.next(); }
27       ready = true; continue; }
28 } }
```

Example 4: Consider $bp_{\wedge\wedge c\wedge b}$ applied on *f* elements. The inverse of path ^^c^b is ⌐*(^^c^b) = /c//f*. Let's also suppose that the current context element is *f7,* of the XML document of Figure 3, with *RTN(f7) = /r/b/b*/b/c/f. The Ancs() PAM with f7 and 'b' as arguments returns ancestors b3, b4 and b5 with RTN-paths '/r/b', '/r/b/b' and '/r/b/b/b', respectively (these can be virtual depending on the driver). For each ancestor we call function *regExprFilter* as follows: *b3: regExprFilter(/r/b/b/b/c/f, /c//f, 2) = false, b4: regExprFilter(/r/b/b/b/c/f, /c//f, 3) = false, b5: regExprFilter(/r/b/b/b/c/f, /c//f, 4) = true.* Therefore, only *b5* is returned by a call on *next()* of $bp_{\wedge\wedge c\wedge b}$. □

The algorithm uses the *sortedElems* structure so as to avoid duplicates. Overlapping windows occur when the *Ancs()* method is called for a context element that is a descendant of the most distant element in the *sortedElems* list (condition in line 18). If this is not the case, then following windows will not overlap with the ones fetched so far, signified by setting *ready* to *true* (line 27). Otherwise, fetching windows must go on (lines 18-26). For each window fetched, its elements are inserted into *sortedElems* and because elements may be found to conform to the given path later than the time they were first fetched, the algorithm needs to mark the ones that are to be returned (line 24). Note that the *Ancestor Paths Separation* technique [2] for duplicate avoidance in the ancestor Staircase join, that "separates the paths in the document tree and evaluates the ancestor step for each context node in its own partition", cannot be applied for the holistic evaluation of a backward path *p*. This is because a context element may have an ancestor reached by traversing via the *p* that is common ancestor with a previously read context element but never been output while processing that context element. For example, consider the

$bp^{LU}{}_{\wedge c\wedge\wedge b}$ operator. Assume that the operator processed context element *f6* of Figure 3. *b3*, being an ancestor of *f6*, yet not reachable from *f6* via backward path ^c^^b, would not be output. When context element *f13* is processed, *b3* should now be output.

Example 5: Let's suppose that {*f2, f3, f5, f6, f8, f11, f13*} is the context sequence of the $bp^{LU}{}_{\wedge\wedge c\wedge b}$ operator. The steps of the algorithm are shown in Figure 4. Elements of the *sortedElems* structure that are marked are underlined. The window of *f2* includes only *b1*, which is put in *sortedElems* (group *A* of Figure 4(a)) and marked since it is reachable from *f2* via backward path ^^c^b. The window of *f3* also includes only *b1*. Since the next context element (*f5*) is not a descendant of *b1*, all marked elements of *sortedElems* are output (that is only *b1*) and *sortedElems* is empty. The window of *f5* consists only of *b2*, and, since the following context element is not a descendant of *b2*, *b2* is the only element of *sortedElems* (group *B* in Figure 4(a)), it is also marked and, thus, it is output. *sortedElems* is empty again. Subsequently, *f6* is read. Its window consists of *b3*, *b4* and *b5* which are kept in *sortedElems*. None of them are marked to this point since they are not reachable from *f6* via ^^c^b. The next context element is *f8*, a descendant of *b3*. Its window consists of *b3*, *b4* which are already in *sortedElem* and *b4* is marked (reachable from *f8* via ^^c^b). Then *f11* is read, also being a descendant of *b3,* and *b7* is added in *sortedElems*. Finally, *f13* is also a descendant of *b3* whose ancestors, *b3* and *b7*, are already in *sortedElems,* but this time *b7* is marked. Since no more context elements exist, the marked elements of *sortedElems* (group *C* in Figure 4(a)), namely *b4* and *b7*, are output. □



		contEl	sortedElements	output
1	next()	f2	{b1}	
		f3		
			{}	b1
2	next()	f5	{b2}	
			{}	b2
3	next()	f6	{b3, b4, b5}	
		f8	{b3, b4, b5}	
		f11	{b3, b4, b5, b7}	
		f13	{b3, b4, b5, b7}	
				b4
4	next()		{}	b7

(a) (b)

Figure 4. Sample run of $bp^{LU}{}_{\wedge\wedge c\wedge b}$ operator

Algorithm 3 can be used for the LU parent operator ($p^{LU}{}_a$) as well: we `feed' the *window* with the *Parent() PAM* and change line 18 to *if(elem.isParentOf(contEl))*

Algorithm 4. Open() and next() implementations of $a^{LU}{}_a$
a: the tag name of the a operator
allowVirtual: whether it is ok for the op to return virtual elements

```
1  open() {
2    contSeq.open();
3    contEl = contSeq.next();
4    lastContEl = null; }
5  next() : Element {
6    while(true) {
7      ret = window.next();
8      if(ret is not null) {
9        if not allowVirtual and ret is virtual
10         return XPAAPI.getElByNameAndID(ret.getName(), ret.getID());
11       else return ret; }
12     else {
13       if(contEl is null) return null;
14       window = XPAAPI.Ancs(contEl, a, lastContEl, true);
15       lastContEl = contEl; contEl = contSeq.next(); }
16 }}
```

Implementation of the lookup a_a physical operator (a^{LU}), illustrated in algorithm 4, is based on an observation that derives from the *Ancestor Paths Separation* technique [2], according to which, for each element read from the context sequence, we need not look for elements that are common ancestors of the current context element and the previous one, memorized in *lastContEl*. Feeding the *Ancs() PAM* with *lastContEl* (line 14) as the *cousinEl* argument prevents duplicate production, at no extra cost (Section 3). However, the algorithm differs from the ancestor staircase join in that it fetches a *narrower* ancestor window for each context element. Moreover, as in $bp^{LU}{}_p$, a fast implementation of the *Ancs() PAM* such that of the *PE-path* system, as described in Section 3.3, makes $a^{LU}{}_a$ much faster on such storage engines, as shown in Section 6.1.

4.3 Lookup cousin operator

To implement the novel *cousin* logical operator [10] $cs^{LUp}{}_{p1,\ p2}$ we combine two of the previously described operators. First a $bp^{LU}{}_{p1,}$,a $p^{LU}{}_{last(p1)}$, or an $a^{LU}{}_{last(p1)}$ operator (depending on *p1*) takes as input sequence the context sequence of the $cs^{LUp}{}_{p1,\ p2}$ operator, while feeding itself as input to the second operator: a $fp^{LU}{}_{p2,}$,a $c^{LU}{}_{last(p2)}$, or an $d^{LU}{}_{last(p2)}$ operator (depending on *p2*). Note that since intermediate results of the backward internal operator are not to be output, we use the *allowVirtual =true* version of the respective operator, to achieve excellent performance, depending on the capabilities of the storage engine (eg on *PE-path*).

4.4 Cost Modeling

For each physical operator, the cost model needs to be provided (via implementation of its *Descriptor*). To define the cost models, we use the variables, functions and constants shown in Tables 2, 3 andTable 7. Table 8 summarizes the cost formulas for the LU operators.

Table 7. Variables and functions used in cost estimation

op	**The logical operator instance associated with the current physical descriptor instance**
cop	**context operator (the input operator of** *op* **)**
OUT(o)	**The cardinality estimation for operator** *o*
tagname(o)	**The tag name of the elements that** *o* **outputs**
cp	**The most precise path known for elements returned by** *cop*
last(p)	**The tag name of the last step of path** *p*
T4	*c8* + **The cost associated with** *regExprFilter()*
allowVirtual	**The** *allowVirtual* **member (0 or 1) variable of the respective backward** *op*

Table 8. Cost Formulas for LU operators

fp^{LU}	*Cost=OUT(cop)*(c1 + Occ(cp, //tagname(op))*(c2+T4))*
c^{LU}	*Cost=OUT(cop)*(c6 + Occ(cp, /tagname(op))*c7)*
d^{LU}	*Cost=OUT(cop) * (c1 + Occ(cp, //tagname(op))*c2)*
bp^{LU}	*If (c4'<c4) Cost=OUT(cop)*(c3' + Occ(cp, ^^tagname(op)) * *(c4'+T4)) +(1-allowVirtual)* OUT(op)*c1 else Cost=OUT(cop)*(c3 + Occ(cp, ^^tagname(op)) * (c4+T4))*
p^{LU}	*If (c5'<c5) Cost=OUT(cop)*c5' +(1-allowVirtual)* OUT(op)*c1 else Cost=OUT(cop)*c5*
a^{LU}	*If (c3'<c3) Cost=OUT(cop)*(c3' + Occ(cp, ^^tagname(op))*c4') + + (1-allowVirtual)* OUT(op)*c1 else Cost=OUT(cop)*(c3 + Occ(cp, ^^tagname(op))*c4)*
cs^{LU}	*Cost(internalBackOp$^{LU,\ allowVirtual=1}$) + Cost(internalForwardOp^{LU})*

The worst case scenario for the fp^{LU}, c^{LU}, d^{LU} , bp^{LU}, p^{LU}, a^{LU}, operators is that they fetch as many windows as there are context elements. Fetching the first element from each such window incurs a cost of *c1* in the case of d^{LU} and fp^{LU}, *c6* in the case of c^{LU}, *c3* n the case of a^{LU} and bp^{LU} and *c5* in the case of p^{LU}. Traversing these windows incurs a cost of *c2* for d^{LU} and fp^{LU}, *c7* for c^{LU}, *c4* for a^{LU} and bp^{LU} and 0 for p^{LU} (singleton window) for each window element. The total number of window elements

these operators is given by *Occ(cp, //tagname(op))* for d^{LU}, fp^{LU} and c^{LU} and *Occ(cp, ^^tagname(op))* for a^{LU} and bp^{LU}.

For the *RTN-path*-based algorithms, processing window elements involves the cost implied by *regExprFilter()*. For backward operators, if the *Ancs() PAM* is capable of returning virtual elements (*c3'*<*c3*, *c5'*<*c5*, as in the *PE-Path* driver), *c3* and *c5* are replaced by *c3'* and *c5'*, respectively. In this case only, if the *allowVirtual* member variable of the backward operator is 0 (false), meaning that the operator is not encapsulated in a *cs* operator, the cost for 'converting' virtual element to 'real' elements is added in the total cost. This cost is *OUT(op)* times the cost for element lookup (*c1*).

5. SORT-MERGE-BASED OPERATORS

The basic strategy of the SortMerge-based *(SM)* XPath operators is to traverse two *DODF* sequences of elements, *left* and *right*. Keeping track of the current elements on both sequences, we try to find matching pairs according to the appropriate navigation axis and condition. The right sequence always consists of all the elements of the requested tag name available in the database. The left sequence is the *context sequence* (as seen earlier), i.e., the operator's input sequence. Single-step SM algorithms are similar to other sort-merge-based structural joins such as the one proposed in [1]. The novelty of the SM-based family of operators derives from the multistep (fp, bp and cs) implementations. Table 9 shows the common variables used in the algorithms to follow.

Table 9. Variables used in algorithms 5 and 6

leftSeq: Sequence	the context sequence (contSeq)
rightSeq: Sequence	the sequence where solutions are sought
leftEl: Element	the current element on the left sequence
rightEl: Element	the current element on the right sequence
docRoot: Element	the root element of the xml document
afterAll: Element	a virtual element located after all nodes in the xml tree

Algorithm 5. Open() and next() implementations of $d^{SM}{}_a$

```
1  open() {
2    leftSeq.open();
3    rightSeq = XPAAPI.Descs(docRoot, a);
4    leftEl = leftSeq.next();
5    rightEl = rightSeq.next(); }
6
7  next() : Element {
8    Element ret; // the element to return, initially null
9    while(true) {
10     if(leftEl is null) return null;
11     moveRightAfterLeft();
12     if(rightEl is null) return null;
13     if(rightEl.isDescOf(leftEl) {
14       ret = rightEl;
15       rightEl = rightSeq.next();
16       return ret;
17     } else leftEl = leftSeq.next();
18 }}
19
20 moveRightAfterLeft() {
21   while(rightEl is not null and not rightEl.isAfter(leftEl)) {
22     rightEl = rightSeq.next(); }
```

For the SortMerge-based d_a algorithm ($d^{SM}{}_a$) (algorithm 5), once a matching pair is found (line 13) the current element on the right is returned. Note that a *null* left or right element (line 10) signifies the end of the respective sequence. If the control reaches line 13, the current right element is after (in document order) the current element on the left, which is guaranteed by the call to *moveRightAfterLeft* (line 10). So if *rightEl* is not a descendant of *leftEl*, then no element subsequently found on the right will be a descendant of this *leftEl* either, so it is safe to advance the left sequence (line 16). This fact along with the implementation of the *moveRightAfterLeft* method achieves fast skipping of irrelevant elements on the right sequence. Method *isAfter()* has usually a very cheap implementation - involving a single comparison between integers in the case of the *RE* driver (section 3.1) or a single lexicographical comparison between strings in the case of the *PE* driver (section 3.2). Note that only two elements must be kept in memory for correct processing (*leftEl* and *rightEl*).

Algorithm 6. Open() and next() implementations of $fp^{SM}{}_p$

```
member variables:
p: the path of the fp operator
rightBound: Element  //the right bound of the window
possibleMatches: list //a list of elements to seek bypassed matches
1  open() {
2    leftSeq.open();
3    rightSeq = XPAAPI.Descs(docRoot, lastTag(p));
4    if(slideWindow()) {
5      rightEl = rightSeq.next();
6      moveRightAfterLeft(); } }
7  next() : Element {
8    Element ret; // the element to return, initially null
9    while(true) {
10     if(rightEl is null) return null;
11     if(rightEl.isAfter(rightBound)) {
12       if(slideWindow()) moveRightAfterLeft();
13       else return null;
14     } else if(weHaveAMatch()) {
15         ret = rightEl;
16         rightEl = rightSeq.next();
17         return ret;
18     } else if(possibleMatches.isEmpty() and
19                                   not rightEl.isDescOf(leftEl)) {
20       if(slideWindow()) moveRightAfterLeft();
21       else return null;
22     } else rightEl = rightSeq.next();
23 }}
24 slideWindow(): Boolean {
25   if(leftEl is not null) possibleMatches.add(leftEl);
26     slide leftEl and rightBound to the next available values
27     from leftSeq, using afterAll when we reach the end of leftSeq
28   return leftEl.getID() != afterAll.getID();
29 }
30 weHaveAMatch(): Boolean {
31 if(rightEl.isDescOf(leftEl) and
32  regExprFilter(rightEl.getRTNPath(), path, leftEl.getLevel())
33   return true;
34   else return weHaveOlderMatch(); }
35 weHaveOlderMatch(): Boolean {
36   foreach(Element el in possibleMatches) {
37     if(rightEl.isDescOf(el)) {
38       if(regExprFilter(rightEl.getRTNPath(), path, el.getLevel()) {
39         return true; }
40     } else possibleMatches.remove(el);
41   } // foreach
42   return false; }
```

In the case of SortMerge-based *fp* (fp^{SM}), whose pseudocode is illustrated in algorithm 6, keeping only the current left and right elements in memory is not sufficient. In order to avoid missing possible matches, we have to keep all ancestors of the current right element in a list (*possibleMatches*), so as to check against them for path matching. For this purpose, we read elements from the left sequence two at a time, forming a window whose bounds are *leftEl* and *rightBound*. We always make sure that the current right element (*rightEl)* is located between these bounds (lines 12, 20). Function *slideWindow* (lines 24-29) adjusts the bounds to the next available pair of elements in *leftSeq* and returns *true/false* indicating success or failure. Each time the window slides one position forward, the previous *leftEl* is added to *possibleMatches* (line 25). Checking for matches (line 14) is a twofold task. We check for a match against the current element on the left (lines 30-31) and against all elements stored in *possibleMatches* (lines 34, 35-42). Note that this involves using both the *getRTNPath()* method of the *Element* interface and regular expression filtering, as described in Section 4.1. Since *rightEl* is at all times after *leftEl* and all elements in *possibleMatches*, we can remove all elements in *possibleMatches* that are not ancestors of the current right

element (line 40). This upper-bounds the size of *possibleMatches* to the maximum recursion level of elements of tag name *tagname(cop)*. Maximum performance is achieved by implementing *possibleMatches* as a linked list, so that removal of elements (line 38) can happen upon encountering the respective list node. Also, as in the case of alg. 5, fast skipping of irrelevant elements is achieved by use of method *moveRightAfterLeft()*.

Example 6: We will track the operation of $fp^{SM}_{/b//f}$ given its context sequence is {*b1, b2 b3, b4, b5*}, using the XML document of Figure 3. Table 10 illustrates all the details of running this algorithm. Inside the *open()* method, *slideWindow()* sets *leftEl* and *rightBound* to *b1* and *b2* respectively while adding nothing into *possibleMatches*. Still inside *open()*, *rightEl* is set to *f1* (recall that in our SM-based operators, *rightSeq* is set to the sequence of all elements of the requested tag name, that being *f* in our example). Entering the main loop, *rightEl* is not null (read: *rightSeq* is not over yet). Also, the *if* condition in line 11 evaluates to *false* because *f1* is not after *b2*. Thus our window remains unchanged and we move on to line 14. *f1* is indeed a descendant of *b1*, so we may check for a path matching situation. Obviously, *f1* does not satisfy the */b//f* path starting from *leftEl=b1* so we move on to line 18, where the condition fails because *rightEl=f1* is a descendant of *leftEl=b1*. Thus, *rightEl* is advanced to *f2*. The details of this iteration are essentially the same, with *rightEl* being set to *f3* and then to *f4* by yet another iteration. This time we are out of bounds, so *leftEl* and *rightBound* are shifted to *b2*, *b3* respectively, while *b1* is added to *possibleMatches*. *rightEl=f4* does not satisfy the path criterion starting from *leftEl=b2*, so it is checked against elements in *possibleMatches.* Checking for matches in this list reveals that *b1* is not an ancestor of *f4*, so it is removed (line 40). In virtually the same way, *rightEl* is advanced one element at a time, until we reach a situation where *possibleMatches={b3,b4}, leftEl=b5, rightBound=b6* and *rightEl* points to *f7*. While checking against *b5* fails, this time *f7* matches with an item in *possibleMatches*, because the path */b/c/f7* is consistent with */b//f* when starting from *b3*. Thus *rightEl* is shifted to *f8* (line 16) before *f7* is returned. □

The processing logic of the a^{SM}_{a} and the bp^{SM}_{p} is essentially the same as in the d^{SM}_{a} and fp^{SM}_{p} cases, respectively, with the roles of *left* and *right* somewhat reversed. c^{SM}_{a} p^{SM}_{a} derives directly from fp^{SM}_{p} and bp^{SM}_{p}. Pseudocodes are omitted due to lack of space. The $cs^{SM}_{p1,p2}$ operator is just a combination of a $bp^{LU}_{p1,}$, $p^{LU}_{last(p1)}$, or $a^{LU}_{last(p1)}$ operator with a $bp^{SM}_{p2,}$, $p^{SM}_{last(p2)}$, or a $a^{SM}_{last(p2)}$ operators. bp^{LU}_{p1} is used to take advantage of the possibility that we can move backwards faster than normal, as in *PE-path*.

Table 10. Sample run of $fp^{SM}_{/b//f}$ operator

		leftEl	rightBound	rightEl	possibleMatches	output
	open()	b1	b2	f1	{}	
1	next()			f2		
				f3		
				f4		
		b2	b3		{b1}	
				f5	{}	
				f6		
		b3	b4		{b2}	
		b4	b5		{b2,b3}	
		b5	b6		{b2,b3,b4}	
				f7	{b3,b4}	
				f8		f7
2	next()		afterAll	f9		f8

5.1 Cost Modeling

Regarding d^{SM} and a^{SM}, the first element in the right sequence must be fetched (performed in the respective *open* function). This implies the existence of *c1* in the cost formula. Subsequent repeated calls to *next* result in traversing the whole *rightSeq* in the worst case scenario, which incurs a *c2*Card(//tagname(op))* cost for traversing *rightSeq*. The same analysis holds for the c^{SM} and p^{SM} cases. As in the previous cases, the cost formula for fp^{SM} includes *c1* so as to fetch the first element from the right sequence. In the worst case scenario, the whole right sequence is traversed, adding the *c2*Card(//tagname(op))* factor to the formula. Contrary to the previous SM-based operators though, regular expression matching involves a non trivial cost which should be taken into account. For each *leftEl,* all of its descendants in the *rightSeq* are checked against the given path, which amount to *Card(cp//tagname(op))*.

Similarly to the previous case, the bp^{SM} operator also involves the cost for path matching in the formula. For each *rightEl*, we check against all its descendant elements found on the left sequence. The number of left elements that are descendants of a *tagname(op)* element as a percentage of the total elements of *tagname(cop)* is *perc=Occ(//tagname(op), //tagname(cop)) / Card(//tagname(cop))*. So for each element on the right there exist *descPerRight = perc*OUT(cop)* descendant elements in the left sequence, meaning a total of *descPerRight * Card(//tagname(op))* path-matching operations. The cost formula for the cs^{SM} operator can be easily derived from the cost models of fp^{SM} and bp^{LU}. Table 11 summarizes the cost formulas for the SM-based operators.

Table 11. Cost Formulas for SM-basd operators

d^{SM},c^{SM}, a^{SM},p^{SM}	Cost=c1+Card(//tagname(op))*c2
fp^{SM}	Cost=c1+Card(//tagname(op))*c2+OUT(cop)*Occ(cp,//tagname(op))*T4
bp^{SM}	Cost=c1+Card(//tagname(op))c2+descPerRight*Card(//tagname(op))*T4 where descPerRight = OUT(cop)* *(Occ(//tagname(op),//tagname(cop))/(Card(//tagname(cop)))

6. EXPERIMENTAL EVALUATION

Experiments were run on an Intel Core 2 Duo 2.67GHz PC with 2GB of RAM, running MS Windows XP SP3. The XML storage systems of Section 3 and their corresponding drivers, our Execution Framework and all the physical operators are implemented in Java (JDK 1.6). We used Berkeley DB Java Edition (version 3.3.62) as B-Tree implementations used in the XML storage systems. This prototype implementation of the framework and of the five storage systems is used for comparative experimental evaluation on the same easy-to-use infrastructure. Our storage systems were given 150MB of cache and every query was executed 2 times. We only report the second time, corresponding to warm cache usage. With a cold cache all the results were similar and are not presented due to lack of space. For performance comparisons with other techniques, we used PathStack (only forward paths), Staircase join and Twig^{2}Stack for which we have implemented operators and incorporated them in our framework. For Staircase join we implemented the $d^{staircase}$ and $a^{staircase}$ operators only, as in [2]. Note that Staircase over the *Edge-PE-Path* corresponds directly to the use in [2]. For the experimental evaluation of our cost models, constants *T1*, *T2* and *T3* described in Table 3 are estimated experimentally, separately for each storage system and dataset.

The first 10 queries of Table 12 are used in our experiments to directly evaluate the performance of the SM-based and LU physical operators for the *d* (q1, q2), *a* (q3, q4), *fp* (q5, q6), *bp* (q7, q8) and *cs* (q9, q10) logical operators. The input sequence for these operators consists of elements of specific tag names that are artificially filtered (at no extra cost) with a given *selectivity factor*. The selectivity factor is the fraction of the elements that survive

the artificial filter on the total number of elements of the specific tag. For example, when query q1 is run with context selectivity factor 0.1, the $d_{listitem}$ physical operator is given as input the sequence produced by retrieving 10% of the *parlist* elements randomly. Smaller filter selectivity means fewer elements in the context sequence. When execution times are reported for queries q1-q10, these include the execution of the context which is common whatever operator is used for evaluating the relative path. Queries are not subject to any rule-based transformation, as we aim to evaluate single operator performance.

Table 12. Query Set

	context	relative path
q1	//parlist	//listitem
q2	//item	//emph
q3	//to	^^item
q4	//emph	^^parlist
q5	//open_auction	//parlist//listitem//bold
q6	//item	//description//parlist//text//keyword
q7	//to	^^mailbox^^item
q8	//emph	^^listitem^^parlist^^listitem^^parlist^^item
q9	//to	^^mailbox^^item//listitem//bold//emph
q10	//emph	^^bold^^listitem^^parlist^^item//mailbox//date
q11	/site/open_auctions[.//bidder/personref]//reserve	
q12	//item[location]/description//keyword	
q13	//item[mailbox/mail/to and description//emph/keyword]//bold/emph	
q14	//regions/europe//item[//mail//to and mailbox//from]/description//parlist[//bold]	
q15	//regions/europe//description//parlist//listitem[//keyword/bold and //keyword/emph]//text	
q16	//item[//description/parlist/listitem and //mailbox/mail/from]//keyword/bold	
q17	/site/categories[//description/parlist//emph]//text[keyword//emph]/bold	
q18	//europe//parlist[//listitem/parlist and //text]//keyword[bold]/emph	
q19	/site/regions[europe//listitem/parlist and namerica//parlist//bold]//keyword/emph	

6.1 Performance Comparisons

We run queries q1-q10, for two context filter selectivity factors, 0.8 and 0.1, respectively, over the *RE-Path* and *PE-Path* drivers for the 560MB XMark dataset. Execution times are summarized in Figure 5 and Figure 6 for *RE-path* and *PE-Path*, respectively. Multi-step queries (eg q5-q10) have been executed in two ways; i) by using the path-based version of the respective operator (fp^{LU}, bp^{LU}, cs^{LU}, fp^{SM}, bp^{SM} or cs^{SM}, labeled as Lookup/SM) and ii) by using a series of operators (labeled LookUp/SM-naive). For example, for q7 we can either use operator $bp^{SM}_{//mailbox//item}$ or (SM-naïve) we could use a plan consisting of $a^{SM}_{mailbox}$ and a^{SM}_{item} with the first being input to the second. Similarly to Lookup/SM-naive, for evaluating a multi-step path according to Staircase we use a series of $d^{Staircase}/a^{Staircase}$ operators.

Notably, in all cases either a SM-based or a LU operator is the fastest. Performance comparison between two techniques s1 and s2 are expressed in precedence improvement as follows: $(t_{s1}-t_{s2})/t_{s1}$.When the context selectivity is 0.8 the SM-based algorithms are the fastest in the majority of the queries on both *RE-path* and *PE-path* drivers (Figure 5(a) and 6(a)). SM-based outperform Staircase (by up to 91% improvement for q8) and LU (up to 84% for q8) because the latter perform many window lookups and, thus, their total cost is higher than simply scanning all elements of the target tag name (as the SM-based algorithms do). SM-based outperforms PathStack (from 17% up to 82% improvement) because the former maintains fewer intermediate results. Besides, for queries on forward or backward paths, SM-based runs much faster due to holistic *RTN-path*-based evaluation.

Reducing the context selectivity to 0.1 makes the respective LU operators the fastest in 70% of the queries over *RE-path* (Figure 5 (b)), and in 90% of the queries over *PE-path* (Figure 6(b)). Both LU and Staircase operators do better than SM-based and PathStack because, when filter context selectivity is lower, the number of window lookups performed by the LU and Staircase operators is smaller enough to keep their total cost lower than scanning all elements of the target tag name. The dominance of LU over Staircase (ranging from 3.6% to 82% improvement) is due to a series of reasons. Firstly, the LU operators search in narrower windows. Particularly for multistep paths, holistic evaluation based on *RTN-path* filtering incorporating our efficient technique for avoiding duplicates, *buffered-leaping*, gives LU operators a significant advantage over Staircase (and PathStack). Over the *RE-Path* driver, using bp^{LU}, cs^{LU} and cs^{SM} is not the best option since the LU-naive counterparts run up to 75% faster (q7-q10 in Figure 5(a) and (b)). Therefore, if navigating backwards is expensive, as in *RE-path*, fetching more ancestors than those included in the final result in an effort to avoid duplicates (the technique used by the bp^{LU}, described in Section 4.2) is a suboptimal option (this is not the case for a^{LU} that fetches only ancestors included in the result sequence by feeding the *Ancs() PAM* with the *cousinEl* argument). On the contrary, over *PE-Path*, the cheap implementation of *Ancs() PAM* makes a^{LU}, bp^{LU} and, especially cs^{LU} very fast. Recall that over this driver, bp^{LU} fetches only *virtual* elements from *Ancs()* calls. If bp^{LU} is not encapsulated in a cs^{SM} or cs^{LU} operator (*allowVirtual=false*), only ancestors to be output are retrieved from the actual data.

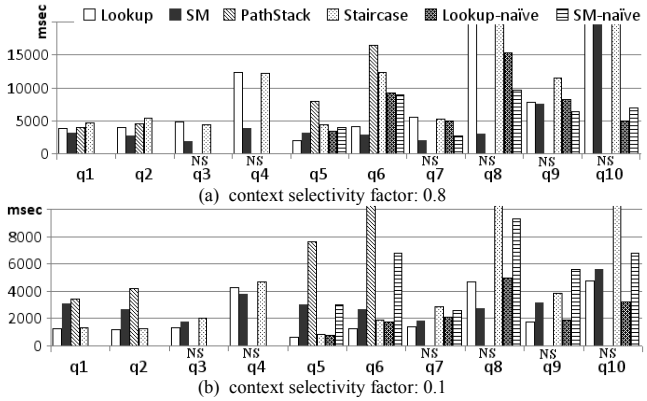


Figure 5. Query execution on the RE-path – 560MB XMark

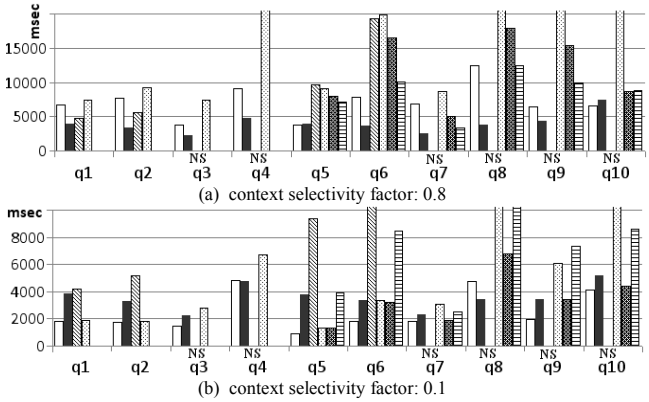


Figure 6. Query execution on the PE-path – 560MB XMark

On the *Edge-based RE-Path* driver, *PAMs DescInRange()* and *AncsInRange()* used in the $d^{staircase}$ and $a^{staircase}$ implementations respectively implement skipping exactly as in [2]. As shown in Figure 7, LU (even LU-naïve) outperform Staircase join in most of the queries (up to 42% improvement for q8) over this driver, as well. Note also that, when elements are not stored per tag name, as in the *Edge-based RE system*, the performance gain from using LU instead of LU-naive is much smaller because the *RTN-path* filtering selectivity is significantly decreased.

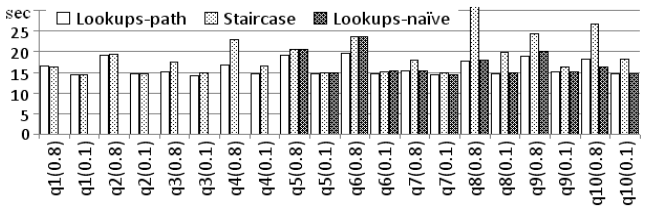


Figure 7. Edge-based RE-path -113MB XMark

6.2 Sensitivity Analysis

In an effort to explore the impact of the cardinality of the context sequence, we have run queries q1-q10 with the context selectivity varying from 0.8 down to 0.01. Figure 8 illustrates the execution times of LU, SM-based, PathStack and Staircase join algorithms on the *PE-Path* driver for q2, q4 and q6 on the 570MB XMark dataset. As expected, the sequential scanning of SM-based and PathStack makes their performance independent of the size of the input sequence, as opposed to LU and Staircase. Also, the higher the context selectivity, the better SM-based performs.

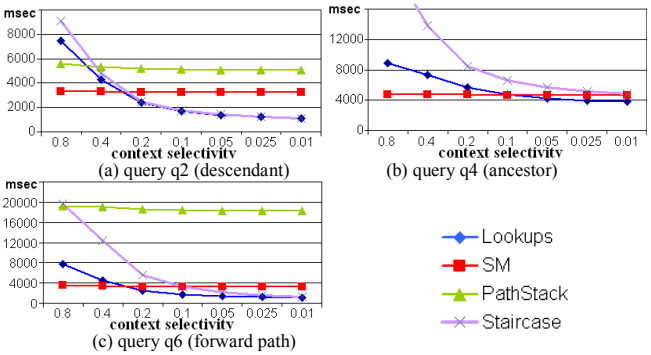


Figure 8. Exec. times as context selectivity decreases

In exploring the performance impact of increasing dataset sizes, we have run queries q4 and q6 on four XMark datasets on top of the *PE-path* driver, with the context selectivity set to 0.1. As can be seen in Figure 9, increasing the dataset size results in linear increase of the execution time for all operators tested, with the performance of PathStack degrading faster than the rest. For all other queries of Table 12 conclusions were similar and are not presented due to lack of space.

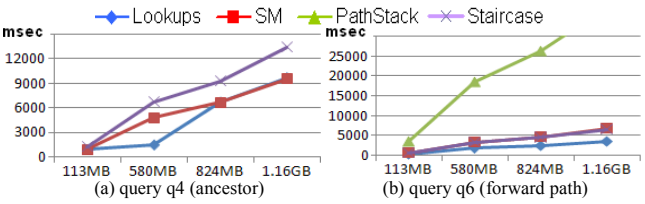


Figure 9. Exec. times as dataset size increases (cont. sel.=0.1)

6.3 Twig matching performance

We run the twig queries shown in Table 11 on the 824MB XMark dataset to compare the performance of Twig2Stack run on the *RE-path* driver (which apart from storing *RTN-paths* is exactly the storage system assumed in [14]) with our techniques. We compare its performance to that of plans comprising of best combinations of LU and SM operators (without applying any rewriting rule; we pick for each logical operator the cheapest estimated LU or SM operator). Predicates are evaluated using *filter* operators [10], whose Boolean operators are the counterparts of the LU operators.

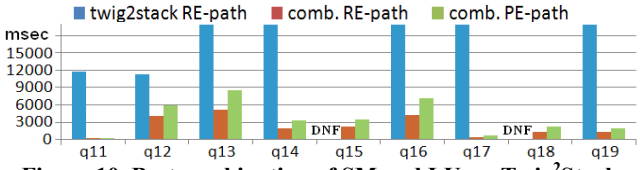


Figure 10. Best combination of SM and LU vs. Twig^{2}Stack

As illustrated in Figure 10, combining LU and SM-based algorithms brings major performance gain in evaluating twig queries (46%-99% improvement). Our algorithms not only outperform Twig^{2}Stack on *RE-path*, but also on *PE-path*, which is inherently slower when it comes to forward navigation. The execution time of Twig^{2}Stack is not reported in two cases as execution resulted in consuming all available memory.

6.4 Cost models evaluation

First, we run a total of 55 queries (q1-q10 of Table 12 for various context selectivity factors and database sizes), for which we compare execution times and cost estimations for both LU and SM-based operators (on the *PE-path* driver). If the operator with the lowest cost estimation is the fastest, then the estimation for that query is considered successful. *For 49 of these queries, a total of 89%, the estimation was successful.* Figure 11 illustrates the execution times and cost estimations (left- and right-side graphs) for queries q5 (a), q7(b) and q9(c), as context selectivity decreases. Cost estimation lines follow the same behavior as execution times (the same holds for graphs of the remaining queries of Table 12, omitted due to lack of space).

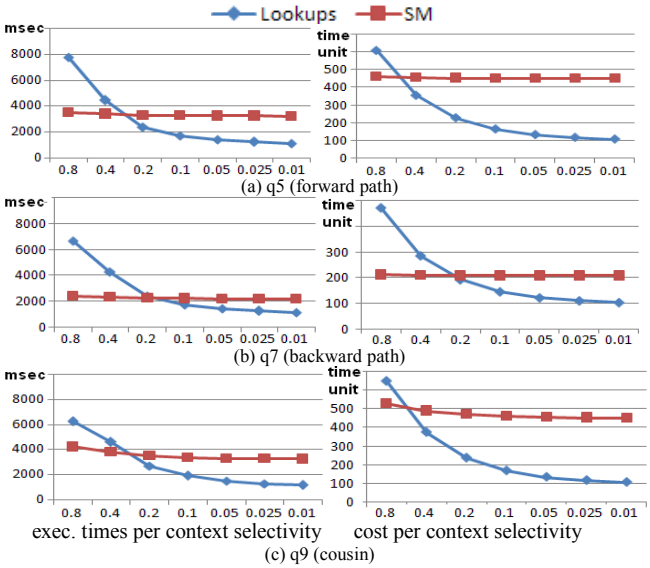


Figure 11. Exec. times and cost estimations for LU and SM

7. RELATED WORK

There is a large number of research works on XPath processing techniques and storage engines, including XPath processing over XML data shredded on relational systems [4][8] and native storage systems where XML documents are stored into disk pages preserving XML hierarchy [6][16]. Many algorithms for particular operations have been proposed, including coarse-grained operations such as twig matches, techniques based on indices on XML data [5], based on structural joins [1][2][5], exploiting novel structural encoding schemes [19], as well as holistic path and twig processing techniques [7][14]. These techniques show promise in

particular situations, but usually are tightly intertwined with specific storage engines, XML encodings and auxiliary data structures. There has been very little work on evaluating techniques on "standardized" storage engines that provide a fixed (but extensible) set of access methods. In [20] the authors defined a formalism for describing the *physical representation* of XML fragments, called XML access modules (XAMs). The optimizer answers queries using properly the available XAMs. Our XPA API provides with the means for developing *physical operators*, their *cost models* and, as a result, *query optimizers* (such as the one presented in [10]) that are *completely agnostic* to the underlying physical storage model. Regarding XPath processing techniques, existing work does not address efficient backward navigation, non-blocking *DODF* is not sufficiently explored, and many techniques use large intermediate results. Using more effective techniques, such as the ones presented, we achieve considerably better performance on a variety of storage engines. The work presented in [18] and [2] on duplicate avoidance is similar to how descendant and ancestor LU operators handle the task. However, our work is the first that suggests efficient and non-blocking techniques for avoiding duplicates during the holistic evaluation of forward or backward paths. The work presented in [17] detects whether explicit sorting could be completely avoided. However, if duplicates are not produced, our techniques have no impact on performance.

There is little work so far on cost estimation for XPath plans or operators. In IBM DB2 [16], an XQuery is translated into a tree consisting of operators in relational algebra extended with three XML-specific operators, and is optimized by the relational optimizer; the XML navigating operator (XSCAN) is very coarse and its cost models are not formally presented. The work presented in [13] deals with a single holistic operator, XNAV, tightly integrated with the storage engine. This is a considerably different task than costing finer grained operators and access methods that interoperate, as in this work. The work on cardinality and selectivity estimation and statistics (e.g. [11][12]) is orthogonal to our work and can be directly incorporated in our framework.

8. CONCLUSIONS

We present two novel families of algorithms for all the major XPath "operations", including forward and backward navigation as well as the novel *cousin* operator [10], and demonstrate experimentally their performance advantages compared to existing techniques. Performance benefits are derived by careful consideration of XPath semantics and the minimization of redundant work when scanning or processing element sequences. An important observation is that, compared to existing techniques that are (explicitly or implicitly) optimized for specific XML encodings and auxiliary data structures, our techniques are more agnostic, and can be useful to a cost-based optimizer in a variety of query settings. We have also presented a comprehensive framework for XPath execution that includes physical operator implementations along with cost models, as well as the necessary infrastructure for their easy deployment. The framework can be effectively used with a variety of different storage engines. Finally, we contribute to the principled development of XML processing engines by providing cost models for our operators and experimental evidence of their accuracy.

Results presented here provide strong evidence of the performance benefits of our framework in general and the LU and SM operators in particular. We plan to develop optimized implementations of the entire framework (e.g. developed in C++) and of the storage systems of Section 3 (e.g. using the C++ version of Berkeley DB as B-Tree implementations) and evaluate its performance against existing state-of-the-art XML DB systems. An XPA driver for a storage system that preserves directly the tree structure of XML (e.g. [18]) is also under development. Finally, we plan to continue our work towards a full XQuery processing infrastructure.

9. REFERENCES

[1] S.S. Al-Khalifa, H. V. Jagadish et al: Structural Joins: A Primitive for Efficient XML Query Pattern Matching. ICDE 2002

[2] T. Grust, M. van Keulen, J. Teubner: Staircase Join: Teach a Relational DBMS to Watch its (Axis) Steps. VLDB 2003

[3] J. Lu, T. W. Ling, C. Y. Chan, T. Chen: From Region Encoding To Extended Dewey: On Efficient Processing of XML Twig Pattern Matching. VLDB 2005

[4] M.Yoshikawa, T. Amagasa et al: XRel: a path-based approach to storage and retrieval of XML documents using relational databases ACM TOIT, Vol. 1, No. 1, 2001.

[5] S.-Y. Chien, Z. Vagena, et al.: Efficient Structural Joins on Indexed XML Documents.VLDB 2002: 263-274

[6] M. Brantner, C.-C. Kanne, et al: Full-fledged Algebraic XPath Processingin Natix. Proc. ICDE: 705–716 (2005)

[7] Nicolas Bruno, Nick Koudas, Divesh Srivastava: Holistic twig joins: optimal XML pattern matching. SIGMOD 2002

[8] H. Georgiadis, V. Vassalos: Improving the Efficiency of XPath Execution on Relational Systems. EDBT 2006.

[9] Y. Wu, S. Paparizos, H. V. Jagadish: Querying XML in Timber. IEEE Data Eng. Bull. 31(4): 15-24 (2008)

[10] H. Georgiadis, M. Charalambides, V. Vassalos, Cost Based Plan Selection for XPath. SIGMOD 2009

[11] N. Polyzotis, M. N. Garofalakis, Y. E. Ioannidis: Selectivity Estimation for XML Twigs. ICDE 2004: 264-275

[12] J.Filho, T. Härder: Statistics for Cost-Based XML Query Optimization. Grundlagen von Datenbanken 2006

[13] Z. Zhang et al.: Statistical Learning Techniques for Costing XML Queries. VLDB 2005

[14] S. Chen, H.G. Li, et al: Twig2Stack: Bottom-up Processing of Generalized-Tree-Pattern Queries over XML Documents. VLDB 2006

[15] I. Tatarinov, S. Viglas, et al: Storing and querying ordered XML using a relational database system. SIGMOD 2002

[16] A. Balmin, T. Eliaz, et al: Cost-based optimization in DB2 XML. IBM Systems Journal 45(2): (2006)

[17] M. F. Fernández, J. Hidders, P. Michiels, et al: Optimizing Sorting and Duplicate Elimination in XQuery Path Expressions. DEXA 2005: 554-563

[18] S. Helmer, C.-C. Kanne et al: Optimized Translation of XPath into Algebraic Expressions Parameterized by Programs Containing Navigational Primitives. WISE 2002

[19] Y. Chen, S. B. Davidson, Y. Zheng: A bi-labeling based XPath processing system. Inf. Syst. 35(2): 170-185 (2010)

[20] A. Arion, V. Benzaken, I. Manolescu: XML Access Modules: Towards Physical Data Independence in XML Databases. XIME-P 2005