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 causing 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 XPath 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)
• 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 Twig2Stack [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 XPathAlgebra. 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](image)

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.

![Figure 2. The XPA API Interfaces](image)

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. DescsInRange(elem1, elem2, tag) returns elements of tag name tag, being descendents 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 \( \text{CostForNextDesc}(e) \). 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. \( \text{Card}(\text{path}) \) returns the cardinality estimation for a given non-predicated absolute forward path (abs-fp) path. \( \text{Sel}(\text{basepath},\text{path}) \) 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. \( \text{Occ}(\text{basepath},\text{path}) \) 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, \( \text{DistValues}\{\text{tag, attributeText,Node}\} \) 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 \( c_e \) 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. \( f_p \) 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 \( S \), \( f_p \) performs backwards navigation via the relative backward path \( p \). In what follows, axes \( /\text{parent} :: /\text{ancestor} :: \) are abbreviated to \( ^ \) and \( ^{\text{..}} \), respectively. The novel cousin operator \( c_{s,p} \) 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 first 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 \( \text{BoolExpr} \), 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

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( c_e )</td>
<td>child</td>
</tr>
<tr>
<td>( d_a )</td>
<td>descendant</td>
</tr>
<tr>
<td>( f_p )</td>
<td>forward path</td>
</tr>
<tr>
<td>( f_p )</td>
<td>backward path</td>
</tr>
<tr>
<td>( f_p )</td>
<td>boolean forward path</td>
</tr>
<tr>
<td>( f_p )</td>
<td>boolean backward path</td>
</tr>
<tr>
<td>( f_p )</td>
<td>boolean value filter</td>
</tr>
</tbody>
</table>

**Example 1:** The XPath \( /\text{e} \) returns all elements with tagname ‘\text{e}’.

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 ElementID, 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 Desct()}
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

| $c_1$ | The result of $CostForDescLookup(p, e)$ |
| $c_2$ | The result of $CostForNextDesc(p, e)$ |
| $c_3$ | The result of $CostForAncestorLookup(p, e)$ |
| $c_3'$ | The result of $CostForAncestorLookup(virtual=true, p, e)$ |
| $c_4$ | The result of $CostForNextAnc(p, e)$ |
| $c_4'$ | The result of $CostForNextAnc(virtual=true, p, e)$ |
| $c_5$ | The result of $CostForParentLookup(p, e)$ |
| $c_5'$ | The result of $CostForParentLookup(virtual=true, p, e)$ |
| $c_6$ | The result of $CostForChildLookup(p, e)$ |
| $c_7$ | The result of $CostForNextChild(p, e)$ |
| $c_8$ | The result of $CostForRTNPathRetrieval(p, e)$ |

Table 3. Constant definitions

| $T_1$ | Average cost for lookup in a B-Tree holding elements |
| $T_2$ | Average cost for moving among leafs of a B-Tree holding elements |
| $T_3$ | Cost for lookup in the Paths B-Tree |

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

<table>
<thead>
<tr>
<th>RE-basic / PE-Basic</th>
<th>RE-Path</th>
<th>PE-Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c_1$</td>
<td>$T_1$</td>
<td>$T_2$</td>
</tr>
<tr>
<td>$c_2$</td>
<td>$T_1$</td>
<td>$T_2$</td>
</tr>
<tr>
<td>$c_3$</td>
<td>$Avg\text{Depth}(\text{tagname}(e)) \times T_1$</td>
<td>$T_3+T_1$</td>
</tr>
<tr>
<td>$c_3'$</td>
<td>$Avg\text{Depth}(\text{tagname}(e)) \times T_1$</td>
<td>$T_3$</td>
</tr>
<tr>
<td>$c_4$</td>
<td>0</td>
<td>$T_1$</td>
</tr>
<tr>
<td>$c_4'$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$c_5$</td>
<td>$T_1$</td>
<td>$T_3+T_1$</td>
</tr>
<tr>
<td>$c_5'$</td>
<td>0</td>
<td>$T_3$</td>
</tr>
<tr>
<td>$c_6$</td>
<td>$T_1 + 0.5 \times \left( \text{Card}(p(e) / \text{tagname}) \times \text{Card}(e) / \text{tagname} \right)$</td>
<td>$T_1$</td>
</tr>
<tr>
<td>$c_7$</td>
<td>$(\text{Card}(p(e) / \text{tagname}) / \text{Card}(e) / \text{tagname}) \times T_2$</td>
<td>$T_3$</td>
</tr>
<tr>
<td>$c_8$</td>
<td>$Avg\text{Depth}(\text{tagname}(e)) \times T_1$</td>
<td>$T_3$</td>
</tr>
</tbody>
</table>

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 $c_1$ depends on the B-Tree height. We assign a constant $T_1$ 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 $T_1$ due to the linked list interconnecting B-Tree leaves. Assuming that $T_2$ is this cost on average, we assign $T_2$ to $c_2$.

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 $c_3$ for ancestor lookup is given by $Avg\text{Depth}(\text{tagname}(e)) \times T_1$. (Note that this is an overestimation when $\text{countEl}$ is provided, since backward navigation stops upon reach of an element that is a $\text{countEl}$ ancestor.) Retrieving ancestors other than the most distant one ($c_4$) incurs zero additional cost, since these have already been prefetched 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 ($c_7$), 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 ($c_6$). Finally, retrieving the RTN-path of an element involves as many B-Tree lookups ($T_1$) as is the average depth of elements having the same tag name ($c_8$). Constants $c_3'$, $c_4'$, $c_5'$ 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 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 ($c_8$) is $T_3$. Due to the relatively small number of RTN-paths, $T_3$ is usually much smaller than $T_1$.

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 ($c_3'$ and $c_5'$). Therefore, the cost for ancestor and parent lookups equals the cost of RTN-path retrieval ($T_3$), while shifting to the next ancestor ($c_4'$) incurs no cost. If virtual is set to false or omitted ($c_3$, $c_4$ and $c_5$), an extra B-Tree lookup ($T_1$) 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 $fp^{LU}$ and $b^{LU}$.

The second, used in $bp^{LU}$, is to detect overlapping windows and make use of an intermediate structure to temporarily keep result elements.

![Figure 3. A sample XML document](image)

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

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>contSeq: Sequence</td>
<td>The sequence given as input to the operator.</td>
</tr>
<tr>
<td>contEl: Element</td>
<td>A pointer to the last element read from contSeq.</td>
</tr>
<tr>
<td>window: Sequence</td>
<td>The current window of elements.</td>
</tr>
<tr>
<td>windEl: Element</td>
<td>A pointer to the last element read from window.</td>
</tr>
</tbody>
</table>

### 4.1 Lookup descendant and forward path

The LookUp $fp$, physical operator ($fp^{LU}$), 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 the chain where regular expression matching begins. Method $regExprFilter$ translates $path$ into a regular expression as described in [8]. The operator of request is very efficient when run upon a driver that provides cheap RTN-path retrieval.

Example 2: If $rtn=\langle r/b/a/b/c/d/e/fg \rangle$, path=$\langle c/f/g \rangle$ and level=5, then $regExprFilter(rtn, r/b/a/b/c/d/e/fg, c/f/g, 5) = true$, since regular expression ‘c(fg)’ matches suffix ‘c/d/e/fg’ 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 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.

```plaintext
Algorithm 1: Open() and next() implementations of $fp^{LU}$.
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();
   4. while (true) {
      5. if (window has more elements)
         6. windEl = window.next();
      7. if (chain is exhausted) {
         8. if (contEl is null return null;
         9. rootAnc = contEl;
         10. window = XPASS.Desc(one step for rootAnc); contEl = contSeq.next();
      } else {
         11. if (contEl is null return null;
         12. chain.add(contEl);
         13. contEl = contSeq.next();
      } }
```

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 $f_3$ of Figure 3 is descendant of $b_3$, but is not reachable via relative path ‘c/f/g’. $f_3$ is also descendant of $b_1$ and $b_2$ and, therefore, by skipping it, as in pruning, we would never take $f_3$ among the results.

Example 3: We will track the operation of $fp^{LU}$ 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, $f_1$ is output by the first next() call, $f_2$ is fetched but rejected and $f_3$ is fetched and output by the second next() call. Since $b1$’s window is exhausted, rootAnc now points to $b2$, the window of $b2$ is initialized, $f_5$ 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 \( b^7 \) are kept in chain. The following iterations of the outer while-loop of line 6, since \( \text{chains}\neq 2 \) (line 12), read each element (\( \text{windEl} \)) from the window of \( \text{rootAnc} \) and outputs it as long as there exists an element \( e \) in chain \( \text{rootAnc} \) such as \( \text{windEl} \) is reachable from \( e \) via \( p \) (line 16). Therefore, first \( f^6 \) is fetched from \( b^3 \) window and is rejected, then \( f^7 \) is fetched and output by the forth \( \text{next()} \) call (it is reachable from \( b^5 \) via \( /e/f \)), \( f^8 \), \( f^9 \), \( f^10 \) and \( f^11 \) are rejected, and, finally \( f^12 \) and \( f^13 \) are output by the fifth and sixth \( \text{next()} \) call, respectively, being reachable from \( b^7 \) via \( /e/f \). During the 7th \( \text{next()} \) call, chain gets empty, the window of \( b^9 \) is initialized and \( f^6 \) is output. □

### Table 6. Sample run of \( \text{fp}^\text{LU}_{\text{req}} \) operator

<table>
<thead>
<tr>
<th>rootAnc</th>
<th>contEl</th>
<th>chain</th>
<th>windEl</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>( b^1 )</td>
<td>( b^2 )</td>
<td>( , )</td>
<td>( f^1 )</td>
<td>( f^1 )</td>
</tr>
<tr>
<td>( b^2 )</td>
<td>( b^3 )</td>
<td>( b^4 )</td>
<td>( f^4 )</td>
<td>( f^4 )</td>
</tr>
<tr>
<td>( b^3 )</td>
<td>( b^5 )</td>
<td>( b^6 )</td>
<td>( f^6 )</td>
<td>( f^6 )</td>
</tr>
<tr>
<td>( b^5 )</td>
<td>( b^6 )</td>
<td>( b^7 )</td>
<td>( f^7 )</td>
<td>( f^7 )</td>
</tr>
<tr>
<td>( b^6 )</td>
<td>( b^7 )</td>
<td>( b^8 )</td>
<td>( f^8 )</td>
<td>( f^8 )</td>
</tr>
<tr>
<td>( b^7 )</td>
<td>( b^8 )</td>
<td>( b^9 )</td>
<td>( f^9 )</td>
<td>( f^9 )</td>
</tr>
<tr>
<td>( b^8 )</td>
<td>( b^9 )</td>
<td>( b^{10} )</td>
<td>( f^{10} )</td>
<td>( f^{10} )</td>
</tr>
<tr>
<td>( b^9 )</td>
<td>( \text{null} )</td>
<td>( , )</td>
<td>( f^{11} )</td>
<td>( f^{11} )</td>
</tr>
<tr>
<td>( b^{10} )</td>
<td>( \text{null} )</td>
<td>( , )</td>
<td>( f^{12} )</td>
<td>( f^{12} )</td>
</tr>
</tbody>
</table>

Changing line 16 to call the \( \text{Children()} \) PAM and the condition of line 9 to ‘if \( \exists e \in \text{chain} \neq \text{rootAnc} \) such as \( \text{windEl} \) isChildOf(\( e \)), gives us the pseudocode for the \( LU_1 \) operator (pLU1). Algorithm 2 illustrates the implementation of the \( \text{open()} \) and \( \text{next()} \) methods of the LU \( d_1 \) operator (\( d_1 \)). \( d_1 \) adopts pruning [2], but it skips and ignores context elements that are descendants of previously read ones (lines 12-14).

### Algorithm 2. \( \text{open()} \) and \( \text{next()} \) implementations of \( d_1\)

1. \( \text{open}() \)
2. \( \text{contEl} = \text{contSeq}\text{.next()}; \)
3. \( \text{rootAnc} = \text{contEl}; \)
4. \( \text{window} \)
5. \( \text{next}(); \)
6. \( \text{while true} \)
7. \( \text{if window has more elements} \)
8. \( \text{return window}\text{.next()}; \)
9. \( \text{else} \)
10. \( \text{if rootAnc is null} \)
11. \( \text{return null; \) \)
12. \( \text{window} = \text{XPath}\text{.Descendants(rootAnc, al); \) \)
13. \( \text{contEl} = \text{contSeq}\text{.next()}; \)
14. \( \text{while contEl is not null and contEl isDescendant(rootAnc)} \)
15. \( \text{contEl = contSeq}\text{.next(); \) \)
16. \( \text{rootAnc = contEl; \) \)
17. \( \}
18. \( \}

Algorithm 3 illustrates the implementation of the \( \text{open()} \) and \( \text{next()} \) methods of the LU \( d_2 \) operator (\( d_2 \)). \( d_2 \) adopts pruning [2], but it skips and ignores context elements that are descendants of previously read ones (lines 12-14).

### Algorithm 3. \( \text{open()} \) and \( \text{next()} \) implementations of \( \text{d}_2\)

1. \( \text{open}() \)
2. \( \text{contSeq}\text{.open()} \)
3. \( \text{contEl} = \text{contSeq}\text{.next()}; \)
4. \( \text{rootAnc} = \text{contEl}; \)
5. \( \text{next}(); \)
6. \( \text{while true} \)
7. \( \text{if window has more elements} \)
8. \( \text{return window}\text{.next()}; \)
9. \( \text{else} \)
10. \( \text{if rootAnc is null} \)
11. \( \text{return null; \) \)
12. \( \text{window} = \text{XPath}\text{.Descendants(rootAnc, al); \) \)
13. \( \text{contEl} = \text{contSeq}\text{.next()}; \)
14. \( \text{while contEl is not null and contEl isDescendant(rootAnc)} \)
15. \( \text{contEl = contSeq}\text{.next(); \) \)
16. \( \text{rootAnc = contEl; \) \)
17. \( \}
18. \( \}

Example 4: Consider \( \text{bp}^\text{LU}\) applied on \( f \) elements. The inverse of path \( ^{b^7}\text{c}^b \) is \( ^{\neg b^7}\text{c}^b \). Let’s also suppose that the current context element is \( f^7 \), of the XML document of Figure 3, with \( \text{RTN}(f^7) = /r/b/b/b/c/f \). The \( \text{Ancs()} \) PAM with \( f^7 \) and ‘b’ as arguments returns ancestors \( b^3, b^4, b^5 \) and \( b^6 \) 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 \( \text{regExprFilter} \) as follows: \( b^3: \text{regExprFilter}(/r/b/b/b/c/f, /c//f, 2) = \text{false}, \) \( b^4: \text{regExprFilter}(/r/b/b/b/c/f, /c//f, 3) = \text{false}, \) \( b^5: \text{regExprFilter}(/r/b/b/b/c/f, /c//f, 4) = \text{true}. \) Therefore, only \( b^5 \) is returned by a call on \( \text{next()} \) of \( \text{bp}^\text{LU}\).

The algorithm uses the \( \text{sortedElems} \) structure so as to avoid duplicates. Overlapping windows occur when the \( \text{Ancs()} \) method is called for a context element that is a descendant of the most distant element in the \( \text{sortedElems} \) 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 \( \text{ready to true} \) (line 27). Otherwise, fetching windows must go on (lines 18-26). For each window fetched, its elements are inserted into \( \text{sortedElems} \) 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 cheap RTN-path retrieval and most efficient when the \( \text{Ancs()} \) implementation is also cheap. The virtual argument of \( \text{Ancs()} \) is set to \( \text{true} \) (line 19), and therefore, for drivers such as PE-Path, the \( \text{Ancs()} \) calls are extremely cheap (cost components \( c^3 \) and \( c^4 \) are smaller than \( c^3 \) and \( c^4 \), respectively). However, in this case, if the \( \text{bp}^\text{LU}\) is not allowed to output virtual elements, these must be reloaded just before they are output (line 13).
Implementation of the lookup $a^\nu$ operator. Assume that the operator processed context element $f_6$ of Figure 3, being an ancestor of $f_6$, yet not reachable from $f_6$ via backward path $\nu$, would not be output. When context element $f_3$ is processed, $b_3$ should now be output.

**Example 5:** Let’s suppose that $\{f_2, f_3, f_5, f_6, f_8, f_{11}, f_{13}\}$ is the context sequence of the $b^\nu$ operator. The steps of the algorithm are shown in Figure 4. Elements of the sortedElements structure that are marked are underlined. The window of $f_2$ includes only $b_1$, which is put in sortedElements (group A of Figure 4(a)) and marked since it is reachable from $f_2$ via backward path $\nu$'. The window of $f_3$ also includes only $b_1$. Since the next context element ($f_5$) is not a descendant of $b_1$, all marked elements of sortedElements are output (that is only $b_1$) and sortedElements is empty. The window of $f_5$ consists only of $b_2$, and, since the following context element is not a descendant of $b_2$, $b_2$ is the only element of sortedElements (group B in Figure 4(a)), it is also marked and, thus, is output. sortedElements is empty again. Subsequently, $f_6$ is read. Its window consists of $b_3, b_4$ and $b_5$ which are kept in sortedElements. Finally, $f_3$ is a descendant of $b_3$ and $b_7$ is added in sortedElements. Finally, $f_3$ is also a descendant of $b_3$ whose ancestors, $b_3$ and $b_7$, are already in sortedElements, but this time $b_7$ is marked. Since no more context elements exist, the marked elements of sortedElements (group C in Figure 4(a)), namely $b_4$ and $b_7$, are output. □

4.3 Lookup cousin operator
To implement the novel cousin logical operator [10] $c^\nu_{p1, p2}$, we combine two of the previously described operators. First a $b^\nu_p$, a $p^\nu_{\text{parent}}$, or an $d^\nu_{\text{parent}}$ operator (depending on $p_1$) takes as input sequence the context sequence of the $c^\nu_{p1, p2}$ operator, while feeding itself as input to the second operator: a $f^\nu_p$, a $p^\nu_{\text{parent}}$, or an $d^\nu_{\text{parent}}$ operator (depending on $p_2$). 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 and Table 7. Table 8 summarizes the cost formulas for the LU operators.

### Table 7. Variables and functions used in cost estimation

<table>
<thead>
<tr>
<th>Variable/Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{tagname}()$</td>
<td>The tag name of the elements that outputs</td>
</tr>
<tr>
<td>$\text{lastp}$</td>
<td>The last step of path $p$</td>
</tr>
<tr>
<td>$\text{allowVirtual}$</td>
<td>The allowVirtual member (0 or 1) variable of the respective backward operator</td>
</tr>
</tbody>
</table>

### Table 8. Cost Formulas for LU operators

<table>
<thead>
<tr>
<th>Description</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{Cost} = \text{OUT}(\text{cost}) + \text{c1} + \text{c2}$</td>
<td>$\text{Cost} = \text{OUT}(\text{cost}) + \text{c1} + \text{c2}$</td>
</tr>
<tr>
<td>$\text{Cost} = \text{OUT}(\text{cost}) + \text{c3}$</td>
<td>$\text{Cost} = \text{OUT}(\text{cost}) + \text{c3}$</td>
</tr>
<tr>
<td>$\text{Cost} = \text{OUT}(\text{cost}) + \text{c4}$</td>
<td>$\text{Cost} = \text{OUT}(\text{cost}) + \text{c4}$</td>
</tr>
<tr>
<td>$\text{Cost} = \text{OUT}(\text{cost}) + \text{c5}$</td>
<td>$\text{Cost} = \text{OUT}(\text{cost}) + \text{c5}$</td>
</tr>
<tr>
<td>$\text{Cost} = \text{OUT}(\text{cost}) + \text{c6}$</td>
<td>$\text{Cost} = \text{OUT}(\text{cost}) + \text{c6}$</td>
</tr>
<tr>
<td>$\text{Cost} = \text{OUT}(\text{cost}) + \text{c7}$</td>
<td>$\text{Cost} = \text{OUT}(\text{cost}) + \text{c7}$</td>
</tr>
</tbody>
</table>

The worst case scenario for the $f^\nu_p$, $c^\nu_p$, $d^\nu_p$, $b^\nu_p$, $p^\nu_p$, and $d^\nu_p$ 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 $d^\nu_p$ and $f^\nu_p$, and in the case of $c^\nu_p$, $c_3$ the cost of $d^\nu_p$ and $b^\nu_p$ and $c_5$ in the case of $p^\nu_p$. Traversing these windows incurs a cost of $c2$ for $d^\nu_p$ and $f^\nu_p$, $c7$ for $c^\nu_p$, $c4$ for $d^\nu_p$ and $b^\nu_p$ and $0$ for $p^\nu_p$ (singleton window) for each window element. The total number of window elements
these operators is given by \(\text{Occ}(\text{cp}, \mathcal{L}(\text{tagname}(\text{op})))\) for \(dLU\) and \(cLU\) and \(\text{Occ}(\text{cp}, \mathcal{L}(\text{tagname}(\text{op})))\) for \(bLU\) and \(fpLU\).

For the RTN-path-based algorithms, processing window elements involves the cost implied by \(\text{regExprFilter()}\). For backward operators, if the \(\text{Ancs()}\) 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 \(\text{allowVirtual}\) 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 \(\text{OUT}(\text{op})\) times the cost for element lookup \(\text{c1}(\text{e})\).

5. SORT-MERGE-BASED OPERATORS

The basic strategy of the SortMerge-based (SM) XPath operators is to traverse two DODF sequences of elements, \(\text{left}\) and \(\text{right}\). 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

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{leftSeq})</td>
<td>The context sequence ((\text{contSeq}))</td>
</tr>
<tr>
<td>(\text{rightSeq})</td>
<td>The sequence where solutions are sought</td>
</tr>
<tr>
<td>(\text{leftEl})</td>
<td>The current element on the left sequence</td>
</tr>
<tr>
<td>(\text{rightEl})</td>
<td>The current element on the right sequence</td>
</tr>
<tr>
<td>(\text{docRoot})</td>
<td>The root element of the xml document</td>
</tr>
<tr>
<td>(\text{afterAll})</td>
<td>A virtual element located after all nodes in the xml tree</td>
</tr>
</tbody>
</table>

Algorithm 5. \(\text{Open()}\) and \(\text{next()}\) implementations of \(dSM\)

The SortMerge-based \(d\) algorithm \((dSM)\) (algorithm 5), once a matching pair is found (line 13) the current element on the right is returned. Note that a null \(\text{left}\) or \(\text{right}\) 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 \(\text{moveRightAfterLeft()}\) (line 10). So if \(\text{rightEl}\) is not a descendant of \(\text{leftEl}\), then no element subsequently found on the right will be a descendant of this \(\text{leftEl}\) either, so it is safe to advance the left sequence (line 16). This fact along with the implementation of the method \(\text{moveRightAfterLeft()}\) achieves fast skipping of irrelevant elements on the right sequence. Method \(\text{isAfter()}\) has usually a very cheap implementation - involving a single comparison between integers in the case of the \(\text{RE}\) driver (section 3.1) or a single lexicographical comparison between strings in the case of the \(\text{PE}\) driver (section 3.2). Note that only two elements must be kept in memory for correct processing (\(\text{leftEl}\) and \(\text{rightEl}\)).

```
Algorithm 6. \(\text{Open()}\) and \(\text{next()}\) implementations of \(fpSM\)
```

For the SortMerge-based \(fp\) \((fpSM)\), 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 \((\text{possibleMatches})\), 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 \(\text{leftEl}\) and \(\text{rightBound}\). We always make sure that the current right element \((\text{rightEl})\) is located between these bounds (lines 12, 20). Function \(\text{slideWindow()}\) (lines 24-29) adjusts the bounds to the next available pair of elements in \(\text{leftSeq}\) and returns true/false indicating success or failure. Each time the window slides one position forward, the previous \(\text{leftEl}\) is added to \(\text{possibleMatches}\) (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 \(\text{possibleMatches}\) (lines 34-42). Note that this involves using both the \(\text{getRTNPath()}\) method of the \text{Element} interface and regular expression filtering, as described in Section 4.1. Since \(\text{rightEl}\) is at all times after \(\text{leftEl}\) and all elements in \(\text{possibleMatches}\), we can remove all elements in \(\text{possibleMatches}\) 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 fpSM on given its context sequence is {b1, b2 b3, b4, b5}, using the XML document of Figure 3. Table 10 illustrates all the details running of 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 fp in our example). Entering the main loop, rightEl is not null (read: rightSeq is not over yet). Also, 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 /f//p operator with a last element as a percentage of the total elements of tagname(cop)) ofperc=Occ(//tagname(op), //tagname(cop)) / Card(//tagname(cop)) So for each element on the right there exist descrPerRight = perc*OUT(cop) descendant elements in the left sequence, meaning a total of descrPerRight * Cardi(tagname(op)) path-matching operations. The cost formula for the SM-operator can be easily derived from the cost models of fpSM and bpLU. Table 11 summarizes the cost formulas for the SM-based operators.

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 TwigStack for which we have implemented operators and incorporated them in our framework. For Staircase join we implemented the dStaircase and dStaircase 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 repeated calls to next result in traversing the whole rightSeq in the worst case scenario, which incurs a c2*Cardi(tagname(op)) cost for traversing rightSeq. The same analysis holds for the cSM and bpLU cases. As in the previous cases, the cost formula for fpSM 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*Cardi(tagname(op)) 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 Cardi(tagname(op)). Similarly to the previous case, the bpSM 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)) ofperc=Occ(//tagname(op), //tagname(cop)) / Cardi(tagname(cop)) So for each element on the right there exist descrPerRight = perc*OUT(cop) descendant elements in the left sequence, meaning a total of descrPerRight * Cardi(tagname(op)) path-matching operations. The cost formula for the SM-operator can be easily derived from the cost models of fpSM and bpLU. Table 11 summarizes the cost formulas for the SM-based operators.

Table 10. Sample run of fpSM operator

<table>
<thead>
<tr>
<th>leftEl</th>
<th>rightBound</th>
<th>rightEl</th>
<th>possibleMatches</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>open()</td>
<td>b1</td>
<td>b2</td>
<td>f1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>b2</td>
<td>b3</td>
<td>b1</td>
<td>f1</td>
</tr>
<tr>
<td></td>
<td>b3</td>
<td>b4</td>
<td>b2</td>
<td>f1</td>
</tr>
<tr>
<td></td>
<td>b4</td>
<td>b5</td>
<td>b3</td>
<td>f1</td>
</tr>
<tr>
<td></td>
<td>b5</td>
<td>b6</td>
<td>b4</td>
<td>f1</td>
</tr>
<tr>
<td></td>
<td>b6</td>
<td>afterAll</td>
<td>b5</td>
<td>f1</td>
</tr>
</tbody>
</table>

5.1 Cost Modeling
Regarding dSM and dSM, 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
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 \( d_{\text{item}} \) physical operator is given as input the sequence produced by retrieving 10% of the \( \text{parlist} \) 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

<table>
<thead>
<tr>
<th>context</th>
<th>relative path</th>
</tr>
</thead>
<tbody>
<tr>
<td>q1</td>
<td>/parlist</td>
</tr>
<tr>
<td>q2</td>
<td>/item</td>
</tr>
<tr>
<td>q3</td>
<td>/to</td>
</tr>
<tr>
<td>q4</td>
<td>/enph</td>
</tr>
<tr>
<td>q5</td>
<td>/open auction</td>
</tr>
<tr>
<td>q6</td>
<td>/item</td>
</tr>
<tr>
<td>q7</td>
<td>/to</td>
</tr>
<tr>
<td>q8</td>
<td>/enph</td>
</tr>
<tr>
<td>q9</td>
<td>/to</td>
</tr>
<tr>
<td>q10</td>
<td>/enph</td>
</tr>
</tbody>
</table>

### 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 (\( \text{fpLU} \), \( \text{bpLU} \), \( \text{csLU} \), \( \text{fpSM} \), \( \text{bpSM} \) or \( \text{csSM} \), 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 \( \text{bpLU} \) or (SM-naive) we could use a plan consisting of \( \text{fpLU} \), \( \text{bpLU} \) and, especially \( \text{csLU} \) very fast. Recall that over this driver, \( \text{bpLU} \) fetches only virtual elements from \( \text{Ance()} \) calls. If \( \text{bpLU} \) is not encapsulated in a \( \text{csSM} \) or \( \text{csLU} \) operator (allowVirtual=false), only ancestors to be output are retrieved from the actual data.

![Figure 5. Query execution on the RE-path – 560MB XMark](image)

![Figure 6. Query execution on the PE-path – 560MB XMark](image)

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: \( (s_1 - s_2)/s_2 \). 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-leaning, gives LU operators a significant advantage over Staircase (and PathStack). Over the RE-Path driver, using \( \text{bpLU} \), \( \text{csLU} \) and \( \text{csSM} \) 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 \( \text{bpLU} \), described in Section 4.2) is a suboptimal option (this is not the case for \( \text{aLU} \) that fetches only ancestors included in the result sequence by feeding the \( \text{Ance()} \) PAM with the cousinEl argument). On the contrary, over PE-Path, the cheap implementation of \( \text{Ance()} \) PAM makes \( \text{aLU} \), \( \text{bpLU} \) and, especially \( \text{csLU} \) very fast. Recall that over this driver, \( \text{bpLU} \) is not encapsulated in a \( \text{csSM} \) or \( \text{csLU} \) operator (allowVirtual=false), only ancestors to be output are retrieved from the actual data.

On the Edge-based RE-Path driver, PAMs DescInRange() and AncsInRange() used in the \( \text{dAnce} \) and \( \text{aAnce} \) implementations respectively implement skipping exactly as in [2]. As shown in Figure 7, LU (even LU-naive) 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.
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.

(a) query q2 (descendant)  (b) query q4 (ancestor)
(c) query q6 (forward path)

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.

(a) query q4 (ancestor)  (b) query q6 (forward path)

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.

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 Twig2Stack on RE-path, but also on PE-path, which is inherently slower when it comes to forward navigation. The execution time of Twig2Stack 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 94%, 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).

(a) q5 (forward path)  (b) q7 (backward path)
(c) q9 (cousin)

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 \textsc{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 \textit{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