XPath Query Simplification with regard to the Elimination of Intersect and Except Operators

Sven Groppe
Digital Enterprise Research Institute (DERI), University of Innsbruck, Institute of Computer Science, A-6020 Innsbruck, Austria
Sven.Groppe@deri.org

Stefan Böttcher
University of Paderborn, Faculty 5, Fürstenallee 11, 33098 Paderborn, Germany
stb@uni-paderborn.de

Jinghua Groppe
Digital Enterprise Research Institute (DERI), University of Innsbruck, Institute of Computer Science, A-6020 Innsbruck, Austria
Jinghua.Groppe@deri.org

Abstract

XPath is widely used as an XML query language and is embedded in XQuery expressions and in XSLT stylesheets. In this paper, we propose a rule set which logically simplifies XPath queries by using a heuristic method in order to improve the processing time. Furthermore, we show how to substitute the XPath 2.0 **intersect** *and* **except** *operators in a given XPath query with computed filter expressions. A performance evaluation comparing the execution times of the original XPath queries, which contain the* **intersect** *and* **except** *operators, and of the queries that are the result of our simplification approach shows that, depending on the used query evaluator and on the original query, performance improvements of a factor of up to 350 are possible.*

1. Introduction

The XPath language ([20], [21]) is used to address XML nodes. The XPath language is not only a stand-alone language, but is even embedded in W3C's transformation language XSLT and in W3C's query language XQuery. Because of their complexity, XPath queries can contain redundant constructs. Furthermore, if the XPath evaluator does not optimize XPath queries, the redundant constructs in the query can cause high processing costs.

Whereas several previous contributions deal with the complexity of XPath evaluation [9] and efficient algorithms for XPath evaluation [8], we present a two step approach for logically transforming XPath queries containing the **intersect** or **except** operators. First, we transform XPath queries which contain the XPath 2.0 **intersect** and **except** operators into equivalent XPath queries, which do not contain these operators any more. Second, we apply a set of simplification rules, which could be also applied to any XPath expression that is not necessarily the result of an elimination of the **intersect** or **except** operators. Our performance evaluation shows that our approach speeds up the execution compared to the original query.

The **intersect** XPath 2.0 operator is used in the approaches for the optimization of applying multiple XPath queries [18]. Furthermore, the **intersect** XPath 2.0 operator can be used for access control by query modification [4], whenever the access rights of a user are expressed by an XPath expression.

The **except** XPath 2.0 operator is used for query optimization based on caching when for answering `Q`, the query processor loads only all nodes matching a given XPath query `Q` except those nodes, which are already in the cache.

Furthermore, our proposed rule set for XPath simplification can be used to check whether a query is unsatisfiable, i.e. the query result is equal to `{}` for every XML document. Given two XPath expressions `XP1` and `XP2`, we can logically test '`XP1` **intersect** `XP2 = {}`' and '`XP1` **except** `XP2 = {}`', which is equivalent to `XP1` $\subseteq$ `XP2`, by the logical satisfiability test of the determined equivalent XPath expression without **intersect** and **except** operators.

In the scenarios with distributed XML data sources, where the content `C` of a data source is described by an XPath expression `XC`, the logical intersection test can be used to prove that `C` does not contain data required for answering a given query `Q` by proving that '`XC` **intersect** `Q = {}`'. If we can prove '`XC` **intersect** `Q = {}`', we can avoid querying the XML data source, which saves time for connecting to

and querying a remote data source and thus reduces network load.

Furthermore, when X(Q) is the XPath expression that describes the XML fragment needed to answer a given query Q, the logical subsumption test can be used to check whether Q can be fully answered from the content of a cache described by an XPath expression C by proving $X(Q) \subseteq C$.

The contributions of this paper are:

- A rule set for logical simplification of any XPath expression.
- The general form of the reverse pattern E^{-1} of an XPath expression E. The application of E^{-1} to the current context node returns a non-empty result if E matches the current context node.
- A general approach for eliminating the **intersect** XPath 2.0 operator and thus the reduction of the logical intersection test of two XPath expressions to the satisfiability test of the resultant XPath expression.
- A general approach for eliminating the **except** XPath 2.0 operator and thus the reduction of the logical subsumption test of two XPath expressions to the satisfiability test of the resultant XPath expression.
- A performance analysis that shows that, depending on the used query evaluator and on the original query, we achieve high speed-up factors if we can simplify the XPath query to the empty expression, if we can eliminate reverse axes, or if we can eliminate many location steps. Otherwise, the simplified queries are in most cases a little bit faster and in few cases a little bit slower.

In Section 2, we introduce the reverse pattern of an XPath query, which is used in Section 3 to eliminate the **intersect** operator and in Section 4 to eliminate the **except** operator. Section 5 describes the proposed rule set for XPath queries. Section 6 presents a performance analysis of the achieved speed-up factors of XPath evaluation. The paper ends up with the further related work in Section 7 and the summary and conclusions in Section 8.

2. Reverse Pattern

Let us assume that an XPath expression E is given, which is used as pattern. We will show that evaluating the *reverse pattern* E^{-1} of E as a filter expression of an XML node $d, i.e. $d[E^{-1}], will return a non-empty result if E matches the XML node $d.

We present an extended variant of the approach in [16] for a superset of the XPath patterns of XSLT. Contrary to our contribution, the approach presented in [16] only supports the XPath patterns of XSLT and not complete XPath, which is supported by our approach.

For the purpose of the determination of the reverse pattern of a given XPath expression, we first define the reverse axes of an XPath axis.

Definition 1 (*reverse axes of an XPath axis*): The *reverse axes* of a given XPath axis are defined in the middle column of Figure 1.

Note that the parent of an attribute or of a namespace node is its element node, but an attribute or namespace node is not a child of its element node. Therefore, attribute nodes and namespace nodes cannot be accessed by the child or descendant axes, and also not by the descendant-or-self axis, if the attribute node or namespace node is not the current context node. An attribute node can only be accessed by the attribute axis and a namespace node only by the namespace axis. Thus, there is more than one reverse axis of the ancestor, ancestor-or-self or parent axis (see Figure 1).

Axis A	Reverse Axes of A	Additional Test
ancestor	1) descendant 2) descendant-or-self::node()/attribute 3) descendant-or-self::node()/namespace	[self **instance of** element()*]
ancestor-or-self	1) descendant-or-self 2) descendant-or-self::node()/attribute 3) descendant-or-self::node()/namespace	
attribute	parent	[self **instance of** attribute()*]
child	parent	[self **instance of** element()*]
descendant	ancestor	[self **instance of** element()*]
descendant-or-self	ancestor-or-self	
following	preceding	[self **instance of** element()*]
following-sibling	preceding-sibling	
namespace	parent	[**not**(self **instance of** element()*) **and not**(self **instance of** attribute()*)]
parent	1) child 2) attribute 3) namespace	[self **instance of** element()*]
preceding	following	[self **instance of** element()*]
preceding-sibling	following-sibling	
self	self	

Figure 1: Reverse axes and additional test of an XPath axis

The reverse axis of the attribute axis, of the child axis and of the namespace axis is the parent axis, which does not differ between attribute

nodes, namespace nodes and other nodes (in comparison to the original axis). Therefore, we will use an *additional test* (see Definition 2) in the definition of the reverse pattern (see Definition 3) to distinguish between the different node types, which describe the restrictions of the resultant nodes of the axes given in Figure 1.

Definition 2 (*additional test*): The *additional test* of a given XPath axis is defined in the right column of Figure 1.

Definition 3 (*reverse pattern of an XPath expression*): The *reverse pattern* of an XPath expression is computed as follows: At first, we transform the XPath expression into its long form. We eliminate as often the innermost **`intersect`** operator or **`except`** operator respectively in the expression according to Section 3 or Section 4 respectively as there are **`intersect`** or **`except`** operators. If there are disjunctions ("`|`") *outside the scope of filter expressions* in the XPath expression, then we factor out the disjunctions and reverse each expression of the disjunctions separately. The whole reverse pattern is the disjunction of all separately reversed expressions. Each filter expression remains unchanged. Without disjunctions outside the scope of filter expressions, a relative XPath expression $E_{relative}$ has the form

$$axis_1::test_1[F_{11}]\ldots[F_{1n_1}]/$$
$$axis_2::test_2[F_{21}]\ldots[F_{2n_2}]/\ldots/$$
$$axis_m::test_m[F_{m1}]\ldots[F_{mn_m}],$$

and an absolute XPath expression $E_{absolute}$ has the form

$$/axis_1::test_1[F_{11}]\ldots[F_{1n_1}]/$$
$$axis_2::test_2[F_{21}]\ldots[F_{2n_2}]/\ldots/$$
$$axis_m::test_m[F_{m1}]\ldots[F_{mn_m}]$$

where $axis_i$ are XPath axes, $test_i$ are node tests and F_{ij} are filter expressions. The reverse pattern of $E_{relative}$ and of $E_{absolute}$ is

$$self::test_m[F_{m1}]\ldots[F_{mn_m}]\ T_m/$$
$$(raxis_{m1}::test_{m-1}|\ldots|$$
$$raxis_{mp_m}::test_{m-1})[F_{(m-1)1}]\ldots[F_{(m-1)n_{m-1}}]\ T_{m-1}/$$
$$\ldots/$$
$$(raxis_{21}::test_1|\ldots|$$
$$raxis_{2p_2}::test_1)[F_{11}]\ldots[F_{1n_1}]\ T_1/$$
$$(raxis_{11}::node()|\ldots|raxis_{1p_1}::node())\ T_{root},$$

where T_{root} is `[self::node() is root()]` for $E_{absolute}$ and T_{root} is `[self::node() is $c]` for $E_{relative}$, `$c` must contain the context node, $raxis_{i1}$ ... $raxis_{1p_i}$ are the reverse axes of $axis_i$, and T_i is the additional test of $axis_i$, or T_i is the empty expression, if there is no additional test of $axis_i$.

Example 1 (*reverse pattern*): The *reverse pattern* of `child::object` is

```
self::object[self instance of element()*]/
parent::node()[self::node() is $c],
```

the reverse pattern of `/` is

```
  self::node()[self::node() is root()],
```

and the reverse pattern of

```
/child::contains/child::object[position()=1] |
 ancestor::object[attribute::name='cockpit']
```

is

```
self::object[position()=1]
            [self instance of element()*]
/parent::contains[self instance of
 element()*]
/parent::node()[self::node is root()]    |
self::object[attribute::name='cockpit']
[self instance of element()*]
/(descendant::node()                    |
descendant-or-self::node()/attribute::node()|
descendant-or-self::node()/namespace::node())
[self::node() is $c].
```

Proposition 1 (*match test by applying the reverse pattern*): An XPath pattern `E` matches an XML node `$d` if $\$d[E^{-1}]$ can be evaluated to a non-empty set, where E^{-1} is the reverse pattern of `E`.

Proof of Proposition 1: Proposition 1 can be proved by considering each step in the evaluation of the proposed match tests and by considering each step in the definition of the reverse pattern for every combination of the axis and of the node tests. □

3. Intersection Operator

In the following section, we present how the **`intersect`** XPath 2.0 operator can be simplified. Given the XPath expressions `XP1` and `XP2`, the XPath expression `XP1` **`intersect`** `XP2` returns those XML nodes, which are contained in `XP1` and in `XP2`. For the purpose of eliminating the **`intersect`** operator, we define the *equivalence of XPath expressions*.

Definition 4 (*equivalence of XPath expressions*): Two XPath expressions `XP1` and `XP2` are *equivalent*, which we notate $XP1 \equiv XP2$, if `XP1` returns the same XML nodes as `XP2` for all possible input XML documents and for all possible context nodes.

Proposition 2 (*intersection without* `intersect` *operator*): $XP1$ **`intersect`** $XP2 \equiv XP1[XP2^{-1}]$, where `XP1` and `XP2` are XPath expressions and $XP2^{-1}$ is the reverse pattern of `XP2`.

Proof of Proposition 2: The evaluation of the XPath expression `XP1` returns the XML nodes of `XP1`. We can test those XML nodes of `XP1`, whether they are also contained in `XP2`, by applying the reverse pattern

of XP2 in a filter after XP1, i.e. XP1[XP2^{-1}]. As the XPath expression XP1 **intersect** XP2 returns those XML nodes, which are contained in XP1 and in XP2, XP1[XP2^{-1}] is equivalent to XP1 **intersect** XP2. □

4. Except Operator

In the following section, we present how the **except** XPath 2.0 operator can be eliminated. Given the XPath expressions XP1 and XP2, the XPath expression XP1 **except** XP2 returns those XML nodes, which are contained in XP1, but which are not contained in XP2.

Proposition 3 (*difference without* `except` *operator*): XP1 **except** XP2 ≡ XP1[**not**(XP2^{-1})], where XP1 and XP2 are XPath expressions and XP2^{-1} is the reverse pattern of XP2.

Proof of Proposition 3: The XPath expression XP1 returns the XML nodes of XP1. We can test those XML nodes of XP1, whether they are *not* contained in XP2, by checking, if the reverse pattern of XP2 cannot be applied in a filter after XP1, i.e. XP1[**not**(XP2^{-1})]. As the XPath expression XP1 **except** XP2 returns those XML nodes, which are contained in XP1, but not contained in XP2, XP1[**not**(XP2^{-1})] is equivalent to XP1 **except** XP2. □

Proposition 4: XPath 1.0 is closed under complementation.

We need following lemma for the proof of Proposition 4.

Lemma 1: The most general XPath query

```
G_all := /descendant-or-self::node() |
         /descendant-or-self::node()/attribute::node()|
         /descendant-or-self::node()/namespace::node()
```

returns all XML nodes of an XML document.

Proof of Lemma 1: `/descendant-or-self::node()` describes the document node of an XML document and all its descendant nodes except of all attribute nodes, which are described by `/descendant-or-self::node()/attribute::node()`, and all namespace nodes, which are described by `/descendant-or-self::node()/namespace::node()`. There are no other XML nodes than these in an XML document. □

Proof of Proposition 4: The complement of a query Q is G_{all} **except** Q ≡ G_{all}[**not**(Q^{-1})]. □

Note that certain subsets of XPath are not closed under complementation [3]. In comparison, XPath 1.0 can test whether a node of the current location step is the document node by `[self::node()` **is** `root()]` or is the context node by `[self::node()` **is** `$c]`, where `$c` must contain the context node. We use these tests in the definition of the reverse patterns.

Any extension of Core XPath [8], which is closed under complementation, can define every first order definable set of paths [15]. Thus, XPath 1.0 is first order complete (see Proposition 4).

5. Simplification

In order to evaluate XP1[XP2^{-1}] or XP1[**not**(XP2^{-1})] respectively, current implementations of XPath evaluators determine first all XML nodes of XP1 and then test the filter expression XP2^{-1} or **not**(XP2^{-1}) respectively. In the following, we present simplification rules, which simplify the given XPath query logically.

The goals to achieve are that

- the XPath evaluator does not first determine *all* XML nodes of XP1 and then apply the filter,
- instead we exclude intermediate XML nodes of location paths, which do not contribute to the final result, as early as possible.

Therefore, the goals of our heuristic method are that

- sub-expressions are reduced to the empty path, wherever possible.
- the XPath query does not contain a reverse axis so that the XPath evaluator processes only forward axes.
- our approach eliminates the **not**(…) operator, wherever possible.
- our approach eliminates location steps with a `self` axis, wherever possible, in order to avoid unnecessary location steps.

5.1. Supported Subset of XPath

By using our approach as already described in Section 2., Section 3. and Section 4., we can eliminate the **except** and **intersect** operator within all XPath expressions.

The proposed rule set (see Section 5.2) simplifies sub-expressions of the original query, which conform to the subset of XPath of Figure 2. Note that sub-expressions, which do not conform to the subset of XPath of Figure 2, do not cause an error, but these sub-expressions are only partially simplified.

```
path ::= relPath | "/" relPath | path "|" path.
relPath ::= axis "::" nodetest | relPath[qualif] |
            relPath "/" relPath | ⊥.
qualif ::= path | qualif "and" qualif | qualif "or"
           qualif | "not(" qualif ")" | "self
           instance of" ("element()*" |
           "attribute()*") | "self::node() is
           root()".
axis::= "child" | "descendant" | "attribute" |
         "self" | "descendant-or-self" | "following-
         sibling" | "following" | "namespace" |
         "parent" | "ancestor" | "ancestor-or-self".
nodetest ::= ("element" | "attribute" | "text" |
              "comment" | "document-node" |
              "comment" | "processing-instruction"
              | "node") "()" | "*" | name.
```

where ⊥ represents the empty expression and `name` represents a name test.

Figure 2: Supported subset of XPath for simplification of sub-expressions.

5.2. Used Rule Set

First, we introduce the term of the *more restrictive node tests*, which will be used in the presented rule set.

Definition 5 (*more restrictive*): We call a node test `t1` *more restrictive* than a given node test `t2`, which we notate `t1` « `t2`, if the following condition holds: `t1` is not identical to `t2`. Furthermore, if `self::t1` returns the context node, then also `self::t2` returns the context node for all possible context nodes.

Proposition 5 (*more restrictive*): The name test is more restrictive than `*`, `attribute()` is more restrictive than `*`, `*` is more restrictive than `element()`, {`text()`, `comment()`, `document-node()`, `processing-instruction()`} are more restrictive than `element()`, the name test and {`*`, `element()`, `attribute()`, `text()`, `comment()`, `document-node()`, `processing-instruction()`} are more restrictive than `node()`.

Proof of Proposition 5: We can conclude Proposition 5 from [21]. □

We notate `t1`⊚`t2` if `t1` is the name node test and `t2`∈{`element()`, `attribute()`}, or `t1`∈{`element()`, `attribute()`} and `t2` is the name node test. These combinations of node tests do not exclude each other, although it holds that `not(t1«t2)` and `not(t2«t1)`.

We use the approach of [17] (and in more detail the rule set $RuleSet_2$ of [17]) in order to eliminate all reverse axes of the given XPath query. As [17] does not contain a rule set in order to simplify XPath queries, we introduce a rule set, the application of which simplifies XPath expressions by a heuristic method after the application of the rule set of [17]. We apply a rule of the proposed rule set in the following way: We start with the subexpressions with the most location steps. We first check whether a subexpression in the XPath expression fits to the left side of a rule and if there is an additional condition on the right side, we also check whether the condition of the right side is fulfilled. If and only if we successfully checked the subexpression, then we replace the subexpression with the subexpression on the right side of the rule. We proceed to simplify the XPath expression as long as we can apply a rule, which modifies the XPath expression, of the proposed rule set.

In the following, let `p`, `p1`, `p2`, `p3` and `p4` be (relative or absolute) paths, let `t1` and `t2` be node tests, let `a1` and `a2` be forward axes (if `a1` and `a2` are not stated to be reverse axes) and `F` stands for an expression, which is the empty expression ⊥ or a predicate.

The rules do not deal explicitly with the special case in which a result starts with `p[p1]`, where `p` is the empty expression ⊥ or the document root `/` and `p1` is an XPath expression. In this case, we implicitly replace `p[p1]` with `p self::node()[p1]`.

If a sub-expression in a path is reduced to the empty expression ⊥, then the entire path is reduced to the empty expression ⊥:

```
p1/⊥/p2 ≡ ⊥
```

Furthermore, if an operand in a disjunctive expression is reduced to the empty expression ⊥, we implicitly replace the whole disjunctive expression with the other operand:

```
⊥ | p ≡ p
p | ⊥ ≡ p
```

The rules for simplifying expressions, which contain the `self` axis, are as follows:

```
a1::t1[self::t2 F] ≡ { a1::t1 F   if t1 « t2 or t1 = t2
                     { a1::t2 F   if t2 « t1
                     { a1::t1[self::t2 F]  if t1⊚t2
                     { ⊥          otherwise

a1::t1/self::t2 F ≡ { a1::t1 F   if t1 « t2 or t1 = t2
                    { a1::t2 F   if t2 « t1
                    { a1::t1/self::t2 F  if t1⊚t2
                    { ⊥          otherwise

p/a1::t1[self instance of element()*] ≡
  { p/a1::element()   if element()«t1 or t1=element()
  { p[self instance of element()*]/a1::t1   if a1=self
  { p/a1::t1          if t1«element(), or a1∈{child,
  {                   descendant, following, following-
  {                   sibling}
  { ⊥                 otherwise

p/self::t1[self::node() is root()] ≡
                  p[self::node() is root()]/self:t1

p/self::t1[not(self::node() is root())] ≡
        p[not(self::node() is root())]/self:t1

/self::node()/p ≡ /p
```

```
p/a1::t1[self::node() is root()]  ≡ ⊥    if a1≠self

/self::t1[self::node() is root()] ≡ /self::t1

p/a1::t1[self instance of attribute()*] ≡
   p/a1::t1            if a1 = attribute
   p/a1::attribute()   if attribute()«t1 or t1=attribute()
   p[self instance of attribute()*]/a1::t1 if a1=self
   ⊥                   otherwise
```

The following rules consider the **not**(…) operator:

```
not(p1/p2)        ≡ not(p1) or p1[not(p2)]
not(p1[p2])       ≡ not(p1) or p1[not(p2)]
not(p1 | p2)      ≡ not(p1) and not(p2)
not(p1 or p2)     ≡ not(p1) and not(p2)
not(p1 and p2)    ≡ not(p1) or not(p2)
not(not(p))       ≡ p
p[p1][not(p1)]    ≡ ⊥
```

The following rules eliminate different operators in filter expressions:

```
p1[p2 or p3]  ≡ p1[p2] | p1[p3]
p1[p2 | p3]   ≡ p1[p2] | p1[p3]
p1[p2 and p3] ≡ p1[p2][p3]
```

We eliminate equivalent expressions in disjunctions and factor out disjunctions:

```
p | p ≡ p
p1(/p2 | /p3) ≡ p1/p2 | p1/p3
(p1 | p2)/p3  ≡ p1/p3 | p2/p3
```

The following rules simplify expressions with **and** or **or** operators:

```
p1 and p1[p2] ≡ p1[p2]
p1 and p1/p2  ≡ p1/p2
p1 and (p2 or p3) ≡ (p1 and p2) or (p1 and p3)
(p1 or p2) and p3 ≡ (p1 and p3) or (p2 and p3)
```

The following rules deal with location steps containing the **not**(…) operator:

```
p/a1::t1[not(parent::t2)]≡ p[not(self::t2)]/a1::t1
         if a1∈{child, attribute, namespace}

p/a1::t1[not(self::t2)] ≡
   ⊥          if t1 = t2 or t1 « t2
   p/a1::t1   if both, t1 and t2 are name tests and t1 ≠ t2,
              if t1 ≠ t2 and t1, t2 ∈ {attribute(), text(),
              comment(), document-node(), processing-
              instruction()},
              if a1∈{child, descendant, following,
              following-sibling} and t2=attribute(), or
              if a1=attribute and t2 is not an attribute(), *
              nor a name test.

p/descendant::t1[not(parent::t2)]≡ p/descendant-or-
self::node()[not(self::t2)]/child::t1

p/following-sibling::t1[not(a2::t2)] ≡
   p[not(a2::t2)]/following-sibling::t1
         if a2 ∈ {parent,ancestor}

p/a1::t1[not(self::node() is root())] ≡ p/a1::t1
         if a1≠self

p/a1::t1[not(ancestor::t2)] ≡
   p[not(self::t2)][not(ancestor::t2)]/a1::t1
         if a1∈{child, attribute, namespace}

/self::t1[not(self::node() is root())]≡⊥
```

We have factored out the disjunctions before so that we can apply simple rules for the simplification of the XPath expressions. After no rule from the above rule set can be applied, we apply the following rules as long as possible in order to combine common sub-expressions of the XPath expression again:

```
p1/p2 | p1/p3 ≡ p1/ (p2 | p3)
p1/p2 | p3/p2 ≡ (p1 | p3)/p2
```

Example 2 (*intersect*): Let `XP1 = /child::a/child::b`, let `XP2 = /child::a/child::b[child::c]`. Then `XP1 intersect XP2 ≡ /child::a/child::b[self::b[child::c][self instance of element()*]/parent::a[self instance of element()*]/parent::node()[self::node() is root()]]` according to Section 3, which can be transformed to the simplified query `/child::a/child::b[child::c]` using our proposed rule set.

Example 3 (*intersect*): Let `XP1 = /child::node()/self::a/child::node()/self::b`, let `XP2 = /descendant-or-self::c/ancestor-or-self::b`. Then `XP1 intersect XP2 ≡ /child::node()/self::a/child::node()/self::b[self::b/descendant-or-self::c/ancestor-or-self::node()[self::node() is root()]]` according to Section 3, which can be transformed to the simplified query `/child::a/child::b[descendant::c]` using our proposed rule set.

Example 4 (*except*): Let `XP1 = /child::a/child::b`, let `XP2 = /child::a/child::b[child::c]`. Then `XP1 except XP2 ≡ /child::a/child::b[not(self::b[child::c][self instance of element()*]/parent::a[self instance of element()*]/parent::node()[self::node() is root()])]` according to Section 4, which can be transformed to the simplified query `/child::a/child::b[not(child::c)]` according to our proposed rule set.

Example 5 (*except*): Let `XP1 = /child::node()/self::a/child::node()/self::b`, let `XP2 = /descendant-or-self::c/ancestor-or-self::b`. Then `XP1 except XP2 ≡ /child::node()/self::a/child::node()/self::b[not(self::b/descendant-or-self::c/ancestor-or-self::node()[self::node () is root()])]` according to Section 3, which can be transformed to the simplified query `/child::a/child::b[not(descendant::c)]` using our proposed rule set.

6. Performance Analysis

We present the experimental environment in Section 6.1. The first data set used in the experiments consists of synthetic data especially designed so that we can show the relationship between the achieved speed-up and the selectivity (see Section 6.2). We use the data set of the XPathMark Benchmark [7] for the second data set in order to show the achieved speed-up factors for typical queries (see Section 6.3).

6.1. Experimental Environment

The test system for all experiments is an Intel Pentium 4 processor 1.7 Gigahertz with 1 Gigabyte RAM, Windows XP as operating system and Java VM build version 1.4.2. We use the XQuery evaluators Saxon [14] version 8.0 and Qizx version 0.4p1 [6] in order to process the XPath expressions.

6.2. First Data Set

The first data set used in the experiments consists of synthetic data. The used XML documents contain root elements <a>, the child nodes of which are <b> elements. The <b> elements have exactly one child node, which is either a <c> element or a <d> element. We vary the size of the XML document by the number of <b> elements and we vary the selectivity of the queries, which is defined to be the division of the size of the result by the size of the input, by varying the number of <c> and <d> elements.

The speed-up factor is defined to be the quotient of the execution time of the original query and the execution time of the simplified query using the same data set. While evaluating the original and the simplified queries of Example 2 and of Example 4, the achieved speed-up factors are between 0.88 and 1.105, i.e. we do not achieve high speed-up factors for the simplified XPath expressions.

We present the execution time of the Qizx evaluator of the original query '`XP1 intersect XP2`' of Example 3 in Figure 3, and the execution time of the simplified query of Example 3 in Figure 4. We present the speed-up factors of the queries of Example 3 in Figure 5 for the Qizx evaluator.

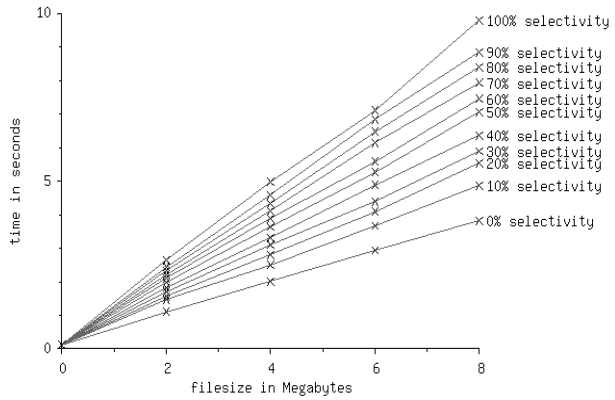

Figure 3: Execution time of the original query **`XP1 intersect XP2`** of Example 3 using the Qizx evaluator

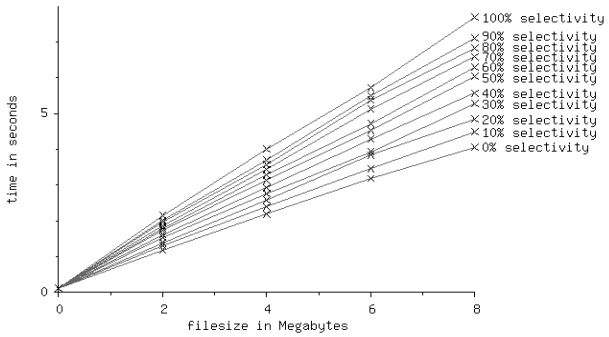

Figure 4: Execution time of the simplified query of Example 3 using the Qizx evaluator

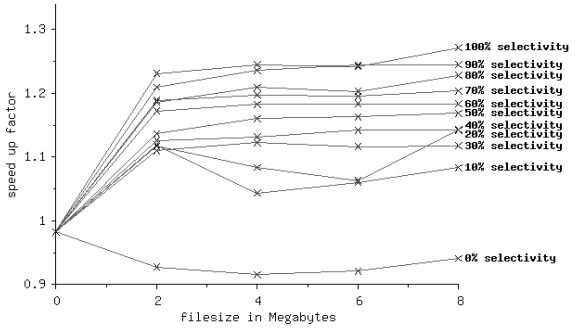

Figure 5: Speed-up factors of the queries of Example 3 using the Qizx evaluator

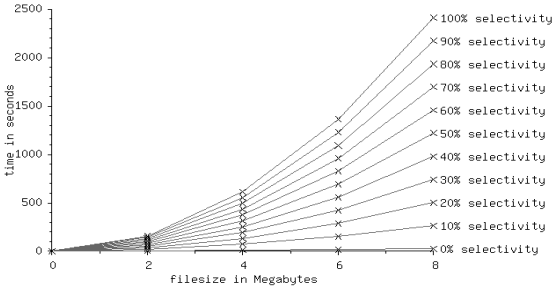

Figure 6: Execution time of the original query **`XP1 intersect XP2`** of Example 3 using the Saxon evaluator

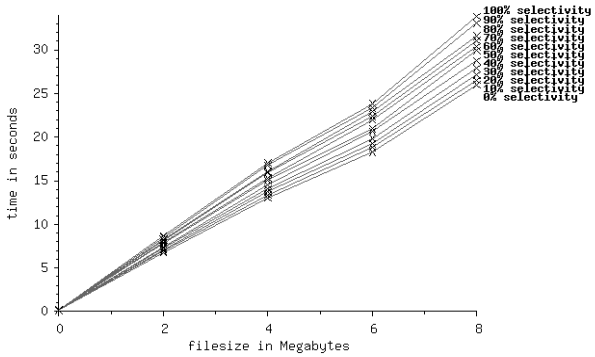

Figure 7: Execution time of the simplified query of Example 3 using the Saxon evaluator

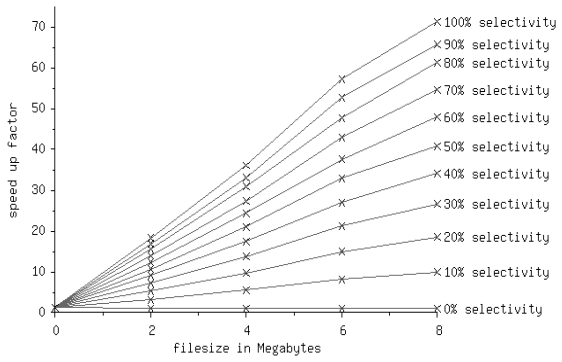

Figure 8: Speed-up factors of the queries of Example 3 using the Saxon evaluator

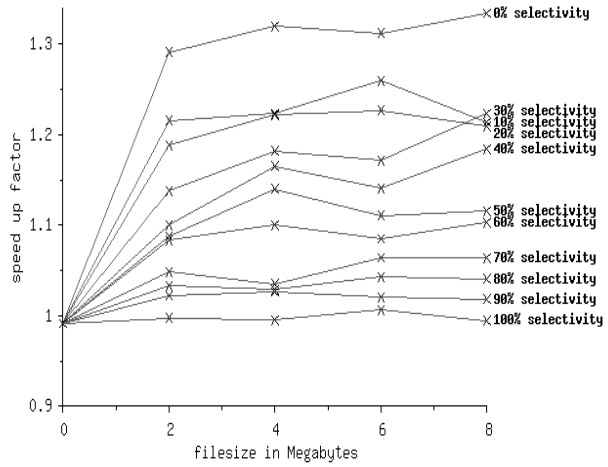

Figure 9: Speed-up factors of the queries of Example 5 using the Qizx evaluator

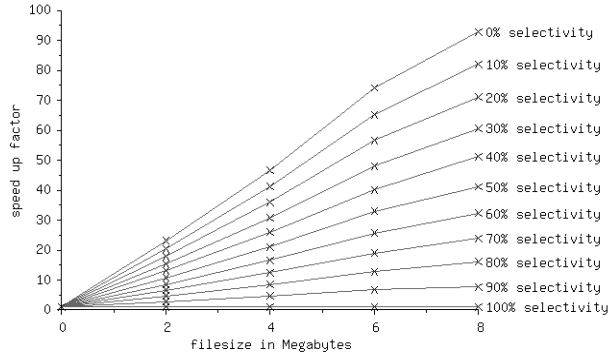

Figure 10: Speed-up factors of the queries of Example 5 using the Saxon evaluator

Furthermore, we present the execution time of the Saxon evaluator of the original query '`XP1` **`intersect`** `XP2`' of Example 3 in Figure 6 and the execution time of the simplified query of Example 3 in Figure 7. To compare the execution times of the original query with the simplified query, we present the speed-up factors of the queries of Example 3 in Figure 8 for the Saxon evaluator.

Furthermore, we show the speed-up factors of the queries of Example 5 for the Qizx evaluator in Figure 9 and for the Saxon evaluator in Figure 10. The Saxon evaluator has exponential runtime for the original queries depending on the size of the input XML document so that the speed-up factors increase with the file size and are up to 93 times. The Qizx evaluator appears to internally optimize more than the Saxon evaluator, but we still achieve speed-up factors of up to 30%.

6.3. Second Data Set, XPathMark Data Set

Combination of XPathMark queries (X_1, X_2)	Name of X_1 **intersect** X_2	Name of X_1 **except** X_2	Combination of XPathMark queries (X_1, X_2)	Name of X_1 **intersect** X_2	Name of X_1 **except** X_2
(Q1, Q5)	I1	E1	(Q10, Q12)	I10	E10
(Q1, Q12)	I2	E2	(Q12, Q36)	I11	E11
(Q1, Q22)	I3	E3	(Q12, Q42)	I12	E12
(Q1, Q36)	I4	E4	(Q22, Q36)	I13	E13
(Q1, Q42)	I5	E5	(Q22, Q42)	I14	E14
(Q5, Q12)	I6	E6	(Q2, Q3)	I15	E15
(Q5, Q22)	I7	E7	(Q2, Q4)	I16	E16
(Q5, Q36)	I8	E8	(Q3, Q4)	I17	E17
(Q5, Q42)	I9	E9			

Figure 11: Names of the original queries (containing an **`intersect`** operator or an **`except`** operator, the operands of which are XPathMark queries) used for the experiments.

In the following experiments, we use the data set and the queries of the XPathMark Benchmark [7] in order to show the achieved speed-ups for typical XPath queries. We have generated data from 0.116 Megabytes to 11.597 Megabytes by using the data generator of the XPathMark Benchmark. Figure 11 presents the used queries in the experiments. Additionally, we use the query `Pi=//keyword(/parent::node()/child::keyword)`^{i} in order to show speed-up factors after eliminating XPath *reverse* axes, where `A`^{i} stands for `i` iterations of an XPath expression `A`. We use `Si=//keyword(/self::keyword)`^{i} in order to demonstrate the achieved speed-up factors after eliminating a location step containing a `self` axis. Altogether, we measured the speed-up factors of 49 queries achieved by using our approach. Figure 12 and Figure 14 present the achieved speed-up factors of the queries `E17`, `E15`, `E16`, `P1` to `P5` and `S10` to `S100` when using the Saxon evaluator. Figure 13 and Figure 15 present the achieved speed-up factors of the queries `E17`, `E15`, `E16`, `P1` to `P5` and `S10` to `S100` when using the Qizx evaluator. Summarizing the experimental results, we achieve high speed-up factors if we can simplify the

XPath query to the empty expression (E7, E15 and E16), if we can eliminate reverse axes (P1 to P5), or if we can eliminate many location steps (S10 to S100). Otherwise, the simplified queries are nearly as slow as the original queries, i.e. their executions vary from 12% slower to 50% faster. The average speed-up factor of all queries is 1.3, i.e. the execution of the simplified queries is 30% faster. The execution of the simplified queries is 900% faster, when using Qizx.

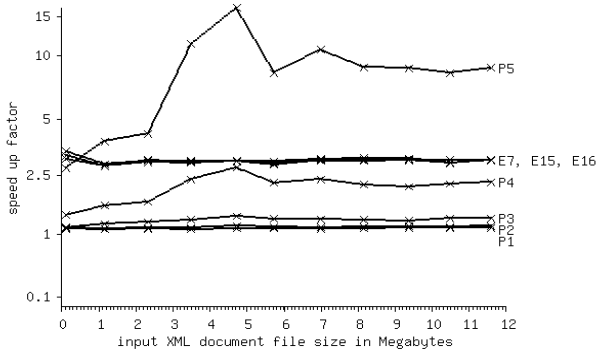

Figure 12: Speed-up factors of queries of P1 to P5, E7, E15 and E16 using Saxon

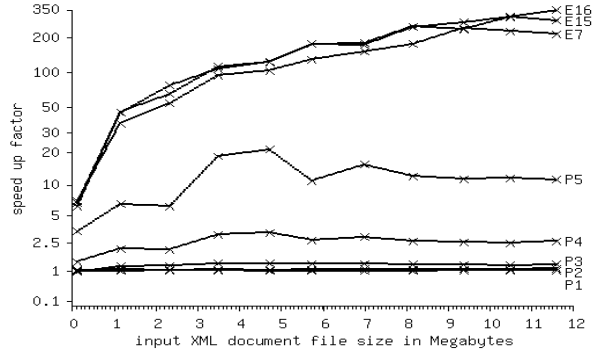

Figure 13: Speed-up factors of queries of P1 to P5, E7, E15 and E16 using Qizx

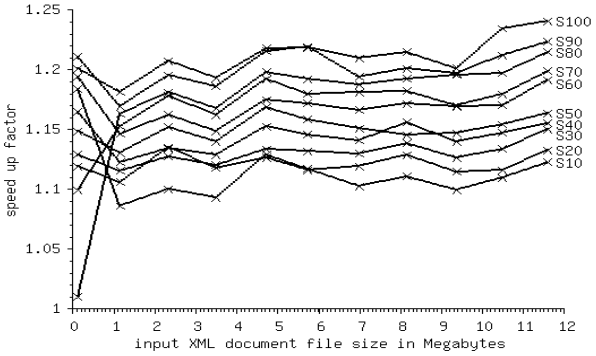

Figure 14: Speed-up factors of queries of S10 to S100 using Saxon

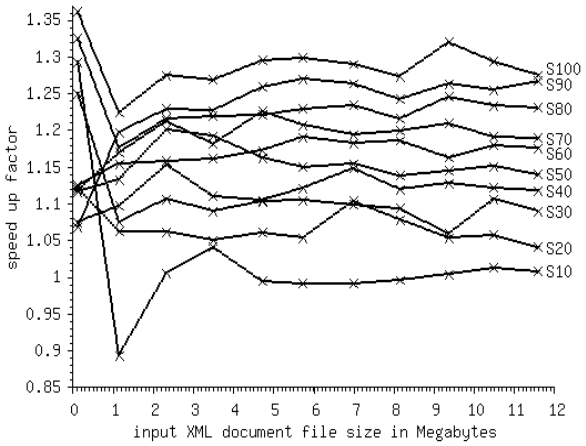

Figure 15: Speed-up factors of queries of S10 to S100 using Qizx

7. Further Related Work

Whereas [8] and [9] describe algorithms in order to evaluate XPath queries, we *logically* simplify the XPath query.

[5] describes how wildcard steps can be eliminated in an XPath query, which we neglect in our approach. [19] shows that, in fact, equivalence and minimality of simple XPath expressions can be decided in polynomial time. In presence of a DTD, the decision whether or not two expressions (without wildcards) are equivalent according to a DTD is coNP-hard [19].

[2] deals with the complexity of XPath satisfiability tests in the presence of DTDs. We can use our contributions to extend the results of [2] to the containment test and the intersection test of XPath expressions. [10] describes the satisfiability test of XPath queries according to the constraints given by an XML Schema definition, while we use a set of rules to check XPath satisfiability. [11] presents how to use the presented approaches of this paper for a static analysis of XSLT stylesheets as part of the optimization of XSLT stylesheets.

[1] deals with the minimization of tree pattern queries both in the absence and in the presence of integrity constraints. Tree patterns queries are tree patterns, where nodes are types and edges are child or descendant relationships, which do not consider order. The goals of our rule set are to eliminate redundant constructs in XPath (which do not occur in tree pattern queries), to support a bigger subset of XPath than tree pattern queries and the optimizations of these queries.

[12] and [13] present approaches for optimizing XQuery [12] or XSLT [13] queries, but these

approaches do not modify the XPath expressions embedded in the XQuery queries or XSLT stylesheets.

In comparison to all other contributions, we present a method, which eliminates the **`intersect`** and **`except`** XPath 2.0 operators. Furthermore, we propose a rule set, which simplifies a given XPath query by a heuristic method.

8. Summary and Conclusions

We first introduce reverse patterns of XPath expressions, which can be used in order to check whether an XML node matches an XPath pattern. Then, we use these reverse patterns in order to eliminate the **`intersect`** and **`except`** XPath 2.0 operators. Afterwards, we apply a rule set to the resultant XPath expression in order to optimize its evaluation time.

The application of the proposed rule set is not restricted to those XPath queries, which are the result of the **`intersect`** and **`except`** elimination, and can be applied in general.

Summarizing the experimental results, we achieve high speed-up factors (up to the factor 350) if we can simplify the XPath query to the empty expression, if we can eliminate reverse axes, or if we can eliminate many location steps. Otherwise, the simplified queries are in most cases a little bit faster and in few cases a little bit slower.

Because XPath expressions play a key role in XQuery expressions, it appears to be promising to investigate how our simplification approach can be extended to the XQuery language.

10. References

[1] Amer-Yahia, S., Cho, S., Lakshmanan, L. V. S., and Srivastava, D., Minimization of Tree Pattern Queries, *ACM SIGMOD 2001*, Santa Barbara, California, USA, 2001.

[2] Benedikt, M., Fan, W., and Geerts, F., XPath Satisfiability in the Presence of DTDs, In Proceedings of *PODS 2005*, Baltimore, Maryland, USA, 2005.

[3] Benedikt, M., Fan, W., and Kuper, G. M., Structural Properties of XPath Fragments, *ICDT 2003*, Siena, Italy, 2003.

[4] Böttcher, S., and Steinmetz, R., Adaptive XML Access Control Based on Query Nesting, Modification and Simplification. *BTW 2005*, Karlsruhe, Germany, 2005.

[5] Chan, C.-Y., Fan, W., and Zeng, Y., Taming XPath Queries by Minimizing Wildcard Steps, In *VLDB*, Toronto, Canada, 2004.

[6] Franc, X., Qizx/open version 0.4p1, http://www.xfra.net/qizxopen/, 2004.

[7] Franceschet, M., XPathMark - An XPath benchmark for XMark. *Research report PP-2005-04*, University of Amsterdam, The Netherlands, 2005.

[8] Gottlob, G., Koch, C., and Pichler, R., Efficient Algorithms for Processing XPath Queries, In *VLDB 2002*, Hong Kong, China, 2002.

[9] Gottlob, G., Koch, C., and Pichler, R., The Complexity of XPath Query Evaluation, In *PODS*, San Diego, California, USA, 2003.

[10] Groppe J., Groppe S. Filtering Unsatisfiabile XPath Queries, *ICEIS 2006*, Paphos-Cyprus, May 2006.

[11] Groppe, S., XML Query Reformulation for XPath, XSLT and XQuery, *Sierke-Verlag*, Göttingen, Germany, 2005, ISBN 3-933893-24-0.

[12] Groppe, S., and Böttcher, S., Schema-based query optimization for XQuery queries, *ADBIS 2005*, Talinn, Estonia, 2005.

[13] Groppe, S., Böttcher, S., Birkenheuer, G., and Höing, A., Reformulating XPath Queries and XSLT Queries on XSLT Views, *Journal Data & Knowledge Engineering*, to appear soon, http://www.sciencedirect.com/science/journal/0169023X.

[14] Kay, M. H., Saxon - The XSLT and XQuery Processor, http://saxon.sourceforge.net, April 2004.

[15] Marx, M., First Order Paths in Ordered Trees, I Proceedings of the *10th International Conference of Database Theory (ICDT 2005)*, Edinburgh, UK, 2005.

[16] Moerkotte, G., Incorporating XSL Processing Into Database Engines. In *VLDB*, Hong Kong, China, 2002.

[17] Olteanu, D., Meuss, H., Furche, T., Bry, F., XPath: Looking Forward, *XMLDM*, Prague, Czech Republic, 2002.

[18] Tajima, K., and Fukui, Y., Answering XPath Queries over Networks by Sending Minimal Views, In *VLDB*, Toronto, Canada, 2004.

[19] Wood, P. T., Minimising Simple XPath Expressions, In *WebDB*, Santa Barbara, California, 2001.

[20] World Wide Web Consortium (W3C), XML Path Language (XPath) Version 1.0, *W3C Recommendation*, http://www.w3.org/TR/xpath/, 1999.

[21] World Wide Web Consortium (W3C), XML Path Language (XPath) Version 2.0, *W3C Working Draft*, http://www.w3.org/TR/xpath20/, 2003.