Lazy Evaluation of Semantic Mapping Rules

Master Thesis

Databases and Information Systems Group
August 20, 2007 - February 20, 2008

Jonas Rutishauser
ETH Zürich
jonasr@ethz.ch

Supervised by:
Prof. Dr. Donald Kossmann
Tim Kraska
Abstract

The concept of Is-a Rules described in the master theses “Mapping Data to Queries” allows the integration of XML data enabling an object oriented usage of XML data items. In this paper it is described how to implement this concept in an efficient way. The rules are now longer evaluated eagerly at the beginning but lazy while processing the XQuery.
# Contents

1 Introduction ........................................... 7

2 Motivation ............................................. 9
   2.1 Data integration ................................ 9

3 The Is-A Rule .......................................... 11
   3.1 Syntax ........................................... 12
   3.2 Data Representation ............................. 12

4 Lazy Evaluation ....................................... 15
   4.1 Introduction .................................... 15
   4.2 Limitations .................................... 15
   4.3 Implementation ................................ 16
      4.3.1 Special iteration cases ..................... 17
      4.3.2 Example ................................... 17
   4.4 Remarks ........................................ 18

5 YFilter Approach ..................................... 21
   5.1 Introduction .................................... 21
   5.2 Limitations .................................... 21
   5.3 Implementation ................................ 21
      5.3.1 Construction of the NFA .................. 21
      5.3.2 Execution of the NFA ..................... 23
      5.3.3 Handling a matched rule ................. 25
      5.3.4 Handling predicates ....................... 25
      5.3.5 Special iteration cases .................. 25
   5.4 Remarks ........................................ 27

6 Correctness ............................................ 29
   6.1 Queries with forward axis ...................... 29
      6.1.1 Path-Dependency ........................... 29
      6.1.2 Predicate-Dependency ..................... 30
   6.2 Queries with parent axis ....................... 30
7 Benchmarks

7.1 Baseline ............................................................................................................ 33
7.2 Simple rewrites .................................................................................................. 36
7.3 Useless rules ....................................................................................................... 39

8 Conclusion

8.1 Query Rewrite vs. Is-A Rules ............................................................................ 41
8.2 Further possible improvements ......................................................................... 42
Chapter 1

Introduction

This thesis is based upon the master thesis “Mapping Data to Queries”\(^1\). That thesis presents a concept of how to integrate XML data in an easy and less error prone way. The concept describes how to write Is-A Rules which introduce a subset relationship. With this subset relationship one can also express an equals relationship. To do so both elements have to be a subtype if the other. A short introduction to the Is-A Rules is given in chapter 3.

This thesis contains concepts of how to implement this rules in a more efficient manner. That means it should be possible to have thousands of rules without much impact on processing time. The most important change in the new concept is that the rules are not applied in the beginning but while the document is processed by the XQuery engine.

Restrictions. To apply the rules while walking through the document it is no longer possible the allow every possible rule. This implementation supports only a subset of rules but in practice there or not many cases when other rules are needed. The exact definition of the new rule syntax is described later in this thesis.

\(^1\)See [1]
Chapter 2

Motivation

This section presents a usecase for which this implementations are design. There are also other usecases where this design is favorable too.

2.1 Data integration

This is the use case which gave a big motivation to have this implementations. It is also a use case which is often present in practice. XML documents from different sources normally have different schemas and therefore it is very difficult to query all this documents at once.

**Rewrite approach.** One approach is to have a query for every schema. Thus a query is duplicated and then transformed to match the different schemas.

**Transformation approach.** An other approach is to transform every document to the local schema and only one query is needed though. In the transformation all data needed for the query will be matched to the local schema. Some data that might be important in the future will then not be transformed and therefore not available.

**View approach.** A third approach is to use views like in the relational data world. The *Is-A Rules* approach is one of these. In this approach only one query is needed and the data is not lost as the documents still have their own schema but are viewed as if they all have the same schema.

**Disadvantages.** The first approach has the disadvantage that the duplication of code is very error prone and very difficult to maintain. The disadvantages of the second approach, the loss of data, has already been mentioned. But there is also the disadvantage that the transformation creates new elements and thus the new element and the old one have not the same identity.

The third approach has the difficulty that the implementation is very complex and may be slow. That’s where this theses comes in. The following chapters will describe a way of how to implement the rule framework in more efficient manner than the simple implementation mentioned in the master thesis “Mapping Data to Queries”.

Advantages. The advantages of the Is-A Rules are the following.

- Duplication of query code to match the different schemas is avoided
- No data is lost as the original data is still available
- No new data is created. The element in the old schema and in the new schema share the same identity.
Chapter 3

The Is-A Rule

Simplified, the Is-A Rule has the following form:

\[ \text{Source is-a Target} \]

With the left hand side a set of sources can be selected which is than matched to the right hand side. The target is either a simple path or an XQuery element constructor.

**Subset relationship**  An Is-A Rule always expresses a subset relationship. This means that whenever we search for a target of a rule all matching sources will also be returned. For example the rule \textit{banana is-a fruit} means that every banana is a fruit too. So if we ask for every fruit by the path expression \texttt{//fruit} every node described by the path expression \texttt{//banana} will also be returned. Therefore \textit{banana is-a fruit} states the following expression \texttt{//banana} \(\subseteq\) \texttt{//fruit}.

**Equal relationship**  It is simple to express an equal relationship using the Is-A Rules. The method to define that A is equal to B is to define the following two rules.

\[
\begin{align*}
A & \text{ is-a } B \\
B & \text{ is-a } A
\end{align*}
\]

**Subtype polymorphism**  As the Is-A Rule expresses a subset relationship they also must ensure that the source nodes conform to the type of the target. This means in the example from above that a banana must also be usable whenever a fruit is used.

This is like in the software engineering where a subtype must conform to its supertype. A subtype can always be used when its supertype is expected.

The programmer of an Is-A Rule has to ensure that these properties always held. If these properties are not ensured a query could have an unexpected result.
3.1 Syntax

Here is a copy of the syntax as described in the master theses “Mapping Data to Queries”

Definition 3.1.

\[
\begin{align*}
\text{ISARule} &::= \text{Source} \ 'is-a' \ \text{Target} \\
\text{Source} &::= \text{SourceNode} \mid \text{SourceNode} \ 'in' \ \text{PathExpr}^1 \\
\text{SourceNode} &::= \text{PathExpr} \mid \text{PathExpr} \ 'as' \ \text{VarRef}^2 \\
\text{Target} &::= \text{TargetNode} \mid \text{TargetNode} \ 'in' \ \text{PathExpr} \\
\text{TargetNode} &::= \text{SimplePath} \mid \text{DirElemConstructor}^3 \\
\text{SimplePath} &::= \text{QName}^4 \mid '/' \ \text{QName} \mid \text{SimplePath} '/' \ \text{QName}
\end{align*}
\]

3.2 Data Representation

To implement the Is-A Rules a different representation of XML data is needed. The mostly used representation of XML data is a tree where XML nodes are mapped to graph nodes. The is-child property is represented as an edge. Figure 3.1 shows an example of an XML tree.

```
<a>
  <b foo="bar"> 
    <c>blub</c>
  </b>
  <d/>
</a>
```

The new representation is described in details in the master thesis “Mapping Data to Queries”. Shortly it can be described as moving the node names from the nodes to the incoming edges. With the Is-A Rules this no longer results in a tree. There are different types of edges such that at every time it’s known which edges are the original ones.

The figure 3.2 shows the same document as in figure 3.1 in the new representation and also after applying the rule \( c \ is-a \ z \).
Figure 3.2: New representation with a rule applied
Chapter 4
Lazy Evaluation

4.1 Introduction

A first very simple attempt to improve the rule framework was to not apply all rules before querying the document. The rules are applied while the XQuery processor walks through the XML document. The framework is no longer eager but lazy. It only applies rules if it reaches a point where it may make sense. Not the whole document must be transformed if the query terminates already in the middle of the XML document.

Every rule defines a state machine where each step in the rule path defines a state. While the XQuery processor walks through the XML document the new data model notifies each rule which edge the XQuery processor will visit next. Each rule than checks if it has to do something or not.

4.2 Limitations

As each step is turned into a state and the XQuery processor will always walk from the top of the XML tree to the bottom there are some limitations for the rules to allow an implementation that’s not to complicate. The lazy evaluation framework will only accept rules which consist of forward axis and have not a in part on the target side.

The new syntax is therefore the following

Definition 4.1. (Only modifications are listed. The modifications are in bold)

Source ::= SourceNode |
| SourceNode ‘in’ ForwardPathExpr
SourceNode ::= ForwardPathExpr |
| ForwardPathExpr ‘as’ VarRef
Target ::= TargetNode
ForwardPathExpr ::= ‘/’ ForwardRelativePathExpr? |
| ‘//’ ForwardRelativePathExpr
ForwardRelativePathExpr ::= ForwardAxisStep (( ‘/’ | ‘//’ ) ForwardAxisStep)*
ForwardAxisStep ::= (’child::’|’descendant::’)? NodeTest1PredicateList2

1See http://www.w3.org/TR/xquery/#prod-xquery-NodeTest
2See http://www.w3.org/TR/xquery/#prod-xquery-PredicateList
4.3 Implementation

As already mentioned in section 4.1 each rule generates a state machine. To generate such a state machine the following steps have to be performed.

**Definition 4.2.** The state machine generation process:
1. generate the start state
2. set the current state to the start state
3. process the in part of the source in the rule as described in definition 4.3 (the current state should now be the last state of the in part)
4. set the current state as the “context state”
5. process the source part of the rule as described in definition 4.3 (the current state should now be the last state)
6. set the current state as the “target state”

**Definition 4.3.** The part processing process:
while there is a next XQuery step
1. generate a new state
2. make a transition from the current state the the new state
3. store the axis, the node test name and the filter of the next step in the transition
4. set the new state as the current state
5. goto 1.

As an example the figure 4.1 shows the state machine for the rule *banana in data//objects[@name="fruits"] is-a fruit*. In the figure the context state is in blue and the target state in green.

![Figure 4.1: State machine](image)

As described in the section 4.1 the rule framework has to be changed such that each node which will be seen next by the XQuery engine is published to each rule. The figure 4.2 will show in abstract how this looks like.

Each rule then looks every time if the next node matches the next state transition. If a rule reaches the context state it saves the current XML node as the context node. If the rule reaches the target state it adds the target path as described in the rule target from the previously saved context node to the current XML node. To check if the predicates are fulfilled one can simple launch the XQuery engine starting from the current node. This may make it a bit slow but simplifies a lot the implementation.
4.3.1 Special iteration cases

There are two cases in the node iteration which must be handled carefully. The first one is when the query ask for a child A which will only be available by a rule whose source path contains more than only a single child step. An example is the query /a and the rule foo/bar is-a a when the root element node of the document is foo. In this case the XQuery engine will only use a single child iterator and make a nodetest. The framework will then only visit the element foo. As after visiting the element foo the rule foo/bar is-a a has not yet matched no new edge is added. Therefore each rule which reaches a state after the context state has to visit all child elements if the query engine uses a child iterator.

The second case is when the XQuery engine uses a descendant iterator. In the eager implementation the descendant iterator returns each edge only once due to the possibility of cycles. With the lazy implementation there may be some matches missing if we stop going deeper if an edge is already visited. To avoid this the descendant iterator simple has to notify the rules over every descendant node. There are no problems with cycles as they can no longer exist with the limited rule syntax.

4.3.2 Example

To show the rule processing an example XML document is described in figure 4.3. The figure 4.4 shows the rule processing for the example XML document and the query //car[1]. The current state and next edge to visit will be colored in magenta.
4.4 Remarks

**Advantages** The advantages of this implementation over the eager implementation is that if the XQuery engine terminates before it sees the whole XML document the rule framework will only insert new edges in the processed part. This implementation doesn’t change anything in the document part that is not interesting for the query. This may save much processing time.

**Disadvantages** As every rule has its own state machine and every next edge to visit is always signaled to all rules the node processing time increases linear to the number of rules. Because of this that approach isn’t good if there are a lot of rules but still better then the eager implementation.
Figure 4.4: Example processing of a lazy rule
Chapter 5

YFilter Approach

5.1 Introduction

This implementation is inspired from the lazy implementation of chapter 4. As the biggest disadvantage there is that every rule has its own state machine the next improvement is to merge this state machines to one state machine.

To do that there is the paper over YFilter\(^1\) which gave a big inspiration and the name for this approach. The basics are the same as in the YFilter paper but there are still some differences to the YFilters which we will see in the section 5.3 that describes the implementation.

5.2 Limitations

As in this approach only the state machines are merged together the limitations are still the same as in the previous lazy implementation. To see the limited rule syntax see section 4.2. This limitations also exist in the same manner in the YFilters.

5.3 Implementation

In this implementation the different state machines are merged into a Nondeterministic Finite Automaton (NFA)\(^2\). The machine is stack based to handle nondeterminism and support backtracking. Figure 5.1(a) will show some rules and figure 5.1(b) the NFA for this rules. For simplification the example contains no predicates and the accepting states contain the number of the respective rule. The symbol “∗” will match any element and the symbol “ǫ” specifies a transition that requires no input (so called epsilon transition).

The overall framework will be the same as in the previous lazy implementation shown in figure 4.2.

5.3.1 Construction of the NFA

At the beginning the NFA consists of only one state which is the start state. Each rule is than added one after the other as described in definition 5.1. Remark that there exists

\(^1\)See [3]
\(^2\)See http://en.wikipedia.org/wiki/Nondeterministic_finite_state_machine
two types of states: normal states and epsilon states. An epsilon state is like a normal state except that always has a “∗” transition to itself.

To make it simple predicates aren’t allowed here. Section 5.3.4 will later describe how to handle the predicates.

Definition 5.1. Adding a rule to the NFA

1. Set the start state of the NFA as the current state
2. If there is no in part in the source set it to “/.”
3. If the in part doesn’t begin with “/” prepend “/”
4. Add the in part as described in definition 5.2
5. Save the current state as the context state of this specific rule
6. Add the main part of the source as described in definition 5.2
7. Set the current state as the accepting state for the specific rule

Definition 5.2. Adding a rule part to the NFA
While there is a next path step

1. If the axis is an descendant axis:
   a) Add an epsilon transition to the current state if it doesn’t exists
   b) Set the current state to the epsilon state
2. Add a transition to the current state with the name test of the path step if it doesn’t exists
3. Set the current state to the target of the transition received before

Figure 5.2 will show an example of adding the rule $a//c$ is-a $x$ to the NFA containing already the rule $a//b$ is-a $x$. The currently active part is in red for better illustration.
5.3.2 Execution of the NFA

For the execution of the NFA there is a stack of states which is initialized to contain an item containing only the start state when the XML document root is passed to the rule framework. Each time when the next XML node is signaled to the rule framework it first checks if the parent of the XML node is the node which describes the top of the stack. If not every item on the stack is removed until the previous condition is fulfilled.

After that for every state in the top stack item several operations are performed. First it is checked if a transition exists which match the current XML node name. If so the resulting state is added to the new stack item. The same procedure is done for the “*” transition. Remark that an epsilon state always contains a “*” transition to itself.

After that it is checked if there is an epsilon transition and if so the same operations are performed for that epsilon state. The new stack item is then pushed with the XML node on the top of the stack. In order that we can later determine the context node every state on the stack is linked to the state where it comes from.

Now for every state in the new stack item it is checked if it is an accepting state. If this is the case the respective rules are handled.

Figure 5.3(d) will show an example of querying a simple XML document which is described in (a) with it’s representation (b). The example uses the rules and the respective NFA from figure 5.2 with it’s states numbered from 0 to 5 (see figure 5.3(c)). The XQuery is “//*” and it only shows until the first rule matches. The edge signaled to the rule framework is drawn in red to highlight it.
Figure 5.3: Example execution of the NFA
5.3.3 Handling a matched rule

If a rule matches the target element node is known and the respective context node must be determined. To get the context node the framework must follow up the links on the stack until it finds the context state. As every entry on the stack also saves the respective node the context node can easily be retrieved. Having the context element node and the target node the same procedure as in the eager implementation can be done to create the new access-path.

As an example we have the matched rule from the previous example (see figure 5.3). The respective context state has the number 0. We go up the stack from the state 4 and get the state 3 which is not the context state. We continue going up until we reach the context state in the first stack entry. This entry corresponds to the document node. We have now the document node as the context and the current element node as the target. The next step is to insert a new edge with name y form the document node to the current node.

5.3.4 Handling predicates

To be able to handle predicates there is not much which must be changed. The predicates are only checked after the rule matched as if it has no predicates.

When the NFA is constructed from a rule at each path step that contains a predicate the rule must store the current state and the predicate. When then the rule has a match we must go the stack up to get the context state. In the same time whenever we reach a state that has a predicate assigned we check this predicate using the XQuery engine. If the predicate doesn’t return true the engine can stop the processing of this match and therefore no new edge will be inserted. It is important that the going up in the stack must be done for the whole stack and can’t be finished after the context state as the context path may also contain predicates.

5.3.5 Special iteration cases

As already mentioned in the lazy implementation there are the two cases which must be handled carefully. The first case which occurs when the XQuery uses a child iterator is simple to handle. The iterator has simply not to publish only the next edge it visits but also that it won’t go any deeper. The rule framework than knows if on the stack are more than only the start state and one of these states has a child state that it has to inspect the child edges of the current node too. To do that it can simply use a child iterator that the XQuery engine would also use. Using the same iterator the framework won’t go deeper as needed because if there are no more child states the framework stops using a child iterator.
Example. As an example the query would be /a and the rule is foo/bar is-a a. The
document 5.4 shows the rule framework in two different cases.

In the first case (b) one can see that the stack contains not only the start states \{0,1\} but also the state 2 which has a child state (3). Therefore the rule framework will create a new child iterator of the node “foo” and use it. This iterator then notifies the rule framework of the edge “bar” and a new edge “a” will be inserted. No new child iterator will then be created as the state 3 doesn’t have a child state.

In the case (c) there is only the start state 1 and therefore further going down is not needed. There is no chance that we can insert any new edge to the document node.

The second case with the descendant iterator can be handled in a simple way by simply going always deeper even if the edges have already been visited. This makes the implementation simple but would lead to bad performance. The better way is to handle it in a similar way as with the child iterator. The descendant iterator simply has to tell the rule framework when the next edge will lead to an already visited node. The rule framework will then decide in the same way as in the child iteration case if a child iterator need to be created. The descendant iterator must therefore visit each edge only once and the rule framework will decide when an already visited edge has to be visited again.

Figure 5.4: Examples of child iteration cases (query: /a)
5.4 Remarks

At every time when the XQuery engine visits a new edge there is nearly a constant overhead (independent of the number of rules) if no rule matches. Only if there are a lot of complicate predicates the overhead might be high. If a rule matches the overhead is one that can’t be avoided because the new path must be inserted.

The system could be improved by adding the predicates of simple target rules which consist only of forward axis to the NFA and insert the new path only when all predicates evaluate to true. An other improvement would be that by inspecting the query rules that will never add something to the result are not added to the NFA.
Chapter 6
Correctness

In this chapter it is showed that the Lazy and the YFilter approach give correct results. In a first step only queries which contain forward axis are considered. After that queries with parent axis are examined too.

6.1 Queries with forward axis

It is easy to see that these approaches doesn’t miss anything if there is only one rule. Therefore the next sections will only show that there’s also nothing missed of there are more than one rule.

It is also easy to see, that there is nothing missed if the rules are independent of each other. The proof will be divided in the cases when the dependency is in the path and when the dependency is in a predicate.

To handle also the cases when a rule depends on it self the same rule can be seen as being more than once in the pool. This wouldn’t change the processing as only the first instance will do something and the others always will remark the target path as already there. The result must still be the same but the proofs have not to handle this case separately.

6.1.1 Path-Dependency

In this section we assume that the rules don’t have predicates. A rule depends on an other rule if the source path expression gives a new result after applying the other rule. An example is the rule $a \text{ is-a } x$ which may have a dependency to the rule $b \text{ is-a } a$. Figure 6.1 will show this dependency for an example XML document.

```
<doc>
  <b/>
</doc>
```

Figure 6.1: Example of a dependency
To have a miss there must be a rule which inserts a new edge needed by an other rule. This edge must not be visited later as otherwise there’s no miss.

As every new edge is added to the end of the child list we can be sure that this edge will be visited later in the processing and therefore no such miss can occur. As this is a simple induction step this must also hold for rules depending on more than one rule.

### 6.1.2 Predicate-Dependency

A rule is dependent of an other rule by a predicate if the other rule inserts a new path which will make a predicate evaluate to true which was false before. A simple example is the rule `a[x] is-a y` which may depend on the rule `b is-a x`. This can be seen in figure 6.2

![Example of a dependency](image)

As the check of the predicates uses the XQuery engine there can be no miss as otherwise the XQuery engine wouldn’t give always a correct result.

### 6.2 Queries with parent axis

If a query uses somewhere the parent axis it wouldn’t give always the same result as with the eager implementation. This may be the case when the query gets a node by walking down the XML tree and then asks for the parent node while there are some rules which would insert multiple parents to this node. In the eager implementation the result then contains multiple nodes. In the lazy implementation there may be some of them missing. An example for that is the query `/a/b/c/parent::*` on the following XML document.

```xml
<a>
  <b>
    <c/>
  </b>
</a>
```

with the rules:

- `b/c is-a d/e`
- `d/e is-a f/g`
The result using the eager implementation is the following:

```xml
<b>
  <c/>
</b>
<d>
  <c/>
</d>
<f>
  <c/>
</f>
```

In the new lazy implementation it may be that the element “f” does not occur.

If somebody asks a query like this he normally expects only one result containing the “b” node because a normal XML document is a tree where every node has at most one parent node. Therefore it’s not a big problem if the lazy implementation doesn’t return all of the additional nodes. The query even should better be written as the following: `/a/b/c/parent::b`. With that query one will always get only the node “b” as expected.

If somebody really wants all the nodes returned as in the eager implementation there should be a warning that the result probably may not be complete when the parent iterator is used.
Chapter 7

Benchmarks

In this chapter some simple benchmarks will compare the Is-A RULES approach using the YFilter implementation and the query rewrite approach. As there is no clear documentation how the query rewrite will work all rewrites are done by hand. Usually there is more than one way to rewrite a query.

7.1 Baseline

Figure 7.1: Baseline benchmark
The first benchmark will compare the different data models. The results are in figure 7.1, figure 7.2 and figure 7.3 and show the time needed for loading and querying different XML documents using the new data model in this paper, the data model Saxon uses normally and the JDOM\textsuperscript{1} data model.

![Graph showing load times](image)

**Figure 7.2: Baseline load times**

The results of figure 7.2 where the load times are shown describe how fast the data model is constructed. The implementation of the new data model is here not as good as the others but the testing implementation may not be the best implementation of the new model. All three models loading time increases linear to the input file size as expected.

If the query times shown in figure 7.3 are considered one can say that the new data model is not slower then the data model JDOM uses. Therefore the new data model used by the rule framework would not make the whole system much slower then with any other data model.

\textsuperscript{1}See http://www.jdom.org
Figure 7.3: Baseline query times
7.2 Simple rewrites

This benchmark uses rules that are only simple rewrites. This benchmark uses multiple documents following simple schemas that describe documents of the form:

```xml
<a01>
  <a02/>
  <a03>
    ...
  </a03>
</a01>
```

Whereby each schema differs in the names of the elements. The renaming is simple: a01 is renamed to b01, a02 is renamed to b02. For the next schema the elements are renamed from b0* to c0*. The rules for this schemas are the following:

- `a01 is-a b01`
- `b01 is-a a01`
- `a02 is-a b02`
- `b02 is-a a02`
- `...`
- `b01 is-a c01`
- `c01 is-a b01`
- `...`

In figure 7.4 are the benchmark results of the query `//a01`. We assume that the unneeded rules are not added to the NFA therefore only rules of the form `*01 is-a *01` where `*` is a single character are given to the system. There are two rewritten forms of the query:

**Rewrite:** `//a01 | //b01 | //c01 | ...

**Rewrite-Optimized:** `//(a01 | b01 | c01 | ...)`

The results show that with every method the time needed grows linear with the number of input files (which is equal to the number of schemas) as expected. The only difference is that the factor is bigger by the rules approach. This is only a small problem which probably can be solved by doing the implementation a little bit better.
Figure 7.4: Benchmark of simple rename //a01
In figure 7.5 are the next benchmark results for the query //a01/a02. Here we pass only rules of the form *01 is-a *01 and *02 is-a *02 to the system. The two rewritten forms are:

**Rewrite:** //a01/a02 | //b01/a02 | //c01/a02... | //a01/b02 | //b01/b02...

**Rewrite-Optimized:** //a01 | b01 | c01 | ...)/(a02 | b02 | c02 | ...)

This results show that if query rewrite doesn’t use a optimized version it has no chance against the rules approach because it has to go through the XML document more than only once. In this example the simple rewrite query lets the number of times going through the document being quadratic to the number of rules. The optimized version is here really much better.

If only the results of the optimized version are compared to the rules approach there is no big difference to the results of the first benchmark results shown in figure 7.4. Both approaches doesn’t need much more time even that there are twice as much rules.
7.3 Useless rules

In this section the two approaches are compared if there are only rules which may affect the result but never match. This is the case for example if the XML document only contains “fruit” nodes, there are rules like \textit{banana is-a fruit} and the query is \\	exttt{//fruit}.

Figure 7.6 shows the results for simple rename rules. The rewrite uses not the optimized version from the two versions used in section 7.2.

![Graph showing benchmark with useless rules](image)

Figure 7.6: Benchmark with useless rules

The results show that the useless rules don’t influence the time needed by the rules approach. This is as expected because if the rules are not applied the overhead for the query processing will only be a constant as already mentioned in the section 5.4.

By contrast the processing time of the rewrite approach grows linear to the number of useless rules even if the optimized version is used. The only difference between the simple rewrite and the optimized rewrite is the factor of how fast the processing time would increase. With both version the rewrite approach will sooner or later need more time than the rules approach even if the YFilter implementation has a big constant.
Chapter 8

Conclusion

This chapter is divided into two parts. One will show the differences between query rewrite and the Rules approach. This section is based on the results of the benchmark. In the other section is showed what other improvements are possible for the rule framework.

8.1 Query Rewrite vs. Is-A Rules

In this section the two different approaches are compared. Until now there is no example known using Is-A RULES which can’t be handled by query rewrite. If the rewrite generates the best possible rewritten query there are also no rules and a query known until now which is faster queried by the rules framework and is realistic. One case where query rewrite is slower is the case where many useless rules exists but in practice this will rarely be the case.

A problem by this comparison is that the assumptions are that query rewrite always can produce the best query and that the rewrite process takes no or nearly no time. But this is not the case in reality. For example the rules:

- \( a \text{ in } b \text{ as } a \text{ is-a } c <d \text{ id}="\{$a/@name\}"/>c >

- \( a \text{ in } x \text{ as } a \text{ is-a } c <d \text{ id}="\{$a/@nm\}"/>c >

With the query //d may already be complicate to be handled by query rewrite. An optimized version would be:

```xml
for $x in //d | a
return
  if (local-name($x) = 'd') then $x
  else if (local-name($x/..) = 'b') then <d id="\{$x/@name\}"/>
  else if (local-name($x/..) = 'x') then <d id="\{$x/@nm\}"/>
  else ()
```

8.2 Further possible improvements

The benchmarks show that the implementation causes only a constant overhead in iteration over a XML document. This may sound good but the problem is that the constant is still to big because the constant becomes a linear factor to the whole query time. Thus this factor should be nearly one such that the system runs quick even for big XML documents.

A problem is also that until now the rule framework applies every rule it can apply even if this rule will not affect the result in any way. This should be avoided and it should not be very complicate to avoid this. A possible implementation would be to generate a dependency graph of the rules and remove rules which don’t affect the query and are not a dependency of any other rule.
Bibliography

